Refinement types for secure implementations

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Sergio Maffeis

Refinement Types for Secure Implementations

Jesper Bengtson, Uppsala University
and
Karthikeyan Bhargavan, Microsoft Research
and
Cédric Fournet, Microsoft Research
and
Andrew D. Gordon, Microsoft Research
and
Sergio Maffeis, Imperial College London

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.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: Security and Protection; C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs.

General Terms: Security, Verification.

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. [2008b] 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 pro-

gram 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 the development of 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, Régis-Gianas and Pottier, 2008]. 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 π-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, named F7, 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., 2008b], 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 anno-

tations 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 named RCF.

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.

Contents. The paper is organized as follows. Section 2 presents our core language with refinement types, and illustrates it by programming access control policies. Section 3 adds typed support for cryptography, using an encoding based on seals, and illustrates it by implementing MAC-based authentication protocols. Section 4 describes our type system and its main properties. Sections 5 and 6 report on the prototype implementation and our experience with programming protocols with our type discipline. Section 7 discusses related work and Section 8 concludes.

Appendixes provide additional details. Appendix A describes the logic and our usage of Z3. Appendix B defines the semantics and safety of expressions. Appendix C establishes properties of the type system. A technical report [Bengtson et al., 2010] includes additional appendixes showing our typed encodings for formal cryptography, the full code

of an extended example, and also detailed proofs of the development in Appendix C. The typechecker, cryptographic libraries, and all examples described in this paper are available as part of the F7 distribution at `http://research.microsoft.com/F7`

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.)

2.1 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 $S \vdash C$ 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 equality. In our intended models, terms denote closed values of RCF, and equality $M = N$ is interpreted as syntactic identity between values. (Appendix A gives the details.)

Formulas and Deducibility:

C	logical formula
$\{C_1, \ldots, C_n\} \vdash C$	logical deducibility

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. We say a phrase is *closed* to mean it has no free variables (although it may have free names).

Syntax of Values and Expressions:

a, b, c	name
x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of recursive type
$M, N ::=$	value

	x	variable
	$()$	unit
	fun $x \to A$	function (scope of x is A)
	(M,N)	pair
	$h\ M$	construction
$A,B ::=$		expression
	M	value
	$M\ N$	application
	$M = N$	syntactic equality
	let $x = A$ **in** B	let (scope of x is B)
	let $(x,y) = M$ **in** A	pair split (scope of x, y is A)
	match M **with**	constructor match
	$h\ x \to A$ **else** B	(scope of x is A)
	$(\nu a)A$	restriction (scope of a is A)
	$A \Rsh B$	fork
	$a!M$	transmission of M on channel a
	$a?$	receive message off channel
	assume C	assumption of formula C
	assert C	assertion of formula C

To evaluate M, return M at once. To evaluate $M\ N$, if $M = \textbf{fun}\, x \to A$, evaluate $A\{N/x\}$. To evaluate $M = N$, if the two values M and N are the same, return **true** $\triangleq$ inr$()$; otherwise, return **false** $\triangleq$ inl$()$. To evaluate **let** $x = A$ **in** B, first evaluate A; if evaluation returns a value M, evaluate $B\{M/x\}$. To evaluate **let** $(x_1,x_2) = M$ **in** A, if $M = (N_1,N_2)$, evaluate $A\{N_1/x_1\}\{N_2/x_2\}$. To evaluate **match** M **with** $h\ x \to A$ **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 $a!M$, emit message M on channel a, and return $()$ at once. To evaluate $a?$, block until some message N is on channel a, 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 $S \vdash C$, 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:

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

The only way for an expression to be unsafe is for an evaluation to lead to an **assert** C, where C does not follow from the current log of assumed formulas. Hence, an expression may fail in other ways while being safe according to this definition. For example, the restriction $(\nu a)a?$ is safe, although it *deadlocks* in the sense no message can be sent on the fresh channel a and so $a?$ blocks forever. The application $()\ (\textbf{fun}\, x \to x)$ is safe, but illustrates another sort of failure: it tries to use $()$ as a function, and so is *stuck* in the sense that evaluation cannot proceed.

Assertions and assumptions are annotations for expressing correctness properties. There is no mechanism in RCF to branch based on whether or not a formula is derivable from

the current log. Our intention is to verify safety statically. If we know statically that an expression is safe, there is no reason to implement the log of assumed expressions because every assertion is known to succeed.

2.2 Types and Subtyping

We outline the type system; the main purpose for typechecking an expression is to establish its safety. We assume a collection of *type variables*, ranged over by α, β. For any phrase ϕ, the set $fnfv(\phi)$ includes the type variables, as well as the names and (value) variables, that occur free in ϕ. Notice that no types or type variables occur in the syntax of values or expressions. If ϕ is a phrase of syntax, we write $\phi\{T/\alpha\}$ for the outcome of substituting a type T for each free occurrence of the type variable α in ϕ.

Syntax of Types:

$H,T,U,V ::=$	type	
	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)
	α	type variable
	$\{x:T \mid C\}$	refinement type (scope of x is C)
$\{C\} \triangleq \{\_:\mathsf{unit} \mid C\}$		ok-type
$\mathsf{bool} \triangleq \mathsf{unit}+\mathsf{unit}$		Boolean 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 type variable is a placeholder for a type, such as a recursive type. 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.

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

THEOREM 1 (SAFETY). *If $\varnothing \vdash A : T$ then A is safe.*

PROOF. See Appendix C. □

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

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

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 $E \vdash \mathsf{Q}(y_1)$ via the rule above plus first-order logic.

The introduction rule for refinement types is as follows.

—If $E \vdash M : T$ and $E \vdash C\{M/x\}$ 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 $E \vdash C$.
—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 $C \vdash C'$ 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 $E \vdash C$ 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.

For example, we can derive $A_{ex} : \mathsf{unit}$ where A_{ex} is the following, where Foo and Bar are nullary predicate symbols.

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{assume}\ \mathsf{Foo}() \Rightarrow \mathsf{Bar}()\ \mathbf{in} \\ \mathbf{let}\ y = \mathbf{assume}\ \mathsf{Foo}()\ \mathbf{in}\ \mathbf{assert}\ \mathsf{Bar}()\end{array}$$

By the rule for assumptions we have:

$$\begin{array}{l}\mathbf{assume}\ \mathsf{Foo}() \Rightarrow \mathsf{Bar}() : \{\mathsf{Foo}() \Rightarrow \mathsf{Bar}()\} \\ \mathbf{assume}\ \mathsf{Foo}() : \{\mathsf{Foo}()\}\end{array}$$

The rule for checking a let-expression is:

—If $E \vdash A : T$ and $E, x : T \vdash B : U$ then $E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U$.

By this rule, to show $A_{ex} : \mathsf{unit}$ it suffices to check

$$E \vdash \mathbf{assert}\ \mathsf{Bar}() : \mathsf{unit}$$

where $E = x : \{\mathsf{Foo}() \Rightarrow \mathsf{Bar}()\}, y : \{\mathsf{Bar}()\}$. We have $E \vdash \mathsf{Bar}()$ since both $E \vdash \mathsf{Foo}() \Rightarrow \mathsf{Bar}()$ and $E \vdash \mathsf{Foo}()$. Hence, by the rule for assertions we have:

$$E \vdash \mathbf{assert}\ \mathsf{Bar}() : \{\mathsf{Bar}()\}$$

In general, $\{C\} <: \mathsf{unit}$, so by subsumption: $E \vdash \mathbf{assert}\ \mathsf{Bar}() : \mathsf{unit}$ and thus A_{ex} is safe.

2.3 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 **let** $x = A$ **in** B is the only construct to allow sequential evaluation of expressions. As usual, $A;B$ is short for **let** $\_ = A$ **in** B, and **let** $f\,x = A$ is short for **let** $f =$ **fun** $x \to A$. More notably, if A and B are proper expressions rather than being values, the application $A\ B$ is short for **let** $f = A$ **in** (**let** $x = B$ **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 ($(\nu a)A$, $A \Rsh B$, $a?$, and $a!M$) 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:

$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$

$\mathsf{chan} \triangleq \mathbf{fun}\, x \to (\nu a)(\mathbf{fun}\, x \to a!x, \mathbf{fun}\, \_ \to a?)$

$\mathsf{send} \triangleq \mathbf{fun}\, c\ x \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ s\ x$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\, c \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\, f \to (f() \Rsh ())$	run f in parallel

We define references in terms of channels.

Functions for References:

$(T)\mathbf{ref} \triangleq (T)\mathsf{chan}$

$\mathbf{ref}\ M \triangleq \mathbf{let}\ r = \mathsf{chan}\ \texttt{"r"}\ \mathbf{in}\ \mathsf{send}\ r\ M; r$	new reference to M
$!M \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathsf{recv}\ M; \mathsf{send}\ M\ N$	update M with N

We also assume standard encodings of strings, numeric types, Booleans, tuples, records, algebraic types (including lists) and pattern-matching, and recursive functions. (An appendix to the technical reports lists the full details.) 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 **val** $x_1 : T_1$ ... **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 **let** $x_1 = A_1$ ... **let** $x_n = A_n$ of *value definitions*, which we interpret by the expression **let** $x_1 = A_1$ **in** ... **let** $x_n = A_n$ **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 **assume** C is treated as a declaration **val** $x : \{C\}$ plus a definition **let** $x =$ **assume** C for some fresh x.

2.4 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. (See, for example, Dean et al. [1996], Pottier et al. [2001], Abadi and Fournet [2003], and Abadi [2007] for a more general discussion of security mechanisms for partially-trusted code.)

