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 proves a single, giant, untyped pi pro-
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 $KeyAB(k_{ab},a,b)\land Send(a,b,x)$, 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 authenticity 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 $\lambda$-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 cryptographic 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](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 $\alpha$, value variables $x$, and names $a$. We identify phrases of syntax up to consistent renaming of bound identifiers. We write $\psi(\phi/t)$ for the capture-avoiding substitution of the phrase $\phi$ for each free occurrence of identifier $t$ in the phrase $\psi$. 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 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 $\alpha$, value variables $x$, and names $a$. We identify phrases of syntax up to consistent renaming of bound identifiers. We write $\psi(\phi/t)$ for the capture-avoiding substitution of the phrase $\phi$ for each free occurrence of identifier $t$ in the phrase $\psi$. 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 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 $\alpha$, value variables $x$, and names $a$. We identify phrases of syntax up to consistent renaming of bound identifiers. We write $\psi(\phi/t)$ for the capture-avoiding substitution of the phrase $\phi$ for each free occurrence of identifier $t$ in the phrase $\psi$. We say a phrase is closed to mean that it has no free type or value variables (although it may contain free names).
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. \( a \to b: utf8 s | (hmacsha1 k_{ab}(\text{request } s)) \)
2. \( b \to a: utf8 t | (hmacsha1 k_{ab}(\text{response } s t)) \)

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 many 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:

\[ \forall a, b, s. \text{RecvRequest}(a, b, s) \Rightarrow (\text{Request}(a, b, s) \lor \text{Bad}(a) \lor \text{Bad}(b)) \]

\[ \forall a, b, s, t. \text{RecvResponse}(a, b, s t) \Rightarrow (\text{Request}(a, b, s) \land \text{Response}(a, b, s t)) \land \text{Bad}(a) \lor \text{Bad}(b) \]

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:

```fsharp
let mkKeyAB a b = let k = hmacsha1 (keygen()) in assume(KeyAB(k,a,b)); k
let request s = concat(utf8 s) \( \Rightarrow \) assume(ReqRequest(k,a,b)); k
let response s t = concat(utf8 s) \( \Rightarrow \) assume(RespResponse(a,b,s t)); k
```

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, \( \text{recv} \) extracts s from its encoded payload by calling utf8, and then verifies that the received MAC matches the MAC recomputed from k and s. The code uses \( \text{concat} \) and \( \text{iconcat} \) to concatenate and split byte arrays. (Crucially for this protocol, \( \text{concat} \) embeds the length of the first array, and \( \text{iconcat} \) splits arrays at this length. Otherwise, for instance, \( \text{response} \) 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 a request:

- 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 {BgAy1CsgM9}9sh2a71DAw3Rrk...
  - (28 bytes)
  - Listening at ::1:8080
  - Received Request 2 + 27
  - Sending {AQA0Ncc}cUL/WoYa5OGGtOtPm...
  - (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

```fsharp
let setup (ars tr) (bs tr) =
  let k = mkMac k a b in
  (fun → client a b k),
  (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 server. Our opponent interfaces declare functions that operate on types of the form \( x : T \{ \text{Pub}(s) \} \): 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 \( x : \text{bytes} \{ \text{Pub}(s) \} \). The types strpub and keypub of public strings and public keys are defined similarly.

In our method, we explicitly give an inductive definition of \( \text{Pub} \), 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):

```fsharp
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 \( I_O \) be the opponent interface for our library. Let \( I_K \) 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 \( I_O, I_K \vdash O : \text{unit} \), 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):**

```fsharp
private val hmac_keygen : unit → k: keypub \{ MKey(k) \}
val hmacsha1 :
  k: key →
  b: bytes →
  h: bytes →
  \{ IsMAC(h, k) \} →
  \{ Pub(h) \} →
  \{ Pub(h) \} →
  \{ IsMAC(h, k) \}
```

(C1. By expanding the definition of IsMAC)

\( \forall h, k. \text{IsMAC}(h, k) \land \text{Bytes}(h) \Rightarrow \text{IsMAC}$\text{says}(h, k) \lor \text{Pub}(h) \)

(C2. MAC keys are public iff they may be used with any logical payload)

\( \forall k. \text{MKey}(k) \Rightarrow \text{Pub}(k) \equiv \forall m. \text{MACSays}(k, m) \)

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:

- \( \text{MKey}(k) \) records that \( k \) has been produced by \( \text{lmac_keygen} \); the adversary can produce other public keys from public values.
- \( \text{MACSays}(k, b) \) is defined by the protocol that relies on \( k \), as its precondition for computing a MAC and its postcondition after verifying a MAC.
- \( \text{IsMAC}(h, k) \) holds when verification that \( h \) is a MAC for \( b \) under \( k \) succeeds; it implies either \( \text{MACSays}(k, b) \) or \( \text{Pub}(k) \).

