Refinement Types for Secure Implementations

Jesper Bengtson
Uppsala University

Karthikeyan Bhargavan
Microsoft Research

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Sergio Maffeis
Imperial College London and University of California at Santa Cruz

Abstract

We present the design and implementation of a typechecker for verifying security properties of the source code of cryptographic protocols and access control mechanisms. The underlying type theory is a λ-calculus equipped with refinement types for expressing pre- and post-conditions within first-order logic. We derive formal cryptographic primitives and represent active adversaries within the type theory. Well-typed programs enjoy assertion-based security properties, with respect to a realistic threat model including key compromise. The implementation amounts to an enhanced typechecker for the general purpose functional language F#; typechecking generates verification conditions that are passed to an SMT solver. We describe a series of checked examples. This is the first tool to verify authentication properties of cryptographic protocols by typechecking their source code.

1 Introduction

The goal of this work is to verify the security of implementation code by typing. Here we are concerned particularly with authentication and authorization properties.

We develop an extended typechecker for code written in F# (a variant of ML) [Syme et al., 2007] and annotated with refinement types that embed logical formulas. We use these dependent types to specify access-control and cryptographic properties, as well as desired security goals. Typechecking then ensures that the code is secure.

We evaluate our approach on code implementing authorization decisions and on reference implementations of security protocols. Our typechecker verifies security properties for a realistic threat model that includes a symbolic attacker, in the style of Dolev and Yao [1983], who is able to create arbitrarily many principals, create arbitrarily many instances of each protocol roles, send and receive network traffic, and compromise arbitrarily many principals.

Verifying Cryptographic Implementations In earlier work, Bhargavan et al. [2007] advocate the cryptographic verification of *reference implementations* of protocols, rather than their handwritten models, in order to minimize the gap between executable and verified code. They automatically extract models from F# code and, after applying various program transformations, pass them to ProVerif, a cryptographic analyzer [Blanchet, 2001, Abadi and Blanchet, 2005]. Their approach yields verified security for very detailed models, but also demands considerable care in programming, in order to control the complexity of global cryptographic analysis for giant protocols. Even if ProVerif scales up remarkably well in practice, beyond a few message exchanges, or a few hundred lines of F#, verification becomes long (up to a few days) and unpredictable (with trivial code changes leading to divergence).

Cryptographic Verification meets Program Verification In parallel with specialist tools for cryptography, verification tools in general are also making rapid progress, and can deal with much larger programs [see for example Flanagan et al., 2002, Filliâtre, 2003, Barnett et al., 2005, Pottier and Régis-Gianas, 2007]. To verify the security of programs with some cryptography, we would like to combine both kinds of tools. However, this integration is delicate: the underlying assumptions of cryptographic models to account for active adversaries typically differ from those made for general-purpose program verification. On the other hand, modern applications involve a large amount of (non-cryptographic) code and extensive libraries, sometimes already verified; we'd rather benefit from this effort.

Authorization by Typing Logic is now a well established tool for expressing and reasoning about authorization policies. Although many systems rely on dynamic authorization engines that evaluate logical queries against local stores of facts and rules, it is sometimes possible to enforce policies statically. Thus, Fournet et al. [2007a,b] treat policy enforcement as a type discipline; they develop their approach for typed pi calculi, supplemented with cryptographic primitives. Relying on a "says" modality in the logic, they also account for partial trust (in logic specification) in the face of partial compromise (in their implementations). The present work is an attempt to develop, apply, and evaluate this approach for a general-purpose programming language.

Outline of the Implementation Our prototype tool takes as input module interfaces (similar to F# module interfaces but with extended types) and module implementations (in plain F#). It typechecks implementations against interfaces, and also generates plain F# interfaces by erasure. Using the F# compiler, generated interfaces and verified implementations can then be compiled as usual.

Our tool performs typechecking and partial type inference, relying on an external theorem prover for discharging the logical conditions generated by typing. We currently use plain first-order logic (rather than an authorization-specific logic) and delegate its proofs to Z3 [de Moura and Bjørner, 2008], a solver for Satisfiability Modulo Theories (SMT). Thus, in comparison with previous work, we still rely on an external prover, but this prover is being developed for general program verification, not for cryptography; also, we use this prover locally, to discharge proof obligations at various program locations, rather than rely on a global translation to a cryptographic model.

Reflecting our assumptions on cryptography and other system libraries, some modules have two implementations: a symbolic implementation used for extended typing and symbolic execution, and a concrete implementation used for plain typing and distributed execution. We have access to a collection of F# test programs already analyzed using dual implementations of cryptography [Bhargavan et al., 2007], so we can compare our new approach to prior work on model extraction to ProVerif. Unlike ProVerif, typechecking requires annotations that include pre- and post-conditions. On the other hand, these annotations can express general authorization policies, and their use makes typechecking more compositional and predictable than the global analysis performed by ProVerif. Moreover, typechecking succeeds even on code involving recursion and complex data structures.

Outline of the Theory We justify our extended typechecker by developing a formal type theory for a core of F#: a concurrent call-by-value λ-calculus.

To represent pre- and post-conditions, our calculus has standard dependent types and pairs, and a form of refinement types [Freeman and Pfenning, 1991, Xi and Pfenning, 1999]. A *refinement type* takes the form $\{x : T \mid C\}$; a value M of this type is a value of type T such that the formula $C\{M/x\}$ holds. (Another name for the construction is *predicate subtyping* [Rushby et al., 1998]; $\{x : T \mid C\}$ is the subtype of T characterized by the predicate C.)

To represent security properties, expressions may assume and assert formulas in first-order logic. An expression is *safe* when no assertion can ever fail at run time. By annotating programs with suitable formulas, we formalize security properties, such as authentication and authorization, as expression safety.

Our F# code is written in a functional style, so pre- and post-conditions concern data values and events represented by logical formulas; our type system does not (and need not for our purposes) directly support reasoning about mutable state, such as heap-allocated structures.

Contributions First, we formalize our approach within a typed concurrent λ-calculus. We develop a type system with refinement types that carry logical formulas, building on standard techniques for dependent types, and establish its soundness.

Second, we adapt our type system to account for active (untyped) adversaries, by extending subtyping so that all values manipulated by the adversary can be given a special universal type (Un). Our calculus has no built-in cryptographic primitives. Instead, we show how a wide range of cryptographic primitives can be coded (and typed) in the calculus, using a seal abstraction, in a generalization of the symbolic Dolev-Yao model. The corresponding robust safety properties then follow as a corollary of type safety.

Third, experimentally, we implement our approach as an extension of F#, and develop a new typechecker (with partial type inference) based on Z3 (a fast, incomplete, first-order logic prover).

Fourth, we evaluate our approach on a series of programming examples, involving authentication and authorization properties of protocols and applications; this indicates that our use of refinement types is an interesting alternative to global verification tools for cryptography, especially for the verification of executable reference implementations.

An online technical report provides details, proofs, and examples omitted from this version of the paper.

2 A Language with Refinement Types

Our calculus is an assembly of standard parts: call-by-value dependent functions, dependent pairs, sums, iso-recursive types, message-passing concurrency, refinement types, subtyping, and a universal type Un to model attacker knowledge. This is essentially the Fixpoint Calculus (FPC) [Gunter, 1992], augmented with concurrency and refinement types. Hence, we adopt the name Refined Concurrent FPC, or RCF for short. This section introduces its syntax, semantics, and type system (apart from Un), together with an example application. Section 3 introduces Un and applications to cryptographic protocols. (Any ambiguities in the informal presentation should be clarified by the semantics in Appendix B and the type system in Section 4.)

Expressions, Evaluation, and Safety An *expression* represents a concurrent, message-passing computation, which may return a *value*. A state of the computation consists of (1) a multiset of expressions being evaluated in parallel;

(2) a multiset of messages sent on channels but not yet received; and (3) the *log*, a multiset of assumed formulas. The multisets of evaluating expressions and unread messages model a configuration of a concurrent or distributed system; the log is a notional central store of logical formulas, used only for specifying correctness properties.

We write $S \models C$ to mean that a formula C logically follows from a set S of formulas. In our implementation, C is some formula in (untyped) first-order logic with equalities $M = N$ interpreted as syntactic identity between values. (Appendix A lists the (standard) syntax.)

We assume collections of *names*, *variables*, and *type variables*. A name is an identifier, generated at run time, for a channel, while a variable is a placeholder for a value. If ϕ is a phrase of syntax, we write $\phi\{M/x\}$ for the outcome of substituting a value M for each free occurrence of the variable x in ϕ. We identify syntax up to the capture-avoiding renaming of bound names and variables. We write $fnfv(\phi)$ for the set of names and variables occurring free in a phrase of syntax ϕ.

