

# Refinement Types for Secure Implementations

JESPER BENGTON, Uppsala University  
KARTHIKEYAN BHARGAVAN, Microsoft Research  
CÉDRIC FOURNET, Microsoft Research  
ANDREW D. GORDON, Microsoft Research  
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  $\lambda$ -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]: General; 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

## ACM Reference Format:

Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., and Maffeis, S. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (January 2011), 45 pages.  
DOI = 10.1145/1890028.1890031 <http://doi.acm.org/10.1145/1890028.1890031>

## 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

---

S. Maffeis was supported by EPSRC grant EP/E044956/1.

Authors' addresses: J. Bengtson, Department of Information Technology, Uppsala University, SE-751 05 Uppsala, Sweden; K. Bhargavan, C. Fournet (corresponding author), A. D. Gordon, Microsoft Research, Roger Needham Building, 7 J. J. Thomson Ave., Cambridge CB3 0FB, United Kingdom; email: [fournet@microsoft.com](mailto:fournet@microsoft.com); S. Maffeis, Department of Computing, Imperial College London, London SW7 2AZ, United Kingdom.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2011 ACM 0164-0925/2011/01-ART8 \$10.00

DOI 10.1145/1890028.1890031 <http://doi.acm.org/10.1145/1890028.1890031>

and Yao [1983], who is able to create arbitrarily many principals, create arbitrarily many instances of each protocol role, 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 program transformations, pass them to ProVerif, a cryptographic analyzer [Blanchet 2001; Abadi and Blanchet 2005]. Their approach yields verified security for very detailed models, but also demands considerable care in programming, in order to control the complexity of global cryptographic analysis for giant protocols. Even if ProVerif scales up remarkably well in practice, beyond a few message exchanges or a few hundred lines of  $F^\#$ , verification becomes long (up to a few days) and unpredictable (with trivial code changes leading to divergence).

*Cryptographic Verification Meets Program Verification.* In parallel with the development of specialist tools for cryptography, verification tools in general are also making rapid progress, and can deal with much larger programs (see, e.g., Flanagan et al. [2002], Filliâtre and Marché [2004], Barnett et al. [2005], and 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 (noncryptographic) 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, 2007b] treat policy enforcement as a type discipline; they develop their approach for typed  $\pi$ -calculus, 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 annotations can express general authorization policies, and their use makes typechecking more compositional and predictable than the global analysis performed by ProVerif. Moreover, typechecking succeeds even on code involving recursion and complex data structures.

*Outline of the Theory.* We justify our extended typechecker by developing a formal type theory for a core of  $F^\#$ : a concurrent call-by-value  $\lambda$ -calculus named RCF.

To represent pre- and post-conditions, our calculus has standard dependent functions 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 runtime. 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  $\lambda$ -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 symbolically coded (and typed) in the calculus, using a seal abstraction [Morris 1973; Sumii and Pierce 2007]. 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 article 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 preliminary, abridged version of this article appears in the proceedings of CSF'08 [Bengtson et al. 2008]. A technical report

[Bengtson et al. 2010] revised in November 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 article 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  $\text{Un}$  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  $\text{Un}$ ), together with an example application. Section 3 introduces  $\text{Un}$  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
$\{C_1, \dots, C_n\} \vdash C$	logical deducibility

We assume collections of *names*, *variables*, and *type variables*. A name is an identifier, generated at runtime, for a channel, while a variable is a placeholder for a value. If  $\phi$  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  $\phi$ . 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  $\phi$ . 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
$\text{inl}$	left constructor of sum type
$\text{inr}$	right constructor of sum type
$\text{fold}$	constructor of recursive type
$M, N ::=$	value
$x$	variable
$()$	unit
$\text{fun } x \rightarrow A$	function (scope of $x$ is $A$ )

$(M, N)$	pair
$h M$	construction
$A, B ::=$	expression
$M$	value
$M N$	application
$M = N$	syntactic equality
<b>let</b> $x = A$ <b>in</b> $B$	let (scope of $x$ is $B$ )
<b>let</b> $(x, y) = M$ <b>in</b> $A$	pair split (scope of $x, y$ is $A$ )
<b>match</b> $M$ <b>with</b>	constructor match
$h x \rightarrow A$ <b>else</b> $B$	(scope of $x$ is $A$ )
$(\nu a)A$	restriction (scope of $a$ is $A$ )
$A \uparrow B$	fork
$a!M$	transmission of $M$ on channel $a$
$a?$	receive message off channel
<b>assume</b> $C$	assumption of formula $C$
<b>assert</b> $C$	assertion of formula $C$

To evaluate  $M$ , return  $M$  at once. To evaluate  $M N$ , if  $M = \mathbf{fun} x \rightarrow A$ , evaluate  $A\{N/x\}$ . To evaluate  $M = N$ , if the two values  $M$  and  $N$  are the same, return **true**  $\triangleq \text{inr}()$ ; otherwise, return **false**  $\triangleq \text{inl}()$ . 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**  $h x \rightarrow A$  **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  $A \uparrow B$ , 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  $()$  (**fun**  $x \rightarrow x$ ) 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. Inasmuch as our notion of safety is relative to the assumptions executed during evaluation, these assumptions must be carefully reviewed.

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.

Once assumed, a formula remains in the log for the whole run. Thus, if an **assert** succeeds, then, later in the run, any other **assert** with the same formula will also

succeed. Hence, our notion of safety captures stable safety properties, which is adequate for verifying the security of concurrent protocols.

## 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  $\alpha, \beta$ . For any phrase  $\phi$ , the set  $fnfv(\phi)$  includes the type variables, as well as the names and (value) variables, that occur free in  $\phi$ . Notice that no types or type variables occur in the syntax of values or expressions. If  $\phi$  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  $\alpha$  in  $\phi$ .

### Syntax of Types:

$H, T, U, V ::= \text{type}$	
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 $\alpha$ is $T$ )
$\alpha$	type variable
$\{x : T \mid C\}$	refinement type (scope of $x$ is $C$ )
$\{C\} \triangleq \{ \_ : \text{unit} \mid C \}$	ok-type
$\text{bool} \triangleq \text{unit} + \text{unit}$	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  $\text{inl } M$  where  $M$  has type  $T$ , or  $\text{inr } N$  where  $N$  has type  $U$ . A value of type  $\mu\alpha.T$  is a construction  $\text{fold } M$ , 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  $\emptyset \vdash A : T$  then  $A$  is safe.*

**PROOF.** See Appendix C.  $\square$

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  $T_1 = \{x_1 : T \mid P(x_1)\}$  and  $T_2 = \{x_2 : \text{unit} \mid \forall z. P(z) \Rightarrow Q(z)\}$ . If  $E = (y_1 : T_1, y_2 : T_2)$  then  $E \vdash Q(y_1)$  via the preceding rule 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  $\{\_ : \text{unit} \mid C\}$ : a type of tokens that a formula holds. For example, up to variable renaming,  $T_2 = \{\forall z. P(z) \Rightarrow Q(z)\}$ . 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 `inr ()`).

- If  $T <: T'$  and  $C \vdash C'$  then  $\{x : T \mid C\} <: \{x : T' \mid C'\}$ .
- $\{x : T \mid \text{True}\} <: T$ .

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

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

- $E \vdash \text{assume } C : \{C\}$ .
- If  $E \vdash C$  then  $E \vdash \text{assert } C : \text{unit}$ .

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  $A_{ex} : \text{unit}$  where  $A_{ex}$  is the following, where `Foo` and `Bar` are nullary predicate symbols.

```
let x = assume Foo() => Bar() in
let y = assume Foo() in assert Bar()
```

By the rule for assumptions we have:

```
assume Foo() => Bar() : {Foo() => Bar()}
assume Foo() : {Foo()}
```

The rule for checking a let-expression is:

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

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

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

where  $E = x : \{\text{Foo}() \Rightarrow \text{Bar}()\}, y : \{\text{Foo}()\}$ . The displayed judgment follows by the rule for assertions as we can derive  $E \vdash \text{Bar}()$ , since we have both  $E \vdash \text{Foo}() \Rightarrow \text{Bar}()$  and  $E \vdash \text{Foo}()$ . 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**  $f = A$  is short for **let**  $f = \text{fun } x \rightarrow A$ . More notably, if  $A$  and  $B$  are proper expressions rather than being values, the

application  $AB$  is short for  $\mathbf{let } f = A \mathbf{ in } (\mathbf{let } x = B \mathbf{ in } f x)$ . In general, the use in  $F^\#$  of arbitrary expressions in place of values can be interpreted by inserting suitable lets.

The second main difference is that the RCF syntax for communication and concurrency ( $(\nu a)A$ ,  $A \uparrow B$ ,  $a?$ , 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:

$(T)\mathbf{chan} \triangleq (T \rightarrow \mathbf{unit}) * (\mathbf{unit} \rightarrow T)$	
$\mathbf{chan} \triangleq \mathbf{fun } x \rightarrow (\nu a)(\mathbf{fun } x \rightarrow a!x, \mathbf{fun } _ \rightarrow a?)$	
$\mathbf{send} \triangleq \mathbf{fun } c x \rightarrow \mathbf{let } (s, r) = c \mathbf{ in } s x$	send $x$ on $c$
$\mathbf{recv} \triangleq \mathbf{fun } c \rightarrow \mathbf{let } (s, r) = c \mathbf{ in } r ()$	block for $x$ on $c$
$\mathbf{fork} \triangleq \mathbf{fun } f \rightarrow (f() \uparrow ())$	run $f$ in parallel

We define references in terms of channels.