The precondition of \( \text{lmacsha1} \) 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 \( \text{lmacsha1} \) 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 \( \text{Pub}(b) \Rightarrow \text{Pub}(h) \) 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 \( \text{lmacsha1Verify} \) similarly covers the two cases for the key. A call \( \text{lmacsha1Verify} \) \( k \) \( b \) \( h \) 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 \( \text{Pub}(k) \Rightarrow \text{MACSays}(k, b) \), for example because \( k \) is not public, to deduce that \( \text{MACSays}(k, b) \).

The library also assumes definitions and theorems relating these predicates, and in particular the inductive definition of \( \text{Pub} \). For convenience, the display above includes two properties for MACs that are corollaries of these definitions: C1 just instantiates the definition of \( \text{IsMAC} \); 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 \( \text{MACSays} \), this may be established by assuming some compromise at the protocol level (predicate \( \text{Bad}(a) \) 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 definitions. 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)
\[ \forall v', v'' : (\text{Requested}(v) \land \text{Responded}(v', v'')) \implies (v \neq v') \]

(request is injective)
\[ \forall v', s, s' : (\text{Request}(v, s) \land \text{Requested}(v', s')) \land v = v' \implies (s = s') \]

(response is injective)
\[ \forall v', v'' : (\text{Response}(v, v') \land \text{Responded}(v', v'')) \land v = v' \implies (v' = v'') \]

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)
\[ \forall a, b, k, m : \text{KeyAB}(k, a, b) \implies (\text{MACSays}(k, m) \equiv (\exists s, \text{Requested}(s) \land \text{Response}(a, s, b)) \lor (\text{Bad}(a) \lor \text{Bad}(b))) \]

(KeyAB Injekte)
\[ \forall a, b, a', b' : \text{KeyAB}(k, a, b) \land \text{KeyAB}(k, a', b') \implies (a = a') \land (b = b') \]

(KeyAB Pub Bad)
\[ \forall a, b, k : \text{KeyAB}(k, a, b) \land \text{Pub}(k) \implies \text{Bad}(a) \lor \text{Bad}(b) \]

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, 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 Injekte) 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_keygen 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 Injekte) holds. Conversely, we would not be able to prove the theorem if mkKeyAB also (erroneously) assumed KeyAB(k,a,b), 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 M(key(k) when KeyAB(k,a,b) is assumed, hence we establish the lemma (\forall a, b, k : KeyAB(k,a,b) \implies M(key(k). By corollary C1, 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 -> m:bytespub([Requested(m,s)])
val response : s:payload -> r:payload -> m:bytespub([Responded(m,s,r)])
val service : payload -> payload

type (as:str, b:str)keyab = k: key { MKey(k) \land KeyAB(a,b,k) }
val mkKeyAB : a:as -> b:bstr -> k:(a,b)keyab -> payload -> unit
val server : as:asstr -> b:bstr -> k:(a,b)keyab -> payload -> 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 C3, 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)\lor Bad(a)\lor Bad(b) in a context where we have KeyAB(k,a,b), Requested(s,v), and IsMAC(h,k,v). By corollary C2, we have MKey(k)\lor Pub(k)\implies \forall v. MACSays(k,v), so we obtain MACSays(k,v) in both cases of the disjunction. By definition of (KeyAB MACSays), this yields

\[ (\text{Requested}(v, s) \lor \exists s', (\text{Request}(v, s') \land \text{Response}(a, s', b))) \lor (\text{Requested}(v, s) \land \exists s', (\text{Response}(a, s', v) \lor \text{Response}(a, s', v))) \lor \text{Bad}(a) \lor \text{Bad}(b) \]

which implies Request(a,b,s)\lor Bad(a)\lor 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 \Rightarrow A'. We represent all reachable run-time program states using expressions in special forms, named structures, ranged over by 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 \{assume_C_1, ..., assume_C_n\} with C = C_1 \land ... \land C_n.
- A C-structure is syntactically statically safe if every RCF-interpretation to satisfy C also satisfies each active assumption.
- An expression A is syntactically safe if and only if, for all expressions A' and structures S, if A \rightarrow^* A' and A' \models S, then S is syntactically statically safe.