Syntax of Values and Expressions:

$v ::= a \mid x$	name or variable
$h ::= \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{fold}$	constructor
$M,N ::=$	value
v	name or variable
$()$	unit
$\mathbf{fun}\, x \rightarrow A$	function (scope of x is A)
(M,N)	pair
$h\,M$	construction
$A,B ::=$	expression
M	value
$M\,N$	application
$M = N$	syntactic equality
$\mathbf{let}\ x = A\ \mathbf{in}\ B$	let (scope of x is B)
$\mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$	pair split (scope of x, y is A)
$\mathbf{match}\ M\ \mathbf{with}$ $h\,x \rightarrow A\ \mathbf{else}\ B$	constructor match (scope of x is A)
$(\nu a)A$	restriction (scope of a is A)
$A \Rsh B$	fork
$M!N$	transmission of N on channel M
$M?$	receive message off channel
$\mathbf{assume}\ C$	assumption of formula C
$\mathbf{assert}\ C$	assertion of formula C

To evaluate M, return M at once. To evaluate $M\ N$, if $M = \mathbf{fun}\, x \rightarrow A$, evaluate $A\{N/x\}$. To evaluate $M = N$, if the two values M and N are the same, return $\mathbf{true} \stackrel{\triangle}{=} \mathsf{inr}()$; otherwise, return $\mathbf{false} \stackrel{\triangle}{=} \mathsf{inl}()$. To evaluate $\mathbf{let}\ x = A\ \mathbf{in}\ B$, first evaluate A; if evaluation returns a value M, evaluate $B\{M/x\}$. To evaluate $\mathbf{let}\ (x_1,x_2) = M\ \mathbf{in}\ A$, if $M = (N_1,N_2)$, evaluate $A\{N_1/x_1\}\{N_2/x_2\}$. To evaluate $\mathbf{match}\ M\ \mathbf{with}\ h\ x \rightarrow A\ \mathbf{else}\ B$, if $M = h\ N$ for some N, evaluate $A\{N/x\}$; otherwise, evaluate B.

To evaluate $(\nu a)A$, generate a globally fresh channel name c, and evaluate $A\{c/a\}$. To evaluate $A \Rsh B$, start a parallel thread to evaluate A (whose return value will be discarded), and evaluate B. To evaluate $M!N$, if $M = c$ for some name c, emit message N on channel c, and return $()$ at once. To evaluate $M?$, if $M = c$ for some name c, block until some message N is on channel c, remove N from the channel, and return N.

To evaluate **assume** C, add C to the log, and return $()$. To evaluate **assert** C, return $()$. If $S \models C$, where S is the set of logged formulas, we say the assertion *succeeds*; otherwise, we say the assertion *fails*. Either way, it always returns $()$.

Expression Safety:

An expression A is *safe* if and only if, in all evaluations of A, all assertions succeed. (see Appendix B for formal details.)

Types and Subtyping We assume a collection of *type variables*, for forming recursive types.

Syntax of Types:

$H,T,U ::=$	type
α	type variable
unit	unit type
$\Pi x : T.\ U$	dependent function type (scope of x is U)
$\Sigma x : T.\ U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
$(T)\mathsf{chan}$	channel type
$\{x : T \mid C\}$	refinement type (scope of x is C)
$\{C\} \stackrel{\triangle}{=} \{\_ : \mathsf{unit} \mid C\}$	ok-type

(The notation _ denotes an anonymous variable that by convention occurs nowhere else.)

A value of type unit is the unit value $()$. A value of type $\Pi x : T.\ U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$. A value of type $\Sigma x : T.\ U$ is a pair (M,N) such that M has type T and N has type $U\{M/x\}$. A value of type $T + U$ is either $\mathsf{inl}\ M$ where M has type T, or $\mathsf{inr}\ N$ where N has type U. A value of type $\mu\alpha.T$ is a construction $\mathsf{fold}\ M$, where M has the (unfolded) type $T\{\mu\alpha.T/\alpha\}$. A value of type $(T)\mathsf{chan}$ is a name c such that for any transmission $c!M$ on c, message M has type T. A value of type $\{x : T \mid C\}$ is a value M of type T such that the formula $C\{M/x\}$ follows from the log.

As usual, we can define syntax-directed typing rules for checking that the value of an expression is of type T, written $E \vdash A : T$, where E is a *typing environment*. The environment tracks the types of variables and names in scope. We write $\varnothing$ for the empty environment.

The core principle of our system is *safety by typing*:

Theorem 1 (Safety by Typing) *If $\varnothing \vdash A : T$ then A is safe.*

Section 4 has all the typing rules. The majority are standard. Here, we explain the intuitions for the rules concerning refinement types, assumptions, and assertions.

The judgment $E \models C$ means C is deducible from the formulas mentioned in refinement types in E. For example:

- If E includes $y : \{x : T \mid C\}$ then $E \models C\{y/x\}$.

Consider the refinement types $T_1 = \{x_1 : T \mid \mathsf{P}(x_1)\}$ and $T_2 = \{x_2 : \mathsf{unit} \mid \forall z.\mathsf{P}(z) \Rightarrow \mathsf{Q}(z)\}$. If $E = (y_1 : T_1, y_2 : T_2)$ then $E \models \mathsf{Q}(y_1)$ (via the rule above plus first-order logic).

The introduction rule for refinement types is as follows.

- If $E \vdash M : T$ and $E \models C\{M/x\}$ then $E \vdash M : \{x : T \mid C\}$.

A special case of refinement is an *ok-type*, written $\{C\}$, and short for $\{\_ : \mathsf{unit} \mid C\}$: a type of tokens that a formula holds. For example, up to variable renaming, $T_2 = \{\forall z.\mathsf{P}(z) \Rightarrow \mathsf{Q}(z)\}$. The specialized rules for ok-types are:

- If E includes $x : \{C\}$ then $E \models C$.
- A value of type $\{C\}$ is $()$, a token that C holds.

The type system includes a subtype relation $E \vdash T <: T'$, and the usual subsumption rule:

- If $E \vdash A : T$ and $E \vdash T <: T'$ then $E \vdash A : T'$.

Refinement relates to subtyping as follows. (To avoid confusion, note that True is a logical formula, which always holds, while **true** is a Boolean value, defined as $\mathsf{inr}\ ()$).

- If $T <: T'$ and $C \models C'$ then $\{x : T \mid C\} <: \{x : T' \mid C'\}$.
- $\{x : T \mid \mathsf{True}\} <:> T$.

For example, $\{x : T \mid C\} <: \{x : T \mid \mathsf{True}\} <: T$.

We typecheck **assume** and **assert** as follows.

- $E \vdash \mathbf{assume}\ C : \{C\}$.
- If $E \models C$ then $E \vdash \mathbf{assert}\ C : \mathsf{unit}$.

By typing the result of **assume** as $\{C\}$, we track that C can subsequently be assumed to hold. Conversely, for a well-typed **assert** to be guaranteed to succeed, we must check that C holds in E. This is sound because when typechecking any A in E, the formulas deducible from E are a lower bound on the formulas in the log whenever A is evaluated.

Formal Interpretation of our Typechecker We interpret a large class of F# expressions and modules within our calculus. To enable a compact presentation of the semantics of RCF, there are two significant differences between expressions in these languages. First, the formal syntax of RCF is in an intermediate, reduced form (reminiscent of A-normal form [Sabry and Felleisen, 1993]) where $\mathbf{let}\ x = A\ \mathbf{in}\ B$ is the only construct to allow sequential evaluation of expressions. As usual, $A;B$ is short for $\mathbf{let}\ \_ = A\ \mathbf{in}\ B$. More notably, if A and B are proper expressions rather than being values, the application $A\ B$ is short for $\mathbf{let}\ f = A\ \mathbf{in}\ (\mathbf{let}\ x = B\ \mathbf{in}\ f\ x)$. In general, the use in F# of arbitrary expressions in place of values can be interpreted by inserting suitable lets.

The second main difference is that the RCF syntax for communication and concurrency ($(\nu a)A$ and $A \Rsh B$ and $M?$ and $M!N$) is in the style of a process calculus. In F# we express communication and concurrency via a small library of functions, which is interpreted within RCF as follows.

Functions for Communication and Concurrency:

$\mathsf{chan} \triangleq \mathbf{fun}\, x \to (\nu a)a$	create new channel
$\mathsf{send} \triangleq \mathbf{fun}\, c \to \mathbf{fun}\, x \to (c!x \Rsh ())$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\, c \to \mathbf{let}\ x = c?\ \mathbf{in}\ x$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\, f \to (f() \Rsh ())$	run f in parallel

We also assume standard encodings of strings, numeric types, Booleans, tuples, records, algebraic types (including lists) and pattern-matching, and recursive functions. RCF lacks polymorphism, but by duplicating definitions at multiple monomorphic types we can recover the effect of having polymorphic definitions.

We use the following notations for functions with preconditions, and non-empty tuples (instead of directly using the core syntax for dependent function and pair types). We usually omit conditions of the form $\{\mathsf{True}\}$ in examples.

