Verified Interoperable Implementations of Security Protocols

KARTHIKEYAN BHARGAVAN and
CÉDRIC FOURNET and
ANDREW D. GORDON
Microsoft Research
and
STEPHEN TSE
University of Pennsylvania

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.

Categories and Subject Descriptors: F.3.2 [**Theory of Computation**]: Logics and meanings of programs—Semantics of Programming Languages

General Terms: Security, Languages, Theory, Verification

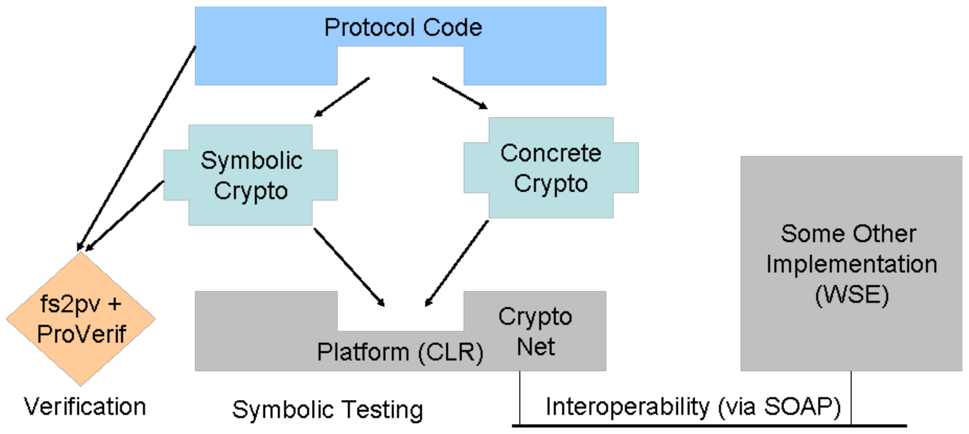
Additional Key Words and Phrases: Functional Programming, Pi Calculus, Web Services, XML Security

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 [Needham and Schroeder 1978]; it has motivated a serious research effort on the formal analysis of cryptographic protocols, starting with Dolev and Yao [1983] 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# [Microsoft Corporation 2005], 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 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 reflect some problematic features, such as for instance ambiguous message formats.)

(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 [Milner 1999; Abadi and Fournet 2001] accepted by ProVerif [Blanchet et al. 2005; Blanchet 2001]. 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 [Allamigeon and Blanchet 2005].

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 [OASIS 2004] standard for securing SOAP [W3C 2003] messages sent to and from XML web services. WS-Security prescribes how to sign and encrypt parts of SOAP messages. Environments such as Apache WSS4J [Apache Software Foundation 2006], IBM WebSphere [IBM Corporation 2006], Microsoft Web Services Enhancements (WSE) [Microsoft Corporation 2004] and Windows Communication Foundation (WCF) [Microsoft Corporation 2006], provide tools and libraries for building web services that are secured via the mechanisms of WS-Security and related specifications. Previous analyses of pi calculus models extracted from WSE by hand have uncovered attacks [Bhargavan et al. 2005; Bhargavan et al. 2004], 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 both WSE and WCF 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 security 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 that 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) A detailed case study of the implementation and verification of a web services security protocol. To the best of our knowledge, the thousand line pi calculus process we verify is the largest model of a cryptographic protocol to be extracted from code. We also provide interoperability results and performance comparisons; as a benchmark, our

implementations pass interoperability tests with at least two production implementations, Microsoft WSE and WCF. 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, states our main theorems, and presents some small examples. Section 5 summarizes our experience in writing and verifying code for web services security protocols; as a case study, it details the implementation and verification of an X.509 mutual authentication protocol. Section 6 concludes. Appendix A introduces an observational equivalence for our pi calculus. Appendix B develops the proof of our safety theorem.

Abridged versions of this work appear in conference [Bhargavan et al. 2006a; Bhargavan et al. 2006b] and summer school [Bhargavan et al. 2007a] proceedings. A companion report [Bhargavan et al. 2007b] provides additional technical details, including omitted proofs. The fs2pv tool and all libraries and example code described in this paper are available online [Microsoft Corporation 2007].

2. A SIMPLE MESSAGE AUTHENTICATION PROTOCOL

We illustrate our method on a very simple, ad hoc protocol example. Next, we describe the structure of the application code implementing client and server roles, and then describe how to derive a concrete implementation, a symbolic prototype, and a formal model suitable for verification. Section 4.4 continues the example, and provides complete listings for its source code and its extracted pi calculus model. 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; it may be informally written as

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

where HMACSHA1 and RSAEncrypt represent keyed cryptographic algorithms and '|' stands for message concatenation. 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 uses the RSA algorithm to encrypt the nonce under B's public key.

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, with type rsa_key. The functions client and server define the two roles of the protocol. 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 other two 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 sk, recomputes the MAC and, if the resulting value differs from the received MAC m, 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 X.509 certificate database. The function S is defined by Crypto; the expression S "A", for example, is an abstract string representing the constant "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 defined in Section 3; they rely in turn 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

```
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"
```

Table I. Two implementations of the Crypto interface

formal verification; it is not called directly from application code and plays no part in the concrete implementation.

Table I shows 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) value; since no inverse function is provided, this abstractly defines a perfect, collision-free one-way function that preserves the secrecy of its arguments. More interestingly, RSA public key encryptions are represented by RsaEncrypt values, decomposed only by a function rsa_decrypt that can verify that the valid decryption key is provided along with the encrypted value.

The symbolic implementation encodes strong but standard assumptions, in the style of Dolev and Yao. Such assumptions are commonly made in automated protocol analyses (in particular in ProVerif [Blanchet 2001]). As usual, our verification results assume that this model is a safe abstraction of concrete cryptography. One may also adapt the symbolic implementation to verify protocols under different assumptions. For instance, one may model hmacsha1 as a function that does not preserve the secrecy of its second argument, or rsa_decrypt as a function that returns a plaintext for any input.

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.

The full code for our symbolic and concrete libraries is available as part of the fs2pv distribution [Microsoft Corporation 2007].

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 nth argument on the command line. As an example, we can execute 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 an encoded message string.

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 [Abadi and Gordon 1999].)

Giving O access to the Crypto and Net interfaces, but not Prins, corresponds to the Dolev and Yao [1983] 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 dynamic access to the secret credentials of some principals, and to allow the dynamic generation of arbitrarily many principal identities, and still prove security goals via ProVerif.) 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.

The message authentication property of our example protocol is expressed as a correspondence [Woo and Lam 1993] 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:

query **ev**:Accept(x) ⇒ **ev**:Send(x)

Syntactic secrecy properties, such as the secrecy of private keys, can also be expressed as correspondences. For all *O*, we want that in every run of *S O*, skB is never obtained by the opponent. We write this property as the correspondence:

```
    query attacker:skB ⇒ ev:Unreachable()
```

where Unreachable() represents an event that can never occur; hence, this query is true if and only if skB may never be obtained by the opponent.

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, proving both message authentication and secrecy of the server's private key:

```
    RESULT ev:Accept(x) ⇒ ev:Send(x) is true
    RESULT attacker:skB[] ⇒ ev:Unreachable() is true.
```

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.

We also verify a weak secrecy property for our example protocol. Via ProVerif, as explained by Blanchet et al. [2005], we can query whether a protocol allows an attacker to guess a weak secret and then verify the guess without contacting the server—if so, the attacker can mount an offline guessing attack. (Online guessing attacks, such that the attacker must interact with the server at every guess, are easily prevented, for instance by limiting the number of retries.)

In the case of our protocol, ProVerif shows the password is protected against offline guessing attacks, by verifying the query:

weaksecret pwdA

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.

We write additional queries to test whether our code is functional, that is, whether some events are ever reachable. For instance, the following query states that the Accept event is never logged:

query Accept(x) $\Rightarrow$ **ev**:Unreachable()

By verifying that this query is *false*, ProVerif shows that this event is, in fact, reachable in our server code. Such falsifiable queries are meant to test our intuitions about the code and the boundaries of the attacker model, and to verify that our security goals are the strongest properties satisfied by the protocool.

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 specify the syntax of F, describe its informal and formal semantics, and define security properties.

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.)

3.1 Syntax and Informal Semantics of F

In the syntax below, ℓ ranges over first-order 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 constructor tuple$n$ creates tuples of length n. We treat each string constant s occurring in a source program as a nullary constructor Ss.

The primitives include the communication functions Pi.send, Pi.recv, and Pi.name described in the previous section. The concurrency operator Pi.fork is a higher-order function; we build Pi.fork into the syntax of F. 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.

Syntax of F:

x, y, z	variable
a, b	name
f	constructor (uncurried)
ℓ	function (curried)
true, false, tuple$n$, Ss	primitive constructors
name, send, recv, log, failwith	primitive functions
$M, N ::=$	value

x	variable
a	name
$f(M_1,\ldots,M_n)$	constructor application
$e ::=$	expression
M	value
$\ell\, M_1\ \ldots\ M_n$	function application
$\mathsf{fork}(\mathbf{fun}() \rightarrow e)$	fork a parallel thread
$\mathbf{match}\ M\ \mathbf{with}(\mid M_i \rightarrow e_i)^{i\in 1..n}$	pattern match
$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	sequential evaluation
$d ::=$	declaration
$\mathbf{type}\ s = (\mid f_i\ \mathbf{of}\ s_{i1} * \ldots * s_{im_i})^{i\in 1..n}$	datatype declaration
$\mathbf{let}\ x\ = e$	value declaration
$\mathbf{let}\ \ell\, x_1 \ldots x_n = e \qquad n > 0$	function declaration
$S ::= d_1 \cdots d_n$	system: list of declarations

We rely on the following syntactic conventions. For any phrase of syntax ϕ, we write $fv(\phi)$ and $fn(\phi)$ for the sets of variables and names occurring free in ϕ. Moreover, we write $n(\phi)$ for the set of all names, whether free or bound, occurring in ϕ. To facilitate the translation from F to the pi calculus, we assume each function ℓ is a pi calculus name, so that, for example, $fn(\ell\, M_1\ \ldots\ M_n) = \{\ell\} \cup fn(M_1) \cup \cdots \cup fn(M_n)$. A phrase of syntax ϕ is *closed* if and only if $fv(\phi) = \varnothing$. We identify phrases of syntax up to consistent renaming of bound variables and names; that is, $\phi = \phi'$ means that ϕ and ϕ' are the same up to such renaming. We write $\phi\{M/x\}$ for the outcome of a capture-avoiding substitution of value M for each free occurrence of x in ϕ. We write $\{M_1/x_1,\ldots,M_n/x_n\}$ as a shorthand for the iterated substitution $\{M_1/x_1\}\ldots\{M_n/x_n\}$. We let σ range over ground substitutions $\{M_1/x_1,\ldots,M_n/x_n\}$ of values for variables, where $fv(M_i) = \varnothing$.

A system S is a space-separated 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). In F, only the constructors f_i and their arities m_i are considered; the type expressions s and s_{ij} are included only for syntactic compatibility with 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. Names model channels, keys, and nonces. Names can only be introduced during evaluation by calling the primitive name. 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 channel M; recv M returns the next value received on channel M; log M logs the event M; failwith M represents a fatal unobservable exception; and fork(**fun**()$\rightarrow e$) evaluates e in parallel. (We need not model exception handling in F as we rely on exceptions only to represent fatal errors that result in termination.) 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.

In addition to the core syntax of F, we recover useful syntax supported by F# as follows. The first three rules allow expressions to be written in places where only values are allowed by the core syntax; these rules only apply when the left-hand side is not within the core syntax.

Derived Expressions:

$f(e_1,\dots,e_n) \triangleq$ **let** $x_1 = e_1$ **in**... **let** $x_n = e_n$ **in** $f(x_1,\dots,x_n)$ $\quad x_i$ fresh
$\ell\ e_1\ \dots\ e_n \triangleq$ **let** $x_1 = e_1$ **in**... **let** $x_n = e_n$ **in** $\ell\ x_1\ \dots\ x_n$ $\quad x_i$ fresh
match e_0 **with**$(\mid M_i \to e_i)^{i\in 1..n} \triangleq$ **let** $x_0 = e_0$ **in match** x_0 **with**$(\mid M_i \to e_i)^{i\in 1..n}$ $\quad x_0$ fresh

$f \triangleq f()$ $\quad$ where constructor f has arity 0
$(e_1,\dots,e_n) \triangleq \text{tuple}n(e_1,\dots,e_n)$ $\quad$ where $n \geq 0$
if e **then** e_1 **else** $e_2 \triangleq$ **match** e **with** $\mid \text{true} \to e_1 \mid \text{false} \to e_2$
$e_1 = e_2 \triangleq$ **match** (e_1,e_2) **with** $\mid (x,x) \to \text{true} \mid (x,y) \to \text{false}$
$e_1;e_2 \triangleq$ **let** $x = e_1$ **in** e_2 $\quad$ x fresh

We also allow the clauses in a **match** to contain **when** clauses; such clauses can be rewritten using conditionals.

3.2 Operational Semantics of F

Next, we formalize the operational semantics of F, in the style introduced by Berry and Boudol [1990] and Milner [1992], and formalize the idea of safety with respect to a query. Let a *configuration*, C, be a multiset of running systems and logged events. We write $C \mid C'$ for the composition of configurations C and C'. To formalize that configurations are multisets, we identify configurations up to a *structural equivalence* relation, $C \equiv C'$, that includes laws of associativity and commutativity for composition. It also includes a law $C \mid \varnothing \equiv C$ to allow deletion of an empty sequence of declarations, $\varnothing$.

Syntax of F Configurations, and Structural Equivalence:

$C ::= S \mid$ **event** $M \mid (C \mid C')$

$C_2 \equiv C_1 \Rightarrow C_1 \equiv C_2$	$C_1 \mid C_2 \equiv C_2 \mid C_1$
$C_1 \equiv C_2, C_2 \equiv C_3 \Rightarrow C_1 \equiv C_3$	$C_1 \mid (C_2 \mid C_3) \equiv (C_1 \mid C_2) \mid C_3$
$C_1 \equiv C_2 \Rightarrow C_1 \mid C \equiv C_2 \mid C$	$C \mid \varnothing \equiv C \qquad C \equiv C$

The following rules define a small-step reduction semantics on configurations.

Reduction Rules: $C \to C'$ where C and C' are closed

$C_1 \to C_2$ if $C_1 \equiv C_1', C_1' \to C_2', C_2' \equiv C_2$
$C_0 \mid d\ \ S \to C_0 \mid S$ $\quad$ if d is a datatype declaration
$C_0 \mid d\ \ S \to C_0 \mid d \mid S$ $\quad$ if d is a function declaration, $S \neq \varnothing$

$C_0 \mid \textbf{let}\ x = M\ \ S \rightarrow C_0 \mid S\{M/x\}$
$C_0 \mid \textbf{let}\ x = \ell\ M_1\ \dots\ M_n\ \ S \rightarrow C_0 \mid \textbf{let}\ x = e\{M_1/x_1,\dots,M_n/x_n\}\ \ S$
$\quad$ if $C_0 \equiv C_1 \mid \textbf{let}\ \ell\ x_1 \dots x_n = e$
$C_0 \mid \textbf{let}\ x = \mathsf{name}\ ()\ \ S \rightarrow C_0 \mid S\{a/x\}$ $\quad$ if $a \notin fn(C_0, S)$
$C_0 \mid \textbf{let}\ x_1 = \mathsf{send}\ M\ N\ \ S_1 \mid \textbf{let}\ x_2 = \mathsf{recv}\ M\ \ S_2 \rightarrow C_0 \mid S_1\{()/x_1\} \mid S_2\{N/x_2\}$
$C_0 \mid \textbf{let}\ x = \mathsf{log}\ M\ \ S \rightarrow C_0 \mid \textbf{event}\ M \mid S\{()/x\}$
$C_0 \mid \textbf{let}\ x = \mathsf{fork}(\textbf{fun}() \rightarrow e)\ \ S \rightarrow C_0 \mid \textbf{let}\ x = e \mid S\{()/x\}$
$C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 1..n}\ \ S \rightarrow C_0 \mid \textbf{let}\ x = e_1\sigma\ \ S$ $\quad$ if $M = M_1\sigma$
$C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 1..n}\ \ S$
$\quad \rightarrow C_0 \mid \textbf{let}\ x = \textbf{match}\ M\ \textbf{with}\ (\mid M_i \rightarrow e_i)^{i \in 2..n}\ \ S$ $\quad$ if $\neg\exists\sigma.\ M = M_1\sigma$
$C_0 \mid \textbf{let}\ x = (\textbf{let}\ y = e_1\ \textbf{in}\ e_2)\ \ S \rightarrow C_0 \mid \textbf{let}\ y = e_1\ \textbf{let}\ x = e_2\ \ S$ $\quad$ $y \notin fv(S)$

The first rule allows configurations to be rearranged up to $C \equiv C'$ when calculating a reduction. The second simply discards a top-level datatype declaration in a system; types have no effect at runtime. The third forks a top-level function declaration d as a separate system consisting just of d; this system is itself inert, but it can be called from other systems running in parallel. (The formation rules for systems, presented later in this section, ensure that functions have distinct names.) The remaining rules apply to a top-level value declaration **let** $x = e$, for some e, running in a context including a configuration C_0, and specify how the expression e evaluates in that context. These rules formalize the description of expression evaluation given earlier in this section.

The only primitive function not to appear in a reduction rule is failwith; applications of the form failwith M are simply stuck (although in F# they raise an exception).

