Modular Verification of Security Protocol Code by Typing

Karthikeyan Bhargavan Cédric Fournet
Andrew D. Gordon

Draft of December 2010

Technical Report

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

An abridged version of this paper appears in the proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, held in Madrid, Spain, on January 20-22, 2010.

Contents

Modular Verification of Security Protocol Code by Typing

Karthikeyan Bhargavan Cédric Fournet Andrew D. Gordon
Microsoft Research

Abstract

We propose a method for verifying the security of protocol implementations. Our method is based on declaring and enforcing invariants on the usage of cryptography. We develop cryptographic libraries that embed a logic model of their cryptographic structures and that specify preconditions and postconditions on their functions so as to maintain their invariants. We present a theory to justify the soundness of modular code verification via our method.

We implement the method for protocols coded in F# and verified using F7, our SMT-based typechecker for refinement types, that is, types carrying formulas to record invariants. As illustrated by a series of programming examples, our method can flexibly deal with a range of different cryptographic constructions and protocols.

We evaluate the method on a series of larger case studies of protocol code, previously checked using whole-program analyses based on ProVerif, a leading verifier for cryptographic protocols. Our results indicate that compositional verification by typechecking with refinement types is more scalable than the best domain-specific analysis currently available for cryptographic code.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques.

General Terms Security, Design, Languages.

1. Introduction

Verifying the Code of Cryptographic Protocols The problem of vulnerabilities in security protocol code is remarkably resistant to the success of formal methods. Consider, for example, the vulnerability in the public-key protocol of Needham and Schroeder (1978), first discovered by Lowe (1996) in his seminal paper on model-checking security protocols. This is the staple example of countless talks and papers on tools for analyzing security protocols. It is hence well known in the formal methods research community, and many tools can now discover it. In spite of these talks, papers, and tools, Cervesato et al. (2008) discovered that the IETF issued a public-key variant of Kerberos, shipped by multiple vendors, containing essentially the same vulnerability.

What to do? Our position is that formal tools are more likely to find such problems if they run directly on security protocol code. Most current tools require a model described in some formalism, such as a process algebra or a modal logic, but designers of new or revised protocols are resistant to writing such models. They are more concerned with functional properties like interoperability and so typically the first (and only) formal descriptions of protocol behaviour are the implementation code itself. Another reason to analyze code rather than models arises from gaps between the two: even if a model is verified, the corresponding code may deviate, and contain vulnerabilities absent from the model.

Several recent projects tackle the problem of verifying security protocol code. The pioneers are Goubault-Larrecq and Parrennes (2005) who use a tool to analyze C code (written in their group) for the Needham-Schroeder public key protocol. Another early tool is FS2PV (Bhargavan et al. 2008c), which compiles implementation code in F# into the applied pi calculus, for analysis with ProVerif (Blanchet 2001), a state-of-the-art domain-specific prover. In terms of lines of code analyzed, the combination of FS2PV and ProVerif is probably by now the leading tool chain for security protocol code. Several substantial case studies have yielded F# reference implementations that interoperate with existing implementations and are verified with FS2PV and ProVerif; these case studies include WS-Security (Bhargavan et al. 2006), CardSpace (Bhargavan et al. 2008b), and TLS (Bhargavan et al. 2008a).

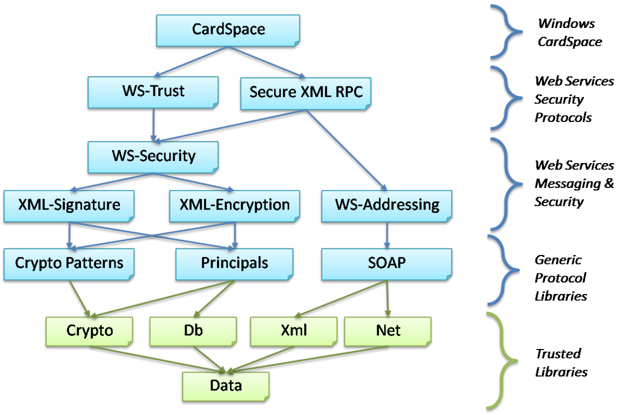

Figure 1. Modules for the Windows CardSpace Implementation

Towards Modular Verification It is challenging to verify security properties by compositional analysis. In particular, for systems involving cryptographic communication protocols, realistic attacker models tend to break modularity and abstraction: the attacker may interact at different layers in the protocol stack, for instance by injecting low-level network messages and controlling high-level actions at the application layer. Moreover, the attacker may compromise parts of the system, for instance gaining access to some cryptographic keys, and we are especially interested in the security properties that still hold in such situations. Accordingly, all protocol verification tools to date rely on high-complexity algorithms that operate on a complete description of the protocol.

The figure above presents the structure of our CardSpace implementation (our main case study), with one box for each F# module. Intuitively, the security properties for these modules are largely independent. Still, the earlier verification using FS2PV ignores this programming structure and passes a single, giant, untyped pi process to ProVerif. On the one hand, ProVerif scales surprisingly well: it often succeeds on input files orders of magnitude longer than the examples in its test suite. On the other, its whole-program analysis has long run times on large case studies such as CardSpace and TLS. Analysis may take hours, or diverge, and small changes in input files have unpredictable effects on run time.

In this paper, we aim for a modular and scalable technique that avoids whole-program analysis. We develop a new methodology, based on logical invariants for the cryptographic structures arising in security protocols. We show how to implement this methodology by typechecking with refinement types, and make several improvements to the existing typechecker F7 (Bengtson et al. 2008).

By proposing a new pattern of using F7 we intend that this paper may vindicate the promise of our initial work on refinement types for secure implementations, and establish that F7 supports scalable and flexible verification. It is flexible because we can formalize as wide a range of cryptographic operations as in FS2PV, for example. It is scalable because the time consuming part of analysis, automated theorem proving, is done compositionally by repeatedly calling an external solver on relatively small logical problems.

Our Method: Invariants for Cryptographic Structures As in the standard method originated by Dolev and Yao (1983), we model cryptographic structures as elements of a symbolic algebra. As in other logical approaches (for example, Paulson 1998, Cohen 2000, and Blanchet 2001), we rely on event predicates to record progress through a protocol and on a public predicate to indicate whether cryptographic structures are known to the adversary. For example, a byte array x is known to the adversary only if the predicate $Pub(x)$ holds. For an example of an event predicate, consider the simple protocol where a and b share a key k_{ab}, and a authenticates each message sent to b by sending also its hash keyed with k_{ab}. Then the *event predicate* $Send(a,b,x)$ holds only if a has started the protocol with the intention of sending message x to b.

The first key idea of our approach is to rely systematically on predicates to define invariants on cryptographic structures. For example, byte array x exists in a protocol run (whether or not it is public) only if the predicate $Bytes(x)$ holds. For another example, a key k_{ab} is shared between principals a and b for the purpose of running our example protocol only if the predicate $KeyAB(k_{ab},a,b)$ holds. Our definitions support deduction of useful properties of these invariants. For instance, in the simple case when all principals are uncompromised and comply with the protocol, our example predicates have the property that $Bytes(hash\ k_{ab}\ x)$ and $KeyAB(k_{ab},a,b)$ imply that $Send(a,b,x)$. This property captures the intuition that, if we can exhibit a byte array x that has been hashed with the key k_{ab}, which is known only to the protocol-compliant principals a and b, then it can only have been hashed by a, during a run of the protocol in which a intends to send x to b.

The second key idea is to rely on pre- and post-conditions on cryptographic algorithms to ensure that the actual code of a security protocol maintains these invariants. In our example, the precondition on applying the *hash* function to argument k_{ab} and x is the formula $KeyAB(k_{ab},a,b) \wedge Send(a,b,x)$, and as a postcondition, we obtain $Bytes(hash\ k_{ab}\ x)$. As a consequence of the implication stated above, we obtain $Send(a,b,x)$ as a postcondition of hash verification with a key satisfying $KeyAB(k_{ab},a,b)$.

We develop our invariants as a collection of predicates defined by axioms in first-order logic. The axioms form inductive definitions of our predicates; during automated code verification we rely on the axioms as well as additional formulas proved to hold in all reachable states. We use first-order logic because it is supported by a wide range of verification tools for a variety of languages.

Our theory is inspired by prior work on proving secrecy and authentication by using domain-specific type systems (Abadi 1999; Gordon and Jeffrey 2003a). Intuitively, the essence of these type systems is a collection of inductive definitions that define invariants preserved by computation. Our work can be understood, in part, as an extraction of this essence as direct inductive definitions of predicates, largely independent of the host language.

Scalable Verification by Typechecking with F7 We implement and evaluate our method for F#, a dialect of ML. We use F# for coding concrete implementations of protocols and libraries and also for specifying their security. Although most of the code is used for both purposes, some cryptographic libraries have *dual implementations*: one that performs concrete cryptographic computations, and one that operates instead on their symbolic representations.

We rely on the F7 typechecker, which verifies F# programs against types enhanced with logical refinements. A *refinement type* is a base type qualified with a logical formula; the formula can express invariants, preconditions, and postconditions. F7 relies on type annotations, including refinements, provided in specific interface files. While checking code, F7 generates many logical problems which it solves by submitting to Z3, an external theorem prover for first-order logic (de Moura and Bjørner 2008). Finally, F7 erases all refinements and yields ordinary F# modules and interfaces.

Our original paper on F7 (Bengtson et al. 2008) reported the underlying type theory, and a treatment of cryptography based on refinement types, public and tainted kinds (Gordon and Jeffrey 2003b), and seals (Morris 1973; Sumii and Pierce 2007). It proposed refinement types as a means for checking security properties in general; one example showed how to enforce access control by typing, others concerned a limited repertoire of cryptographic operations. The cryptographic library described in this paper is far more expressive.

We adopt F7 as a basis for implementing our method; refinement types are an excellent way to blend typechecking with verification. Still, although effective, both the theory of kinds and the use of seals necessarily depend on details of the host programming language. (Kinds are predicates on the syntax of types, and seals are λ-abstractions, only available in certain languages.) Therefore, we implement our new method, based on invariants for cryptographic structures, using F7 without seals and without the theory of kinds. (A detailed comparison of our method to the use of seals is outside the scope of this paper, but it appears that our method can flexibly model a wide range of cryptographic primitives, more so perhaps than can directly be modelled with seals.)

Another reason to choose F# is to enable a direct comparison with FS2PV and ProVerif, using previously-mentioned reference implementations for WS-Security and CardSpace. We develop our new method for cryptographic libraries that extend those already supported by FS2PV. Thus, we illustrate the flexibility of our method, and we can experimentally measure its performance versus ProVerif. Still, our method relies on user-supplied program invariants (within refinement types), while ProVerif can infer invariants. The previous F7 theory based on kinds and seals relied on a different cryptographic library, which did not allow a comparison with FS2PV code. To the best of our knowledge, the reference implementations checked with FS2PV and ProVerif are currently the most sizeable body of verified code for security protocols. So implementing our method for F# and the same libraries as used with FS2PV allows for a direct comparison against what is probably the state of the art.

Although we worked through the details of our approach in the setting of refinement types and F7, it is essentially language-independent. Hence, it should adapt easily to other settings, such as verification tools for imperative languages such as C.

Summary of Contributions

(1) A new modular method for verifying the code of security protocols, based on invariants for cryptographic structures.

(2) An implementation for the F# language by embedding invariants as refinement types, verified by the F7 typechecker. Typing relies on an external prover for logical entailments, and is compositional: the prover is called on a series of small problems.

(3) A collection of well-typed *refined modules* for cryptographic primitives and constructions, more expressive than in previous work with F7.

(4) Experimental evidence that typechecking is faster and succeeds on more protocol code than whole-program analysis with the leading automatic prover ProVerif.

In the long run, we expect the most scalable techniques for security protocol code to be those that can exploit progress in tools for proving general-purpose logical invariants. This is the specification style pioneered by Floyd, Hoare, and Dijkstra in the 1970s. Tools for enforcing and even inferring invariants in code are likely to get better and better over time.

Structure of the Paper Section 2 reviews RCF. Section 3 introduces our method of invariants for cryptographic structures and our typed cryptographic library by studying a simple RPC protocol. Section 4 provides a theory of *refined modules* to justify proofs of security by typing implementation code. Section 5 gives some detailed examples of refined modules for cryptography. Section 6 outlines our more substantial case studies. Section 7 evaluates the performance of our implementation by comparison with a tool chain based on whole-program analysis. Section 8 discusses related work and Section 9 concludes.

Appendix A lists and explains the typed interface for our library of cryptographic primitives. Appendix C recalls the formal definition of RCF, the theoretical foundation for F7.

Source code for our libraries and examples is available online at `http://research.microsoft.com/en-us/projects/f7/`.

2. RCF, the Formal Foundation for F7 (Review)

We begin with a review of the syntax and semantics of RCF (Bengtson et al. 2008), our core language for F#. RCF consists of the standard Fixpoint Calculus (Gunter 1992; Plotkin 1985) augmented with local names and message-passing concurrency (as in the pi calculus) and with refinement types. Formally, we slightly simplify the original calculus by omitting the use of public and tainted kinds. For a detailed tutorial presentation of RCF, see Gordon and Fournet (2009).

We state some syntactic conventions. Our phrases of syntax may contain three kinds of identifier: type variables α, value variables x, and names a. We identify phrases of syntax up to consistent renaming of bound identifiers. We write $\psi\{\phi/\iota\}$ for the capture-avoiding substitution of the phrase ϕ for each free occurrence of identifier ι in the phrase ψ. We say a phrase is *closed* to mean that it has no free type or value variables (although it may contain free names).

Expressions and types of RCF contain formulas C to specify intended properties. Specification formulas are written in first-order logic with equality, with *atomic formulas*, $p(M_1,\dots,M_n)$, built from a fixed set of predicate symbols p applied to RCF values.

Syntax of FOL/F Formulas:

$$C ::= p(M_1,\dots,M_n) \mid (M = M') \mid (M \neq M') \mid \mathit{False} \mid \mathit{True} \mid C \wedge C' \mid C \vee C' \mid C \Rightarrow C' \mid \neg C \mid C \Leftrightarrow C' \mid \forall x.C \mid \exists x.C$$

(This is the logic FOL/F of Bengtson et al. 2008.)

We recall standard definitions for (untyped) first-order logic with equality (see Paulson 2008 for example). An *interpretation* $\mathscr{I}$ is a pair (D,I) where D is a set, the *domain*, and I is an operation that maps function symbols to functions on D and predicate symbols to relations on D. A *valuation* V is a function from variables into D. An interpretation $\mathscr{I}$ *satisfies* a closed formula C, written $\models_{\mathscr{I}} C$ when, for all valuations V, we have $\models_{\mathscr{I},V} C$, which is defined by structural induction on C, following Tarski. A closed formula C is *valid* if all interpretations satisfy the formula.

We are only concerned with *RCF-interpretations*, that is, interpretations (D,I) where D is the set of closed phrases of RCF and I maps each function symbol f of arity n to the function $M_1,\dots,M_n \mapsto f(M_1,\dots,M_n)$, and maps the equality predicate to syntactic equality. (The only function symbols in our formulas are the syntactic constructors of RCF. In an RCF-interpretation (D,I) we fix the meaning of function symbols and equality, but allow the meaning of predicates to vary.)

For a given proof system, we write $C_1,\dots,C_n \vdash C$ when C can be deduced from $C_1,\dots,C_n$. We say that the proof system is *sound* when, for all formulas $C_1,\dots,C_n$ and C with free variables $x_1,\dots,x_k$, if $C_1,\dots,C_n \vdash C$, then $\forall x_1.\dots.\forall x_k.(C_1 \wedge \cdots \wedge C_n \Rightarrow C)$ is valid. In the following, we rely on a standard, sound proof system for first-order logic, as implemented by Z3.

Core Syntax of the Values and Expressions of RCF:

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

Much of RCF is standard functional notation. Expressions are in the style of A-normal form; let-expressions are for sequencing and not for polymorphism. In the style of the pi calculus, RCF includes restriction (name generation), fork, and message transmission and reception for communication and concurrency. Names range over countable, pairwise-distinct constants, used to represent channels, fresh values, and keys, for instance. RCF does not have names as primitive values, but we encode them as functional values with free names. For example, a as a pure name is coded as **fun** $\_ \to a?$.

An *expression context* X is an expression with a hole '_'. We write $X[A]$ for the outcome of filling the hole with expression or expression context A, where variables free in A may be bound by binders in X. (We use expression contexts to represent modules.)

The expressions **assume** and **assert** have no observable effect at run-time, and are used only to specify logic-based safety properties. Execution of **assume** C limits attention to logical interpretations in which C holds. Assumptions are used to state inductive definitions or to record events, for example. Execution of **assert** C indicates an error unless C holds in interpretations satisfying the previously executed assumptions.

The type system of RCF is based on FPC, but with dependent function and pair types, plus *refinement types* $x : T\{C\}$. The values of this type are the values M of type T such that $C\{M/x\}$ holds.

Core Syntax of Types of RCF:

$T,U,V ::=$	type
unit	unit type

$x : T \to U$	dependent function type (scope of x is U)
$x : T * U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mathbf{rec}\ \alpha.T$	iso-recursive type (scope of α is T)
α	type variable (abstract or iso-recursive)
$x : T\{C\}$	refinement type (scope of x is C)

As detailed by Bengtson et al. (2008), RCF supports standard encodings of a wide range of F# programming constructs, including let-polymorphism (eliminated by code duplication), mutable references (channels), and algebraic types (recursive sums of product types); it is closely related to the internal language of the F7 typechecker. Our code examples rely on these encodings.

In addition, code written in RCF has access to a few pre-defined trusted libraries, depicted at the bottom of Figure 1. The library module **Data** defines standard datatypes such as strings, byte arrays, lists, options, and provides functions for manipulating and converting between values of these types; **Crypto** provides primitive cryptographic operations; **Db** provides functions for storing and retrieving values from a global, shared, secure database; **Xml** provides functions and datatypes for manipulating XML documents; **Net** provides functions for establishing TCP connections and exchanging messages over them. We write **Lib** for the composition of **Data**, **Net**, and **Crypto**, and **LibX** for the composition of **Lib**, **Db**, and **Xml**. These libraries are trusted in the sense that their concrete implementations are not verified. Instead, we define idealized symbolic implementations, in the style of Dolev and Yao (1983), for each of these five modules and show that they meet their typed RCF interfaces.

Each judgment of the RCF type system is given relative to an *environment*, E, which is a sequence $\mu_1, \ldots, \mu_n$, where each μ_i may be a *subtype assumption* $\alpha <: \alpha'$, an *abstract type* α, or an entry for a name $a \updownarrow T$ or a variable $x : T$. We write $E \vdash T$ to mean that type T is well-formed in E, and $E \vdash \diamond$ to mean that E is well-formed (which implies that all types in E are well-formed). The two main judgments are *subtyping*, $E \vdash T <: U$, and *type assignment*, $E \vdash A : T$. The full rules for these judgments and the rest of RCF are in Appendix C.

F7 checks type assignment, where the expression A is obtained from an F# source file, and the type T is obtained from an F7-specific interface file.

F7 relies on various type inference algorithms, and calls out to Z3 to handle the logical goals that arise when checking refinements. F7 adds the formula C to the current logical environment when processing **assume** C, and conversely checks that formula C is provable when processing **assert** C.

In Section 4 we discuss the operational semantics, safety properties, and theorems for proving safety by typing, but first we introduce our new method by example.

3. Invariants for Authenticated RPCs (Example)

We consider a protocol intended to authenticate remote procedure calls (RPC) over a TCP connection. We first informally discuss the security of this protocol and identify a series of underlying assumptions. We then explain how to formalize these assumptions, and how to verify an implementation of the protocol.

3.1 Informal Description

We have a population of principals, ranged over by a and b. The security goals of our RPC protocol are that (1) whenever a principal b accepts a request message s from a, principal a has indeed sent the message to b and, conversely, (2) whenever a accepts a response message t from b, principal b has indeed sent the message in response to a matching request from a.

To this end, the protocol uses message authentication codes (MACs) computed as keyed hashes, such that each symmetric MAC key k_{ab} is associated with (and known to) the pair of principals a and b. Our protocol may be informally described as follows.

An Authenticated RPC Protocol:

1. $a \to b :\ \mathit{utf8}\ s \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{request}\ s))$
2. $b \to a :\ \mathit{utf8}\ t \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{response}\ s\ t))$

In this protocol narration, each line indicates the communication of data from one principal to another. This data is built using five functions: *utf8* marshals the strings s and t into byte arrays (the message payloads); *request* and *response* build message digests (the authenticated values); *hmacsha1* computes keyed hashes of these values (the MACs); and '|' concatenates the message parts.