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
```

For instance, the fact CanRead("C:/README") represents read access to "C:/README". 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(M) for its string argument M (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.

More operationally, to illustrate our formal definition of expression safety, we may include assertions, and define

```
let read file = assert(CanRead(file)); "data"
let delete file = assert(CanWrite(file))
```

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"
```

The F# library function failwith throws an exception, so it never returns and can safely be given the polymorphic type string $\rightarrow \alpha$, 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 returns two type errors:

```
let test:unit =
  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:unit =
  let reader: unit →string =
```

```
    (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.

We finally illustrate the use of refinement-typed values within imperative data structures to "store" valid formulas. We may declare an access control list (ACL) database as

```
type entry =
  | Readable of x:string{CanRead(x)}
  | Writable of x:string{CanWrite(x)}
  | Nothing
val acls : (string,entry) Db.t
val safe_read: string → string
val readable: file:string → unit{ CanRead(file) }
```

(where Db.t is a type constructor for our simplified typed database library, parameterized by the types of the keys and entries stored in the database) and implement it as:

```
let acls: (string,entry) Db.t = Db.create()
let safe_read file =
  match Db.select acls file with
  | Readable file → read file
  | Writable file → read file
  | _ → failwith "unreadable"
let readable file =
  match Db.select acls file with
  | Readable f when f = file → ()
  | Writable f when f = file → ()
  | _ → failwith "unreadable"
```

Both safe_read and readable lookup an ACL entry and, by matching, either "retrieve" a fact sufficient for reading the file, or fail. The code below illustrates their usage:

```
let test_acls:unit =
  Db.insert acls tmp (Writable(tmp)); // ok
// Db.insert acls tmp (Readable(pwd)); // type error
  Db.insert acls pwd (Nothing); // ok
  let v6 = safe_read pwd in // ok (but dynamically fails)
  let v7 = readable tmp; read tmp in () // ok
```

3. MODELLING CRYPTOGRAPHIC PROTOCOLS

We introduce our technique for specifying security properties of cryptographic protocols by typing.

3.1 Roles and Opponents as Functions

Following Bhargavan et al. [2008b], 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

A closed expression O is an *opponent* iff O contains no occurrence of **assert**.
A closed 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.)

3.2 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, $(\Pi x : \mathsf{Un}.\ \mathsf{Un})$, $(\Sigma x : \mathsf{Un}.\ \mathsf{Un})$, $(\mathsf{Un} + \mathsf{Un})$, and $(\mu\alpha.\mathsf{Un})$. 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 $E, x : T \vdash C$

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). *If* $\varnothing \vdash A : \mathsf{Un}$ *then* A *is robustly safe.*

PROOF. See Appendix C. □

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.

3.3 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){p=P(x)}
val unpickle: p:α pickled → (x:α){p=P(x)}
```

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 message authentication codes (MACs). This interface declares

```
type α hkey = HK of (α pickled,Un) Table
type hmac = HMAC of Un
type α hmacpred = IsHMAC of α hkey * α pickled * hmac
assume (∀k,x,h. IsHMAC(k,x,h) ⇔
    (∃n,l,rl,hm. k = HK((n,l,rl)) ∧ h = HMAC(hm) ∧ Table(n,x,hm)))
val mkHKey: unit → α hkey
private val hmacsha1: k:α hkey → x:α pickled → h:hmac{IsHMAC(k,x,h)}
private val hmacsha1Verify: k:α hkey → xx:Un → h:hmac → x:α pickled{IsHMAC(k,x,h) ∧ x =
    xx}
```

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 pickled value with matching types. The function hmacsha1Verify verifies whether the joint hash of a key and a value (presumed to be the pickled representation of some value of type α) matches 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.

3.4 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., 2008b].

We present snippets of the protocol code to illustrate our typechecking method; an appendix to the technical report 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 **private** may occur only in typechecked implementations. Conversely, the other values (addr, client, server) must have public types, and may be made available to the opponent.

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
  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 **private** 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. Other such mistakes include using the authentication key to hash a plain string, and rebinding text to any other value between the **assume** and the actual MAC computation.

3.5 Example: Logs and Queries

We now relate our present approach to more traditional correspondence properties, stated in terms of run time events. To this end, we explicitly code calls to a secure log function that exclusively records begin- and end-events, and we formulate our security property on the series of calls to this function.

Continuing with our MAC example protocol, we modify the interface as follows:

```
type event = Send of string | Recv of string
private val log : e:event{ ∀x. (e = Recv(x) ⇒ Send(x)) } →
  r:unit{ ∀x. (e = Send(x) ⇒ Send(x)) }
```

The intended correspondence property Recv(x)⇒ Send(x) can now be read off the declared type of log. (In this type, Send and Recv are used both as $F^{\#}$ datatype constructors and predicate constructors.)

We also slightly modify the implementation, as follows:

```
let log x = match x with
  | Send text →assume (Send(text))
  | Recv text →assert(Send(text))

let client text =
  log (Send(text)); // we log instead of assuming
  let c = connect addr in
  send c (make hk text)

let server () =
  let c = listen addr in
  let text = check hk (recv c) in
  log (Recv text) // we log instead of asserting
```

The main difference is that **assume** is relegated to the implementation of log; we also omit the redundant **assert** in server code, as the condition follows from the type of both check and log. As a corollary of type soundness, we obtain that, for all runs, every call to log with a Recv event is preceded by a call to log with a matching Send event (by induction on the series of calls to log).

3.6 Example: Principals and Compromise

We now extend our example to multiple principals, with keys shared 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 an appendix to the technical report; here we give only the types.

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

```
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 hkey 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 shared between a and b 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 the event 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. Hence, whenever a message is accepted, either this message has been sent (with matching sender, receiver, and content), or a key for its apparent sender has been leaked.

3.7 Discussion: Modelling Secrecy

Although this paper focuses on authentication and authorization properties, our type system also guarantees secrecy properties. Without key secrecy, for instance, we would not be able to obtain authenticity by typing for the protocol examples given above.

In a well-typed program, the opponent is given access only to a public interface, so any value passed to the opponent must first be given a public type. On the other hand, the local type of the value does not yield in itself any guarantee of secrecy, since the same value may be given a public type in another environment, under stronger logical assumptions.

Informally, the logical formulas embedded in a type indicate the conditions that must hold before values of that type are considered public.

To give a more explicit account of secrecy, we consider a standard "no escape" property that deems a value secret as long as no opponent can gain direct access to the value. (This form of secrecy is adequate for some values; it is weaker than equivalence-based forms of secrecy that further exclude any implicit flow of information from the actual value of a secret to the opponent.)

Robust Secrecy:

Let A be an expression with free variable s. The expression A *preserves the secrecy of* s *unless* C iff the expression **let** $s = (\textbf{fun}\ \_ \rightarrow \textbf{assert}\ C)$ **in** A is robustly safe.

This definition does not rely on types; instead, it tests whether the opponent may gain knowledge of s: then, the opponent may also call the function, thereby triggering the guarded assertion **assert** C. By definition of robust safety, the formula C must then follow from the assumptions recorded in the log.

As a simple corollary of Theorem 2 (Robust Safety), we establish a principle of robust secrecy by typing.

THEOREM 3 (ROBUST SECRECY). *If* $s : \{C\} \rightarrow \mathit{unit} \vdash A : \mathit{Un}$, *then* A *preserves the secrecy of* s *unless* C.

PROOF. (In this proof, we anticipate the typing rules of Section 4.) By hypothesis, $s : \{C\} \rightarrow \mathsf{unit} \vdash A : \mathsf{Un}$, hence $\varnothing \vdash C$, and thus $\{C\} \vdash \textbf{assert}\ C : \mathsf{unit}$ by (Exp Assert), $\varnothing \vdash (\textbf{fun}\ \_ \rightarrow \textbf{assert}\ C) : \{C\} \rightarrow \mathsf{unit}$ by (Val Fun), and $\varnothing \vdash \textbf{let}\ s = (\textbf{fun}\ \_ \rightarrow \textbf{assert}\ C)\ \textbf{in}\ A : \mathsf{Un}$ by (Exp Let). We conclude by Theorem 2 (Robust Safety). □

By inspection of the rules for public kinding, we see that the type $\{C\} \rightarrow \mathsf{unit}$ given to s is public only in environments that entail C, and thus is indeed a type of secrets "unless C holds".

We illustrate secrecy on a two-message protocol example, relying on authenticated, symmetric encryptions instead of MACs. The first messages is a session key (k) encrypted under a long-term key; the second message is a secret payload (s) encrypted under the session key. Secrecy is stated unless Leak(a), a fact used below to illustrate the usage of assumptions for modelling key compromise.

We use the following declarations.

```
type empty = u:unit { Leak(a) }
type secret = empty →unit
type payload = secret

private val s: payload
private val k0: (payload symkey) symkey

// The protocol uses a fresh session key
// and relies on its authenticated encryption
// client →server : { fresh k }k0
// server →client : { s }k

val addr : (enc, enc) addr
val client: unit →unit
```

```
val server: unit →unit
```

Both s (the payload) and k0 (the long-term key) must be declared as private values; otherwise we obtain kinding errors.

We give a definition only for the test secret—the rest of the protocol definitions are similar to those listed above.

