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.

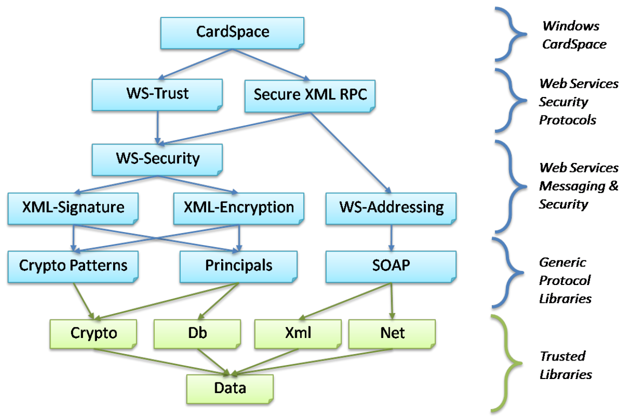
Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques.

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. 2006b), 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. 2006a), CardSpace (Bhargavan et al. 2008b), and TLS (Bhargavan et al. 2008a).

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 pro-

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01...$10.00

cess 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.

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 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 refinements types, public and tainted kinds (Gordon and Jeffrey 2003b), and seals (Morris 1973). 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.

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.

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.

(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.

Contents Section 2 reviews F7. Section 3 illustrates our method on an RPC protocol. Section 4 provides a theory to justify proofs of security by typechecking. Section 5 gives examples of crypographic libraries. Section 6 outlines more substantial case studies. Section 7 evaluates the performance of our implementation. Section 8 discusses related work.

A website `http://research.microsoft.com/f7` hosts a technical report with details, proofs, and examples omitted from this version of the paper, as well as our typechecker with libraries and sample code for all examples.

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.

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:

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

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.

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.)

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)
$A \upharpoonleft B$	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.

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
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 the figure on the first page. The library module **Data** defines standard datatypes such as strings, bytestrings, 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 $\mu_1,\dots,\mu_n$, 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 $x : T$. 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 the companion technical report.

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.

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.

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. $a \rightarrow b :$ *utf8 s* | (*hmacsha1* k_{ab} (*request s*))
2. $b \rightarrow a :$ *utf8 t* | (*hmacsha1* k_{ab} (*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.

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:

- $Request(a,b,s)$ before *a* sends message 1;
- $Response(a,b,s,t)$ before *b* sends message 2;
- $KeyAB(k,a,b)$ before issuing a key *k* associated with *a* and *b*;
- $Bad(a)$ 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:

- $RecvRequest(a,b,s)$ after *b* accepts message 1;
- $RecvResponse(a,b,s,t)$ after *a* accepts message 2;

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

$\forall a,b,s.\ RecvRequest(a,b,s) \Leftrightarrow (Request(a,b,s) \lor Bad(a) \lor Bad(b))$
$\forall a,b,s,t.\ RecvResponse(a,b,s,t) \Leftrightarrow$
$\quad (Request(a,b,s) \land Response(a,b,s,t)) \lor Bad(a) \lor Bad(b)$

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.

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
```

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. 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):

```
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)
```

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. 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.)

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

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 the companion technical report, 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):