We consider systems in which there are multiple concurrent RPCs between any principals a and b of the population. The adversary controls the network. Some keys may also become compromised, that is, fall under the control of the adversary. Intuitively, the security of the protocol depends on the following assumptions:

(1) The function *hmacsha1* is cryptographically secure, so that MACs cannot be forged without knowing their key.

(2) The principals a and b are not compromised—otherwise the adversary may just use k_{ab} to form MACs.

(3) The functions *request* and *response* are injective and their ranges are disjoint—otherwise, an adversary may for instance replace the first message payload with *utf8* s' for some $s' \neq s$ such that *request* s' = *request* s and thus get s' accepted instead of s, or use a request MAC to fake a response message.

(4) The key k_{ab} is a genuine MAC key shared between a and b, used exclusively for building and checking MACs for requests from a to b and responses from b to a—otherwise, for instance, if b also uses k_{ab} for authenticating requests from b to a, it would accept its own reflected messages as valid requests from a.

These assumptions can be precisely expressed (and verified) as *program invariants* of the protocol implementation. Moreover, the abstract specification of *hmacsha1*, *request*, and *response* given above should suffice to establish the protocol invariant, irrespective of their implementation details.

3.2 Adding Events and Assertions

We use event predicates to record the main steps of each run of the protocol, to record the association between keys and principals, and to record principal compromise. To mark an event in code, we assume a corresponding logical fact:

- $\mathit{Request}(a,b,s)$ before a sends message 1;
- $\mathit{Response}(a,b,s,t)$ before b sends message 2;
- $\mathit{KeyAB}(k,a,b)$ before issuing a key k associated with a and b;
- $\mathit{Bad}(a)$ before leaking any key associated with a.

We state each intended security goal in terms of these events, by asserting that a logical formula always holds at a given location in our code, in any system configuration, and despite the presence of an active adversary. In our protocol, we assert:

- $\mathit{RecvRequest}(a,b,s)$ after b accepts message 1;
- $\mathit{RecvResponse}(a,b,s,t)$ after a accepts message 2;

where the predicates *RecvRequest* and *RecvResponse* are defined by the two formulas:

$$\forall a,b,s.\ \mathit{RecvRequest}(a,b,s) \Leftrightarrow (\mathit{Request}(a,b,s) \lor \mathit{Bad}(a) \lor \mathit{Bad}(b))$$

$$\forall a,b,s,t.\ RecvResponse(a,b,s,t) \Leftrightarrow$$
$$(Request(a,b,s) \wedge Response(a,b,s,t)) \vee Bad(a) \vee Bad(b)$$

The disjunctions above account for the potential compromise of either of the two principals with access to the MAC key; the disjunctions would not appear with a simpler (weaker) attacker model.

3.3 Implementing the RPC Protocol

We give below an implementation for the two roles of our protocol, coded in F#. Except for protocol narrations, all the code displayed in this paper is extracted from F7 interfaces and F# implementations that have been typechecked.

Code for the Authenticated RPC Protocol:

```
let mkKeyAB a b = let k = hmac_keygen() in assume (KeyAB(k,a,b)); k
let request s = concat (utf8(str "Request")) (utf8 s)
let response s t = concat (utf8(str "Response")) (concat (utf8 s) (utf8 t))

let client (a:str) (b:str) (k:keyab) (s:str) =
  assume (Request(a,b,s));
  let c = Net.connect p in
  let mac = hmacsha1 k (request s) in
  Net.send c (concat (utf8 s) mac);
  let (pload',mac') = iconcat (Net.recv c) in
  let t = iutf8 pload' in
  hmacsha1Verify k (response s t) mac';
  assert(RecvResponse(a,b,s,t))

let server(a:str) (b:str) (k:keyab) : unit =
  let c = Net.listen p in
  let (pload,mac) = iconcat (Net.recv c) in
  let s = iutf8 pload in
  hmacsha1Verify k (request s) mac;
  assert(RecvRequest(a,b,s));
  let t = service s in
  assume (Response(a,b,s,t));
  let mac' = hmacsha1 k (response s t) in
  Net.send c (concat (utf8 t) mac')
```

(We omit the definition of the application-level *service* function.) Compared to the protocol narration, the code details message processing, and in particular the series of checks performed when receiving messages. For example, upon receiving a request, *server* extracts *s* from its encoded payload by calling *iutf8*, and then verifies that the received MAC matches the MAC recomputed from *k* and *s*. The code uses *concat* and *iconcat* to concatenate and split byte arrays. (Crucially for this protocol, *concat* embeds the length of the first array, and *iconcat* splits arrays at this length. Otherwise, for instance, *response* is not injective and the protocol is insecure.)

In our example, the code assumes events that mark the generation of a key for our protocol and the intents to send a request from *a* to *b* or a response from *b* to *a*. The code asserts two properties, after receiving a request or a response, and accepting it as genuine.

We test that our code is functionally correct by linking it to a concrete cryptographic library and performing an RPC between *a* and *b*. The messages exchanged over TCP are:

```
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYS0GGtOtPm...} (23 bytes)
Received Response 4
```

3.4 Modelling the Opponent

We model an opponent as an arbitrary program with access to a given *public interface* that reflects all its (potential) capabilities. Thus, our opponent has access to the network (modelling an active adversary), to the cryptographic library (modelling access to the MAC algorithms), and to a protocol-specific *setup* function that creates new instances of the protocol for a given pair of principals. This function returns four capabilities: to run the client with some payload, to run the server, to corrupt the client, and to corrupt the server (that is, here, to get their key). We detail the code for *setup* below: it allocates a key, specializes our client and server functions, and leaks that key upon request after assuming an event that records the compromise of either *a* or *b*.

Protocol-Specific Implementation for the Opponent Interface:

```
let setup (a:str) (b:str) =
  let k = mkKeyAB a b in
  (fun s → client a b k s),
  (fun _ → server a b k),
  (fun _ → assume (Bad(a)); k),
  (fun _ → assume (Bad(b)); k)
```

Formally, the opponent ranges over arbitrary F# code well-typed against an interface that includes (at least) the declarations below. (Demanding that the opponent be well-typed is innocuous as long as the interface only operates on plain types such as bitstrings.) Let an *opponent O* be an expression containing no **assume** or **assert**. Our opponent interfaces declare functions that operate on types of the form $x{:}T\ \{\ Pub(x)\}$; intuitively, these types reflect the global invariant that the opponent may obtain and construct at most the cryptographic values tracked as public in our logic model. Hence, bytespub is defined as x: bytes $\{Pub(x)\}$. The types strpub and keypub of public strings and public keys are defined similarly.

In our method, we explicitly give an inductive definition of *Pub*, and the typechecker ensures that, whenever an expression is given a public type (for instance when sending bytes on a public network), the fact that the value will indeed be public logically follows from that inductive definition.

Opponent Interface (excerpts):

```
type port = A of string ∗ string
type conn = C of string

val http: string → string → port
val connect: port → conn
val listen: port → conn
val close: conn → unit

val send: conn → bytespub → unit
val recv: conn → bytespub
val hmacsha1 : keypub → bytespub → bytespub
val hmacsha1Verify : keypub → bytespub → bytespub → unit
val setup: strpub → strpub →
  (strpub → unit) ∗ (unit → unit) ∗ (unit → keypub) ∗ (unit → keypub)
```

(The adversary is not given access to the key-generating function *hmac_keygen* because it can directly build public keys from public bytes.)

As explained next, we write more refined interfaces for typechecking our code: each value declaration will be given a refined type that is a subtype of the one listed in the opponent interface.

We are now ready to formally state our target security theorem for this protocol. We say that an expression is *semantically safe* when every executed assertion logically follows from previously-executed assumptions. Let I_L be the opponent interface for our library (introduced precisely in Appendix A.9). Let I_R be the opponent interface for our protocol (the *setup* function displayed above). Let X be the expression context representing the composition of the library with the protocol implementation. (We give a precise definition of X in Section 4.6.)

THEOREM 1 *For any opponent O, if* $I_L, I_R \vdash O$: unit*, then* $X[O]$ *is semantically safe.*

With the specification of events and formulas given in Section 3.2, semantics safety for the RPC protocol entails in particular two protocol-verification *correspondence properties* (Gollmann 2003) between "end" events marking message accepts (the *RecvRequest* and *RecvResponse* assertions) and "begin" events marking message sends (the *Request* and *Response* assumptions).

3.5 Refinement-Typed Interface for MACs

Our example theorem relies on typechecking our library and protocol code against their opponent interfaces. For the library, this is done once for all, using an intermediate, more refined interface that operates on values that are not necessarily public. This interface and its logical model are explained in Appendix A, so here we only outline their declarations and formulas as regards MACs. So the main task for verifying the RPC protocol is to typecheck it.

We first outline the refined interface for MACs, then explain how to define and enforce a logical model for the RPC protocol.

Refinement Types for MACs (from the *Crypto* library):

val *hmac_keygen*: unit $\rightarrow k$:key$\{\mathit{MKey}(k)\}$
val *hmacsha1*:
  k:key $\rightarrow$
  b:bytes$\{\ (\mathit{MKey}(k) \wedge \mathit{MACSays}(k,b)) \vee (\mathit{Pub}(k) \wedge \mathit{Pub}(b))\ \} \rightarrow$
  h:bytes$\{\ \mathit{IsMAC}(h,k,b) \wedge (\mathit{Pub}(b) \Rightarrow \mathit{Pub}(h))\ \}$
val *hmacsha1Verify*:
  k:key$\{\mathit{MKey}(k) \vee \mathit{Pub}(k)\} \rightarrow b$:bytes $\rightarrow h$:bytes $\rightarrow$ unit$\{\mathit{IsMAC}(h,k,b)\}$

(C1. By expanding the definition of IsMAC)
$\forall h,k,b.\ \mathit{IsMAC}(h,k,b) \wedge \mathit{Bytes}(h) \Rightarrow (\mathit{MKey}(k) \Rightarrow \mathit{MACSays}(k,b)) \vee \mathit{Pub}(k)$
(C2. MAC keys are public iff they may be used with any logical payload)
$\forall k.\ \mathit{MKey}(k) \Rightarrow (\mathit{Pub}(k) \Leftrightarrow \forall m.\ \mathit{MACSays}(k,m))$

This interface defines functions for creating keys, computing MACs, and verifying them. (The **private** modifier indicates that a value is not included in the opponent interface.) It is designed for flexibility; simpler, more restrictive interfaces may be obtained by subtyping, for instance, when key compromise need not be considered. Its logical model is built from the following predicates:

- *MKey*(k) records that k has been produced by *hmac_keygen*; the adversary can produce other public keys from public values.
- *MACSays*(k,b) is defined by the protocol that relies on k, as its precondition for computing a MAC and its postcondition after verifying a MAC. Intuitively, this predicate represents the logical payload of MACs with key k.
- *IsMAC*(h,k,b) holds when verification that h is a MAC for b under k succeeds; it implies either $\mathit{MACSays}(k,b)$ or $\mathit{Pub}(k)$.

The precondition of *hmacsha1* is a disjunction that covers two cases for the key: either it is a correctly-generated key, or the key is public. The latter case is necessary to type MAC computations using a key received from the opponent, and to show that *hmacsha1* has the type declared in the opponent interface. (In type systems without formulas, such disjunctions in logical refinements could instead be expressed using union types.) The postcondition $\mathit{Pub}(b) \Rightarrow \mathit{Pub}(h)$ states that the MACs produced by the protocol are public (hence can be sent) provided the plaintext is public. Cryptographically, this reflects that MACs provide payload authentication but not secrecy.

The precondition of *hmacsha1Verify* similarly covers the two cases for the key. A call *hmacsha1Verify k b h* raises an exception in case the supplied hash h does not in fact match the MAC of b with the key k. (At present, F7 does not support exception handling, and treats an exception as terminating execution.) Otherwise, its postcondition also leads to a disjunction (corollary C1), so the protocol that verifies a MAC must also know that $\mathit{Pub}(k) \Rightarrow \mathit{MACSays}(k,b)$, for example because k is not public, to deduce that $\mathit{MACSays}(k,b)$.

The library also assumes definitions and theorems relating these predicates, and in particular the inductive definition of *Pub*. For convenience, the display above includes two properties for MACs that are corollaries of these definitions: C1 just inlines the definition of *IsMAC*; C2 expresses a *secrecy invariant* for MAC keys: a key k is public if and only if its associated logical payload holds for any value. Hence, as a prerequisite for releasing a key k as a public value, a protocol must ensure that all potential consequences of MAC verification with key k hold. Depending on how the protocol defines *MACSays*, this may be established by assuming some compromise at the protocol level (predicate $\mathit{Bad}(a)$ in our protocol).

3.6 Logical Invariants for the RPC Protocol

To verify a protocol, we state some of its intended logical properties (both defining its specific usage of cryptography and stating theorems about it), we typecheck the protocol code under those assumptions, and, if need be, we prove protocol-specific theorems, as illustrated below.

We first introduce two auxiliary predicates for the payload formats: *Requested* and *Responded* are the (typechecked) postconditions of the functions *request* and *response*; we omit their definition. Typechecking involves the automatic verification that our formatting functions are injective and have disjoint ranges, as explained in informal assumption (3). Verification is triggered by asserting the formulas below, so that Z3 proves them.

Properties of the Formatting Functions *request* and *response*:

(request and response have disjoint ranges)
$\forall v,v',s,s',t'.\ (\mathit{Requested}(v,s) \wedge \mathit{Responded}(v',s',t')) \Rightarrow (v \neq v')$
(request is injective)
$\forall v,v',s,s'.\ (\mathit{Requested}(v,s) \wedge \mathit{Requested}(v',s') \wedge v = v') \Rightarrow (s = s')$
(response is injective)
$\forall v,v',s,s',t,t'.$
$(\mathit{Responded}(v,s,t) \wedge \mathit{Responded}(v',s',t') \wedge v = v') \Rightarrow (s = s' \wedge t = t')$

For typechecking the rest of the protocol, we can instead assume these formulas; this confirms that the security of our protocol depends only on these properties, rather than a specific format. In addition, typechecking involves the following three assumptions:

Formulas Assumed for Typechecking the RPC protocol:

(KeyAB MACSays)
$\forall a,b,k,m.\ \mathit{KeyAB}(k,a,b) \Rightarrow (\ \mathit{MACSays}(k,m) \Leftrightarrow$
$(\ (\exists s.\ \mathit{Requested}(m,s) \wedge \mathit{Request}(a,b,s)) \vee$
  $(\exists s,t.\ \mathit{Responded}(m,s,t) \wedge \mathit{Response}(a,b,s,t)) \vee$
  $(\mathit{Bad}(a) \vee \mathit{Bad}(b))))$

(KeyAB Injective)
  $\forall k,a,b,a',b'.\ \mathit{KeyAB}(k,a,b) \wedge \mathit{KeyAB}(k,a',b') \Rightarrow (a{=}a') \wedge (b{=}b')$

(KeyAB Pub Bad)
  $\forall a,b,k.\ \mathit{KeyAB}(k,a,b) \wedge \mathit{Pub}(k) \Rightarrow \mathit{Bad}(a) \vee \mathit{Bad}(b)$

The formula (KeyAB MACSays) is a *definition* for the library predicate *MACSays*. It states the intended usage of keys in this protocol by relating *MACSays* to the protocol-specific predicates *Request*, *Requested*, *Respond*, *Responded*, and *Bad*. The definition has four cases: the MAC is for an authentic request s formatted by function *request*, the MAC is for an authentic response to a prior request formatted by function *response*, or the sender is compromised, or the receiver is compromised.

The formula (KeyAB Injective) is a *theorem* stating that each key is used by a single pair of principals. Our informal invariant on key usage (assumption (4)) directly follows, since $\mathit{KeyAB}(k,a,b)$ is a precondition of both *client* and *server*. The proof is by induction on any run of a program that assumes *KeyAB* only in the body of *mkKeyAB*. It follows from a more general property of our library: *hmac_kgen* returns a key built from a fresh name, hence this key is different from any value previously recorded in any event. Whenever a new event $KeyAB(k,a,b)$ is assumed, and for any event $KeyAB(k',a',b')$ previously assumed, we have $k \neq k'$, so any new instance of (KeyAB Injective) holds. Conversely, we would not be able to prove the theorem if *mkKeyAB* also (erroneously) assumed $KeyAB(k,b,a)$, for instance, as that might enable reflection attacks.

The formula (KeyAB Pub Bad) is a *secrecy theorem* for the MAC keys allocated by the protocol, stating that those keys remain secret until one of the two recorded owners is compromised. This theorem validates our key-compromise model, but is not needed for typechecking. Its proof goes as follows. Relying on the postcondition of the call to *hmac_keygen* within *mkKeyAB*, we always have $MKey(k)$ when $KeyAB(k,a,b)$ is assumed, hence we establish the lemma $\forall a,b,k.\ KeyAB(k,a,b) \Rightarrow MKey(k)$. By corollary C2, $KeyAB(k,a,b)$ and $Pub(k)$ thus imply that $\forall m.\ MACSays(k,m)$. By inspecting (KeyAB MACSays), it suffices to show that there always exists at least one value M such that we have neither $Requested(M,s)$ nor $Responded(M,s,t)$, for any s, t. This trivially follows from the definitions of these two predicates; not every bytestring is a well-formatted request or response.

3.7 Refinement Types for the RPC Protocol

Using F7, we check that our protocol code (with the *Net* and *Crypto* library interfaces, and the assumed formulas above) is a well-typed implementation of the interface below.

Typed Interface for the RPC Protocol:

```
type payload = strpub
val request: s:payload → m:bytespub{Requested(m,s)}
val response: s:payload → t:payload → m:bytespub{Responded(m,s,t)}
val service: payload → payload
type (;a:str,b:str)keyab = k:key { MKey(k) ∧ KeyAB(k,a,b) }
val mkKeyAB: a:str → b:str → k: (;a,b)keyab
val client: a:str → b:str → k: (;a,b)keyab → payload → unit
val server: a:str → b:str → k: (;a,b)keyab → unit
```

This interface is similar but more precise than the one in F#. The type payload is a refinement of string (str) that also states that the payload is a public value, so that in particular it may be sent in the clear. The value-dependent type keyab is a refinement of key that also states that the key is a MAC key for messages from a to b.

We briefly comment on the (fully automated) usage of our logical rules during typechecking.

- To type the calls to *hmacsha1*, the precondition follows from the refinement in the type of k from either the first or the second disjunct of (KeyAB MACSays).
- To type the calls to *send*, we rely on the postcondition of *hmacsha1* to show that the computed MAC is public.
- To type the leaked key k as keypub within *setup*, we need to show $Pub(k)$. This follows from $MKey(k)$ (from the refinement in the type of k), corollary C2, and the definition of *MACSays*, using the just-assumed formula $Bad(a)$ or $Bad(b)$ to satisfy either the third or the fourth disjunct of (KeyAB MACSays).
- To type the *RecvRequest* protocol assertion, we must prove the formula $Request(a,b,s) \vee Bad(a) \vee Bad(b)$ in a context where we have $KeyAB(k,a,b)$, $Requested(v,s)$, and $IsMAC(h,k,v)$. By corollary C1, we have $MACSays(k,v) \vee Pub(k)$. By corollary C2, we have $MKey(k) \wedge Pub(k) \Rightarrow \forall v.\ MACSays(k,v)$, so we obtain $MACSays(k,v)$ in both cases of the disjunction. By definition of (KeyAB MACSays), this yields

$$\begin{array}{l}(Requested(v,s) \wedge \exists s.\ (Requested(v,s) \wedge Request(a,b,s))) \vee \\ (Requested(v,s) \wedge \exists s,t.\ (Responded(v,s,t) \wedge Response(a,b,s,t))) \vee \\ Bad(a) \vee Bad(b)\end{array}$$

  which implies $Request(a,b,s) \vee Bad(a) \vee Bad(b)$ by using the properties of our formatting functions.

4. Semantic Safety by Modular Typing

This section develops the theory underpinning our verification technique. First, we introduce *semantic safety*, which allows us to make inductive definitions of predicates in RCF. Second, we formalize F7 modules within RCF, and in particular introduce *refined modules*, which are modules packaged with inductive definitions of predicates and associated theorems.