**Theorem 2 (Bengtson et al. 2008)**

If \emptyset \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, ..., x_k. (C_1 \land ... \land 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.
- If P is a logic program, let \mathcal{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 S, if A \rightarrow^* A' and A' \models S, then 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 if 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, \mathcal{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 \vdash 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, ..., x_n\}. We treat the concrete syntax for composing F# modules as syntactic sugar, writing, for example, T X_1 X_2 for the module X_1 X_2.

- An interface I is a typing environment \mu_1, ..., \mu_n where each \mu_i is either an abstract type \alpha_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 if I \vdash \phi then always implies I \vdash \phi' and I \vdash \phi'').

1. A module X implements I in E, written E \vdash X \odot ε I, when E \vdash X[x_1, ..., x_n] : (x_1 : T_1 * ... * x_n : T_n) and (x_i : T_i)_=1..n \vdash \phi.

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

When the following lemma applies, we can prove contextual theorems from the inductive definitions P of assume P \vdash 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 \land Q also satisfies C. If Y is an expression context independent from P, then C is a contextual theorem of assume P \vdash Y.

1. Let a refined module be a triple M = (E, X, I) such that there are closed formulas \mathcal{M}^{def} and \mathcal{M}^{thm}, and a module Y where:
   1. X is factual and X = assume \mathcal{M}^{def} \vdash Y;
   2. E \cup \mathcal{M}^{def}, \mathcal{M}^{thm} \vdash Y \odot ε I;

2. \mathcal{M}^{thm} is a contextual theorem of X.

When we write a formula such as \mathcal{M}^{def} as an environment entry, we mean it as a shorthand for \:: \{ \mathcal{M}^{def} \} \land \mathcal{M}^{thm} \land \text{unit} \{ \mathcal{M}^{def} \}, where each occurrence of \text{unit} stands for a fresh variable. This type is only populated when \mathcal{M}^{def} holds, so the effect of the entry is simply to add \mathcal{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 \text{Lib}' be the F7 interface, which includes, for example, the functions labelled...
“Refinement Types for MACs” in Section 3. The inductive definitions Lib^def include formulas defining the Pub and Bytes predicates, while Lib^i is includes the corellaries C1 and C2 in Section 3.

**Lemma 2** Lib = (�, assume Lib^def ⊔ Lib, I^1) is a refined module.

As another example, our RPC protocol consists of a refined module of the form: RPC = (I^2, assume RPC^def ⊔ RPC, (I^1, I^2)). Let RPC be the F# code for the protocol. The inductive definitions RPC^def include the right to left form of (KeyAB MACSays) from Section 3. The theorems RPC^i include (KeyAB Injective), (KeyAB Pub Bad), and the left to right form of (KeyAB MACSays) from Section 3. The exported interface (I^1, I^2) is made available to the opponent. Let I^2 be the library’s opponent interface, which is imported in Section 3. Let I^1 be the protocol-specific opponent interface from Section 3. As mentioned in that section, the module below imports I^2 and exports its members at the more abstract interface I^1. By introducing abstract types such as bytes with representation type x: bytes (Pub(x)).

**Lemma 3** RPC is a refined module.

**Composition of Refined Modules**

- We say M_1 = (E_1, X_1, I_1) composes with M_2 = (E_2, X_2, I_2) iff I_1 ⊂ E_2 and X_1 and X_2 are independent.
- For any triples M_1 = (E_1, assume M^1^def ⊔ Y_1, I_1) and M_2 = (E_2, assume M^2^def ⊔ Y_2, I_2) their composition M_1; M_2 is the triple (E_1, assume (M^1^def ⊔ M^2^def) ⊔ Y_1, Y_2, I_1).

**Lemma 4** (Composition). If refined module M_1 composes with refined module M_2 then M_1; M_2 is a refined module.

For example, the triple Lib; RPC is: (�, assume (Lib^def ⊔ RPC^def ⊔ L)[Y], (I^1, I^2)). By Lemma 4 (Composition), Lib; RPC is a refined module.

**Safety and Robust Safety Through Typing for Modules**

- A refined module (�, X,�) is semantically safe if and only if, the expression X[[]] is semantically safe.
- An I-opponent is an opponent O such that I ⊃ O = unit.
- A refined module (�, 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 (�, X,�) is semantically safe.