Derived Notation for Functions and Tuples:

$$[x_1 : T_1]\{C_1\} \to U \triangleq \Pi x_1 : \{x_1 : T_1 \mid C_1\}.\ U$$

$$(x_1 : T_1 * \cdots * x_n : T_n)\{C\} \triangleq \begin{cases} \Sigma x_1 : T_1. \ \ldots \Sigma x_{n-1} : T_{n-1}.\ \{x_n : T_n \mid C\} & \text{if } n > 0 \\ \{C\} & \text{otherwise} \end{cases}$$

To treat **assume** and **assert** as F# library functions, we follow the convention that constructor applications are interpreted as formulas (as well as values). If h is an algebraic type constructor of arity n, we treat h as a predicate symbol of arity n, so that $h(M_1, \ldots, M_n)$ is a formula.

All of our example code is extracted from two kinds of source files: either extended typed interfaces (.fs7) that declare types, values, and policies; or the corresponding F# implementation modules (.fs) that define them.

We sketch how to interpret interfaces and modules as tuple types and expressions. In essence, an *interface* is a sequence $\mathbf{val}\ x_1 : T_1\ \ldots\ \mathbf{val}\ x_n : T_n$ of *value declarations*, which we interpret by the tuple type $(x_1 : T_1 * \cdots * x_n : T_n)$. A *module* is a sequence $\mathbf{let}\ x_1 = A_1\ \ldots\ \mathbf{let}\ x_n = A_n$ of *value definitions*, which we interpret by the expression $\mathbf{let}\ x_1 = A_1\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = A_n\ \mathbf{in}\ (x_1, \ldots, x_n)$. If A and T are the interpretations of a module and an interface, our tool checks whether $A : T$. Any type declarations are simply interpreted as abbreviations for types, while a policy statement $\mathbf{assume}\ C$ is treated as a declaration $\mathbf{val}\ x : \{C\}$ plus a definition $\mathbf{let}\ x = \mathbf{assume}\ C$ for some fresh x.

Example: Access Control in Partially-Trusted Code This example illustrates static enforcement of file access control policies in code that is typechecked but not necessarily trusted, such as applets or plug-ins [Pottier et al., 2001, Abadi and Fournet, 2003, Abadi, 2006].

We first declare a type for the logical facts in our policy. We interpret each of its constructors as a predicate symbol: here, we have two basic access rights, for reading and writing a given file, and a property stating that a file is public.

```
type facts =
   CanRead of string // read access
  | CanWrite of string // write access
  | PublicFile of string // some file attribute
```

We use these facts to give restrictive types to sensitive primitives. For instance, the declarations

```
val read: file:string{CanRead(file)} →string
val delete: file:string{CanWrite(file)} →unit
```

demand that the function read be called only in contexts that have previously established the fact CanRead A for its string argument A (and similarly for write). These demands are enforced at compile time, so in F# the function read just has type string →string and its implementation may be left unchanged.

Library writers are trusted to include suitable **assume** statements. They may declare policies, in the form of logical deduction rules, declaring for instance that every file that is writable is also readable:

```
assume ∀x. CanWrite(x) ⇒CanRead(x)
```

and they may program helper functions that establish new facts. For instance, they may declare

```
val publicfile: file : string →unit{ PublicFile(file) }
assume ∀x. PublicFile(x) ⇒CanRead(x)
```

and implement publicfile as a partial function that dynamically checks its filename argument.

```
let publicfile f =
  if f = "C:/public/README" then assume (PublicFile(f))
  else failwith "not a public file"
```

where **let** $f\ x = A$ is short for **let** $f =$ **fun** $x \to A$.

The F# library function failwith throws an exception, so it never returns and can safely be given the polymorphic type string $\to\alpha$, where α can be instantiated to any RCF type. (We also coded more realistic dynamic checks, based on dynamic lookups in mutable, refinement-typed, access-control lists. We omit their code for brevity.)

To illustrate our code, consider a few sample files, one of them writable:

```
let pwd = "C:/etc/password"
let readme = "C:/public/README"
let tmp = "C:/temp/tempfile"
let _ = assume (CanWrite(tmp))
```

Typechecking the test code below reports two type errors:

```
let test =
  delete tmp; // ok
  delete pwd; // type error
  let v1 = read tmp in // ok, using 1st logical rule
  let v2 = read readme in // type error
  publicfile readme; let v3 = read readme in () // ok
```

For instance, the second delete yields the error "Cannot establish formula CanWrite(pwd) at acls.fs(39,9)-(39,12)."

In the last line, the call to publicfile dynamically tests its argument, ensuring PublicFile(readme) whenever the final expression read readme is evaluated. This fact is recorded in the environment for typing the final expression.

From the viewpoint of fully-trusted code, our interface can be seen as a self-inflicted discipline—indeed, one may simply **assume** ∀x.CanRead(x). In contrast, partially-trusted code (such as mobile code) would not contain any **assume**. By typing this code against our library interface, possibly with a policy adapted to the origin of the code, the host is guaranteed that this code cannot call read or write without first obtaining the appropriate right.

Although access control for files mostly relies on dynamic checks (ACLs, permissions, and so forth), a static typing discipline has advantages for programming partially-trusted code: as long as the program typechecks, one can safely re-arrange code to more efficiently perform costly dynamic checks. For example, one may hoist a check outside a loop, or move it to the point a function is created, rather than called, or move it to a point where it is convenient to handle dynamic security exceptions.

In the code below, for instance, the function reader can be called to access the content of file readme in any context with no further run time check.

```
let test_higher_order =
  let reader = (publicfile readme; (fun () →read readme)) in
  let v4 = read readme in // type error
  let v5 = reader () in () // ok
```

Similarly, we programmed (and typed) a function that merges the content of all files included in a list, under the assumption that all these files are readable, declared as

```
val merge: (file:string{ CanRead(file) }) list →string
```

where list is a type constructor for lists, with a standard implementation typed in RCF.

3 Modelling Cryptographic Protocols

Following Bhargavan et al. [2007], we start with plain F# functions that create instances of each role of the protocol (such as client or server). The protocols make use of various libraries (including cryptographic functions, explained below) to communicate messages on channels that represent the public network. We model the whole protocol as an F# module, interpreted as before as an expression that exports the functions representing the protocol roles, as well as the network channel [Sumii and Pierce, 2007]. We express authentication properties (correspondences [Woo and Lam, 1993]) by embedding suitable **assume** and **assert** expressions within the code of the protocol roles.

The goal is to verify that these properties hold in spite of an active opponent able to send, receive, and apply cryptography to messages on network channels [Needham and Schroeder, 1978]. We model the opponent as some arbitrary (untyped) expression O which is given access to the protocol and knows the network channels [Abadi and Gordon, 1999]. The idea is that O may use the communication and concurrency features of RCF to create arbitrary parallel instances of the protocol roles, and to send and receive messages on the network channels, in an attempt to force failure of an **assert** in protocol code. Hence, our formal goal is *robust safety*, that no **assert** fails, despite the best efforts of an arbitrary opponent.

Formal Threat Model: Opponents and Robust Safety

An expression O is an *opponent* iff O contains no occurrence of **assert** and each type annotation within O is Un.
An expression A is *robustly safe* iff the application $O\ A$ is safe for all opponents O.

(An opponent must contain no **assert**, or less it could vacuously falsify safety. The constraint on type annotations is a technical convenience; it does not affect the expressiveness of opponents.)

Typing the Opponent To allow type-based reasoning about the opponent, we introduce a *universal type* Un of data known to the opponent, much as in earlier work [Abadi, 1999, Gordon and Jeffrey, 2003a]. By definition, Un is type equivalent to (both a subtype and a supertype of) all of the following types: unit, $(\Pi x : \mathsf{Un}.\ \mathsf{Un})$, $(\Sigma x : \mathsf{Un}.\ \mathsf{Un})$, $(\mathsf{Un} + \mathsf{Un})$, $(\mu\alpha.\mathsf{Un})$, and $(\mathsf{Un})\mathsf{chan}$. Hence, we obtain *opponent typability*, that $O : \mathsf{Un}$ for all opponents O.

It is useful to characterize two *kinds* of type: *public types* (of data that may flow to the opponent) and *tainted types* (of data that may flow from the opponent).

Public and Tainted Types:

Let a type T be *public* if and only if $T <: \mathsf{Un}$.
Let a type T be *tainted* if and only if $\mathsf{Un} <: T$.

We can show that refinement types satisfy the following kinding rules. (Section 4 has kinding rules for the other types, following prior work [Gordon and Jeffrey, 2003b].)