```
let s () = assert(Leak(a)) // our test secret
```

We obtain an instance of Theorem 3 (Robust Secrecy) for the expression A that consists of library code plus the protocol code (without the definition of s). As we typecheck the protocol definitions, we would obtain typing errors, for instance, if the client code attempted to leak k0, k, or s on a public channel, or if the server code attempted to encrypt s under a public key instead of k.

We can model the compromise of the client machine by releasing k0 (its only initial secret) to the opponent. The code used to model this situation is typable only with sufficient assumptions: we may for instance define a public function `let leak()= assume(Leak(a)); k0`, with an assumption that records the potential loss of secrecy for s.

In a refined example with multiple clients, each with its own long-term key, we may use a more precise secrecy condition, such as $C = \exists a.(\,\mathsf{Leak}(a) \wedge \mathsf{Accept}(a))$ where Leak(a) records the compromise of a principal named a and Accept(a) records that the server actually accepted to run a session with a as client. Thus, for instance, we may be able to check the secrecy of s despite the compromise of unauthorized clients.

We refer to Gordon and Jeffrey [2005] and Fournet et al. [2007b] for a more general account of secrecy and authorization despite compromise.

3.8 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 $T \to \mathsf{Un}$, and the *unseal function for* k, of type $\mathsf{Un} \to T$. 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: string →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 reference s with the seal k, and store the current list in 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 reference s that holds an empty list; the seal itself is the pair of functions (seal s, unseal s). The code uses the abbreviations **ref**, !, and := displayed in Section 2.

The code also relies on library functions for list lookups:

```
let rec first f xs = match xs with
  | x::xs →(let r = f x in match r with
                              Some(y) → r
                            | None → first f xs)
  | [] → None
let left z (x,y) = if z = x then Some y else None
let right z (x,y) = if z = y then Some x else None
```

The function first, of type $(\alpha \to \beta\ \text{option}) \to \alpha\ \text{list} \to \beta\ \text{option}$, takes as parameters a function and a list; it applies the function to the elements of the list, and returns the 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 SealRef = ((α * Un) list) ref
let seal: α SealRef → α → Un = fun s m →
  let state = !s in match first (left m) state with
  | Some(a) → a
  | None →
    let a: Un = Pi.name "a" in
    s := ((m,a)::state); a
let unseal: α SealRef → Un → α = fun s a →
  let state = !s in match first (right a) state with
  | Some(m) → m
  | None → failwith "not a sealed value"
let mkSeal (n:string) : α Seal =
  let s = ref []:α SealRef in
    (seal s, unseal s)
```

Irrespective of the type α for M, sealing returns a public name a, which may be communicated on some unprotected network, and possibly passed to the opponent.

In a variant of seal, we always generate a fresh value a, rather than perform a list lookup; this provides support for non-deterministic encryption and signing (with different, unrelated values for different encryptions of the same value).

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 π-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.

An appendix to the technical report includes listings for the interface and the (typed) symbolic implementation of cryptography. 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 =
  let t = table "hkey" fail (fun () → Pi.name "hkey") in
    HK t
let hmacsha1 (HK(key)) text =
    let (n,h,_) = key in
```

```
    HMAC (h text)
let hmacsha1Verify (HK key) text (HMAC h) =
  let (n,_,hv) = key in
  let x:α pickled = hv 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).

Our abstract functions for defining cryptographic primitives can be seen as symbolic counterparts to the *oracle functions* commonly used in cryptographic definitions of security [see, for instance, Bellare and Rogaway, 1993]. For example, in a random-oracle model for keyed hash functions, an oracle function would take an input to be hashed, perform a table lookup of previously-hashed inputs, and either return the previous hash value, or generate (and record) a fresh hash value. The main difference is that we rely on symbolic name generation, whereas the oracle relies on probabilistic sampling.

4. A TYPE SYSTEM FOR ROBUST SAFETY

The type system consists of a set of inductively defined judgments. Each is defined relative to a *typing environment*, E, which defines the variables and names in scope.

Judgments:

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

Syntax of Kinds:

$\nu ::= \mathbf{pub} \mid \mathbf{tnt}$	kind (public or tainted)

Let $\overline{\nu}$ satisfy $\overline{\mathbf{pub}} = \mathbf{tnt}$ and $\overline{\mathbf{tnt}} = \mathbf{pub}$.

Syntax of Typing Environments:

$\mu ::=$	environment entry
α	type variable
$\alpha :: \nu$	kinding for recursive type α
$\alpha <: \alpha'$	subtyping for recursive types $\alpha \neq \alpha'$
$a \updownarrow T$	channel name
$x : T$	variable
$E ::= \mu_1, \ldots, \mu_n$	environment

$dom(\alpha) = \{\alpha\}$
$dom(\alpha :: \nu) = \{\alpha\}$
$dom(\alpha <: \alpha') = \{\alpha, \alpha'\}$
$dom(a \updownarrow T) = \{a\}$
$dom(x : T) = \{x\}$
$dom(\mu_1, \ldots, \mu_n) = dom(\mu_1) \cup \cdots \cup dom(\mu_n)$

$$recvar(E) = \{\alpha, \alpha' \mid (\alpha <: \alpha') \in E\} \cup \{\alpha \mid (\alpha :: \nu) \in E\}$$

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 $recvar(E)$ be just the type variables occurring in kinding and subtyping entries of E. Let E be *executable* if and only if $recvar(E) = \varnothing$. Such environments contain names, variables, and type variables (but no entries $\alpha :: \nu$ or $\alpha <: \alpha'$). Let $fnfv(E) = \bigcup\{fnfv(T) \mid (a \updownarrow T) \in E \vee (x : T) \in E\}$.

Rules of Well-Formedness and Deduction:

$$\frac{}{\varnothing \vdash \diamond}\ (\text{Env Empty}) \qquad \frac{E \vdash \diamond \quad fnfv(\mu) \subseteq dom(E) \quad dom(\mu) \cap dom(E) = \varnothing}{E, \mu \vdash \diamond}\ (\text{Env Entry})$$

$$\frac{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)}{E \vdash T}\ (\text{Type}) \qquad \frac{E \vdash \diamond \quad fnfv(C) \subseteq dom(E) \quad \mathsf{forms}(E) \vdash C}{E \vdash C}\ (\text{Derive})$$

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

The function $\mathsf{forms}(E)$ maps an environment E to a set of formulas $\{C_1, \ldots, C_n\}$. We occasionally use this set in a context expecting a formula, in which case it should be interpreted as the conjunction $C_1 \wedge \cdots \wedge C_n$, or True in case $n = 0$. For example:

$$\begin{aligned} \mathsf{forms}(x : \{C\}) &= \mathsf{forms}(x : \{y : \mathsf{unit} \mid C\}) \quad y \notin fv(C) \\ &= \{C\{x/y\}\} \cup \mathsf{forms}(x : \mathsf{unit}) \\ &= \{C\} \end{aligned}$$

Observe also that $\mathsf{forms}(E) = \varnothing$ if E contains only names; formulas are derived only from the types of variables, not from the types of channel names.

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** $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$

$$\frac{E \vdash \diamond \quad (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu}\ (\text{Kind Var}) \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: \nu}\ (\text{Kind Unit}) \qquad \frac{E \vdash T :: \overline{\nu} \quad E, x : T \vdash U :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu}\ (\text{Kind Fun})$$

$$\frac{E \vdash T :: \nu \quad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu}\ (\text{Kind Pair}) \qquad \frac{E \vdash T :: \nu \quad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}\ (\text{Kind Sum}) \qquad \frac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu\alpha.T) :: \nu}\ (\text{Kind Rec})$$

$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}}\ (\text{Kind Refine Public}) \qquad \frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}\ (\text{Kind Refine Tainted})$$

The following rules for ok-types are derivable.

$$\frac{E \vdash \{C\}}{E \vdash \{C\} :: \mathbf{pub}}\ (\text{Kind Ok Public}) \qquad \frac{E \vdash \{C\} \quad E \vdash C}{E \vdash \{C\} :: \mathbf{tnt}}\ (\text{Kind Ok Tainted})$$

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: $E \vdash T <: U$

(Sub Refl)
$$\frac{E \vdash T \quad \mathit{recvar}(E) \cap \mathit{fnfv}(T) = \varnothing}{E \vdash T <: T}$$

(Sub Public Tainted)
$$\frac{E \vdash T :: \mathbf{pub} \quad E \vdash U :: \mathbf{tnt}}{E \vdash T <: U}$$

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

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

(Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

(Sub Var)
$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(Sub Rec)
$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathit{fnfv}(T') \quad \alpha' \notin \mathit{fnfv}(T)}{E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')}$$

(Sub Refine Left)
$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$$

(Sub Refine Right)
$$\frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

The universal type Un is type equivalent to all types that are both public and tainted; we (arbitrarily) define $\mathsf{Un} \triangleq \mathsf{unit}$. We can show that this definition satisfies the intended meaning: that T is public if and only if T is a subtype of Un, and that T is tainted if and only if T is a supertype of Un. (See Lemma 15 (Public Tainted) in Appendix C.)

The following congruence rule for refinement types is derivable from the two primitive rules for refinement types (Sub Refine Left) and (Sub Refine Right). We also list the special case for ok-types.

(Sub Refine)
$$\frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \vdash C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}}$$

(Sub Ok)
$$\frac{E, \_ : \{C\} \vdash C'}{E \vdash \{C\} <: \{C'\}}$$

PROOF. To derive (Sub Refine), we are to show that $E \vdash T <: T'$ and $E, x : \{x : T \mid C\} \vdash C'$ imply $E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}$. By Lemma 2 (Derived Judgments) in Appendix C, $E, x : \{x : T \mid C\} \vdash C'$ implies $E \vdash \{x : T \mid C\}$. By (Sub Refine Left), $E \vdash \{x : T \mid C\}$ and $E \vdash T <: T'$ imply $E \vdash \{x : T \mid C\} <: T'$. By (Sub Refine Right), this and $E, x : \{x : T \mid C\} \vdash C'$ imply $E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}$. □

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: $E \vdash M : T$

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T}\ \text{(Val Var)} \qquad \frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}\ \text{(Val Unit)} \qquad \frac{E, x:T \vdash A : U}{E \vdash \mathbf{fun}\, x \to A : (\Pi x:T.\, U)}\ \text{(Val Fun)}$$

$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M,N) : (\Sigma x:T.\, U)}\ \text{(Val Pair)} \qquad \frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x:T \mid C\}}\ \text{(Val Refine)}$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\, M : U}\ \text{(Val Inl Inr Fold)}$$

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

We can derive an introduction rule for ok-types.

$$\frac{E \vdash C}{E \vdash () : \{C\}}\ \text{(Val Ok)}$$

PROOF. From $E \vdash C$ we know that $E \vdash \diamond$ and that $E \vdash C\{()/x\}$. By (Val Unit), $E \vdash () : \mathsf{unit}$. By (Val Refine), $E \vdash () : \{x : \mathsf{unit} \mid C\}$, that is, $E \vdash () : \{C\}$. □

Our final set of rules is for typing arbitrary expressions.