**Theorem 5** (Robust Safety). Every refined module (�, X, I) is robustly safe.

We can now prove Theorem 1. We have that Lib; RPC = (�, X, (I^1, I^2)) where X = assume (Lib^def ⊔ RPC^def) ⊔ Lib[RPC] is a refined module. By Theorem 5 (Robust Safety), (�, X, (I^1, I^2)) is robustly safe, which is to say that X[O] is semantically safe for every opponent O with I^1, I^2 ⊃ 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 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 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 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 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 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.
antec; 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:

```
private val aes_keygen: unit → kkey(SKey(k))
val aes_encrypt: (AES CBC *) → kkey → bbytes(SKKey(k) ∨ CanSymEncrypt(k, b)) ∨ (Pub(k) ∧ Pub(b)) → ebytes(IsEncryption(k, e, b))
val aes_decrypt: (AES CBC *) → kkey(SKey(k) ∨ Pub(k)) → ebytes → bbytes[(\(\forall p. IsEncryption(e, k, p) ⇒ e = p\)) ∧ (Pub(k) ⇒ 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 key `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 condition 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 return `p`. (The second precondition 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 Namprmpre 2008).

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

```
a → b: e | hmacsha1 k_pb e where e = aes k_ab t
```

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

```
a → b: aes k_pb(t | hmacsha1 k_ab t)
```

**MAC-and-Encrypt (as in SSH):**

```
a → b: aes k_pb(t | hmacsha1 k_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 plaintext `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 → (ekey + mk_key)[AuthEncKeyPair(ekey)]
val encrypt_then_mac: ec_key → mk_key → bbytes[(AuthEncKeyPair(ekey) ∧ CanSymEncrypt(e, b)) ∨ (Pub(e) ∧ Pub(mk) ∧ Pub(b))] → ebytes[IsAuthEncryption(e, mk, b)]
val verify_then_decrypt: ek_key → mk_key[AuthEncKeyPair(ekey) ∨ (Pub(e) ∧ Pub(mk) ∧ Pub(b))] → ebytes[CanSymEncrypt(e, mk, 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(e, e)` to get both integrity and confidentiality for `b`:

**Authenticated Encryption Key Usage:**

```
(\(\forall mk,ek,p,e.\ AuthEncKeyPair(mk,ek) ∧ IsEncryption(e,ek,p) \& CanSymEncrypt(e,ek,mk) ⇒ MACSays(mk,ek)\))
```

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 → b: rsa_oaep pk_b k_ab | 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 → (pk_key + sk_key)
{HyPubKey(pk) ∧ HyPrivKey(sk) ∧ PubPrivKeyPair(pk, sk)}
val hybrid_encrypt: kkey → bbytes
{(HyPubKey(k) ∧ CanHyEncrypt(k, b)) ∨ (Pub(k) ∧ Pub(b))} → ebytes[IsHyEncryption(e, k, b)]
val hybrid_decrypt: sk_key → bbytes[ebytes[HyPrivKey(sk) ∨ (Pub(sk) ∧ Pub(e))]] → bbytes[CanSymEncrypt(e, sk) ∨ (Pub(k) ⇒ 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 `hybdirEncrypt` is linked to the underlying `CanSymEncrypt` and `CanAsymEncrypt` cryptographic predicates as follows:

**Hybrid Encryption Key Usage:**

```
(\(\forall pk, kb. \ HyPubKey(pk) \& CanHyKey(SymKey(kb), pk) ⇒ CanSymEncrypt(pk, kb)\))
(\(\forall pk, kb. \ HySymKey(k, pk) \& CanHyEncrypt(pk, kb) ⇒ CanSymEncrypt(k, b)\))
```
To typecheck hybridDecryt, 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. $a \rightarrow b$: id $| a | b | aenc ka (na | id | a | b)$
2. $b \rightarrow s$: id $| a | b | aenc ka (na | id | a | b) | aenc kb (nb | id | a | b)$
3. $s \rightarrow b$: id $| aenc ka (na | kab) | aenc kb (nb | kab)$
4. $b \rightarrow a$: id $| aenc ka (na | kab)$

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 the full sequence of messages $\langle a, b, s \rangle$, 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 $i = 1$):**

1. $i \rightarrow a \rightarrow b$: id $| aenc k (i | m_i)$
2. $i+1, b \rightarrow a$: id $| aenc k (i+1 | m_{i+1})$