#### Functions for References:

$(T)\mathbf{ref} \triangleq (T)\mathbf{chan}$	
$\mathbf{ref } M \triangleq \mathbf{let } r = \mathbf{chan } "r" \mathbf{ in } \mathbf{send } r M; r$	new reference to $M$
$!M \triangleq \mathbf{let } x = \mathbf{recv } M \mathbf{ in } \mathbf{send } M x; x$	dereference $M$
$M := N \triangleq \mathbf{recv } M; \mathbf{send } M N$	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 report lists the full details.) RCF lacks full-fledged 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 nonempty tuples (instead of directly using the core syntax for dependent function and pair types). We usually omit conditions of the form  $\{\mathbf{True}\}$  in examples.

#### Derived Notation for Functions and Tuples:

$\{x_1 : T_1 \mid C_1\} \rightarrow U \triangleq \Pi x_1 : \{x_1 : T_1 \mid C_1\}. U$
$(x_1 : T_1 * \dots * x_n : T_n)\{C\} \triangleq \begin{cases} \Sigma x_1 : T_1. \dots \Sigma x_{n-1} : T_{n-1}. \{x_n : T_n \mid C\} & \text{if } n > 0 \\ \{C\} & \text{otherwise} \end{cases}$

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  $h(M_1, \dots, M_n)$  is a formula.

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

We sketch how to interpret interfaces and modules as tuple types and expressions. In essence, an *interface* is a sequence  $\mathbf{val } x_1 : T_1 \dots \mathbf{val } x_n : T_n$  of *value declarations*, which we interpret by the tuple type  $(x_1 : T_1 * \dots * x_n : T_n)$ . A *module* is a sequence  $\mathbf{let } x_1 = A_1 \dots \mathbf{let } x_n = A_n$  of *value definitions*, which we interpret by the expression  $\mathbf{let } x_1 = A_1 \mathbf{ in } \dots \mathbf{let } x_n = A_n \mathbf{ in } (x_1, \dots, x_n)$ . 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  $\mathbf{val } x : \{C\}$  plus a definition  $\mathbf{let } x = \mathbf{assume } C$  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  $\forall x. \text{CanWrite}(x) \Rightarrow \text{CanRead}(x)$ 
```

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

```
val publicfile: file : string → unit{ PublicFile(file) }
assume  $\forall x. \text{PublicFile}(x) \Rightarrow \text{CanRead}(x)$ 
```

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 →  $\alpha$` , where  $\alpha$  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 following test code 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**  $\forall x. \text{CanRead}(x)$ . 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 rearrange 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 that follows, for instance, the function reader can be called to access the content of file readme in any context with no further runtime 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 acs: (string,entry) Db.t = Db.create()
let safe_read file =
  match Db.select acs file with
  | Readable file → read file
  | Writable file → read file
  | _ → failwith "unreadable"
let readable file =
  match Db.select acs 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 next code illustrates their usage.

```

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

```

### 3. MODELING 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 shortly) 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* in that no **assert** fails, despite the best efforts of an arbitrary opponent.

#### Formal Threat Model: Opponents and Robust Safety

<p>A closed expression <math>O</math> is an <i>opponent</i> iff <math>O</math> contains no occurrence of <b>assert</b>.</p> <p>A closed expression <math>A</math> is <i>robustly safe</i> iff the application <math>O A</math> is safe for all opponents <math>O</math>.</p> <p>(An opponent must contain no <b>assert</b>, since otherwise it could vacuously falsify safety.)</p>
---

#### 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,  $\text{Un}$  is type equivalent to (both a subtype and a supertype of) all of the following types:  $\text{unit}$ ,  $(\Pi x : \text{Un}. \text{Un})$ ,  $(\Sigma x : \text{Un}. \text{Un})$ ,  $(\text{Un} + \text{Un})$ , and  $(\mu\alpha.\text{Un})$ . Hence, we obtain *opponent typability*, that  $O : \text{Un}$  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  $T <: \text{Un}$ .

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

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].)

—  $E \vdash \{x : T \mid C\} <: \text{Un}$  iff  $E \vdash T <: \text{Un}$

—  $E \vdash \text{Un} <: \{x : T \mid C\}$  iff  $E \vdash \text{Un} <: T$  and  $E, x : T \vdash C$

Consider the type  $T_1 \triangleq \{x : \text{string} \mid \text{CanRead}(x)\}$ . According to the preceding rules, this type is public because  $\text{string}$  is public, but it is only tainted if  $\text{CanRead}(x)$  holds for all  $x$  (and let's suppose  $\text{CanRead}(x)$  does not hold for all  $x$ ). If we have a value  $M$  of this type we can conclude  $\text{CanRead}(M)$ . The type cannot be tainted, for if it were, we could conclude  $\text{CanRead}(M)$  for any  $M$  chosen by the opponent.

Dually, for a type that is tainted but not public, consider the function type  $T_2 \triangleq T_1 \rightarrow \text{Un}$ . Specializing the rules to come in Section 4, a function type  $T \rightarrow U$  is public when  $T$  is tainted and  $U$  is public, so  $T_2$  is not public. On the other hand, a function type  $T \rightarrow U$  is tainted when  $T$  is public and  $U$  is tainted, so  $T_2$  is tainted.

To see why it would be unsafe to consider the type  $T_2$  public, consider the function **fun**  $x \rightarrow$  **assert**  $\text{CanRead}(x)$ , which has type  $T_2$ . The assertion is safe because  $x$  has the refinement type  $T_1$ , and so the formula  $\text{CanRead}(x)$  holds. Intuitively, it should be safe to give any value of public type to the untyped opponent, but it is unsafe to do so with this function, because the opponent could apply it to some argument  $x$  that does not satisfy  $\text{CanRead}(x)$ .

To summarize, we have the strict inclusions  $T_1 <: \text{Un} <: T_2$ . This shows that not all types are public (consider  $T_2$ ), not all types are tainted (consider  $T_1$ ), hence not all types are equivalent to  $\text{Un}$ , and  $\text{Un}$  is not the top type (because  $T_2$  is not its subtype).

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

**THEOREM 2 (ROBUST SAFETY).** *If  $\emptyset \vdash A : \text{Un}$  then  $A$  is robustly safe.*

**PROOF.** See Appendix C.  $\square$

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.

(Some of our example interfaces include declarations of the form **private val**  $x_i : T_i$ . These declarations are available only within protocol code and are not exported to the attacker, and hence  $T_i$  is not necessarily public. We include these declarations for the sake of exposition, and also to inform our typechecker of the enhanced type  $T_i$ .)

### 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:  $x:\alpha \rightarrow (p:\alpha \text{ pickled})$ 
val unpickle:  $p:\alpha \text{ pickled} \rightarrow (x:\alpha)$ 
```

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  $\alpha$ hkey = HK of  $\alpha$ pickled Seal
type hmac = HMAC of Un
val mkHKey: unit  $\rightarrow \alpha$ hkey
val hmacsha1:  $k:\alpha$  hkey  $\rightarrow x:\alpha$  pickled  $\rightarrow h:hmac$ 
val hmacsha1Verify:  $k:\alpha$  hkey  $\rightarrow xx:Un$   $\rightarrow h:hmac \rightarrow x:\alpha$  pickled
```

where `hmac` is the type of hashes and  `$\alpha$ hkey` is the type of keys used to compute hashes for values of type  `$\alpha$` .

The function `mkHKey` generates 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  `$\alpha$` ) matches some given hash. If verification succeeds, this value is returned, now with the type  `$\alpha$`  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  `$\alpha$`  is the refinement type  `$x : \text{string}\{\text{CanRead}(x)\}$` . 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 post-condition 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 =  $x:\text{string}\{\text{Send}(x)\}$  // 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  $\rightarrow$  content  $\rightarrow$  message
val client: string  $\rightarrow$  unit // start a client
```

```
private val check: content hkey  $\rightarrow$  message  $\rightarrow$  content
val server: unit  $\rightarrow$  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, and hence are not available to the opponent. 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 precondition 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 runtime (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 runtime 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{  $\forall x. (e = \text{Recv}(x) \Rightarrow \text{Send}(x))$  }  $\rightarrow$ 
  r:unit{  $\forall x. (e = \text{Send}(x) \Rightarrow \text{Send}(x))$  }
```

The intended correspondence property  $\text{Recv}(x) \Rightarrow \text{Send}(x)$  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 shortly. 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  $(;a,b)\text{content}$  `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  $\text{Leak}(a)$  as recorded in its result type, then calls  $\text{getKey } a \ b$  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  $\text{Send}(a,b,x)$  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: Modeling Secrecy

Although this article 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 earlier.

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  $\text{let } s = (\text{fun } \_ \rightarrow \text{assert } C) \text{ in } A$  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  $\text{assert } C$ . 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  $s : \{C\} \rightarrow \text{unit} \vdash A : \text{Un}$ , then  $A$  preserves the secrecy of  $s$  unless  $C$ .*

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

By inspection of the rules for public kinding, we see that the type  $\{C\} \rightarrow \text{unit}$  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 message 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  $\text{Leak}(a)$ , a fact used next to illustrate the usage of assumptions for modeling key compromise.