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

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

$$\frac{\begin{array}{l} E \vdash M : (\Sigma x:T.\, U) \\ E, x:T, y:U, \_ : \{(x,y) = M\} \vdash A : V \\ \{x,y\} \cap fv(V) = \varnothing \end{array}}{E \vdash \mathbf{let}\ (x,y) = M\ \mathbf{in}\ A : V}\ \text{(Exp Split)}$$

$$\frac{\begin{array}{l} E \vdash M : T \quad h : (H,T) \\ E, x:H, \_ : \{h\, x = M\} \vdash A : U \\ E, \_ : \{\forall x.h\, x \neq M\} \vdash B : U \end{array}}{E \vdash \mathbf{match}\ M\ \mathbf{with}\ h\, x \to A\ \mathbf{else}\ B : U}\ \text{(Exp Match Inl Inr Fold)}$$

$$\frac{E \vdash M : T \quad E \vdash N : U \quad x \notin fv(M,N)}{E \vdash M = N : \{x : \mathsf{bool} \mid (x = \mathbf{true} \wedge M = N) \vee \quad (x = \mathbf{false} \wedge M \neq N)\}}\ \text{(Exp Eq)}$$

$$\frac{E \vdash \diamond \quad fnfv(C) \subseteq dom(E)}{E \vdash \mathbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}\ \text{(Exp Assume)} \qquad \frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathsf{unit}}\ \text{(Exp Assert)}$$

$$\frac{E \vdash A : T \quad E, x:T \vdash B : U \quad x \notin fv(U)}{E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U}\ \text{(Exp Let)} \qquad \frac{E, a \updownarrow T \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a)A : U}\ \text{(Exp Res)}$$

$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}}\ \text{(Exp Send)} \qquad \frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}\ \text{(Exp Recv)}$$

$$\frac{\begin{array}{l} E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \\ E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2 \end{array}}{E \vdash (A_1 \Rsh A_2) : T_2}\ \text{(Exp Fork)}$$

In rules for pattern-matching pairs and constructors, we use equations and inequations within refinement types to track information about the matched variables: (Exp Split) records that M is the pair (x,y); (Exp Match Inl Inr Fold) records that M is $h\ x$ when A runs and that M is not of that form when B runs. Rule (Exp Eq) similarly tracks the result of equality tests.

The final rule, (Exp Fork) 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: $\overline{A}$

$\overline{(\nu a)A} = \exists a.\overline{A}$ $\quad$ $\overline{A_1 \Rsh A_2} = \overline{A_1} \wedge \overline{A_2}$ $\quad$ $\overline{\textbf{let } x = A_1 \textbf{ in } A_2} = \overline{A_1}$ $\quad$ $\overline{\textbf{assume } C} = C$

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

5. IMPLEMENTING REFINEMENT TYPES FOR F#

We implement a typechecker, known as F7, that takes as input a series of extended RCF interface files and F# implementation files and, for every implementation file, performs the following tasks: (1) typecheck the implementation against its RCF interface, and any other RCF interfaces it may use; (2) kindcheck 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.

For simplicity, we do not provide syntactic support for extended types or non-atomic formulas in implementation files. To circumvent this limitation, one can always move extended types and complex formulas to the RCF interface by adding auxiliary declarations.

5.1 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. Hence, for example, our typechecker eliminates type abbreviations by inlining, and compiles records to tuples. The remaining constructs constitute straightforward generalizations of our core calculus. For example, polymorphic functions represent a family of functions, one for each instance of a type variable; hence, when checking a specific function application, our typechecker uses the argument type and expected result type to first instantiate the function type and then typecheck it.

5.2 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

	F# code (lines)	F# decl. (lines)	F7 decl. (lines)	Time (S)	Z3 proofs (goals)
Typed Libraries	440	125	146	12.1	12
Access Control (Section 2.4)	104	16	34	8.3	16
MAC Protocol (Section 3.4)	40	9	12	2.5	3
Logs and Queries (Section 3.5)	37	10	16	2.8	6
Secrecy (Section 3.7)	51	18	41	2.7	6
Principals & Compromise (Section 3.6)	48	13	26	3.1	12
Flexible Signatures (Section 6)	167	25	52	14.6	28

Table I. Typechecking Example Programs

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 declares 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.

5.3 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"
```

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.

5.4 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.

5.5 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.

5.6 Evaluation

We have typechecked all the examples of this paper and a few larger programs. Table I 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 second-last row is an example adapted from earlier work [Fournet et al., 2007a]; it illustrates the recursive verification of any chain of certificates. 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. Recent experiments [Bhargavan et al., 2009] indicate that our typechecker scales well to large examples; it can verify custom cryptographic protocol code with around 2000 lines of F# in less than 3 minutes. In comparison with an earlier tool FS2PV [Bhargavan et al., 2008b] that compiles F# code to ProVerif, our typechecker succeeds on examples with recursive functions, such as the last row in Table I, where ProVerif fails to terminate. It also scales better, 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 format (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 are 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 k: (verified,unit) privkey
private val sk: verified sigkey
val vk: verified verifkey
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) =
  let signed: siginfo → siginfo pickled = pickle in
  unpickle (rsasha1Verify vk (signed msg) dsig)
let isRequest msg =
  let si = isMessage msg in
  let i = find_id si in
  let r = find_request si in
  (i,r)
```

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

```
let mkMessage siginfo =
  let signed: verified → verified pickled = pickle in
  (siginfo, rsasha1 sk (signed siginfo))
let mkPlainRequest (id,payload) =
  let l1: element list = [] in
  let ide: element = IdHdr(id) in
  let reqe : element = RequestBody(payload) in
  let ls:element list = ide::reqe::l1 in
  mkMessage ls

let mkRequest (id,payload) extra : msg =
```

```
  check_harmless extra;
  let ide: element = IdHdr(id) in
  let reqe : element = RequestBody(payload) in
  let ls:element list = ide::reqe::extra in
  mkMessage ls
```

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.

To conclude this example, we provide an alternative declaration for type verified. This type specifies a more restrictive interpretation if signatures: it assumes that the relevant elements appear in a fixed order at the head of the list. (This corresponds roughly to our most precise model in earlier work, which relied on an ad hoc specification of list within ProVerif.)

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

Formally, our typechecker confirms that verified is a subtype of prefixverified. For instance, we may use it instead of verified for typing mkRequest (and even remove the call to check_harmless), but not for typing isRequest.

7. RELATED WORK

RCF is intended for verifying security properties of implementation code, and is related to various prior type systems and static analyses. We describe some of the more closely related approaches. (See also Section 1 for a comparison with prior work of the authors.)

Verification tools for cryptographic protocol implementations. CSur was the first tool to analyze the source code of cryptographic protocols [Goubault-Larrecq and Parrennes,

2005]; it can verify protocol code in C annotated with logical assertions, by generating proof obligations for an external first-order-logic theorem-prover.

In prior work [Bhargavan et al., 2008b,a] a subset of F# was translated into different variants of the applied π-calculus which could be verified by Blanchet's theorem provers ProVerif [Blanchet, 2001] and CryptoVerif [Blanchet, 2005] respectively. The use of specialized provers enables the verification of complex cryptographic protocols but is problematic with large implementations.

ASPIER [Chaki and Datta, 2009] has been applied to verify code of the central loop of OpenSSL. It performs no interprocedural analysis and relies on unverified user-supplied abstractions of all functions called from the central loop. ASPIER is based on software model-checking techniques, and proves properties of OpenSSL assuming bounded numbers of active sessions.

Program verification using dependent types. Like standard forms of constructive type theory [Martin-Löf, 1984, Constable et al., 1986, Coquand and Huet, 1988, Parent, 1995], our system RCF relies on dependent types (that is, types which contain values), and it can establish logical properties by typechecking. There are, however, three significant differences in style between RCF and constructive type theory. Most notably, RCF does not rely on the Curry-Howard correspondence, which identifies types with logical formulas; instead, RCF has a fixed set of type constructors, and is parameterized by the choice of an authorization logic, which may or may not be constructive. Secondly, types in RCF may contain only values, but not arbitrary expressions, such as function applications. Thirdly, properties of functions are stated by refining their argument and result types with preconditions and postconditions, rather than by developing a behavioural equivalence on functions.

Our treatment of refinement types follows Sage [Gronski et al., 2006], a functional programming language with a rich type system including refinement types. Typechecking generates proof obligations that are sent to an automatic theorem prover; those that cannot be proved automatically are compiled down to run time checks.

Our approach of annotating programs with pre- and post-conditions 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 have not been used to 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 recent work, Régis-Gianas and Pottier [2008] enrich a core functional programming language with higher order logic proof obligations. These are then discharged either by an automatic or an interactive theorem prover depending on the complexity of the proof.

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. More precise stateful properties can still be specified and verified within RCF using a refined state monad [Borgström et al., 2009].

One direction for further research is to avoid the need for refinement type annotations, by inference. A potential starting point is a recent line of work based on Liquid Types [Rondon et al., 2008, Kawaguchi et al., 2009, Rondon et al., 2010], a polymorphic system of refinement types for ML, together with a type inference algorithm based on predicate abstraction.

Type systems for security. Type systems for information flow have been developed for code written in many languages, including Java [Myers, 1999], ML [Pottier and Simonet, 2003], and Haskell [Li and Zdancewic, 2006]. Further works extend them with support for cryptographic mechanisms [for example, Askarov and Sabelfeld, 2005, 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.

These systems also feature various privileged primitives for declassifying confidential information and endorsing untrusted information, which play a role similar to our **assume** primitive for injecting formulas.

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, Maffeis et al., 2008] and the λ-calculus [Jagadeesan et al., 2008] 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 AURA type system [Vaughan et al., 2008, Jia et al., 2008] also enforces authorization by relying on value-dependent types, but it takes advantage of the Curry-Howard isomorphism for a particular intuitionistic logic [Abadi, 2007]; hence, proofs are manipulated at run time, and may be stored for later auditing; in contrast, we erase all formulas and discard proofs after typechecking.