- $E \vdash \{x : T \mid C\} <: \mathsf{Un}$ iff $E \vdash T <: \mathsf{Un}$
- $E \vdash \mathsf{Un} <: \{x : T \mid C\}$ iff $E \vdash \mathsf{Un} <: T$ and $E, x : T \models C$

Consider the type $\{x : \mathsf{string} \mid \mathsf{CanRead}(x)\}$. According to the rules above, this type is public, because string is public, but it is only tainted if $\mathsf{CanRead}(x)$ holds for all x. If we have a value M of this type we can conclude $\mathsf{CanRead}(M)$. The type cannot be tainted, for if it were, we could conclude $\mathsf{CanRead}(M)$ for any M chosen by the opponent. It is the presence of such non-trivial refinement types that prevents all types from being equivalent to Un.

Verification of protocols versus an arbitrary opponent is based on a principle of *robust safety by typing*.

Theorem 2 (Robust Safety by Typing) *If* $\varnothing \vdash A : \mathsf{Un}$ *then* A *is robustly safe.*

To apply the principle, if expression A and type T are the RCF interpretations of a protocol module and a protocol interface, it suffices by subsumption to check that $A : T$ and T is public. The latter amounts to checking that T_i is public for each declaration **val** $x_i : T_i$ in the protocol interface.

A Cryptographic Library We provide various libraries to support distributed programming. They include polymorphic functions for producing and parsing network representations of values, declared as

```
val pickle: x:α →(p:α pickled)
val unpickle: p:α pickled →(x:α)
```

and for messaging: addr is the type of TCP duplex connections, established by calling connect and listen, and used by calling send and recv. All these functions are public.

The cryptographic library provides a typed interface to a range of primitives, including hash functions, symmetric encryption, asymmetric encryption, and digital signatures. We detail the interface for HMACSHA1, a keyed hash function, used in our examples to build messages authentication codes (MACs). This interface declares

```
type α hkey = HK of α pickled Seal
type hmac = HMAC of Un
val mkHKey: unit → α hkey
val hmacsha1: α hkey → α pickled → hmac
val hmacsha1Verify: α hkey → Un → hmac → α pickled
```

where hmac is the type of hashes and α hkey is the type of keys used to compute hashes for values of type α.

The function mkHKey generate a fresh key (informally fresh random bytes). The function hmacsha1 computes the joint hash of a key and a value with matching types α. The function hmacsha1Verify verifies whether the joint hash of a key and a value (a priori the pickled representation of any type β) match some given hash. If verification succeeds, this value is returned, now with the type α indicated in the key. Otherwise, an exception is raised.

Although keyed-hash verification is concretely implemented by recomputing the hash and comparing it to the given hash, this would not meet its typed interface: assume

α is the refinement type ⟨x:string⟩{CanRead(x)}. In order to hash a string x, one needs to prove CanRead(x) as a precondition for calling hmacsha1. Conversely, when receiving a keyed hash of x, one would like to obtain CanRead(x) as a postcondition of the verification—indeed, the result type of hmacsha1Verify guarantees it. At the end of this section, we describe a well-typed symbolic implementation of this interface.

Example: A Protocol based on MACs Our first cryptographic example implements a basic one-message protocol with a message authentication code (MAC) computed as a shared-keyed hash; it is a variant of a protocol described and verified in earlier work [Bhargavan et al., 2007].

We present snippets of the protocol code to illustrate our typechecking method; Appendix C lists the full source code for a similar, but more general protocol. We begin with a typed interface, declaring three types: event for specifying our authentication property; content for authentic payloads; and message for messages exchanged on a public network.

```
type event = Send of string // a type of logical predicate
type content = x:string{Send(x)} // a string refinement
type message = (string ∗ hmac) pickled // a wire format
```

The interface also declares functions, client and server, for invoking the two roles of the protocol.

```
val addr : (string ∗ hmac, unit) addr // a public server address
private val hk: content hkey // a shared secret

private val make: content hkey →content →message
val client: string →unit // start a client

private val check: content hkey →message →content
val server: unit →unit // start a server
```

The client and server functions share two values: a public network address addr where the server listens, and a shared secret key hk. Given a string argument s, client calls the make function to build a protocol message by calling hmacsha1 hk (pickled s). Conversely, on receiving a message at addr, server calls the check function to check the message by calling hmacsha1Verify.

In the interface, values marked as **priv** may occur only in typechecked implementations. Conversely, the other values (addr, client, server) are available to the opponent, as well as Un-typed values declared in libraries.

Authentication is expressed using a single event Send(s) recording that the string s has genuinely been sent by the client—formally, that client(s) has been called. This event is embedded in a refinement type, content, the type of strings s such that Send(s). Thus, following the type declarations for make and check, this event is a pre-condition for building the message, and a post-condition after successfully checking the message.

Consider the following code for client and server:

```
let client text =
  assume (Send(text)); // privileged, carefully review
  let c = connect addr in
  send c (make hk text)
let server () =
  let c = listen addr in
  let text = check hk (recv c) in
  assert(Send text) // guaranteed by typing
```

The calls to **assume** before building the message and to **assert** after checking the message have no effect at run time (the implementations of these functions simply return ()) but they are used to specify our security policy. In the terminology of cryptographic protocols, **assume** marks a "begin" event, while **assert** marks an "end" event.

Here, the server code expects that the call to check only returns text values previously passed as arguments to client. This guarantee follows from typing, by relying on the types of the shared key and cryptographic functions. On the other hand, this guarantee does not presume any particular cryptographic implementation—indeed, simple variants of our protocol may achieve the same authentication guarantee, for example, by authenticated encryption or digital signature.

Conversely, some implementation mistakes would result in a compile-time type error indicating a possible attack. For instance, removing **priv** from the declaration of the authentication key hk, or attempting to leak hk within client would not be type-correct; indeed, this would introduce an attack on our desired authentication property.

Example: Principals and Compromise We now extend our example with multiple principals, with a shared key between each pair of principals. Hence, the keyed hash authenticates not only the message content, but also the sender and the intended receiver. The full implementation is in Appendix C; here we give only the types.

We represent principal names as strings; Send events are now parameterized by the sending and receiving principals.

```
type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type (;a:prin,b:prin) content = x:string{ Send(a,b,x) }
```

The second event Leak is used in our handling of principal compromise, as described below. The type definition of content has two *value parameters*, a and b; they bind expression variables in the type being defined, much like type parameters bind type variables. (Value parameters appear after type parameters, separated by a semicolon; here, content has no type parameters before the semicolon.)

We store the keys in a (typed, list-based) private database containing entries of the form (a,b,k) where k is a symmetric key of type (;a,b)content shared between a and b.

```
val genKey: prin →prin →unit
private val getKey: a:
  string →b:string →((;a,b) content) hkey
```

Trusted code can call getKey a b to retrieve a key shared between a and b. Both trusted and opponent code can also call genKey a b to trigger the insertion of a fresh key into the database.

To model the possibility of key leakage, we allow opponent code to obtain a key by calling the function leak:

```
assume ∀a,b,x. ( Leak(a) ) ⇒ Send(a,b,x)
val leak:
  a:prin → b:prin → (unit{ Leak(a) }) ∗ ((;a,b) content) hkey
```

This function first assumes Leak(a), as recorded in its result type, then calls getKey a b and returns the key. Since the opponent gets a key shared between a and b, it can generate seemingly authentic messages on a's behalf; accordingly, we declare the policy that Send(a,b,x) holds for any x after the compromise of a, so that leak can be given a public type—without this policy, a subtyping check fails during typing.

Implementing Formal Cryptography Morris [1973] describes *sealing*, a programming language mechanism to provide "authentication and limited access." Sumii and Pierce [2007] provide a primitive semantics for sealing within a λ-calculus, and observe the close correspondence between sealing and various formal characterizations of symmetric-key cryptography.

In our notation, a *seal* k for a type T is a pair of functions: the *seal function for* k, of type $T \rightarrow$ Un, and the *unseal function for* k, of type Un $\rightarrow T$. The seal function, when applied to M, wraps up its argument as a *sealed value*, informally written $\{M\}_k$ in this discussion. This is the only way to construct $\{M\}_k$. The unseal function, when applied to $\{M\}_k$, unwraps its argument and returns M. This is the only way to retrieve M from $\{M\}_k$. Sealed values are opaque; in particular, the seal k cannot be retrieved from $\{M\}_k$.

We declare a type of seals, and a function mkSeal to create a fresh seal, as follows.

```
type α Seal = (α → Un) ∗ (Un → α)
val mkSeal: unit → α Seal
```

To implement a seal k, we maintain a list of pairs $[(M_1,a_1);\ldots;(M_n,a_n)]$. The list records all the values M_i that have so far been sealed with k. Each a_i is a fresh name representing the sealed value $\{M_i\}_k$. The list grows as more values are sealed; we associate a channel s with the seal k, and store the current list as the one and only message on s. We maintain the invariant that both the M_i and the a_i are pairwise distinct: the list is a one-to-one correspondence.