4.1 Syntactic Safety by Typing (Review)

We recall the operational semantics and notion of *syntactic safety* for RCF, together with one of the main theorems of Bengtson et al. (2008). (In the original paper, syntactic safety is known simply as safety.)

The semantics of expressions is defined by a small-step reduction relation, written $A \to A'$, which is defined up to structural rearrangements, written $A \Rrightarrow A'$. We represent all reachable run-time program states using expressions in special forms, named *structures*, ranged over by $\mathbf{S}$. A structure is a parallel composition of active subexpressions running in parallel, within the same scope for all restricted names. (We say a subexpression is *active* to mean that it occurs in evaluation context, that is, nested within restriction, fork, or let-expressions.) In particular, from a given structure, one can extract a finite set of active assumptions and assertions. (This extraction is defined for the whole structure, up to injective renamings on the restricted names.)

- A *C-structure* is a structure whose active assumptions are exactly $\{\textbf{assume}\ C_1, \dots, \textbf{assume}\ C_n\}$ with $C = C_1 \wedge \dots \wedge C_n$.
- A C-structure is *syntactically statically safe* if every RCF-interpretation to satisfy C also satisfies each active assertion.
- An expression A is *syntactically safe* if and only if, for all expressions A' and structures $\mathbf{S}$, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is syntactically statically safe.

THEOREM 2 *(Bengtson et al. 2008)*
If $\varnothing \vdash A : T$, then A is syntactically safe.

PROOF: The Safety Theorem of Bengtson et al. (2008) is formulated in terms of *safety* and *static safety*, which are equivalent to our syntactic safety and syntactic static safety, but defined in terms of a sound inference system FOL/F. We detail the argument for our reformulated theorem. To show that A is syntactically safe, consider any expression A' and structure $\mathbf{S}$ with $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$. Suppose $\mathbf{S}$ is a C-structure. It remains to show that $\mathbf{S}$ is syntactically statically safe, which is to say that for every RCF-interpretation $\mathscr{I}$, and for every active assertion $\textbf{assert}\ C'$ occurring in $\mathbf{S}$, if $\mathscr{I}$ satisfies C then $\mathscr{I}$ satisfies C'. By the Safety Theorem (Theorem 6 in Appendix C), $\varnothing \vdash A : T$ implies that the C-structure $\mathbf{S}$ is *statically safe*, which means that $C \vdash C'$ is derivable in the logic FOL/F for each active assertion $\textbf{assert}\ C'$ occurring in $\mathbf{S}$. By soundness of FOL/F, every RCF-interpretation to satisfy C also satisfies C'. Thus, $\mathbf{S}$ is syntactically statically safe. □

4.2 Inductive Definitions and Semantic Safety by Typing

A key technique in this paper is to consider in RCF predicates given by inductive rules, such as the predicates *Bytes* and *Pub* mentioned in the previous section. We intend to define these predicates in RCF by assuming Horn clauses corresponding to the inductive rules. Formally, we introduce a standard notion of logic program, which is guaranteed by the Tarski-Knaster fixpoint theorem to have a least interpretation.

- A *Horn clause* is a closed formula $\forall x_1, \ldots, x_k.(C_1 \wedge \cdots \wedge C_n \Rightarrow C)$ where C_1, ..., C_n range over atomic formulas and equations and C ranges over atomic formulas.
- A *logic program*, P, is a finite conjunction of Horn clauses.
- Consider RCF-interpretations $\mathscr{I}$ and $\mathscr{I}'$. We let $\mathscr{I} \leq \mathscr{I}'$ mean that, for all predicate symbols p, if R_p and R'_p are the relations assigned to p by $\mathscr{I}$ and $\mathscr{I}'$ then $R_p \subseteq R'_p$.
- If P is a logic program, let $\mathscr{I}_P$ be the least RCF-interpretation to satisfy P (which exists uniquely, by Tarski-Knaster).

We construct the least RCF-interpretation of a logic program P as follows.

LEMMA 1 *If P is a logic program, there is a least RCF-interpretation to satisfy P, obtained as the least fixpoint of a certain function on RCF-interpretations.*

PROOF: Given a logic program P, we construct a function F_P on RCF-interpretations as follows. Given input $\mathscr{I} = (D, I)$, let $F_P(\mathscr{I})$ be the RCF-interpretation (D, I') such that I' associates each predicate p of arity n to the relation $R \subseteq D^n$ given by:

$$\{(N_1 V, \ldots, N_n V) \mid \text{valuation } V \in \{\vec{x}\} \to D \text{ and } \models_{\mathscr{I},V} C \\ \text{where } (\forall \vec{x}.C \Rightarrow p(N_1, \ldots, N_n)) \text{ is a Horn clause from } P\}$$

We say that $\mathscr{I}$ is *F_P-closed* to mean that $F_P(\mathscr{I}) \leq \mathscr{I}$.

So $\mathscr{I}$ is F_P-closed if and only if for all predicates p of arity n, for all Horn clauses $(\forall \vec{x}.C \Rightarrow p(N_1, \ldots, N_n))$ from P, for all valuations $V \in \{\vec{x}\} \to D$, if $\models_{\mathscr{I},V} C$ then $\models_{\mathscr{I}} p(N_1 V, \ldots, N_n V)$.

Slightly rephrased, we have that $\mathscr{I}$ is F_P-closed if and only if for all Horn clauses $(\forall \vec{x}.C \Rightarrow p(N_1, \ldots, N_n))$ from P, for all valuations $V \in \{\vec{x}\} \to D$, $\models_{\mathscr{I},V} (C \Rightarrow p(N_1, \ldots, N_n))$.

This shows that $\mathscr{I}$ is F_P-closed if and only if $\mathscr{I}$ satisfies P.

The set of RCF-interpretations under the ordering $\leq$ forms a lattice. Since P is a collection of Horn clauses, the function F_P is monotone, that is, if $\mathscr{I}_1 \leq \mathscr{I}_2$ then $F_P(\mathscr{I}_1) \leq F_P(\mathscr{I}_2)$. Let $\mu X.F_P(X) = \bigcap\{\mathscr{I} \mid F_P(\mathscr{I}) \leq \mathscr{I}\}$. We have that $\mu X.F_P(X)$ is the least F_P-closed interpretation. By the Tarski-Knaster theorem (see Davey and Priestley (1990), for example), $\mu X.F_P(X)$ is the least fixpoint of F_P, that is, the least RCF-interpretation $\mathscr{I}$ such that $F_P(\mathscr{I}) = \mathscr{I}$. The corresponding induction principle is that $\mu X.F_P(X) \subseteq \mathscr{I}$ for any F_P-closed RCF-interpretation $\mathscr{I}$. □

Syntactic safety asks assertions to hold in *all* interpretations that satisfy the assumptions. Instead, if we move to considering assumptions as inductive definitions, we want a weaker notion, which we name *semantic safety*, that asks assertions to hold only in the *least* interpretation that satisfies the assumptions. Considering only the least interpretation allows us to prove safety by exploiting theorems proved by induction and case analysis on the inductive definitions.

- An expression is *factual* if and only if each of its assumptions (active or not) is a logic program.
- A C-structure is *semantically statically safe* if the least RCF-interpretation to satisfy C also satisfies each asserted formula.
- An expression A is *semantically safe* if and only if, for all expressions A' and structures $\mathbf{S}$, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is semantically statically safe.

Semantic safety may not be well-defined if least interpretations do not exist. A sufficient condition for semantic safety of expression A to be well-defined is when A is factual, for then the active assumptions in each reachable structure form a logic program. Given this condition, syntactic safety implies semantic safety, but not the converse, since semantic safety may rely on properties of the least interpretations.

In the following, we call such a property a "theorem of A", and state a new result for proving semantic safety for A.

- Let C be a *theorem* of A if and only if A is factual and, for all P, $\mathscr{I}_P$ satisfies C for all P-structures reachable from A.

THEOREM 3 *Consider closed expression A and formula C where:*

(1) *the expression* **assume** *$C \Rsh A$ is syntactically safe; and*
(2) *C is a theorem of A.*

Then A is semantically safe.

PROOF: Consider any A' and $\mathbf{S}$ and P, such that $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$. We are to show that $\mathbf{S}$ is semantically statically safe. Suppose that $\mathbf{S}$ is a P-structure. The formula P must be a logic program since, by assumption (2), A is factual, and all expressions reachable from a factual expression are themselves factual. Moreover, by that assumption, we have that $\mathscr{I}_P$ satisfies C, and recall that $\mathscr{I}_P$ is the least RCF-interpretation to satisfy P. Consider any active assertion **assert** C' in $\mathbf{S}$. To see that $\mathbf{S}$ is semantically statically safe, we must show that the interpretation $\mathscr{I}_P$ satisfies the formula C'. We have that **assume** $C \Rsh A \to^*$ **assume** $C \Rsh A'$ and **assume** $C \Rsh A' \Rrightarrow \mathbf{S}'$ where $\mathbf{S}'$ is the same as the P-structure $\mathbf{S}$ but for the additional assumptions **assume** C. Hence, $\mathbf{S}'$ is a $(C \wedge P)$-structure. By assumption (1), **assume** $C \Rsh A$ is syntactically safe. It follows that $\mathbf{S}'$ is syntactically statically safe, and hence that every RCF-interpretation to satisfy $C \wedge P$ also satisfies C'. By assumption (2), $\mathscr{I}_P$ satisfies C. By definition, $\mathscr{I}_P$ satisfies the logic program P. Since, then, $\mathscr{I}_P$ is an RCF-interpretation to satisfy $C \wedge P$, it also satisfies the formula C', as desired. □

4.3 A Simple Formalization of Modules

We formalize F7 modules (including whole programs) and interfaces as RCF expression contexts and environments.

- A *module* X is an expression context of the form **let** $x_1 = A_1$ **in** ... **let** $x_n = A_n$ **in** _ where $n \geq 0$ and the bound variables x_i are distinct. We let $bv(X) = \{x_1, \ldots, x_n\}$. We treat the concrete syntax for composing F# modules as syntactic sugar, writing $X_1\ X_2$ for the module $X_1[X_2[\_]]$.
- An *interface* I is a typing environment $\mu_1, \ldots, \mu_n$ where each μ_i is either an abstract type α_i or a variable typing $x_i : T_i$.
- We lift subtyping to interfaces by the following axioms and rules, plus reflexivity and transitivity, and well-formedness conditions (so that $I <: I'$ always implies $I \vdash \diamond$ and $I' \vdash \diamond$).

$$I_0, (I_1\{T/\alpha\}) <: I_0, \alpha, I_1 \qquad \frac{I_0 \vdash T <: U}{I_0, x : T, I_1 <: I_0, x : U, I_1}$$

$$I_0, \mu, I_1 <: I_0, I_1$$

- A module X *implements* I in E, written $E \vdash X \rightsquigarrow I$, when $E \vdash X[(x_1, \ldots, x_n)] : (x_1 : T_1 * \ldots * x_n : T_n)$ and $(x_i : T_i)_{i=1..n} <: I$.

LEMMA 2 (Modular Typechecking). *If $E, I \vdash A : T$ and $E \vdash X \rightsquigarrow I$, then $E \vdash X[A] : U$ where U is T for some instantiation of the type variables of I.*

We have a similar lemma for composing two modules, rather than a module and an expression.

4.4 Refined Modules

We use an expression context **assume** $P \Rsh Y$ to formalize the idea of a module Y packaged with a (closed) logic program P to make inductive definitions of predicates. We call such contexts *refined modules*. We want to exploit theorems following from P when typechecking Y. To do so, we introduce the notion of a *contextual theorem*, a theorem that holds in any expression containing **assume** $P \Rsh Y$ as a component.

- The *support* of a logic program is the set of predicate symbols occurring in the head of any clause. The *support* of an expression or expression context is the support of its assumptions. (Intuitively, the support is the set of predicates being defined.) Logic programs, expressions, or expression contexts are *independent* when their supports are disjoint.
- Let C be a *contextual theorem* of expression context **assume** $P \Rsh Y$ if and only if C is a theorem of **assume** $P \Rsh Z[Y[A]]$ whenever Z and A are factual and independent of **assume** $P \Rsh Y$.

Lemma 3 *Suppose C is a contextual theorem of expression context **assume** $P \Rsh Y$. If P', Y', and Y'' are independent of P then C is a contextual theorem of **assume** $(P \wedge P') \Rsh Y'[Y[Y'']]$.*

Proof: Let $Z =$ **assume** $P' \Rsh Y'$ and $A = Y''$ so that the reachable structures of **assume** $P \Rsh Z[Y[A]]$ are the same as the reachable structures of **assume** $(P \wedge P') \Rsh Y'[Y[Y'']]$, up to structural rearrangements. Hence, since C is a contextual theorem of **assume** $P \Rsh Y$, it is also a contextual theorem of **assume** $(P \wedge P') \Rsh Y'[Y[Y'']]$.□

Lemma 4 *If C_1 and C_2 are contextual theorems of expression context **assume** $P \Rsh Y$, then so is $C_1 \wedge C_2$.*

Proof: Immediate from the definitions. □

When the following lemma applies, we can prove contextual theorems from the inductive definitions P of **assume** $P \Rsh Y$, without explicit consideration of the operational semantics.

Lemma 5 (Contextual). *Let C be a formula and P a logic program such that, for all Q independent from P, the least RCF-interpretation to satisfy $P \wedge Q$ also satisfies C. If Y is an expression context independent from P, then C is a contextual theorem of **assume** $P \Rsh Y$.*

Proof: Consider expression context Z and expression A that are factual and independent of **assume** $P \Rsh Y$. We are to show that C is a theorem of **assume** $P \Rsh Z[Y[A]]$. Since Z, Y, and A are all independent of P, it follows for every R, that is an R-structure is reachable from **assume** $P \Rsh Z[Y[A]]$ then R takes the form $R = P \wedge Q$ where Q is independent from P. By assumption, the least RCF-interpretation to satisfy $P \wedge Q$ also satisfies C. Hence, C is a theorem of **assume** $P \Rsh Z[Y[A]]$, as required. □

- Let a *refined module* be a triple $\mathbf{M} = (E, X, I)$ such that there are closed formulas $\mathbf{M}^{def}$ and $\mathbf{M}^{thm}$, and a module Y where:
  (1) X is factual and $X =$ **assume** $\mathbf{M}^{def} \Rsh Y$;
  (2) $E, \mathbf{M}^{def}, \mathbf{M}^{thm} \vdash Y \rightsquigarrow I$;
  (3) $\mathbf{M}^{thm}$ is a contextual theorem of X.

(When we write a formula such as $\mathbf{M}^{def}$ as an environment entry, we mean it as a shorthand for $\_ : \{\mathbf{M}^{def}\}$ where the type $\{\mathbf{M}^{def}\} = \_ :$ unit$\{\mathbf{M}^{def}\}$, where each occurrence of _ stands for a fresh variable. This type is only populated when $\mathbf{M}^{def}$ holds, so the effect of the entry is simply to add $\mathbf{M}^{def}$ as a logical assumption.)

Our example relies on **Lib**, the composition of the library modules **Data**, **Net**, and **Crypto**, which together form a refined module. Let *Lib* be the F# code of the library, that is, the composition *Data Net Crypto* of the code of the libraries. Let I_L^7 be the F7 interface, which includes, for example, the functions labelled "Refinement Types for MACs" in Section 3. The inductive definitions $\mathbf{Lib}^{def}$ include formulas defining the *Pub* and *Bytes* predicates, while $\mathbf{Lib}^{thm}$ includes the corollaries C1 and C2 in Section 3.

Lemma 6 $\mathbf{Lib} = (\varnothing,$ ***assume*** $\mathbf{Lib}^{def} \Rsh Lib, I_L^7)$ *is a refined module.*

As another example, our RPC protocol consists of a refined module of the form: $\mathbf{RPC} = (I_L^7,$ **assume** $\mathbf{RPC}^{def} \Rsh RPC, (I_L, I_R))$. Let *RPC* be the F# code for the protocol. The inductive definitions $\mathbf{RPC}^{def}$ include the right to left form of (KeyAB MACSays) from Section 3. The theorems $\mathbf{RPC}^{thm}$ include (KeyAB Injective), (KeyAB Pub Bad), and the left to right form of (KeyAB MACSays) from Section 3. The exported interface (I_L, I_R) is made available to the opponent. Let I_L be the library's opponent interface, which is excerpted in Section 3. Let I_R be the protocol-specific opponent interface from Section 3. As mentioned in that section, the module below imports I_L^7 and exports its members at the more abstract interface I_L, by introducing abstract types such as bytespub with representation type x: bytes $\{Pub(x)\}$.

Lemma 7 **RPC** *is a refined module.*

The proofs of Lemmas 6 and 7 are in Appendix A.9, and rely on Lemma 5 (Contextual).

4.5 Composition of Refined Modules

- We say $\mathbf{M}_1 = (E_1, X_1, I_1)$ *composes with* $\mathbf{M}_2 = (E_2, X_2, I_2)$ iff $I_1 <: E_2$ and X_1 and X_2 are independent.
- For any triples $\mathbf{M}_1 = (E_1,$ **assume** $\mathbf{M}_1^{def} \Rsh Y_1, I_1)$ and $\mathbf{M}_2 = (E_2,$ **assume** $\mathbf{M}_2^{def} \Rsh Y_2, I_2)$ their *composition* $\mathbf{M}_1;\mathbf{M}_2$ is the triple $(E_1,$ **assume** $(\mathbf{M}_1^{def} \wedge \mathbf{M}_2^{def}) \Rsh Y_1[Y_2], I_2)$.

Lemma 8 (Composition). *If refined module $\mathbf{M}_1$ composes with refined module $\mathbf{M}_2$ then $\mathbf{M}_1;\mathbf{M}_2$ is a refined module.*

Proof: For $i \in 1..2$, we have $\mathbf{M}_i = (E_i, X_i, I_i)$ where $X_i =$ **assume** $\mathbf{M}_i^{def} \Rsh Y_i$ is factual, $E_i, \mathbf{M}_i^{def}, \mathbf{M}_i^{thm} \vdash Y_i \rightsquigarrow I_i$, and $\mathbf{M}_i^{thm}$ is a contextual theorem of X_i. Since $\mathbf{M}_1$ composes with $\mathbf{M}_2$, we have $I_1 <: E_2$ and X_1 and X_2 are independent, that is, the supports of X_1 and X_2 are disjoint.

Consider the composition $\mathbf{M}_1;\mathbf{M}_2 = (E_1, X_{12}, I_2)$ where $X_{12} =$ **assume** $\mathbf{M}_{12}^{def} \Rsh Y_1[Y_2]$ with $\mathbf{M}_{12}^{def} = (\mathbf{M}_1^{def} \wedge \mathbf{M}_2^{def})$.

To see that $\mathbf{M}_1;\mathbf{M}_2$ is a refined module, we must show that:

(1) X_{12} is factual;
(2) $E_1, \mathbf{M}_1^{def} \wedge \mathbf{M}_2^{def}, \mathbf{M}_1^{thm} \wedge \mathbf{M}_2^{thm} \vdash Y_1[Y_2] \rightsquigarrow I_2$;
(3) $\mathbf{M}_1^{thm} \wedge \mathbf{M}_2^{thm}$ is a contextual theorem of X_{12}.

Point (1) follows because the constituent parts of X_{12} come from X_1 and X_2, which are themselves factual.

Point (2) follows from $E_1, \mathbf{M}_1^{def}, \mathbf{M}_1^{thm} \vdash Y_1 \rightsquigarrow I_1$ and $I_1 <: E_2$ and $E_2, \mathbf{M}_2^{def}, \mathbf{M}_2^{thm} \vdash Y_2 \rightsquigarrow I_2$, by weakening and substitution properties of the RCF type system.

For point (3), by Lemma 3, since X_1 and X_2 are independent, $\mathbf{M}_1^{thm}$ is a contextual theorem of X_{12}. By symmetric reasoning, $\mathbf{M}_2^{thm}$ is a contextual theorem of X_{12}. By Lemma 4, $\mathbf{M}_1^{thm} \wedge \mathbf{M}_2^{thm}$ is a contextual theorem of X_{12}. □

For example, the triple $\mathbf{Lib};\mathbf{RPC}$ is: $(\varnothing,$ **assume** $(\mathbf{Lib}^{def} \wedge \mathbf{RPC}^{def}) \Rsh Lib[RPC], (I_L, I_R))$. By Lemma 8 (Composition), $\mathbf{Lib};\mathbf{RPC}$ is a refined module.