```
private val hmac_keygen: unit → k:key{MKey(k)}
val hmacsha1:
  k:key →
  b:bytes{ (MKey(k) ∧ MACSays(k,b)) ∨ (Pub(k) ∧ Pub(b)) } →
  h:bytes{ IsMAC(h,k,b) ∧ (Pub(b) ⇒ Pub(h)) }
val hmacsha1Verify:
  k:key{MKey(k) ∨ Pub(k)} → b:bytes → h:bytes → unit{IsMAC(h,k,b)}
```

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

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.
- *IsMAC*(*h*,*k*,*b*) holds when verification that *h* is a MAC for *b* under *k* succeeds; it implies either $MACSays(k,b)$ or *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 $Pub(b) \Rightarrow Pub(h)$ 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 $Pub(k) \Rightarrow MACSays(k,b)$, for example because *k* is not public, to deduce that *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 $Bad(a)$ in our protocol).

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)
$\forall v,v',s,s',t'.\ (Requested(v,s) \wedge Responded(v',s',t')) \Rightarrow (v \neq v')$
(request is injective)
$\forall v,v',s,s'.\ (Requested(v,s) \wedge Requested(v',s') \wedge v = v') \Rightarrow (s = s')$
(response is injective)
$\forall v,v',s,s',t,t'.$
$(Responded(v,s,t) \wedge Responded(v',s',t') \wedge v = v') \Rightarrow (s = s' \wedge t = t')$

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

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

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

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 $KeyAB(k,a,b)$ 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. 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.

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 $\rightarrow m$:bytespub$\{Requested(m,s)\}$
val *response*: s:payload $\rightarrow t$:payload $\rightarrow m$:bytespub$\{Responded(m,s,t)\}$
val *service*: payload $\rightarrow$ payload
type $(;a$:str,b:str)keyab = k:key $\{\ MKey(k) \wedge KeyAB(k,a,b)\ \}$
val *mkKeyAB*: a:str $\rightarrow b$:str $\rightarrow k$: $(;a,b)$keyab
val *client*: a:str $\rightarrow b$:str $\rightarrow k$: $(;a,b)$keyab $\rightarrow$ payload $\rightarrow$ unit
val *server*: a:str $\rightarrow b$:str $\rightarrow k$: $(;a,b)$keyab $\rightarrow$ 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

  $(Requested(v,s) \wedge \exists s.\ (Requested(v,s) \wedge Request(a,b,s))) \vee$
  $(Requested(v,s) \wedge \exists s,t.\ (Responded(v,s,t) \wedge Response(a,b,s,t))) \vee$
  $Bad(a) \vee Bad(b)$

  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.

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). The semantics of expressions is defined by a small-step reduction relation, written $A \rightarrow A'$, 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 $C = C_1 \wedge \cdots \wedge C_n$.
- 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.

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 $\forall x_1, \dots, x_k.(C_1 \wedge \cdots \wedge C_n \Rightarrow C)$ where $C_1, \dots, C_n$ range over atomic formulas and equations and C ranges over atomic formulas.
- A *logic program*, P, is a finite conjunction of Horn clauses.
- If P is a logic program, let $\mathscr{I}_P$ be the least RCF-interpretation to satisfy P (which exists uniquely, by Tarski-Knaster).

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** *$C \upharpoonright A$ is syntactically safe; and (2) C is a theorem of A. Then A is semantically safe.*

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 $bv(X) = \{x_1, \dots, x_n\}$. 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 $\mu_1, \dots, \mu_n$ 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 $E \vdash X[(x_1, \dots, x_n)] : (x_1 : T_1 * \ldots * x_n : T_n)$ and $(x_i : T_i)_{i=1..n} <: I$.

Refined Modules We use an expression context **assume** $P \upharpoonright Y$ 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** $P \upharpoonright Y$ 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** $P \upharpoonright Y$ if and only if C is a theorem of **assume** $P \upharpoonright Z[Y[A]]$ whenever Z and A are factual and independent of **assume** $P \upharpoonright Y$.

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

LEMMA 1 (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** *$P \upharpoonright Y$.*

- 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** $\mathbf{M}^{def} \upharpoonright Y$;
  (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 $\{\mathbf{M}^{def}\} = \_ : \text{unit}\{\mathbf{M}^{def}\}$, 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 2 $\mathbf{Lib} = (\varnothing, \textbf{\textit{assume}}\ \mathbf{Lib}^{def} \upharpoonright \mathit{Lib}, I_L^7)$ *is a refined module.*

As another example, our RPC protocol consists of a refined module of the form: $\mathbf{RPC} = (I_L^7, \textbf{assume}\ \mathbf{RPC}^{def} \upharpoonright \mathit{RPC}, (I_L, I_R))$. 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 3 **RPC** *is a refined module.*

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 $\mathbf{M}_1 = (E_1, \textbf{assume}\ \mathbf{M}_1^{def} \upharpoonright Y_1, I_1)$ and $\mathbf{M}_2 = (E_2, \textbf{assume}\ \mathbf{M}_2^{def} \upharpoonright Y_2, I_2)$ their *composition* $\mathbf{M}_1;\mathbf{M}_2$ is the triple $(E_1, \textbf{assume}\ (\mathbf{M}_1^{def} \wedge \mathbf{M}_2^{def}) \upharpoonright Y_1[Y_2], I_2)$.

LEMMA 4 (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.*

For example, the triple $\mathbf{Lib};\mathbf{RPC}$ is: $(\varnothing, \textbf{assume}\ (\mathbf{Lib}^{def} \wedge \mathbf{RPC}^{def}) \upharpoonright L[Y], (I_L, I_R))$. By Lemma 4 (Composition), $\mathbf{Lib};\mathbf{RPC}$ is a refined module.

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.

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

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

We can now prove Theorem 1. We have that $\mathbf{Lib};\mathbf{RPC} = (\varnothing, X, (I_L, I_R))$ where $X = \textbf{assume}\ (\mathbf{Lib}^{def} \wedge \mathbf{RPC}^{def}) \upharpoonright \mathit{Lib}[\mathit{RPC}]$ 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 $I_L, I_R \vdash O$: 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. We refer to the companion paper for a more complete description, and for a detailed presentation of the **Crypto** module. (Section 3 also presents its interface for MACs.) Both modules are fully verified, and demonstrate the flexibility of our approach. Relying on these libraries, their logical definitions, and their theorems, we build (and verify) a series of modular protocols, leading to Windows CardSpace.

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 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 infrastucture, 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:

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