We use the following declarations.

```

type empty = u:unit { Leak(a) }
type secret = { nonce:bytes; value:(empty  $\rightarrow$  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 before.

```

let s = { nonce = mkNonce();
         value = fun () → 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  $C = \exists a. (\text{Leak}(a) \wedge \text{Accept}(a))$  where  $\text{Leak}(a)$  records the compromise of a principal named  $a$  and  $\text{Accept}(a)$  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  $\lambda$ -calculus, and observe the close correspondence between sealing and various formal characterizations of symmetric-key cryptography.

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

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

```

type  $\alpha$ Seal = ( $\alpha \rightarrow \text{Un}$ ) * ( $\text{Un} \rightarrow \alpha$ )
val mkSeal: string →  $\alpha$ Seal

```

To implement a seal  $k$ , we maintain a list of pairs  $[(M_1, \alpha_1); \dots; (M_n, \alpha_n)]$ . The list records all the values  $M_i$  that have so far been sealed with  $k$ . Each  $\alpha_i$  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  $\alpha_i$  are pairwise distinct: the list is a one-to-one correspondence.

The function `mkSeal` that follows creates a fresh seal, by generating a fresh reference  $s$  that holds an empty list; the seal itself is the pair of functions  $(\text{seal } s, \text{unseal } s)$ . 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  $(\alpha \rightarrow \beta \text{ option}) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ option}$ , 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  $\alpha_i$  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  $\alpha$ SealRef = (( $\alpha$  * Un) list) ref
let seal:  $\alpha$ SealRef →  $\alpha$  → Un = fun s m →
  let state = !s in match List.first (List.left m) state with
  | Some(a) → a
  | None →
    let a: Un = Pi.name "a" in
    s := ((m,a)::state); a
let unseal:  $\alpha$ SealRef → Un →  $\alpha$  = fun s a →
  let state = !s in match List.first (List.right a) state with
  | Some(m) → m
  | None → failwith "not a sealed value"
let mkSeal (n:string) :  $\alpha$ Seal =
  let s = ref []:  $\alpha$  SealRef in
  (seal s, unseal s)
```

Irrespective of the type  $\alpha$  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 nondeterministic 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  $\pi$ -calculi [Gordon and Jeffrey 2003b; Fournet et al. 2007b] where all cryptographic functions were defined and typed as primitives, we can now implement these functions and retrieve their typing rules by typechecking their implementations.

An appendix to the technical report includes listings for the interface and the (typed) symbolic implementation of cryptography. We use seals to derive formal models for MACs (HMACSHA1), symmetric encryption (AES), asymmetric encryption (RSA), and

digital signatures (RSASHA1). For example, the functions that model HMACSHA1 are as follows.

```

let mkHKey () :  $\alpha$  hkey =
  let s = mkSeal "hkey" in
    HK s
let hmacsha1 (HK(key)) text =
  let (h,_) = key in
  let t = h text in
    HMAC (t)
let hmacsha1Verify (HK key) text (HMAC h) =
  let (_,hv) = key in
  let x :  $\alpha$  pickled = hv h in
    if x = text then x else failwith "hmac verify failed"

```

Keys are modeled as seals; computing and verifying MACs then correspond to uses of sealing and unsealing.

Following the same style, we model RSA encryption using the types and functions that follow.

```

type  $\beta$  dekey = DK of  $\beta$  symkey Seal
type  $\beta$  enkey = EK of ( $\beta$  symkey  $\rightarrow$  Un)
type penc = RSA of Un

```

```

let mkRsaDecKey () :  $\beta$  dekey =
  let s = mkSeal "rsaakey" in
    DK(s)
let rsaEncKey (DK dk) =
  let (e,d) = dk in EK(e)
let rsaEncrypt (EK (e)) t = RSA(e t)
let rsaDecrypt (DK k) (RSA msg) =
  let (e,d) = k in d msg

```

RSA decryption keys are modeled as seals. RSA encryption keys are public and can be derived from the corresponding decryption key. Encryption and decryption are modeled as sealing and unsealing.

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 $\nu$
$E \vdash T <: U$	in $E$ , type $T$ is a subtype of type $U$
$E \vdash A : T$	in $E$ , expression $A$ has type $T$

## Syntax of Kinds:

$v ::= \mathbf{pub} \mid \mathbf{tnt}$	kind (public or tainted)
Let $\bar{v}$ satisfy $\mathbf{pub} = \mathbf{tnt}$ and $\mathbf{tnt} = \mathbf{pub}$ .	

## Syntax of Typing Environments:

$\mu ::=$	environment entry
$\alpha$	type variable
$\alpha :: v$	kinding for recursive type $\alpha$
$\alpha <: \alpha'$	subtyping for recursive types $\alpha \neq \alpha'$
$a \downarrow T$	channel name
$x : T$	variable
$E ::= \mu_1, \dots, \mu_n$	environment
$dom(\alpha) = \{\alpha\}$	
$dom(\alpha :: v) = \{\alpha\}$	
$dom(\alpha <: \alpha') = \{\alpha, \alpha'\}$	
$dom(a \downarrow T) = \{\alpha\}$	
$dom(x : T) = \{x\}$	
$dom(\mu_1, \dots, \mu_n) = dom(\mu_1) \cup \dots \cup dom(\mu_n)$	
$recvar(E) = \{\alpha, \alpha' \mid (\alpha <: \alpha') \in E\} \cup \{\alpha \mid (\alpha :: v) \in E\}$	

If  $E = \mu_1, \dots, \mu_n$  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  $recvar(E) = \emptyset$ . Such environments contain names, variables, and type variables (but no entries  $\alpha :: v$  or  $\alpha <: \alpha'$ ). Let  $fnfv(E) = \bigcup \{fnfv(T) \mid (a \downarrow T) \in E \vee (x : T) \in E\}$ .

## Rules of Well-Formedness and Deduction:

(Env Empty)	(Env Entry)
$\frac{}{\emptyset \vdash \diamond}$	$\frac{}{E \vdash \diamond \quad fnfv(\mu) \subseteq dom(E) \quad dom(\mu) \cap dom(E) = \emptyset}$
(Type)	(Derive)
$\frac{}{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)} \quad E \vdash T$	$\frac{}{E \vdash \diamond \quad fnfv(C) \subseteq dom(E) \quad forms(E) \vdash C} \quad E \vdash C$
$forms(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup forms(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ forms(E_1) \cup forms(E_2) & \text{if } E = (E_1, E_2) \\ \emptyset & \text{otherwise} \end{cases}$	

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

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

Observe also that  $forms(E) = \emptyset$  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  $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$

(Kind Var) $\frac{E \vdash \diamond \ (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu}$	(Kind Unit) $\frac{E \vdash \diamond}{E \vdash \mathbf{unit} :: \nu}$	(Kind Fun) $\frac{E \vdash T :: \bar{\nu} \quad E, x : T \vdash U :: \nu}{E \vdash (\Pi x : T. U) :: \nu}$
(Kind Pair) $\frac{E \vdash T :: \nu \quad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T. U) :: \nu}$	(Kind Sum) $\frac{E \vdash T :: \nu \quad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}$	(Kind Rec) $\frac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu \alpha. T) :: \nu}$
(Kind Refine Public) $\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}}$	(Kind Refine Tainted) $\frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$	

The following rules for ok-types are derivable.

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

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

Subtype:  $E \vdash T <: U$

(Sub Refl) $\frac{E \vdash T \quad \text{recvar}(E) \cap \text{fnfv}(T) = \emptyset}{E \vdash T <: T}$	(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) $\frac{E \vdash \diamond \ (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$
(Sub Rec) $\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \text{fnfv}(T') \quad \alpha' \notin \text{fnfv}(T)}{E \vdash (\mu \alpha. T) <: (\mu \alpha'. T')}$	
(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  $\text{Un}$  is type equivalent to all types that are both public and tainted; we (arbitrarily) define  $\text{Un} \triangleq \mathbf{unit}$ . We can show that this definition satisfies the intended meaning: that  $T$  is public if and only if  $T$  is a subtype of  $\text{Un}$ , and that  $T$  is tainted if and only if  $T$  is a supertype of  $\text{Un}$ . (See Lemma 16 (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  $E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}$ .  $\square$

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:  $E \vdash M : T$

(Val Var) $\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$	(Val Unit) $\frac{}{E \vdash () : \text{unit}}$	(Val Fun) $\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun} \ x \rightarrow A : (\Pi x : T. U)}$
(Val Pair) $\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\Sigma x : T. U)}$	(Val Refine) $\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$	
(Val Inl Inr Fold) $\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h M : U}$		
inl:( $T, T+U$ )    inr:( $U, T+U$ )    fold:( $T\{\mu\alpha.T/\alpha\}, \mu\alpha.T$ )		

We can derive an introduction rule for ok-types.

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

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

Our final set of rules is for typing arbitrary expressions.

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

(Exp Subsum) $\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$	(Exp Appl) $\frac{E \vdash M : (\Pi x : T. U) \quad E \vdash N : T}{E \vdash M N : U\{N/x\}}$
(Exp Split) $\frac{E \vdash M : (\Sigma x : T. U) \quad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \quad \{x, y\} \cap \text{fv}(V) = \emptyset}{E \vdash \mathbf{let} \ (x, y) = M \ \mathbf{in} \ A : V}$	(Exp Match Inl Inr Fold) $\frac{E \vdash M : T \quad h : (H, T) \quad E, x : H, \_ : \{h x = M\} \vdash A : U \quad E, \_ : \{\forall x. h x \neq M\} \vdash B : U}{E \vdash \mathbf{match} \ M \ \mathbf{with} \ h \ x \rightarrow A \ \mathbf{else} \ B : U}$
(Exp Eq) $\frac{E \vdash M : T \quad E \vdash N : U \quad x \notin \text{fv}(M, N)}{E \vdash M = N : \{x : \text{bool} \mid (x = \mathbf{true} \wedge M = N) \vee (x = \mathbf{false} \wedge M \neq N)\}}$	
(Exp Assume) $\frac{E \vdash \diamond \quad \text{fnfv}(C) \subseteq \text{dom}(E)}{E \vdash \mathbf{assume} \ C : \{\_ : \text{unit} \mid C\}}$	(Exp Assert) $\frac{}{E \vdash \mathbf{assert} \ C : \text{unit}}$

$\frac{\text{(Exp Let)}}{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin \text{fv}(U)} \quad E \vdash \mathbf{let } x = A \mathbf{ in } B : U$	$\frac{\text{(Exp Res)}}{E, a \Downarrow T \vdash A : U \quad a \notin \text{fn}(U)} \quad E \vdash (va)A : U$
$\frac{\text{(Exp Send)}}{E \vdash M : T \quad (a \Downarrow T) \in E} \quad E \vdash a!M : \text{unit}$	$\frac{\text{(Exp Recv)}}{E \vdash \diamond (a \Downarrow T) \in E} \quad E \vdash a? : T$
$\frac{\text{(Exp Fork)}}{E, - : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, - : \{\overline{A_1}\} \vdash A_2 : T_2} \quad E \vdash (A_1 \uparrow A_2) : T_2$	

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  $A_1 \uparrow A_2$ , 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:  $\overline{A}$

$(va)\overline{A} = \exists a.\overline{A}$	$A_1 \uparrow A_2 = \overline{A_1} \wedge \overline{A_2}$	$\mathbf{let } x = A_1 \mathbf{ in } A_2 = \overline{A_1}$	$\mathbf{assume } C = C$
$\overline{A} = \text{True}$ if $A$ matches no other rule.			

## 5. IMPLEMENTING REFINEMENT TYPES FOR F<sup>#</sup>

We implement a typechecker, known as F7, that takes as input a series of extended RCF interface files and F<sup>#</sup> 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<sup>#</sup> 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 nonatomic 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<sup>#</sup> Language Features

Our typechecker processes F<sup>#</sup> 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. (An appendix in the technical report provides additional details on these codings.)

## 5.2. Annotating Standard Libraries

Any F<sup>#</sup> program may use the set of *pervasive* types and functions in the standard library; this library includes operations on built-in types such as strings, Booleans, lists, options, and references, and also provides system functions such as reading and writing files and pretty-printing. Hence, to check a program, we must provide the typechecker with declarations for all the standard library functions and types it uses. When the types for these functions are F<sup>#</sup> types, we can simply use the F<sup>#</sup> 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 retypecheck) these functions; for other library modules such as String and Printf, we trust the F<sup>#</sup> implementation.

## 5.3. Implementing Trusted Libraries

In addition to the standard library, our F<sup>#</sup> 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 follows.