4.6 Safety and Robust Safety by Typing for Modules

- A refined module $(\varnothing, X, \varnothing)$ is *semantically safe* if and only if, the expression $X[()]$ is semantically safe.
- An *I-opponent* is an opponent O such that $I \vdash O :$ unit.
- A refined module $(\varnothing, X, I)$ is *robustly safe* if and only if, the expression $X[O]$ is semantically safe for every I-opponent O.

The proofs of the following rely on Theorem 2 and Theorem 3.

THEOREM 4 (Safety).
Every refined module $(\varnothing, X, \varnothing)$ is semantically safe.

PROOF: Consider a refined module $\mathbf{M} = (\varnothing, X, \varnothing)$. We are to show that expression $X[()]$ is semantically safe. Since $\mathbf{M}$ is a refined module, there is $\mathbf{M}^{thm}$ such that $\varnothing, \mathbf{M}^{thm} \vdash X \rightsquigarrow \varnothing$, and therefore, $\varnothing \vdash \textbf{assume } \mathbf{M}^{thm} \Rsh X[()] : \text{unit}$. By Theorem 2, $\textbf{assume } \mathbf{M}^{thm} \Rsh X[()]$ is syntactically safe. Since $\mathbf{M}$ is a refined module, $\mathbf{M}^{thm}$ is a contextual theorem of X, which implies that $\mathbf{M}^{thm}$ is a theorem of $X[()]$. Hence, by Theorem 3, we conclude that $X[()]$ is semantically safe. □

THEOREM 5 (Robust Safety).
Every refined module $(\varnothing, X, I)$ is robustly safe.

PROOF: We know that $X = \textbf{assume } \mathbf{M}^{def} \Rsh Y$ for some $\mathbf{M}^{def}$ and Y. We consider any opponent O such that $I \vdash O : \text{unit}$. We are to show that $X[O]$ is semantically safe.

Let $\mathbf{O} = (I, X_O, \varnothing)$ where $\mathbf{O}^{def} = \mathit{True}$ and $\mathbf{O}^{thm} = \mathit{True}$ and $X_O = \textbf{assume } \mathbf{O}^{def} \Rsh Y_O$ and $Y_O = (\textbf{let } x = O \textbf{ in } \_)$. We have that $\mathbf{O}$ is a refined module because:

(1) X_O is factual (because no **assume** occurs in the opponent O) and has the form $\textbf{assume } \mathbf{O}^{def} \Rsh Y_O$;
(2) $I, \mathbf{O}^{def}, \mathbf{O}^{thm} \vdash Y_O \rightsquigarrow \varnothing$ (because $I \vdash O : \text{unit}$); and
(3) $\mathbf{M}^{thm}$ is trivially a contextual theorem of X_O.

We have that $(\varnothing, X, I)$ composes with $\mathbf{O}$ and both are refined modules. By Lemma 8 (Composition), their composition

$$(\varnothing, \textbf{assume } (\mathbf{M}^{def} \wedge \mathit{True}) \Rsh Y[Y_O], \varnothing)$$

is a refined module. Hence, by Theorem 4 (Safety), their composition is semantically safe, that is, the following expression is semantically safe:

$$\textbf{assume } (\mathbf{M}^{def} \wedge \mathit{True}) \Rsh Y[\textbf{let } x = O \textbf{ in } ()]$$

Hence, it follows that the expression $X[O]$, that is,

$$\textbf{assume } \mathbf{M}^{def} \Rsh Y[O]$$

is semantically safe. □

We can now prove Theorem 1. We have that $\mathbf{Lib};\mathbf{RPC} = (\varnothing, X, (I_L, I_R))$ where $X = \textbf{assume } (\mathbf{Lib}^{def} \wedge \mathbf{RPC}^{def}) \Rsh \mathit{Lib}[\mathit{RPC}]$ is a refined module. By Theorem 5 (Robust Safety), $(\varnothing, X, (I_L, I_R))$ is robustly safe, which is to say that $X[O]$ is semantically safe for every opponent O with $I_L, I_R \vdash O : \text{unit}$.

5. Library Modules for Cryptographic Protocols

In this section, we describe intermediate refined modules, built on top of the **Crypto** module, that implement derived mechanisms and composite patterns commonly used in cryptographic protocol implementations. (Section 3 also presents its interface for MACs.)

- Keys can be encrypted, authenticated, and selectively released (modelling key compromises).
- All derived modes for authenticated encryption are obtained by composing MACs and symmetric encryption.
- Hybrid encryption is obtained by composing symmetric and public-key encryption.
- Multiple keys can be derived from a secret seed, yielding separate keys for authentication and encryption.
- MACs and signatures can be nested, enabling multiple principals to jointly authenticate parts of a message.

Relying on these libraries, their logical definitions, and their theorems, we build (and verify) a series of modular protocols, leading to Windows CardSpace.

5.1 Key Management

The **Principals** library generalizes the treatment of keys and principals illustrated in the example protocol of Section 3. (To facilitate the comparison, we illustrate here mostly the treatment of MAC keys.) Instead of a fixed population of principals and keys, the library maintains a database of keys shared between an extensible set of principals. Pragmatically, this functionality may be implemented using some existing public-key infrastructure, or an in-memory database recording the outcome of prior key-exchange protocols. Formally, our implementation of **Principals** relies on **Db**, a channel-based abstraction for databases.The main purpose of the library is to systematically link cryptographic keys to application-level principals, while keeping track of their potential compromise.

Principal identifiers are represented by a type prin defined as a public string. Each principal may have a number of MAC keys, encryption keys, and public/private key pairs. The library maintains a database that may be used by multiple protocols to store and retrieve keys. Keys are grouped by usage (set by the protocol that generates the key) to distinguish between the intended usage of each key, and associated with one (for public/private keypairs) or two principals.

For instance, a MAC key *mk* managed by the library for some usage `"RPC"` shared between principals *a* and *b* is given the type (*mk*:key)$\{$*MACKey*(`"RPC"`,*a*,*b*,*mk*)$\}$ (where key is the type of keys in **Crypto**). For managed MAC keys, **Principals** provides functions:

val *mkMACKey*: *u*:usage → *a*:prin → *b*:prin →
  mk:key$\{$*MACKey*(*u*,*a*,*b*,*mk*)$\}$
val *genMACKey*: *u*:usage → *a*:prin → *b*:prin → unit
private val *getMACKey*: *u*:usage → *a*:prin → *b*:prin →
  mk:key$\{$*MACKey*(*u*,*a*,*b*,*mk*)$\}$

The function *mkMACKey* generates a fresh MAC key, associates it with a particular usage and pair of principals, and returns the key. The function *genMACKey* calls *mkMACKey* to generate a key then stores it in the database. The function *getMACKey* retrieves a key from the database. Of these three functions, only *genMACKey* is available in the opponent interface.

Managed keys can be used for standard cryptographic operations. To this end, **Principals** links key-level predicates used in **Crypto** (defined by **Principals**) to principal-level predicates used in **Principals** (to be defined by the protocol): *Send*(*u*,*a*,*b*,*s*) means that the principal *a* intends to MAC *s* before sending it to *b*; *Encrypt* (*u*,*a*,*b*,*s*) records that *s* may be encrypted towards *b* using symmetric encryption; *SendFrom* and *EncryptTo* similarly record intended asymmetric signatures and encryption with a managed key.

The **Principals** library also provides functions for compromising keys. Compromise is dealt with at the level of principals: *Bad* (*a*) indicates that principal *a* has been compromised, and thus that all the keys it could access may have been leaked. For each kind of key, the module has a function that can be used for modelling compromises. For compromised MAC keys, for instance, it has a function

val *leakMACKey*: *u*:usage → *a*:prin → *b*:prin →
  mk:keypub$\{$*Bad*(*a*) ∧ *Bad*(*b*) ∧ *MACKey*(*u*,*a*,*b*,*mk*)$\}$

For MACs, for instance, the library interface assumes the formulas below.

MAC Key Usage:

(MACKey MACSays Send)
$\forall u,a,b,mk,m.\ MACKey(u,a,b,mk) \wedge Send(u,a,b,m) \Rightarrow MACSays(mk,m)$
(MACKey MACSays Bad)
$\forall u,a,b,mk,m.$
$MACKey(u,a,b,mk) \wedge (Bad(a) \vee Bad(b)) \Rightarrow MACSays(mk,m)$

(Inv MACKey MACSays)
$\forall u,a,b,mk,m.\ MACKey(u,a,b,mk) \wedge MACSays(mk,m) \Rightarrow$
$(Send(u,a,b,m) \vee Bad(a) \vee Bad(b))$

(MACKey Secrecy)
$\forall u,a,b,mk.\ MACKey(u,a,b,mk) \wedge Pub(mk) \Rightarrow$
$(Bad(a) \vee Bad(b) \vee (\forall v.\ Send(u,a,b,v)))$

The two first clauses are definitions, enabling *hmacsha1* to be called with a managed MAC key once the protocol has assumed an adequate definition of *Send*, with a more liberal precondition in case of compromise. The third and fourth clauses are theorems: MAC verification with a managed key yields a principal-level guarantee; and a MAC key shared between two principals remains secret until one of them gets compromised.

Our model of key compromise is among the most general models for protocol verification. It supports three kinds of keys: those generated by the attacker, those generated by the principals library and kept secret, and those generated by the principals library and leaked to the attacker. It allows cryptographic operations to be performed with all three categories of keys. Moreover, all keys may be encrypted, MACed, or signed under other keys. For instance, if a key is used to encrypt some collection of other keys (as tracked by *Send*), our logical model rightfully demands, as a precondition for compromising any principal with access to that key, that the conditions for leaking each of these encrypted keys be also recursively satisfied. Although this leads to complex refinement types and assumptions, most of this complexity is factored out in the library and can be used with a low overhead.

Recall that **LibX** is the composition of **Lib**, **Db**, and **Xml**.

LEMMA 9 **LibX**;**Principals** *is a refined module.*

5.2 Authenticated Encryption

The **Crypto** module provides plain (unauthenticated) symmetric encryption:

Refinement Types for Encryption (from the *Crypto* library):

private val *aes_keygen*: unit $\to k$:key$\{SKey(k)\}$
val *aes_encrypt*: `(* AES CBC *)`
  k:key $\to$
  b:bytes$\{(SKey(k) \wedge CanSymEncrypt(k,b)) \vee (Pub(k) \wedge Pub(b))\} \to$
  e:bytes$\{IsEncryption(e,k,b)\}$
val *aes_decrypt*: `(* AES CBC *)`
  k:key$\{SKey(k) \vee Pub(k)\} \to e$:bytes $\to$
  b:bytes$\{(\forall p.\ IsEncryption(e,k,p) \Rightarrow b = p) \wedge (Pub(k) \Rightarrow Pub(b))\}$

The function *aes_keygen* generates symmetric keys, logically tracked by *SKey*. The function *aes_encrypt* can be called in two ways; either with a "good" key k generated by *aes_keygen* and a plaintext b such that *CanSymEncrypt*(k,b) holds, or with any public k and b (known to or provided by the attacker). In both cases, it returns encrypted bytes e, tracked by *IsEncryption*. The function *aes_decrypt* takes a key k and bytes e and extracts a plaintext b. Since encryption is unauthenticated, if e is not a valid encryption under k, decryption may still succeed and return some unspecified (garbage) bytes. Hence, the postcondition of *aes_decrypt* just says that (1) if the caller knows that e is the valid encryption of some (possibly unknown) plaintext p under e, then decryption does returns p; and besides (2) if the key is public, so is the plaintext.

The **Patterns** module shows how to derive authenticated encryption, for each of the three standard composition methods for encryption and MACs (see, e.g., Bellare and Namprempre 2008).

Encrypt-then-MAC (as in IPSEC in tunnel mode):

$a \to b$: $\quad e \mid hmacsha1\ k^m_{ab}\ e$ where $e = aes\ k^e_{ab}\ t$

MAC-then-Encrypt (as in SSL/TLS):

$a \to b$: $\quad aes\ k^e_{ab}\ (t \mid hmacsha1\ k^m_{ab}\ t)$

MAC-and-Encrypt (as in SSH):

$a \to b$: $\quad aes\ k^e_{ab}\ t \mid hmacsha1\ k^m_{ab}\ t$

Depending on the method, the message is first encrypted, then the encryption is MACed, or the message is first MACed and then both the message and the MAC are encrypted, or the message is first MACed but the MAC is left unencrypted. For each method, the goal is to securely communicate plaintexts t from a to b relying on pre-established shared keys, but the underlying cryptographic assumptions slightly differ. Cryptographers prefer the first method, as it prevents chosen-ciphertext attacks and does not require secrecy assumptions on the MAC function. We implemented and verified all three (using a secrecy-preserving MAC in the third case, as expected). We focus on encrypt-then-MAC, since this was not implementable in our previous work with F7.

Authenticated Encryption API:

val *authenc_keygen*: unit $\to (ek$:key $*$ mk:key$)\{AuthEncKeyPair(ek,mk)\}$
val *encrypt_then_mac*: ek:key $\to mk$:key $\to$
  b:bytes$\{(AuthEncKeyPair(ek,mk) \wedge CanSymEncrypt(ek,b)) \vee$
    $(Pub(ek) \wedge Pub(mk) \wedge Pub(b))\} \to$
  e:bytes$\{IsAuthEncryption(e,ek,mk,b)\}$
val *verify_then_decrypt*:
  ek:key $\to$
  mk:key$\{(AuthEncKeyPair(ek,mk) \vee (Pub(ek) \wedge Pub(mk)))\} \to$
  e:bytes $\to$
  b:bytes$\{(CanSymEncrypt(ek,b) \vee Pub(ek)) \wedge (Pub(ek) \Rightarrow Pub(b))\}$

The function *AuthEncKeyPair* links pairs of keys for the method; encryption returns a concatenation of an encryption and a MAC, tracked by *IsAuthEncryption*. *verify_then_decrypt* has a stronger postcondition than *aes_decrypt*; its result must have been encrypted using *encrypt_then_mac*, thereby excluding garbage. To verify these functions and obtain both integrity and confidentiality for b, for each key pair (*AuthEncKeyPair*(ek,mk)), we link *MACSays*(mk,b) and *CanSymEncrypt*(ek,e) to get both integrity and confidentiality for b:

Authenticated Encryption Key Usage:

(AuthEncKeyPair MACSays)
$\forall mk,ek,c,p.\ AuthEncKeyPair(ek,mk) \wedge IsEncryption(c,ek,p) \wedge$
  $CanSymEncrypt(ek,p) \Rightarrow MACSays(mk,c)$

The correctness of *verify_then_decrypt* relies on theorems stating that this is the only use of these keys, and linking their potential compromise.

5.3 Hybrid encryption

Hybrid encryption is the standard method of implementing public-key encryption for large plaintexts: generate a fresh symmetric key; use it to encrypt the plaintext; then encrypt the key using the public key of the intended receiver.

Hybrid Encryption:

$a \to b$: $\quad$ $\mathit{rsa\_oaep}\ pk_b\ k_{ab} \mid \mathit{aes}\ k_{ab}\ t$

This hybrid encryption combines authenticated asymmetric encryption (RSA-OAEP) with unauthenticated symmetric encryption, and provides unauthenticated asymmetric encryption (analogous to RSA without OAEP). The library has three functions for it:

Hybrid Encryption API:

val *hybrid_keygen*: unit $\to$ (pk:key $*$ sk:key)
$\quad\{\mathit{HyPubKey}(pk) \land \mathit{HyPrivKey}(sk) \land \mathit{PubPrivKeyPair}(pk,sk)\}$
val *hybridEncrypt*: k:key $\to$ b:bytes
$\quad\{(\mathit{HyPubKey}(k) \land \mathit{CanHyEncrypt}(k,b)) \lor (\mathit{Pub}(k) \land \mathit{Pub}(b))\ \} \to$
$\quad e$:bytes$\{\mathit{IsHyEncryption}(e,k,b)\}$
val *hybridDecrypt*: sk:key $\to$
$\quad e$:bytes$\{\mathit{HyPrivKey}(sk) \lor (\mathit{Pub}(sk) \land \mathit{Pub}(e))\} \to$
$\quad b$:bytes$\{(\forall pk,x.\ (\mathit{PubPrivKeyPair}(pk,sk)$
$\quad \land \mathit{IsHyEncryption}(e,pk,x)) \Rightarrow x = b) \land (\mathit{Pub}(sk) \Rightarrow \mathit{Pub}(b))\}$

Their code is straightforward, but their verification is challenging (since it must rely on the assumption that the symmetric key is used for a *single* hybrid encryption). Predicates *HyPubKey*, *HyPrivKey*, and *HySymKey* track the three kinds of keys used in the code. The protocol-defined precondition of *hybridEncrypt* is linked to the underlying *CanSymEncrypt* and *CanAsymEncrypt* cryptographic predicates as follows:

Hybrid Encryption Key Usage:

(HyPubKey CanAsymEncrypt)
$\forall pk,kb.\ \mathit{HyPubKey}(pk) \land \mathit{HySymKey}(\mathit{SymKey}(kb),pk) \Rightarrow$
$\qquad\qquad\qquad\qquad \mathit{CanAsymEncrypt}(pk,kb)$
(HySymKey CanSymEncrypt)
$\forall pk,k,b.\ \mathit{HySymKey}(k,pk) \land \mathit{CanHyEncrypt}(pk,b) \Rightarrow \mathit{CanSymEncrypt}(k,b)$

To typecheck *hybridDecrypt*, we establish theorems stating that hybrid encryption keys are used only as above, and linking the compromise of the inner symmetric encryption key to that of the outer private key. After hiding auxiliary predicates, hybrid encryption has exactly the same interface as plain RSA in **Crypto**, showing that the derivation does not entail any loss of flexibility.

5.4 Derived Keys

Cryptographic protocols often use key derivation functions to obtain separate keys from the same shared secret. For instance, our library supports the use of the cryptographic hash function *psha1* to derive a MAC key from a shared secret seed and a fresh nonce. The sample protocol below applies it to secure a single message t.

Using a Derived MAC Key:

$a \to b$: $\quad$ $t \mid n \mid \mathit{hmacsha1}\ (\mathit{psha1}\ k_{ab}\ n)\ t$

A new key predicate keeps track of secret seeds that may be used for key derivation. Derived MAC keys may be used anywhere a MAC key is expected; their logical properties are encoded within the definition of *MKey*.

5.5 Endorsing Signatures

Much like hybrid encryption, we can compose symmetric and asymmetric authentication mechanisms. An *endorsing signature* is a (private-key) signature of a MAC over a message. It provides the same authentication as a signature of the message, with the additional flexibility of signing later, for instance to endorse a received message.

MAC-then-Sign: Endorsing a MAC

$a \to b$: $\quad$ $h \mid \mathit{rsasha1}\ sk_a\ h$ where $h = \mathit{hmacsha1}\ mk_{ab}\ t$

The API and proofs of this mechanism are quite similar in spirit to hybrid encryption. We define a set of endorsing signature key pairs (sk,mk); for such keys we link *SignSays*(sk,mac) with *MACSays*(mk,m) and *IsMAC*(mac,Kab,m).

LEMMA 10 **LibX**;**Patterns** *is a refined module.*

5.6 Example: The Otway-Rees Protocol

Using the **Principals** and **Patterns** libraries, we can build up several protocol implementations and establish their security with minimal effort. We outline our implementation of the Otway-Rees protocol, a well-known protocol for establishing a fresh short-term key between two principals a and b (Otway and Rees 1987).

Otway-Rees Protocol:

1. $a \to b$: $\quad id \mid a \mid b \mid \mathit{aenc}\ ka\ (na \mid id \mid a \mid b)$
2. $b \to s$: $\quad id \mid a \mid b \mid \mathit{aenc}\ ka\ (na \mid id \mid a \mid b)$
   $\qquad\qquad\qquad \mid \mathit{aenc}\ kb\ (nb \mid id \mid a \mid b)$
3. $s \to b$: $\quad id \mid \mathit{aenc}\ ka\ (na \mid kab) \mid \mathit{aenc}\ kb\ (nb \mid kab)$
4. $b \to a$: $\quad id \mid \mathit{aenc}\ ka\ (na \mid kab)$

Here, *aenc k x* stands for the *authenticated encryption* of x under the key pair k, implemented using the Encrypt-Then-MAC mechanism. Using **Principals** we create a population of principals, ranged over by p, together with a server s. The server shares a set of long-term key pairs with principals. Each long-term key pair kp is associated with and known to principal p and to s.

