Crypto-Verifying Protocol Implementations in ML

Karthikeyan Bhargavan$^{1,2}$ Ricardo Corin$^{1,3}$ Cédric Fournet$^{1,2}$
$^1$ MSR-INRIA Joint Centre $^2$ Microsoft Research $^3$ University of Twente
June 2007

Abstract

We intend to narrow the gap between concrete implementations and verified models of cryptographic protocols. We consider protocols implemented in F#, a variant of ML, and verified using CryptoVerif, Blanchet’s protocol verifier for computational cryptography. We experiment with compilers from F# code to CryptoVerif processes, and from CryptoVerif declarations to F# code. We present two case studies: an implementation of the Otway-Rees protocol, and an implementation of a simplified password-based authentication protocol. In both cases, we obtain concrete security guarantees for a computational model closely related to executable code.

1 Introduction

There has been much progress in formal methods and tools for cryptography, enabling, in principle, the automated verification of complex security protocols. In practice, however, these methods and tools remain difficult to apply. Often, verification occurs independently of the development process, rather than during early design, prototyping, and testing. Also, as the protocol or its implementation evolve, it is difficult to carry over the guarantees of past formal verification. More generally, the verification of a system that uses a given protocol involves more than the cryptographic verification of an abstract model; it may rely as well on more standard analyses of code (e.g. to ensure memory safety) and system configuration (e.g. to protect a TCB). For these reasons, we are interested in the integration of protocol verifiers into the arsenal of software testing and verification tools.

In recent work, Bhargavan et al. [2] advocate the automatic extraction and verification of cryptographic models from executable code. They verify protocol implementations written in F# [13], a dialect of ML [10, 11], by compilation to ProVerif [3]. Their approach relies on sharing as much code as possible between implementations and models: their code differs mostly in the implementation of core cryptographic libraries, which use bitstrings for concrete execution and symbolic terms for verification. (Symbolic “Dolev-Yao” cryptography is conveniently coded in ML as pattern matching on algebraic data types.)

In this work, we explore a similar approach to extend the benefit of computational cryptographic verification to protocol implementations. For automated verification, we rely on CryptoVerif, Blanchet’s recent tool for concrete cryptography [5, 6]. (We refer to these papers for an explanation of CryptoVerif syntax and semantics.) We present experimental results based on simple protocol implementations in F# for the Otway-Rees protocol and for a simplified password-based authentication protocol. In both cases, we obtain computational verification results for executable code. On the other hand, although we discuss their design, we have not yet developed general, automated translations between ML and CryptoVerif.
The diagram below outlines our proposed architecture:

---

We cross-compile between F# and CryptoVerif, as follows. In Section 2, given a CryptoVerif file crypto.cv that declares cryptographic operations and assumptions, we generate two files: an F# module interface crypto.fsi that exposes the cryptographic operations; and a symbolic module implementation crypto-a.fs of this interface that supports the equations of crypto.cv.

In addition, we write by hand a concrete module implementation crypto.fs that calls .NET implementations of standard algorithms. In Section 3, given an F# protocol implementation protocol.fs written against crypto.fsi and a CryptoVerif file query.cv that describes target computational security goals, we generate a CryptoVerif script protocol.cv. We can then "execute" our protocol in three different ways:

- run CryptoVerif on protocol.cv, crypto.cv, and query.cv to verify our target properties;
- compile protocol.fs with crypto-a.fs and run the protocol symbolically for testing;
- compile protocol.fs with crypto.fs and run the protocol over some network.

To illustrate our approach, we implement the Otway-Rees protocol (Section 4) and a protocol for password-based authentication (Section 5); the corresponding source files are available at http://msr-inria.inria.fr/projects/sec/fs2cv/. We conclude in Section 6.

2 From Cryptographic Assumptions to F#

Compared to symbolic models, computational models adopt a more concrete, less optimistic approach to cryptography; they are also more complicated and typically involve probabilistic polynomial-time semantics. In contrast to symbolic models, where the adversary can essentially perform the same computations as ordinary protocol participants, computational models specify both minimal positive assumptions (guaranteeing, for instance, that the correct decryption of an encrypted message yields the original plaintext) and minimal negative assumptions (guaranteeing, for instance, that for a particular usage of encryption, a probabilistic polynomial adversary cannot distinguish between two encrypted values).

In CryptoVerif scripts, cryptographic assumptions are introduced through type and function declarations, equations, inequalities, and game-based equivalences. We design a compiler that translates the declarations and equations in a CryptoVerif file crypto.cv into an F# interface crypto.fsi and its symbolic implementation crypto-a.fs.