3.3 A Simple Example in F

We consider an example system S_{10} representing transmission of a single encrypted message from an initiator to a responder. The system S_{10} consists of a sequence of ten declarations, which we define as follows.

$$S_{10} \triangleq d_{\mathrm{Ev}}\ d_{\mathrm{Cipher}}\ d_{\mathrm{enc}}\ d_{\mathrm{dec}}\ d_{\mathrm{net}}\ d_{\mathrm{key}}\ d_{\mathrm{init}}\ d_{\mathrm{resp}}\ d_{\mathrm{u1}}\ d_{\mathrm{u2}}$$

The first two declarations are of types: a type of events (as in Section 2) and a type of symmetric-key authenticated encryptions (a much simplified version of the type bytes from Section 2).

$$d_{\mathrm{Ev}} \triangleq \textbf{type}\ \mathrm{Ev} = \mathrm{Send}\ \textbf{of}\ \mathrm{string} \mid \mathrm{Accept}\ \textbf{of}\ \mathrm{string}$$
$$d_{\mathrm{Cipher}} \triangleq \textbf{type}\ \mathrm{Cipher} = \mathrm{Enc}\ \textbf{of}\ \mathrm{string} * \mathrm{name}$$

Next, we declare an encryption function enc and a decryption function dec. (The latter includes a pattern (Enc(p,z),z) containing two occurrences of the same variable. As mentioned above, such patterns are allowed in F but not literally in F#, although we can achieve the same effect in F# by writing (Enc(p,z),z') **when** z=z'.)

$$d_{\mathrm{enc}} \triangleq \textbf{let}\ \mathrm{enc\ x\ y} = \mathrm{Enc(x,y)}$$
$$d_{\mathrm{dec}} \triangleq \textbf{let}\ \mathrm{dec\ x\ y} = \textbf{match}\ \mathrm{(x,y)}\ \textbf{with} \mid \mathrm{(Enc(p,z),z)} \rightarrow \mathrm{p}$$

The next four declarations generate names for a shared network channel (net) intended to be public, and a shared symmetric key (key) intended to be known only to the initiator and responder, and define the initiator and responder role as functions init and resp. The initiator logs a Send event, creates an encryption, and sends it on the network channel. The responder receives a message, decrypts it, and, if the decryption succeeds, logs an Accept event.

$$\begin{aligned}
d_{\mathsf{net}} &\triangleq \textbf{let}\ \mathsf{net} = \mathsf{name}()\\
d_{\mathsf{key}} &\triangleq \textbf{let}\ \mathsf{key} = \mathsf{name}()\\
d_{\mathsf{init}} &\triangleq \textbf{let}\ \mathsf{init}\ \mathsf{x} = \mathsf{log}\ (\mathsf{Send}(\mathsf{x}));\ \textbf{let}\ \mathsf{c} = \mathsf{enc}\ \mathsf{x}\ \mathsf{key}\ \textbf{in}\ \mathsf{send}\ \mathsf{net}\ \mathsf{c}\\
d_{\mathsf{resp}} &\triangleq \textbf{let}\ \mathsf{resp}\ () = \textbf{let}\ \mathsf{m} = \mathsf{recv}\ \mathsf{net}\ \textbf{in}\ \textbf{let}\ \mathsf{x} = \mathsf{dec}\ \mathsf{m}\ \mathsf{key}\ \textbf{in}\ \mathsf{log}\ (\mathsf{Accept}(\mathsf{x}))
\end{aligned}$$

The final two declarations simply fork a single instance of the initiator role and a single instance of the responder role.

$$\begin{aligned}
d_{\mathsf{u1}} &\triangleq \textbf{let}\ \mathsf{u1} = \mathsf{fork}(\textbf{fun}() \rightarrow \mathsf{init}\ \texttt{"msg1"})\\
d_{\mathsf{u2}} &\triangleq \textbf{let}\ \mathsf{u2} = \mathsf{fork}(\textbf{fun}() \rightarrow \mathsf{resp}\ ())
\end{aligned}$$

To illustrate the rules of the formal semantics, we calculate a reduction sequence in which an encryption of "msg1" flows from the initiator to the responder. We eliminate empty systems with the equation $C \mid \varnothing \equiv C$. We begin the calculation with the following steps: the two type declarations are discarded, and the first two function declarations are forked as separate systems.

$$\begin{aligned}
S_{10} &\rightarrow d_{\mathsf{Cipher}}\ d_{\mathsf{enc}}\ d_{\mathsf{dec}}\ d_{\mathsf{net}}\ d_{\mathsf{key}}\ d_{\mathsf{init}}\ d_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\rightarrow d_{\mathsf{enc}}\ d_{\mathsf{dec}}\ d_{\mathsf{net}}\ d_{\mathsf{key}}\ d_{\mathsf{init}}\ d_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}}\ d_{\mathsf{net}}\ d_{\mathsf{key}}\ d_{\mathsf{init}}\ d_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d_{\mathsf{net}}\ d_{\mathsf{key}}\ d_{\mathsf{init}}\ d_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}
\end{aligned}$$

The next part of the computation generates fresh, distinct names n and k and binds them to the variables net and key, respectively. The following abbreviations record the outcome of substituting these names for the variables in init and resp.

$$\begin{aligned}
d^{\mathsf{n}}_{\mathsf{init}} &\triangleq d_{\mathsf{init}}\{\mathsf{n}/\mathsf{net}\} \qquad & d^{\mathsf{n\ k}}_{\mathsf{init}} &\triangleq d^{\mathsf{n}}_{\mathsf{init}}\{\mathsf{k}/\mathsf{key}\}\\
d^{\mathsf{n}}_{\mathsf{resp}} &\triangleq d_{\mathsf{resp}}\{\mathsf{n}/\mathsf{net}\} \qquad & d^{\mathsf{n\ k}}_{\mathsf{resp}} &\triangleq d^{\mathsf{n}}_{\mathsf{resp}}\{\mathsf{k}/\mathsf{key}\}
\end{aligned}$$

We have the following reductions in which n and k are generated, and the initiator and responder functions are forked as separate systems.

$$\begin{aligned}
&d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d_{\mathsf{net}}\ d_{\mathsf{key}}\ d_{\mathsf{init}}\ d_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\quad\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d_{\mathsf{key}}\ d^{\mathsf{n}}_{\mathsf{init}}\ d^{\mathsf{n}}_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\quad\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d^{\mathsf{n\ k}}_{\mathsf{init}}\ d^{\mathsf{n\ k}}_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\quad\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d^{\mathsf{n\ k}}_{\mathsf{init}} \mid d^{\mathsf{n\ k}}_{\mathsf{resp}}\ d_{\mathsf{u1}}\ d_{\mathsf{u2}}\\
&\quad\rightarrow d_{\mathsf{enc}} \mid d_{\mathsf{dec}} \mid d^{\mathsf{n\ k}}_{\mathsf{init}} \mid d^{\mathsf{n\ k}}_{\mathsf{resp}} \mid d_{\mathsf{u1}}\ d_{\mathsf{u2}}
\end{aligned}$$

In the next segment of the computation, we fork instances of the initiator and responder as separate threads. As a shorthand, let $C_0 = d_{\text{enc}} \mid d_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}}$.

$$\begin{aligned}
& d_{\text{enc}} \mid d_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}} \mid d_{\text{u1}}\ d_{\text{u2}} \\
= \;& C_0 \mid \textbf{let } \text{u1} = \text{fork}(\textbf{fun}() \to \text{init } \texttt{"msg1"})\ \textbf{let } \text{u2} = \text{fork}(\textbf{fun}() \to \text{resp } ()) \\
\to \;& C_0 \mid \textbf{let } \text{u1} = \text{init } \texttt{"msg1"} \mid \textbf{let } \text{u2} = \text{fork}(\textbf{fun}() \to \text{resp } ()) \\
\to \;& C_0 \mid \textbf{let } \text{u1} = \text{init } \texttt{"msg1"} \mid \textbf{let } \text{u2} = \text{resp } ()
\end{aligned}$$

The initiator logs a Send event and prepares to send the encrypted message on the channel n. Let $C_1 = C_0 \mid \textbf{let } \text{u2} = \text{resp } ()$.

$$\begin{aligned}
& C_0 \mid \textbf{let } \text{u1} = \text{init } \texttt{"msg1"} \mid \textbf{let } \text{u2} = \text{resp } () \\
\to \;& C_1 \mid \textbf{let } \text{u1} = (\text{log } (\text{Send}(\texttt{"msg1"}));\ \textbf{let } \text{c=enc } \texttt{"msg1"}\text{k } \textbf{in } \text{send n c}) \\
\to \;& C_1 \mid \textbf{let } \text{u3} = \text{log } (\text{Send}(\texttt{"msg1"}))\ \textbf{let } \text{u1=}(\textbf{let } \text{c=enc } \texttt{"msg1"}\text{k } \textbf{in } \text{send n c}) \\
\to \;& C_1 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{u1=}(\textbf{let } \text{c=enc } \texttt{"msg1"}\text{k } \textbf{in } \text{send n c}) \\
\to \;& C_1 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{c=enc } \texttt{"msg1"}\text{k } \textbf{let } \text{u1=send n c} \\
\to \;& C_1 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{c=Enc}(\texttt{"msg1"}\text{,k})\ \textbf{let } \text{u1=send n c} \\
\to \;& C_1 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{u1=send n } (\text{Enc}(\texttt{"msg1"}\text{,k}))
\end{aligned}$$

Next, we consider reductions of the responder **let** u2 = resp (). In fact, it could have reduced in parallel with some of the reductions shown above; we are not here attempting to show all possible interleavings. As a further abbreviation, let $C_2 = C_0 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{u1=send n } (\text{Enc}(\texttt{"msg1"}\text{,k}))$.

$$\begin{aligned}
& C_1 \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{let } \text{u1=send n } (\text{Enc}(\texttt{"msg1"}\text{,k})) \\
= \;& C_2 \mid \textbf{let } \text{u2} = \text{resp } () \\
\to \;& C_2 \mid \textbf{let } \text{u2} = (\textbf{let } \text{m} = \text{recv n } \textbf{in let } \text{x} = \text{dec m k } \textbf{in } \text{log } (\text{Accept}(\text{x}))) \\
\to \;& C_2 \mid \textbf{let } \text{m} = \text{recv n } \textbf{let } \text{u2} = (\textbf{let } \text{x} = \text{dec m k } \textbf{in } \text{log } (\text{Accept}(\text{x})))
\end{aligned}$$

At this point, the encrypted message can pass between the sender and the receiver. We end the calculation with the following steps. Let $C_3 = C_0 \mid \textbf{event } \text{Send}(\texttt{"msg1"})$.

$$\begin{aligned}
& C_2 \mid \textbf{let } \text{m} = \text{recv n } \textbf{let } \text{u2} = (\textbf{let } \text{x} = \text{dec m k } \textbf{in } \text{log } (\text{Accept}(\text{x}))) \\
= \;& C_3 \mid \textbf{let } \text{u2} = (\textbf{let } \text{x} = \text{dec } (\text{Enc}(\texttt{"msg1"}\text{,k}))\text{ k } \textbf{in } \text{log } (\text{Accept}(\text{x}))) \\
\to \;& C_3 \mid \textbf{let } \text{x} = \text{dec } (\text{Enc}(\texttt{"msg1"}\text{,k}))\text{ k } \textbf{let } \text{u2} = \text{log } (\text{Accept}(\text{x}))) \\
\to \;& C_3 \mid \textbf{let } \text{x} = \textbf{match } (\text{Enc}(\texttt{"msg1"}\text{,k}),\text{k})\ \textbf{with} \mid (\text{Enc}(\text{p,z}),\text{z}) \to \text{p} \\
& \quad \textbf{let } \text{u2} = \text{log } (\text{Accept}(\text{x}))) \\
\to \;& C_3 \mid \textbf{let } \text{x} = \texttt{"msg1"}\textbf{let } \text{u2} = \text{log } (\text{Accept}(\text{x}))) \\
\to \;& C_3 \mid \textbf{let } \text{u2} = \text{log } (\text{Accept}(\texttt{"msg1"}))) \\
\to \;& C_3 \mid \textbf{event } \text{Accept}(\texttt{"msg1"})
\end{aligned}$$

In summary, we have calculated the following sequence of reductions.

$$S_{10} \to^{+} d_{\text{enc}} \mid_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}} \mid \textbf{event } \text{Send}(\texttt{"msg1"}) \mid \textbf{event } \text{Accept}(\texttt{"msg1"})$$

3.4 Modelling Adversaries and Robust Safety in F

Formation Judgments for Expressions and Systems. We use system interfaces to control the capabilities of the opponent. An *interface*, I, records 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 arity of constructors and (curried) functions. The arity of a constructor is the width of its tuple of arguments, and may be zero. The arity of a function is the number of its curried arguments, and may not be zero.

Interfaces:

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

For example, let Prim be the following interface, which describes the F primitives, where m is an arbitrary maximum width of tuples, and *Strings* is an arbitrary set of string constants.

true: **ctor** 0, false: **ctor** 0, $(\text{tuple}i\text{: }\mathbf{ctor}\ i)^{i\in 1..m}$, $(\text{S}s\text{: }\mathbf{ctor}\ 0)^{s\in \mathit{Strings}}$
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

As another example, let I_{pub} be the following interface, which enumerates the functions exported by the symbolic libraries together with the application code for the example protocol in Section 2.

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** 2, Crypto.iconcat: **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

The functions Crypto.iS, Crypto.ibase64, Crypto.iutf8, and Crypto.iconcat do not appear in our code, but are inverses of the message formatting functions Crypto.S, Crypto.base64, Crypto.utf8, and Crypto.concat, which do appear. By including these inverses in the interface, we ensure that the attacker can deconstruct messages sent on the wire.

To define when a system exports an interface, we introduce inductively-defined *formation judgments* for expressions and systems. Let $dom(I)$ be the set of variables, constructors, and functions mentioned in I. We write $I \vdash \diamond$ to mean that the interface I mentions no value, constructor, or function twice, that is, there is no split $I = I', I''$ with $dom(I') \cap dom(I'') \neq \varnothing$. We write $I \vdash \mu$ to mean that $I \vdash \diamond$ and moreover μ is a member of I, that is, $I = I', \mu$ for some I'.

The formation 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'. The formation judgment $I \vdash e$ means that all occurrences of variables in e are bound and all occurrences of constructors and functions in e have the correct arity. We define these judgments inductively via the rules in the following table.

In the rule for **match**, we write $fv(M_i)$:**val** as a shorthand for x_1:**val**$,\dots,x_n$:**val** where $\{x_1,\dots,x_n\} = fv(M_i)$.

Formation Rules for F:

$$\frac{I \vdash x{:}\mathbf{val}}{I \vdash x} \qquad \frac{I \vdash f{:}\mathbf{ctor}\ n \quad I \vdash M_i \quad \forall i \in 1..n}{I \vdash f(M_1,\dots,M_n)} \qquad \frac{I \vdash \ell{:}\mathbf{fun}\ n \quad I \vdash M_i \quad \forall i \in 1..n}{I \vdash \ell\ M_1\ \dots\ M_n}$$

$$\frac{I \vdash e}{I \vdash \mathrm{fork}(\mathbf{fun}() \rightarrow e)} \qquad \frac{I \vdash e_1 \quad I,x{:}\mathbf{val} \vdash e_2}{I \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2} \qquad \frac{\begin{array}{c} I \vdash M \quad I,fv(M_i){:}\mathbf{val} \vdash M_i \quad fn(M_i) = \varnothing \\ I,fv(M_i){:}\mathbf{val} \vdash e_i \quad \forall i \in 1..n \end{array}}{I \vdash \mathbf{match}\ M\ \mathbf{with}\ (|\ M_i \rightarrow e_i)^{i \in 1..n}}$$

$$\frac{I \vdash \diamond}{I \vdash \varnothing : \varnothing} \qquad \frac{I_s = (f_i{:}\mathbf{ctor}\ n_i)^{i \in 1..n} \quad I,I_s \vdash S : I'}{I \vdash \mathbf{type}\ s\ =\ (|\ f_i\ \mathbf{of}\ s_{i1} * \cdots * s_{in_i})^{i \in 1..n}\ S : I_s, I'}$$

$$\frac{I \vdash e \quad I,x{:}\mathbf{val} \vdash S : I'}{I \vdash \mathbf{let}\ x\ = e\ S : x{:}\mathbf{val}, I'} \qquad \frac{I,\ell{:}\mathbf{fun}\ n, x_1{:}\mathbf{val},\dots,x_n{:}\mathbf{val} \vdash e \quad I,\ell{:}\mathbf{fun}\ n \vdash S : I'}{I \vdash \mathbf{let}\ \ell\ x_1 \dots x_n = e\ S : \ell{:}\mathbf{fun}\ n, I'}$$

These formation rules are an abstraction of the typing rules of F# for the fragment we consider. They are enforced by the F# compiler during typechecking.

Recall the system $S_{10} = d_{\mathrm{Ev}}\ d_{\mathrm{Cipher}}\ d_{\mathrm{enc}}\ d_{\mathrm{dec}}\ d_{\mathrm{net}}\ d_{\mathrm{key}}\ d_{\mathrm{init}}\ d_{\mathrm{resp}}\ d_{\mathrm{u1}}\ d_{\mathrm{u2}}$ and the interface Prim given earlier. We can derive that Prim $\vdash S_{10} : I_{10}$, where I_{10} is the interface:

Send: **ctor** 1, Accept: **ctor** 1, Enc: **ctor** 2, enc: **fun** 2, dec: **fun** 2,
net: **val**, key: **val**, init: **fun** 1, resp: **fun** 1, u1: **val**, u2: **val**

If $I \vdash S : I'$ then I' is a function of S:

LEMMA 1. *If $I_1 \vdash S : I'_1$ and $I_2 \vdash S : I'_2$ then $I'_1 = I'_2$.*

The formation rules are compositional in the following sense.