```

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

```

Similarly, the network send and rcv 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

Table I. Typechecking Example Programs

	F <sup>#</sup> code (lines)	F <sup>#</sup> 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

subexpressions, our typechecker performs type inference using standard unification-based techniques for plain F<sup>#</sup> 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-encodings. Thus, constructors are coded as injective, uninterpreted, disjoint functions. Hence, for instance, a type definition for lists

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

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

**assume**  $\forall x,y. \text{Cons}(x,y) \neq \text{Nil}$ .

**assume**  $\forall x,y,x',y'.$

$(x = x' \wedge y = y') \Leftrightarrow \text{Cons}(x,y) = \text{Cons}(x',y')$ .

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 article and a few larger programs. Table I summarizes our results; for each example, it gives the number of lines of typed F<sup>#</sup> code, of generated F<sup>#</sup> interfaces, and of declarations in RCF interfaces, plus typechecking time, and the number of proof obligations passed to Z3. Since F<sup>#</sup> programmers are expected to write interfaces anyway, the line difference between RCF and F<sup>#</sup> 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 article 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 message
  | RequestBody of string // Payload for a request message
  | ResponseBody of string // Payload for a response message
  | Whatever of string // Any other elements
type signinfo = element list
type msg = signinfo * 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:signinfo {
  (vid, b.(Mem(IdHdr(id),x)  $\wedge$  Mem(RequestBody(b),x))
     $\Rightarrow$  Request(id,b) )

```

$$\wedge (\text{vid, req, b.}(\text{Mem}(\text{IdHdr}(\text{id}),\text{x}) \wedge \text{Mem}(\text{ResponseBody}(\text{b}),\text{x}) \\ \wedge \text{Mem}(\text{InReplyTo}(\text{req}),\text{x})) \Rightarrow \text{Response}(\text{id},\text{req},\text{b}) ) ) \}$$