Security verification using RCF. Our type system and its typechecker have been used to verify implementations of complex cryptographic protocols and security mechanisms.

—Backes et al. [2009] use RCF as the formal basis of a compiler for zero-knowledge protocols; the compiler takes a verified (well-typed) protocol model and generates a well-typed RCF program, hence preserving the desired security properties.

—Baltopoulos and Gordon [2009] use F7 to validate an improved compilation strategy for the Links multi-tier programming language [Cooper et al., 2006], where keyed hashes and encryption protect the integrity and secrecy of web application data held in HTML forms.

—Bhargavan et al. [2009] use F7 as a component of a verifying protocol compiler for multi-party sessions; the compiler generates a protocol implementation along with type annotations and the typechecker verifies that the implementation meets its high-level security specification.

—Guts et al. [2009] also use our typechecker to verify the correct use of security audit logs in distributed applications; well-typed programs are guaranteed to log enough information to later convince a judge that a particular sequence of events occurred.

—Bhargavan et al. [2010] develop a revised set of cryptographic libraries for F7, with embedded logical invariants, and use them to verify a web services security protocol stack and an implementation of the widely-deployed Cardspace protocol for federated identity management.

Finally, a tutorial article [Gordon and Fournet, 2010] develops the calculus RCF in several stages (but without kinds), and summarizes the various projects building on it.

8. CONCLUSION

The use of logical formulas as computational effects is a valuable way to integrate program logics and type systems, with application to security.

Acknowledgments. Discussions with Bob Harper and Dan Licata were useful. Aslan Askarov, Michael Greenberg, and Aleks Nanevski commented on drafts of this paper. Kenneth Knowles suggested a proof technique. Cătălin Hriţcu and Matteo Maffei reported an error in a proof and suggested the use of Lemma 13 (Replacing Tainted Bounds) in Appendix C. Nikolaj Bjørner and Leonardo de Moura provided help with Z3. Sergio Maffeis was supported by EPSRC grant EP/E044956/1.

A. LOGICS

Formally, RCF is parameterized by the choice of an authorization logic, in the sense that our typed calculus depends only on a series of abstract properties of the logic, rather than on a particular 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 2.1 (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 between run time equality in RCF and term equality in F$^{\#}$.

We first give an abstract definition of the authorization logic used for the theorems, and then give a concrete definition of the logic used in the implementation. Other interesting instances of authorization 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].

A.1 Definition of Authorization Logic

We give a generic, partial definition of logic that captures only the logical properties that are used to establish our typing theorems.

An *authorization logic* is given as a set of *formulas* defined by a grammar that includes the one given below and a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas that meets the properties listed below.

Minimal Syntax of Formulas:

p	predicate symbol
$C ::=$	formula
$\quad p(M_1, \ldots, M_n)$	atomic formula
$\quad M = M'$	equation
$\quad C \wedge C'$	conjunction
$\quad C \vee C'$	disjunction

$\neg C$	negation
$\forall x.C$	universal quantification
$\exists x.C$	existential quantification

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

Properties of Deducibility: $S \vdash C$

S, C stands for $S \cup \{C\}$; in (Subst), σ ranges over substitutions of values for variables and permutations of names.

(Axiom) $\dfrac{}{C \vdash C}$ $\quad$ (Mon) $\dfrac{S \vdash C}{S, C' \vdash C}$ $\quad$ (Subst) $\dfrac{S \vdash C}{S\sigma \vdash C\sigma}$ $\quad$ (Cut) $\dfrac{S \vdash C \quad S, C \vdash C'}{S \vdash C'}$ $\quad$ (And Intro) $\dfrac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$ $\quad$ (And Elim) $\dfrac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$

(Or Intro) $\dfrac{S \vdash C_i}{S \vdash C_0 \vee C_1}\ i = 0, 1$ $\quad$ (Exists Intro) $\dfrac{S \vdash C\{M/x\}}{S \vdash \exists x.C}$ $\quad$ (Exists Elim) $\dfrac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin fv(S, C')}{S \vdash C'}$

(Eq) $\dfrac{}{\varnothing \vdash M = M}$ $\quad$ (Ineq) $\dfrac{M \neq N \quad fv(M, N) = \varnothing}{\varnothing \vdash M \neq N}$ $\quad$ (Ineq Cons) $\dfrac{h\,N = M \text{ for no } N \quad fv(M) = \varnothing}{\varnothing \vdash \forall x.hx \neq M}$

We have a derived property (True) $\varnothing \vdash \mathsf{True}$.

Although these properties are mostly standard in first-order logic, they are not complete; for instance, we do not set any axiom for negation, so our typing results apply both to intuitionistic and classical logics. Also, we do not provide enough properties to discharge the proof obligations when typing our examples.

We use property (Mon) for the soundness of typing sub-expressions, and use property (Subst) for establishing substitution lemmas. We also implicitly use (Subst) for handling the terms of RCF up to α-conversion on bound names and variables.

We use the properties (And Intro), (And Elim), (Exists Intro), (Exists Elim), and (True) in the proof of Lemma 27 ($\Rrightarrow$ Preserves Logic), to show that the formula $\overline{A}$ extracted from an expression A is preserved by structural equivalence.

We use the properties (Eq), (Ineq), and (Or Intro) in the proof of Lemma 29 ($\rightarrow$ Preserves Logic), for the soundness of the typing rule (Exp Eq). Similarly, we use property (Ineq Cons) for the soundness of (Exp Match Inl Inr Fold).

Since functions $\mathbf{fun}\, x \rightarrow A$ are values, they may occur in atomic formulas or equations. Still, these functions are essentially inert in the logic; they can be compared for equality but the logic does not allow reasoning about the application of functions. Said otherwise, the equational theory $M = M'$ is only up to α-conversion, but not for instance β-conversion. Recall that we identify the syntax of values up to the consistent renaming of bound variables, so that, for example, $\mathbf{fun}\, x \rightarrow x$ and $\mathbf{fun}\, y \rightarrow y$ are the same value. Hence, $\varnothing \vdash \mathbf{fun}\, x \rightarrow x = \mathbf{fun}\, y \rightarrow y$ is an instance of (Eq).

A.2 An Authorization Logic based on First-Order Logic

For the sake of a self-contained exposition, we review classical first-order logic (predicate calculus) with equality, as supported by the Z3 prover used by our typechecker.

First-Order Logic (Review). The syntax of first-order logic consists of sets of *formulas*, C, and *terms*, t, induced by sets of *predicate symbols*, p, and *function symbols*, f.

Syntax of First-Order Terms and Formulas:

$$t ::= x \mid f(t_1,\ldots,t_n)$$
$$C ::= p(t_1,\ldots,t_n) \mid (t = t') \mid \mathsf{False} \mid C \wedge C' \mid C \vee C' \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C$$
$$\neg C \triangleq (C \Rightarrow \mathsf{False}) \qquad t \neq t' \triangleq \neg(t = t')$$

We recall a proof system, FOL, for classical first-order logic with equality in the style of Gentzen's natural-deduction. (More precisely, this is the theory of classical first-order logic with equality as implemented in Isabelle [Paulson, 1991], presented using sequents following, for example, Dummett [1977] and Paulson [1987].

Proof Theory FOL: $S \vdash C$

$$\text{(FOL Assume)}\ \frac{C \in S}{S \vdash C} \qquad \text{(FOL Refl)}\ \frac{}{S \vdash t = t} \qquad \text{(FOL Subst)}\ \frac{S \vdash t = t' \quad S \vdash C\{t/x\}}{S \vdash C\{t'/x\}}$$

$$\text{(FOL And Intro)}\ \frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1} \qquad \text{(FOL And Elim)}\ \frac{S \vdash C_0 \wedge C_1}{S \vdash C_i} \qquad \text{(FOL Or Intro)}\ \frac{S \vdash C_i}{S \vdash C_0 \vee C_1}\ i = 0,1$$

$$\text{(FOL Or Elim)}\ \frac{S \vdash C_0 \vee C_1 \quad S, C_0 \vdash C' \quad S, C_1 \vdash C'}{S \vdash C'} \qquad \text{(FOL False)}\ \frac{S \vdash \mathsf{False}}{S \vdash C} \qquad \text{(FOL Classical)}\ \frac{S, \neg C \vdash C}{S \vdash C}$$

$$\text{(FOL Imply Intro)}\ \frac{S, C \vdash C'}{S \vdash C \Rightarrow C'} \qquad \text{(FOL Imply Elim)}\ \frac{S \vdash C \Rightarrow C' \quad S \vdash C}{S \vdash C'} \qquad \text{(FOL All Intro)}\ \frac{S \vdash C \quad x \notin fv(S)}{S \vdash \forall x.C} \qquad \text{(FOL All Elim)}\ \frac{S \vdash \forall x.C}{S \vdash C\{t/x\}}$$

$$\text{(FOL Exists Intro)}\ \frac{S \vdash C\{t/x\}}{S \vdash \exists x.C} \qquad \text{(FOL Exists Elim)}\ \frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin fv(S, C')}{S \vdash C'}$$

The only rule of FOL that is specific to classical logic is (FOL Classical). The proof theory IFOL [Paulson, 1991] for intuitionistic first-order logic consists of all the rules of FOL apart from (FOL Classical).

An Authorization Logic. To construct an authorization logic from FOL, we begin by specifying a particular instance of FOL, and translation from the formulas of authorization logic into this instance.

The syntaxes of formulas in the two logics are essentially the same. The only subtlety in the translation is that the phrases of RCF syntax, including values M and expressions within values, that may occur in authorization logic formulas include binders, while the syntax of first-order terms does not. Our solution is to use the standard first-order *locally-nameless representation* of syntax with binders introduced by de Bruijn [1972]. Each bound name or variable in an RCF phrase is represented as a numeric index, while each free name or variable is represented by itself. We assume that the set of variables of RCF coincides with the variables of FOL, and that each of the (countable) set of names of RCF is included as a nullary function symbol (that is, a constant) in FOL. Moreover, we assume there is a function symbol for each form of RCF phrase, zero and successor symbols to

represent indexes, a function symbol to form a bound variable from an index, and one to form a bound name from an index. We refer to these function symbols (including names) as *syntactic*. Hence, any phrase of RCF has a representation as a first-order term; in particular, we write $\underline{M}$ for the term representing the value M. (We omit the standard details of the locally nameless representation; for a discussion see, for example, Gordon [1994] and Aydemir et al. [2008].) Notice that if M is obtained from N by consistent renaming of bound names and variables then $\underline{M}$ and $\underline{N}$ are identical first-order terms.