LEMMA 2. *If $I_0 \vdash S_1 : I_1$ and $I_2 \vdash S_2 : I_2$ with $I_0, I_1 = I_2, I'_2$ then $I_0 \vdash S_1\ S_2 : I_1, I_2$.*

Event-Based Security Properties of F. We express authentication and other properties in terms of event-based queries, using a syntax borrowed from ProVerif. 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, ..., B_n containing no free names, with $fv(B_i) \subseteq fv(E)$ for each $i \in 1..n$.
Let σ stand for a substitution $\{M_1/x_1,\dots,M_n/x_n\}$.
Let $C \models$ **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n if and only if
whenever $C \equiv$ **event** $E\sigma \mid C'$, we have $C' \equiv$ **event** $B_i\sigma \mid C''$ for some $i \in 1..n$.
Let $C \rightarrow^*_{\equiv} C'$ if and only if either $C \equiv C'$ or $C \rightarrow^* C'$.
Let S be *safe for q* if and only if $C \models q$ whenever $S \rightarrow^*_{\equiv} C$.

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). Our example system S_{10} satisfies this property. For example, let C_{10} be any one of the configurations shown earlier such that $S_{10} \rightarrow^* C_{10}$. We can easily see that $C_{10} \models$ **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x), since an Accept event only occurs in the final configuration, which includes a matching Send event.

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

Consider again our small example S_{10}, its interface I_{10}, and the query $q =$ **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x) given earlier. We already noted that S_{10} is safe for q and that Prim $\vdash S_{10} : I_{10}$, but S_{10} is not robustly safe for q and I_{10}. The interface I_{10} exposes too much to the opponent, and hence does not reflect our intended threat model. For example, the secret key is included in I_{10}, allowing the following opponent O_1 to intercept the encrypted message, and replace it with another.

$$O_1 \stackrel{\triangle}{=} \textbf{let}\ \text{u1} = \text{recv net}\ \textbf{let}\ \text{u2} = \text{send net (enc(}\texttt{"bogus"}\text{,key))}$$

Moreover, the constructor Enc exposed in I_{10} allows the following opponent O_2 to use pattern matching to discover the secret key, and hence to send a bogus message.

$$O_2 \stackrel{\triangle}{=} \textbf{let}\ \text{u} = \textbf{match}\ \text{recv net}\ \textbf{with}\ \text{Enc(m,k)} \rightarrow \text{send net (enc(}\texttt{"bogus"}\text{,k))}$$

The concrete counterpart to this symbolic attack is the ability to extract the encryption key from any ciphertext, a major failure of a cryptosystem. Since this possibility is not normally included in the threat model for protocols, we would not normally export encryption constructors, such as Enc, to the symbolic opponent.

For either O_1 or O_2 we can calculate the following computation, which ends in a configuration that does not satisfy the query q.

$$S_{10}\ O_i \rightarrow^+ d_{\text{enc}} \mid d_{\text{dec}} \mid d^{\text{n k}}_{\text{init}} \mid d^{\text{n k}}_{\text{resp}} \mid \textbf{event}\ \text{Send(}\texttt{"msg1"}\text{)} \mid \textbf{event}\ \text{Accept(}\texttt{"bogus"}\text{)}$$

On the other hand, S_{10} is robustly safe for q and the following interface that reflects our intended threat model. The interface does not expose the secret key to the attacker, and by not exporting the constructor Enc prevents the attacker from extracting keys from ciphertexts. It does allow the attacker to initiate protocol roles, to send and receive network traffic, and to encrypt and decrypt messages.

enc: **fun** 2, dec: **fun** 2, net: **val**, init: **fun** 1, resp: **fun** 1

For the example protocol in Section 2, let S be the system that consists of application code and symbolic libraries. We have that $S :: I_{pub}$, where I_{pub} is the example interface given earlier in this section. 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 Section 4.2. 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 [Blanchet and Podelski 2005]. 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 [Bhargavan, Fournet, Gordon, and Pucella 2004, Bhargavan, Corin, Fournet, and Gordon 2007, Bhargavan, Fournet, and Gordon 2004]. In principle, however, we may benefit from any other verification tool.

4.1 Translation Outline

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 [Milner 1992]. 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 algebra of values—terms built from variables, names, and constructors—as F, so values are unchanged by the translation. Moreover, our pi calculus includes value destructors defined by rewrite rules on the algebra, and whenever possible after inlining, our implementation maps simple functions to destructors. (Our formal translation in Section 4.3 does not cover this optimization.) 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.

4.2 A Pi Calculus

As a basis for describing our translation, we formalize a subset of ProVerif's pi calculus input language. This section describes its syntax and semantics, based in part on a presentation of the applied pi calculus [Abadi and Fournet 2001; Blanchet et al. 2005]. Our implementation relies on all the features of this pi calculus, although some of them are not used by the translation in Section 4.3.

The following table defines the syntax of scripts; each script consists of a set of declarations followed by a process. Values are identical to those in F. Our calculus supports the declaration and application of *destructors*, functions defined by equational rewrites given in **reduc** declarations. The syntax of processes includes a conventional pi calculus core, plus assertion of events, pattern matching, and destructor application.

Processes, Declarations, and Scripts:

M, N	values (as in F)
g	destructor function
$P, Q, R ::=$	process
in$(M,x);P$	input of x from M (x has scope P)
out$(M,N);P$	output of N on M
new $a;P$	make new name a (a has scope P)
$!P$	replication of P
$P \mid Q$	parallel composition
0	inactivity
event M	event M
let $x_1,\dots,x_n$ **suchthat** $M = N$ **in** P **else** Q	match ($x_1, \dots, x_n$ have scope N and P)
let $x = g(M_1,\dots,M_n)$ **in** P **else** Q	destructor application (x has scope P)
$\Delta ::=$	declaration
free a	name a
data f/n	data constructor
private fun f/n	private constructor
reduc $g(M_1,\dots,M_n) = M$	destructor
private reduc $g(M_1,\dots,M_n) = M$	private destructor
$\Delta s ::= \Delta_1. \cdots \Delta_n.$	set of declarations ($n \geq 0$)
$\Sigma ::= \Delta s$ **process** P	script

A top level script $\Sigma = \Delta s$ **process** P defines a process P, which may use names, constructors, and destructors as introduced by the set Δs of declarations. In addition, the declarations indicate whether the implicit attacker, a process deemed to run alongside and interact with P, has access to each constructor and destructor. We assume that Δs contains no two declarations for the same name, constructor, or destructor. We write $\varnothing$ for the empty set of declarations.

A declaration **free** a introduces a name a, that may occur free in P, and also may occur free in the attacker. We assume the process P has no free variables, and that each free name a occurring in P is introduced by a declaration **free** a in Δs.

A declaration **data** f/n introduces a constructor f that has arity n and may occur in the attacker process. Dually, a declaration **private fun** f/n introduces a constructor f that has arity n and may not occur in the attacker process. A declaration **reduc** $g(M_1,\dots,M_n) = M$ introduces a destructor g that has arity n and defining equation $g(M_1,\dots,M_n) = M$, and that may occur in the attacker process. Dually, a declaration **private reduc** $g(M_1,\dots,M_n) = M$ introduces a destructor g that has arity n and defining equation $g(M_1,\dots,M_n) = M$, and that may not occur in the attacker process. For each defining equation, we assume $fv(M) \subseteq fv(M_1,\dots,M_n)$ and $fn(M_1, \dots, M_n, M) = \varnothing$. As in pattern-matching in F, we do not prohibit multiple occurrences of the same variable in the values $M_1, \dots, M_n$. In any occurrence of a constructor or destructor in Σ, we assume that the number of its arguments equals its arity as declared in Σ.

The intended meaning of our process syntax is as follows. An input **in**$(M,x);P$ and an output **out**$(M,N);Q$ attempt to receive and send, respectively, a message on the channel identified by the value M; if M is a channel name, and **in**$(M,x);P$ and **out**$(M,N);Q$ are running in parallel, they may evolve into $P\{M/x\}$ running in parallel with Q. A restriction **new** $a;P$ creates a fresh name a, and acts as P. A replication $!P$ behaves as an unbounded array of replicas of P. A composition $P \mid Q$ behaves as P and Q running in parallel. Inactivity $\mathbf{0}$ does nothing. An event **event** M represents an event, labelled with the value M. A match **let** $x_1,\dots,x_n$ **suchthat** $M = N$ **in** P **else** Q attempts to match the pattern N against the value M: if $M = N\sigma$ for some substitution σ, it behaves as $P\sigma$; otherwise it behaves as Q. A destructor application **let** $x = g(M'_1,\dots,M'_n)$ **in** P **else** Q attempts to rewrite $g(M'_1,\dots,M'_n)$ using the defining equation $g(M_1,\dots,M_n) = M$ of the destructor g: if $M_i\sigma = M'_i$ for each i, for some substitution σ, it behaves as $P\sigma$; otherwise as Q.

We depend on the following abbreviations, including a pattern-matching version of input and a standalone (asynchronous) version of output.

Derived Processes:

$\mathbf{in}(M,N);P \stackrel{\triangle}{=} \mathbf{in}(M,x);\mathbf{let}\ fv(N)\ \mathbf{suchthat}\ x = N\ \mathbf{in}\ P$ $\qquad x$ fresh
$\mathbf{out}(M,N) \stackrel{\triangle}{=} \mathbf{out}(M,N);\mathbf{0}$
$\tau;P \stackrel{\triangle}{=} \mathbf{new}\ a;\mathbf{let}\ x = a\ \mathbf{in}\ P\ \mathbf{else}\ \mathbf{0}$ $\qquad a$ and x fresh
$\mathbf{record}\ M;P \stackrel{\triangle}{=} \tau;(\mathbf{event}\ M \mid P)$
$\mathbf{new}\ a_1,\dots,a_n;P \stackrel{\triangle}{=} \mathbf{new}\ a_1;\dots;\mathbf{new}\ a_n;P$

The semantics of processes is given as a reduction relation $P \rightarrow Q$, itself defined from an auxiliary structural equivalence relation $P \equiv Q$. The rules of reduction make precise the informal semantics of the calculus. The purpose of structural equivalence is to allow a process to be rewritten so that the rules of reduction may apply. The reduction relation implicitly depends on a fixed set of *ambient declarations*, Δs_a, known from the context.

Structural Equivalence of Processes:

$P \equiv P$	$!P \equiv P \mid !P$
$Q \equiv P \Rightarrow P \equiv Q$	$a \notin fn(P) \Rightarrow \mathbf{new}\ a;(P \mid Q) \equiv P \mid \mathbf{new}\ a;Q$
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	$\mathbf{new}\ a;\mathbf{new}\ b;P \equiv \mathbf{new}\ b;\mathbf{new}\ a;P$
$P \mid \mathbf{0} \equiv P$	$\mathbf{new}\ a;\mathbf{0} \equiv \mathbf{0}$
$P \mid Q \equiv Q \mid P$	$P \equiv P' \Rightarrow \mathbf{new}\ a;P \equiv \mathbf{new}\ a;P'$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$P \equiv P' \Rightarrow P \mid R \equiv P' \mid R$

Reduction Semantics for Processes: (relative to ambient declarations Δs_a)

$P \equiv Q, Q \rightarrow Q', Q' \equiv P' \Rightarrow P \rightarrow P'$
$P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q$
$P \rightarrow P' \Rightarrow \mathbf{new}\ a;P \rightarrow \mathbf{new}\ a;P'$
$\mathbf{in}(M,x);P \mid \mathbf{out}(M,N);Q \rightarrow P\{N/x\} \mid Q$

$$\mathbf{let}\ x_1,\dots,x_n\ \mathbf{suchthat}\ M = N\ \mathbf{in}\ P\ \mathbf{else}\ Q \rightarrow \begin{cases} P\sigma & \text{if } M = N\sigma \text{ and } dom(\sigma) = \{x_1,\dots,x_n\} \\ Q & \text{otherwise (when there is no such } \sigma) \end{cases}$$

$$\mathbf{let}\ x = g(M'_1,\dots,M'_n)\ \mathbf{in}\ P\ \mathbf{else}\ Q \rightarrow \begin{cases} P\{M\sigma/x\} & \text{if } M'_i = M_i\sigma \text{ for all } i \in 1..n \\ Q & \text{otherwise (when there is no such } \sigma) \end{cases}$$

where $(g(M_1,\dots,M_n) = M)$ declared in Δs_a

Let $P \rightarrow^*_{\equiv} P'$ if and only if $P \equiv P'$ or $P \rightarrow^* P'$.

We already introduced queries, ranged over by q, in the setting of F in Section 3. We make pi calculus definitions of queries and safety that correspond closely to those for F.

Query Satisfaction and Safety:

$P \models$ **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n if and only if whenever $P \equiv$ **new** as; (**event** $E\sigma \mid P'$), we have $P' \equiv$ **event** $B_i\sigma \mid P''$ for some $i \in 1..n$ and some P''.

A process P is *safe for* q if and only if, for all reductions $P \rightarrow_{\equiv}^{*} P'$, we have $P' \models q$.

Finally, we formalize the idea of an opponent process, and introduce a robust form of safety relative to a query. If we express a security property as robust safety of a suitably constructed script Δs **process** P, we can check the property by running ProVerif.

Opponent Processes and Robust Safety:

A Δs*-opponent* is a process O with no events, such that Δs **process** O is well formed and O contains no constructor or destructor declared **private** in Δs.

A script Δs **process** P is *robustly safe for* q if and only if
for all Δs-opponents O, $P \mid O$ is safe for q.

Robust safety for F quantifies over I-opponents that may include their own datatype declarations. In contrast, ProVerif assumes an arbitrary but fixed set of declarations. In preparation for our proof of Theorem 1, we remark that, without loss of generality, it is safe to consider only I-opponents that systematically use instead some fixed datatype, for instance by declaring a fresh constructor Box:**ctor** 2, introducing a fresh name a_f for each type constructor that does not occur in I, and recursively applying the encoding $f(M_1,\ldots,M_n) = \mathrm{Box}(a_f,(M_1,\ldots,M_n))$ to every value of the I-opponent. For such opponents to be well formed, the parameter m specifying the maximum width of tuples m must be chosen to be greater than the arity of these constructors f. We believe that the encoding does not affect any safety properties for queries that do not use the eliminated constructors.

4.3 A Formally-Correct Translation From F to Pi

We now explain our formal translation from F to ProVerif. (The actual translation that we implemented is similar in spirit but also features various optimizations.) We present the translation of expressions $\mathscr{E}[\![e]\!](x,P)$, the translation of systems $\mathscr{S}[\![S]\!](P)$, and lastly the translation of systems with exported interfaces $[\![S :: I_{pub}]\!]$. (Recall that $S :: I_{pub}$ if and only if $\mathrm{Prim} \vdash S : I_{pub}, I_{priv}$ for some I_{priv}.)

Functions defining script $[\![S :: I_{pub}]\!]$, assuming that $S :: I_{pub}$

Ambient declarations $\Delta s[\![S :: I_{pub}]\!]$	Ambient declarations for the script
Process $\mathscr{E}[\![e]\!](x,P)$	Bind x to value of e then run P
Process $\mathscr{S}[\![d]\!](P)$	Elaborate declaration d then run P
Process $\mathscr{S}[\![S]\!](P)$	Elaborate system S then run P
Process $\mathscr{P}[\![S :: I_{pub}]\!]$	Elaborate system S then export I_{pub}

$[\![S :: I_{pub}]\!] \stackrel{\triangle}{=} \Delta s[\![S :: I_{pub}]\!]$ **process** $\mathscr{P}[\![S :: I_{pub}]\!]$

Next, we present the ambient declarations $\Delta s[\![S :: I_{pub}]\!]$ obtained from S with attacker interface I_{pub}. In making the definition, we assume that $S :: I_{pub}$, so that, by Lemma 1, there is a unique I such that Prim $\vdash S : I$. The constructors in the public interface I_{pub} are public, while the rest of the constructors in I are private. A constructor Box is available to model constructors used by the opponent, as discussed in the final paragraph of Section 4.2. We assume Box does not occur in S.

Ambient declarations $\Delta s[\![S :: I_{pub}]\!]$, assuming that $S :: I_{pub}$:

$$
\begin{array}{lll}
\Delta s[\![S :: I_{pub}]\!] \triangleq & \textbf{free}\ \mathsf{publish}. & \\
& \textbf{data}\ f/n. & \text{for each } f{:}\textbf{ctor}\ n \in I_{pub}, \mathsf{Prim}, \mathsf{Box}{:}\textbf{ctor}\ 2 \\
& \textbf{private fun}\ f/n. & \text{for each } f{:}\textbf{ctor}\ n \in I \setminus I_{pub} \text{ where } \mathsf{Prim} \vdash S : I
\end{array}
$$

The translation of expressions $\mathcal{E}[\![e]\!](x,P) = Q$ takes an F expression e and a ProVerif process P with a free variable x, and returns another ProVerif process Q. The intention is that Q simulates the evaluation of e to a value M, and then runs the process $P\{M/x\}$. The variable x can be considered bound, with scope P; that is, if $x' \notin fv(P)$ then $\mathcal{E}[\![e]\!](x,P) = \mathcal{E}[\![e]\!](x',P\{x'/x\})$. Although we define the translation formally only on the core expressions of F, our tools directly implement the translation on a richer syntax of F# expressions that includes the derived expressions in Section 3.

In this translation, we assume each F function $\ell \notin dom(\mathsf{Prim})$ is a pi calculus name.