The function mkSeal below creates a fresh seal, by generating a fresh channel s; the seal itself is the pair of functions (seal s, unseal s). The code uses the channel-based abbreviations chan, send, and recv displayed in Section 2. The code also relies on library functions for list lookups: the function first, of type ($\alpha \rightarrow \beta$ option)$\rightarrow \alpha$ list $\rightarrow \beta$ option, takes as parameters a function and a list; it applies the function to the elements of the list, and returns its first non-None result, if any; otherwise it returns None. This function is applied to a pair-filtering function left, defined as **let** left z (x,y)= **if** z = x **then** Some y **else** None, to retrieve the first a_i associated with the value being sealed, if any, and is used symmetrically with a function right to retrieve the first M_i associated with the value being unsealed, if any.

```
type α SealChan = ((α ∗ Un) list) Pi.chan
let seal: α SealChan → α → Un = fun s M →
  let state = recv s in match first (left M) state with
  | Some(a) → send s state; a
  | None →
    let a: Un = Pi.name "a" in
    send s ((M,a)::state); a
let unseal: α SealChan → Un → α = fun s a →
  let state = recv s in match first (right a) state with
  | Some(M) → send s state; M
  | None → failwith "not a sealed value"
let mkSeal () : α Seal =
  let s:α SealChan = chan "seal" in
  send s []; (seal s, unseal s)
```

Within RCF, we derive formal versions of cryptographic operations, in the spirit of Dolev and Yao [1983], but based on sealing rather than algebra. Our technique depends on being within a calculus with functional values. Thus, in contrast with previous work in cryptographic pi calculi [Gordon and Jeffrey, 2003b, Fournet et al., 2007b] where all cryptographic functions were defined and typed as primitives, we can now implement these functions and retrieve their typing rules by typechecking their implementations.

As an example, we derive a formal model of the functions we use for HMACSHA1 in terms of seals as follows.

```
let mkHKey ():α hkey = HK (mkSeal ())
let hmacsha1 (HK key) text = HMAC (fst key text)
let hmacsha1Verify (HK key) text (HMAC h) =
  let x:α pickled = snd key h in
    if x = text then x else failwith "hmac verify failed"
```

Similarly, we derive functions for symmetric encryption (AES), asymmetric encryption (RSA), and digital signatures (RSASHA1).

4 A Type System for Robust Safety

We describe the full type system.

Judgments, and Syntax of Environments:

$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, type T is syntactically well-formed
$E \models C$	formula C is derivable from E
$E \vdash T :: \nu$	in E, type T has kind $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Syntax of Typing Environments:

$\mu ::=$	environment entry
$\quad \alpha$	type variable
$\quad \alpha :: \{\mathbf{pub}, \mathbf{tnt}\}$	kinding
$\quad a : (T)\mathsf{chan}$	name (of channel type)
$\quad x : T$	variable (of any type)
$E ::= \mu_1, \ldots, \mu_n$	environment

A name can only have a channel type. If $E = \mu_1, \ldots, \mu_n$ we write $\mu \in E$ to mean that $\mu = \mu_i$ for some $i \in 1..n$. We write $T <:> T'$ for $T <: T'$ and $T' <: T$. Let $\mathit{dom}(E)$ be the set of type variables, names, and variables defined in E. Let $\mathit{fnfv}(E) = \bigcup\{\mathit{fnfv}(T) \mid (u : T) \in E\}$.

Rules of Well-Formedness and Deduction:

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{\begin{array}{c} E \vdash \diamond \\ \mathit{fnfv}(\mu) \subseteq \mathit{dom}(E) \\ \mathit{dom}(\mu) \cap \mathit{dom}(E) = \varnothing \end{array}}{E, \mu \vdash \diamond} \qquad \frac{\begin{array}{c} E \vdash \diamond \\ \mathit{fnfv}(T) \subseteq \mathit{dom}(E) \end{array}}{E \vdash T}$$

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E) \quad \mathsf{forms}(E) \models C}{E \models C}$$

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup \mathsf{forms}(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

The next set of rules axiomatizes the sets of public and tainted types, of data that can flow to or from the opponent.

Kinding Rules: $E \vdash T :: \nu$ **for** $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$

$$\frac{E \vdash \diamond \quad (\alpha :: \{\mathbf{pub}, \mathbf{tnt}\}) \in E}{E \vdash \alpha :: \nu} \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: \nu}$$

$$\frac{\begin{array}{c} E \vdash T :: \mathbf{tnt} \\ E, x : T \vdash U :: \mathbf{pub} \end{array}}{E \vdash (\Pi x : T.\ U) :: \mathbf{pub}} \qquad \frac{\begin{array}{c} E \vdash T :: \mathbf{pub} \\ E, x : T \vdash U :: \mathbf{tnt} \end{array}}{E \vdash (\Pi x : T.\ U) :: \mathbf{tnt}}$$

$$\frac{E \vdash T :: \nu \quad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\ U) :: \nu} \qquad \frac{E \vdash T :: \nu \quad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}$$

$$\frac{\begin{array}{c} E, \alpha :: \{\mathbf{pub}, \mathbf{tnt}\} \vdash T :: \mathbf{pub} \\ E, \alpha :: \{\mathbf{pub}, \mathbf{tnt}\} \vdash T :: \mathbf{tnt} \end{array}}{E \vdash (\mu\alpha.T) :: \nu} \qquad \frac{\begin{array}{c} E \vdash T :: \mathbf{pub} \\ E \vdash T :: \mathbf{tnt} \end{array}}{E \vdash (T)\mathsf{chan} :: \nu}$$

$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}} \qquad \frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \models C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$$

The following rules of subtyping are standard [Cardelli, 1986, Pierce and Sangiorgi, 1996, Aspinall and Compagnoni, 2001]. The two rules for subtyping refinement types are the same as in Sage [Gronski et al., 2006].

Subtype: $E \vdash T <: U$

$$\frac{E \vdash T :: \mathbf{pub} \quad E \vdash U :: \mathbf{tnt}}{E \vdash T <: U} \qquad \frac{E \vdash \diamond \quad \alpha \in \mathit{dom}(E)}{E \vdash \alpha <: \alpha}$$

$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}} \qquad \frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\ U) <: (\Pi x : T'.\ U')}$$

$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')}$$

$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + T') <: (U + U')} \qquad \frac{E, \alpha \vdash T <:> T'}{E \vdash (\mu\alpha.T) <: (\mu\alpha.T')}$$

$$\frac{E \vdash T <: T' \quad E \vdash T' <: T}{E \vdash (T)\mathsf{chan} <: (T')\mathsf{chan}}$$

$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \qquad \frac{E \vdash T <: T' \quad E, x : T \models C}{E \vdash T <: \{x : T' \mid C\}}$$

The universal type Un is to be type equivalent to all types that are both public and tainted; we (arbitrarily) define $\mathsf{Un} \triangleq (\mathsf{unit})\mathsf{chan}$. We can show that this definition satisfies the intended meaning: $E \vdash T :: \mathbf{pub}$ iff $E \vdash T <: \mathsf{Un}$, and $E \vdash T :: \mathbf{tnt}$ iff $E \vdash \mathsf{Un} <: T$.

The following congruence rule for refinement types is derivable from the two primitive rules for refinement types. We also list the special case for ok-types.

$$\frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \models C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}} \qquad \frac{E, \_ : \{C\} \models C'}{E \vdash \{C\} <: \{C'\}}$$

Next, we present the rules for typing values. The rule for constructions $h\,M$ depends on an auxiliary relation $h : (T, U)$ that delimits the possible argument T and result U of each constructor h.

Rules for Values: $E \vdash A : T$

$$\frac{E \vdash \diamond \quad (v : T) \in E}{E \vdash v : T} \qquad \frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

$$\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun}\, x \to A : (\Pi x : T.\ U)} \qquad \frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\Sigma x : T.\ U)}$$

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\,M : U} \qquad \frac{E \vdash M : T \quad E \models C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

$$\mathsf{inl}{:}(T, T{+}U) \qquad \mathsf{inr}{:}(U, T{+}U) \qquad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$$

Our final set of rules is for typing arbitrary expressions. In the rules for pattern-matching pairs and constructions, we use equations within refinement types to track information about the matched variables.