The main authentication goal is that a, b, and s agree on all the main parameters of the protocol: the principals involved a, b, s, the session identifier id, and the established key kab. The main secrecy goal is that kab must be known only to a, b, and s. These goals are established mainly by typing the code against the **Principals** and **Patterns** interfaces. The only theorems proved by hand state the freshness of nonces and keys generated in the protocol.

The proof of the following is in Appendix **??**.

LEMMA 11 **LibX**;**Patterns**;**Principals**;**OtwayRees** *is a refined module.*

5.7 Example: Secure Conversations

Next, we build a protocol for authenticated conversations between two principals. To illustrate compositionality, the key k is established by the Otway-Rees protocol, then used for authenticated encryption, as described above.

Session Sequence Integrity (initially $i = 1$):

$i \quad\ .\ a \to b$: $\quad id \mid \mathit{aenc}\ k\ (i \mid m_i)$
$i+1.\ b \to a$: $\quad id \mid \mathit{aenc}\ k\ (i+1 \mid m_{i+1})$

After key establishment, the conversation protocol loops between request and response messages, incrementing a sequence number at each step. The authentication goal is that a and b must agree on the full sequence of messages $(m_i)_{i \geq 1}$ sent and received (possibly excluding the last message in transit). Verification of such unbounded protocols is typically beyond the reach of automated verification tools, since it requires a form of induction. Nonetheless, we are able to implement and verify this protocol by typing, relying on recursive predicates that record the entire history of the session, and show that the local histories at both a and b are consistent.

We use event predicates as follows to record the full session at each participant. Each participant maintains the current sequence number i and a list l of all the messages sent and received so far.

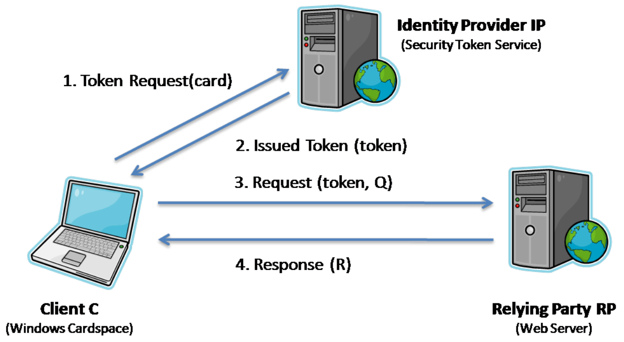

Figure 3. Windows CardSpace Protocol

- We assume $Message(id,i,m_i)$ before sending message i.
- We define a history predicate $Messages(id,i,l)$ where l is the sequence of messages up to sequence number i exchanged in the session:

(Empty Log)
$\forall id.\ Messages(id,0,[])$.
(Cons Log)
$\forall id,i,m,t.\ Message(id,Succ(i),m) \wedge$
$\quad Messages(id,i,t) \Rightarrow Messages(id,Succ(i),m::t)$

In any system configuration, in the presence of an active adversary, we intend that the following assertions are safe:

- After accepting message i, the receiving principal b asserts:

$$(Messages(id,i,l) \vee Pub(kab))$$

Hence, the session sequence is authenticated unless the key is public. In particular, if *kab* were established using Otway-Rees, this means that the session sequence at *a* and *b* agree with each other unless one of them is *Bad*.

The proof of the following is in Appendix **??**.

LEMMA 12 **LibX**;**Patterns**;**Principals**;**OtwayRees**;**Sessions** *is a refined module.*

6. Case Study: Windows CardSpace

We describe our main case study, verifying an implementation of the federated identity-management protocol Windows CardSpace. The protocol consists of three roles, a client *C*, a web server (named relying party) *RP*, and an identity provider *IP*. To access *RP*, *C* first obtains an *identity token* from *IP*, and then uses this token to authenticate its messages to *RP*. Hence, the protocol uses two message exchanges, between *C* and *IP* then between *C* and *RP*. Structurally, CardSpace is similar to many other server-based identification protocols, such as Kerberos, Passport, and SAML. A distinguishing feature is that it is built using the standard mechanisms of web services security.

Our code is written in F# and was developed for an earlier verification case study (Bhargavan et al. 2008b) using ProVerif. Its modular structure is shown in the figure on the first page. In addition to the trusted libraries **LibX** and the protocol libraries **Principals** and **Patterns**, the implementation consists of library modules implementing various web services security specifications and modules implementing the CardSpace protocol. (We added type annotations, but did not need to change any code for the XML protocol stack.)

Flexible Message Formats: XML Digital Signatures In standardized protocols such as CardSpace, most of the programming effort is in correctly implementing the message formats for interoperability. Protocols built on web service security must also deal with the inherently flexible nature of the XML message format.

An XML signature is far more than a few bytes containing a MAC or signature value; it carries XML metadata indicating how those bytes were computed (in two stages) and how to use the signature. For the first stage, it embeds a list of references to the XML elements it is authenticating, a cryptographic hash of each of these elements, and the names of algorithms used to canonicalize and hash those elements; for the second stage, it embeds a signature computed on those hashes, its algorithm, and a reference to its signing key. For example, a typical signature of *n* elements *t1*, ..., *tn* using an RSA signing key *ska* takes the form:

```
<Signature>
si= <SignedInfo>
      <CanonicalizationMethod Algorithm=C14n />
      <SignatureMethod Algorithm=RSA-SHA1 />
      <Reference uri="#1">
        <Transforms> <Transform Algorithm=C14n /> </>
        <DigestMethod Algorithm=SHA1 />
        <DigestValue> base64 (sha1 (utf8 (c14n t1))) </>
      </Reference>
      ...
      <Reference uri="#n">
        ...
      </Reference>
    </SignedInfo>
    <SignatureValue> base64 (rsa_sign ska (utf8 (c14n si)))</>
    <KeyInfo>... a's X.509 Certificate ...</>
</Signature>
```

To process such a signature, the verifier retrieves the elements, verification key, and the algorithms, and reconstructs the signature value. The signature may include any number of target elements, so the verifier may have to check a signature of unbounded length. This is beyond most cryptographic verification techniques: earlier analyses of XML signature protocols limit the maximum number of signed elements, essentially treating lists as tuples (Bhargavan et al. 2006; Kleiner and Roscoe 2005). With explicit type annotations, however, we capture the full flexibility of XML signatures. We use a recursive predicate *IsReferenceList* to represent the list of `<Reference>` elements, and use it to define a predicate *IsSignedInfo* that reflects the schema of the `<SignedInfo>` element. We enforce the invariant that all messages signed with XML signature keys have the structure defined in *IsSignedInfo*.

Using similar predicates, we verify modules implementing each of the needed web services security specifications. We write **LibWS** for our web services security library composed of **LibX**, **Principals**, **Patterns**, **SOAP**, **WS-Addressing**, **XML-Signature**, **XML-Encryption**, **WS-Security**, and **WS-Trust**.

LEMMA 13 **LibWS** *is a refined module.*

Composing Cryptographic Patterns: Secure XML Request/Response Each message exchange in CardSpace implements a secure request/response protocol built on top of the web services security library. Unlike the RPC protocol of Section 3, this protocol guarantees both authentication and confidentiality, and uses many of the composite cryptographic patterns introduced in Section 5. XML flexibility also has a cost: the messages we verify are large (up to 15k) and complex (up to 17 cryptographic operations).

We describe an instance of the protocol using asymmetric keys. Assume principal *a* has a private key *ska*, *b* has a public key *pkb*, and both *a* and *b* have exchanged their public keys using X.509 certificates. The protocol below uses four cryptographic patterns implemented for XML: derived keys, hybrid encryption, sign-then-encrypt, and endorsing signatures.

Secure XML Request/Response (X.509 Mutual Authentication):

a: Generate *kab*, *n*1, *n*2

Protocols and Libraries	F# Program		F7 Typechecking		FS2PV Verification	
	Modules	Implementation	Interface	Checking Time	Queries	Verifying Time
Trusted Libraries (Symbolic)	5	926 lines*	1167 lines	29s	(Not Verified Separately)	
RPC Protocol (Section 3)	5+1	+ 91 lines	+ 103 lines	10s	4	6.65s
Principals (Section 5)	1	207 lines	253 lines	9s	(Not Verified Separately)	
Cryptographic Patterns (Section 5)	1	250 lines	260 lines	17.1s	(Not Verified Separately)	
Otway-Rees (Section 5.6)	2+1	+ 234 lines	+ 255 lines	1m 29.9s	10	8m 2.2s
Otway-Rees (No MACs)	2+1	+ 265 lines	-	(Type Incorrect)	10	2m 19.2s
Secure Conversations (Section 5.6)	2+1+1	+ 123 lines	+ 111 lines	29.64s	(Cannot Be Verified)	
Web Services Security Library	7	1702 lines	475 lines	48.81s	(Not Verified Separately)	
X.509-based Client Auth (Section 6)	7+1	+ 88 lines	+ 22 lines	+ 10.8s	2	20.2s
Password-X.509 Mutual Auth	7+1	+ 129 lines	+ 44 lines	+ 12.0s	15	44m
X.509-based Mutual Auth (Section 6)	7+1	+ 111 lines	+ 53 lines	+ 10.9s	18	51m
Windows CardSpace (Section 6)	7+1+1	+ 1429 lines	+ 309 lines	+ 6m 3s	6	66m 21s*

Figure 2. Verification Times and Comparison with ProVerif

a: Derive $k1 = psha1\ kab\ n1$, $k2 = psha1\ kab\ n2$
1. $a \to b$: $rsa\ pkb\ kab \mid n1 \mid n2$
$\mid XML\text{-}Encrypt\ k2\ S1\ (where\ S1 = XML\text{-}Sign\ k1\ [m1])$
$\mid XML\text{-}Encrypt\ k2\ S2\ (where\ S2 = XML\text{-}Sign\ ska\ [S1])$
$\mid XML\text{-}Encrypt\ k2\ m1$
b: Generate $n3$, $n4$
b: Derive $k3 = psha1\ kab\ n3$, $k4 = psha1\ kab\ n4$
2. $b \to a$: $n3 \mid n4$
$\mid XML\text{-}Encrypt\ k4\ S3\ (where\ S3 = XML\text{-}Sign\ k3\ m2)$
$\mid XML\text{-}Encrypt\ k4\ m2$

Before sending the request (message 1), a generates a fresh keyseed kab and two nonces $n1$ and $n2$. It uses kab and the nonces to derive a MAC key $k1$ and an encryption key $k2$. It signs the message $m1$ with $k1$ to obtain the XML signature $S1$, and then signs $S1$ with ska to obtain the endorsing XML signature $S2$. Finally, it separately encrypts $S1$, $S2$, and $m1$ with the encryption key $k2$. The response (message 2) is simpler; b derives two keys $k3$ and $k4$ and uses them to sign and then encrypt the response message $m2$.

The security goals are mutual authentication of a and b, plus authentication and secrecy of $m1$ and $m2$. These goals are verified by typechecking the protocol code against the web services security library **LibWS** (including **Patterns**).

LEMMA 14 **LibWS**;**SecureRPC** *is a refined module.*

In traditional protocol verification techniques, each layer of encryption or signature can add significant complexity to the proof. Indeed, when analyzing this protocol using ProVerif, each additional cryptographic pattern significantly increases the verification time. Our compositional proof technique, however, is particularly suited to verify such protocols.

Composing Protocols: CardSpace We assemble CardSpace by composing two XML request/response exchanges. To avoid repeating the message formats, we abstractly represent each request message by $Request_i\ k1\ k2\ [m1;\ldots;mn]$, where $k1$ and $k2$ are the keys of the sender and recipient (ska and pkb in the XML request/response protocol above), and $[m1;\ldots;mn]$ is the list of message elements protected by the protocol ($m1$ above). The corresponding responses are represented by $Response_i\ [m1;\ldots;mn]$.

CardSpace Protocol (using X.509 Mutual Authentication):

1. $C \to IP$: $Request_1\ skC\ pkIP\ [TokenRequest(RP, pkRP)]$
IP: Issue token $t = Token(id, C, RP, kt)$
2. $IP \to C$: $Response_1\ [t; XML\text{-}Encrypt\ pkRP\ t]$
3. $C \to RP$: $Request_2\ kt\ pkRP\ [t; m1]$
4. $RP \to C$: $Response_2\ [m2]$

In the first exchange, the client C requests a token from identity provider IP for use at RP. The IP responds with a signed token t (in the syntax of SAML), containing C's identity information id, and a key kt that C may use at RP to prove its possession of t. The IP also encrypts t for RP and sends it to C; C forwards this token in its subsequent request to RP, and uses the key kt to authenticate the request ($m1$). The RP decrypts the token t and checks IP's signature on it to convince itself of C's identity, before responding with $m2$.

The security goal of the protocol is the authentication of C's identity id at RP, and the secrecy and authentication of $m1$ and $m2$.

LEMMA 15 **LibWS**;**SecureRPC**;**CardSpace** *is a refined module.*

7. Performance Evaluation

Figure 2 summarizes our verification results for the protocols and libraries described in this paper. Each row lists the number of modules and lines of code in the F# protocol implementation, the number of lines in the F7 typed interface, and the time for verification by typechecking. The F7 interface extends the F# module interface with security assumptions, theorems, and goals, as well as type annotations needed for verification. For comparison, the table also lists, where applicable, the results of verifying the protocol implementation through the FS2PV/ProVerif tool chain: it lists the number of queries (security goals) proved and their verification time. All experiments were performed on an Intel Xeon workstation with two processors at 2.83 GHz, with 32GB memory, and running Windows Server 2008. (Most of these ProVerif results have been published in earlier work.)

The first part of the table corresponds to the RPC protocol of Section 3. The first row is for the trusted libraries **Lib**; the $*$ indicates that we verify their idealized symbolic implementation, not their concrete code. The second row is for the RPC protocol; since the libraries are verified once and for all, this row shows only the incremental lines of code and type checking for verifying **RPC**. In contrast, ProVerif verifies both **Lib** and **RPC** together. For small examples such as this, we find that the domain-specific analysis of ProVerif is faster than F7.

The second part corresponds to the libraries and protocols of Section 5. The first and second rows are for **Principals** and **Patterns**. The third row corresponds to the Otway-Rees protocol. We find that the incremental typechecking time of Otway-Rees is only 1m 29.9s, whereas ProVerif takes 8m 2.2s to verify the protocol. Even adding verification times for the libraries, we find that typechecking with F7 is much faster than ProVerif. Our typed cryptography is more realistic than typical ProVerif models; for instance it tells the difference between authenticated and unauthenticated encryption: with unauthenticated encryption, typechecking fails to verify Otway-Rees (fourth row) but ProVerif still succeeds. (Weaker assumptions can sometimes be coded in ProVerif but are not provided by default.) The protocol in the fifth row implements the unbounded secure conversations protocol. The typechecker easily verifies this recursive code, but ProVerif cannot, and fails to terminate. For recursive code, typechecking lets the programmer provide hand-written (recursive) invariants; fully automated model checkers and theorem provers (like ProVerif) lack this facility.

The third part corresponds to protocols of Section 6, arranged in increasing complexity leading up to the CardSpace protocol. The first row presents verification results for the web services security libraries **LibWS**. We then present verification results for a single-message client authentication protocol, two secure request/response protocols, and the CardSpace protocol. We find that the incremental typechecking time scales almost linearly with the size of the protocol code. In contrast, the ProVerif verification time increases exponentially with the protocol complexity (for each extra layer of encryption or signature, or each extra message). For instance, ProVerif takes less than a minute to analyze the client authentication protocol but up to an hour to verify mutual authentication protocols. The jump in analysis time is primarily because ProVerif has to account for all possible dependencies between the two messages, such as whether the adversary may use the second message of a session to compromise the first message of another session. The increase in verification complexity makes it infeasible to verify the whole CardSpace protocol using ProVerif. Indeed, in the last row of the table, the $*$ indicates that the ProVerif verification only applies when the number of clients and servers are limited to at most two each (one honest and one compromised principal for each role) and when the full XML message formats in the web services security libraries are abstractly represented as tuples. Even with these restrictions, ProVerif takes 66m 21s to verify the protocol implementation. In contrast, typechecking incrementally verifies CardSpace in a few minutes.

We conclude that typechecking scales far better than whole-program analyses for security protocols. As a trade-off, the programmer must declare their usage of cryptography by providing annotations in the typed interface of each protocol.

8. Related Work

This paper builds on the method and type system of the original F7 reported by Bengtson et al. (2008). We believe this paper is a major improvement, for the following reasons:

1. The use of semantic safety and logical invariants for cryptographic structures is new. In the original F7, we had to rely instead on global rules for kinding. For instance, the built-in kind *Public* is now replaced with a library-defined predicate *Pub*, yielding more modularity and expressivity.
2. The cryptographic libraries presented in the paper are entirely new. They support a broader range of primitives and coding patterns, which we could not encode in the style of the original F7.
3. We report verification of substantial preexisting code, not written with refinement types in mind. That was not possible with the seal-based library of the original F7, which we used only to verify sample protocols written to illustrate our type system. (To give a rough comparison, the examples verified in this paper amount to 5405 lines of code, compared to 740 lines in the original F7.)
4. We are pleased to re-use a proper subset of RCF, obtained by eliminating kinds (saving the need to develop yet another dependent type system). Kinds are the only security-specific feature of RCF. Hence, our new libraries can also be used from other languages, in conjunction with any general-purpose verification tool that can check pre- and post-conditions. This is a big improvement over any specialized cryptographic tool, not just ProVerif or RCF with kinds.

Code Verification for Cryptographic Protocols We discuss some approaches to code verification for security protocols not mentioned in Section 1.

Poll and Schubert (2007) verify safety properties of an implementation of SSH in Java. They show that the Java code implements the protocol as defined by finite state machines based on the SSH specification. Their analysis shows that the code never throws an exception. Pistachio (Udrea et al. 2008) checks C code, including code for SSH, against rules describing its intended behaviour. These tools are aimed at showing compliance with protocol specifications, rather than to directly show security properties of the code.

Elyjah (O'Shea 2008) extracts Lysa models from Java implementations of some abstract protocols.

FS2CV (Bhargavan et al. 2008a) is the first tool to verify properties in the computational model of implementation code of security protocols. FS2CV generates inputs to CryptoVerif (Blanchet 2006) from the implementation code in F#. It has been applied to an F# implementation of TLS.

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 (up to three servers and clients).

Backes et al. (2009, 2010) extend RCF with intersection and union types to check code that uses zero-knowledge proofs. Their work extends the original theory including its use of public and tainted kinds.

Code Verification for Cryptographic Algorithms Our work targets cryptographic protocols, while assuming the correct implementation of the underlying cryptographic algorithms. Domain specific languages such as Cryptol (Lewis 2007; Pike et al. 2006) and CAO (Barbosa et al. 2005) support the development of verified implementations of cryptographic algorithms.

Extraction of Code from Verified Models Our stance is to verify user-written code in a general purpose language, rather than to build a compiler for some custom description language designed for ease of verification. Still, there are several studies (Lukell et al. 2003; Muller and Millen 2001; Perrig et al. 2001; Pozza et al. 2004) of how to extract executable code from verified models of cryptographic protocols. Mukhamedov et al. (2009) show how to extract C, suitable for direct unmanaged execution, from F# code verified with FS2PV and ProVerif. The most accomplished work in this direction is by Pironti (2010), whose tool, spi2java, produces implementations—of some standard protocols, including SSH and TLS—that pass interoperability tests with several preexisting implementations. Pironti also develops runtime filters that pre-existing implementations only send messages in compliance with verified descriptions of protocols.

Bhargavan et al. (2009) develop a a high-level graphical notation for describing multiparty sessions; their compiler synthesises a suitable cryptographic protocol, and compiles to ML code whose security properties are verified using F7.

Refinement Types The RCF system of refinement types is similar to that of systems such as DML (Xi 2007), SAGE (Flanagan 2006) and Dsolve (Rondon et al. 2008), although neither of these systems allows full first-order formulas as refinements. Still, we expect with a little adaptation tools such as these could support our method.

In future work, we aim to reduce the annotation burden by applying techniques from prior studies of inference for refinement types restricted to base types (Knowles and Flanagan 2007; Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009). In particular, a recent paper (Bhargavan et al. 2010) makes progress towards type inference for F7. Techniques for inferring types and effects in cryptographic models (Kikuchi and Kobayashi 2007) may also be useful.

Other recent dependently typed systems for security, if not for writing cryptographic protocols, include Aura (Jia et al. 2008), Fable (Swamy et al. 2008), and Fine (Swamy et al. 2010).