Process $\mathcal{E}[\![e]\!](x,P)$:

$\mathcal{E}[\![M]\!](x,P) \triangleq P\{M/x\}$
$\mathcal{E}[\![\ell\ M_1\ \ldots\ M_n]\!](x,P) \triangleq \textbf{new}\ k; (\textbf{out}(\ell,(M_1,\ldots,M_n,k)) \mid \textbf{in}(k,x);P)$
  for $\ell \notin dom(\mathsf{Prim})$, k fresh
$\mathcal{E}[\![\mathsf{name}\ ()]\!](x,P) \triangleq \textbf{new}\ a;P\{a/x\}$ for $a \notin fn(P)$
$\mathcal{E}[\![\mathsf{send}\ M\ N]\!](x,P) \triangleq \textbf{out}(M,N);P\{()/x\}$
$\mathcal{E}[\![\mathsf{recv}\ M]\!](x,P) \triangleq \textbf{in}(M,x);P$
$\mathcal{E}[\![\mathsf{log}\ M]\!](x,P) \triangleq \textbf{record}\ M;P\{()/x\}$
$\mathcal{E}[\![\mathsf{failwith}\ M]\!](x,P) \triangleq \textbf{new}\ k;\textbf{in}(k,x);P$
$\mathcal{E}[\![\mathsf{fork}(\textbf{fun}() \rightarrow e)]\!](x,P) \triangleq \mathcal{E}[\![e]\!](x,\mathbf{0}) \mid P\{()/x\}$
$\mathcal{E}[\![\textbf{match}\ M\ \textbf{with}(\mid M_i \rightarrow e_i)^{i \in 1..n}]\!](x,P) \triangleq$
  $\textbf{let}\ fv(M_1)\ \textbf{suchthat}\ M = M_1\ \textbf{in}\ \mathcal{E}[\![e_1]\!](x,P)\ \textbf{else}$
  $\ldots$
  $\textbf{let}\ fv(M_n)\ \textbf{suchthat}\ M = M_n\ \textbf{in}\ \mathcal{E}[\![e_n]\!](x,P)\ \textbf{else}\ \mathbf{0}$
  where we assume $(fv(M) \cup fv(P)) \cap fv(M_i) = \varnothing$ for each i
$\mathcal{E}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!](y,P) \triangleq \mathcal{E}[\![e_1]\!](x,\mathcal{E}[\![e_2]\!](y,P))$ for $x \notin fv(P)$

If the expression is a value, then it is simply substituted for x in the process P. If the expression is an application of a function $\ell \notin dom(\mathsf{Prim})$, then the arguments $M_1,\ldots,M_n$ and the continuation channel k are sent onto the function channel ℓ—this function channel ℓ is defined below in the translation of program scripts. The primitives name, send, recv, log, fork correspond to the pi calculus primitives restriction, output, input, event, and parallel composition, respectively. We compile the primitive failwith to an inactive pi calculus process. (It is convenient for the sake of the proofs in Appendix B to include the unreachable continuation P in the translation, so that all translated expressions are linear contexts, Lemma 9.) We compile the **match** of F to the **let**... **suchthat**... construct of the pi calculus. We can always satisfy the condition $(fv(M) \cup fv(P)) \cap fv(M_i) = \varnothing$ by renaming the bound variables $fv(M_i)$. Lastly, the sequential evaluation **let** $x = e_1$ **in** e_2 translates to nested compilations of e_1 and e_2.

Next, we define processes $\mathscr{S}[\![d]\!](P)$ and $\mathscr{S}[\![S]\!](P)$ representing sequential evaluation of d and S then P:

Processes $\mathscr{S}[\![d]\!](P)$ and $\mathscr{S}[\![S]\!](P)$:

$$\mathscr{S}[\![\mathbf{type}\ s\ =\ (\mid f_i\ \mathbf{of}\ s_{i1} * \cdots * s_{in_i})^{i\in 1..n}]\!](P) \triangleq P$$
$$\mathscr{S}[\![\mathbf{let}\ x = e]\!](P) \triangleq \mathscr{E}[\![e]\!](x,P)$$
$$\mathscr{S}[\![\mathbf{let}\ \ell\ x_1 \ldots x_n = e]\!](P) \triangleq (!\mathbf{in}(\ell,(x_1,\ldots,x_n,k)); \mathscr{E}[\![e]\!](x, \mathbf{out}(k,x))) \mid P \qquad k \text{ fresh}$$

$$\mathscr{S}[\![\varnothing]\!](P) \triangleq P$$
$$\mathscr{S}[\![d\ S]\!](P) \triangleq \mathscr{S}[\![d]\!](\mathscr{S}[\![S]\!](P))$$

A type declaration is discarded, while a value declaration uses the translation of expressions. For a function declaration for ℓ, the translation listens on the channel ℓ, and outputs the result of the computation to the continuation parameter k.

The translation of a system $\mathscr{S}[\![S]\!](P)$ is an extension of the translation of declaration $\mathscr{S}[\![d]\!](P)$ by folding the system as an ordered list of declarations.

Finally, we present the top-level process $\mathscr{P}[\![S :: I_{pub}]\!]$ representing execution of S with public interface I_{pub}. We use restrictions to hide the names of user-defined functions. We publish all public values and forwarded functions onto a fresh publish channel, making them accessible to the attacker. Let $a \rightarrow b$ be a forwarder process, defined as $!\mathbf{in}(a,z).\mathbf{out}(b,z)$, that receives messages on channel a and sends them on channel b. We use forwarders to publish aliases for function names, as opposed to the names themselves, to enable the pi calculus attacker to call but not to re-define functions implemented as processes by receiving on their names.

Process $\mathscr{P}[\![S :: I_{pub}]\!]$, assuming that $S :: I_{pub}$:

$$\begin{array}{lll} \mathscr{P}[\![S :: I_{pub}]\!] \triangleq & \mathbf{new}\ \ell; & \\ & & \text{for each } \ell \in dom(I) \text{ where Prim} \vdash S : I \\ & \mathbf{new}\ \ell_{pub}; \ell_{pub} \rightarrow \ell \mid & \\ & & \text{for each } \ell \in dom(I_{pub}) \text{ and } \ell_{pub} \text{ fresh} \\ & \mathscr{S}[\![S]\!](\mathbf{out}(\text{publish}, xs)) & \\ & & \text{where the tuple } xs \text{ collects all } x \in dom(I_{pub}) \\ & & \text{and all } \ell_{pub} \text{ with } \ell \in dom(I_{pub}) \end{array}$$

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}.

```
fsc.exe -o tiny.exe –define fs -r System.Security.dll pi.fsi pi.fs crypto.fsi crypto.fs net.fsi net.fs tiny.fs
./tiny.exe

  Sending FADCIzZhW3XmgUABgRJj1KjnWyDvEoAAezcg5gaDY5lsP0CWOCoFR9a0...

fsc.exe -o tiny-a.exe –define fs ../lib/pi.fsi ../lib/pi.fs crypto.fsi crypto-a.fs net.fsi net-a.fs tiny.fs
./tiny-a.exe

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

fs2pv.exe -o tiny.pv crypto.fsi crypto-a.fs net.fsi net-a.fs tiny.fsi tiny.fs
analyzer.exe -in pi tiny.pv | grep RESULT

  RESULT ev:Accept(x) ⇒ ev:Send(x) is true.
  RESULT attacker:skB[] ⇒ ev:Unreachable() is true.
  RESULT equivalence proof succeeded (bad not derivable).
  RESULT ev:Send(x) ⇒ ev:Unreachable() is false.
  RESULT ev:Accept(x) ⇒ ev:Unreachable() is false.
```

Table II. Building and executing three versions of Tiny.

The proof of Theorem 1 appears in Appendix B; 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}.

Our tools rely on classic program transformations in F, applied on systems before translation to the pi calculus. For instance, we use code inlining for function applications (replacing the expression $\ell\, M_1\ \dots\ M_n$ with $e\{M_1/x_1,\dots,M_n/x_n\}$ within the scope of a declaration **let** $\ell\, x_1 \dots x_n = e$) and dead code elimination for function declarations (eliminating a function declaration if the function is never applied). We easily check that these transformations preserve all well-formed conditions and do not affect any robust safety properties. We omit their standard formal treatment.

4.4 Translation for the Example of Section 2

We provide the complete source code and translated pi calculus code for the example of Section 2. To improve the readability of pi calculus code, we use customized versions of our libraries, obtained by erasing code that is unnecessary in this example, and replacing all calls to the Prins library with generation of fresh passwords and keys. (Our tests also include variants of this example linked with unmodified libraries.)

Table II gives the command lines used to build and execute three versions of the example: tiny.exe is compiled using the F# compiler (fsc.exe) with the concrete libraries; tiny-a.exe is similarly compiled with the symbolic libraries; tiny.pv is compiled using our model extractor (fs2pv.exe) and verified using the ProVerif tool (analyzer.exe). The flag '–define

```
module Crypto
type str
type bytes
type rsa_key
val S: string → str
val iS: str → string
val base64: bytes → str
val ibase64: str → bytes
val utf8: str → bytes
val iutf8: bytes → str
val concat: bytes → bytes → bytes
val concat3: bytes → bytes → bytes → bytes
val iconcat: bytes → bytes * bytes
val iconcat3: bytes → bytes * bytes * bytes
val mkNonce: unit → bytes
val mkPassword: unit → str
val hmacsha1: bytes → bytes → bytes
val rsa_keygen: unit → rsa_key
val rsa_pub: rsa_key → rsa_key
val rsa_encrypt: rsa_key → bytes → bytes
val rsa_decrypt: rsa_key → bytes → bytes
val aes_encrypt: bytes → bytes → bytes
val aes_decrypt: bytes → bytes → bytes
val mkKey: unit → bytes
```

```
module Pi
val fork: (unit → unit) → unit
type name
type 'a chan
val name: string → name
val chan: unit → 'a chan
val send: 'a chan → 'a → unit
val recv: 'a chan → 'a
type 'a trace
val trace: unit → 'a trace
val log: 'a trace → 'a → unit
```

```
module Net
val accept: Crypto.str → Crypto.str
val send: Crypto.str → Crypto.str → unit
```

```
module Tiny
val pkB: Crypto.rsa_key
val client: Crypto.str → unit
val server: unit → unit
```

Table III. Interfaces for Crypto, Pi, Net, and Tiny (files crypto.fsi, pi.fsi, net.fsi, and tiny.fsi).

fs' includes pretty-printing code, otherwise omitted for model extraction.

Table III lists all interfaces used by these command lines. Table IV lists the F# implementation of Tiny, which includes the code fragments explained in Section 2. (As a minor difference, the primitive log takes here an additional parameter tr, discarded during model extraction.) Table V lists the resulting ProVerif script produced by fs2pv.exe.

4.5 Verification Results for Simple Protocols

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

Tables VI and VII summarize our results for these protocols. For each protocol, Table VI 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 VII 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, such as the functionality queries in Section 2, are designed to test the boundaries of the attacker model and are meant to fail during verification. Finally, the table

```
open Pi
open Crypto

let marshall (m,en,text) = base64(concat3 m en (utf8 text))
let unmarshall v =
  let m,en,text = iconcat3 (ibase64 v) in (m,en,iutf8 text)

let mac nonce password text =
  hmacsha1 nonce (concat (utf8 password) (utf8 text))
let make text pk password =
  let nonce = mkNonce() in
  (mac nonce password text, rsa_encrypt pk nonce, text)
let verify (m,en,text) sk password =
  let nonce = rsa_decrypt sk en in
  if m = mac nonce password text then () else failwith "bad MAC"

let pwdA = mkPassword()
let skB = rsa_keygen ()
let pkB = rsa_pub skB

type events = Send of str | Accept of str | Unreachable // security events
let tr = trace ()

let address = S "http://localhost:8080/pwdmac"
let client text =
  log tr (Send(text));
  Net.send address (marshall (make text pkB pwdA))
let server () =
  let m,en,text = unmarshall (Net.accept address) in
  verify (m,en,text) skB pwdA; log tr (Accept(text))
```

Table IV. F# implementation of Tiny (file tiny.fs).

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.

In the following, we describe the first three of these protocols. The next section describes larger protocols based on web services security.

Password-based authentication. 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 VI and VII) 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. (Bhargavan et al. [2007c] details this extended attacker model.)

Otway-Rees. As a benchmark, we wrote a program for the four message Otway-Rees key establishment protocol [Otway and Rees 1987], with two additional messages after key establishment to probe the secrecy of message payloads encrypted with this key. To

```
free publish.
data True/0.
data False/0.
data Box/2.
reduc equal(x,x) = True().
(∗ deleted unused F# primitives ∗)

data SS/0. (∗ string constants ∗)
data Shttplocalhost8080pwdmacS/0.
data SnonceS/0.
data SrsausecretS/0.

private fun Literal/1.
private fun Base64/1.
private fun C/2.
private fun Utf8/1.
private fun Name/1.
private fun HmacSha1/2.
private fun RsaKey/1.
private fun RsaEncrypt/2.
private fun SK/1.
private fun PK/1.

reduc S(s) = Literal(s).
reduc iS(Literal(s)) = s.
reduc base64(b) = Base64(b).
reduc ibase64(Base64(s)) = s.
reduc concat(x,y) = C(x,y).
reduc iconcat(C(x,y)) = (x,y).
reduc concat3(r1,r3,r4) = C(r1,C(r3,r4)).
reduc iconcat3(C(r5,C(r7,r8))) = (r5,r7,r8).
reduc utf8(x) = Utf8(x).
reduc iutf8(Utf8(s)) = s.

free mkNoncePUB.
reduc hmacsha1(key,text) = HmacSha1(key,text).
reduc rsaupub(SK(s)) = PK(s).
free rsaukeygenPUB.
reduc rsauencrypt(key,text) =
  RsaEncrypt(key,text).
reduc rsaudecrypt
  (SK(keyP),RsaEncrypt(PK(keyP),text)) = text.

private fun Send/1.
private fun Accept/1.
private fun Unreachable/0.

free sendPUB. free acceptPUB.
free clientPUB. free serverPUB.

query ev:Ev(Accept(x)) ⇒ ev:Ev(Send(x)).
query ev:Ev(Accept(x)) ⇒ ev:Ev(Unreachable()).
query ev:Ev(Send(x)) ⇒ ev:Ev(Unreachable()).

process
((!in(mkNoncePUB, (W9,K7));
  new T30; out(K7, Name(T30)))
|(!in(rsaukeygenPUB, (W7,K5));
  new T23; out(K5, SK(Name(T23))))
|(new httpchan; new respchan;
  ((!in(sendPUB, (addr16,msg17,K4));
    out(httpchan, msg17); out(K4, ()))
  |(!in(acceptPUB, (address15,K3));
    in(httpchan, T15); out(K3, T15))
  |(new T14;
    let pwdA = base64(Name(T14)) in
    new T12; let pkB = rsaupub(SK(Name(T12))) in
    let address = S(Shttplocalhost8080pwdmacS()) in
    ((!in(clientPUB, (text5,K2));
      event Ev(Send(text5));
      new T10; let T9 =
          (mac(Name(T10),pwdA,text5),
          rsauencrypt(pkB,Name(T10)),text5) in
      let T8 = marshall(T9) in
      out(httpchan, T8); out(K2, ()))
    |(!in(serverPUB, (W1,K1));
      in(httpchan, T4);
      let T3 = unmarshall(T4) in let (m1,en2,text3) = T3 in
      let W2 = verify((m1,en2,text3),SK(Name(T12)),pwdA) in
      event Ev(Accept(text3)); out(K1, ()))
    |(out(publish, pkB)))))))

private reduc marshall((r10,r11,r13)) = Base64(C(r10,C(r11,Utf8(r13)))).
private reduc unmarshall(Base64(C(r14,C(r15,Utf8(r18))))) = (r14,r15,r18).
private reduc mac(r19,r23,r24) = HmacSha1(r19,C(Utf8(r23),Utf8(r24))).
private reduc verify((HmacSha1(r27,C(Utf8(r28),Utf8(r29))),RsaEncrypt(PK(r25),r27),r29),SK(r25),r28) = ().
```

Table V. ProVerif script for Tiny (file tiny.pv, up to reformatting and renaming).

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-X.509 mutual auth	149	2	6206; 3187	486; 542
WS X.509 mutual auth	117	2	4533; 4836	304; 531

Table VI. 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	1min50s
WS Password-based signing	5	no	msg, sender	yes	456	5.3s
WS X.509 signing	5	no	msg, sender	yes	460	2.6s
WS Password-X.509 mutual auth	15	no	session	yes	503	44min
WS X.509 mutual auth	18	msg	session	yes	612	51min

Table VII. Verification Results

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.

5. VERIFYING WEB SERVICES SECURITY PROTOCOLS

As a larger, more challenging case study than the example protocols of Section 4.5, we implemented and verified several web services security protocols.

Web services are applications that exchange XML messages conforming to the SOAP standard [W3C 2003]. To secure these exchanges, messages may include a security header, defined in the WS-Security standard [OASIS 2004], 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 [Gordon and Pucella 2002; Bhargavan et al. 2005; Bhargavan et al. 2007c; Kleiner and Roscoe 2004; 2005] and uncovered attacks [Bhargavan et al. 2005; Bhargavan et al. 2004]. 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. We first detail our methodology on an example web services security protocol implementation; we then present our verification results for other such protocol implementations. These implementations rely on a web services security library; we end the section with a description of the design of this library.

5.1 X.509 Mutual Authentication

As our main case study, we consider a mutual authentication protocol based on X.509 public key certificates. Both WSE and WCF implement this protocol as part of their sample code.

We begin with an informal narration of the protocol, then provide a complete implementation in F#. The code is quite short, as it mostly relies on our WS-Security libraries. We describe executions of the protocol, both symbolically (to produce readable message traces) and concretely (to evaluate its performance). We also report on interoperability testing with the WSE and WCF implementations. Finally, we present verification results for this implementation.

Protocol Narration. The protocol has two roles, a client and a server. Every session of the protocol involves a principal A acting as client and a principal B acting as server. Each principal is associated with an RSA key-pair, consisting of a private key and a corresponding public key; A's key-pair is written (sk_A, pk_A), and B's key-pair is written (sk_B, pk_B). We assume that the principals have already exchanged their public key certificates. Hence, the principals can identify one another using their public keys.

