Verified Interoperable Implementations of Security Protocols

Karthikeyan Bhargavan* Cédric Fournet* Andrew D. Gordon* Stephen Tse†

*Microsoft Research †University of Pennsylvania

Abstract

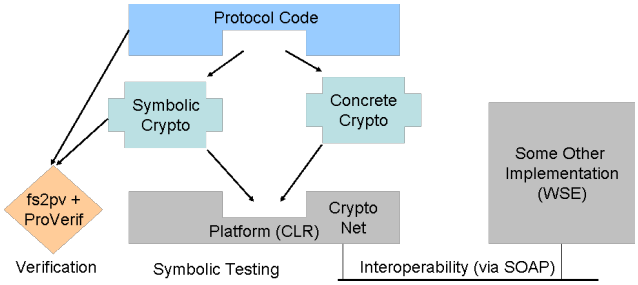
We present an architecture and tools for verifying implementations of security protocols. Our implementations can run with both concrete and symbolic implementations of cryptographic algorithms. The concrete implementation is for production and interoperability testing. The symbolic implementation is for debugging and formal verification. We develop our approach for protocols written in F#, a dialect of ML, and verify them by compilation to ProVerif, a resolution-based theorem prover for cryptographic protocols. We establish the correctness of this compilation scheme, and we illustrate our approach with protocols for Web Services security.

1. Introduction

The design and implementation of code involving cryptography remains dangerously difficult. The problem is to verify that an active attacker, possibly with access to some cryptographic keys but unable to guess other secrets, cannot thwart security goals such as authentication and secrecy [33]; it has motivated a serious research effort on the formal analysis of cryptographic protocols, starting with Dolev and Yao [16] and eventually leading to effective verification tools. Hence, it is now feasible to verify abstract models of protocols against demanding threat models.

Still, as with many formal methods, a gap remains between protocol models and their implementations. Distilling a cryptographic model is delicate and time consuming, so that verified protocols tend to be short and to abstract many potentially troublesome details of implementation code. At best, the model and its implementation are related during tedious manual code reviews. Even if, at some point, the model faithfully covers the details of the protocol, it is hard to keep it synchronized with code as it is deployed and used. Hence, despite verification of the abstract model, security flaws may appear in its implementation.

Our thesis is that to verify production code of security protocols against realistic threat models is an achievable research goal. The present paper advances in this direction by contributing a new approach to deriving automatically verifiable models from code. We demonstrate its application, if not to production code, at least to code constituting a working reference implementation—one suitable for interoperability testing with efficient production systems but itself optimized for clarity not performance.



Our prototype tools analyze cryptographic protocols written in F# [39], a dialect of ML. F# is a good fit for our purposes: it has a simple formal semantics; its datatypes offer a convenient way of programming operations on XML, important for our motivating application area, web services security. Semantically, F# is not so far from languages like Java or C#, and we expect our techniques could be adapted to such languages. We run F# programs on the Common Language Runtime (CLR), and rely on the .NET Framework libraries for networking and cryptographic functions.

The diagram above describes our new language-based approach, which derives verifiable models from executable code. We prefer not to tackle the converse problem, turning a formal model into code, as, though feasible, it amounts to language design and implementation, which generally is harder and takes more engineering effort than model extraction from an existing language. Besides, modern programming environments provide better tool support for writing code than for writing models.

We strive to share most of the code, syntactically and semantically, between the implementation and its model. Our approach is modular, as illustrated by the diagram: we write application code defining protocols against restrictive typed interfaces defining the services exposed by the underlying

To appear in the proceedings of the *19th IEEE Computer Security Foundations Workshop (CSFW 2006)*, July 5–7, 2006, Venice, Italy.

cryptographic, networking, and other libraries. Further, we write distinct versions of library code only for a few core interfaces, such as those featuring cryptographic algorithms. For example, cryptographic operations are on an abstract type bytes. We provide dual *concrete* and *symbolic* implementations of each operation. For instance, the concrete implementation of bytes is simply as byte arrays, subject to actual cryptographic transforms provided by the .NET Framework. On the other hand, the symbolic implementation defines bytes as algebraic expressions subject to abstract rewriting in the style of Dolev and Yao, and assumed to be a safe abstraction of the concrete implementation.

We formalize the active attacker as an arbitrary program in our source language, able to call interfaces defined by the application code and also the libraries for cryptography and networking. Our verification goals are to show secrecy and authentication properties in the face of all such attackers. Accordingly, we can adapt our threat model by designing suitable interfaces for the benefit of the attacker. The application code implements functions for each role in the protocol, so the attacker can create multiple instances of, say, initiators and responders, as well as monitor and send network traffic and, in some models, create new principals and compromise some of their credentials.

Given dual implementations for some libraries, we can compile and execute programs both concretely and symbolically. This supports the following tasks:

(1) To obtain a *reference implementation*, we execute application code against concrete libraries. We use the reference implementation for interoperability testing with some other available, black-box implementation. Experimental testing is essential to confirm that the protocol code is functionally correct, and complete for at least a few basic scenarios. (Otherwise, it is surprisingly easy to end up with a model that does not support some problematic features.)

(2) To obtain a *symbolic prototype*, we execute the same application code against symbolic libraries. This allows basic testing and debugging, especially for the expected message formats. Though this guarantees neither wire format interoperability nor any security properties, it is pragmatically useful during the initial stages of code development.

(3) To perform *formal verification*, we run our model extraction tool, called fs2pv, to derive a detailed formal model from the application code and symbolic libraries. Our models are in a variant of the pi calculus [30, 1] accepted by ProVerif [13, 12]. ProVerif compiles our models to logical clauses and runs a resolution semi-algorithm to prove properties automatically. In case a security property fails, ProVerif can often construct an explicit attack [4].

The fs2pv/ProVerif tool chain is applicable in principle to a broad range of cryptographic protocols, but our motivating examples are those based on the WS-Security [32] standard for securing SOAP [23] messages sent to and from XML web services. WS-Security prescribes how to sign and encrypt parts of SOAP messages. WSE [28] is an implementation of security protocols based on WS-Security. Previous analyses of pi calculus models extracted from WSE by hand have uncovered attacks [9, 10], but there has been no previous attempt to check conformance between these models and code automatically. To test the viability of our new approach, we have developed a series of reference implementations of simple web services protocols. They are both tested to be interoperable with WSE and verified via our tool chain. The research challenge in developing these implementations is to confront at once the difficulty of processing standard wire formats, such as WS-Security, and the difficulty of extracting verifiable models from code.

Our model extraction tool, fs2pv, accepts an expressive first-order subset of F# we dub F, with primitives for communications and concurrency. It has a simple formal semantics facilitating model extraction, but disallows higher-order functions and some imperative features. The application code and the symbolic libraries must be within F, but the concrete libraries are in unrestricted F#, with calls to the platform libraries. Formally, we define the attacker to be an arbitrary F program well formed with respect to a restrictive *attacker interface* implemented by the application code. The attacker can only interact with the application code via this interface, which is supplied explicitly to the model extraction tool along with the application code. Although we compile to the pi calculus for verification, the properties proved can be understood independently of the pi calculus. We prove theorems to justify that verification with ProVerif implies properties of source programs defined in terms of F. The principal difficulty in the proofs arises from relating the attacker models at the two levels.