The function *mkMACKey* generates and returns a fresh MAC; *genMACKey* calls *mkMACKey* then stores the key in the database; *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 (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 asymmetric 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 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 5 **LibX**;**Principals** *is a refined module.*

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

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

private val *aes_keygen*: unit $\rightarrow k$:key$\{SKey(k)\}$
val *aes_encrypt*: `(* AES CBC *)`
  k:key $\rightarrow$
  b:bytes$\{(SKey(k) \wedge CanSymEncrypt(k,b)) \vee (Pub(k) \wedge Pub(b))\} \rightarrow$
  e:bytes$\{IsEncryption(e,k,b)\}$
val *aes_decrypt*: `(* AES CBC *)`
  k:key$\{SKey(k) \vee Pub(k)\} \rightarrow e$:bytes $\rightarrow$
  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*; *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); it returns encrypted bytes e, tracked by *IsEncryption*; *aes_decrypt* takes a key k and bytes e and extracts a plaintext b. Since encryption is unauthenticated, decryption may succeed even if e is not a valid encryption, returning some unspecified (garbage) bytes, so the first postcondition of *aes_decrypt* just says that, if the caller knows that e is an encryption of some (possibly unknown) plaintext p under e, then decryption does returns p. (The second postcondition enables re-encryptions.)

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):

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

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

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

MAC-and-Encrypt (as in SSH):

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

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

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:

$a \rightarrow b$: $rsa\_oaep\ pk_b\ k_{ab} \mid aes\ k_{ab}\ t$

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 $\rightarrow (pk$:key $*$ sk:key$)$
  $\{HyPubKey(pk) \wedge HyPrivKey(sk) \wedge PubPrivKeyPair(pk,sk)\}$
val *hybridEncrypt*: k:key $\rightarrow b$:bytes
  $\{(HyPubKey(k) \wedge CanHyEncrypt(k,b)) \vee (Pub(k) \wedge Pub(b))\ \} \rightarrow$
  e:bytes$\{IsHyEncryption(e,k,b)\}$
val *hybridDecrypt*: sk:key $\rightarrow$
  e:bytes$\{HyPrivKey(sk) \vee (Pub(sk) \wedge Pub(e))\} \rightarrow$
  b:bytes$\{(\forall pk,x.\ (PubPrivKeyPair(pk,sk)$
  $\wedge IsHyEncryption(e,pk,x)) \Rightarrow x = b) \wedge (Pub(sk) \Rightarrow Pub(b))\}$

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)
$\forall pk,kb.\ HyPubKey(pk) \wedge HySymKey(SymKey(kb),pk) \Rightarrow$
  $CanAsymEncrypt(pk,kb)$
(HySymKey CanSymEncrypt)
$\forall pk,k,b.\ HySymKey(k,pk) \wedge CanHyEncrypt(pk,b) \Rightarrow CanSymEncrypt(k,b)$

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.

Derived Keys and Endorsing Signatures The library also provides support for deriving separate keys from a secret seed, and for endorsing signatures (that is, composing MACs and asymmetric signatures).

LEMMA 6 **LibX**;**Patterns** *is a refined module.*

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 academic protocol for establishing a fresh short-term key between two principals a and b.

Otway-Rees Protocol:

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

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.

LEMMA 7 **LibX**;**Patterns**;**Principals**;**OtwayRees** *is a refined module.*

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.

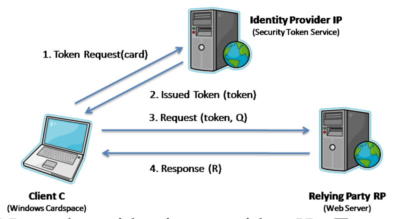
Session Sequence Integrity (initially $i = 1$):

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

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.

LEMMA 8 **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> ...
      <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. 2006a; 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

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)	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	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*

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

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 10 **LibWS**;**SecureRPC** *is a refined module.*

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 $\mathit{Request}_i\ k1\ k2\ [m1;\ldots;mn]$, 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 $\mathit{Response}_i\ [m1;\ldots;mn]$.

CardSpace Protocol (using X.509 Mutual Authentication):

1. $C \rightarrow IP$:	$\mathit{Request}_1\ skC\ pkIP\ [\mathit{TokenRequest}(RP, pkRP)]$
IP:	Issue token $t = \mathit{Token}(id, C, RP, kt)$
2. $IP \rightarrow C$:	$\mathit{Response}_1\ [t; \mathit{XML\text{-}Encrypt}\ pkRP\ t]$
3. $C \rightarrow RP$:	$\mathit{Request}_2\ kt\ pkRP\ [t;m1]$
4. $RP \rightarrow C$:	$\mathit{Response}_2\ [m2]$

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 11 **LibWS**;**SecureRPC**;**CardSpace** *is a refined module.*

7. Performance Evaluation

The table above 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 let 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

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

The RCF system of refinement types is similar to that of recent systems such as 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.

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

References

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

M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. 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. See also CSF'08.

K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-Security protocols. In *WS-FM'06, LNCS 4184*, 2006a.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW'06*, 2006b.

K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM CCS*, 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 *ASIACCS'08*, pages 123–135, 2008b.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01*, pages 82–96, 2001.

B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, 2006.

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 *CSF'09*, 2009.

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

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340. Springer, 2008. LNCS 4963.

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

C. Flanagan. Hybrid type checking. In *ACM POPL'06*, pages 245–256, 2006.

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

A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. 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.

E. Kleiner and A. W. Roscoe. On the relationship between web services security and traditional protocols. In *MFPS XXI*, 2005.

G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, pages 147–166, 1996. LNCS 1055.

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

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

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

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

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

P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *ACM PLDI'08*, pages 159–169, 2008.