The goal of the protocol is to exchange two XML messages: a request and a response, such that both the client and server can authenticate the two-message session and keep the messages secret, even in the presence of an active attacker. To accomplish this goal, we rely on XML digital signatures and XML Encryption. The abstract message sequence of the protocol can be written as follows (where | denotes concatenation):

$$
\begin{array}{rl}
A \rightarrow B: & TS \mid \\
& \text{RSA-SHA1}\{sk_A\}[\mathit{request} \mid TS] \mid \\
& \text{RSA-Encrypt}\{pk_B\}[\mathit{symkey}_1] \mid \\
& \text{AES-Encrypt}\{\mathit{symkey}_1\}[\mathit{request}] \\
B \rightarrow A: & \text{RSA-SHA1}\{sk_B\}[\mathit{response} \mid \text{RSA-SHA1}\{sk_A\}[\mathit{request} \mid TS]] \mid \\
& \text{RSA-Encrypt}\{pk_A\}[\mathit{symkey}_2] \mid \\
& \text{AES-Encrypt}\{\mathit{symkey}_2\}[\mathit{response}]
\end{array}
$$

The client acting for principal A sends a message *request* at time TS to the server acting for B. To support message authentication, the client jointly signs *request* and TS using the signature algorithm RSA-SHA1 keyed with A's private key sk_A. To protect the secrecy of the message, the client uses AES-Encrypt to encrypt it under a fresh symmetric key symkey_1. The symmetric key is in turn encrypted using RSA-Encrypt under pk_B. (This standard, two-step encryption is motivated by the relative costs of symmetric and asymmetric encryptions for large messages.) In addition, the protocol assumes that RSA-SHA1 preserves the secrecy of the message.

The server repeatedly processes request messages. After accepting a request, the server returns a *response* to the client. Like the request, the response is signed (using sk_B) then encrypted (using a fresh symkey_2 encrypted under pk_A). To correlate requests and responses, the server jointly signs the response and the signature value of the request. (Otherwise, since clients and servers may run several sessions in parallel, an attacker may confuse the client by swapping two responses.) This correlation mechanism is called *signature confirmation*.

The security goals of the protocol are as follows:

—*Request Authentication:* B accepts a *request* from A with timestamp TS only if A sent such a *request* with timestamp *TS*.

—*Response Authentication and Correlation: A* accepts a *response* to its *request* only if *B* sent *response* on receiving *A*'s *request*.

—*Secrecy:* the message payloads *request* and *response* are kept secret from all principals other than *A* and *B*.

Note that although these goals require that the timestamps on both messages be authenticated, they do not rely on the actual values of the timestamps. Our formal model treats timestamps as opaque strings and does not, for instance, compare them as numbers.

Implementation. Our protocol implementation is listed in Table VIII. The module consists of four functions: mkEnvelope and isEnvelope generate and check the protocol messages, while client and server implement the two protocol roles.

To parse and generate standards-compliant SOAP envelopes, and to sign and encrypt XML elements, we rely on functions of the web services security library. As an example, consider the mkEnvelope function. Depending on its arguments, mkEnvelope constructs either a request message or a response message. To construct a request, it takes a message body containing the *request*, the X.509 entry snd for the sending principal *A*, the X.509 certificate rcvcert for the receiving principal *B*, and an empty list corr. (When constructing a response, snd is the X.509 entry for *B*, rcvcert is the X.509 certificate for *A*, and corr contains the signature value of the request.) The code for mkEnvelope successively calls the following library functions, defined in modules wssecurity.fs and soap.fs:

—mkTimestamp and genTimestamp create a new timestamp and serialize it to XML;

—mkX509Signature generates the XML digital signature for the message;

—mkX509Encdatakey generates the two encrypted components;

—mkX509SecurityHeader generates the security header;

—genEncryptedEnvelope generates the whole SOAP envelope for the message.

Finally, the function returns the envelope (for sending) paired with its signature value (kept for correlating the response).

Unlike mkEnvelope and isEnvelope, the client and server functions are part of the attacker interface; both these functions are included in the interface X509MutualAuth.fsi for the protocol module X509MutualAuth.fs:

```
val client: str → str → str → str → unit
val server: str → str → str → unit
```

Hence, an attacker can call these functions to initiate sessions and instantiate roles.

The four arguments to client are the name of the client and server principals (clPrin, srvPrin), and the HTTP URI and SOAP action (servUri, servAction) that identify the server location. The client first calls the request function from the service.fs module (described in the next subsection) to compute the XML request payload (req). It then calls the logsecret function to log this payload as a secret; if the attacker ever obtains a value v logged as secret, it can call a checksecret function to trigger an event NotSecret(v), indicating that v is no longer secret. We use this event to specify our secrecy goals; it is more flexible than the usual attacker:v specification in ProVerif, since it does not require the secret value v to be defined at the top level of the program. It then instantiates both principals; it gets the X.509 entry (cl) for clPrin from a private database; the entry consists of an

```
(∗ Opening Library Modules ∗)
open Data (∗ Standard datatypes: str, bytes, item ∗)
open Events (∗ Protocol Events ∗)

(∗ Constructing Messages ∗)
let mkEnvelope (body:item) (snd:Prins.principalX) (rcvcert:bytes)
                    (corr:item list) : item∗bytes =
  let ts = Wssecurity.genTimestamp(Wssecurity.mkTimestamp()) in
  let (dsig,sv) = Wssecurity.mkX509Signature snd (body::ts::corr) in
  let (ed,ek) = Wssecurity.mkX509Encdatakey rcvcert body in
  let sec = Wssecurity.mkX509SecurityHeader (Prins.cert snd) ek ts dsig in
  let envXml = Soap.genEncryptedEnvelope [sec] ed in
  (envXml,sv)

(∗ Checking Messages ∗)
let isEnvelope (envXml:item) (sndcert:bytes) (rcv:Prins.principalX)
                    (corr:item list) : item∗bytes =
  let ([sec],ed) = Soap.parseEncryptedEnvelope envXml in
  let (ts,ek,dsig) = Wssecurity.isX509SecurityHeader sec in
  let body = Wssecurity.isX509Encdatakey rcv ek ed in
  let sv = Wssecurity.isX509Signature dsig sndcert (body::ts::corr) in
  (body,sv)

(∗ Client Role ∗)
let client (clPrin: str) (srvPrin:str) (servUri:str) (servAction:str) =
  let req = Service.request () in
  logsecret req [srvPrin];
  let cl = Prins.getX509 clPrin in
  let srvCert = Prins.getX509Cert srvPrin in
  let (reqXml,sv) = mkEnvelope req cl srvCert [] in
  log (ClientSend(clPrin,srvPrin,req));
  let respXml = Net.request servUri servAction reqXml in
  let sc = Wssecurity.genSigConf sv in
  let (resp,_) = isEnvelope respXml srvCert cl [sc] in
  let _ = Service.isResponse resp in
  log (ClientCorr(clPrin,srvPrin,req,resp))

(∗ Server Role ∗)
let server (clPrin:str) (srvPrin:str) (servUri:str) =
  let clCert = Prins.getX509Cert clPrin in
  let srv = Prins.getX509 srvPrin in
  let reqXml = Net.accept servUri in
  let (req,sv) = isEnvelope reqXml clCert srv [] in
  let _ = Service.isRequest req in
  log (ServerRecv(clPrin,srvPrin,req));
  let resp = Service.response req in
  logsecret resp [clPrin];
  let sc = Wssecurity.genSigConf sv in
  let (respXml,_) = mkEnvelope resp srv clCert [sc] in
  log (ServerCorr(clPrin,srvPrin,req,resp));
  Net.respond respXml
```

Table VIII. Protocol Module: X509MutualAuth.fs

X.509 certificate and its associated private key; it then extracts the certificate (srvCert) for the server principal srvPrin. Next, it prepares the request message (reqXml), using mkEnvelope, logs an event ClientSend(clPrin,srvPrin,req) to indicate that it is sending the first message, and makes an HTTP request to the server, using Net.request. The client remembers the signature value (sv) of the request for correlating the response, and uses it to construct the expected signature confirmation element (sc). When the client receives a response (respXml), it uses isEnvelope to check that the response message is valid and that it includes the signature confirmation (sc). It calls isResponse to check that the body of the response message is a valid application-level response, and then logs the event ClientCorr(clPrin,srvPrin,req,resp) indicating that a valid response has been received and correlated with the request.

The server proceeds symmetrically: it uses the client certificate and the server X.509 entry to check requests and issue responses. After accepting a request, the server logs an event ServerRecv(clPrin,srvPrin,req); it then calls Service.response(req) to compute the response resp, and logs the event ServerCorr(clPrin,srvPrin,req,resp) before issuing the response.

Protocol Execution. To run the protocol, we write a main module X509Main.fs, listed below. (This module is not used for verification; formally, it is just a simple instance of the attackers considered in our theorems.)

```
let clntPrin = S "client.com"
let srvPrin = S "localhost"
do Prins.genX509 clntPrin
do Prins.genX509 srvPrin
do match Sys.argv.(1) with
  | "client" → client clntPrin srvPrin Service.uri Service.action;
  | "server" → server clntPrin srvPrin Service.uri;
  | "local" → Pi.fork (fun () → server clntPrin srvPrin Service.uri);
               client clntPrin srvPrin Service.uri Service.action
```

This module first instantiates the client and server principals (identified by their X.509 common names “client.com” and “localhost”), and then runs either the client, or the server, or both, depending on the command-line argument. The X509Main.fs module is used only for executing the protocol; they are not used for verification.

We also write a module service.fs to encode an exemplary addition service. The module consists of two functions: Service.request extracts two numbers from the command line and returns them in a request body; Service.response computes the sum of the two numbers in a request and returns it in a response body.

For verification, we write a dual, symbolic implementation of this module that generalizes the two functions by allowing the attacker to choose some payloads: the symbolic version of Service.request (Service.response) returns a request (response) body that it either received from the attacker or it computed from a secret value. Hence, our security goals require request and response authentication even when the attacker is allowed to choose arbitrary payloads, and require secrecy of the secret payloads.

Symbolic Messages. To run the protocol symbolically, we compile the X509MutualAuth.fs and X509Main.fs modules with the web services library and the symbolic version of the modules crypto.fs, net.fs, prins.fs, and service.fs to generate an executable run.exe. We can

```
<Envelope>
  <Header>
    <Security>
ts1 = <Timestamp Id='Timestamp'>
        <Created>Now1</>
        <Expires>PlusOneMinute</></>
      <BinarySecurityToken EncodingType='Base64Binary' ValueType='X509v3'
                           Id='X509Token-client.com'>
       X509(Root,client.com,sha1RSA,PK(rsa_secret1))</>
      <EncryptedKey Id='Encrkey'>
        <EncryptionMethod Algorithm='rsa-1_5' />
        <KeyInfo>
          <SecurityTokenReference>
            <X509Data>
              <X509IssuerSerial>
                <X509IssuerName>Root</>
                <X509SerialNumber>guid4</></></></></></>
        <CipherData>
          <CipherValue>RSA−Enc{PK(rsa_secret3)}[key5]</></>
        <ReferenceList>
          <DataReference URI='guid6' /></></>
      <Signature>
si1 = <SignedInfo>
          <CanonicalizationMethod Algorithm='xml-exc-c14n#' />
          <SignatureMethod Algorithm='rsa-sha1' />
          <Reference URI='Body'>
            <Transforms><Transform Algorithm='xml-exc-c14n#' /></>
            <DigestMethod Algorithm='sha1' />
            <DigestValue>SHA1(
              <Body Id='Body'>req</>)</></>
          <Reference URI='Timestamp'>
            <Transforms><Transform Algorithm='xml-exc-c14n#' /></>
            <DigestMethod Algorithm='sha1' />
            <DigestValue>SHA1(ts)</></></>
        <SignatureValue>
sv1 = RSA−SHA1{rsa_secret1}[si]
        </>
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI='X509Token-client.com' ValueType='X509v3' />
          </></></></></></>
  <Body Id='Body'>
    <EncryptedData Id='guid6' Type='Content'>
      <EncryptionMethod Algorithm='aes128-cbc' />
      <CipherData>
        <CipherValue>AES−Enc{key5}[
req = <Add>
            <n1>100</>
            <n2>15.99</></></>]</></></></></>
```

Table IX. Symbolic Request Message

```
<Envelope>
  <Header>
    <Security>
ts2 = <Timestamp Id='Timestamp'>
        <Created>Now2</>
        <Expires>PlusOneMinute</></>
      <BinarySecurityToken EncodingType='Base64Binary' ValueType='X509v3'
                           Id='X509Token-localhost'>
        X509(Root,localhost,sha1RSA,PK(rsa_secret3))</>
      <EncryptedKey Id='Encrkey'>
        <EncryptionMethod Algorithm='rsa-1_5' />
        <KeyInfo>
          <SecurityTokenReference>
            <X509Data>
              <X509IssuerSerial>
                <X509IssuerName>Root</>
                <X509SerialNumber>guid2</></></></></></>
        <CipherData><CipherValue>RSA-Enc{PK(rsa_secret1)}[key7]</></>
        <ReferenceList>
          <DataReference URI='guid8' /></></>
      <Signature>
si2 = <SignedInfo>
          <CanonicalizationMethod Algorithm='xml-exc-c14n#' />
          <SignatureMethod Algorithm='rsa-sha1' />
          <Reference URI='Body'>
            <Transforms><Transform Algorithm='xml-exc-c14n#' /></>
            <DigestMethod Algorithm='sha1' />
            <DigestValue>SHA1(
              <Body Id='Body'>resp</>)</></>
          <Reference URI='Timestamp'>
            <Transforms><Transform Algorithm='xml-exc-c14n#' /></>
            <DigestMethod Algorithm='sha1' />
            <DigestValue>SHA1(ts)</></>
          <Reference URI='SigConf'>
            <Transforms><Transform Algorithm='xml-exc-c14n#' /></>
            <DigestMethod Algorithm='sha1' />
            <DigestValue>SHA1(
               <SignatureConfirmation Value='sv1' Id='SigConf' />
               )</></></>
        <SignatureValue>
sv2 = RSA-SHA1{rsa_secret3}[si2]
        </>
        <KeyInfo>
          <SecurityTokenReference>
            <Reference URI='X509Token-localhost' ValueType='X509v3' />
          </></></></></>
  <Body Id='Body'>
    <EncryptedData Id='guid8' Type='Content'>
      <EncryptionMethod Algorithm='aes128-cbc' />
      <CipherData><CipherValue>AES-Enc{key7}[
resp = <AddResponse>
            <n>115.99</></></>]</></></></></>
```

Table X. Symbolic Response Message

then execute the command run local 100 15.99, for example. Our implementation pretty-prints the communicated messages, using an abbreviated XML-like format with embedded symbolic expressions. Table IX shows the first message of the protocol; Table X shows the second message. The first message has 304 symbols while the second has 531.

In Table IX, ts1 is the symbolic timestamp and req is the serialized request. The timestamp is modeled as an XML element containing a fresh globally unique string (Now1) representing its creation time and a constant expiration time. The message has a security header that contains ts1, an encrypted symmetric key key1, and an XML digital signature for req and ts1. The key key1 is encrypted using the public key certificate for the server; in this message the certificate is issued by Root and has a serial number guid4 and public key PK(rsa_secret3). The XML signature value sv1 is computed as the RSA−SHA1 signature of the element si, which in turn contains the SHA1 hashes of req and ts1. Finally, the body of the message is the request req encrypted under the symmetric key key5.

The message in Table X can be read similarly; the main difference is that the signature includes a new <SignatureConfirmation> element containing the signature value sv1 from the first message.

Concrete runs and Performance. To run the protocol concretely, we compile X509MutualAuth.fs, X509Main.fs, and the web services library with the concrete versions of crypto.fs, net.fs, prins.fs, and service.fs to generate a new run.exe. We can then execute the command run server on one machine, and execute run client 100 15.99 on another. The resulting 4-kilobyte messages are instances of the symbolic messages, where each symbol expression is replaced by a concrete, string-encoded value. For instance, the timestamp ts1 is now the concrete XML element

```
<Timestamp Id="Timestamp"
  xmlns="http://...wss-wssecurity-utility-1.0.xsd">
  <Created>2006−04−27T09:12:17Z</Created>
  <Expires>2006−04−27T09:13:17Z</Expires>
</Timestamp>
```

and the signature value sv1 is now the 172-character base64-encoded string

```
4Bpd7K+2n6eW+brpEwYO9hdwHrcNPOAoK+Bqn4........KCstFrZQ24=
```

To test our concrete implementation for interoperability, we run our client with servers implemented with WSE and WCF. The response message generated by the WCF server does not include the X.509 certificate of the server, since the client is expected to have it already. We easily modify our client to ignore this difference and it successfully executes the protocol with WCF. The WSE server, however, does not support the signature confirmation mechanism for message correlation. Moreover, the key-sizes and encryption algorithms supported by WSE are different from and more limited than WCF. After disabling signature confirmation and using WSE key sizes and algorithms, our client successfully executes the protocol with the WSE server.

Each session of our implementation takes 1.2 seconds to complete the protocol. We expect that this is comparable to the performance of the WSE and WCF implementations because all three implementations use the same .NET cryptography libraries, XML parsers, and X.509 certificate stores. Indeed, in the default configuration, both WSE and WCF take around one second per session for our protocol. A direct comparison of the performance of the three protocol implementations has little significance, because WCF, and to a lesser extent WSE, is a full web services implementation running within a web server, whereas ours is a partial implementation focusing on security. The WSE implementation consists of around 185 lines of C# code, while the WCF implementation consists of around 70 lines of C# code and 160 lines of security-related XML configuration. In contrast, our implementation consists of 104 lines of F# code that can be executed concretely or symbolically, as well as automatically verified.