Since security properties within the Dolev-Yao model are undecidable, and we rely on an automatic verifier, there is correct code within F that fails to verify. A cost of our method, then, is that we must adopt a programming discipline within F suitable for automatic verification. For example, we avoid certain uses of recursion. The initial performance results for our prototype tools are encouraging, as much of the performance is determined by the concrete libraries; nonetheless, there is a tension between efficiency of execution and feasibility of verification. To aid the latter, fs2pv chooses between a range of potential semantics for each F function definition (based on abstractions, rewrite rules, relations, and processes).

Our method relies on explicit interfaces describing low-level cryptographic and communication libraries, and on some embedded specifications describing the intended se-

curity properties. Model extraction directly analyzes application code using these interfaces plus the code of the symbolic libraries, while ignoring the code of the concrete libraries. Hence, our method can discover bugs in the application code, but not in the trusted concrete libraries.

At present, we have assessed our method only on new code written by ourselves in this style. Many existing protocol implementations rely on well defined interfaces providing cryptographic and other services, so we expect our method will adapt to existing code bases, but this remains future work.

In general, the derivation of security models from code amounts to translating the security-critical parts of the code and safely abstracting the rest. Given an arbitrary program, this task can hardly be automated—some help from the programmer is needed, at least to assert the intended security properties. Further work may discover how to compute safe abstractions directly from the code of concrete libraries. For now, we claim the benefit of symbolic verification of a reference implementation is worth the cost of adding some security assertions in application code and adopting a programming discipline compatible with verification.

In summary, our main contributions are as follows:

(1) An architecture and language semantics to support extraction of verifiable formal models from implementation code of security protocols.

(2) A prototype model extractor fs2pv that translates from F to ProVerif. This tool is one of the first to extract verifiable models from working protocol implementations. Moreover, to the best of our knowledge, it is the first to extract models from code that uses a standard message format (WS-Security) and hence interoperates with other implementations (WSE).

(3) Theorems justifying model extraction: low-level properties proved by ProVerif of a model extracted by fs2pv imply high-level properties expressed in terms of F.

(4) Reference implementations of some typical web services security protocols and mechanisms, both formally verified and tested for interoperability. Our implementation is modular, so that most code is expressed in reusable libraries that give a formal semantics to informal web services security specifications.

Section 2 informally introduces many ideas of the paper in the context of a simple message authentication protocol. Section 3 defines our source language, F, as a subset of F#, and formalizes our desired security properties. Section 4 outlines our techniques for model extraction, and states our main theorems. Section 5 summarizes our experience in writing and verifying code for web services security protocols. Section 6 concludes.

A companion report [11] provides additional technical details, including definitions for the source (F) and target (pi calculus) languages, the formal translation, and all proofs.

2. A Simple Message Authentication Protocol

We illustrate our method on a very simple, ad hoc protocol example. Section 5 discusses more involved examples.

The protocol Our example protocol has two roles, a client that sends a message, and a server that receives it. For the sake of simplicity, we assume that there is only one principal A acting as a client, and only one principal B acting as a server. (Further examples support arbitrarily many principals in each role.)

Our goal here is that the server authenticate the message, even in the presence of an active attacker. To this end, we rely on a password-based message authentication code (MAC). The protocol consists of a single message:

$$A \rightarrow B : \mathrm{HMACSHA1}\{nonce\}[pwd_A \mid text] \mid \mathrm{RSAEncrypt}\{pk_B\}[nonce] \mid text$$

The client acting for principal A sends a single message *text* to the server acting for B. The client and server share A's password pwd_A, and the client knows B's public key pk_B. To authenticate the message *text*, the client uses the one-way keyed hash algorithm HMAC-SHA1 to bind the message with pwd_A and a freshly generated value *nonce*. Since the password is likely to be a weak secret, that is, a secret with low entropy, it may be vulnerable to offline dictionary attacks if the MAC, the message *text*, and the nonce are all known. To protect the password from such guessing attacks, the client encrypts the nonce with pk_B.

Application code Given interfaces Crypto, Net, and Prins defining cryptographic primitives, communication operations, and access to a database of principal identities, our verifiable application code is a module that implements the following typed interface.

```
pkB: rsa_key
client: str → unit
server: unit → unit
```

The value pkB is the public encryption key for the server. Calling client with a string parameter should send a single message to the server, while calling server creates an instance of the server role that awaits a single message.

In F#, str → unit is the type of functions from the type str, which is an abstract type of strings defined by the Crypto interface, to the empty tuple type unit. The Crypto interface also provides the abstract type rsa_key of RSA keys.

The exported functions client and server rely on the following functions to manipulate messages.

```
let mac nonce password text =
  Crypto.hmacsha1 nonce
    (concat (utf8 password) (utf8 text))

let make text pk password =
  let nonce = mkNonce() in
  (mac nonce password text,
   Crypto.rsa_encrypt pk nonce, text)

let verify (m,en,text) sk password =
  let nonce = Crypto.rsa_decrypt sk en in
  if not (m = mac nonce password text)
  then failwith "bad MAC"
```

The first function, mac, takes three arguments—a nonce, a shared password, and the message text—and computes their joint cryptographic hash using some implementation of the HMAC-SHA1 algorithm provided by the cryptographic library. As usual in dialects of ML, types may be left implicit in code, but they are nonetheless verified by the compiler; mac has type bytes → str → str → bytes. The functions concat and utf8 provided by Crypto perform concatenation of byte arrays and an encoding of strings into byte arrays.

The two other functions define message processing, for senders and receivers, respectively. Function make creates a message: it generates a fresh nonce, computes the MAC, and also encrypts the nonce under the public key pk of the intended receiver, using the rsa_encrypt algorithm. The resulting message is a triple comprising the MAC, the encrypted nonce, and the text. Function verify performs the converse steps: it decrypts the nonce using the private key skd, recomputes the MAC and, if the resulting value differs from the received MAC, throws an exception (using the failwith primitive).

Although fairly high-level, our code includes enough details to be executable, such as the details of particular algorithms, and the necessary utf8 conversions from strings (for password and text) to byte arrays.

In the following code defining protocol roles, we rely on events to express intended security properties. Events roughly correspond to assertions used for debugging purposes, and they have no effect on the program execution. Here, we define two kinds of events, Send(text) to mark the intent to send a message with content text, and Accept(text) to mark the acceptance of text as genuine. Accordingly, client uses a primitive function log to log an event of the first kind before sending the message, and server logs an event of the second kind after verifying the message. Hence, if our protocol is correct, we expect every Accept(text) event to be preceded by a matching Send(text) event. Such a correspondence between events is a common way of specifying authentication.