Hence, we may obtain an FOL formula $\underline{C}$ from an authorization logic formula C via a homomorphic translation with base cases $\underline{p(M_1,\ldots,M_n)} = p(\underline{M_1},\ldots,\underline{M_n})$ and $\underline{M=N} = \underline{M} = \underline{N}$. We extend the translation to sets of formulas: $\underline{S} = \{\underline{C_1},\ldots,\underline{C_n}\}$ when $S = \{C_1,\ldots,C_n\}$.

In our intended model, the semantics of a term is an element of a domain defined as the free algebra with constructors corresponding to each of the syntactic function symbols. Hence, the domain is the set of closed phrases of RCF in de Bruijn representation.

We extend the theory FOL with standard axioms valid in the underlying free algebra, that syntactic function symbols yield distinct results, and are injective. (The notation $\vec{x} = \vec{y}$ means $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$ where $\vec{x}$ and $\vec{y}$ are the lists $x_1,\ldots,x_n$ and $y_1,\ldots,y_n$.)

Additional Rules for FOL/F:

(F Disjoint)
$$\frac{f \neq f' \text{ syntactic}}{S \vdash \forall \vec{x}.\forall \vec{y}.f(\vec{x}) \neq f'(\vec{y})}$$

(F Injective)
$$\frac{f \text{ syntactic}}{S \vdash \forall \vec{x}.\forall \vec{y}.f(\vec{x}) = f(\vec{x}) \Rightarrow \vec{x} = \vec{y}}$$

We can use Z3, or some other general SMT solver, to check whether a sequent $S \vdash C$ is derivable in FOL/F by simply declaring an axiom for each instance of (F Disjoint) and (F Injective). (The problem is semi-decidable so the SMT solver may fail to determine whether or not the sequent is derivable.)

Now, we define our authorization logic: we take the set of formulas to be exactly the minimal syntax of Appendix A.1, and we define the deducibility relation $S \vdash C$ to hold if and only if the sequent $\underline{S} \vdash \underline{C}$ is derivable in the theory FOL/F.

THEOREM 4 (LOGIC). *FOL/F is an authorization logic.*

Since the derivations do not need (FOL Classical), the intuitionistic variation IFOL/F could also serve as an authorization logic.

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:

$\prod_{i\in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \ldots \Rsh A_n$

$\mathscr{L} ::= \{\} \mid (\textbf{let } x = \mathscr{L} \textbf{ in } B)$

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

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

Structures formalize the idea, explained in Section 2.1, that a state has three components:

(1) a series of elementary expressions e_k being evaluated in parallel contexts;
(2) a series of messages M_j sent on channels but not yet received; and
(3) the *log*, a series of assumed formulas C_i.

Heating: $A \Rrightarrow A'$

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

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)
$A \Rrightarrow A' \Rightarrow \textbf{let}\ x = A\ \textbf{in}\ B \Rrightarrow \textbf{let}\ x = A'\ \textbf{in}\ B$	(Heat Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)
$() \Rsh A \equiv A$	(Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$\textbf{assume}\ C \Rrightarrow \textbf{assume}\ C \Rsh ()$	(Heat Assume ())
$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$	(Heat Res Fork 1)
$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$	(Heat Res Fork 2)
$a \notin fn(B) \Rightarrow$ $\textbf{let}\ x = (\nu a)A\ \textbf{in}\ B \Rrightarrow (\nu a)\textbf{let}\ x = A\ \textbf{in}\ B$	(Heat Res Let)
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$	(Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$	(Heat Fork Comm)
$\textbf{let}\ x = (A \Rsh A')\ \textbf{in}\ B \equiv$ $A \Rsh (\textbf{let}\ x = A'\ \textbf{in}\ B)$	(Heat Fork Let)

LEMMA 1 (STRUCTURE).
For every expression A, there is a structure $\mathbf{S}$ *such that* $A \Rrightarrow \mathbf{S}$.

Reduction: $A \to A'$

$(\textbf{fun}\, x \to A)\ N \to A\{N/x\}$	(Red Fun)
$(\textbf{let}\ (x_1, x_2) = (N_1, N_2)\ \textbf{in}\ A) \to$ $A\{N_1/x_1\}\{N_2/x_2\}$	(Red Split)
$(\textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B) \to$ $\begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$	(Red Match)
$M = N \to \begin{cases} \textbf{true} & \text{if } M = N \\ \textbf{false} & \text{otherwise} \end{cases}$	(Red Eq)
$a!M \Rsh a? \to M$	(Red Comm)
$\textbf{assert}\ C \to ()$	(Red Assert)
$\textbf{let}\ x = M\ \textbf{in}\ A \to A\{M/x\}$	(Red Let Val)

$A \to A' \Rightarrow \textbf{let}\ x = A\ \textbf{in}\ B \to \textbf{let}\ x = A'\ \textbf{in}\ B$ (Red Let)
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$ (Red Res)
$A \to A' \Rightarrow (A \Rsh B) \to (A' \Rsh B)$ (Red Fork 1)
$A \to A' \Rightarrow (B \Rsh A) \to (B \Rsh A')$ (Red Fork 2)
$A \to A' \quad \text{if } A \Rrightarrow B, B \to B', B' \Rrightarrow A'$ (Red Heat)

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. PROPERTIES OF THE TYPE SYSTEM

The structure of this appendix is as follows.

—Appendix C.1 develops basic properties of the type system, such as weakening, strengthening, and exchange lemmas.

—Appendix C.2 contains the proof of Lemma 14 (Public Down/Tainted Up), which characterizes the relationship between the public and tainted kinds and subtyping.

—Appendix C.3 establishes properties of subtyping, principally Lemma 19 (Transitivity), that subtyping is transitive.

—Appendix C.4 presents an alternative characterization of the expression typing relation, avoiding the non-structural rule (Val Refine), by building its effect into each of the structural rules for values; this characterization is useful in various subsequent proofs.

—Appendix C.5 proves various properties of substitution.

—Appendix C.6 establishes Theorem 1 (Safety). The main lemmas in the proof are Proposition 28 ($\Rrightarrow$ Preserves Types) and Proposition 30 ($\to$ Preserves Types).

—Finally, Appendix C.7 establishes Theorem 2 (Robust Safety); the main additional lemma needed is Lemma 33 (Opponent Typability), that any opponent expression can be typed within the system.

C.1 Basic Properties

We begin with some standard properties of our type system. To state them, we let $\mathscr{J}$ range over $\{\diamond, T, C, T :: \nu, T <: T', A : T\}$.

LEMMA 2 (DERIVED JUDGMENTS).

(1) *If* $E \vdash T$ *then* $E \vdash \diamond$ *and* $fnfv(T) \subseteq dom(E)$.
(2) *If* $E \vdash C$ *then* $E \vdash \diamond$ *and* $fnfv(C) \subseteq dom(E)$.
(3) *If* $E \vdash T :: \nu$ *then* $E \vdash T$.
(4) *If* $E \vdash T <: T'$ *then* $E \vdash T$ *and* $E \vdash T'$.
(5) *If* $E \vdash A : T$ *then* $E \vdash T$ *and* $fnfv(A) \subseteq dom(E)$.

LEMMA 3 (STRENGTHENING).
If $E, \mu, E' \vdash \mathscr{J}$ *and* $dom(\mu) \cap (fnfv(E') \cup fnfv(\mathscr{J})) = \varnothing$ *and* $forms(E, E') \vdash C$ *for all* $C \in forms(\mu)$, *then* $E, E' \vdash \mathscr{J}$.

LEMMA 4 (EXCHANGE).
If $E, \mu_1, \mu_2, E' \vdash \mathscr{J}$ *and* $dom(\mu_1) \cap fnfv(\mu_2) = \varnothing$ *then* $E, \mu_2, \mu_1, E' \vdash \mathscr{J}$.

LEMMA 5 (WEAKENING).
If $E, E' \vdash \mathscr{J}$ *and* $E, \mu, E' \vdash \diamond$ *then* $E, \mu, E' \vdash \mathscr{J}$.

The following lemma captures the idea that the formulas in a tainted type cannot be relied upon, because any data produced by the opponent may flow into a tainted type.

LEMMA 6 (KINDING).
If $E \vdash T :: \mathbf{tnt}$, *then* $E \vdash C$ *for all* $C \in \mathsf{forms}(\_ : T)$.

If T is a subtype of T', then the set of formulas $\mathsf{forms}(x : T)$ is logically stronger than the set of formulas $\mathsf{forms}(x : T')$.

LEMMA 7 (LOGICAL SUBTYPING).
If $E \vdash T <: T'$ *and* $x \notin dom(E)$ *then* $\mathsf{forms}(E), \mathsf{forms}(x : T) \vdash \mathsf{forms}(x : T')$.

Our system enjoys a standard bound weakening property, that an occurrence of T in the environment of a judgment can be replaced by a subtype T'.

LEMMA 8 (BOUND WEAKENING).
Suppose that $E \vdash T' <: T$. *If* $E, x : T, E' \vdash \mathscr{J}$ *then* $E, x : T', E' \vdash \mathscr{J}$. *Moreover the depth of the derivation of the second judgment equals that of the first (except where* $\mathscr{J}$ *is a typing judgment).*

Recall that an ok-type $\{C\}$ is a token witnessing that the formula C holds, and is defined to be the refinement type $\{\_ : \mathsf{unit} \mid C\}$. The final lemmas in this section state some simple properties of ok-types.