Security Goals and Theorem. We use the fs2pv/ProVerif tool chain to verify our protocol implementation against its security goals, both before and after modifying it for interoperability with WCF. Recall the three security goals for our protocol. Let G be these security goals expressed as ProVerif queries:

```
query ev:ServerRecv(u,s,x) ⇒ ev:ClientSend(u,_,x) | ev:Leak(u).
query ev:ClientCorr(u,s,x,y) ⇒ ev:ServerCorr(u,s,x,y) | ev:Leak(s).
query ev:NotSecret(v) ⇒
        (ev:ClientSend(u,s,DataTxt(DataBase64(DataFresh(v)))) & ev:Leak(s))
      | (ev:ServerCorr(u,s,r,DataTxt(DataBase64(DataFresh(v)))) & ev:Leak(u)).
```

—The first query formalizes request authentication: it says that, if the server principal s accepts a request x from a client principal u (ServerRecv(u,s,x)), then u has sent the request x (ClientSend(u,_,x)) or else u has been compromised—the Prins library emits the event Leak(u) to record that the attacker has been given the keys for principal u.

—The second query formalizes response authentication and correlation: if the client principal u accepts a response y for request x from server principal s (ClientCorr(u,s,x,y)), then s must have sent the response y to u for request x (ServerCorr(u,s,x,y)), or else s has been compromised.

—The third query expresses the secrecy of the request and response. It says that the only secrets v available to the attacker (NotSecret(v)) are those that have been sent within requests or responses to compromised servers or clients, respectively.

Let S be the F# system consisting of the X509MutualAuth.fs module, the web services library, and the symbolic implementations for the modules crypto.fs, net.fs, prins.fs, and service.fs. Let I_{pub} be the attacker interface from Section 3 extended with the protocol interface X509MutualAuth.fsi. We use fs2pv to compile S to a script consisting of 988 lines of pi calculus code. Then we run ProVerif to verify all three queries in G above. By Theorem 1, we obtain:

THEOREM 2. *For each $q \in G$, the system S is robustly safe for q and I_{pub}.*

Hence, we verify the security of our protocol implementation and all the functions it uses from the web services library against a powerful attacker model. The only modules we trust to be correct, and do not verify, are crypto.fs, net.fs, prins.fs, and service.fs.

Vulnerabilities and Attacks. Modifying our protocol implementation for for interoperability to WCF makes no difference to protocol correctness: we are still able to automatically establish Theorem 2. The modification for WSE, however, weakens the protocol: the second query (response authentication) fails and ProVerif reports an attack. Indeed, since the modified protocol does not use signature confirmation, an attacker can forward to the client a response generated by the server in reply to another request by the same client.

Trusted Library (+ CLR)			**Web Services Library**		**Protocol Module**	ProVerif **Script**
Modules	Concrete LoC	Symbolic LoC	Modules	LoC	LoC	LoC
4	793	575	5	1648	85-149	1090-1167

Table XI. Sizes of implementation modules and generated scripts

As a result, requests and responses are not securely correlated—this is a known issue in WS-Security 1.0, which led to the design of signature confirmation in WS-Security 1.1. More precisely, we can still capture a weaker notion of response authentication that holds for WSE, using the following, weaker variant of the second query:

ev:ClientCorr(u,s,x,y) ⇒ **ev**:ServerCorr(_,s,_,y) | **ev**:Leak(s).

This query states that the client authenticates the server s and the response message y, but does not correlate y with the request x. We verify that all variants of our protocol implementation satisfy this query.

The X.509 mutual authentication protocol presented in this section meets our specific set of authentication and secrecy goals, but is not unconditionally secure. We discuss two of its limitations.

—The protocol fails to guarantee certain other security properties. For instance, it fails to protect (stronger variants of) secrecy of *request* or *response* against guessing attacks, when these messages have low entropy. If such protection is required, we can either encrypt the signature in addition to the message content, or we can add a nonce to the message content.

—The protocol also fails to prevent certain replay attacks on the server. If the client produces a new timestamp for each request and if the server maintains a cache of these timestamps, then replays can be detected and discarded. Indeed, our formal model generates fresh globally unique timestamps for each message. Alternatively, we can include a unique message identifier in each request.

We also coded stronger variants of the protocol that meet at least the requirements of Theorem 2 and also address these limitations, and verified their implementation using additional queries. We omit the details for the sake of brevity.

5.2 Verification Results for Web Services Protocols

In addition to the X.509 Mutual Authentication protocol, we have implemented several other sample WSE and WCF protocols in F# and verified them. Tables VI and VII report our experimental results for four such protocols. *WS Password-based signing* is the web services version of our simple password-based authentication protocol of Section 2; it consists of a single SOAP message from a client to a web service, where the message contains an embedded XML digital signature keyed using a shared password. *WS X.509 signing* is a single message protocol where the message is signed using a private key. *WS Password-X.509 mutual auth* is a request-response protocol where the request is signed using a shared password and the response is signed using a private key. Finally, *WS X.509 mutual auth* is our case study implementing X.509 mutual authentication.

Table XI breaks down the size of the protocol implementation in terms of its logical components. The trusted library consists of four modules written in 793 lines of code and uses functionality provided by the CLR, such as System.Cryptography for cryptographic functions; the symbolic model of these modules and the underlying CLR is written in 575 lines of F#. The verified web services library consists of five modules written in 1648 lines of code. The protocol module varies between the different examples and takes around a hundred lines. The ProVerif script is generated from the symbolic trusted library, the web services library and the protocol module; it varies between the different examples and is around a thousand lines of pi calculus.

5.3 Implementing the Verified WS-Security Library

Programming a security protocol based on WS-Security is an exercise in modularity. The messages of the protocol include elements, such as timestamps, addresses, encrypted keys, and signatures, that are defined by different specifications. Many of these elements eventually rely on low-level cryptographic computations. To assemble the complete SOAP message, each element must be encoded in some XML format.

To support this kind of programming, we structure our WS-Security library as follows. For each specification, we define an F# module Spec.fs and an interface Spec.fsi. Within a module, each high-level message component is defined as a datatype T. Operations to generate and check elements of type T (typically using cryptographic functions) are written as functions mkT and isT. Finally, for each datatype T, the module defines functions genT and parseT to translate elements of T to and from XML items. In this way, users of the library can ignore the XML representation and instead program with the more abstract representation T and its corresponding functions.

For instance, the soap.fs module partially implements the SOAP standard [W3C 2003]. It has the following interface:

```
type envelope = { header: item list; body: item }
val parseEnvelope: item → envelope
val genEnvelope: envelope → item
```

A SOAP envelope is abstractly represented as a record that contains a list of headers and a body. The functions parseEnvelope and genEnvelope translate such records to and from XML items. Since there is no cryptography involved in constructing an envelope, there are no other functions in the interface.

Similarly, the wsaddressing.fs module implements the headers of the WS-Addressing specification [W3C 2004]; it has a record type that abstractly represents optional headers and it has functions to translate records to and from SOAP header elements.

The full WS-Security library consists of five F# modules, including both soap.fs and wsaddressing.fs, with a total of 1648 lines of code. We believe that these modules are usable not only by programmers aiming to write verifiable web services security protocols, but also by protocol designers looking for precise executable specifications for the web services standards. In the rest of this section, we look in more detail at the modules that implement the security mechanisms of WS-Security.

XML Signature. The XML Signature standard "specifies XML syntax and processing rules for creating and representing digital signatures." [Eastlake et al. 2002] An XML signature, as defined in the standard, cryptographically attests to the integrity and authenticity of a set of XML items. An example is the <Signature> element in the protocol messages in Tables IX and X. It includes metadata describing the computation of the signature value: each signed element is first transformed using the specified canonicalization method (xml−exc−c14n), then hashed using the specified digest method (SHA1); the digests and metadata are finally signed using the specified signature method (RSA−SHA1). The recipient of such a signature recomputes the digests and checks the received signature value before accepting the signed elements as authentic.

In our library, the xmldsig.fs module implements XML signatures. The datatype for an XML signature is a record dsig that includes the relevant contents of the <Signature> element as well as additional values needed for computing and checking the signature:

```
type dsig = {
   siginfo: item;
   sigval: bytes;
   keyinfo: item;
   signkey: keybytes option;
   verifkey: keybytes option;
   targets: item list }
```

The field siginfo corresponds to the <SignedInfo> element containing the metadata and all the digests; sigval contains the signature value; keyinfo identifies the signing key. The module contains auxiliary functions for generating siginfo from the list of signed elements (targets). To compute the sigval, we use a signing key (signkey); to check a received sigval, we use the corresponding verification key (verifkey).

The module provides functions for constructing and checking signatures using both symmetric and asymmetric signing algorithms, such as HMAC−SHA1 and RSA−SHA1:

```
val mkSignature: item list →item →keybytes →str →dsig
val isSignature: item list →keybytes →dsig →bytes
```

The function call, mkSignature targets keyinfo signkey alg, constructs a dsig element for the elements listed in targets, using signature key signkey (identified by keyinfo) and signing algorithm alg. Conversely, isSignature targets verifkey dsig uses verifkey to check that dsig is a valid XML signature computed from targets and returns the signature value, so that it can be used for signature confirmation. The full module consists of 307 lines of code.

There are several challenges in implementing XML Signature. First, our functions must correctly implement the low-level details of the signature. This includes not only the details of the XML format such as name spaces and attributes, but also the use of the canonicalization, digest, and signature algorithms. In xmldsig.fs, the functions parseSignature and genSignature translate records of type dsig to and from XML. We test these functions by inspecting the message traces as well as by extensive interoperability testing with other implementations. Our datatype and functions hide these details from the programmer, so all programs using these functions are guaranteed to generate standards-conformant XML signatures.

Second, the standard offers several options for each step of signature computation and an implementation is expected to support a subset. In our implementation, we choose one canonicalization and one digest algorithm, but allow two signature algorithms and several ways of referring to signing keys. These choices do not affect the module interface: the types and functions remain the same. Hence, we can easily add implementations for additional algorithms as the need arises and rely on the F# module and type system to integrate them.

XML Encryption. The XML Encryption standard "specifies a process for encrypting data and representing the result in XML" [Eastlake et al. 2002]. When parts of a message are to be encrypted using a symmetric key, the encrypted data mechanism can be used; when only an asymmetric key is available for encryption, one first generates a fresh symmetric key, uses it to encrypt data, and then protects the symmetric key using the encrypted key mechanism. Both these mechanisms are depicted in the protocol messages in Tables IX and X; the <EncryptedData> element contains a cipher value computed by applying a symmetric encryption algorithm (AES−128) to the message body using a key encrypted within an <EncryptedKey> element using an asymmetric algorithm (RSA−1.5).

The xmlenc.fs module implements XML encryption, in a similar style to xmldsig.fs. It defines two record types encdata and encrkey representing encrypted data and encrypted keys. It provides functions to construct (encrypt) and decrypt records of these types and functions to translate them to and from XML. It also provides functions to combine common encryption tasks; for instance, the function call, mkEncDatakey ek str plain, generates a fresh symmetric key, uses it to encrypt the plain-text plain as an encrypted data block, uses the public-key ek to in turn encrypt the symmetric key, and returns both the encrypted data and the encrypted key.

The module xmlenc.fs is implemented in 419 lines of code. It implements two symmetric algorithms for encrypting data, AES−128 and AES−256, and two asymmetric algorithms for encrypting keys, RSA−1.5 and RSA−OAEP. Our choices are motivated by the default settings in WSE and WCF; WSE supports AES−128 and RSA−1.5, while WCF uses AES−256 and RSA−OAEP.

WS-Security. The wssecurity.fs module implements the content of the security header, as specified in the WS-Security standard [OASIS 2004]. The security header contains several optional elements, such as a message timestamp, tokens identifying principals, XML signatures, and encrypted keys. The record representing this header is as follows:

```
type security = {
    timestamp: ts;
    utoks: utok list;
    xtoks: xtok list;
    ekeys: encrkey list;
    dsigs: dsig list }
```

It consists of a timestamp (ts), generated using the mkTimeStamp function, username tokens (utoks) identifying users and passwords, X.509 tokens (xtoks) containing public-key certificates, encrypted keys (ekeys), and XML signatures (dsigs).

The module offers functions for constructing different kinds of tokens and for generating signatures and encrypted blocks using them. The call mkX509Signature prin targets, for instance, generates an X.509 token corresponding to principal prin and uses its private key to compute an XML signature for the element list targets. The module also provides functions for translating security headers to and from XML. For instance, the function genX509SecurityHeader takes a certificate, an encrypted key, a timestamp, and a signature and generates the corresponding XML security header; parseX509SecurityHeader does the reverse.

The wssecurity.fs module consists of 538 lines of F# code. It does not yet support several token types defined in WS-Security, such as Kerberos and SAML tokens.

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.

We use this architecture to build and verify several web services security protocol implementations; our tools find vulnerabilities as well as prove strong security theorems.

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 case studies are among the largest examples of verified cryptographic protocol implementations to date. Since the publication of the conference version of this paper, our verification tool, fs2pv, has been used to verify implementations of the Windows Cardspace protocol [Bhargavan et al. 2008a], which is built using several web services security protocols, and the Transport Layer Security (TLS) protocol [Bhargavan et al. 2008b]. In both these studies, our tools were able to prove strong authentication and security properties of thousands of lines of interoperable protocol code. 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. Besides, lower level attacks (based for instance on timing analysis, power analysis, or fault injection) fall outside the scope of our model. We would need more precise, ad hoc programming models to account for them.

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 [Abadi and Rogaway 2002; Backes et al. 2003], and on automatically verifying computational models of protocols [Blanchet 2007]. Using an architecture similar to ours, computational models have recently been extracted and automatically verified for fragments of an implementation of the TLS protocol [Bhargavan et al. 2008b]. Still, for the time being, these tools 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 [Sumii and Pierce 2001; 2004]. 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 [Perrig et al. 2001; Muller and Millen 2001; Lukell et al. 2003; Pozza et al. 2004] 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 et al. [2005] 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 [2004] 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 [2005] 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 [2004] translate the policy files for web services to the TulaFale modelling language [Bhargavan, Fournet, Gordon, and Pucella 2004], 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 [Gordon and Pucella 2002; Bhargavan et al. 2005; Bhargavan et al. 2007c; Kleiner and Roscoe 2004; 2005] of web services security protocols has uncovered a range of potential attacks, but has no formal connection to source code. Goubault-Larrecq and Parrennes [2005] 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. Elyjah [O'Shea 2006] derives symbolic models in the LySa process calculus from implementation code in Java, and verifies properties of bounded instances of the models using an analyzer for LySa [Bodei et al. 2003]. The Java code represents various protocol examples as concurrent processes within a single machine, and uses custom message formats.

There is also recent research on verifying implementations of cryptographic algorithms, as opposed to protocols. For instance, Cryptol [Galois Connections 2005] is a language-based approach to verifying implementations of algorithms such as AES.

Acknowledgments. James Margetson and Don Syme helped us enormously with using and adapting the F# compiler. Bruno Blanchet helped us with understanding and debugging ProVerif scripts. Tony Hoare, David Langworthy, and the anonymous referees suggested improvements to the presentation.

A. AN OBSERVATIONAL EQUIVALENCE FOR THE PI CALCULUS

In this appendix, we define an observational equivalence relation, which we use in the pi calculus proofs in Appendix B. We begin with a definition of evaluation contexts. A context is a process containing a hole $[\_]$; we write $E[P]$ for the outcome of filling the hole in E with the process P.

Evaluation Context:

$E ::= [\_] \mid \textbf{new}\ a; E \mid (P \mid E)$ evaluation context

LEMMA 3. *If P is safe for q then **new** $a;P$ is safe for q.*

(Recall that our definition of queries excludes a from occurring in q.)

LEMMA 4. *If Δs **process** P is robustly safe for q and the process $E[\mathbf{0}]$ is a Δs-opponent then $E[P]$ is safe for q.*

Proof: If $E[\mathbf{0}]$ is a Δs-opponent then $E[P] \equiv \textbf{new}\ as; (P \mid O)$ for some names as and a Δs-opponent O. By definition, $P \mid O$ is safe for q. By Lemma 3, **new** $as; (P \mid O)$ is safe for q. Safety is preserved by the relation $\equiv$, so $E[P]$ is safe for q. □

We define an observational equivalence on processes induced by query satisfaction.

Observational Equivalence: $P \approx Q$

Let $\approx$ be the largest symmetric relation on closed processes such that $P \approx Q$ implies: (1) $E[P] \approx E[Q]$ for all closed evaluation contexts E; (2) $P \rightarrow P'$ implies there is Q' with $P' \approx Q'$ and $Q \rightarrow^{*}_{\equiv} Q'$; and (3) $P \models q$ implies $Q \models q$ for all q.

Recall the standard pi calculus notion of a forwarder: let $a \rightarrow b$ be $!\textbf{in}(a,z).\textbf{out}(b,z)$. We use forwarders in our translation and rely on the following property phrased in terms of observational equivalence. We give a detailed proof outline for this standard property; there are proofs of similar properties in the literature [Merro and Sangiorgi 1998; Fournet and Gonthier 2005].

LEMMA 5. *Let P be a process that uses the name a only for sending asynchronous messages. Then we have the observational equivalence **new** $a; (P \mid a \rightarrow b) \approx P\{b/a\}$.*

We rely on a similar property for eliminating channel publish in the proof of Lemma 21.

LEMMA 6. *Let Q be a process with no event in evaluation context, xs a tuple that carries the free names and variables of Q, C a context, and* publish *a fresh channel. We* *have*

$$C[Q] \approx \textbf{new}\ \text{publish}; ((!\textbf{in}(\text{publish}, xs); Q) \mid C[\textbf{out}(\text{publish}, xs)])$$

B. PROOFS OF CORRESPONDENCE AND SAFETY THEOREMS

This appendix introduces definitions and lemmas for proving Theorem 1, that robust safety for programs in F follows from robust safety of the translation of F to the pi calculus. Many proof details are omitted; they appear in the technical report [Bhargavan et al. 2007b].