The client code relies on the network address of the server, the shared password, and the server's public key:

```
let address = S "http://server.com/pwdmac"
let pwdA = Prins.getPassword(S "A")
let pkB = Prins.getPublicKey(S "B")

type Ev = Send of str | Accept of str

let client text =
  log(Send(text));
  Net.send address (marshall (make text pkB pwdA))
```

Here, the function getPassword retrieves *A*'s password from the password database, and getPublicKey extracts *B*'s public key from the local X.509 certificate database. The function S is defined by Crypto; the expression S "A", for example, is an abstract string representing the literal "A". The function client then runs the protocol for sending text; it builds the message, then uses Net.send, a networking function that posts the message as an HTTP request to address.

Symmetrically, the function server attempts to receive a single message by accepting a message and verifying its content, using *B*'s private key for decryption.

```
let skB = Prins.getPrivateKey(S "B")
let server () =
  let m,en,text = unmarshall (Net.accept address) in
  verify (m,en,text) skB pwdA; log(Accept(text))
```

The functions marshall and unmarshall serialize and deserialize the message triple—the MAC, the encrypted nonce, and the text—as a string, used here as a simple wire format. (We present an example of the resulting message below.) These functions are also part of the verified application code; we omit their details.

Concrete and symbolic libraries The application code listed above makes use of a Crypto library for cryptographic operations, a Net library for network operations, and a Prins library offering access to a principal database. The concrete implementations of these libraries are F# modules containing functions that are wrappers around the corresponding platform (.NET) cryptographic and network operations.

To obtain a complete symbolic model of the program, we also develop symbolic implementations of these libraries as F# modules with the same interfaces. These symbolic libraries are within the restricted subset F we define in the next section, and rely on a small module Pi defining name creation, channel-based communication, and concurrency in the style of the pi calculus. Functions Pi.send and Pi.recv allow message passing on channels, functions Pi.name and Pi.chan generate fresh names and channels, and a function Pi.fork runs its function argument in parallel. The members of Pi are primitive in the semantics of F. The Pi module is called from the symbolic libraries during symbolic evaluation and formal verification; it is not called directly from application code and plays no part in the concrete implementation.

```
module Crypto // concrete code in F#
open System.Security.Cryptography
type bytes = byte[]
type rsa_key = RSA of RSAParameters
...
let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
  let x = Bytearray.make 16 in
  rng.GetBytes x; x
...
let hmacsha1 k x =
  new HMACSHA1(k).ComputeHash x
...
let rsa = new RSACryptoServiceProvider()
let rsa_keygen () = ...
let rsa_pub (RSA r) = ...
let rsa_encrypt (RSA r) (v:bytes) = ...
let rsa_decrypt (RSA r) (v:bytes) =
  rsa.ImportParameters(r);
  rsa.Decrypt(v,false)
```

```
module Crypto // symbolic code in F
type bytes =
  | Name of Pi.name
  | HmacSha1 of bytes * bytes
  | RsaKey of rsa_key
  | RsaEncrypt of rsa_key * bytes
  ...
and rsa_key = PK of bytes | SK of bytes
...
let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"
...
let hmacsha1 k x = HmacSha1(k,x)
...
let rsa_keygen () = SK (freshbytes "rsa")
let rsa_pub (SK(s)) = PK(s)
let rsa_encrypt s t = RsaEncrypt(s,t)
let rsa_decrypt (SK(s)) e = match e with
  | RsaEncrypt(pke,t) when pke = PK(s) → t
  | _ → failwith "rsa_decrypt failed"
```

The listings above show the two implementations of the Crypto interface. The concrete implementation defines bytes as primitive arrays of bytes, and essentially forwards all calls to standard cryptographic libraries of the .NET platform. In contrast, the symbolic implementation defines bytes as an algebraic datatype, with symbolic constructors and pattern matching for representing cryptographic primitives. This internal representation is accessible only in this library implementation. For instance, hmacsha1 is implemented as a function that builds an HmacSha1(k,x) term; since no inverse function is provided, this abstractly defines a perfect, collision-free one-way function. More interestingly, RSA public key encryptions are represented by RsaEncrypt terms, decomposed only by a function rsa_decrypt that can verify that the valid decryption key is provided along with the encrypted term.

Similarly, the concrete implementation of Net contains functions, such as send and accept, that call into the platform's HTTP library (System.Net.WebRequest), whereas the symbolic implementation of these functions simply enqueues and dequeues messages from a shared buffer implemented with the Pi module as a channel. We outline the symbolic implementation of Net below.

```
module Net // symbolic code in F
...
let httpchan = Pi.chan()
let send address msg =
  Pi.send httpchan (address,msg)
let accept address =
  let (addr,msg) = Pi.recv httpchan in
  if addr = address then msg else ...
```

The function send adds a message to the channel httpchan and the function accept removes a message from the channel.

In this introductory example, we have a fixed population of two principals, so the values for *A*'s password and *B*'s key pair can simply be retrieved from the third interface Prins: the concrete implementation of Prins binds them to constants; its symbolic implementation binds them to fixed names generated by calling Pi.name. In general, a concrete implementation would retrieve keys from the operating system key store, or prompt the user for a password. The symbolic version implements a database of passwords and keys using a channel kept hidden from the attacker.

Next, we describe how to build both a concrete reference implementation and a symbolic prototype, in the sense of Section 1.

Concrete execution To test that the protocol runs correctly, we run the F# compiler on the F application code, the concrete F# implementations of Crypto, Net, and Prins, together with the following top-level F# code to obtain a single executable, say run. Depending on its command line argument, this executable runs in client or server mode:

```
do match Sys.argv.(1) with
  | "client" → client (S Sys.argv.(2))
  | "server" → server ()
  | _ → printf "Usage: run client txt\n";
        printf " or: run server\n"
```

The library function call Sys.argv.(*n*) returns the *n*th argument on the command line. As an example, we can exe-

cute the command run client Hi on some machine, execute run server on some other machine that listens on address, and observe the protocol run to completion. This run of the protocol involves our concrete implementation of (HTTP-based) communications sending and receiving the encoded string "FADCIzZhW3XmgUABgRJj1KjnWy...".

Symbolic execution To experiment with the protocol code symbolically, we run the F# compiler on the F application code, the symbolic F implementations of Crypto, Net, and Prins, and the F# implementation of the Pi interface, together with the following top-level F code, that conveniently runs instances of the client and of the server within a single executable.

```
do Pi.fork (fun()→ client (S "Hi"))
do server ()
```

The communicated message prints as follows

```
HMACSHA1{nonce3}[pwd1 | 'Hi'] |
  RSAEncrypt{PK(rsa_secret2)}[nonce3] | 'Hi'
```

where pwd1, rsa_secret2, and nonce3 are the symbolic names freshly generated by the Pi module. This message trace reveals the structure of the abstract byte arrays in the communicated message, and hence is more useful for debugging than the concrete message trace. We have found it useful to test application code by symbolic execution (and even symbolic debugging) before testing them concretely on a network.

Modelling the opponent We introduce our language-based threat model for protocols developed in F. (Section 3 describes the formal details.)