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) the following code.

```
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 rege : element = RequestBody(payload) in
  let ls:element list = ide::rege::l1 in
  mkMessage ls

let mkRequest (id,payload) extra : msg =
  check_harmless extra;
  let ide: element = IdHdr(id) in
  let rege : element = RequestBody(payload) in
  let ls:element list = ide::rege::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)) )
```

```

^ (  $\forall$ s. not(Mem(InReplyTo(s),x)))
^ (  $\forall$ s. not(Mem(RequestBody(s),x)))
^ (  $\forall$ 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 of 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:signinfo{
  (  $\forall$ id, b, extra.( x = IdHdr(id)::RequestBody(b)::extra  $\Rightarrow$  Request(id,b) ))
^ (  $\forall$ id, req, b, extra.( x = IdHdr(id)::InReplyTo(req)::ResponseBody(b)::extra
   $\Rightarrow$  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. 2008a, 2008b] a subset of F# was translated into different variants of the applied  $\pi$ -calculus which could be verified by Blanchet's theorem provers ProVerif [Blanchet 2001] and CryptoVerif [Blanchet 2006] 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 a 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 post-conditions, rather than by developing a behavioral 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 runtime 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. 2010].

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, 2010; Kawaguchi et al. 2009], 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], and Fournet and Rezk [2008]).

These systems seek to guarantee noninterference 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  $\pi$ -calculus [Fournet et al. 2007b; Cirillo et al. 2007; Maffeis et al. 2008] and the  $\lambda$ -calculus [Jagadeesan et al. 2008] can guarantee that expressions follow their access

control policies. Type systems for variants of the  $\pi$ -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 runtime, and may be stored for later auditing; in contrast, we erase all formulas and discard proofs after typechecking.

Fable [Swamy et al. 2008] is a core formalism for expressing security policies; its type system does not in itself guarantee security properties, but additional proofs can build on type safety to establish properties including access control, information flow, and provenance. Fine [Swamy et al. 2010; Chen et al. 2010] is another refinement type system for  $F^\#$ , partly inspired by RCF; it extends the  $F^\#$  source language with dependent, refinement, and affine types that can be used to express and statically verify information flow and stateful authorization policies. Moreover, source programs typechecked with Fine can be compiled to proof-carrying code in a low-level intermediate language. To use the Fine typechecker and compiler, the programmer writes in an extended source language with extensive type annotations. In contrast, our typechecker works with pure  $F^\#$  programs with all annotations provided in an external RCF interface. Moreover, our verification case studies focus on the use of cryptography to enforce security policies, while the use of cryptographic primitives with Fine remains future work.

Beyond typechecking, many verification techniques also rely on checking logical properties of protocols, using, for instance, pre- and post-conditions in a protocol logic with domain-specific axioms [Durgin et al. 2003; Datta et al. 2007].

*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 multitier 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 multiparty 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. [2010b] extend the  $F7$  typechecker with support for implicit predicates representing the pre- and post-conditions of functions, in order to verify applications that use higher-order functions to perform cryptographic operations over recursive data structures, such as lists.
- Bhargavan et al. [2010a] 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. Their main motivation is to extend the scope of cryptographic verification by typing.

Their libraries rely on different, more logic-oriented design principles, and different types for the opponent. Kinds in RCF provide a simple, built-in mechanism for constraining types in opponent interfaces; they suffice to model many cryptographic primitives, but not all of those found in large case studies. Bhargavan et al. also rely on RCF and F7, but do not rely on kinds.

Instead, their approach involves developing (and assuming) a logical theory of symbolic cryptographic structures (including in particular a *Pub* predicate that represents the attacker knowledge). To verify a given protocol, one can then extend and customize the logical theory, rather than just declaring types.

Intuitively, instead of types with nested constructors subject to kinding, they mostly use refinement types of the form  $\{x : \text{bytes} \mid C\}$ , carrying logical specifications  $C$  subject to subtyping. Instead of the kinding judgments  $E \vdash T :: \mathbf{pub}$  and  $E \vdash T :: \mathbf{tnt}$ , they use subtyping judgments  $E \vdash T <: \{x : \text{bytes} \mid \text{Pub}(x)\}$  and  $E \vdash \{x : \text{bytes} \mid \text{Pub}(x)\} <: T$ , respectively, where *Pub* is an ordinary predicate of their theory.

Both sets of libraries are distributed with F7. Theirs yield a more general treatment of secrecy, and a flexible model for additional cryptographic patterns. For example, their libraries include weaker, unauthenticated encryption algorithms that one can use to build composite patterns such as hybrid encryption, which are not easily encodable with our libraries. Conversely, their libraries are more complex, and their usage sometimes requires hand proofs as well as typechecking.

—This article focuses on verifying security protocols written in RCF against a formal model of cryptography expressed with seals. Subsequent work by Backes et al. [2010] and by Fournet [2009] develops techniques for verifying RCF code against the computational model of cryptography. Additionally, Backes et al. show a formal correspondence between seals and a formal algebra in the style of Dolev and Yao [1983].

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.

## APPENDIXES

### A. LOGICS

Formally, RCF is parameterized by the choice of a 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 runtime equality in RCF and term equality in F#.

We first give an abstract definition of the logic used for the theorems, and then give a concrete definition of the logic used in the implementation. Other interesting instances of logics for our verification purposes include authorization 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]. Accordingly, we refer to our parametric logic as an authorization logic.

### 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 shortly and a *deducibility relation*  $S \vdash C$ , from finite multisets of formulas to formulas that meets the properties listed next.

Minimal Syntax of Formulas:

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

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

Properties of Deducibility:  $S \vdash C$

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

(Axiom)	(Mon)	(Subst)	(Cut)	(And Intro)	(And Elim)
$\frac{}{C \vdash C}$	$\frac{}{S \vdash C}$	$\frac{}{S \vdash C}$	$\frac{}{S \vdash C}$	$\frac{}{S \vdash C_0 \quad S \vdash C_1}$	$\frac{}{S \vdash C_0 \wedge C_1}$
$\frac{}{S, C' \vdash C}$	$\frac{}{S\sigma \vdash C\sigma}$	$\frac{}{S \vdash C'}$	$\frac{}{S \vdash C'}$	$\frac{}{S \vdash C_0 \wedge C_1}$	$\frac{}{S \vdash C_i}$
(Or Intro)	(Exists Intro)	(Exists Elim)	(Exists Elim)		
$\frac{}{S \vdash C_i} \quad i = 0, 1$	$\frac{}{S \vdash C\{M/x\}}$	$\frac{}{S \vdash \exists x.C}$	$\frac{}{S \vdash \exists x.C}$	$\frac{}{S \vdash C'}$	$\frac{}{S \vdash C'}$
(Eq)	(Ineq)	(Ineq Cons)	(Ineq Cons)		
$\frac{}{\emptyset \vdash M = M}$	$\frac{}{M \neq N \quad fv(M, N) = \emptyset}$	$\frac{}{h N = M \text{ for no } N \quad fv(M) = \emptyset}$	$\frac{}{h N = M \text{ for no } N \quad fv(M) = \emptyset}$		
	$\frac{}{\emptyset \vdash M \neq N}$	$\frac{}{\emptyset \vdash \forall x.hx \neq M}$	$\frac{}{\emptyset \vdash \forall x.hx \neq M}$		

We have a derived property (True)  $\emptyset \vdash \text{True}$ .

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 subexpressions, and use property (Subst) for establishing substitution lemmas. We also implicitly use (Subst) for handling the terms of RCF up to  $\alpha$ -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 28 ( $\Rightarrow$  Preserves Logic), to show that the formula  $\bar{A}$  extracted from an expression  $A$  is preserved by structural equivalence.

We use the properties (Eq), (Ineq), and (Or Intro) in the proof of Lemma 30 ( $\rightarrow$  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 **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  $\alpha$ -conversion, but not, for instance,  $\beta$ -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,  $\emptyset \vdash \mathbf{fun} x \rightarrow x = \mathbf{fun} y \rightarrow y$  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:

$$\begin{array}{l} t ::= x \mid f(t_1, \dots, t_n) \\ C ::= p(t_1, \dots, t_n) \mid (t = t') \mid \text{False} \mid C \wedge C' \mid C \vee C' \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C \\ \neg C \triangleq (C \Rightarrow \text{False}) \quad t \neq t' \triangleq \neg(t = t') \end{array}$$

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:  $S \vdash C$