The main difficulty in this development concerns the statement of Lemmas 16 and 17, which relate the reduction of an F configuration to reduction steps of its translation into the pi calculus. The difficulty concerns the reduction step needed to return a result from the process translation of a function call. There is no exactly corresponding reduction at the F level, as the reduction rule for the function call simply inlines the function body. To solve this problem, we phrase Lemmas 16 and 17 in terms of an auxiliary *guarded reduction* relation, which allows certain guarded reductions to be anticipated. Hence, the step of calling a function at the F level corresponds at the process level to an ordinary reduction (for the call) followed by a guarded reduction (for the return).

Theorem 1 concerns a system S and an interface I_{pub}. Throughout this section, we fix these variables and assume the following. As opposed to S, we let $\hat{S}$ range over arbitrary systems. (Since our translation does not operate on values, our development does not depend on the details of the ambient declarations.)

Assumptions about system S and interface I_{pub}:

We assume $S :: I_{pub}$ and ambient declarations $\Delta s[\![S :: I_{pub}]\!]$.

In addition to the translations of Section 4.3, we translate F running configurations into pi calculus processes as follows; the main translation function $\mathscr{C}[\![C]\!]$ is defined in terms of an auxiliary translation $\mathscr{C}'[\![C]\!]$.

Processes $\mathscr{C}[\![C]\!]$ and $\mathscr{C}'[\![C]\!]$ representing configuration C:

$\mathscr{C}[\![C]\!] \stackrel{\triangle}{=} \textbf{new}\ as; \mathscr{C}'[\![C]\!] \quad$ where $as = fn(C)$

$\mathscr{C}'[\![C_1 \mid C_2]\!] \stackrel{\triangle}{=} \mathscr{C}'[\![C_1]\!] \mid \mathscr{C}'[\![C_2]\!]$
$\mathscr{C}'[\![\textbf{event}\ M]\!] \stackrel{\triangle}{=} \textbf{event}\ M$
$\mathscr{C}'[\![\hat{S}]\!] \stackrel{\triangle}{=} \mathscr{S}[\![\hat{S}]\!](\mathbf{0})$

The translated processes, as shown by the next two lemmas, respect structural equivalence as well as substitution of values. These lemmas are useful in proving the reduction correspondence. To state the second lemma, we need some additional terminology. We say that x is bound in a declaration when the declaration takes the form **let** $x = e$. We say that x is bound in a system $S = d_1 \ldots d_n$ when x is bound in d_i for some $i \in 1..n$.

LEMMA 7. *If $C \equiv C'$ then $\mathscr{C}'[\![C]\!] \equiv \mathscr{C}'[\![C']\!]$.*

LEMMA 8 SUBSTITUTION.

(1) $\mathscr{E}[\![e]\!](x,P)\{M/y\} = \mathscr{E}[\![e\{M/y\}]\!](x,P\{M/y\})$ *if $x \neq y$ and $x \notin fv(M)$.*
(2) $\mathscr{S}[\![d]\!](P)\{M/x\} = \mathscr{S}[\![d\{M/x\}]\!](P\{M/x\})$ *if x not bound by d.*

(3) $\mathscr{S}[\![\hat{S}]\!](P)\{M/x\} = \mathscr{S}[\![\hat{S}\{M/x\}]\!](P\{M/x\})$ *if x not bound by $\hat{S}$.*

To understand the relationship between the operational semantics of an F configuration, and its translation to a process, it is convenient to define a specialized *guarded reduction* relation, written $P \rightsquigarrow Q$. A guarded reduction of a process anticipates a reduction step that is currently guarded, but may be enabled after subsequent reductions. In particular, we define guarded reduction to anticipate the eventual transmission of a message on a continuation, so that $\textbf{new}\ k;(\mathscr{E}[\![e]\!](x,\textbf{out}(k,x)) \mid \textbf{in}(k,x);P) \rightsquigarrow \mathscr{E}[\![e]\!](x,P)$. The eventual reduction step is deterministic, so guarded reduction does not resolve any nondeterminism.

Consider the set of *linear contexts*, ranged over by the metavariable L, defined as follows. Such a context is linear in the sense that the hole in the context, written $[\_]$, is activated at most once. Guarded reduction is defined in terms of a notion of *guarded linear context*.

Linear Contexts and Guarded Linear Contexts:

$L ::=$
$[\_]$
$L\sigma$
$\textbf{out}(M,N);L$
$\textbf{in}(M,x);L$
$\textbf{new}\ a;L$
$P \mid L$
$\textbf{let}\ x_1,\ldots,x_n\ \textbf{suchthat}\ M = N\ \textbf{in}\ L\ \textbf{else}\ P$
$\textbf{let}\ x_1,\ldots,x_n\ \textbf{suchthat}\ M = N\ \textbf{in}\ L\ \textbf{else}\ L$

Let L be a *guarded linear context* if every occurrence of $[\_]$ is within an **out**, **in**, or **let**.
Let $bv(L)$ be the set of variables in scope for any occurrences of $[\_]$.

The definition of linear context is tailored to the following lemma; in particular, the two separate clauses for **suchthat** arise from the translation of pattern matching. In the context $L\sigma$, σ is an explicit substitution; its application is deferred until the hole in L is filled with a process. We write $L[P]$ for the outcome of filling the holes in L with the process P; in particular, $(L\sigma)[P]$ is $L[P]\sigma$, that is, the outcome of actually applying the substitution to $L[P]$.

LEMMA 9. *For all e and x, $\mathscr{E}[\![e]\!](x,L)$ is a linear context whenever L is.*

LEMMA 10. *If L is a linear context that is not guarded, there is an evaluation context E and a substitution σ, such that for all processes Q, $L[Q] = E[Q\sigma]$.*

Guarded Reduction: $P \rightsquigarrow Q$

$\textbf{new}\ k;(G[\textbf{out}(k,M)] \mid \textbf{in}(k,x);P) \rightsquigarrow G[P\{M/x\}]$
where $k \notin n(G) \cup fn(M,P)$, and G is a guarded linear context, and $fv(P) \cap bv(G) \subseteq \{x\}$
$P \rightsquigarrow P' \Rightarrow \textbf{new}\ a;P \rightsquigarrow \textbf{new}\ a;P'$
$P \rightsquigarrow P' \Rightarrow P \mid R \rightsquigarrow P' \mid R$
$P \equiv Q, Q \rightsquigarrow Q', Q' \equiv P' \Rightarrow P \rightsquigarrow P'$

The following lemma is an alternative characterization of guarded reduction in terms of evaluation contexts. Each evaluation context, E, is a linear context, but is not guarded.

LEMMA 11. *$P \rightsquigarrow P'$ if and only if there is an evaluation context E, a guarded linear context G, a value M, a process Q with $fv(Q) \cap bv(G) \subseteq \{x\}$, and a name $k \notin n(G) \cup fn(M,Q)$, such that*

$$P \equiv E[\mathbf{new}\ k; (G[\mathbf{out}(k,M)] \mid \mathbf{in}(k,x);Q)] \qquad P' \equiv E[G[Q\{M/x\}]]$$

As intended, guarded reduction formalizes the eventual communication of the result x from an F expression e on its continuation channel k to a continuation process P.

LEMMA 12. *If $k \notin n(e,P)$ then $\mathbf{new}\ k; (\mathscr{E}[\![e]\!](x, \mathbf{out}(k,x)) \mid \mathbf{in}(k,x);P)\ (\rightarrow \cup \rightsquigarrow)\ \mathscr{E}[\![e]\!](x,P)$.*

The following lemma makes explicit the decomposition of a reduction that may involve a guarded context.

LEMMA 13. *Let E be an evaluation context, G be a guarded linear context, and P a process. If $E[G[P]] \rightarrow Q'$, then there exist E' evaluation context and L linear context such that (1) for all processes R, $E[G[R]] \rightarrow E'[L[R]]$; and (2) $Q' \equiv E'[L[P]]$.*

The following lemma formalizes the intuition that a guarded reduction anticipates a subsequent reduction step.

LEMMA 14. *If $P \rightsquigarrow\rightarrow P'$ then either $P \rightarrow\rightsquigarrow P'$ or $P \rightarrow\rightarrow P'$.*

Proof: By Lemma 11, $P \rightsquigarrow P^\circ \rightarrow P'$ implies that there is an evaluation context E, a guarded linear context G, a value M, a process Q with $fv(Q) \cap bv(G) \subseteq \{x\}$, and a name $k \notin n(G) \cup fn(M,Q)$, such that:

$$P \equiv E[\mathbf{new}\ k; (G[\mathbf{out}(k,M)] \mid \mathbf{in}(k,x);Q)] \qquad P^\circ \equiv E[G[Q\{M/x\}]]$$

By Lemma 13, we obtain E' and L such that $E[G[R]] \rightarrow E'[L[R]]$ for all processes R, and $P' \equiv E'[L[Q\{M/x\}]]$. Without loss of generality, we assume that k does not occur in L or E'. For $R = \mathbf{out}(k,M) \mid \mathbf{in}(k,x);Q$, we have a reduction:

$$E[G[\mathbf{out}(k,M) \mid \mathbf{in}(k,x);Q]] \ \rightarrow\ E'[L[\mathbf{out}(k,M) \mid \mathbf{in}(k,x);Q]]$$

By induction on the derivation for this reduction, and since there are no further occurrences of k in the processes above, we obtain the following reduction from P:

$$P \ \equiv\ E[\mathbf{new}\ k; (G[\mathbf{out}(k,M)] \mid \mathbf{in}(k,x);Q)] \ \rightarrow\ E'[\mathbf{new}\ k; (L[\mathbf{out}(k,M)] \mid \mathbf{in}(k,x);Q)]$$

We distinguish two cases, depending on L:

—If L is guarded, then

$$P \rightarrow E'[\mathbf{new}\ k; (L[\mathbf{out}(k,M)] \mid \mathbf{in}(k,x);Q)] \ \rightsquigarrow\ E'[L[Q\{M/x\}]] \ \equiv\ P'$$

—Otherwise, by Lemma 10, L is of the form $E''[\_\sigma]$ for some substitution σ and we have

$$P \rightarrow E'[\mathbf{new}\ k; (E''[\mathbf{out}(k,M\sigma)] \mid \mathbf{in}(k,x);Q)] \ \rightarrow\ E'[E''[Q\{M\sigma/x\}]] \ \equiv\ P'$$

(Intuitively, substitutions in L record variables previously bound by a guard of G, such as received variables bound in input guards.) □

Guarded reductions do not affect query satisfaction, as they never introduce events in evaluation contexts.

LEMMA 15. *If $P \models q$ and $P \rightsquigarrow P'$ then $P' \models q$.*

Now, the main lemma supporting the safety theorem can be formally stated: a reduction of a configuration at the F level corresponds to a series of reductions followed by a series of guarded reductions at the pi calculus level.

LEMMA 16. *If $C \to C'$ and $as = fn(C') \setminus fn(C)$ then $\mathscr{C}'[\![C]\!] \to_{\equiv}^{*} \rightsquigarrow^{*}$ **new** $as; \mathscr{C}'[\![C']\!]$.*

Proof: The proof is by induction on the derivation of $C \to C'$. We consider each rule of the definition of $C \to C'$, and in each case let $as = fn(C') \setminus fn(C)$. By definition of $C \to C'$, both C and C' are closed. We show the case for function application, the one case that requires the use of guarded reduction.

—Case $C_0 \mid \mathbf{let}\ x = \ell\ M_1\ \ldots\ M_n\ \hat{S} \to C_0 \mid \mathbf{let}\ x = e\{M_1/x_1, \ldots, M_n/x_n\}\ \hat{S}$
if $C_0 = C_1 \mid \mathbf{let}\ \ell\ x_1 \ldots x_n = e$, with $as = \varnothing$.
Let *LHS* and *RHS* be the translations of the configurations before and after the reduction. In the following, we choose k_2 to be fresh, and we may choose the intermediate variable x in $\mathscr{E}[\![e]\!](x, \mathbf{out}(k_1, x))$ to be the same as the bound variable x in $\mathbf{let}\ x = \ell\ M_1\ \ldots\ M_n$. We make iterated appeals below to the substitution lemma, Lemma 8(1) (as we may assume the bound variable x is not free in $M_1, \ldots, M_n$, and that $x \neq x_i$ for each $i \in 1..n$). At the last step for the *LHS*, we appeal to Lemma 12.

$$\begin{array}{rcl}
LHS & = & \mathscr{C}'[\![C_1 \mid \mathbf{let}\ \ell\ x_1 \ldots x_n = e \mid \mathbf{let}\ x = \ell\ M_1\ \ldots\ M_n\ \hat{S}]\!] \\
& = & \mathscr{C}'[\![C_1]\!] \mid \mathscr{C}'[\![\mathbf{let}\ \ell\ x_1 \ldots x_n = e]\!] \mid \mathscr{C}'[\![\mathbf{let}\ x = \ell\ M_1\ \ldots\ M_n\ \hat{S}]\!] \\
& = & \mathscr{C}'[\![C_1]\!] \mid \mathscr{S}[\![\mathbf{let}\ \ell\ x_1 \ldots x_n = e]\!](\mathbf{0}) \mid \mathscr{S}[\![\mathbf{let}\ x = \ell\ M_1\ \ldots\ M_n\ \hat{S}]\!](\mathbf{0}) \\
& = & \mathscr{C}'[\![C_1]\!] \mid !\mathbf{in}(\ell, (x_1, \ldots, x_n, k_1)); \mathscr{E}[\![e]\!](x, \mathbf{out}(k_1, x)) \mid \mathbf{0} \\
& & \mid \mathscr{E}[\![\ell\ M_1\ \ldots\ M_n]\!](x, \mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& \equiv & \mathscr{C}'[\![C_0]\!] \mid \mathbf{in}(\ell, (x_1, \ldots, x_n, k_1)); \mathscr{E}[\![e]\!](x, \mathbf{out}(k_1, x)) \mid \\
& & \quad \mathbf{new}\ k_2; (\mathbf{out}(\ell, (M_1, \ldots, M_n, k_2)) \mid \mathbf{in}(k_2, x); \mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& \equiv & \mathscr{C}'[\![C_0]\!] \mid \mathbf{new}\ k_2; (\mathbf{in}(\ell, (x_1, \ldots, x_n, k_1)); \mathscr{E}[\![e]\!](x, \mathbf{out}(k_1, x)) \mid \\
& & \qquad \mathbf{out}(\ell, (M_1, \ldots, M_n, k_2)) \mid \mathbf{in}(k_2, x); \mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& \to & \mathscr{C}'[\![C_0]\!] \mid \mathbf{new}\ k_2; (\mathscr{E}[\![e]\!](x, \mathbf{out}(k_1, x))\{M_1/x_1, \ldots, M_n/x_n, k_2/k_1\} \mid \\
& & \qquad \mathbf{in}(k_2, x); \mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& = & \mathscr{C}'[\![C_0]\!] \mid \mathbf{new}\ k_2; (\mathscr{E}[\![e\{M_1/x_1, \ldots, M_n/x_n\}]\!](x, \mathbf{out}(k_2, x)) \mid \\
& & \qquad \mathbf{in}(k_2, x); \mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& \to \cup \rightsquigarrow & \mathscr{C}'[\![C_0]\!] \mid \mathscr{E}[\![e\{M_1/x_1, \ldots, M_n/x_n\}]\!](x, \mathscr{S}[\![\hat{S}]\!](\mathbf{0}))
\end{array}$$

$$\begin{array}{rcl}
RHS & = & \mathscr{C}'[\![C_0 \mid \mathbf{let}\ x = e\{M_1/x_1, \ldots, M_n/x_n\}\ \hat{S}]\!] \\
& = & \mathscr{C}'[\![C_0]\!] \mid \mathscr{C}'[\![\mathbf{let}\ x = e\{M_1/x_1, \ldots, M_n/x_n\}\ \hat{S}]\!] \\
& = & \mathscr{C}'[\![C_0]\!] \mid \mathscr{S}[\![\mathbf{let}\ x = e\{M_1/x_1, \ldots, M_n/x_n\}\ \hat{S}]\!](\mathbf{0}) \\
& = & \mathscr{C}'[\![C_0]\!] \mid \mathscr{S}[\![\mathbf{let}\ x = e\{M_1/x_1, \ldots, M_n/x_n\}]\!](\mathscr{S}[\![\hat{S}]\!](\mathbf{0})) \\
& = & \mathscr{C}'[\![C_0]\!] \mid \mathscr{E}[\![e\{M_1/x_1, \ldots, M_n/x_n\}]\!](x, \mathscr{S}[\![\hat{S}]\!](\mathbf{0}))
\end{array}$$

□

LEMMA 17 REDUCTION CORRESPONDENCE. *If $C \to C'$ then $\mathscr{C}[\![C]\!] \to_{\equiv}^{*} \rightsquigarrow^{*} \mathscr{C}[\![C']\!]$.*

Proof: Assume $C \to C'$. Let $as_1 = fn(C)$, $as_2 = fn(C')$, and $as = as_2 \setminus as_1$. By Lemma 16,

$$\mathscr{C}'[\![C]\!] \to_{\equiv}^{*} \rightsquigarrow^{*} \mathbf{new}\ as; \mathscr{C}'[\![C']\!]$$

By definition, the relations $\equiv$, $\rightsquigarrow$, and $\to$ are closed under restriction, hence so too is

$\rightarrow_{\equiv}^{*}\rightsquigarrow^{*}$. Hence, we add as_1 to both sides to obtain:

$$\textbf{new } as_1; \mathscr{C}'[\![C]\!] \rightarrow_{\equiv}^{*}\rightsquigarrow^{*} \textbf{new } as_1; \textbf{new } as; \mathscr{C}'[\![C']\!]$$

By definition, $\rightarrow_{\equiv}^{*}\rightsquigarrow^{*}$ is closed on the right by $\equiv$, so we get:

$$\textbf{new } as_1; \mathscr{C}'[\![C]\!] \rightarrow_{\equiv}^{*}\rightsquigarrow^{*} \textbf{new } as_2; \mathscr{C}'[\![C']\!]$$

Hence, by definition of $\mathscr{C}[\![C]\!]$ and $\mathscr{C}'[\![C']\!]$, we have $\mathscr{C}[\![C]\!] \rightarrow_{\equiv}^{*}\rightsquigarrow^{*} \mathscr{C}[\![C']\!]$. $\square$

Finally, the following lemmas lead to our main result.