Let *S* be the F program that consists of the application code plus the symbolic libraries. The program *S*, which largely consists of code shared with the concrete implementation, constitutes our formal model of the protocol.

Let *O* be any F program that is well formed with respect to the interface exported by the application code (in this case, the value pkB and the functions client and server), plus the interfaces Crypto and Net. By well formed, we mean that *O* only uses external values and calls external functions explicitly listed in these interfaces. Moreover, *O* can call all the operations in the Pi interface, as these are primitives available to all F programs. We take the program *O* to represent a potential attacker on the formal model *S* of the protocol, a counterpart to an active attacker on a concrete implementation. (Treating an attacker as an arbitrary F program develops the idea of an attacker being an arbitrary parallel process, as in the spi calculus [2].)

Giving *O* access to the Crypto and Net interfaces, but not Prins, corresponds to the Dolev-Yao [16] model of an attacker able to perform symbolic cryptography, and monitor and send network traffic, but unable to access principals' credentials directly. In particular, Net.send enables the attacker to send any message to the server while Net.accept enables the attacker to intercept any message sent to the server. The functions Crypto.rsa_encrypt and Crypto.rsa_decrypt enable encryption and decryption with keys known to the attacker; Crypto.rsa_keygen and Crypto.mkNonce enable the generation of fresh keys and nonces; Crypto.hmacsha1 enables MAC computation.

Giving *O* access to client and server allows it to create arbitrarily many instances of protocol roles, while access to pkB lets *O* encrypt messages for the server. (We can enrich the interface to give the opponent access to the secret credentials of some principals, and to allow the generation of arbitrarily many principal identities.) Since pwdA, skB, and log are not included in the attacker interface, the attacker has no direct access to the protocol secrets and cannot log events directly.

Formal verification aims to establish secrecy and authentication properties for all programs *S O* assembled from the given system *S* and any attacker program *O*.

In particular, the message authentication property of our example protocol is expressed as correspondences [40] between events logged by code within *S*. For all *O*, we want that in every run of *S O*, every Accept event is preceded by a corresponding Send event. In our syntax (based on that of ProVerif), we express this correspondence assertion as:

```
ev:Accept(x) ⇒ ev:Send(x)
```

Formal verification We can check correspondences at runtime during any particular symbolic run of the program; the more ambitious goal of formal verification is to prove them for all possible runs and attackers. To do so, we run our model extractor fs2pv on the F application code, the symbolic F implementations of Crypto, Net, and Prins, and the attacker interface as described above. The result is a pi calculus script with embedded correspondence assertions suitable for verification with ProVerif. In the simplest case, F functions compile to pi calculus processes, while the attacker interface determines which names are published to the pi calculus attacker. For our protocol, ProVerif immediately succeeds.

Conversely, consider for instance a variant of the protocol where the MAC computation does not actually depend on the text of the message—essentially transforming the MAC into a session cookie:

```
let mac nonce password text = hmacsha1 nonce
  (concat (utf8 password) (utf8 (S "cookie")))
```

For the resulting script, ProVerif automatically finds and reports an active attack, whereby the attacker intercepts the client message and substitutes any text for the client's text in the message. Experimentally, we can confirm the attack

found in the analysis, by writing in F an instance of the attacker program O that exploits our interface. Here, the attack may be written:

```
do fork(fun()→ client (S "Hi"));
   let (nonce, mac, _) = unmarshall (Net.accept address) in
   fork(fun()→ server());
   Net.send address (marshall (nonce, mac, S "Foo"))
```

This code first starts an instance of the client, intercepts its message, starts an instance of the server, and forwards an amended message to it. Experimentally, we observe that the attack succeeds, both concretely and symbolically. At the end of those runs, two events Send `"Hi"` and Accept `"Foo"` have been emitted, and our authentication query fails. Once the attack is identified and the protocol corrected, this attacker code may be added to the test suite for the protocol.

In addition to authentication, we verify secrecy properties for our example protocol. Via ProVerif [13], we can query whether a protocol allows an attacker to guess a weak secret and then verify the guess—if so, the attacker can mount an offline guessing attack. In the case of our protocol, ProVerif shows the password is protected against offline guessing attacks. Conversely, if we consider a variant of the protocol that passes the nonce in the clear, we find an attack that can also be written as a concrete F program.

3. Formalizing a Subset of F#

This section defines the untyped subset F of F# in which we write application code and symbolic libraries. We define the syntax, sketch the fairly standard semantics, and define security properties. Some details are left to the technical report [11].

The language F consists of: a first-order functional core; algebraic datatypes with pattern-matching (such as the type bytes in the symbolic implementation of Crypto); a few concurrency primitives in the style of the pi calculus; and a simple type-free module system with which we formalize the attacker model introduced in the previous section. (Although we do not rely on type safety in the formal definition, F programs can be typechecked by the F# compiler.)

In the syntax below, ℓ ranges over functions (such as freshBytes or hmacsha1 in Crypto) and f ranges over datatype constructors (such as Name or Hmacsha1 in the type bytes in Crypto). Functions and constructors are either primitive, or introduced by function or datatype declarations. The primitives include the communication and concurrency functions Pi.send, Pi.recv, Pi.name, Pi.fork described in the previous section. In F, we treat Pi.chan as a synonym for Pi.name; they have different types but both create fresh atomic names. We omit the "Pi." prefix for brevity. Tuple $((e_1,\ldots,e_n))$, conditional (**if** e **then** e_1 **else** e_2), equality ($e_1 = e_2$), sequencing ($e_1;e_2$), and other expressions can be derived within this core syntax.

Syntax of F:

x, y, z		variable
a, b		name
f		constructor (uncurried)
ℓ		function (curried)
true, false, tuple$n$	$n \geq 0$	primitive constructors
name, send, recv, log, failwith		primitive functions
$M, N ::=$		value
$\quad x$		variable
$\quad a$		name
$\quad f(M_1,\ldots,M_n)$		constructor application
$e ::=$		expression
$\quad M$		value
$\quad \ell\ M_1\ \ldots\ M_n$		function application
$\quad$ fork(**fun**()→e)		fork a parallel thread
$\quad$ **match** M **with**$(\mid M_i \rightarrow e_i)^{i\in 1..n}$		pattern match
$\quad$ **let** $x = e_1$ **in** e_2		sequential evaluation
$d ::=$		declaration
$\quad$ **type** $s = (\mid f_i$ **of** $s_{i1} * \ldots * s_{im_i})^{i\in 1..n}$		datatype declaration
$\quad$ **let** $x\ = e$		value declaration
$\quad$ **let** $\ell\ x_1 \ldots x_n = e$	$n > 0$	function declaration
$S ::= d_1 \cdots d_n$		system: list of declarations

A system S is a sequence of declarations. We write the list S as $\varnothing$ when it is empty. A datatype declaration introduces a new type and its constructors (much like a union type with tags in C); the type expressions s, s_{ij} are ignored in F. A value declaration **let** $x\ = e$ triggers the evaluation of expression e and binds the result to x. A function declaration **let** $\ell\ x_1 \ldots x_n = e$ defines function ℓ with formal parameters $x_1 \ldots x_n$ and function body e. These functions may be recursive.