$$\begin{array}{c} \text{(FOL Assume)} \quad \text{(FOL Refl)} \quad \text{(FOL Subst)} \\ \frac{C \in S}{S \vdash C} \quad \frac{}{S \vdash t = t} \quad \frac{S \vdash t = t' \quad S \vdash C\{t/x\}}{S \vdash C\{t'/x\}} \\ \text{(FOL And Intro)} \quad \text{(FOL And Elim)} \quad \text{(FOL Or Intro)} \\ \frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1} \quad \frac{S \vdash C_0 \wedge C_1}{S \vdash C_i} \quad \frac{S \vdash C_i}{S \vdash C_0 \vee C_1} \quad i = 0, 1 \\ \text{(FOL Or Elim)} \quad \text{(FOL False)} \quad \text{(FOL Classical)} \\ \frac{S \vdash C_0 \vee C_1 \quad S, C_0 \vdash C' \quad S, C_1 \vdash C'}{S \vdash C'} \quad \frac{}{S \vdash \text{False}} \quad \frac{}{S, \neg C \vdash C} \\ \text{(FOL Imply Intro)} \quad \text{(FOL Imply Elim)} \quad \text{(FOL All Intro)} \quad \text{(FOL All Elim)} \\ \frac{S, C \vdash C'}{S \vdash C \Rightarrow C'} \quad \frac{S \vdash C \Rightarrow C' \quad S \vdash C}{S \vdash C'} \quad \frac{S \vdash C \quad x \notin \text{fv}(S)}{S \vdash \forall x.C} \quad \frac{}{S \vdash \forall x.C} \\ \text{(FOL Exists Intro)} \quad \text{(FOL Exists Elim)} \\ \frac{S \vdash C\{t/x\}}{S \vdash \exists x.C} \quad \frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin \text{fv}(S, C')}{S \vdash C'} \end{array}$$

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  $\underline{M}$  is obtained from  $\underline{N}$  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  $p(\underline{M}_1, \dots, \underline{M}_n) = p(\underline{M}_1, \dots, \underline{M}_n)$  and  $\underline{M} = \underline{N} = \underline{M} = \underline{N}$ . We extend the translation to sets of formulas:  $\underline{S} = \{\underline{C}_1, \dots, \underline{C}_n\}$  when  $S = \{C_1, \dots, C_n\}$ .

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  $x_1 = y_1 \wedge \dots \wedge x_n = y_n$  where  $\vec{x}$  and  $\vec{y}$  are the lists  $x_1, \dots, x_n$  and  $y_1, \dots, y_n$ .)

#### Additional Rules for FOL/F:

(F Disjoint) $f \neq f'$ syntactic	(F Injective) $f$ syntactic
$S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) \neq f'(\vec{y})$	
$S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) = f(\vec{y}) \Rightarrow \vec{x} = \vec{y}$	

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 semidecidable 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:**

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

$$\mathcal{L} ::= \{\} \mid (\mathbf{let} \ x = \mathcal{L} \ \mathbf{in} \ B)$$

$$\mathbf{S} ::= (va_1) \dots (va_o) ((\prod_{i \in 1..m} \mathbf{assume} \ C_i) \uparrow (\prod_{j \in 1..n} c_j!M_j) \uparrow (\prod_{k \in 1..o} \mathcal{L}_k\{e_k\}))$$

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

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:  $A \Rightarrow A'$** 

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

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

LEMMA 1 (STRUCTURE).

For every expression  $A$ , there is a structure  $\mathbf{S}$  such that  $A \Rightarrow \mathbf{S}$ .

**Reduction:  $A \rightarrow A'$** 

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

<b>assert</b> $C \rightarrow ()$	(Red Assert)
<b>let</b> $x = M$ <b>in</b> $A \rightarrow A\{M/x\}$	(Red Let Val)
$A \rightarrow A' \Rightarrow$ <b>let</b> $x = A$ <b>in</b> $B \rightarrow$ <b>let</b> $x = A'$ <b>in</b> $B$	(Red Let)
$A \rightarrow A' \Rightarrow (va)A \rightarrow (va)A'$	(Red Res)
$A \rightarrow A' \Rightarrow (A \uparrow B) \rightarrow (A' \uparrow B)$	(Red Fork 1)
$A \rightarrow A' \Rightarrow (B \uparrow A) \rightarrow (B \uparrow A')$	(Red Fork 2)
$A \rightarrow A'$ if $A \Rightarrow B, B \rightarrow B', B' \Rightarrow A'$	(Red Heat)

---

**Expression Safety:**

An expression  $A$  is *safe* if and only if, for all  $A'$  and  $\mathbf{S}$ , if  $A \rightarrow^* A'$  and  $A' \Rightarrow \mathbf{S}$ , 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 15 (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 20 (Transitivity), that subtyping is transitive.
- Appendix C.4 presents an alternative characterization of the expression typing relation, avoiding the nonstructural 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 29 ( $\Rightarrow$  Preserves Types) and Proposition 31 ( $\rightarrow$  Preserves Types).
- Finally, Appendix C.7 establishes Theorem 2 (Robust Safety); the main additional lemma needed is Lemma 34 (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  $\mathcal{J}$  range over  $\{\diamond, T, C, T :: v, T <: T', A : T\}$ .

LEMMA 2 (DERIVED JUDGMENTS).

- (1) If  $E \vdash T$  then  $E \vdash \diamond$  and  $\text{fnfv}(T) \subseteq \text{dom}(E)$ .
- (2) If  $E \vdash C$  then  $E \vdash \diamond$  and  $\text{fnfv}(C) \subseteq \text{dom}(E)$ .
- (3) If  $E \vdash T :: v$  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  $\text{fnfv}(A) \subseteq \text{dom}(E)$ .

LEMMA 3 (TYPE VARIABLE STRENGTHENING).

Let entry  $\mu$  be one of  $\alpha, \alpha :: v'$ , or  $\alpha <: \alpha'$ , and assume that  $\text{dom}(\mu) \cap \text{fnfv}(E') = \emptyset$ .

- (1) If  $E, \mu, E' \vdash \diamond$  then  $E, E' \vdash \diamond$ .
- (2) If  $E, \mu, E' \vdash C$  and  $\text{dom}(\mu) \cap \text{fnfv}(C) = \emptyset$  then  $E, E' \vdash C$ .
- (3) If  $E, \mu, E' \vdash T :: v$  and  $\text{dom}(\mu) \cap \text{fnfv}(T) = \emptyset$  then  $E, E' \vdash T :: v$ .
- (4) If  $E, \mu, E' \vdash T <: T'$  and  $\text{dom}(\mu) \cap \text{fnfv}(T, T') = \emptyset$  then  $E, E' \vdash T <: T'$ .

PROOF. By an induction on the depth of derivation of  $E, \mu, E' \vdash \mathcal{J}$ , using property (Cut) of the logic.  $\square$

LEMMA 4 (ANON VARIABLE STRENGTHENING).

If  $E, - : \{C\}$ ,  $E' \vdash \mathcal{J}$  and  $\text{forms}(E, E') \vdash C$  then  $E, E' \vdash \mathcal{J}$ .

LEMMA 5 (EXCHANGE).

If  $E, \mu_1, \mu_2, E' \vdash \mathcal{J}$  and  $\text{dom}(\mu_1) \cap \text{fnfv}(\mu_2) = \emptyset$  then  $E, \mu_2, \mu_1, E' \vdash \mathcal{J}$ .

LEMMA 6 (WEAKENING).

If  $E, E' \vdash \mathcal{J}$  and  $E, \mu, E' \vdash \diamond$  then  $E, \mu, E' \vdash \mathcal{J}$ .

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 7 (KINDING).

If  $E \vdash T :: \mathbf{tnt}$ , and  $x \notin \text{dom}(E)$ , then  $\text{forms}(E) \vdash \text{forms}(x : T)$ .

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

LEMMA 8 (LOGICAL SUBTYPING).

If  $E \vdash T <: T'$  and  $x \notin \text{dom}(E)$  then  $\text{forms}(E), \text{forms}(x : T) \vdash \text{forms}(x : T')$ .

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 9 (BOUND WEAKENING).

Suppose that  $E \vdash T' <: T$ . If  $E, x : T, E' \vdash \mathcal{J}$  then  $E, x : T', E' \vdash \mathcal{J}$ . Moreover the depth of the derivation of the second judgment equals that of the first (except where  $\mathcal{J}$  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  $\{- : \text{unit} \mid C\}$ . The final lemmas in this section state some simple properties of ok-types.

LEMMA 10 (BOUND WEAKENING OK).

Suppose that  $E, C' \vdash C$ . If  $E, x : \{C\}, E' \vdash \mathcal{J}$  then  $E, x : \{C'\}, E' \vdash \mathcal{J}$ .

LEMMA 11 (SUB REFINE LEFT REFL).

If  $E \vdash \{x : T \mid C\}$  then  $E \vdash \{x : T \mid C\} <: T$ .

LEMMA 12 (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 13 (OK  $\wedge$ ).

We have  $E, - : \{C_1\}, - : \{C_2\}, E' \vdash \mathcal{J}$  if and only if  $E, - : \{C_1 \wedge C_2\}, E' \vdash \mathcal{J}$ .

## C.2. Properties of Kinding

We introduced in Section 3.2 a universal type  $\text{Un}$  of data known to the opponent. Lemma 16 (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  $\text{Un}$ , and a type is tainted if and only if it is a supertype of  $\text{Un}$ . The next two lemmas are needed in the proof of this main lemma.

LEMMA 14 (REPLACING TAINTED BOUNDS).

If  $E, x : T, E' \vdash U :: v$  and  $E \vdash T :: \mathbf{tnt}$  and  $E \vdash V$  then  $E, x : V, E' \vdash U :: v$ .

LEMMA 15 (PUBLIC DOWN/TAINTED UP).

Suppose that  $E$  is executable.

- (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 16 (PUBLIC TAINTED).  
For all  $T$  and executable  $E$ :

- (1)  $E \vdash T :: \mathbf{pub}$  if and only if  $E \vdash T <: Un$ .
- (2)  $E \vdash T :: \mathbf{tnt}$  if and only if  $E \vdash Un <: T$ .

### 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 17 (REC KINDING).  
If  $E \vdash T :: v$  and  $(\alpha <: \alpha') \in E$  then  $\alpha \notin \mathit{fnfv}(T)$  and  $\alpha' \notin \mathit{fnfv}(T)$ .

LEMMA 18 (REC SUBTYPING).  
If  $E \vdash T <: T'$  and  $(\alpha <: \alpha') \in E$  we have that:  $\{\alpha, \alpha'\} \cap \mathit{fnfv}(T) = \emptyset$  if and only if  $\{\alpha, \alpha'\} \cap \mathit{fnfv}(T') = \emptyset$ .

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:

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

LEMMA 19 (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)  $E, x : T, E' \vdash U :: v$  iff  $E, x : (T)^\sharp, E' \vdash U :: v$ .
- (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 20 (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 nonstructural rule (Val Refine), and hence is useful in the proofs of Lemma 23 (Substitution), Proposition 29 ( $\Rightarrow$  Preserves Types), and Proposition 31 ( $\rightarrow$  Preserves Types).

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

$$\frac{\begin{array}{c} \text{(Val Var Refine)} \\ E \vdash C\{x/y\} \quad (x : T) \in E \\ \hline E \vdash x : \{y : T \mid C\} \end{array}}{\begin{array}{c} \text{(Val Unit Refine)} \\ E \vdash C\{\()/y\} \\ \hline E \vdash () : \{y : \mathit{unit} \mid C\} \end{array}} \quad \frac{\begin{array}{c} \text{(Val Fun Refine)} \\ E, x : T \vdash A : U \quad E \vdash C\{\mathbf{fun} x \rightarrow A/y\} \\ \hline E \vdash \mathbf{fun} x \rightarrow A : \{y : (\Pi x : T. U) \mid C\} \end{array}}{}$$

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

LEMMA 21 (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 before together with the original rules for expressions.

LEMMA 22 (FORMULAS).

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