LEMMA 9 (BOUND WEAKENING OK).
Suppose that $E, C' \vdash C$. *If* $E, x : \{C\}, E' \vdash \mathscr{J}$ *then* $E, x : \{C'\}, E' \vdash \mathscr{J}$.

LEMMA 10 (SUB REFINE LEFT REFL).
If $E \vdash \{x : T \mid C\}$ *then* $E \vdash \{x : T \mid C\} <: T$.

LEMMA 11 (AND SUB).
If $E \vdash \{x : T \mid C_1 \wedge C_2\}$ *then:* $E \vdash \{x : T \mid C_1 \wedge C_2\} <:> \{x : \{x : T \mid C_1\} \mid C_2\}$.

LEMMA 12 (OK $\wedge$).
We have $E, \_ : \{C_1\}, \_ : \{C_2\}, E' \vdash \mathscr{J}$ *if and only if* $E, \_ : \{C_1 \wedge C_2\}, E' \vdash \mathscr{J}$.

C.2 Properties of Kinding

We introduced in Section 3.2 a universal type Un of data known to the opponent. Lemma 15 (Public Tainted) is a standard characterization [Gordon and Jeffrey, 2003b] of the public and tainted kinds: a type T is public if and only if it is a subtype of Un, and a type is tainted if and only if it is a supertype of Un. The next two lemmas are needed in the proof of this main lemma.

LEMMA 13 (REPLACING TAINTED BOUNDS).
If $E, x : T, E' \vdash U :: \nu$ *and* $E \vdash T :: \mathbf{tnt}$ *and* $E \vdash V$ *then* $E, x : V, E' \vdash U :: \nu$.

LEMMA 14 (PUBLIC DOWN/TAINTED UP).

(1) *If* $E \vdash T <: T'$ *and* $E \vdash T' :: \mathbf{pub}$ *then* $E \vdash T :: \mathbf{pub}$.

(2) *If* $E \vdash T :: \mathbf{tnt}$ *and* $E \vdash T <: T'$ *then* $E \vdash T' :: \mathbf{tnt}$.

LEMMA 15 (PUBLIC TAINTED).
For all T *and executable* E*:*

(1) $E \vdash T :: \mathbf{pub}$ *if and only if* $E \vdash T <: \mathsf{Un}$.

(2) $E \vdash T :: \mathbf{tnt}$ *if and only if* $E \vdash \mathsf{Un} <: T$.

C.3 Properties of Subtyping

The main result in this section is transitivity of subtyping, perhaps the most difficult proof in the development, because it needs a relatively complex inductive argument.

The proof of transitivity depends on the following lemma, the first two of which concern the use of recursive type variables declared by entries $\alpha <: \alpha'$ in the typing environment.

LEMMA 16 (REC KINDING).
If $E \vdash T :: \nu$ *and* $(\alpha <: \alpha') \in E$ *then* $\alpha \notin fnfv(T)$ *and* $\alpha' \notin fnfv(T)$.

LEMMA 17 (REC SUBTYPING).
If $E \vdash T <: T'$ *and* $(\alpha <: \alpha') \in E$ *we have that:* $\{\alpha, \alpha'\} \cap fnfv(T) = \varnothing$ *if and only if* $\{\alpha, \alpha'\} \cap fnfv(T') = \varnothing$.

The following lemma formalizes the intuition that the formulas decorating the type in the environment are all that matter as far as the kinding and subtyping judgments are concerned. In particular, we can replace an environment entry $x : T$ with $x : (T)^\sharp$, where $(T)^\sharp$ is the refinement of the unit type given as follows.

Formulizing a Type:

$$(T)^\sharp \triangleq \{x : \mathsf{unit} \mid \mathsf{forms}(x : T)\}$$

LEMMA 18 (FORMULIZE TYPE). *Assume* $E, x : T, E' \vdash \diamond$.

(1) $E, x : (T)^\sharp, E' \vdash \diamond$.

(2) $E, x : T, E' \vdash C$ *iff* $E, x : (T)^\sharp, E' \vdash C$.

(3) $E, x : T, E' \vdash U :: \nu$ *iff* $E, x : (T)^\sharp, E' \vdash U :: \nu$.

(4) $E, x : T, E' \vdash U <: U'$ *iff* $E, x : (T)^\sharp, E' \vdash U <: U'$.

Moreover, the depth of the derivations of each pair of judgments is the same.

LEMMA 19 (TRANSITIVITY).
If E *is executable and* $E \vdash T <: T'$ *and* $E \vdash T' <: T''$ *then* $E \vdash T <: T''$.

C.4 Alternative Formulation of Typing

We present an alternative definition of expression typing, which avoids the non-structural rule (Val Refine), and hence is useful in the proofs of Lemma 22 (Substitution), Proposition 28 ($\Rightarrow$ Preserves Types) and Proposition 30 ($\rightarrow$ Preserves Types).

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

$$\frac{E \vdash C\{x/y\} \quad (x:T) \in E}{E \vdash x : \{y : T \mid C\}} \text{ (Val Var Refine)} \qquad \frac{E \vdash C\{()/y\}}{E \vdash () : \{y : \mathsf{unit} \mid C\}} \text{ (Val Unit Refine)}$$

$$\frac{E, x : T \vdash A : U \quad E \vdash C\{\mathbf{fun}\, x \to A/y\}}{E \vdash \mathbf{fun}\, x \to A : \{y : (\Pi x : T.\ U) \mid C\}} \text{ (Val Fun Refine)}$$

$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\} \quad E \vdash C\{(M,N)/y\}}{E \vdash (M,N) : \{y : (\Sigma x : T.\ U) \mid C\}} \text{ (Val Pair Refine)}$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U \quad E \vdash C\{h\, M/y\}}{E \vdash h\, M : \{y : U \mid C\}} \text{ (Val Inl Inr Fold Refine)}$$

LEMMA 20 (ALTERNATIVE TYPING).
Assuming that E is executable, the expression typing relation $E \vdash A : T$ is the least relation closed under the alternative rules for values displayed above together with the original rules for expressions.

LEMMA 21 (FORMULAS).
If $E \vdash M : T$ and $x \notin dom(E)$ then $\mathsf{forms}(E) \vdash \mathsf{forms}(x : T)\{M/x\}$.

C.5 Properties of Substitution

To state the two substitution lemmas in this section, we need a notation for applying a substitution to the entries in environments. If $x \notin dom(E)$, let $E\{M/x\}$ be the outcome of applying $\{M/x\}$ to each type occurring in E. Similarly, if $\alpha \notin dom(E)$, let $E\{T/\alpha\}$ be the outcome of applying $\{T/\alpha\}$ to each type occurring in E. We define these notations as follows.

Substitution into Typing Environments:

$$E\{M/x\} = (\mu_1\{M/x\}, \ldots, \mu_n\{M/x\}) \quad \text{where } x \notin dom(E) \text{ and } E = \mu_1, \ldots, \mu_n$$

$$\mu\{M/x\} = \begin{cases} y : (U\{M/x\}) & \text{if } \mu = (y : U) \text{ and } x \neq y \\ a \updownarrow (U\{M/x\}) & \text{if } \mu = a \updownarrow U \\ \mu & \text{otherwise} \end{cases}$$

$$E\{T/\alpha\} = (\mu_1\{T/\alpha\}, \ldots, \mu_n\{T/\alpha\}) \quad \text{where } \alpha \notin dom(E) \text{ and } E = \mu_1, \ldots, \mu_n$$

$$\mu\{T/\alpha\} = \begin{cases} y : (U\{T/\alpha\}) & \text{if } \mu = (y : U) \\ a \updownarrow (U\{T/\alpha\}) & \text{if } \mu = a \updownarrow U \\ \mu & \text{otherwise} \end{cases}$$

Our first substitution lemma shows how substitution of a value M for a variable x affects various judgments.

LEMMA 22 (SUBSTITUTION).

(1) *If $h : (T,U)$ then $h : (T\{M/x\}, U\{M/x\})$.*
(2) *If $x \notin dom(E)$ then $\mathsf{forms}(E)\{M/x\} = \mathsf{forms}(E\{M/x\})$.*

(3) *If* $E, x : U, E' \vdash \diamond$ *and* $E \vdash M : U$ *then* $E, (E'\{M/x\}) \vdash \diamond$.

(4) *If* $E, x : U, E' \vdash C$ *and* $E \vdash M : U$ *then* $E, (E'\{M/x\}) \vdash C\{M/x\}$.

(5) *Suppose that* $E \vdash M : U$.
—*If* $E, x : U, E' \vdash T$ *then* $E, (E'\{M/x\}) \vdash T\{M/x\}$.
—*If* $E, x : U, E' \vdash T :: \nu$ *then* $E, (E'\{M/x\}) \vdash T\{M/x\} :: \nu$.
—*If* $E, x : U, E' \vdash T <: T'$ *then* $E, (E'\{M/x\}) \vdash T\{M/x\} <: T'\{M/x\}$.
—*If* $E, x : U, E' \vdash A : T$ *then* $E, (E'\{M/x\}) \vdash A\{M/x\} : T\{M/x\}$.

The following auxiliary lemma expresses that kinding judgments do not depend on type declarations of the form $\alpha <: \alpha'$.

LEMMA 23.
If $E, \alpha <: \alpha', E' \vdash T :: \nu$ *then* $E, \alpha :: \mathbf{pub}, \alpha' :: \mathbf{tnt}, E' \vdash T :: \nu$.

Our second substitution lemma shows how substitution of a type T for a variable α affects various judgments.

LEMMA 24 (TYPE SUBSTITUTION).

(1) *If* $E, \alpha, E' \vdash \mathscr{J}$ *and* $E \vdash T$ *and* $recvar(E) \cap fnfv(T) = \varnothing$ *then* $E, (E'\{T/\alpha\}) \vdash \mathscr{J}\{T/\alpha\}$.

(2) *If* $E, \alpha :: \nu, E' \vdash \diamond$ *and* $E \vdash T :: \nu$ *then* $E, (E'\{T/\alpha\}) \vdash \diamond$.