Rules for Expressions: $E \vdash A : T$

$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'} \qquad \frac{E \vdash M : (\Pi x : T.\ U) \quad E \vdash N : T}{E \vdash M\,N : U\{N/x\}}$$

$$\frac{E \vdash M : (\Sigma x : T.\, U) \quad E, x : T, y : U, \_ : \{(x,y) = M\} \vdash A : V \quad \{x,y\} \cap fv(V) = \varnothing}{E \vdash \textbf{let } (x,y) = M \textbf{ in } A : V}$$

$$\frac{E \vdash M : T \quad h : (H,T) \quad E, x : H, \_ : \{h\, x = M\} \vdash A : U \quad x \notin fv(U) \quad E, \_ : \{\forall x. h\, x \neq M\} \vdash B : U}{E \vdash \textbf{match } M \textbf{ with } h\, x \rightarrow A \textbf{ else } B : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \{b : \mathsf{bool} \mid b = \textbf{true} \Leftrightarrow M = N\}}$$

$$\frac{E \vdash \diamond \quad fnfv(C) \subseteq dom(E)}{E \vdash \textbf{assume } C : \{C\}} \qquad \frac{E \models C}{E \vdash \textbf{assert } C : \mathsf{unit}}$$

$$\frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin fv(U)}{E \vdash \textbf{let } x = A \textbf{ in } B : U}$$

$$\frac{E, a : (T)\mathsf{chan} \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a)A : U}$$

$$\frac{E \vdash M : (T)\mathsf{chan} \quad E \vdash N : T}{E \vdash M!N : \mathsf{unit}} \qquad \frac{E \vdash M : (T)\mathsf{chan}}{E \vdash M? : T}$$

$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}$$

The final rule, for $A_1 \Rsh A_2$, relies on an auxiliary function to extract the top-level formulas from A_2 for use while typechecking A_1, and to extract the top-level formulas from A_1 for use while typechecking A_2. The function $\overline{A}$ returns a formula representing the conjunction of each C occurring in a top-level **assume** C in an expression A, with restricted names existentially quantified.

Formula Extraction: $\overline{A}$

$\overline{(\nu a)A} = (\exists a.\overline{A})$ $\qquad$ $\overline{A_1 \Rsh A_2} = (\overline{A_1} \wedge \overline{A_2})$

$\overline{\textbf{let } x = A_1 \textbf{ in } A_2} = \overline{A_1}$ $\qquad$ $\overline{\textbf{assume } C} = C$

$\overline{A} = \mathsf{True}$ if A matches no other rule

5 Implementing Refinement Types for F#

We implement a typechecker that takes as input a series of extended RCF interface files and F# implementation files and, for every implementation file, perform the following tasks: (1) typecheck the implementation against its RCF interface, and any other RCF interfaces it may use; (2) kind-check its RCF interface, ensuring that every public value declaration has a public type; and then (3) generate a plain F# interface by erasure from its RCF interface. The programming of these tasks almost directly follows from our type theory. In the rest of this section, we only highlight some design choices and implementation decisions.

Handling F# Language Features Our typechecker processes F# programs with many more features than the calculus of Section 2. Thus, type definitions also feature mutual recursion, algebraic datatypes, type abbreviations, and record types; value definitions also feature mutual recursion, polymorphism, nested patterns in let- and match-expression, records, exceptions, and mutable references. As described in Section 2, these constructs can be expanded out to simpler types and expressions within RCF.

Annotating Standard Libraries Any F# program may use the set of *pervasive* types and functions in the standard library; this library includes operations on built-in types such as strings, Booleans, lists, options, and references, and also provides system functions such as reading and writing files and pretty-printing. Hence, to check a program, we must provide the typechecker with declarations for all the standard library functions and types it uses. When the types for these functions are F# types, we can simply use the F# interfaces provided with the library and trust their implementation. However, if the program relies on extended RCF types for some library functions, we must provide our own RCF interface. For example, the following code declare two functions on lists:

```
assume
  (∀x, u. Mem(x,x::u)) ∧
  (∀x, y, u. Mem(x,u) ⇒ Mem(x,y::u)) ∧
  (∀x, u. Mem(x,u) ⇒ (∃y, v. u = y::v ∧ (x = y ∨ Mem(x,v))))
val mem: x:α → u:α list → r:bool{r=true ⇒ Mem(x,u)}
val find: (α → bool) → (u:α list → r:α{ Mem(r,u) })
```

We declare an inductive predicate Mem for list membership and use it to annotate the two library functions for list membership (mem) and list lookup (find). Having defined these extended RCF types, we have a choice: we may either trust that the library implementation satisfies these types, or reimplement these functions and typecheck them. For lists, we reimplement (and re-typecheck) these functions; for other library modules such as String and Printf, we trust the F# implementation.

Implementing Trusted Libraries In addition to the standard library, our F# programs rely on libraries for cryptography and networking. We write their concrete implementations on top of .NET Framework classes. For instance, we define keyed hash functions as:

```
open System.Security.Cryptography
type α hkey = bytes
type hmac = bytes
let mkHKey () = mkNonce()
let hmacsha1 (k:α hkey) (x:bytes) =
  (new HMACSHA1 (k)).ComputeHash x
let hmacsha1Verify (k:α hkey) (x:bytes) (h:bytes) =
  let hh = (new HMACSHA1 (k)).ComputeHash x in
    if h = hh then x else failwith "hmac verify failed"
```

	F# Definitions	F# Declarations	RCF Declarations	Analysis Time	Z3 Obligations
Typed Libraries	440 lines	125 lines	146 lines	12.1s	12
Access Control (Section 2)	104 lines	16 lines	34 lines	8.3s	16
MAC Protocol (Section 3)	40 lines	9 lines	12 lines	2.5s	3
Logs and Queries	37 lines	10 lines	16 lines	2.8s	6
Principals & Compromise (Section 3)	48 lines	13 lines	26 lines	3.1s	12
Flexible Signatures (Section 6)	167 lines	25 lines	52 lines	14.6s	28

Table 1. Typechecking Example Programs

Similarly, the network send and recv are implemented using TCP sockets (and not typechecked in RCF).

We also write symbolic implementations for cryptography and networking, coded using seals and channels, and typechecked against their RCF interfaces. These implementations can also be used to compile and execute programs symbolically, sending messages on local channels (instead of TCP sockets) and computing sealed values (instead of bytes); this is convenient for testing and debugging, as one can inspect the symbolic structure of all messages.

Type Annotations and Partial Type Inference Type inference for dependently-typed calculi, such as RCF, is undecidable in general. For top-level value definitions, we require that all types be explicitly declared. For subexpressions, our typechecker performs type inference using standard unification-based techniques for plain F# types (polymorphic functions, algebraic datatypes) but it may require annotations for types carrying formulas.

Generating Proof Obligations for Z3 Following our typing rules, our typechecker must often establish that a condition follows from the current typing environment (such as when typing function applications and kinding value declarations). If the formula trivially holds, the typechecker discharges it; for more involved first-order-logic formulas, it generates a proof obligation in the Simplify format [Detlefs et al., 2005] and invokes the Z3 prover. Since Z3 is incomplete, it sometimes fails to prove a valid formula.

The translation from RCF typing environments to Simplify involves logical re-codings. Thus, constructors are coded as injective, uninterpreted, disjoint functions. Hence, for instance, a type definition for lists

type (α) list = Cons **of** $\alpha * \alpha$ list | Nil

generates logical declarations for a constant Nil and a binary function Cons, and the two assumptions

assume $\forall$x,y. Cons(x,y) $\neq$ Nil.
assume $\forall$x,y,x',y'.
  (x = x' $\wedge$ y = y') $\leftrightarrow$ Cons(x,y) = Cons(x',y').

Each constructor also defines a predicate symbol that may be used in formulas. Not all formulas can be translated to first-order-logic; for example, equalities between functional values cannot be translated and are rejected.

Evaluation We have typechecked all the examples of this paper and a few large programs. Table 1 summarizes our results; for each example, it gives the number of lines of typed F# code, of generated F# interfaces, and of declarations in RCF interfaces, plus typechecking time, and the number of proof obligations passed to Z3. Since F# programmers are expected to write interfaces anyway, the line difference between RCF and F# declarations roughly indicates the additional annotation burden of our approach.

The first row is for typechecking our symbolic implementations of lists, cryptography, and networking libraries. The second row is an extension of the access control example of Section 2; the next three rows are variants of the MAC protocol of Section 3. The final row implements the protocol described next in Section 6.

The examples in this paper are small programs designed to exercise the features of our type system; our results indicate that typechecking is fast and that annotations are not too demanding. In comparison with an earlier tool FS2PV that compiles F# code to ProVerif [Bhargavan et al., 2007], our typechecker succeeds on examples with recursive functions, such as the last row in Table 1, where ProVerif fails to terminate. We expect our method to scale better to larger examples, since we can typecheck one module at a time, rather than construct a large ProVerif model. On the other hand, FS2PV requires no type annotations, and ProVerif can also prove injective correspondences and equivalence-based properties [Blanchet et al., 2008].

6 Application: Flexible Signatures

We illustrate the controlled usage of cryptographic signatures with the same key for different intents, or different protocols. Such reuse is commonplace in practice (at least for long-term keys) but it is also a common source of errors (see Abadi and Needham [1996]), and it complicates protocol verification.

The main risk is to issue *ambiguous signatures*. As an informal design principle, one should ensure that, whenever a signature is issued, (1) its content follows from the current protocol step; and (2) its content cannot be interpreted otherwise, by any other protocol that may rely on the same key. To this end, one may for instance sign nonces, identities, session identifiers, and tags as well as the message payloads to make the signature more specific.