Generating the F# Interface. Each CryptoVerif type declaration in crypto.cv is compiled to an abstract type in crypto.fsi and a function that generates constant values of this type. For example, `type host[bounded]` (representing host names) is compiled to the F# type declaration `type host` and the function declaration `val constHost : string -> host`. In addition, for each generative type (declared using the fixed annotation), the interface also declares a function that
generates fresh values of this type. Hence, type nonce[fixed, large] (representing nonces) is compiled to an additional function declaration val newNonce : unit -> nonce.

Compiling function declarations is straightforward, with only a slight change in syntax from CryptoVerif to F#. For example, fun enc (blocksize, key) : blocksize is compiled to val enc : (blocksize \times key) -> blocksize. For functions declared as decomposable, the compiler additionally generates an inverse function. Hence, the CryptoVerif constructor concat4:

fun concat4 (nonce, nonce, host, host) : blocksize [compos]

is compiled to the two functions:

val concat4 : (nonce \times nonce \times host \times host) -> blocksize
val iconcat4 : blocksize -> (nonce \times nonce \times host \times host)

Generating a Symbolic Model. The compiler also generates an F# module crypto-a.fs that symbolically implements crypto.fsi. This symbolic implementation uses ML datatypes and pattern-matching to model cryptographic operations. It may rely on several pi-calculus primitives: for communication (Pi.send, Pi.recv), for logging events (Pi.log), and for generating fresh values (Pi.name). For example, the symbolic implementation of a nonce type is a pi-calculus name, and the newNonce function uses Pi.name to generate a fresh nonce:

type nonce = N of Pi.name
let newNonce () : nonce = N (Pi.name "nonce")

Functions that have no equations constraining them are written as constructors of their result types, while functions defined through equations are written using pattern-matching. For example, the symbolic model generated for symmetric encryption is as follows:

type blocksize = Enc of blocksize \times key | ...
type key = KGen of keyseed | ...
let enc (b, k) = Enc(b, k)
let dec (e, k) = match (e, k) with
  | (Enc(m, KGen(r)), KGen(r')) when r = r' -> m
  | _ -> failwith "decrypt failed"

Pattern-matching is also used to define inverse functions such as iconcat4.

Writing a Concrete Implementation. Our concrete implementation crypto.fs uses the .NET cryptography libraries. For example, the symmetric encryption function enc is defined using the System.Security.Cryptography.RijndaelManaged class that implements AES encryption. The choice of a library implementation that matches the cryptographic assumptions expressed in crypto.cv should be carefully reviewed by a cryptographer. However, for a given set of cryptographic assumptions, this review needs to occur only once: the concrete implementation may then be used as a library module over and over again.

3 From Protocol Implementations to CryptoVerif

We write protocol implementations as F# modules that use the cryptographic module crypto.fsi and additional libraries, say for networking and events. The symbolic implementations of libraries may use the pi-calculus primitives; however their concrete implementations and the protocol modules do not, instead they rely on .NET libraries. We design a compiler that translates protocol modules along with symbolic library implementations to CryptoVerif scripts. The core of our translation is the same as in [2]: functions are translated to processes and algebraic datatypes are translated to constructor functions.

Functions as Processes. As a first step, the compiler flattens all modules (with suitable renaming of functions and values), inlines all non-recursive functions in the protocol code,
and then eliminates all functions that are both unused and do not appear in any interface. It then translates each remaining function to a CryptoVerif process. As a simple example, the function \( \text{let } f \ x = x \) is compiled as a process \( \text{let } f = \in (\text{callf, } x); \text{out}(\text{resultf, } x) \) where callf and resultf are public channels (so that the opponent may call the function as an oracle). The pi-calculus primitives are translated specially: \( \text{Pisend} \) and \( \text{Pirecv} \) compile to message sending (\( \text{out} \)) and receiving (\( \text{in} \)); \( \text{Pilog} \) triggers an event; \( \text{Piname} \) compiles to fresh name generation (\( \text{new} \)).

**Algebraic Datatypes.** For each algebraic datatype, the compiler generates a CryptoVerif type with decomposable constructors; hence, the implementation of a datatype is always exposed to the opponent. For example, the type \( \text{type } t = \text{of } \text{blocksize} \) is compiled to type \( \text{type } t \{ \text{bounded} \} \) and constructor \( \text{fun } A(\text{blocksize}): t \{ \text{compos} \} \).