A value M is a variable, a name, or a constructor application. A name is only introduced during evaluation by the primitive name to model the generation of channels, keys, and nonces; source programs contain no free names. Expressions denote potentially concurrent computations that return values. Primitive functions mostly represent communication and concurrency: name() returns a freshly generated name; send M N sends N on the channel M; recv M returns the next value received on channel M; log M logs the event M; failwith M represents a thrown exception; and fork(**fun**()→e) evaluates e in parallel. (We need not model exception handling in F as we rely on exceptions only to represent fatal errors.) If ℓ has a declaration, the application $\ell\ M_1\ \ldots\ M_n$ invokes the body of the declaration with actual parameters $M_1, \ldots, M_n$. A **match** M **with**$(\mid M_i \rightarrow e_i)^{i\in 1..n}$ runs e_i for the least i such that pattern M_i matches the value M; if the pattern M_i contains variables, they are bound

in e_i by matching with M. If there are two or more occurrences of a variable in a pattern, matching must bind each to the same value. (Strictly speaking, F# forbids patterns with multiple occurrences of the same variable. Still, the effect of any such pattern in F can be had in F# by renaming all but one of the occurrences and adding one or more equality constraints via a **when** clause.) Finally, **let** $x = e_1$ **in** e_2 first evaluates e_1 to a value M, then evaluates $e_2\{M/x\}$, that is, the outcome of substituting M for each free occurrence of x in e_2.

Next, we sketch the operational semantics of F and the idea of safety with respect to a query. A *configuration*, C, is a multiset of running systems and logged events. We equip configurations with a small-step reduction semantics: $C \to C'$ means that C can take one step to C'. Most reductions arise from evaluation of expressions within systems as described above.

Operational Semantics of F:

$C ::= S \mid$ **event** $M \mid (C \mid C')$ multiset of events and systems
$C \equiv C'$ equality up to laws that $\mid$ is associative and commutative, with $\varnothing$ as neutral element.
Let $C \to C'$ mean there is a computation step from C to C'.
Let $C \to^*_{\equiv} C'$ if and only if either $C \equiv C'$ or $C \to^* C'$.

We express authentication and other properties in terms of event-based queries. The general form of a query is **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n, which means that every reachable configuration containing an event matching the pattern E also contains an event matching one of the B_i patterns.

Queries and Safety:

A *query* q is written **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n
for values $E, B_1, \ldots, B_n$ containing no free names.
Let σ stand for a substitution $\{M_1/x_1, \ldots, M_n/x_n\}$.
Let $C \models$ **query ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n if and only if
$C' \equiv$ **event** $B_i\sigma \mid C''$ for some $i \in 1..n$,
whenever $C \equiv$ **event** $E\sigma \mid C'$.
Let S be *safe for* q if and only if $C \models q$ whenever $S \to^*_{\equiv} C$.

For example, a system is *safe* for query **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) from Section 2 if every reachable configuration containing **event** Accept(M) also contains **event** Send(M).

We introduce *interfaces* I to record the set of values, constructors, and functions imported or exported by a system. Since our verification method does not depend on types, F interfaces omit type structure and track only the distinction between values, constructors, and functions, plus the arities of constructors and functions. The judgment $I \vdash S : I'$ means S refers only to external values, constructors, and functions listed in I, and provides declarations for the values, constructors, and functions listed in I'. (Our technical report contains the simple inference rules for this judgment. These rules are an abstraction of the typing rules of F# for the fragment we consider. They are automatically enforced by the F# compiler.)

Interfaces:

$\mu ::=$	mention
x:**val** $\mid f$:**ctor** $n \mid \ell$:**fun** n	value, constructor, or function
$I ::= \mu_1, \ldots, \mu_n$	interface (unordered sequence)

Let $I \vdash S : I'$ mean S is well formed given I, and exports I'.

For example, here is the F interface that corresponds to the typed interface implemented by application code in Section 2.

pkB: **val**, client: **fun** 1, server: **fun** 1

We define a Prim interface to describe the F primitives, where m is an arbitrary maximum width of tuples:

true: **ctor** 0, false: **ctor** 0, (tuple$i$: **ctor** i)$^{i \in 1..m}$,
failwith: **fun** 1, log: **fun** 1, Pi.name: **fun** 1, Pi.chan: **fun** 1,
Pi.send: **fun** 2, Pi.recv: **fun** 1, Pi.fork: **fun** 1

We define a robust safety property, that is, safety in the presence of an opponent. To avoid vacuous failures, we forbid the opponent from logging events. If I is an interface, an I-opponent is a system O that depends only on I and Prim, but not log.

Formal Threat Model: Opponents and Robust Safety

Let $S :: I_{pub}$ iff Prim $\vdash S : I_{pub}, I_{priv}$ for some I_{priv}.
Let O be an *I-opponent* iff Prim$\backslash$log$, I \vdash O : I'$ for some I'.
Let S be *robustly safe for q and I* iff
$S :: I$ and $S\ O$ is safe for q for all I-opponents O.

Hence, setting a verification problem for a system S essentially amounts to selecting the subset I_{pub} of its interface that is made available to the opponent.

For the example protocol in Section 2, let S be the system that consists of application code and symbolic libraries, and let I_{pub} be the following interface.

Net.send: **fun** 2, Net.accept: **fun** 1,
Crypto.S: **fun** 1, Crypto.iS: **fun** 1,
Crypto.base64: **fun** 1, Crypto.ibase64: **fun** 1,
Crypto.utf8: **fun** 1, Crypto.iutf8: **fun** 1,
Crypto.concat: **fun** 1, Crypto.iconcat: **fun** 1,
Crypto.concat3: **fun** 1, Crypto.iconcat3: **fun** 1,
Crypto.mkNonce: **fun** 1, Crypto.mkPassword: **fun** 1,
Crypto.rsa_keygen: **fun** 1, Crypto.rsa_pub: **fun** 1,
Crypto.rsa_encrypt: **fun** 2, Crypto.rsa_decrypt: **fun** 2,
Crypto.hmacsha1: **fun** 2,
pkB: **val**, client: **fun** 1, server: **fun** 1

Our verification problem is to show that S is robustly safe for **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) and I_{pub}.

4. Mapping F# to a Verifiable Model

We target the script language of ProVerif for verification purposes. ProVerif can establish correspondence and secrecy properties for protocols expressed in a variant of the pi calculus, whose syntax and semantics are detailed in our technical report. In this calculus, active attackers are represented as arbitrary processes that run in parallel, communicate with the protocol on free channels, and perform symbolic computations. Given a script that defines the protocol, the capabilities of the attacker, and some target query, ProVerif generates logical clauses then uses a resolution-based semi-algorithm. When ProVerif completes successfully, the script is *robustly safe* for the target query, that is, the query holds against all (pi calculus) attackers; otherwise, ProVerif attempts to reconstruct an attack trace. ProVerif may also diverge, or fail, as can be expected since query verification in the pi calculus is not decidable. (ProVerif is known to terminate for the special class of *tagged* protocols [14]. However, the protocols in our main application area of web services rarely fall in this class.) ProVerif is a good match for our purposes, as it offers both general soundness theorems and an effective implementation. Pragmatically, we also rely on previous positive experience in generating large verification scripts for ProVerif. In principle, however, we may benefit from any other verification tool.