Our example is adapted from protocol code for XML digital signatures, as prescribed in web services security standards [Eastlake et al., 2002, Nadalin et al., 2004]. These signatures consist of an XML "signature information", which represents a list of (hashed) elements covered by the signature, together with a binary "signature value", a signed cryptographic hash of the signature information. Web services normally treat received signed-information lists as sets, and only check that these sets cover selected elements of the message—possibly fewer than those signed, to enable partial erasure as part of intermediate message processing. This flexibility induces protocol weaknesses in some configurations of services. For instance, by providing carefully-crafted inputs, an adversary may cause a naive service to sign more than intended, and then use this signature (in another XML context) to gain access to another service.

For simplicity, we only consider a single key and two interpretations of messages. We first declare types for these interpretations (either requests or responses) and their network representation (a list of elements plus their joint signature).

```
type id = int // representing message GUIDs
type events =
    Request of id ∗ string // id and payload
  | Response of id ∗ id ∗ string // id, request id, and payload
type element =
    IdHdr of id // Unique message identifier
  | InReplyTo of id // Identifier for some related messsage
  | RequestBody of string // Payload for a request message
  | ResponseBody of string // Payload for a response message
  | Whatever of string // Any other elements
type siginfo = element list
type msg = siginfo ∗ dsig
```

Depending on their constructor, signed elements can be interpreted for requests (RequestBody), responses, (InReplyTo, ResponseBody), both (IdHdr), or none (Whatever). We formally capture this intent in the type declaration of the information that is signed:

```
type verified = x:siginfo{
  (∀id, b.(Mem(IdHdr(id),x) ∧ Mem(RequestBody(b),x))
                                    ⇒ Request(id,b) )
∧ (∀id, req, b.(Mem(IdHdr(id),x) ∧ Mem(ResponseBody(b),x)
      ∧ Mem(InReplyTo(req),x)) ⇒ Response(id,req,b) ) }
```

Thus, the logical meaning of a signature is a conjunction of message interpretations, each guarded by a series of conditions on the elements included in the signature information.

We only present code for requests. We use the following declarations for the key pair and for message processing.

```
private val sk: verified privkey
val vk: verified pubkey
private val mkMessage: verified → msg
private val isMessage: msg → verified
type request = (id:id ∗ b:string){ Request(id,b) }
val isRequest: msg → request
private val mkPlainRequest: request → msg
private val mkRequest: request → siginfo → msg
```

To accept messages as a genuine requests, we just verify its signature and find two relevant elements in the list:

```
let isMessage (msg,dsig) =
  unpickle (rsasha1Verify vk (pickle msg) dsig)
let isRequest msg =
  let si = isMessage msg in (find_id si, find_request si)
```

For producing messages, we may define (and type):

```
let mkMessage siginfo = (siginfo, rsasha1 sk (pickle siginfo))
let mkPlainRequest (id,payload) =
  mkMessage (IdHdr(id)::RequestBody(payload)::[])
let mkRequest (id,payload) extra : msg =
  check_harmless extra;
  mkMessage (IdHdr(id)::RequestBody(payload)::extra)
```

While mkPlainRequest uses a fixed list of signed elements, mkRequest takes further elements to sign as an extra parameter. In both cases, typing the list with the refinement type verified ensures (1) Request(id,b), from its input refinement type; and (2) that the list does not otherwise match the two clauses within verified. For mkRequest, this requires some dynamic input validation check_harmless extra where check_harmless is declared as

```
val check_harmless: x: siginfo → r: unit {
  ( ∀s. not(Mem(IdHdr(s),x)))
∧ ( ∀s. not(Mem(InReplyTo(s),x)))
∧ ( ∀s. not(Mem(RequestBody(s),x)))
∧ ( ∀s. not(Mem(ResponseBody(s),x))) }
```

and recursively defined as

```
let rec check_harmless m = match m with
  | IdHdr(_)::_ → failwith "bad"
  | InReplyTo(_)::_ → failwith "bad"
  | RequestBody(_)::_ → failwith "bad"
  | ResponseBody(_)::_ → failwith "bad"
  | _::xs → check_harmless xs
  | [] → ()
```

On the other hand, the omission of this check, or an error in its implementation, would be caught as a type error.

7 Related Work

Type systems for information flow have been developed for code written in many languages, including Java [Myers, 1999] and ML [Pottier and Simonet, 2003]. Further works extend them with support for cryptographic mechanisms [for example, Askarov et al., 2006, Vaughan and Zdancewic, 2007, Fournet and Rezk, 2008]. These systems seek to guarantee non-interference properties for programs annotated with confidentiality and integrity levels. In contrast, our system seeks to guarantee assertion-based security properties, commonly used in authorization policies and cryptographic protocol specifications, and disregards implicit flows of information.

Type systems with logical effects, such as ours, have also been used to reason about the security of models of distributed systems. For instance, type systems for variants of the π-calculus [Fournet et al., 2007b, Cirillo et al., 2007] and the λ-calculus [Cirillo et al., 2007] can guarantee that expressions follow their access control policies. Type systems for variants of the π-calculus, such as Cryptyc [Gordon and Jeffrey, 2002], have been used to verify secrecy, authentication, and authorization properties of protocol models. Unlike our tool, none of these typecheckers operates on source code.

The tool CSur has been used to check cryptographic properties of C code using an external first-order-logic theorem-prover [Goubault-Larrecq and Parrennes, 2005]; it does not rely on typing.

Our approach of annotating programs with pre- and postconditions has similarities with extended static checkers used for program verification, such as ESC/Java [Flanagan et al., 2002], Spec# [Barnett et al., 2005], and ESC/Haskell [Xu, 2006]. Such checkers cannot verify security properties of cryptographic code, but they can find many other kinds of errors. For instance, Poll and Schubert [2007] use ESC/Java2 [Cok and Kiniry, 2004] to verify that an SSH implementation in Java conforms to a state machine specification. Combining approaches can be even more effective, for instance, Hubbers et al. [2003] generate implementation code from a verified protocol model and check conformance using an extended static checker.

In comparison with these approaches, we propose subtyping rules that capture notions of public and tainted data, and we provide functional encodings of cryptography. Hence, we achieve typability for opponents representing active attackers. Also, we use only stable formulas: in any given run, a formula that holds at some point also holds for the rest of the run; this enables a simple treatment of programs with concurrency and side-effects. (This would not be the case, say, with predicates on the current state of shared mutable memory.)

One direction for further research is to avoid the need for refinement type annotations, by inference. A potential starting point is a recent paper [Rondon et al., 2008], which presents a polymorphic system of refinement types for ML, quite related to RCF, together with a type inference algorithm based on predicate abstraction.

Acknowledgments Discussions with Bob Harper and Dan Licata were useful. Aleks Nanevski commented on a draft of this paper. Kenneth Knowles suggested a proof technique. Nikolaj Bjørner and Leonardo de Moura provided help with Z3. Sergio Maffeis was supported by EPSRC grant EP/E044956/1.

A Logic

Formally, our typed calculus is parameterized by the choice of an authorization logic, in the sense that it relies only on a series of abstract properties of the logic, rather than on a particular syntax or semantics for logic formulas.

Experimentally, our prototype implementation uses ordinary first order logic with equality, with terms that include all the values M, N of Section 2 (including functional values). During typechecking, this logic is partially mapped to the SIMPLIFY input of Z3, with the implementation restriction that no term should include any functional value. (This restriction prevents discrepancies on term equalities between the calculus and the logic.)

We use the following abstract syntax.

First-Order Logic with Equality:

p	predicate symbol
$C ::=$	formula
$C \wedge C'$	conjunction
$C \vee C'$	disjunction
$\neg C$	negation
$\forall x.C$	universal quantification
$\exists x.C$	existential quantification
$p(M_1,\dots,M_n)$	atomic predicate
$M = M'$	equation

$\mathsf{True} \triangleq \forall x.x = x$
$\mathsf{False} \triangleq \neg\mathsf{True}$
$M \neq M' \triangleq \neg(M = M')$
$(C \Rightarrow C') \triangleq (\neg C \vee C')$
$(C \Leftrightarrow C') \triangleq (C \Rightarrow C') \wedge (C' \Rightarrow C)$

As usual with first order logic, the logical terms may include both variables and function symbols (coded as datatype constructors). In addition, they may include function abstractions $\mathbf{fun}\, x \to A$, considered up to consistent renaming of bound variables. (These functions are inert in the logic; they can be compared but not applied.)

Other interesting logics for our verification purposes include logics with "says" modalities [Abadi et al., 1993], which may be used to give a logical account of principals and partial trust by typing [Fournet et al., 2007b].

B Semantics and Safety of Expressions

This appendix formally defines the operational semantics of expressions, and the notion of expression safety, as introduced in Section 2.