9. Conclusions

We proposed a modular, compositional approach to verifying the code of security protocols. We have empirical evidence that the method scales better than the best prior work, a whole-program analysis relying on ProVerif.

With the intent to verify security properties of protocol code, we developed a method of invariants for cryptographic structures in the setting of a formal model of cryptography. For the purpose of a direct comparison with prior work on whole-program analysis of security protocol code, we worked with the formal model implemented by FS2PV and ProVerif.

In future, we may consider alternative, more accurate formal models, such as ones sensitive to message length. Another natural next step is to recast our method in the computational model.

Acknowledgements Aslan Askarov, François Dupressoir, Nataliya Guts, and Cătălin Hriţcu suggested improvements to the paper.

A. The Core Library (Lib)

This appendix describes the library **Lib**, which is the composition of **Data**, **Net**, and **Crypto**. The library is based on one developed for use with F# and the FS2PV tool (Bhargavan et al. 2008c).

The interface exported by the library specifies a collection of operations, including cryptographic algorithms and functions for network-based communication, on abstract types of strings, byte arrays, and keys:

- str is the type of text strings;
- bytes is the type of variable-length byte arrays;
- key is the type of cryptographic keys.

We use this interface for writing additional libraries and reference implementations of protocols.

The library interface has two distinct implementations. One implementation relies on actual cryptography and is used for execution, for interoperability testing or actual production use. The other library implementation is a symbolic model of cryptography in terms of algebraic types for strings and bytes, in the style of Dolev and Yao (1983). This symbolic implementation is the formal basis for verification.

Our verification results hold in spite of an attacker in possession of *public* data, which includes all messages exchanged by protocol participants, and also the key material and other private data known to any principals that become compromised. To this end, we model the attacker as an arbitrary F# program with access to an *attacker interface* providing operations on the following abstract types:

- strpub is the type of public text strings;
- bytespub is the type of public byte arrays;
- keypub is the type of public cryptographic keys.

We view the attacker as well-typed F# code that manipulates these abstract types of strings, bytes, and keys only via the functions exported in the attacker interface. Although the attacker accesses the same symbolic implementation code as the trusted protocol code, the type assigned to each function in the attacker interface is expressed in terms of the public types above, and is a supertype of the type exposed to the trusted protocol code.

In the following, for each group of cryptographic primitives, we give both the programming interface (for verified protocol code), and the attacker interface (modelling its capabilities). The types, logical assumptions, and functions labelled **private** are available only when typechecking the library implementation, and are exported neither to the protocol code nor to the attacker.

A.1 Strings and Byte Arrays

At the core of our model are the following algebraic types, inherited from FS2PV, for symbolic cryptography. (We explain our representation of keys in the next section.) We use a primitive type *Pi.name*, whose values are atoms in the style of pi calculus names; the only operations on names are to test for equality and to freshly generate new names.

Underlying Type of Strings and Bytes:

```
type dstr =
  | Literal of string
  | Base64 of dbytes
and dbytes =
  | Concat of dbytes * dbytes
  | Utf8 of dstr
  | Fresh of Pi.name
  | Bin of blob
and blob =
  | Hash of dbytes
  | DerivedKey of dbytes * dbytes
  | DerivedSKey of dbytes * dbytes
  | MAC of dbytes * dbytes
  | SymEncrypt of dbytes * dbytes
  | PK of dbytes
  | AsymSign of dbytes * dbytes
  | AsymEncrypt of dbytes * dbytes
  | X509Cert of dstr * dstr * dstr * dstr * dbytes
```

(The auxiliary type *blob* gathers the *dbytes* constructors meant to be opaque, such as *Hash*, in contrast with those meant to be transparent, such as *Concat*.)

We are not concerned with all possible values of these types, but only those that preserve certain invariants. (For example, as described in Section 3 we only consider a value $Bin(MAC(k,b))$ of type *dbytes* when k is a MAC key and the intuitive logical payload $MACSays(k,b)$ holds.) We represent these invariants by predicates with the following intended meanings:

- $String(s)$ holds when string s appears in the protocol run;
- $Bytes(b)$ holds when bytes b appear in the protocol run;
- $Pub(x)$ holds when the data x may be known to the opponent. (This predicate is overloaded in that x may have type *dstr*, *dbytes*, and indeed other types introduced below.)

The predicates *String*, *Bytes*, and *Pub* are the least relations closed under the inductive rules in the tables in the remainder of this appendix.

Our types of strings and bytes are defined as follows:

Types with Invariants:

```
type str = s:dstr {String(s)}
type bytes = b:dbytes {Bytes(b)}
type strpub = s:str {Pub(s)}
type bytespub = x:bytes {Pub(x)}
```

The types str and bytes represent data manipulated by known protocol code, while the types strpub and bytespub are the implementations of abstract types of public strings and public bytes manipulated by the unknown attacker. By construction, we have the following subtype relationships, that bytespub <: bytes and strpub <: str. Moreover, we can show that $Pub(b)$ implies $Bytes(b)$ when $b : dbytes$, and that $Pub(s)$ implies $String(s)$ when $s : dstr$.

The programming interface hides the implementation of *dstr*, *dbytes*, and the full definitions of the predicates *String* and *Bytes*, but exports the refinement type definitions shown above, together with a set of functions acting on these types. In other words, protocol code cannot directly access the constructors (like *Literal*, *Concat*, *Hash*, and so on) either to create new values or to pattern-match existing data. The attacker sees a more limited interface, just the abstract types strpub and bytespub, and the functions in the Attacker Interfaces listed below.

A.2 Cryptographic Keys

The FS2PV library relies on an abstract type to package the byte arrays used as cryptographic keys. This is the type of all keys used as parameters of cryptographic operations. By distinguishing key material from other byte arrays, we prevent some basic programming errors (although we obtain no strong security guarantees from this distinction). Keys are implemented as an algebraic type, with a constructor for each kind of key:

Type of Tagged Keys:

```
type key =
  | SymKey of bytes
  | AsymPrivKey of bytes
  | AsymPubKey of bytes
```

- $SymKey(b)$ contains the bytes b of a key used for symmetric encryption or for keyed cryptographic hashes.
- $AsymPrivKey(b)$ contains the bytes b of the private part of a key pair, for signing or for decrypting.
- $AsymPubKey(b)$ contains the bytes b of the public part of a key pair, for verifying signatures or for encrypting.

The following inductive rules define the public predicate $Pub(x)$ when x is a key.

Inductive Rules:

```
(Pub SymKey)
  ∀b. Pub(b) ⇒ Pub(SymKey(b))
(Pub AsymPubKey)
  ∀b. Pub(b) ⇒ Pub(AsymPubKey(b))
(Pub AsymPrivKey)
  ∀b. Pub(b) ⇒ Pub(AsymPrivKey(b))
```

By inspection of the other inductive clauses that define *Pub*, we prove the following inversion theorems.

Theorems:

```
(Pub SymKey)
  ∀b. Pub(SymKey(b)) ⇒ Pub(b)
(Pub AsymPubKey)
  ∀b. Pub(AsymPubKey(b)) ⇒ Pub(b)
(Pub AsymPrivKey)
  ∀b. Pub(AsymPrivKey(b)) ⇒ Pub(b)
```

Having extended the *Pub* predicate to the key type, we implement the abstract type keypub, of keys known to the attacker, with the following refinement type.

Types with Invariants:

```
type keypub = k:key { Pub(k) }
```

We have introduced the key type deliberately to reduce the abilities of protocol code accidentally to use arbitrary byte arrays as key material. Still, we need to avoid restricting the abilities of the symbolic attacker, who can use byte arrays as they wish. Hence, as part of the attacker interface, we provide the following functions, to allow the attacker to access the underlying bytes within a key, and also to turn any bytes into a key tagged with any of the three key constructors.

Attacker Interface:

```
val symkey: bytespub → keypub
val asympubkey: bytespub → keypub
val asymprivkey: bytespub → keypub
val bytesofkey: keypub → bytespub
```

Typechecking these functions against their public interface relies on the six logical implications listed above; for instance (*Pub SymKey*) enables us to type **let** *symkey x* = *SymKey*(*x*).)

A.3 Encodings: Strings, Unicode, and Base64

The functions in the programming interface below deal with common message formats. For instance, *base64* and *utf8* are standard encodings; whereas *ibase64* and *iutf8* are their partial inverse (they throw an exception if decode fails). The functions str and *istr* translate between strings and the refined str datatype.

The programming interface includes refinements, using predicates with the following intended meanings.

- $IsLiteral(s,l)$ holds when string s represents the literal l;
- $IsBase64(s,b)$ holds when string s is the Base64 encoding of the bytes b;
- $IsUtf8(b,s)$ holds when bytes b are the Utf8 encoding of the string b.

Programming Interface:

```
val str: l:string → s:strpub{IsLiteral(s,l)}
val istr: s:str → l:string{IsLiteral(s,l)}
val base64: b:bytes → s:str{IsBase64(s,b)}
val ibase64: s:str → b:bytes{IsBase64(s,b)}
val utf8: s:str → b:bytes{IsUtf8(b,s)}
val iutf8: b:bytes → s:str{IsUtf8(b,s)}
```

For example, using this interface, F7 verifies that the function **fun** *s* → *iutf8*(*utf8*(*s*)) can be typed as *s*: str → *s*':str{ *s* = *s*' }.

The corresponding attacker interface is as follows.

Attacker Interface:

```
val str: string → strpub
val istr: strpub → string
val base64: bytespub → strpub
val ibase64: strpub → bytespub
val utf8: strpub → bytespub
val iutf8: bytespub → strpub
```

The symbolic implementation relies on the following internal representations (with constructors defined in Section A.1), recorded in the logical definition of the three predicates above.

- String $Literal(c)$ represents a string constant c.
- String $Base64(b)$ represents the Base64 encoding of bytes b.
- Bytes $Utf8(s)$ represents the Utf8 encoding of string s.

Equational Abbreviations:

(IsLiteral)
$\forall s,l.\ \mathit{IsLiteral}(s,l) \Leftrightarrow s{=}\mathit{Literal}(l)$
(IsBase64)
$\forall s,b.\ \mathit{IsBase64}(s,b) \Leftrightarrow s{=}\mathit{Base64}(b)$
(IsUtf8)
$\forall b,s.\ \mathit{IsUtf8}(b,s) \Leftrightarrow b{=}\mathit{Utf8}(s)$

The inductive rules below define the predicates *String*, *Bytes*, and *Pub*, on strings of the form $\mathit{Literal}(c)$ and $\mathit{Base64}(b)$ and bytes of the form $\mathit{Utf8}(s)$.

Inductive Rules:

private (String Literal)
$\forall c.\ \mathit{String}(\mathit{Literal}(c))$
private (String Base64)
$\forall b.\ \mathit{Bytes}(b) \Rightarrow \mathit{String}(\mathit{Base64}(b))$
private (Bytes Utf8)
$\forall s.\ \mathit{String}(s) \Rightarrow \mathit{Bytes}(\mathit{Utf8}(s))$
(Pub Literal)
$\forall c.\ \mathit{Pub}(\mathit{Literal}(c))$
(Pub Base64)
$\forall b.\ \mathit{Pub}(b) \Rightarrow \mathit{Pub}(\mathit{Base64}(b))$
(Pub Utf8)
$\forall s.\ \mathit{Pub}(s) \Rightarrow \mathit{Pub}(\mathit{Utf8}(s))$

Theorems:

private (Bytes Base64)
$\forall b.\ \mathit{String}(\mathit{Base64}(b)) \Rightarrow \mathit{Bytes}(b)$
private (String Utf8)
$\forall s.\ \mathit{Bytes}(\mathit{Utf8}(s)) \Rightarrow \mathit{String}(s)$
(Pub Base64)
$\forall b.\ \mathit{Pub}(\mathit{Base64}(b)) \Rightarrow \mathit{Pub}(b)$
(Pub Utf8)
$\forall s.\ \mathit{Pub}(\mathit{Utf8}(s)) \Rightarrow \mathit{Pub}(s)$

The logical assumptions on *Bytes* and *String* are marked as private, since the corresponding types are abstract after typechecking the libraries. Conversely, the assumptions on *Pub* are visible when typechecking protocol code, and used for instance to show that the messages they form are public.

A.4 Concatenation

The programming interface relies on the following predicate.

- $\mathit{IsConcat}(c,b_1,b_2)$ holds when the bytes c represent b_1 paired with b_2, with sufficient length information to retrieve b_1 and b_2.

Programming Interface:

val *concat*: *b1*:bytes → *b2*:bytes → *c*:bytes{ $\mathit{IsConcat}(c,b1,b2)$}
val *iconcat*: *c*:bytes → (*b1*:bytes ∗ *b2*:bytes){ $\mathit{IsConcat}(c,b1,b2)$ }

The corresponding attacker interface is as follows.

Attacker Interface:

val *concat*: bytespub → bytespub → bytespub
val *iconcat*: bytespub → bytespub ∗ bytespub

The symbolic implementation relies on the following representation, with corresponding logical clauses.

- Bytes $\mathit{Concat}(b_1,b_2)$ represents the concatenation of the bytes b_1 and b_2.

Equational Abbreviations:

(IsConcat)
$\forall c,b1,b2.\ \mathit{IsConcat}(c,b1,b2) \Leftrightarrow c{=}\mathit{Concat}(b1,b2)$

Inductive Rules:

private (Bytes Concat)
$\forall b1,b2.\ \mathit{Bytes}(b1) \land \mathit{Bytes}(b2) \Rightarrow \mathit{Bytes}(\mathit{Concat}(b1,b2))$
(Pub Concat)
$\forall b1,b2.\ \mathit{Pub}(b1) \land \mathit{Pub}(b2) \Rightarrow \mathit{Pub}(\mathit{Concat}(b1,b2))$

Theorems:

(Bytes Concat Invert)
$\forall b1,b2.\ \mathit{Bytes}(\mathit{Concat}(b1,b2)) \Rightarrow (\mathit{Bytes}(b1) \land \mathit{Bytes}(b2))$
(Pub Concat Invert)
$\forall b1,b2.\ \mathit{Pub}(\mathit{Concat}(b1,b2)) \Rightarrow (\mathit{Pub}(b1) \land \mathit{Pub}(b2))$
(Concat Injective)
$\forall b1,b2,b3,b4.\ \mathit{Concat}(b1,b2) = \mathit{Concat}(b3,b4) \Rightarrow b1{=}b3 \land b2{=}b4$

A.5 Fresh Bytes

Next, we explain the generation of fresh values, such as nonces or different sorts of key. The programming interface uses the following predicate to record the usage.

- $\mathit{FreshBytes}(b,u)$ holds when the bytes b have been freshly generated with intended usage u.

The usage datatype lists all such kinds of byte arrays. Calling the function *freshbytes* with usage u generates the event *FreshBytes* (b,u). This event applies only to freshly created byte arrays. So no fresh byte array satisfies more than one usage. The function *freshbytes* is used to implement other functions in the library, such as those for creating nonces or keys.

Programming Interface:

type usage =
| *KeySeedName*
| *MKeyName*
| *SKeyName*
| *SingleUseKeyName* **of** bytes
| *PKeyName*
| *PasswordName*
| *GuidName*
| *NonceName*
| *AttackerName*
val *freshbytes* : *u*:usage → string → *b*:bytes{$\mathit{FreshBytes}(b,u)$}

The following function allows the attacker to generate fresh byte arrays. Each call to *mkbytespub* returns the result of the expression *freshbytes AttackerName* `"attacker"`.

Attacker Interface:

val *mkbytespub*: unit → bytespub

Rather than use randomized generation of actual byte arrays, our symbolic implementation uses abstract new names, and also records the intended usage of each fresh value. If $a : \mathit{Pi.name}$ is a freshly generated name, then the value $\mathit{Fresh}(a)$: bytes represents a randomly generated byte array.

- Bytes $\mathit{Fresh}(a)$ such that $\mathit{FreshBytes}(\mathit{Fresh}(a),u)$ represents a randomly generated byte array with usage u.

In general, proofs about clients of the library are on the basis of the pre- and post-conditions of functions. When reasoning about freshness, however, it is convenient to expose implementation detail. In particular, we expose the symbolic implementation of the *freshbytes* function as follows. (The string parameter s is ignored; we have an alternative implementation which uses s for debugging purposes.)

Transparency Theorem:

$\mathbf{let}\, \mathit{freshbytes}\ u\ s = (\nu a)\mathbf{assume}\ \mathit{FreshBytes}(\mathit{Fresh}(a),u);\mathit{Fresh}(a)$

The equation above is an example of a *transparency theorem*, an equation of the form $f = A$, where A is the implementation code for the value f. A transparency theorem is simply a logical formula exported by the module, but since its purpose is to export the implementation code of a function it is convenient to state the theorem using the code itself. We read a function definition **let** $f\ x = B$ as defining the transparency theorem $f = \mathbf{fun}\, x \to B$.

In particular, the equation above exposes that our symbolic implementation of key generation relies on the restriction primitive, $(\nu a)A$, a primitive inherited by RCF from the pi calculus, and whose operational semantics picks the name a to be fresh and globally unique.

The following inductive rule asserts that the bytes generated by the attacker are always public.

Inductive Rules:

(Pub Attacker)
$\forall n.\ \mathit{FreshBytes}(\mathit{Fresh}(n),\mathit{AttackerName}) \Rightarrow \mathit{Pub}(\mathit{Fresh}(n))$

We have the following theorems concerning the *FreshBytes* predicate. They follow from the fact that the only way for *FreshBytes* to hold is following a call to the function *freshbytes*.

Theorems:

private (Bytes Fresh)
$\forall b,u.\ \mathit{FreshBytes}(b,u) \Rightarrow \mathit{Bytes}(b)$
(Name Constraint)
$\forall b,u,u'.\ \mathit{FreshBytes}(b,u) \land \mathit{FreshBytes}(b,u') \Rightarrow u{=}u'$
(FreshBytes Fresh)
$\forall b,u.\ \mathit{FreshBytes}(b,u) \Rightarrow \exists n.\ b = \mathit{Fresh}(n)$

A.6 Nonces

Nonces represent large, fresh values generated at random. They are initially secret. The postcondition of functions that generate fresh nonces is the following:

- $\mathit{Nonce}(b)$ holds iff bytes b satisfy $\mathit{FreshBytes}(b,\mathit{NonceName})$.

Equational Abbreviations:

private (Nonce)
$\forall b.\ \mathit{Nonce}(b) \Leftrightarrow \mathit{FreshBytes}(b,\mathit{NonceName})$

Programming Interface:

$\mathbf{val}\ \mathit{mkNonce}\colon \mathrm{unit} \to b{:}\mathrm{bytes}\{\mathit{Nonce}(b)\}$
$\mathbf{val}\ \mathit{mkNonce256}\colon \mathrm{unit} \to b{:}\mathrm{bytes}\{\mathit{Nonce}(b)\}$

There is no specific attacker interface for nonces, as the attacker can use *mkpubbytes* from Appendix A.5 to create fresh byte arrays.

We have the following representation in the symbolic implementation of our library.

- Bytes $\mathit{Fresh}(n)$ such that $\mathit{FreshBytes}(\mathit{Fresh}(n),\mathit{NonceName})$ represents a random nonce.

The inductive rule below allows nonces to become public only when the user-defined predicate *PubNonce* holds.

Inductive Rules:

(Pub Nonce)
$\forall b.\ \mathit{Nonce}(b) \land \mathit{PubNonce}(b) \Rightarrow \mathit{Pub}(b)$

Theorems:

(Inv Pub Nonce)
$\forall b.\ \mathit{Nonce}(b) \land \mathit{Pub}(b) \Rightarrow \mathit{PubNonce}(b)$

A.7 Message Authentication Codes (MACs)

We support the keyed hash algorithm HMACSHA1 for generating and verifying MACs, and also provide an algorithm for deriving keys from a secret seed (such as a strong password).

- $\mathit{MKey}(k)$ means that k is a valid MAC key, that is, either k has been generated pseudo-randomly with *hmac_keygen* or by key derivation from an existing key with *sha1*.
- $\mathit{MACSays}(k,b)$ means that the logical property to be conveyed by key k holds of the bytes b. This predicate is defined by clients of the library.
- $\mathit{IsMAC}(h,k,b)$ means that the bytes h match the outcome of applying the MAC algorithm to bytes b with key k.