LEMMA 18 EVENT CORRESPONDENCE. *If $\mathscr{C}'[\![C]\!] \equiv \textbf{event } M \mid P$ then $C \equiv \textbf{event } M \mid C'$ for some C'.*

Proof: The proof is by induction on the structure of C. In the case for $C = \hat{S}$ and $\mathscr{C}'[\![C]\!] = \mathscr{S}[\![\hat{S}]\!](\mathbf{0})$, the proof relies on the fact that there are no N and P such that $\mathscr{S}[\![\hat{S}]\!](\mathbf{0}) \equiv \textbf{event } N \mid P$. To achieve this property, we must take care that **event** M is not prematurely activated in the definition of the process $\mathscr{E}[\![\text{log } M]\!](x,Q)$. We ensure this by taking $\mathscr{E}[\![\text{log } M]\!](x,Q)$ to be **record** $M; Q\{()/x\}$; the definition of the **record** abbreviation guards the activation of **event** M by a τ-step. $\square$

LEMMA 19 QUERY CORRESPONDENCE. *If $\mathscr{C}[\![C]\!] \models q$ then $C \models q$.*

Proof: Let $q = \textbf{ev}{:}E \Rightarrow \textbf{ev}{:}B_1 \vee \cdots \vee \textbf{ev}{:}B_n$. To prove $C \models q$, suppose that $C \equiv \textbf{event } E\sigma \mid C'$. We are to show that $C' \equiv \textbf{event } B_i\sigma \mid C''$ for some $i \in 1..n$ and C''. By Lemma 7, we have:

$$\begin{aligned}
\mathscr{C}[\![C]\!] &\equiv \mathscr{C}[\![\textbf{event } E\sigma \mid C']\!] \\
&= \textbf{new } as; \mathscr{C}'[\![\textbf{event } E\sigma \mid C']\!] \\
&= \textbf{new } as; (\mathscr{C}'[\![\textbf{event } E\sigma]\!] \mid \mathscr{C}'[\![C']\!]) \\
&= \textbf{new } as; (\textbf{event } E\sigma \mid \mathscr{C}'[\![C']\!])
\end{aligned}$$

where $as = fn(E\sigma, \mathscr{C}'[\![C']\!])$. Given that $\mathscr{C}[\![C]\!] \models q$, we have $\mathscr{C}'[\![C']\!] \equiv \textbf{event } B_i\sigma \mid P''$ for some $i \in 1..n$. By Lemma 18, $C' \equiv \textbf{event } B_i\sigma \mid C''$ for some $i \in 1..n$ and C''. $\square$

LEMMA 20 REFLECTION OF SAFETY. *If $\mathscr{C}[\![\hat{S}]\!]$ is safe for q then $\hat{S}$ is safe for q.*

Proof: Suppose that $\hat{S} \rightarrow_{\equiv}^{*} C$. We are to show $C \models q$. By Lemma 17 and induction on the reductions in F, we have $\mathscr{C}[\![\hat{S}]\!](\rightarrow_{\equiv}^{*}\rightsquigarrow^{*})^{*}\mathscr{C}[\![C]\!]$. By Lemma 14, $\mathscr{C}[\![\hat{S}]\!] \rightarrow_{\equiv}^{*} P$ and $P \rightsquigarrow^{*} \mathscr{C}[\![C]\!]$ for some process P. Since $\mathscr{C}[\![\hat{S}]\!]$ is safe for q, and $\mathscr{C}[\![\hat{S}]\!] \rightarrow_{\equiv}^{*} P$, we have $P \models q$. By Lemma 15, this and $P \rightsquigarrow^{*} \mathscr{C}[\![C]\!]$ imply $\mathscr{C}[\![C]\!] \models q$. By Lemma 19, $C \models q$. $\square$

Throughout this section we are assuming $S :: I_{pub}$, with the reduction relation implicitly depending on the ambient declarations $\Delta s[\![S :: I_{pub}]\!]$.

We now deal with robust safety, relating opponent top-level programs in F and opponent parallel-contexts in the pi calculus.

LEMMA 21. *Let O be an I_{pub}-opponent. Let ℓs be the functions declared in S. For some evaluation context E_O such that $E_O[\mathbf{0}]$ is a $\Delta s[\![S :: I_{pub}]\!]$-opponent, we have:*

$$\textbf{new } \ell s; \mathscr{S}[\![S\ O]\!](\mathbf{0}) \approx E_O[\mathscr{P}[\![S :: I_{pub}]\!]]$$

Proof: Since O is an I_{pub}-opponent, a function $\ell \in \ell s$ may occur in O only when $\ell \in dom(I_{pub})$, and a variable x may occur free in O only when $x \in dom(I_{pub})$. By definition of the translation, the same property holds for $\mathscr{S}[\![O]\!](\mathbf{0})$.

Let publish be a fresh name and let $\ell's$ be fresh distinct names in bijection with the names $\ell s \cap dom(I_{pub})$, that is, the names of the published functions. We write $\prod_{\ell' \in \ell' s} \ell' \to \ell$ for the parallel composition consisting of a forwarder $\ell' \to \ell$ for each $\ell' \in \ell' s$, where $\ell \in \ell s \cap dom(I_{pub})$ is the name in bijection with ℓ'. In the definition of $\mathscr{P}[\![S :: I_{pub}]\!]$, the tuple xs carries $\ell' s$ plus each variable $x \in dom(I_{pub})$.

We calculate the desired equation as follows:

$$\begin{array}{lll}
& \textbf{new}\ \ell s; \mathscr{S}[\![S\ O]\!](\mathbf{0}) & \\
= & \textbf{new}\ \ell s; \mathscr{S}[\![S]\!](\mathscr{S}[\![O]\!](\mathbf{0})) & \\
\approx & \textbf{new}\ \ell' s, \ell s; (\prod_{\ell' \in \ell' s} \ell' \to \ell \mid \mathscr{S}[\![S]\!](\mathscr{S}[\![O]\!](\mathbf{0})\{\ell' s/\ell s\})) & (1) \\
\approx & \begin{array}{l}\textbf{new}\ \text{publish}; (!\textbf{in}(\text{publish}, xs); \mathscr{S}[\![O]\!](\mathbf{0})\{\ell' s/\ell s\} \\ \mid \textbf{new}\ \ell' s, \ell s; (\prod_{\ell' \in \ell' s} \ell' \to \ell \mid \mathscr{S}[\![S]\!](\textbf{out}(\text{publish}, xs))))\end{array} & (2) \\
= & \textbf{new}\ \text{publish}; (!\textbf{in}(\text{publish}, xs); \mathscr{S}[\![O]\!](\mathbf{0})\{\ell' s/\ell s\} \mid \mathscr{P}[\![S :: I_{pub}]\!]) & (3)
\end{array}$$

The observational equivalence (1) is obtained by applying Lemma 5 to introduce a forwarder $\ell' \to \ell$ for every name in ℓs. We need that the process $\mathscr{S}[\![S]\!](\mathscr{S}[\![O]\!](\mathbf{0}))\{\ell' s/\ell s\}$ uses each name $\ell' \in \ell' s$ only for sending asynchronous messages, which follows by checking that the process $\mathscr{S}[\![S]\!](\mathscr{S}[\![O]\!](\mathbf{0}))$ uses each name $\ell \in \ell s$ only for sending asynchronous messages.

The observational equivalence (2) is an instance of Lemma 6; we rely on the hypothesis that all the names and variables bound in the context $\textbf{new}\ \ell' s, \ell s; (\prod_{\ell' \in \ell' s} \ell' \to \ell \mid \mathscr{S}[\![S]\!]([\_]))$ that occur in $\mathscr{S}[\![O]\!](\mathbf{0})\{\ell' s/\ell s\}$ are included in xs.

The final step (3) uses the definition of the top-level translation. To conclude, we let

$$E_0 = \textbf{new}\ \text{publish}; (!\textbf{in}(\text{publish}, xs); \mathscr{S}[\![O]\!](\mathbf{0})\{\ell' s/\ell s\} \mid [\_])$$

and check that $E_0[\mathbf{0}]$ is a $\Delta s[\![S :: I_{pub}]\!]$-opponent. □

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

Proof: Recall that $S :: I_{pub}$ means that $\text{Prim} \vdash S : I$ where $I = I_{pub}, I_{priv}$ for some I_{priv} and that $[\![S :: I_{pub}]\!]$ is the script defining the process $\mathscr{P}[\![S :: I_{pub}]\!]$ with the ambient declarations $\Delta s[\![S :: I_{pub}]\!]$ we have assumed throughout this section.

Suppose $\text{Prim} \backslash \text{log}, I_{pub} \vdash O : I_O$. Without loss of generality, we assume that I_O mentions just one constructor declaration, Box:**ctor** 2. We are to show that $S\ O$ is safe for q.

By Lemma 21, we have $\textbf{new}\ \ell s; \mathscr{S}[\![S\ O]\!](\mathbf{0}) \approx E_O[\mathscr{P}[\![S :: I_{pub}]\!]]$ where ℓs is the set of functions declared in S and E_0 is an evaluation context such that $E_0[\mathbf{0}]$ is a $\Delta s[\![S :: I_{pub}]\!]$-opponent. By assumption, $[\![S :: I_{pub}]\!]$ is robustly safe for q. By Lemma 4, $E_O[\mathscr{P}[\![S :: I_{pub}]\!]]$ is safe for q. By definition of $\approx$, $\textbf{new}\ \ell s; \mathscr{S}[\![S\ O]\!](\mathbf{0})$ is also safe for q. By definition, $\mathscr{C}[\![S\ O]\!] = \textbf{new}\ as; \mathscr{S}[\![S\ O]\!](\mathbf{0})$ where $as = fn(S\ O)$. Each name in ℓs occurs free in $\mathscr{S}[\![S\ O]\!](\mathbf{0})$, so $\ell s \subseteq as$. Since $\textbf{new}\ \ell s; \mathscr{S}[\![S\ O]\!](\mathbf{0})$ is safe for q, Lemma 3 implies $\mathscr{C}[\![S\ O]\!]$ is safe for q. By Lemma 20, $S\ O$ is safe for q. □

Assuming that fs2pv computes a script that consists of declarations $\Delta s[\![I_{pub}, I_{priv}]\!]$ and process $\mathscr{P}[\![S : I_{pub}]\!]$, and that ProVerif correctly decides whether a process is robustly safe for a query, Theorem 1 justifies relying on the fs2pv-ProVerif tool chain to determine robust safety of S.

REFERENCES

ABADI, M. AND FOURNET, C. 2001. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. 104–115.

ABADI, M. AND GORDON, A. D. 1999. A calculus for cryptographic protocols: The spi calculus. *Information and Computation 148*, 1–70.

ABADI, M. AND ROGAWAY, P. 2002. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology 15,* 2, 103–127.

ALLAMIGEON, X. AND BLANCHET, B. 2005. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 140–154.

Apache Software Foundation 2006. *Apache WSS4J*. Apache Software Foundation. At `http://ws.apache.org/wss4j/`.

ASKAROV, A. AND SABELFELD, A. 2005. Security-typed languages for implementation of cryptographic protocols: A case study. In *10th European Symposium on Research in Computer Security (ESORICS'05)*. LNCS, vol. 3679. Springer, 197–221.

BACKES, M., PFITZMANN, B., AND WAIDNER, M. 2003. A composable cryptographic library with nested operations. In *10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM Press, 220–230.

BERRY, G. AND BOUDOL, G. 1990. The chemical abstract machine. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*. 81–94.

BHARGAVAN, K., CORIN, R., FOURNET, C., AND GORDON, A. D. 2007c. Secure sessions for web services. *ACM Transactions on Information and System Security 10,* 2. Article 8.

BHARGAVAN, K., CORIN, R., FOURNET, C., AND ZĂLINESCU, E. 2008b. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS'08)*. To appear.

BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. 2005. A semantics for web services authentication. *Theor. Comput. Sci. 340,* 1, 102–153.

BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. 2006b. Verified reference implementations of WS-Security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*. LNCS, vol. 4184. Springer, 88–106.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND PUCELLA, R. 2004. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*. LNCS, vol. 3188. Springer, 197–222.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND SWAMY, N. 2008a. Verified implementations of the Information Card federated identity-management protocol. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*. 123–135.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2006a. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 139–152.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2007a. Verified interoperable implementations of security protocols. In *Software System Reliability and Security*. IOS Press, 87–115.

BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2007b. Verified interoperable implementations of security protocols. Tech. Rep. MSR–TR–2006–46, Microsoft Research.

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

BLANCHET, B. 2007. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. 97–111.

BLANCHET, B., ABADI, M., AND FOURNET, C. 2005. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*. 331–340.

BLANCHET, B. AND PODELSKI, A. 2005. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science 333,* 1-2, 67–90.

BODEI, C., BUCHHOLTZ, M., DEGANO, P., AND NIELSON, F. 2003. Automatic validation of protocol narration. In *16th IEEE Computer Security Foundations Workshop (CSFW'03)*. 126–140.

DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory IT–29,* 2, 198–208.

EASTLAKE, D., REAGLE, J., IMAMURA, T., DILLAWAY, B., AND SIMON, E. 2002. *XML Encryption Syntax and Processing*. W3C. W3C Recommendation.

EASTLAKE, D., REAGLE, J., SOLO, D., BARTEL, M., BOYER, J., FOX, B., LAMACCHIA, B., AND SIMON, E. 2002. *XML-Signature Syntax and Processing*. W3C. W3C Recommendation.

FOURNET, C. AND GONTHIER, G. 2005. A hierarchy of equivalences for asynchronous calculi. *Journal of Logic and Algebraic Programming 63*, 131–173.

Galois Connections 2005. *Cryptol Reference Manual*. Galois Connections.

GIAMBIAGI, P. AND DAM, M. 2004. On the secure implementation of security protocols. *Science of Computer Programming 50*, 73–99.

GORDON, A. D. AND PUCELLA, R. 2002. Validating a web service security abstraction by typing. In *2002 ACM workshop on XML Security*. 18–29.

GOUBAULT-LARRECQ, J. AND PARRENNES, F. 2005. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*. LNCS, vol. 3385. Springer, 363–379.

GUTTMAN, J. D., HERZOG, J. C., RAMSDELL, J. D., AND SNIFFEN, B. T. 2005. Programming cryptographic protocols. In *Trusted Global Computing (TGC'05)*. LNCS, vol. 3705. Springer, 116–145.

IBM Corporation 2006. *IBM WebSphere Application Server*. IBM Corporation. At `http://www.ibm.com/software/websphere/`.

KLEINER, E. AND ROSCOE, A. W. 2004. Web services security: A preliminary study using Casper and FDR. In *Automated Reasoning for Security Protocol Analysis (ARSPA 04)*.

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

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

MERRO, M. AND SANGIORGI, D. 1998. On asynchrony in name-passing calculi. In *Automata, Languages and Programming (ICALP'98)*. LNCS, vol. 1443. Springer, 856–867.

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

Microsoft Corporation 2005. *F#*. Microsoft Corporation. Project website at `http://research.microsoft.com/fsharp/`.

Microsoft Corporation 2006. *Windows Communication Foundation (WCF)*. Microsoft Corporation. At `http://wcf.netfx3.com/`.

Microsoft Corporation 2007. *FS2PV: A Cryptographic-Protocol Verifier for F#*. Microsoft Corporation. Project website at `http://research.microsoft.com/projects/samoa/`.

MILNER, R. 1992. Functions as processes. *Mathematical Structures in Computer Science 2,* 2, 119–141.

MILNER, R. 1999. *Communicating and Mobile Systems: the π-Calculus*. CUP.

MULLER, F. AND MILLEN, J. 2001. Cryptographic protocol generation from CAPSL. Tech. Rep. SRI–CSL–01–07, SRI.

NEEDHAM, R. AND SCHROEDER, M. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM 21,* 12, 993–999.

OASIS 2004. *Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)*. OASIS. OASIS Standard.

O'SHEA, N. 2006. Elyjah: A security analyzer for Java implementations of communications protocols. Fourth year project report, Computer Science, Division of Informatics, University of Edinburgh. Feb.

OTWAY, D. AND REES, O. 1987. Efficient and timely mutual authentication. *Operation Systems Review 21,* 1, 8–10.

PERRIG, A., SONG, D., AND PHAN, D. 2001. AGVI – automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*. LNCS. Springer, 241–245.

POZZA, D., SISTO, R., AND DURANTE, L. 2004. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*. Vol. 1. 400–405.

SUMII, E. AND PIERCE, B. C. 2001. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*. 256–269.

SUMII, E. AND PIERCE, B. C. 2004. A bisimulation for dynamic sealing. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*. 161–172.

W3C 2003. *SOAP Version 1.2*. W3C. W3C Recommendation.

W3C 2004. *Web Services Addressing (WS-Addressing)*. W3C. W3C Member Submission.

WOO, T. AND LAM, S. 1993. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*. 178–194.

Received October 2007.