To obtain a ProVerif script, we translate F programs to pi calculus processes and rewrite rules. To help ProVerif succeed, we use a flexible combination of several translations. To validate our usage of ProVerif, we also formally relate arbitrary attackers in the pi calculus to those expressible in F.

At its core, our translation maps functions to processes using the classic call-by-value encoding from lambda calculus to pi calculus [29]. For instance, we may translate the mac function declaration of Section 2

```
let mac nonce pwd text =
  Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

into the process

```
!in(mac, (nonce,pwd,text,k));
 out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

This process is a replicated input on channel mac; each message on mac carries the functional arguments (nonce,pwd,text) as well as a continuation channel k. When the function completes, it sends back a message that carries its result on channel k. Similarly, we translate the server function declaration of Section 2 into:

```
!in(server, (arg,kR));
 new kX; out(accept, (address,kX)); in(kX,xml);
 new kM; out(unmarshall, (xml,kM)); in(kM,(m,en,text));
 new kV; out(verify, ((m,en,text),sk,pwd,kV)); in(kV,());
 event Ev(Accept(text));
 out(kR, ())
```

This process first calls function accept as follows: it generates a fresh continuation channel kX; it sends a message that carries the argument address and kX on channel accept; and it receives the function result xml on channel kX. The process then similarly calls the functions unmarshall and verify. If both calls succeed, the process finally logs the event Accept(text) and returns an (empty) result on kR.

Our pi calculus includes the same term algebra—values built from variables, names, and constructors—as F, so values are unchanged by the translation. Moreover, our pi calculus includes term destructors defined by rewrite rules on the term algebra, and whenever possible after inlining, our implementation maps simple functions to destructors. For instance, we actually translate the mac function declaration into the native ProVerif reduction:

```
reduc mac(nonce,pwd,text) =
  HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```

Both formulations of mac are equivalent, but the latter is more efficient. On the other hand, complex functions with side-effects, recursion, or non-determinism are translated as processes. Our tool also supports a third potential translation for mac, into a ProVerif predicate declaration; predicates are more efficient than processes and more expressive than reductions. Our translation first performs aggressive inlining of F functions, constant propagation, and similar optimizations. It then globally picks the best applicable formulation for each reachable function, while eliminating dead code.

Finally, the translation gives to the pi calculus context the capabilities available to attackers in F. For example, the channel httpchan representing network communication is exported to the context in an initialization message. More interestingly, every public function coded as a process is made available on an exported channel.

For instance, the server function is available to the attacker; accordingly, we generate the process:

```
!in(serverPUB, (arg,kR)); out(server, (arg,kR))
```

This enables the attacker to trigger instances of the server using the public channel serverPUB. Conversely, the private channel server is used only by the translation, so that the attacker cannot intercept local function calls.

Formally, we define translations for expressions e, declarations d, and systems S. The translation $\mathscr{E}[\![e]\!](x,P)$ is a process that binds variable x to the value of e and then runs process P. The translations $\mathscr{S}[\![d]\!](P)$ and $\mathscr{S}[\![S]\!](P)$ are processes that elaborate d and S, and then run process P. At the top level, the translation $[\![S :: I_{pub}]\!]$ is a ProVerif script that includes constructor definitions for the datatypes in S and

defines a process that elaborates S and then exports I_{pub}. Details of these translations are in the technical report.

Our main correctness result is the following.

Theorem 1 (Reflection of Robust Safety) *If $S :: I_{pub}$ and $[\![S :: I_{pub}]\!]$ is robustly safe for q, then S is robustly safe for q and I_{pub}.*

In the statement of the theorem, S is the series of modules that define our system; I_{pub} is a selection of the values, constructors, and functions declared in S that are made available to the attacker; q is our target security query; and $[\![S :: I_{pub}]\!]$ is the ProVerif script obtained from S and I_{pub}.

The proof of Theorem 1 appears in our technical report; it relies on an operational correspondence between reductions on F configurations and reductions in the pi calculus.

We implement our translation as a command line tool fs2pv that intercepts code after the F# compiler front-end. The tool takes as input a series of module implementations defining S and module interfaces bounding the attacker's capabilities, much like I_{pub}. The tool relies on the typing discipline of F# (which is stronger than the scope discipline of F) to enforce that $S :: I_{pub}$. It then generates the script $[\![S :: I_{pub}]\!]$ and runs ProVerif. If ProVerif completes successfully, it follows that $[\![S :: I_{pub}]\!]$ is robustly safe for q. Hence, by Theorem 1, we conclude that S is robustly safe for q and I_{pub}.

As a simple example, recall the system S and its interface I_{pub}, as stated at the end of Section 3. Our tool runs successfully on this input, proving that S is robustly safe for the query **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) and I_{pub}.

5. Verification of Interoperable Code

To validate our approach experimentally, we implemented a series of cryptographic protocols and verified their security against demanding threat models.

Tables 1 and 2 summarize our results for these protocols. For each protocol, Table 1 gives the program size for the implementation (in lines of F# code, excluding interfaces and code for shared libraries), the number of messages exchanged, and the size of each message, measured both in bytes for concrete runs and in number of constructors for symbolic runs. Table 2 concerns verification; it gives the number of queries and the kinds of security properties they express. A secrecy query requires that a password (pwd) or key (key) be protected; a weak-secrecy query further requires that a weak secret (weak pwd) be protected from a guessing attack. An authentication query requires that a message content (msg), its sender (sender), or the whole exchange (session) be authentic. Some queries can be verified even in the presence of attackers that control some corrupted principals, thereby getting access to their keys and passwords. Not all queries hold for all protocols; in fact some queries are designed to test the boundaries of the attacker model and are meant to fail during verification. Finally, the table gives the size of the logical model generated by ProVerif (the number of logical clauses) and its total running time to verify all queries for the protocol.

For example, consider the simple authentication protocol of Section 2, named *Password-based MAC* in the tables; its implementation has 38 lines of specific code; ProVerif takes less than one second to verify the message authentication query and to verify that the protocol protects the password from guessing attacks. A variant of our implementation for this protocol (second row of Tables 1 and 2) produces the same message, but is more modular and relies on more realistic libraries; it supports distributed runs and enables the verification of queries against active attackers that may selectively corrupt some principals and get access to their keys and passwords.

As a benchmark, we wrote a program for the four message Otway-Rees key establishment protocol [34], with two additional messages after key establishment to probe the secrecy of message payloads encrypted with this key. To complete a concrete, distributed implementation, we had to code detailed message formats, left ambiguous in the description of the protocol. In the process, we inadvertently enabled a typing attack, immediately found by verification. We experimented with a series of 16 authentication and secrecy queries; their verification takes a few minutes.

A Library for Web Services Security As a larger, more challenging case study, we implemented and verified several web services security protocols.

Web services are applications that exchange XML messages conforming to the SOAP standard [23]. To secure these exchanges, messages may include a security header, defined in the WS-Security standard [32], that contains signatures, ciphertexts, and a range of security elements, such as tokens that identify particular principals. Hence, each secure web service implements a security protocol by composing mechanisms defined in WS-Security. Previous analyses of such WS-Security protocols established correctness theorems [21, 9, 7, 25, 26] and uncovered attacks [9, 10]. However, these analyses operated on models of protocols and not on their implementations. In the rest of this section, we present the first verification results for the security of interoperable web services implementations.

First, we develop a library in F that implements the formats and mechanisms of the web services messaging and security specifications. Like WSE [28], our library is a partial implementation of these specifications; we selected features based on the need to interoperate with protocols implemented by WSE. Our library provides several modules:

- Soap implements the SOAP formats for requests, responses, and faults, and their exchange via HTTP.

Protocol	Implementation			
	LOCs	messages	bytes	symbols
Password-based MAC	38	1	208	16
Password-based MAC variant	75	1	238	21
Otway-Rees	148	4	74; 140; 134; 68	24; 40; 20; 11
WS password-based signing	85	1	3835	394
WS X.509 signing	85	1	4650	389
WS password-based MAC	85	1	6206	486
WS request-response	149	2	6206; 3187	486; 542

Table 1. Summary of example protocols

Protocol	Security Goals				Verification	
	queries	secrecy	authentication	insiders	clauses	time
Password-based MAC	4	weak pwd	msg	no	69	0.8s
Password-based MAC variant	5	pwd	msg, sender	yes	213	2.2s
Otway-Rees	16	key	msg, sender	yes	155	1m50s
WS password-based signing	5	no	msg, sender	yes	456	5.3 s
WS X.509 signing	5	no	msg, sender	yes	460	2.6 s
WS password-based MAC	3	weak pwd	msg, sender	no	436	10.9s
WS request-response	15	no	session	yes	503	44m45s

Table 2. Verification Results

- Wsaddressing implements the WS-Addressing [15] header formats, for message routing and correlation.
- Xmldsig and Xmlenc implement the standards for XML digital signature [18] and XML encryption [17], which provide flexible formats for selectively signing and encrypting parts of an XML document.
- Wssecurity implements the WS-Security header format and common security tokens, such as username tokens, encrypted keys, and X.509 certificates.

These modules rely on the Crypto module for cryptographic functions and a new Xml module (with dual symbolic and concrete implementations) for raw XML manipulation.

Applications written with this library produce and consume SOAP messages that conform to the web services specifications. Such applications can interoperate with other conformant web services, such as those that use WSE.

The requirement to produce concrete, interoperable, and verifiable code is quite demanding, but it yields very precise executable models for the informal WS-Security specifications, more detailed than any available in the literature. For verifiability, we adopt a programming discipline that reduces the flexibility of message formats wherever possible. In particular, we fix the order of headers in a message and limit the number of headers that can be signed. We avoid higher-order functions (such as List.map) and recursion over lists and XML, and instead inline these functions by hand.

The library consists of 1200 lines of F code. We can quickly write security protocols using this library, such as an authentication protocol that uses a password or an X.509 certificate to generate an XML digital signature (protocols *WS Password-based signing* and *WS X.509 signing* in Tables 1 and 2). Only 85 additional lines of code need to be written to implement these protocols; their verification takes a few seconds.

A Simple Authentication Protocol over WS-Security As a case study, we used our web services library to implement an existing password-based authentication protocol (*WS password-based MAC*) taken from the WSE samples. The protocol is quite similar to *Password-based MAC*, except that the message is now a standards-compliant XML document. This message is sent as the body of a SOAP envelope that includes a WS-Security security header that contains a *username token*, representing the client's identity, and an *X.509 token*, representing the server's identity. The username token includes a freshly generated nonce used, along with a shared password, to derive a key for message authentication. This nonce is protected by encrypting the entire username token with the server's public key, using XML encryption. The message is authenticated by an XML digital signature that includes a cryptographic keyed hash of the body using a key derived from the username token.

In earlier work [10], we wrote a non-executable formal model for this protocol and analyzed it with ProVerif. Here, we extract the model directly from a full-fledged implemen-

tation. Moreover, we encode a more realistic threat model that enables the attacker to gain access to some passwords and keys. In particular, the Prins module has two additional functions in its interface: leakPassword and leakPrivateKey.

The leakPassword function is defined as follows:

```
let leakPassword (u:str) =
  let pwd = getPassword u in log Leak(u); pwd
```

When the attacker calls leakPassword for a principal u, the function extracts the password for u from the database and returns it to the attacker; but before leaking the password, the function logs an event Leak(u) recording that the principal u has been compromised.

We implement the client and server roles using our library, with slightly different Send and Accept events from the ones in Section 2. To enable sender authentication, the client logs Send(u,m), where u is the principal that sends the XML message m. Similarly, on receiving the message, the server logs Accept(u,m). The datatype of events and the authentication query becomes