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. 2006b) 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 $t_1, \ldots, t_n$ using an RSA signing key $ska$ takes the form:

```
<Signature>
  <SignedInfo>...
  <Reference uri="#1"/>
  <Transforms><Transform Algorithm=C14n /> /></Transforms>
  <Reference uri="#2"/>
  <DigestMethod Algorithm=SHA1 /></Reference>
  <Reference uri="#3"/>
  <Reference uri="#4"/>
  <Reference uri="#5"/>
  <Reference uri="#6"/>
  <SignedInfo>
    <SignatureValue>base64 (sha1 (utf8 (c14n t1)))</SignatureValue>
    <KeyInfo>..a's X.509 Certificate</KeyInfo>
  </SignedInfo>
</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

<table>
<thead>
<tr>
<th>F# Program</th>
<th>F7 Typechecking</th>
<th>FS2PV Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>Interface</td>
<td>Queries</td>
</tr>
<tr>
<td>Implementation</td>
<td>Checking Time</td>
<td>Verifying Time</td>
</tr>
<tr>
<td>1167 lines</td>
<td>1103 lines</td>
<td>103 lines</td>
</tr>
<tr>
<td>260 lines</td>
<td>+ 255 lines</td>
<td>1m 29.9s</td>
</tr>
<tr>
<td>29 lines</td>
<td>+ 26 lines</td>
<td>10s</td>
</tr>
<tr>
<td>1702 lines</td>
<td>+ 11 lines</td>
<td>29 64s</td>
</tr>
<tr>
<td>475 lines</td>
<td>+ 7 lines</td>
<td>48.81s</td>
</tr>
<tr>
<td>7+1 lines</td>
<td>42+1 lines</td>
<td>12.0s</td>
</tr>
<tr>
<td>10s</td>
<td>+ 53 lines</td>
<td>10.9s</td>
</tr>
<tr>
<td>129 lines</td>
<td>+ 309 lines</td>
<td>+ 6m 3s</td>
</tr>
</tbody>
</table>

### Interfaces

- principals
- cryptographic patterns
- owra-rees
- owra-rees
- secure conversations
- web services security library
- x.509-based client auth
- password x.509 mutual auth
- x.509-based mutual auth
- windows cardspace

### Verifying Time

- 7+1 lines
- 5+1 lines
- 260 lines
- 475 lines
- 1702 lines
- 475 lines
- 7+1 lines
- 7+1 lines
- 7+1 lines
- 7+1 lines
- 7+1 lines
- 7+1 lines

### Interface

<table>
<thead>
<tr>
<th>Queries</th>
<th>Verifying Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1s</td>
<td>7+1+1</td>
</tr>
<tr>
<td>6.65s</td>
<td>7+1+1</td>
</tr>
<tr>
<td>48.81s</td>
<td>7+1+1</td>
</tr>
<tr>
<td>207 lines</td>
<td>6m 3s</td>
</tr>
<tr>
<td>29s</td>
<td>(Type Incorrect)</td>
</tr>
<tr>
<td>123 lines</td>
<td>(Not Verified Separately)</td>
</tr>
<tr>
<td>53 lines</td>
<td>(Not Verified Separately)</td>
</tr>
<tr>
<td>255 lines</td>
<td>(Type Incorrect)</td>
</tr>
<tr>
<td>111 lines</td>
<td>(Not Verified Separately)</td>
</tr>
<tr>
<td>475 lines</td>
<td>(Not Verified Separately)</td>
</tr>
</tbody>
</table>

### Libraries

- LibWS
- library
- authentication and secrecy of
- each of the needed web services security specifications.
- We write
- uses them to sign and then encrypt the response message
- signature keys have the structure defined in
- IsSignedInfo
- for our web services security library composed of LibX.
- Principals, Patterns, SOAP, WS-Addressing, XML-Signature,

### Composing Protocols: 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.

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):

1. **C → IP:** Request, ska pkIP [TokenRequest(RP, pkRP)]
2. **IP:** Issue token t = Token(id, C, RP, kt)
3. **IP → C:** Response1 [r; XML-Encrypt pkRP t]
4. **C → RP:** Request1, kt pkRP [r; m1]

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.

### 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, k1 k2 [m1; . . . ; 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; . . . ; mn] is the list of message elements protected by the protocol (m1 above). The corresponding responses are represented by Response, [m1; . . . ; mn].

### CardSpace Protocol (using X.509 Mutual Authentication):

1. **C → IP:** Request, ska pkIP [TokenRequest(RP, pkRP)]
2. **IP:** Issue token t = Token(id, C, RP, kt)
3. **IP → C:** Response1 [r; XML-Encrypt pkRP t]
4. **C → RP:** Request1, kt pkRP [r; m1]

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.
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 1 m 29.9 s, whereas ProVerif takes 8 m 2.2 s to verify the protocol. Even adding verification times for the libraries, we find that type-checking with F7 is much faster than ProVerif. Our typified 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 86 m 21 s 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

F62CV (Bhargavan et al. 2008a) is the first tool to verify properties in the computational model of implementation code of security protocols. F62CV 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 Catalin Hritcu suggested improvements to the paper.

### References