### 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  $x \notin \text{dom}(E)$ , let  $E\{M/x\}$  be the outcome of applying  $\{M/x\}$  to each type occurring in  $E$ . Similarly, if  $\alpha \notin \text{dom}(E)$ , 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:

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

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

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

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

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

LEMMA 23 (SUBSTITUTION).

- (1) If  $h : (T, U)$  then  $h : (T\{M/x\}, U\{M/x\})$ .
- (2) If  $x \notin \text{dom}(E)$  then  $\text{forms}(E)\{M/x\} = \text{forms}(E\{M/x\})$ .
- (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  $E, x : U, E' \vdash T :: v$  then  $E, (E'\{M/x\}) \vdash T\{M/x\} :: v$ .
  - 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 24.

If  $E, \alpha <: \alpha', E' \vdash T :: v$  then  $E, \alpha :: \mathbf{pub}, \alpha' :: \mathbf{tnt}, E' \vdash T :: v$ .

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

LEMMA 25 (TYPE SUBSTITUTION).

- (1) If  $E, \alpha, E' \vdash \mathcal{J}$  and  $E \vdash T$  and  $\text{recvar}(E) \cap \text{fnfv}(T) = \emptyset$  then  $E, (E'\{T/\alpha\}) \vdash \mathcal{J}\{T/\alpha\}$ .
- (2) If  $E, \alpha :: v, E' \vdash \diamond$  and  $E \vdash T :: v$  then  $E, (E'\{T/\alpha\}) \vdash \diamond$ .
- (3) If  $E, \alpha :: v, E' \vdash U$  and  $E \vdash T :: v$  then  $E, (E'\{T/\alpha\}) \vdash U\{T/\alpha\}$ .
- (4) If  $E, \alpha :: v, E' \vdash T' :: v'$  and  $E \vdash T :: v$  then  $E, (E'\{T/\alpha\}) \vdash T'\{T/\alpha\} :: v'$ .
- (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 26 (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  $\bar{A}$  is the set of formulas extracted from the expression  $A$ . For example,  $(\nu\alpha)(\text{assume } \text{Foo}(a, x) \bar{\vdash} \text{assume } \text{Bar}(z)) = \exists\alpha.(\text{Foo}(a, x) \wedge \text{Bar}(z))$ . 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 27.

If  $E \vdash A : T$  then  $E \vdash \{\bar{A}\}$ .

The next two lemmas assert that heating  $A \Rightarrow A'$  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 28 ( $\Rightarrow$  PRESERVES LOGIC).

If  $A \Rightarrow A'$  then  $\bar{A}' \vdash \bar{A}$ .

PROPOSITION 29 ( $\Rightarrow$  PRESERVES TYPES).

If  $E$  is executable,  $E \vdash A : T$ , and  $A \Rightarrow A'$ , then  $E \vdash A' : T$ .

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

LEMMA 30 ( $\rightarrow$  PRESERVES LOGIC).

If  $A \rightarrow A'$  then  $\bar{A}' \vdash \bar{A}$ .

PROPOSITION 31 ( $\rightarrow$  PRESERVES TYPES).

If  $E$  is executable,  $\text{fv}(A) = \emptyset$ ,  $E \vdash A : T$ , and  $A \rightarrow A'$ , then  $E \vdash A' : T$ .

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

LEMMA 32 (STATIC SAFETY).

If  $\emptyset \vdash \mathbf{S} : T$  then  $\mathbf{S}$  is statically safe.

RESTATEMENT OF THEOREM 1 (SAFETY)

If  $\emptyset \vdash A : T$  then  $A$  is safe.

PROOF. Consider any  $A'$  and  $\mathbf{S}$  such that  $A \rightarrow^* A'$  and  $A' \equiv \mathbf{S}$ ; it suffices to show that  $\mathbf{S}$  is statically safe. By Proposition 31 ( $\rightarrow$  Preserves Types),  $\emptyset \vdash A : T$  and  $A \rightarrow^* A'$  imply  $\emptyset \vdash A' : T$ . By Proposition 29 ( $\equiv$  Preserves Types), this and  $A' \equiv \mathbf{S}$  imply  $\emptyset \vdash \mathbf{S} : T$ . By Lemma 32 (Static Safety), this implies  $\mathbf{S}$  is statically safe.  $\square$

### C.7. Proof of Theorem 2 (Robust Safety)

First, we note that  $\text{Un}$  is type equivalent to a range of types.

LEMMA 33 (UNIVERSAL TYPE).

Given  $E \vdash \diamond$  we have  $E \vdash \text{Un} <:\!> T$  for each  $T$  that follows:

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

The next lemma establishes that any opponent can be well-typed using  $\text{Un}$  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 34 (OPPONENT TYPABILITY).

Suppose  $E \vdash \diamond$  and that  $E$  is executable. If  $O$  is an expression containing no **assert** such that  $(a \downarrow \text{Un}) \in E$  for each name  $a \in \text{fn}(O)$ , and  $(x : \text{Un}) \in E$  for each variable  $x \in \text{fv}(O)$ , then  $E \vdash O : \text{Un}$ .

Finally, we prove that robust safety follows by typing.

RESTATEMENT OF THEOREM 2 (ROBUST SAFETY)

If  $\emptyset \vdash A : \text{Un}$  then  $A$  is robustly safe.

PROOF. Consider any opponent  $O$  with  $\text{fn}(O) = \{a_1, \dots, a_n\}$ . We are to show the application  $O A$  is safe. Let  $E = a_1 \downarrow \text{Un}, \dots, a_n \downarrow \text{Un}$ . By Lemma 34 (Opponent Typability),  $E \vdash O : \text{Un}$ . By (Exp Subsum) and Lemma 33 (Universal Type),  $E \vdash O : (\Pi x : \text{Un}. \text{Un})$ . By Lemma 6 (Weakening),  $E \vdash A : \text{Un}$ . We can easily derive  $E \vdash \text{let } f = O \text{ in } (\text{let } x = A \text{ in } f x) : \text{Un}$ , that is,  $E \vdash O A : \text{Un}$ . Hence, we can derive  $\emptyset \vdash (\nu a_1) \dots (\nu a_n)(O A) : \text{Un}$ . By Theorem 1 (Safety),  $(\nu a_1) \dots (\nu a_n)(O A)$  is safe. Restriction does not affect safety, so it follows that  $O A$  is itself safe, as required.  $\square$

## ACKNOWLEDGMENTS

Discussions with B. Harper and D. Licata were useful. A. Askarov, M. Greenberg, and A. Nanevski commented on drafts of this article. K. Knowles suggested a proof technique. C. Hrițcu and M. Maffei reported an error in a proof and suggested the use of Lemma 14 (Replacing Tainted Bounds) in Appendix C.2. P.-Y. Strub reported an error in a proof in Appendix C.3. N. Bjørner and L. de Moura provided help with Z3.

## REFERENCES

- ABADI, M. 1999. Secrecy by typing in security protocols. *J. ACM* 46, 5, 749–786.
- ABADI, M. 2007. Access control in a core calculus of dependency. In *Computation, Meaning, and Logic: Articles Dedicated to Gordon Plotkin*, ENTCS, vol. 172. Elsevier, 5–31.
- ABADI, M. AND BLANCHET, B. 2005. Analyzing security protocols with secrecy types and logic programs. *J. ACM* 52, 1, 102–146.
- ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. 1993. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.* 15, 4, 706–734.
- ABADI, M. AND FOURNET, C. 2003. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Symposium (NDSS'03)*. Internet Society.

- ABADI, M. AND GORDON, A. D. 1999. A calculus for cryptographic protocols: The spi calculus. *Inform. Comput.* 148, 1–70.
- ABADI, M. AND NEEDHAM, R. 1996. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Engin.* 22, 1, 6–15.
- ASKAROV, A., HEDIN, D., AND SABELFELD, A. 2006. Cryptographically-masked flows. In *Proceedings of the Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 4134. Springer, 353–369.
- ASKAROV, A. AND SABELFELD, A. 2005. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'05)*. Lecture Notes in Computer Science, vol. 3679. Springer, 197–221.
- ASPINALL, D. AND COMPAGNONI, A. 2001. Subtyping dependent types. *Theor. Comput. Sci.* 266, 1–2, 273–309.
- AYDEMIR, B., CHARGÉRAUD, A., PIERCE, B. C., POLLACK, R., AND WEIRICH, S. 2008. Engineering formal metatheory. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM, 3–17.
- BACKES, M., GROCHULLA, M., HRIȚCU, C., AND MAFFEI, M. 2009. Achieving security despite compromise using zero-knowledge. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF'09)*. IEEE Computer Society, 308–323.
- BACKES, M., MAFFEI, M., AND UNRUH, D. 2010. Computationally sound verification of source code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM Press, 387–398.
- BALTOPOULOS, I. AND GORDON, A. D. 2009. Secure compilation of a multi-tier web language. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'09)*. 27–38.
- BARNETT, M., LEINO, M., AND SCHULTE, W. 2005. The Spec# programming system: An overview. In *Proceedings of the CASSIS'05*. Lecture Notes in Computer Science, vol. 3362. Springer, 49–69.
- BELLARE, M. AND ROGAWAY, P. 1993. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the ACM Conference on Computer and Communications Security*. 62–73.
- BENGTSON, J., BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2008. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*. 17–32.
- BENGTSON, J., BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2010. Refinement types for secure implementations. Tech. rep. MSR-TR-2008-118, Microsoft Research.
- BHARGAVAN, K., CORIN, R., DENIÉLOU, P.-M., FOURNET, C., AND LEIFER, J. J. 2009. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF'09)*. 124–140.
- BHARGAVAN, K., FOURNET, C., CORIN, R., AND ZALINESCU, E. 2008a. Cryptographically verified implementations for TLS. In *Proceedings of the ACM Conference on Computer and Communications Security*. 459–468.
- BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. 2010a. Modular verification of security protocol code by typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'10)*. ACM, 445–456.
- BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2008b. Verified interoperable implementations of security protocols. *ACM Trans. Program Lang. Syst.* 31, 5.
- BHARGAVAN, K., FOURNET, C., AND GUTS, N. 2010b. Typechecking higher-order security libraries. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'10)*. 47–62.
- BLANCHET, B. 2001. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW'01)*. 82–96.
- BLANCHET, B. 2006. A computationally sound mechanized prover for security protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 140–154.
- BLANCHET, B., ABADI, M., AND FOURNET, C. 2008. Automated verification of selected equivalences for security protocols. *J. Logic Algeb. Program.* 75, 1, 3–51.
- BORGSTRÖM, J., GORDON, A. D., AND PUCELLA, R. 2010. Roles, stacks, histories: A triple for Hoare. *J. Function. Program.* Cambridge University Press.