Programming Interface:

$\mathbf{val}\ \mathit{hmac\_keygen}\colon \mathrm{unit} \to k{:}\mathrm{key}\{\mathit{MKey}(k)\}$
$\mathbf{val}\ \mathit{hmacsha1}\colon$
$k{:}\mathrm{key} \to$
$b{:}\mathrm{bytes}\{\ (\mathit{MKey}(k) \land \mathit{MACSays}(k,b)) \lor (\mathit{Pub}(k) \land \mathit{Pub}(b))\ \} \to$
$h{:}\mathrm{bytes}\{\ \mathit{IsMAC}(h,k,b) \land (\mathit{Pub}(b) \Rightarrow \mathit{Pub}(h))\ \}$
$\mathbf{val}\ \mathit{hmacsha1Verify}\colon$
$k{:}\mathrm{key}\{\mathit{MKey}(k) \lor \mathit{Pub}(k)\} \to b{:}\mathrm{bytes} \to h{:}\mathrm{bytes} \to \mathrm{unit}\{\mathit{IsMAC}(h,k,b)\}$
$\mathbf{val}\ \mathit{hmac\_keyseed}\colon \mathrm{unit} \to b{:}\mathrm{bytes}\{\mathit{KeySeed}(b)\}$
$\mathbf{val}\ \mathit{psha1}\colon$
$b1{:}\mathrm{bytes}\{\mathit{KeySeed}(b1) \lor \mathit{Pub}(b1)\} \to$
$b2{:}\mathrm{bytes} \to$
$k{:}\mathrm{key}\{\mathit{IsDerivedKey}(k,b1,b2)\}$

Attacker Interface:

$\mathbf{val}\ \mathit{hmacsha1} : \mathrm{keypub} \to \mathrm{bytespub} \to \mathrm{bytespub}$
$\mathbf{val}\ \mathit{hmacsha1Verify} : \mathrm{keypub} \to \mathrm{bytespub} \to \mathrm{bytespub} \to \mathrm{unit}$
$\mathbf{val}\ \mathit{psha1} : \mathrm{bytespub} \to \mathrm{bytespub} \to \mathrm{keypub}$

We have the following representations in the symbolic implementation of our library.

- Bytes $\mathit{Fresh}(n)$ such that $\mathit{FreshBytes}(\mathit{Fresh}(n),\mathit{MKeyName})$ represents a pseudorandom MAC key.
- Bytes $\mathit{Fresh}(n)$ such that $\mathit{FreshBytes}(\mathit{Fresh}(n),\mathit{KeySeedName})$ represents a pseudorandom keyseed.
- Bytes $\mathit{Bin}(\mathit{DerivedKey}(b,b_s))$ where b_s is a pseudorandom keyseed, represents a MAC key derived from b_s via b.
- Bytes $\mathit{Bin}(\mathit{MAC}(b_k,b_p))$ represents the MAC of b_p with MAC key b_k.

The following transparency theorem exposes that the symbolic implementation of *hmac_keygen* relies on the *freshbytes* function, which itself uses the RCF restriction operator to model a freshly generated key as a new name.

Transparency Theorem:

$\mathbf{let}\ \mathit{hmac\_keygen}\ () =$
$\quad \mathbf{let}\ \mathit{kb} = \mathit{freshbytes}\ \mathit{MKeyName}$ `"hkey"` $\mathbf{in}$
$\qquad \mathit{SymKey}(\mathit{kb})$

Given these representations, we make the following predicate definitions. Predicates *Data.IsMAC*, *Data.IsDerivedKey*, *IsMAC*, and *IsDerivedKey* capture the syntactic structure of MACs and derived keys. The predicate *MACVerified* tracks verified MAC payloads.

Equational Abbreviations:

(Data.IsMAC)
$\forall m,k,b.\ \mathit{Data.IsMAC}(m,k,b) \Leftrightarrow m{=}\mathit{Bin}(\mathit{MAC}(k,b))$
(Data.IsDerivedKey)
$\forall k,n1,n2.\ \mathit{Data.IsDerivedKey}(k,n1,n2) \Leftrightarrow k{=}\mathit{Bin}(\mathit{DerivedKey}(n1,n2))$

private (IsMAC)
$\forall m,k,b.\ \mathit{IsMAC}(m,k,b) \Leftrightarrow \exists kb.\ k{=}\mathit{SymKey}(kb) \land \mathit{Data.IsMAC}(m,kb,b)$

private (MACVerified)
$\forall k,b.\ \mathit{MACVerified}(k,b) \Leftrightarrow$
$\mathit{Bytes}(b) \land (\mathit{MACSays}(k,b) \lor (\mathit{Pub}(k) \land \mathit{MKey}(k)))$

private (MCompKey)
$\forall k.\ \mathit{MCompKey}(k) \Leftrightarrow (\mathit{MKey}(k) \land \mathit{Pub}(k))$

private (IsDerivedKey)
$\forall k,b1,b2.\ \mathit{IsDerivedKey}(k,b1,b2) \Leftrightarrow$
$\exists b.\ k{=}\mathit{SymKey}(b) \land \mathit{Data.IsDerivedKey}(b,b1,b2)$

private (KeySeed)
$\forall b.\ \mathit{KeySeed}(b) \Leftrightarrow \mathit{FreshBytes}(b,\mathit{KeySeedName})$

Inductive Rules:

private (MKey MKeyName)
$\forall b.\ \mathit{FreshBytes}(b,\mathit{MKeyName}) \Rightarrow \mathit{MKey}(\mathit{SymKey}(b))$
(Pub MKey)
$\forall k.\ \mathit{MKey}(k) \land (\forall b.\ \mathit{MACSays}(k,b)) \Rightarrow \mathit{Pub}(k)$

private (Bytes IsMAC)
$\forall m,k,b.\ \mathit{MKey}(k) \land \mathit{MACVerified}(k,b) \land \mathit{IsMAC}(m,k,b) \Rightarrow \mathit{Bytes}(m)$
(Pub IsMAC)
$\forall m,k,b.\ \mathit{Pub}(b) \land \mathit{IsMAC}(m,k,b) \land \mathit{MKey}(k) \land \mathit{MACVerified}(k,b)$
$\Rightarrow \mathit{Pub}(m)$

private (Bytes IsMAC Pub)
$\forall m,kb,b.\ \mathit{Bytes}(kb) \land \mathit{Pub}(\mathit{SymKey}(kb)) \land \mathit{Bytes}(b) \land \mathit{Pub}(b) \land$
$\mathit{IsMAC}(m,\mathit{SymKey}(kb),b) \Rightarrow \mathit{Bytes}(m)$

(Pub IsMAC Pub)
$\forall m,k,b.\ \mathit{Pub}(b) \land \mathit{IsMAC}(m,k,b) \land \mathit{Pub}(k) \Rightarrow \mathit{Pub}(m)$

The bytes clauses for MAC cover two construction cases, by the protocol and by the adversary, respectively. The public clauses for MAC conservatively specify that MACs never protect the secrecy of *b*, only its integrity. The public clause for *MKey* states that a valid MAC key becomes public only if it is explicitly leaked by the protocol, which is tracked by the predicate *MCompKey*. The definition of *MACVerified* covers two cases: either this is a genuine text for the protocol, or the key is public.

Additional Inductive Rules for Derived Keys:

private (Bytes IsDerivedKey)
$\forall b1,b2,b.\ \mathit{Bytes}(b1) \land \mathit{Bytes}(b2) \land \mathit{Data.IsDerivedKey}(b,b1,b2)$
$\Rightarrow \mathit{Bytes}(b)$
(Pub IsDerivedKey)
$\forall b1,b2,k.\ \mathit{Pub}(b1) \land \mathit{Bytes}(b2) \land \mathit{Data.IsDerivedKey}(k,b1,b2)$
$\Rightarrow \mathit{Pub}(k)$

(Pub KeySeed)
$\forall ks.\ \mathit{KeySeed}(ks) \land$
$(\forall k,n.\ \mathit{IsDerivedKey}(k,ks,n) \Rightarrow \forall b.\ \mathit{MACSays}(k,b)) \land$
$(\forall k,n.\ \mathit{IsDerivedSKey}(k,ks,n) \Rightarrow (\forall b.\ \mathit{CanSymEncrypt}(k,b) \Rightarrow \mathit{Pub}(b)))$
$\Rightarrow \mathit{Pub}(ks)$

private (MKey IsDerivedKey)
$\forall b1,b2,k.\ \mathit{KeySeed}(b1) \land \mathit{Bytes}(b2) \land \mathit{IsDerivedKey}(k,b1,b2) \Rightarrow \mathit{MKey}(k)$

Theorems:

(IsMAC Injective)
$\forall m,k,k',b,b'.\ \mathit{IsMAC}(m,k,b) \land \mathit{IsMAC}(m,k',b') \Rightarrow k{=}k' \land b{=}b'$

(MKey Inversion)
$\forall k.\ \mathit{MKey}(k) \Rightarrow$
$(\exists kb.\ k = \mathit{SymKey}(kb) \land \mathit{FreshBytes}(kb,\mathit{MKeyName})) \lor$
$(\exists b1,b2.\ \mathit{KeySeed}(b1) \land \mathit{Bytes}(b2) \land \mathit{IsDerivedKey}(k,b1,b2))$

(IsMAC MACVerified)
$\forall m,k,b.\ \mathit{IsMAC}(m,k,b) \land \mathit{Bytes}(m) \land \mathit{MKey}(k) \Rightarrow \mathit{MACVerified}(k,b)$

(Inv MKey Pub)
$\forall k,b.\ \mathit{Pub}(k) \land \mathit{MKey}(k) \Rightarrow \mathit{MACSays}(k,b)$

(MKey Inversion) states that possession of a MAC with a valid key (e.g., just after a MAC verification) entails that its payload *b* is valid. (MKey MCompKey) states that a MAC key is public only if it has been compromised.

A.8 Network Operations

We list (most of) our programming interface for networking. The interface requires that all exchanged messages be public; it can also be used by the attacker. (For simplicity, we use the plain string type instead of strpub for network addresses and port numbers.)

Programming Interface (and Attacker Interface):

type port = *A* **of** string $*$ string
type conn = *C* **of** string

val *http*: string → string → port
val *connect*: port → conn
val *listen*: port → conn
val *close*: conn → unit

val *send*: conn → bytespub → unit
val *recv*: conn → bytespub

A.9 Proofs of Lemmas 6 and 7

RESTATEMENT OF LEMMA 6
Lib $= (\varnothing,$ ***assume*** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}, I_L^7)$ *is a refined module.*

PROOF: Recall the notations:

- Let *Lib* be the F# code for the library.
- Let I_L^7 consist of the declarations displayed as Programming Interface in this appendix.
- Let I_L consist of the declarations displayed as Attacker Interface in this appendix.
- Let $\mathbf{Lib}^{def}$ consist of all the formulas displayed as Inductive Rules in this appendix.

- Let $\mathbf{Lib}^{thm}$ consist of all the formulas displayed as Theorems in this appendix.

To show that **Lib** is a refined module, it suffices to show:

(1) **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$ is factual;
(2) $\varnothing, \mathbf{Lib}^{def}, \mathbf{Lib}^{thm} \vdash \mathit{Lib} \leadsto I_L^7$;
(3) $\mathbf{Lib}^{thm}$ is a contextual theorem of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$.

For (1), we have by construction that each of the assumptions (active or not) of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$ is a logic program, which is to say that **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$ is factual.

For (2), we have $\varnothing, \mathbf{Lib}^{def}, \mathbf{Lib}^{thm} \vdash \mathit{Lib} \leadsto I_L^7$ by running F7.

For (3), we begin by noting that the following formulas are contextual theorems of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$.

private (Bytes Fresh)
$\forall b,u.\ \mathit{FreshBytes}(b,u) \Rightarrow \mathit{Bytes}(b)$
(Name Constraint)
$\forall b,u,u'.\ \mathit{FreshBytes}(b,u) \wedge \mathit{FreshBytes}(b,u') \Rightarrow u{=}u'$
(FreshBytes Fresh)
$\forall b,u.\ \mathit{FreshBytes}(b,u) \Rightarrow \exists n.\ b = \mathit{Fresh}(n)$

Let R be the conjunction of (Name Constraint) and (FreshBytes Fresh). We must show that R is a theorem of **assume** $\mathbf{Lib}^{def} \Rsh Z[\mathit{Lib}[A]]$ whenever Z and A are factual and independent of the expression **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$. The only occurrence of the predicate *FreshBytes* in *Lib* is in the following definition of *freshbytes*, displayed in Appendix A.7 as a transparency theorem.

let $\mathit{freshbytes}\ u\ s = (\nu a)$**assume** $\mathit{FreshBytes}(\mathit{Fresh}(a),u); \mathit{Fresh}(a)$

Since each call to the function *freshbytes* generates a fresh name a (disjoint from any previous name), the two conjuncts of R hold in all reachable states.

Let P be the conjunction of the formulas in $\mathbf{Lib}^{def}$ apart from those in R. Note that $\mathbf{Lib}^{def}$ is a logic program with support disjoint from that of *Lib*. By Lemma 5 (Contextual), to prove that $R \Rightarrow P$ is a contextual theorem of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$, it suffices to show that for all Q independent of $\mathbf{Lib}^{def}$, the least interpretation of $\mathbf{Lib}^{def} \wedge Q$ satisfies $R \Rightarrow P$. We can prove this by assuming R, and proving each conjunct of P individually. We have mechanised the proofs using the Coq proof assistant. By interpreting the formulas $\mathbf{Lib}^{def}$ as inductive definitions and the conjuncts of R as logical parameters we have built a Coq module with proofs for all the theorems in P. Thus, we obtain that both R and $R \Rightarrow P$ are contextual theorems of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$, that therefore that $\mathbf{Lib}^{thm}$ is a contextual theorem of **assume** $\mathbf{Lib}^{def} \Rsh \mathit{Lib}$. □

RESTATEMENT OF LEMMA 7
RPC *is a refined module.*

PROOF: The proof is similar to that of the previous lemma. The code of **RPC** is checked by running F7. We have mechanised proofs using the Coq proof assistant. The following are the contextual theorems that need to be proved by code inspection.

(KeyAB Injective)
$\forall k,a,b,a',b'.\ \mathit{KeyAB}(k,a,b) \wedge \mathit{KeyAB}(k,a',b') \Rightarrow (a{=}a') \wedge (b{=}b')$

(not every message is a request or response)
$\exists v.\ \forall s,s',t'.\ \mathit{not}\ (\mathit{Requested}(v,s)) \wedge \mathit{not}\ (\mathit{Responded}(v,s',t'))$

The only occurrence of the predicate *KeyAB* in **RPC** is in the following definition of *mkkeyAB* (displayed already in Section 3.3).

let *mkKeyAB a b* =
  let k = *hmac_keygen*() **in assume** $(\mathit{KeyAB}(k,a,b))$; k

The transparency theorems of Appendix A.7 constrain the implementation of the function *hmac_keygen* in the **Lib** interface to be in terms of code using a restriction to generate a fresh name. It follows that in any run, whenever **assume**$(\mathit{KeyAB}(k,a,b))$ is reached, we have that $k = \mathit{Fresh}(a)$ for some new name a. Therefore, (KeyAB Injective) is indeed a contextual theorem.

It remains to note that the second formula displayed above, follows at once from the definitions of the *Requested* and *Responded* predicates, shown below, since not every byte array has the form of either a request or a response.

Definitions:

(Requested)
$\forall m,s.\ \mathit{Requested}(m,s) \Leftrightarrow$
  $m = \mathit{Concat}(\mathit{Utf8}(\mathit{Literal}(\texttt{"Request"})),\mathit{Utf8}(s))$
(Responded)
$\forall m,s,t.\ \mathit{Responded}(m,s,t) \Leftrightarrow$
  $m = \mathit{Concat}(\mathit{Utf8}(\mathit{Literal}(\texttt{"Response"})),\mathit{Concat}(\mathit{Utf8}(s),\mathit{Utf8}(t)))$

This completes the proof that **RPC** is a refined module. □

B. The Library Principals

We provide additional details on **Principals**, our library for managing keys (and their compromise) by principals; we refer to Section 5.1 for an overview of the library.

The interface uses a single predicate for compromise:

- *Bad*(a) records that principal a has been corrupted, and hence that all the keys it could access are potentially compromised. (This fact is dynamically assumed by each of the key-leaking functions formally included in the attacker interface.)

B.1 Public and Private Key Pairs

We begin with asymmetric keys used for (potentially both) signing and encryption. Our model keeps track of the principal a that owns the private key.

- *PublicKeyPair*(u,a,pk,sk) records that (pk, sk) is a public/private key pair for principal a, with intended usage u; it is dynamically assumed by *mkPublicKeyPair*.
- *SendFrom*(u,a,m) records that principal a intends to sign message m for usage u; it is defined by the protocol that uses managed keys.
- *EncryptTo*(u,a,m) records that the message m can be encrypted towards a for usage u; it is defined by the protocol that uses managed keys.

Public Key Programming Interface:

private val *mkPublicKeyPair*: u:usage $\rightarrow a$:prin $\rightarrow$
  $(pk$:key $*$ sk:key$)\{\mathit{PublicKeyPair}(u,a,pk,sk)\}$
val *genPublicKeyPair*: u:usage $\rightarrow a$:prin $\rightarrow$ unit
private val *getPublicKeyPair*: u:usage $\rightarrow a$:prin $\rightarrow$
  $(pk$:key $*$ sk:key$)\{\mathit{PublicKeyPair}(u,a,pk,sk)\}$
private val *getPrivateKey*: u:usage $\rightarrow a$:prin $\rightarrow$
  sk:key$\{\exists pk.\ \mathit{PublicKeyPair}(u,a,pk,sk)\}$
val *getPublicKey*: u:usage $\rightarrow a$:prin $\rightarrow$
  pk:key$\{\exists sk.\ \mathit{PublicKeyPair}(u,a,pk,sk)\}$
val *leakPrivateKey*: u:usage $\rightarrow a$:prin $\rightarrow$
  sk:keypub$\{\mathit{Bad}(a) \wedge (\exists pk.\ \mathit{PublicKeyPair}(u,a,pk,sk))\}$

Public Key Definitions:

(SignSays SendFrom)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{SendFrom}(u,x,m) \Rightarrow \mathit{SignSays}(sk,m)$
(SignSays Bad)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge \mathit{Bad}(x) \Rightarrow \mathit{SignSays}(sk,m)$
(CanAsymEncrypt EncryptTo)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{EncryptTo}(u,x,m) \wedge (\mathit{Bad}(x) \Rightarrow \mathit{Pub}(m)) \Rightarrow \mathit{CanAsymEncrypt}(pk,m)$
(CanAsymEncrypt Bad)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{Bad}(x) \wedge \mathit{Pub}(m) \Rightarrow \mathit{CanAsymEncrypt}(pk,m)$

Public Key Theorems:

(PublicKeyPair PubPrivKeyPair)
$\forall u,x,pk,sk.\ \mathit{PublicKeyPair}(u,x,pk,sk) \Rightarrow \mathit{Crypto.PubPrivKeyPair}(pk,sk)$
(Inv PublicKeyPair SignSays)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{SignSays}(sk,m) \Rightarrow ((\mathit{SendFrom}(u,x,m)) \vee \mathit{Bad}(x))$
(Inv PublicKeyPair CanAsymEncrypt 1)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{CanAsymEncrypt}(pk,m) \Rightarrow ((\mathit{EncryptTo}(u,x,m)) \vee \mathit{Bad}(x))$
(Inv PublicKeyPair CanAsymEncrypt 2)
$\forall u,x,pk,sk,m.\ \mathit{PublicKeyPair}(u,x,pk,sk) \wedge$
$\mathit{CanAsymEncrypt}(pk,m) \wedge \mathit{Bad}(x) \Rightarrow \mathit{Pub}(m)$
(PrivKey Secrecy)
$\forall u,a,sk.\ \mathit{PrivateKey}(u,a,sk) \wedge \mathit{Pub}(sk) \Rightarrow$
$(\mathit{Bad}(a) \vee ((\forall v.\ \mathit{SendFrom}(u,a,v)) \wedge (\forall v.\ \mathit{EncryptTo}(u,a,v) \Rightarrow \mathit{Pub}(v))))$

B.2 MAC Keys

Managed keys are shared between pairs of principal, a "sender" a (that creates MACs) and a "receiver" b (that verifies MACs); see also Section 5.1.

- $\mathit{MACKey}(u,a,b,k)$ records that k is a MAC key generated for protecting messages from a to b with intent u; it is dynamically assumed by *mkMACKey*.
- $\mathit{Send}(u,a,b,m)$ records that m is a message (potentially) sent from a to b with intent u; this predicate is defined by the protocol.