(3) *If* $E, \alpha :: \nu, E' \vdash U$ *and* $E \vdash T :: \nu$ *then* $E, (E'\{T/\alpha\}) \vdash U\{T/\alpha\}$.

(4) *If* $E, \alpha :: \nu, E' \vdash T' :: \nu'$ *and* $E \vdash T :: \nu$ *then* $E, (E'\{T/\alpha\}) \vdash T'\{T/\alpha\} :: \nu'$.

(5) *If* $E, \alpha <: \alpha', E' \vdash T <: T'$ *and* $E \vdash U <: U'$ *then* $E, (E'\sigma) \vdash T\sigma <: T'\sigma$ *where* $\sigma = \{U/\alpha\}\{U'/\alpha'\}$.

C.6 Proof of Theorem 1 (Safety)

We need the following inversion lemma, for analyzing instances of subtyping.

LEMMA 25 (INVERSION).

(1) *Let* T *be* $\{y : (\Pi x : T''.\ U'') \mid C\}$ *or* $(\Pi x : T''.\ U'')$.
If $E \vdash T <: \Pi x : T'.\ U'$ *then* $E \vdash T' <: T''$ *and* $E, x : T' \vdash U'' <: U'$.

(2) *Let* T *be* $\{y : (\Sigma x : T''.\ U'') \mid C\}$ *or* $(\Sigma x : T''.\ U'')$.
If $E \vdash T <: \Sigma x : T'.\ U'$ *then* $E \vdash T'' <: T'$ *and* $E, x : T'' \vdash U'' <: U'$.

(3) *Let* T *be* $\{y : (\mu\alpha.U) \mid C\}$ *or* $(\mu\alpha.U)$.
If $E \vdash T <: \mu\alpha'.U'$ *then* $E \vdash U\{\mu\alpha.U/\alpha\} <: U'\{\mu\alpha'.U'/\alpha'\}$.

(4) *Let* T *be* $\{y : T_1 + T_2) \mid C\}$ *or* $T_1 + T_2$.
If $E \vdash T <: U_1 + U_2$ *then* $E \vdash T_1 <: U_1$ *and* $E \vdash T_2 <: U_2$.

(5) *Let* h *be* inl, inr, *or* fold. *Let* T *be* $\{y : U \mid C\}$ *or* U *for any* U *such that* $h : (H, U)$. *For any* H' *and* U' *such that* $h : (H', U')$, *if* $E \vdash T <: U'$ *then* $E \vdash H <: H'$.

Recall from Section 4 that $\overline{A}$ is the set of formulas extracted from the expression A. For example, $\overline{(\nu a)(\mathbf{assume}\ \mathsf{Foo}(a,x)\ \Rsh\ \mathbf{assume}\ \mathsf{Bar}(z))} = \exists a.(\mathsf{Foo}(a,x) \wedge \mathsf{Bar}(z))$. The following states that if A is well-typed in environment E, then the formulas extracted from A are well-formed in E, that is, all their free variables are declared in E.

LEMMA 26.
If $E \vdash A : T$ *then* $E \vdash \{\overline{A}\}$.

The next two lemmas assert that heating $A \Rrightarrow A'$ preserves the extracted formulas of an expression (that is, the formulas extracted from A' follow from those extracted from A) and also that heating preserves types.

LEMMA 27 ($\Rrightarrow$ PRESERVES LOGIC).
If $A \Rrightarrow A'$ then $\overline{A'} \vdash \overline{A}$.

PROPOSITION 28 ($\Rrightarrow$ PRESERVES TYPES).
If E is executable, $E \vdash A : T$, and $A \Rrightarrow A'$, then $E \vdash A' : T$.

Similarly, the next two lemmas assert that reduction $A \to A'$ preserves the extracted formulas of an expression and also that reduction preserves types.

LEMMA 29 ($\to$ PRESERVES LOGIC).
If $A \to A'$ then $\overline{A'} \vdash \overline{A}$.

PROPOSITION 30 ($\to$ PRESERVES TYPES).
If E is executable, $fv(A) = \varnothing$, $E \vdash A : T$, and $A \to A'$, then $E \vdash A' : T$.

Our next results are that typing implies static safety and indeed safety.

LEMMA 31 (STATIC SAFETY).
If $\varnothing \vdash \mathbf{S} : T$ then $\mathbf{S}$ is statically safe.

RESTATEMENT OF THEOREM 1 (SAFETY)
If $\varnothing \vdash A : T$ then A is safe.

PROOF. Consider any A' and $\mathbf{S}$ such that $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$; it suffices to show that $\mathbf{S}$ is statically safe. By Proposition 30 ($\to$ Preserves Types), $\varnothing \vdash A : T$ and $A \to^* A'$ imply $\varnothing \vdash A' : T$. By Proposition 28 ($\Rrightarrow$ Preserves Types), this and $A' \Rrightarrow \mathbf{S}$ imply $\varnothing \vdash \mathbf{S} : T$. By Lemma 31 (Static Safety), this implies $\mathbf{S}$ is statically safe. □

C.7 Proof of Theorem 2 (Robust Safety)

First, we note that Un is type equivalent to a range of types.

LEMMA 32 (UNIVERSAL TYPE).
Given $E \vdash \diamond$ we have $E \vdash Un <:> T$ for each T below:

$$\{unit, (\Pi x : Un.\ Un), (\Sigma x : Un.\ Un), (Un + Un), (\mu\alpha.Un)\}$$

The next lemma establishes that any opponent can be well-typed using Un to type its free names. The lemma is a little more general—it applies to any expression containing no Assert; an opponent is any such expression with no free variables.

LEMMA 33 (OPPONENT TYPABILITY).
*Suppose $E \vdash \diamond$ and that E is executable. If O is an expression containing no **assert** such that $(a \updownarrow Un) \in E$ for each name $a \in fn(O)$, and $(x : Un) \in E$ for each variable $x \in fv(O)$, then $E \vdash O : Un$.*

Finally, we prove that robust safety follows by typing.

RESTATEMENT OF THEOREM 2 (ROBUST SAFETY)
If $\varnothing \vdash A : Un$ then A is robustly safe.

PROOF. Consider any opponent O with $fn(O) = \{a_1, \ldots, a_n\}$. We are to show the application $O\ A$ is safe. Let $E = a_1 \updownarrow \mathsf{Un}, \ldots, a_1 \updownarrow \mathsf{Un}$. By Lemma 33 (Opponent Typability), $E \vdash O : \mathsf{Un}$. By (Exp Subsum) and Lemma 32 (Universal Type), $E \vdash O : (\Pi x : \mathsf{Un}.\ \mathsf{Un})$. By Lemma 5 (Weakening), $E \vdash A : \mathsf{Un}$. We can easily derive $E \vdash \mathbf{let}\ f = O\ \mathbf{in}\ (\mathbf{let}\ x = A\ \mathbf{in}\ f\ x) : \mathsf{Un}$, that is, $E \vdash O\ A : \mathsf{Un}$. By Theorem 1 (Safety), $O\ A$ is safe. □

REFERENCES

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

M. Abadi. Access control in a core calculus of dependency. In *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, pages 5–31. Elsevier, 2007. Volume 172 of ENTCS.

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 and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *LNCS*, pages 197–221. Springer, 2005.

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.

B. Aydemir, A. Chargéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 3–17. ACM, 2008.

M. Backes, M. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 308–323. IEEE Computer Society, 2009.

I. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2009)*, pages 27–38, 2009.

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.

M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR–TR–2008–118, Microsoft Research, 2010. Revision of February 2010.

K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM Conference on Computer and Communications Security*, pages 459–468, 2008a.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31:5:1–5:61, December 2008b.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 124–140, July 2009.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456. ACM, 2010.

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. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, 2005.

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.

J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. Technical Report MSR–TR–2009–97, Microsoft Research, 2009.

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

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium*, pages 172–185, 2009.

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.

R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS. Springer-Verlag, 2006.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76 (2-3):95–120, 1988.

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

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

D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy*, 1996.

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.

M. A. E. Dummett. *Elements of intuitionism*. Clarendon Press, 1977.

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.

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. A mechanisation of name-carrying syntax up to alpha-conversion. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications. Proceedings, 1993*, number 780 in LNCS, pages 414–426. Springer, 1994.

A. D. Gordon and C. Fournet. Principles and applications of refinement types. In *Proceedings of the NATO Summer School Marktoberdorf 2009*. IOS Press, 2010. A preliminary version appears as Technical Report MSR-TR-2009-147, Microsoft Research, October 2009.

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.

A. D. Gordon and A. S. A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR 2005—Concurrency Theory*, volume 3653 of *LNCS*, pages 186–201. Springer, 2005.

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 languages*. MIT Press, 1992.

N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability by typing. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 168–183. Springer, 2009.

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.

R. Jagadeesan, A. S. A. Jeffrey, C. Pitcher, and J. Riely. Lambda-RBAC: Programming with role-based access control. *Logical Methods In Computer Science*, 2008.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *International Conference on Functional Programming (ICFP'08)*, pages 27–38. ACM, 2008.

M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *Programming Language Design and Implementation (PLDI'09)*, pages 304–315. ACM, 2009.

P. Li and S. Zdancewic. Encoding information flow in Haskell. In *IEEE Computer Security Foundations Workshop (CSFW'06)*, page 16, 2006.

S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon. Code-carrying authorization. In *13th European Symposium on Research in Computer Security (ESORICS'08)*, volume 5283 of *LNCS*, pages 563–579. Springer, Oct. 2008.

P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

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.

C. Parent. Synthesizing Proofs from Programs in the Calculus of Inductive Constructions. *Mathematics of Program Construction (MPC'95)*, 947:351–379, 1995.

L. C. Paulson. *Logic and computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1991.

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 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.

Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction*, volume 5133 of *LNCS*, pages 305–335. Springer, 2008.

P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*, pages 159–169. ACM, 2008.

P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 131–144. ACM, 2010.

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.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-Based Audit. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 177–191, 2008.

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.