An expression can be thought of as denoting a *structure*, given as follows. We define the meaning of **assume** C and **assert** C in terms of a structure being *statically safe*.

Let an *elementary expression*, e, be any expression apart from a let, restriction, fork, message send, or an assumption.

Structures and Static Safety:

$\prod_{i\in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \ldots \Rsh A_n$
$\mathscr{L} ::= \{\} \mid (\textbf{let}\ x = \mathscr{L}\ \textbf{in}\ B)$

$$\mathbf{S} ::= (\nu a_1)\ldots(\nu a_\ell)\ ((\prod_{i\in 1..m} \textbf{assume}\ C_i) \Rsh (\prod_{j\in 1..n} M_j!N_j) \Rsh (\prod_{k\in 1..o} \mathscr{L}_k\{e_k\}))$$

Let structure $\mathbf{S}$ be *statically safe* if and only if, for all $p \in 1..o$ and C, if $e_p = \textbf{assert}\ C$ then $\{C_1,\ldots,C_m\} \vdash C$.

Heating: $A \Rrightarrow A'$

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$

$A \Rrightarrow A' \Rightarrow \textbf{let}\ x = A\ \textbf{in}\ B \Rrightarrow \textbf{let}\ x = A'\ \textbf{in}\ B$
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$

$() \Rsh A \equiv A$
$M!N \Rrightarrow M!N \Rsh ()$
$\textbf{assume}\ C \Rrightarrow \textbf{assume}\ C \Rsh ()$

$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$
$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$
$a \notin fn(B) \Rightarrow \textbf{let}\ x = (\nu a)A\ \textbf{in}\ B \Rrightarrow (\nu a)\textbf{let}\ x = A\ \textbf{in}\ B$

$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$
$\textbf{let}\ x = (A \Rsh A')\ \textbf{in}\ B \equiv A \Rsh (\textbf{let}\ x = A'\ \textbf{in}\ B)$

Reduction: $A \to A'$

$(\textbf{fun}\, x \to A)\ N \to A\{N/x\}$
$(\textbf{let}\ (x_1,x_2) = (N_1,N_2)\ \textbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\}$
$(\textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B) \to$
$$\begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$$
$$M = N \to \begin{cases} \textbf{true} & \text{if } M = N \\ \textbf{false} & \text{otherwise} \end{cases}$$

$c!M \Rsh c? \to M$
$\textbf{assert}\ C \to ()$
$\textbf{let}\ x = M\ \textbf{in}\ A \to A\{M/x\}$

$A \to A' \Rightarrow \textbf{let}\ x = A\ \textbf{in}\ B \to \textbf{let}\ x = A'\ \textbf{in}\ B$
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$
$A \to A' \Rightarrow (A \Rsh B) \to (A' \Rsh B)$
$A \to A' \Rightarrow (B \Rsh A) \to (B \Rsh A')$

$A \to A'$ if $A \Rrightarrow B, B \to B', B' \Rrightarrow A'$

Expression Safety:

An expression A is *safe* if and only if, for all A' and $\mathbf{S}$, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is statically safe.

C Example Code

We provide the complete interface and implementation code for the final MAC-based authentication protocol of Section 3.

Refinement-Typed Interface

```
module M
open Pi
open Crypto
open Net

type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type (;a:prin,b:prin) content = x:string{ Send(a,b,x) }
type message = (prin ∗ prin ∗ string ∗ hmac) pickled

private val mkContentKey:
  a:prin →b:prin →((;a,b)content) hkey
private val hkDb:
  (prin∗prin, a:prin ∗ b:prin ∗ k:(;a,b) content hkey) Db.t
val genKey: prin →prin →unit
private val getKey: a:
  string →b:string →((;a,b) content) hkey

assume ∀a,b,x. ( Leak(a) ) ⇒Send(a,b,x)
val leak:
  a:prin →b:prin →(unit{ Leak(a) }) ∗ ((;a,b) content) hkey

val addr : (prin ∗ prin ∗ string ∗ hmac, unit) addr
private val check:
  b:prin →message →(a:prin ∗ (;a,b) content)
val server: string →unit

private val make:
  a:prin →b:prin →(;a,b) content →message
val client: prin →prin →string →unit
```

F# Implementation Code

```
module M
open Pi
open Crypto // Crypto Library
open Net // Networking Library

// Simple F# types for principals, events, payloads, and messages:
type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type content = string
type message = (prin * prin * string * hmac) pickled

// Key database:
let hkDb : ((prin*prin),(prin*prin*(content hkey))) Db.t =
  Db.create ()
let mkContentKey (a:prin) (b:prin) : content hkey =
  mkHKey()
let genKey a b =
  let k = mkContentKey a b in
  Db.insert hkDb (a,b) (a,b,k)
let getKey a b =
  let a',b',sk = Db.select hkDb (a,b) in
  if (a',b') = (a,b) then sk else failwith "select failed"

// Key compromise:
let leak a b =
  assume (Leak(a)); ((),getKey a b)
// removed: assume (Leak(b));

// Server code:
let addr : (prin * prin * string * hmac, unit) addr =
  http "http://localhost:7000/pwdmac"
let check b m =
  let a,b',text,h = unpickle m in
    if b = b' then
      let k = getKey a b in
  (a,
   unpickle(hmacsha1Verify k (pickle (text:string)) h))
    else failwith "Not the intended recipient"
let server b =
  let c = listen addr in
  let (a,text) = check b (recv c) in
    assert(Send(a,b,text))

// Client code:
let make a b s =
  pickle (a,b,s,hmacsha1 (getKey a b) (pickle s))
let client a b text =
  assume (Send(a,b,text));
  let c = connect addr in
  send c (make a b text)

// Execute one instance of the protocol:
let _ = genKey "A" "B"
let _ = fork (fun (u:unit) →client "A" "B" "Hello")
let _ = server "B"
```

References

M. Abadi. Access control in a core calculus of dependency. In *International Conference on Functional Programming (ICFP'06)*, 2006.

M. Abadi. Secrecy by typing in security protocols. *JACM*, 46(5): 749–786, Sept. 1999.

M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *JACM*, 52(1):102–146, 2005.

M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Symposium (NDSS'03)*. Internet Society, February 2003.

M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148: 1–70, 1999.

M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM TOPLAS*, 15(4): 706–734, 1993.

A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 353–369. Springer, 2006.

D. Aspinall and A. Compagnoni. Subtyping dependent types. *TCS*, 266(1–2):273–309, 2001.

M. Barnett, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'05*, volume 3362 of *LNCS*, pages 49–69. Springer, January 2005.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. Technical Report MSR–TR–2006–46, Microsoft Research, 2007. See also CSFW'06 and WS-FM'06.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

L. Cardelli. Typechecking dependent types and subtypes. In *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 45–57. Springer, 1986.

A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Do As I SaY! Programmatic access control with explicit identities. In *IEEE Computer Security Foundations Symposium (CSF'07)*, pages 16–30, 2007.

D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS'05*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAC'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *JACM*, 52(3):365–473, 2005.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation, at `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`.

J.-C. Filliâtre. Why: a multi-language multi-prover verification condition generator. `http://why.lri.fr/`, 2003.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335, Jan. 2008.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *ACM TOPLAS*, 29(5), 2007a. In press.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies in distributed systems. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 31–45, 2007b.

T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM, 1991.

A. D. Gordon and A. S. A. Jeffrey. Cryptyc: Cryptographic protocol type checker. At `http://cryptyc.cs.depaul.edu/`, 2002.

A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003a.

A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2003b.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, pages 363–379, 2005.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In R. Findler, editor, *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

C. Gunter. *Semantics of programming language*. MIT Press, 1992.

E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing*, pages 213–226, 2003.

J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999.

A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, Mar. 2004. At `http://www.oasis-open.org/committees/download.php/5941/oasis-200401-wss-soap-message-security-1.0.pdf`.

R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS'07*, pages 164–177, 2007.

F. Pottier and Y. Régis-Gianas. Extended static checking of call-by-value functional programs. Draft, July 2007. URL `http://cristal.inria.fr/~fpottier/publis/pottier-regis-gianas-escfp.ps.gz`.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, 2003.

F. Pottier, C. Skalka, and S. Smith. A systematic approach to access control. In *Programming Languages and Systems (ESOP 2001)*, volume 2028 of *LNCS*, pages 30–45. Springer, 2001.

P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*. ACM, 2008. To appear.

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.

E. Sumii and B. Pierce. A bisimulation for dynamic sealing. *TCS*, 375(1–3):169–192, 2007. Extended abstract at POPL'04.

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

J. A. Vaughan and S. Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206, Washington, DC, USA, 2007.

T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM, 1999.

D. N. Xu. Extended static checking for Haskell. In *ACM SIGPLAN workshop on Haskell (Haskell'06)*, pages 48–59. ACM, 2006.