- CARDELLI, L. 1986. Typechecking dependent types and subtypes. In *Foundations of Logic and Functional Programming*. Lecture Notes in Computer Science, vol. 306. Springer, 45–57.
- CHAKI, S. AND DATTA, A. 2009. ASPIER: An automated framework for verifying security protocol implementations. In *Proceedings of the IEEE Computer Security Foundations Symposium*. 172–185.
- CHEN, J., CHUGH, R., AND SWAMY, N. 2010. Type-Preserving compilation for end-to-end verification of security enforcement. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, 412–423.

- CIRILLO, A., JAGADEESAN, R., PITCHER, C., AND RIELY, J. 2007. Do As I SaY! Programmatic access control with explicit identities. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF'07)*. 16–30.
- COK, D. R. AND KINIRY, J. 2004. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the CASSIS'05*. Lecture Notes in Computer Science, vol. 3362. Springer, 108–128.
- CONSTABLE, R., ALLEN, S., BROMLEY, H., CLEAVELAND, W., CREMER, J., HARPER, R., HOWE, D., KNOBLOCK, T., MENDLER, N., PANANGADEN, P., ET AL. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.
- COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. 2006. Links: Web Programming Without Tiers. In *Proceedings of 5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Lecture Notes in Computer Science. Springer-Verlag.
- COQUAND, T. AND HUET, G. 1988. The calculus of constructions. *Inform. Comput.* 76, 2-3, 95–120.
- DATTA, A., DEREK, A., MITCHELL, J. C., AND ROY, A. 2007. Protocol composition logic (PCL). In *Electronic Notes in Theoretical Computer Science* (Gordon D. Plotkin Festschrift), Vol. 172, Feb. 311–358.
- DE BRUJN, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 34, 381–392.
- DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Lecture Notes in Computer Science, vol. 4963. Springer, 337–340.
- DEAN, D., FELTEN, E., AND WALLACH, D. 1996. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*.
- DETLEFS, D., NELSON, G., AND SAXE, J. 2005. Simplify: A theorem prover for program checking. *J. ACM* 52, 3, 365–473.
- DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Trans. Inform. Theory* IT-29, 2, 198–208.
- DUMMETT, M. A. E. 1977. *Elements of Intuitionism*. Clarendon Press.
- DURGIN, N., MITCHELL, J. C., AND PAVLOVIC, D. 2003. A compositional logic for proving security properties of protocols. *J. Comput. Secur.* (Special Issue of Selected Papers from CSFW-14) 11, 4, 677–721.
- EASTLAKE, D., REAGLE, J., SOLO, D., BARTEL, M., BOYER, J., FOX, B., LAMACCHIA, B., AND SIMON, E. 2002. XML-signature syntax and processing. W3C Recommendation. <http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/>.
- FILLIÀTRE, J. AND MARCHÉ, C. 2004. Multi-prover Verification of C Programs. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'04)*. Lecture Notes in Computer Science, vol. 3308. Springer, 15–29.
- FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. *SIGPLAN Not.* 37, 5, 234–245.
- FOURNET, C. 2009. On the computational soundness of cryptographic verification by typing. In *Proceedings of the Workshop on Formal and Computational Cryptography (FCC'09)*.
- FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2007a. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 25.
- FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2007b. A type discipline for authorization policies in distributed systems. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*. 31–45.
- FOURNET, C. AND REZK, T. 2008. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. 323–335.
- FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'91)*. ACM, 268–277.
- GORDON, A. D. 1994. A mechanisation of name-carrying syntax up to alpha-conversion. In *Proceedings of the Conference on Higher Order Logic Theorem Proving and its Applications*, J. J. Joyce and C.-J. H. Seger, Eds. Lecture Notes in Computer Science, vol. 780. Springer, 414–426.
- GORDON, A. D. AND FOURNET, C. 2010. Principles and applications of refinement types. In *Logics and Languages for Reliability and Security: Proceedings of the NATO Summer School Marktoberdorf*, J. Esparza, B. Spanfelner, and O. Grumberg, Eds., IOS Press, 73–104.
- GORDON, A. D. AND JEFFREY, A. S. A. 2002. Cryptyc: Cryptographic protocol type checker. <http://cryptyc.cs.depaul.edu/>
- GORDON, A. D. AND JEFFREY, A. S. A. 2003a. Authenticity by typing for security protocols. *J. Comput. Secur.* 11, 4, 451–521.

- GORDON, A. D. AND JEFFREY, A. S. A. 2003b. Types and effects for asymmetric cryptographic protocols. *J. Comput. Secur.* 12, 3/4, 435–484.
- GORDON, A. D. AND JEFFREY, A. S. A. 2005. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proceedings of the CONCUR'05*. Lecture Notes in Computer Science, vol. 3653. Springer, 186–201.
- GOUBAULT-LARRECQ, J. AND PARRENNES, F. 2005. Cryptographic protocol analysis on real C code. In *Proceedings of the Conference on Verification Model-Checkior and Abstract Implementation (VMCAI'05)*. Lecture Notes in Computer Science, vol. 3385, Springer, 363–379.
- GRONSKI, J., KNOWLES, K., TOMB, A., FREUND, S. N., AND FLANAGAN, C. 2006. Sage: Hybrid checking for flexible specifications. In *Proceedings of the Scheme and Functional Programming Workshop*. R. Findler. Ed., 93–104.
- GUNTER, C. 1992. *Semantics of Programming Languages*. MIT Press.
- GUTS, N., FOURNET, C., AND ZAPPA NARDELLI, F. 2009. Reliable evidence: Auditability by typing. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*. Lecture Notes in Computer Science, Springer, 168–183.
- HUBBERS, E., OOSTDIJK, M., AND POLL, E. 2003. Implementing a formally verifiable security protocol in Java Card. In *Security in Pervasive Computing*, 213–226.
- JAGADEESAN, R., JEFFREY, A. S. A., PITCHER, C., AND RIELY, J. 2008. Lambda-RBAC: Programming with role-based access control. *Logical Methods Comput. Sci.* 4, 1.
- JIA, L., VAUGHAN, J., MAZURAK, K., ZHAO, J., ZARKO, L., SCHORR, J., AND ZDANCEWIC, S. 2008. Aura: A programming language for authorization and audit. In *Proceedings of the International Conference on Functional Programming (ICFP'08)*. ACM, 27–38.
- KAWAGUCHI, M., RONDON, P., AND JHALA, R. 2009. Type-based data structure verification. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, 304–315.
- LI, P. AND ZDANCEWIC, S. 2006. Encoding information flow in Haskell. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW'06)*. 16–27.
- MAFFEIS, S., ABADI, M., FOURNET, C., AND GORDON, A. D. 2008. Code-carrying authorization. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*. Lecture Notes in Computer Science, vol. 5283. Springer, 563–579.
- MARTIN-LÖF, P. 1984. *Intuitionistic Type Theory*. Bibliopolis.
- MORRIS, JR., J. H. 1973. Protection in programming languages. *Comm. ACM* 16, 1, 15–21.
- MYERS, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'99)*. 228–241.
- NADALIN, A., KALER, C., HALLAM-BAKER, P., AND MONZILLO, R. 2004. OASIS Web services security: SOAP message security 1.0. <http://www.oasis-open.org/committees/download.php/5941/oasis-200401-wss%-soap-message-security-1.0.pdf>
- NEEDHAM, R. AND SCHROEDER, M. 1978. Using encryption for authentication in large networks of computers. *Comm. ACM* 21, 12, 993–999.
- PARENT, C. 1995. Synthesizing proofs from programs in the calculus of inductive constructions. *Math. Program Construct.* 947, 351–379.
- PAULSON, L. C. 1987. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press.
- PAULSON, L. C. 1991. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science, vol. 828. Springer.
- PIERCE, B. AND SANGIORGI, D. 1996. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.* 6, 5, 409–454.
- POLL, E. AND SCHUBERT, A. 2007. Verifying an implementation of SSH. In *Proceedings of the Workshop on Information Technologies and Systems Meetings (WITS'07)*. 164–177.
- POTTIER, F. AND SIMONET, V. 2003. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1, 117–158.
- POTTIER, F., SKALKA, C., AND SMITH, S. 2001. A systematic approach to access control. In *Proceedings of the Conference on Programming Languages and Systems (ESOP'01)*. Lecture Notes in Computer Science, vol. 2028. Springer, 30–45.
- RÉGIS-GIANAS, Y. AND POTTIER, F. 2008. A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 5133. Springer, 305–335.
- RONDON, P., KAWAGUCHI, M., AND JHALA, R. 2008. Liquid types. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 159–169.

- RONDON, P., KAWAGUCHI, M., AND JHALA, R. 2010. Low-level liquid types. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'10)*. ACM, 131–144.
- RUSHBY, J., OWRE, S., AND SHANKAR, N. 1998. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Softw. Engin.* 24, 9, 709–720.
- SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation-passing style. *LISP Symb. Comput.* 6, 3-4, 289–360.
- SUMII, E. AND PIERCE, B. 2007. A bisimulation for dynamic sealing. *Theor. Comput. Sci.* 375, 1-3, 169–192.
- SWAMY, N., CHEN, J., AND CHUGH, R. 2010. Enforcing stateful authorization and information flow policies in Fine. In *Proceedings of the 19th European Symposium on Programming (ESOP'10)*. 529–549.
- SWAMY, N., CORCORAN, B. J., AND HICKS, M. 2008. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*. 96–110.
- SYME, D., GRANICZ, A., AND CISTERMINO, A. 2007. *Expert F#*. Apress.
- VAUGHAN, J. A., JIA, L., MAZURAK, K., AND ZDANCEWIC, S. 2008. Evidence-Based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*. 177–191.
- VAUGHAN, J. A. AND ZDANCEWIC, S. 2007. A cryptographic decentralized label model. In *Proceedings of the IEEE Symposium on Security and Privacy*. 192–206.
- WOO, T. AND LAM, S. 1993. A semantic model for authentication protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*. 178–194.
- XI, H. AND PFENNING, F. 1999. Dependent types in practical programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'99)*. ACM, 214–227.
- XU, D. N. 2006. Extended static checking for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell'06)*. ACM, 48–59.

Received February 2010; accepted May 2010