```
type Ev = Send of str*item
        | Accept of str*item
        | Leak of str
q = ev:Accept(u,m) ⇒ ev:Send(u,m) ∨ ev:Leak(u)
```

where item is the datatype of XML elements. The query q asks that the server authenticate the message m and the sending principal u, unless u has been leaked. Let W be the system that consists of the client and server code, the symbolic libraries (Crypto, Net, Prins, and Xml), and the web services library. Let I_{pub} be the interface of Section 3 extended with the item datatype. Using fs2pv and ProVerif, we prove that W is robustly safe for q and I_{pub}. The verification of message and sender authentication takes only a few seconds. As with *Password-based MAC*, we also prove that the password is protected even if it is a weak secret.

We experimentally checked that our concrete implementation complies with the web services specifications: we can run our client with a WSE server, and conversely access our server from a WSE client. Many details of our model would have been difficult to determine from the specifications alone, without interoperability testing. The resulting messages exchanged by the concrete execution are around 6 kilobytes in size, while the symbolic execution of the protocol generates messages with 486 symbols. The runtime performance of our concrete implementation is comparable to WSE, which is not surprising for this protocol, since the execution time is dominated by XML processing and communication.

We also implemented and verified an extension of the protocol described above, where the server, upon accepting the request message, sends back a response message signed with the private key associated with its X.509 certificate. For this two message protocol, the security goals are authentication of the request and the response, as well as correlation between the messages. Correlation relies on a mechanism called *signature confirmation* (described in a draft revision of WS-Security), where the response echoes and signs the password-based signature value of the request. The protocol is named *WS request-response* in the tables; ProVerif establishes all our authentication and correlation goals, but takes almost 45 minutes for the analysis.

Our protocol implementation can also be used as part of a larger web application, while still benefiting from our results. The client functions can be exported as a library invoked by applications written in any language running on the CLR, such as C# or Visual Basic. Similarly, the server functions can be embedded in the security stack of a web server that checks all incoming messages for conformance to the protocol before handing over the message body to a web application written in any language. In both cases, assuming the application code does not have access to secret passwords or keys, the security results transparently apply.

6. Conclusions

We describe an architecture and programming model for security protocols. For production use, protocol code runs against concrete cryptography and low-level networking libraries. For initial development, the same code runs against symbolic cryptography and intra-process communication libraries. For verification, much of the code translates to a low-level pi calculus model for analysis against a Dolev-Yao attacker. The attacker can be understood and customized in source-level terms as an arbitrary program running against an interface exported by the protocol code.

Our prototype implementation is the first, we believe, to extract verifiable models from code implementing standard security protocols, and hence able to interoperate with other implementations. Our prototype has many limitations; still, we conclude that it significantly reduces the gap between symbolic models of cryptographic protocols and their implementations.

Limits of our model As usual, formal security guarantees hold only within the boundaries of the model being considered. Automated model extraction, such as ours, enables the formal verification of large, detailed models closely related to implementations. In our experience, such models are more likely to encompass security flaws than those focusing on protocols in isolation. Independently of our work, modelling can be refined in various directions. Certified compilers and runtime environments can give strong guarantees that program executions comply with their formal semantics; in our setting, they may help bridge the gap between the semantics of F and a low-level model of its native-code execution, dealing for instance with memory safety.

Our approach also crucially relies on the soundness of symbolic cryptography with regards to one implementation of concrete cryptography, which is far from obvious. Pragmatically, our modelling of symbolic cryptography is flexible enough to accommodate many known weaknesses of cryptographic algorithms (introducing for instance symbolic cryptographic functions "for the attacker only"). There is a lot of interesting research on reconciling symbolic cryptography with more precise computational models [3, 6]. Still, for the time being, these models do not support automated analyses on the scale needed for our protocols.

Related work The ideas of modelling protocol roles as functions and modelling an active attacker as an arbitrary functional context appear earlier in Sumii and Pierce's studies of cryptographic protocols within a lambda calculus [37, 38]. Unlike our functional language, which has state and concurrency, their calculus cannot directly capture linearity properties (such as replay detection via nonces), as its only imperative feature is name generation. Several systems [35, 31, 27, 36] operate in the reverse direction, and generate runnable code from abstract models of cryptographic protocols in formalisms such as strand spaces, CAPSL, and the spi calculus. These systems need to augment the underlying formalisms to express implementation details that are ignored in proofs, such as message sizes and error handlers. Going further in the direction of growing a formalism into a programming language, Guttman, Herzog, Ramsdell, and Sniffen [24] propose a new programming language CPPL for writing security protocols; CPPL combines features for communication and cryptography with a trust management engine for logically-defined authorization checks. CPPL programs can be verified using strand space techniques, although there is no automatic support for this at present. A limitation of all of these systems is that they do not implement standard message formats and hence do not interoperate with other implementations. In terms of engineering effort, it seems easier to achieve interoperability by starting from an existing general purpose language such as F# than by developing a new compiler.

Giambiagi and Dam [20] take a different approach to showing the conformance of implementation to model. They neither translate model to code, nor code to model. Instead, they assume both are provided by the programmer, and develop a theory to show that the information flows allowed by the implementation of a cryptographic protocol are none other than those allowed by the abstract model of the protocol. They treat the abstract protocol as a specification for the implementation, and implicitly assume correctness of the abstract protocol.

Askarov and Sabelfeld [5] report a substantial distributed implementation within the Jif security-typed language of a cryptographic protocol for online poker without a trusted third party. Their goal is to prevent some insecure information flows by typing. They do not derive a formal model of the protocol from their code.

There are only a few works on compiling implementation files for cryptographic protocols to formal models. Bhargavan, Fournet, and Gordon [8] translate the policy files for web services to the TulaFale modelling language [10], for verification by compilation to ProVerif. This translation can detect protocol errors in policy settings, but applies to configuration files rather than executable source code. Other symbolic modelling [21, 9, 7, 25, 26] of web services security protocols has uncovered a range of potential attacks, but has no formal connection to source code. Goubault-Larrecq and Parrennes [22] are the first to derive a Dolev-Yao model from implementation code written in C. Their tool Csur performs an interprocedural points-to analysis on C code to yield Horn clauses suitable for input to a resolution prover. They demonstrate Csur on code implementing the initiator role of the Needham-Schroeder public-key protocol.

There is also recent research on verifying implementations of cryptographic algorithms, as opposed to protocols. For instance, Cryptol [19] is a language-based approach to verifying implementations of algorithms such as AES.

Acknowledgements James Margetson and Don Syme helped us enormously with using and adapting the F# compiler. Tony Hoare and David Langworthy suggested improvements to the presentation.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 140–154, 2005.

[5] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *10th European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *LNCS*, pages 197–221. Springer, 2005.

[6] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *Proceedings*

of the 10th ACM Conference on Computer and Communications Security (CCS'03), pages 220–230. ACM Press, 2003.

[7] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. In *2004 ACM Workshop on Secure Web Services (SWS)*, pages 11–22, Oct. 2004.

[8] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, Oct. 2004.

[9] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. *Theoretical Comput. Sci.*, 340(1):102–153, June 2005.

[10] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004.

[11] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. Technical Report MSR–TR–2006–46, Microsoft Research, 2006.

[12] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

[13] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340, 2005.

[14] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, 2005.

[15] D. Box, F. Curbera, et al. *Web Services Addressing (WS-Addressing)*, Aug. 2004. W3C Member Submission.

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

[17] D. Eastlake, J. Reagle, et al. *XML Encryption Syntax and Processing*, 2002. W3C Recommendation.

[18] D. Eastlake, J. Reagle, D. Solo, et al. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation.

[19] Galois Connections. *Cryptol Reference Manual*, 2005.

[20] P. Giambiagi and M. Dam. On the secure implementation of security protocols. *Science of Computer Programming*, 50:73–99, 2004.

[21] A. D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *2002 ACM workshop on XML Security*, pages 18–29, 2002.

[22] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.

[23] M. Gudgin et al. *SOAP Version 1.2*, 2003. W3C Recommendation.

[24] J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trusted Global Computing (TGC'05)*, volume 3705 of *LNCS*, pages 116–145. Springer, 2005.

[25] E. Kleiner and A. W. Roscoe. Web services security: A preliminary study using Casper and FDR. In *Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.

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

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

[28] Microsoft Corporation. *Web Services Enhancements (WSE) 2.0*, 2004. At `http://msdn.microsoft.com/webservices/building/wse/default.aspx`.

[29] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[30] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.

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

[32] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*, Mar. 2004. OASIS Standard 200401.

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

[34] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operation Systems Review*, 21(1):8–10, 1987.

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

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

[37] E. Sumii and B. C. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 256–269, 2001.

[38] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 161–172, 2004.

[39] D. Syme. *F#*, 2005. At `http://research.microsoft.com/projects/ilx/fsharp.aspx`.

[40] T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.