MAC Key Programming Interface:

$\mathbf{val}\ \mathit{mkMACKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$mk{:}\mathrm{key}\{\mathit{MACKey}(u,a,b,mk)\}$
$\mathbf{val}\ \mathit{genMACKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow \mathrm{unit}$
$\mathbf{private\ val}\ \mathit{getMACKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$mk{:}\mathrm{key}\{\mathit{MACKey}(u,a,b,mk)\}$
$\mathbf{val}\ \mathit{leakMACKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$mk{:}\mathrm{keypub}\{\mathit{Bad}(a) \wedge \mathit{Bad}(b) \wedge \mathit{MACKey}(u,a,b,mk)\}$

MAC Key Definitions

(MACKey MACSays Send)
$\forall u,a,b,mk,m.\ \mathit{MACKey}(u,a,b,mk) \wedge \mathit{Send}(u,a,b,m) \Rightarrow \mathit{MACSays}(mk,m)$
(MACKey MACSays Bad)
$\forall u,a,b,mk,m.$
$\mathit{MACKey}(u,a,b,mk) \wedge (\mathit{Bad}(a) \vee \mathit{Bad}(b)) \Rightarrow \mathit{MACSays}(mk,m)$

MAC Key Theorems

(Inv MACKey MACSays)
$\forall u,a,b,mk,m.\ \mathit{MACKey}(u,a,b,mk) \wedge \mathit{MACSays}(mk,m) \Rightarrow$
$(\mathit{Send}(u,a,b,m) \vee \mathit{Bad}(a) \vee \mathit{Bad}(b))$
(MACKey MKey)
$\forall u,a,b,mk.\ \mathit{MACKey}(u,a,b,mk) \Rightarrow \mathit{Crypto.MKey}(mk)$

B.3 Symmetric Encryption Keys

Similarly, encryption keys are shared between a sender and a receiver.

- $\mathit{EncryptionKey}(u,a,b,k)$ records that k is a key for encrypting messages from a to b with intent u; it is dynamically assumed by *mkEncryptionKey*.
- $\mathit{Send}(u,a,b,m)$ records that m is a message (potentially) encrypted from a to b with intent u; this predicate is defined by the protocol.

Symmetric Encryption Key Programming Interface:

$\mathbf{val}\ \mathit{mkEncryptionKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$ek{:}\mathrm{key}\{\mathit{EncryptionKey}(u,a,b,ek)\}$
$\mathbf{val}\ \mathit{genEncryptionKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow \mathrm{unit}$
$\mathbf{private\ val}\ \mathit{getEncryptionKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$ek{:}\mathrm{key}\{\mathit{EncryptionKey}(u,a,b,ek)\}$
$\mathbf{val}\ \mathit{leakEncryptionKey}: u{:}\mathrm{usage} \rightarrow a{:}\mathrm{prin} \rightarrow b{:}\mathrm{prin} \rightarrow$
$ek{:}\mathrm{keypub}\{\mathit{Bad}(a) \wedge \mathit{Bad}(b) \wedge \mathit{EncryptionKey}(u,a,b,ek)\}$

Encryption Key Definitions:

(EncryptionKey CanSymEncrypt Encrypt)
$\forall u,x1,x2,ek,m.\ \mathit{EncryptionKey}(u,x1,x2,ek) \wedge$
$\mathit{Encrypt}(u,x1,x2,m) \wedge ((\mathit{Bad}(x1) \vee \mathit{Bad}(x2)) \Rightarrow \mathit{Pub}(m)) \Rightarrow$
$\mathit{CanSymEncrypt}(ek,m)$
(EncryptionKey CanSymEncrypt Bad)
$\forall u,x1,x2,ek,m.\ \mathit{EncryptionKey}(u,x1,x2,ek) \wedge$
$(\mathit{Bad}(x1) \vee \mathit{Bad}(x2)) \wedge \mathit{Pub}(m) \Rightarrow \mathit{CanSymEncrypt}(ek,m)$

Encryption Key Theorems:

(EncryptionKey SKey)
$\forall u,x1,x2,ek.\ \mathit{EncryptionKey}(u,x1,x2,ek) \Rightarrow \mathit{Crypto.SKey}(ek)$
(Inv EncryptionKey CanSymEncrypt)
$\forall u,x1,x2,ek,m.\ \mathit{EncryptionKey}(u,x1,x2,ek) \wedge \mathit{CanSymEncrypt}(ek,m) \Leftrightarrow$
$((\mathit{Encrypt}(u,x1,x2,m) \vee m = \mathit{encryptionSecret} \vee \mathit{Bad}(x1) \vee \mathit{Bad}(x2))$
$\wedge$
$(\mathit{Bad}(x1) \vee \mathit{Bad}(x2)) \Rightarrow \mathit{Pub}(m))$
(EncryptionKey Secrecy)
$\forall u,x1,x2,ek.\ \mathit{EncryptionKey}(u,x1,x2,ek) \wedge \mathit{Pub}(ek) \Rightarrow (\mathit{Bad}(x1) \vee \mathit{Bad}(x2)$
$)$

C. Refined Concurrent FPC (RCF)

We recall the subset of RCF from Bengtson et al. (2008) obtained by omitting the syntax and rules for public and tainted kinds. We introduce a notion of expression *safety*, and the development culminates in a safety-by-typing result, Theorem 6, that well-typed expressions are safe. We do not directly use this notion of safety in the main body of the paper, but instead use a closely connected notion of syntactic safety. Theorem 2 in the main body of the paper is essentially Theorem 6, but reformulated in terms of syntactic safety rather than safety.

C.1 Authorization Logics

The calculus relies on logical formulas to specify correctness properties. These formulas are drawn from any choice of authorization logic, a logic satisfying the properties below. (In our initial implementation, the authorization logic is simply first-order logic with equality.)

An *authorization logic* is given as a set of *formulas* defined by a grammar that includes the one given below and a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas that meets the properties listed below. (The set of values, ranged over by M, is defined in Section C.2.)

Minimal Syntax of Formulas:

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

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

Properties of Deducibility: $S \vdash C$

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

(Axiom) $$\frac{}{C \vdash C}$$

(Mon) $$\frac{S \vdash C}{S, C' \vdash C}$$

(Subst) $$\frac{S \vdash C}{S\sigma \vdash C\sigma}$$

(Cut) $$\frac{S \vdash C \quad S, C \vdash C'}{S \vdash C'}$$

(And Intro) $$\frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$$

(And Elim) $$\frac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$$

(Or Intro) $$\frac{S \vdash C_i}{S \vdash C_0 \vee C_1} \quad i = 0,1$$

(Eq) $$\frac{}{\varnothing \vdash M = M}$$

(Ineq) $$\frac{M \neq N \quad fv(M,N) = \varnothing}{\varnothing \vdash M \neq N}$$

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

(Exists Intro) $$\frac{S \vdash C\{M/x\}}{S \vdash \exists x.C}$$

(Exists Elim) $$\frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin fv(S, C')}{S \vdash C'}$$

FOL/F, which is first-order logic with the axiom schemas displayed below, is an example of an authorization logic. (The intended model consists of the phrases of syntax of RCF identified up to consistent renaming of bound names and variables. A *syntactic* function symbol is one used to represent the phrases of RCF as a term, using the locally nameless representation of de Bruijn. RCF variables are identified with the variables of the logic, while each RCF name is a constant, that is, a nullary syntactic function symbol.)

Additional Rules for FOL/F:

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

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

C.2 Expressions, Evaluation, and Safety

Syntax of Values and Expressions:

a, b, c	name
x, y, z	variable
$h ::=$	value constructor
$\quad inl$	$\quad$ left constructor of sum type
$\quad inr$	$\quad$ right constructor of sum type
$\quad fold$	$\quad$ constructor of recursive type
$M, N ::=$	value
$\quad x$	$\quad$ variable
$\quad ()$	$\quad$ unit
$\quad \mathbf{fun}\, x \to A$	$\quad$ function (scope of x is A)
$\quad (M, N)$	$\quad$ pair
$\quad h\,M$	$\quad$ construction
$A, B ::=$	expression
$\quad M$	$\quad$ value
$\quad M\,N$	$\quad$ application
$\quad M = N$	$\quad$ syntactic equality
$\quad \mathbf{let}\ x = A\ \mathbf{in}\ B$	$\quad$ let (scope of x is B)
$\quad \mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$	$\quad$ pair split (scope of x, y is A)
$\quad \mathbf{match}\ M\ \mathbf{with}$	$\quad$ constructor match
$\qquad h\,x \to A\ \mathbf{else}\ B$	$\qquad$ (scope of x is A)
$\quad (\nu a)A$	$\quad$ restriction (scope of a is A)
$\quad A \Rsh B$	$\quad$ fork
$\quad a!M$	$\quad$ transmission of M on channel a
$\quad a?$	$\quad$ receive message off channel
$\quad \mathbf{assume}\ C$	$\quad$ assumption of formula C
$\quad \mathbf{assert}\ C$	$\quad$ assertion of formula C

$\mathbf{true} \triangleq inl\ () \qquad \mathbf{false} \triangleq inr\ ()$

The formal syntax of expressions is in an intermediate, reduced form (reminiscent of A-normal form (Sabry and Felleisen 1993)) where $\mathbf{let}\ x = A\ \mathbf{in}\ B$ is the only construct to allow sequential evaluation of expressions. As usual, $A;B$ is short for $\mathbf{let}\ \_ = A\ \mathbf{in}\ B$. (The notation $\_$ denotes an anonymous variable that by convention occurs nowhere else.) More notably, if A and B are proper expressions rather than being values, the application $A\ B$ is short for $\mathbf{let}\ f = A\ \mathbf{in}\ (\mathbf{let}\ x = B\ \mathbf{in}\ f\ x)$.

Examples: Communication and Concurrency

$(T)chan \triangleq (T \to \text{unit}) \times (\text{unit} \to T)$

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

Structures and Static Safety:

$e ::= M \mid M\ N \mid M = N \mid \mathbf{let}\ (x,y) = M\ \mathbf{in}\ A \mid$
$\qquad \mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B \mid a? \mid \mathbf{assert}\ C$
$\prod_{i \in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \ldots \Rsh A_n$
$\mathscr{L} ::= \{\} \mid (\mathbf{let}\ x = \mathscr{L}\ \mathbf{in}\ B)$

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

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

Structures formalize the idea that a state has three parts: (1) the *log*, a multiset $\prod_{i \in 1..m} \mathbf{assume}\ C_i$ of assumed formulas; (2) a series of messages M_j sent on channels but not yet received; and (3) a series of elementary expressions e_k being evaluated in parallel contexts.

Heating: $A \Rrightarrow A'$

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

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ $\quad$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)
$A \Rrightarrow A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \Rrightarrow \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Heat Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)

$() \Rsh A \equiv A$ (Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$ (Heat Msg ())
$\textbf{assume } C \Rrightarrow \textbf{assume } C \Rsh ()$ (Heat Assume ())

$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$ (Heat Res Fork 1)
$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$ (Heat Res Fork 2)
$a \notin fn(B) \Rightarrow$ (Heat Res Let)
$\quad \textbf{let } x = (\nu a)A \textbf{ in } B \Rrightarrow (\nu a)\textbf{let } x = A \textbf{ in } B$

$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$ (Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$ (Heat Fork Comm)
$\textbf{let } x = (A \Rsh A') \textbf{ in } B \equiv$ (Heat Fork Let)
$\quad A \Rsh (\textbf{let } x = A' \textbf{ in } B)$

LEMMA 16 (Structure). *For every expression A, there is a structure* **S** *such that* $A \Rrightarrow \mathbf{S}$.

Reduction: $A \rightarrow A'$

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

$A \rightarrow A' \Rightarrow \textbf{let } x = A \textbf{ in } B \rightarrow \textbf{let } x = A' \textbf{ in } B$ (Red Let)
$A \rightarrow A' \Rightarrow (\nu a)A \rightarrow (\nu a)A'$ (Red Res)
$A \rightarrow A' \Rightarrow (A \Rsh B) \rightarrow (A' \Rsh B)$ (Red Fork 1)
$A \rightarrow A' \Rightarrow (B \Rsh A) \rightarrow (B \Rsh A')$ (Red Fork 2)

$A \rightarrow A' \quad \text{if } A \Rrightarrow B, B \rightarrow B', B' \Rrightarrow A'$ (Red Heat)

A closed expression A is *safe* if and only if, in all evaluations of A, all assertions succeed.

Expression Safety:

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

C.3 A Type System for Safety

Syntax of Types:

$H, T, U, V ::=$	type
unit	unit type
$x : T \rightarrow U$	dependent function type (scope of x is U)
$x : T * U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\textbf{rec } \alpha.T$	iso-recursive type (scope of α is T)
α	iso-recursive type variable
$x : T\{C\}$	refinement type (scope of x is C)

Some Derivable Types:

$\{C\} \triangleq \_ : \text{unit}\{C\}$ (ok-type)
$bool \triangleq \text{unit} + \text{unit}$
$\text{int} \triangleq \textbf{rec } \alpha.\text{unit} + \alpha$
$(T)list \triangleq \textbf{rec } \alpha.\text{unit} + (T \times \alpha)$
$T \rightarrow U \triangleq \_ : T \rightarrow U$
$[x_1 : T_1]\{C_1\} \rightarrow U \triangleq x_1 : x_1 : T_1\{C_1\} \rightarrow U$
$(x_1 : T_1 * \cdots * x_n : T_n)\{C\} \triangleq$
$$\begin{cases} x_1 : T_1 * \ldots x_{n-1} : T_{n-1} * x_n : T_n\{C\} & \text{if } n > 0 \\ \{C\} & \text{otherwise} \end{cases}$$

Syntax of Typing Environments:

$\mu ::=$	environment entry
α	type variable
$\alpha <: \alpha'$	subtype ($\alpha \neq \alpha'$)
$a \updownarrow T$	name of a typed channel
$x : T$	variable
$E ::= \mu_1, \ldots, \mu_n$	environment

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

The type system consists of five inductively defined judgments.

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

Rules of Well-Formedness and Deduction:

(Env Empty)
$$\frac{}{\varnothing \vdash \diamond}$$

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

(Type)
$$\frac{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)}{E \vdash T}$$

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

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

General Rules:

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

(Val Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Exp Subsum)
$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

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

(Exp Assume)
$$\frac{E \vdash \diamond \quad fnfv(C) \subseteq dom(E)}{E \vdash \textbf{assume } C : \_ : \text{unit}\{C\}}$$

(Exp Assert)
$$\frac{E \vdash C}{E \vdash \textbf{assert } C : \text{unit}}$$

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

Rules for Unit Type:

(Sub Unit) (Val Unit)

$$\frac{E \vdash \diamond}{E \vdash \text{unit} <: \text{unit}} \qquad \frac{E \vdash \diamond}{E \vdash () : \text{unit}}$$

Rules for Function Types:

(Sub Fun)

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

(Val Fun)

$$\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun}\, x \to A : (x : T \to U)}$$

(Exp Appl)

$$\frac{E \vdash M : (x : T \to U) \quad E \vdash N : T}{E \vdash M\, N : U\{N/x\}}$$

Rules for Pair Types:

(Sub Pair)

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

(Val Pair)

$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (x : T * U)}$$

(Exp Split)

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

Rules for Sums and Recursive Types:

(Sub Sum) (Sub Var)

$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')} \qquad \frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(Sub Rec)

$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin fnfv(T') \quad \alpha' \notin fnfv(T)}{E \vdash (\mathbf{rec}\ \alpha.T) <: (\mathbf{rec}\ \alpha'.T')}$$

$inl{:}(T, T+U) \quad inr{:}(U, T+U) \quad fold{:}(T\{\mathbf{rec}\ \alpha.T/\alpha\}, \mathbf{rec}\ \alpha.T)$

(Val Inl Inr Fold)

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\, M : U}$$

(Exp Match Inl Inr Fold)

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

Rules for Refinement Types:

(Sub Refine Left) (Sub Refine Right)

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

(Val Refine)

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : x : T\{C\}}$$

Rules for Concurrency:

(Exp Res)

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

(Exp Send) (Exp Recv)

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

(Exp Fork)

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

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

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

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

Let E be *executable* if and only if $recvar(E) = \varnothing$.

LEMMA 17 (Static Safety). *If $\varnothing \vdash \mathbf{S} : T$ then $\mathbf{S}$ is statically safe.*

PROPOSITION 18 ($\Rrightarrow$ Preserves Types). *If E is executable and $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.*

PROPOSITION 19 ($\to$ Preserves Types). *If E is executable, $fv(A) = \varnothing$, and $E \vdash A : T$ and $A \to A'$ then $E \vdash A' : T$.*

THEOREM 6 *If $\varnothing \vdash A : T$ then A is safe.*

References

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

M. Backes, C. Hritcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. In *Workshop on Foundations of Computer Security*, 2009.

M. Backes, C. Hritcu, and M. Maffei. Union and intersection types for secure protocol implementations. Unpublished draft, 2010. URL `http://www.infsec.cs.uni-saarland.de/~hritcu/publications/rcf-and-or-coq-submitted.pdf`.

M. Barbosa, R. Noad, D. Page, and N. P. Smart. First steps toward a cryptography-aware language compiler. Available at `http://eprint.iacr.org/2005/160.pdf`, 2005.

M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4), 2008.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR–TR–2008–118, Microsoft Research, 2008. A preliminary, abridged version appears in the proceedings of CSF'08.

K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *TCS*, 340(1):102–153, 2005.

K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-Security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *LNCS*, pages 88–106. Springer, 2006.

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

K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135. ACM Press, 2008b.

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

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140. IEEE Computer Society, 2009.

K. Bhargavan, C. Fournet, and N. Guts. Pre- and post-conditions for security typechecking. Draft available from the authors, 2010.

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

B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society, 2006.

S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and protocols for the oasis security assertion markup language (saml) v2.0, 2005.

I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, 206 (2-4):402–424, 2008.

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

E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 144–158, 2000.

B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. CUP, 1990.

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

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

D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon. *XML Encryption Syntax and Processing*, 2002a. W3C Recommendation, at `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/`.

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

C. Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.

D. Gollmann. Authentication by correspondence. *IEEE Journal on Selected Areas in Communication*, 21(1):88–95, 2003.

A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR–TR–2009–147, Microsoft Research, 2009. To appear in the proceedings of the 2009 Marktoberdorf Summer School.

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

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

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

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

L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 27–38, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: http://doi.acm.org/10.1145/1411204.1411212.

D. Kikuchi and N. Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *APLAS*, volume 4807 of *LNCS*, pages 191–205. Springer, 2007.

E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. In *Mathematical Foundations of Programming Semantics (MFPS XXI)*, 2005.

K. W. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, volume 4421 of *LNCS*, pages 505–519. Springer, 2007.

J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, page 41. ACM, 2007.

G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

S. Lukell, C. Veldman, and A. C. M. Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunication Networks and Applications Conference (SATNAC)*, 2003.

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

A. Mukhamedov, A. D. Gordon, and M. Ryan. Towards a verified reference implementation of a Trusted Platform Module. In *Seventeenth International Workshop on Security Protocols*, LNCS. Springer, 2009. To appear.

F. Muller and J. Millen. Cryptographic protocol generation from CAPSL. Technical Report SRI–CSL–01–07, SRI, 2001.

A. Nanda. *A Technical Reference for the Information Card Profile V1.0*. Microsoft Corporation, December 2006. At `http://go.microsoft.com/fwlink/?LinkId=87444`.

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

N. O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *FCS-ARSPA-WITS'08*, pages 211–226, 2008.

D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, Jan. 1987.

L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

L. C. Paulson. Logic and proof. University of Cambridge lecture notes, 2008.

A. Perrig, D. Song, and D. Phan. AGVI – automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, LNCS, pages 241–245. Springer, 2001.

L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, 2006.

A. Pironti. *Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications*. PhD thesis, University of Turin, 2010.

G. D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes, CSLI, Stanford University, July 1985.

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

D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, volume 1, pages 400–405, 2004.

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

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

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

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383, 2008.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP*, pages 529–549, 2010.

T. Terauchi. Dependent types from counterexamples. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 119–130, 2010.

O. Udrea, C. Lumezanu, and J. S. Foster. Rule-based static analysis of network protocol implementations. *Inf. Comput.*, 206(2-4):130–157, 2008.

H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 277–288, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-568-0. doi: http://doi.acm.org/10.1145/1599410.1599445.

H. Xi. Dependent ml an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.