**Top-level Process.** For each value definition \( \text{let } x = e \) in the protocol code, the compiler generates a process context that evaluates the expression \( e \) and binds the variable \( x \). The top-level process representing the full system thus consists of bindings for all value definitions and \( N \) replicas of each function process, where \( N \) is a security parameter.

**Assembling the CryptoVerif script.** The full CryptoVerif script consists of the type declarations, function declarations and processes generated above, the cryptographic assumptions in crypto.cv, and the security goals in query.cv. Typical security goals include authentication properties, expressed as (non-)injective correspondences between events, and strong secrecy properties of keys and payloads.

### 4 Otway Rees

The Otway-Rees protocol [12] is a classic key-distribution protocol, in which a server \( S \) distributes a session key between \( A \) and \( B \). It is included as a sample protocol in the CryptoVerif distribution. The protocol has four messages; we detail only the first message:

\[
A \rightarrow B : \quad M \| A \| B \| \text{enc}(N_a \| M \| A \| B, K_{as})
\]

where \( \| \) stands for concatenation and \( \text{enc} \) is the symmetric encryption function specified in crypto.cv. The F# implementation for the role \( A \) computes the first message as follows:

```fsharp
let Na = newNonce() in
let cab = concat3 M A B in
let eal = enc (concat2(Na, cab)) Kas in
Net.send AB (cab, eal);
```

where concat2, concat3 are concatenation functions, and \( AB \) is a connection over a public network intended for communications between \( A \) and \( B \). The CryptoVerif code generated from this excerpt is:

```plaintext
new Na : nonce;
let cab = concat3(M, A, B) in
let eal = enc(concat2(Na, cab), Kas) in
out((net, (cab, eal)));
```

Here, the nonce \( N_a \) is sampled uniformly at random in the \( \text{nonce} \) type. The key \( K_{as} \) is bound at the top level process as the result of running the key generation algorithm with a fresh seed. All communications occur over a single global channel \( net \).

For the generated script, CryptoVerif proves secrecy of the established key and mutual authentication between \( A \) and \( B \) for any polynomial number of instances of the protocol.
5 Password-Based Authentication

As a second example, we study the implementation of a simplified password-based authentication protocol (inspired by a web services security protocol with a richer message format [21]). In this one-message protocol, $A$ authenticates a text message to $B$ using a shared password $pwd$ and $B$'s public key:

$$A \rightarrow B : \text{text} \parallel \text{RSAEnc}(PK_B, \text{HMACSHA1}(pwd, \text{text}))$$

Hence, $A$ computes a message authentication code (MAC) of the text keyed with $pwd$ and then, to prevent offline dictionary attacks on the possibly guessable weak secret $pwd$, $A$ encrypts the MAC under $B$'s public encryption key $PK_B$.

Verifying authentication. In query $cv$, we specify the desired authenticity of text as a correspondence between two events in the F# code—a begin event just before $A$ sends the message and an end event after $B$ decrypts and checks the MAC. If we assume that $pwd$ is a strong key, CryptoVerif finds a proof of the correspondence.

Verifying weak secrecy. Even if $pwd$ is only a weak secret, we would still like to protect it against offline guessing attacks [1, 4, 8, 7]; this motivated the MAC encryption in this protocol. To this end, we disable $B$'s code and add a secrecy goal to query $cv$ requiring that $A$'s code in isolation preserves the secrecy of $pwd$. CryptoVerif automatically finds a proof of strong secrecy for $pwd$, guaranteeing the absence of offline guessing attacks.

In contrast, the protocol does not protect $pwd$ against online guessing attacks. In particular, if $B$ performs some visible action after accepting a message, the opponent can guess a $pwd'$ and then verify the guess by sending

$$I(A) \rightarrow B : \text{text}' \parallel \text{RSAEnc}(PK_B, \text{HMACSHA1}(pwd', \text{text}'))$$

If $B$ visibly accepts the message, the opponent then infers $pwd = pwd'$. Indeed, if our generated code for $B$ is enabled, CryptoVerif fails to prove the strong secrecy of $pwd$.

6 Conclusions and Future Work

Although our initial results are encouraging, numerous difficulties remain. Reflecting the specificity of the underlying cryptographic games, CryptoVerif seems more sensitive to the code structure than symbolic tools; this may hinder the direct verification of production code, and require preliminary code transforms. Also, for larger protocols, it is necessary to (safely) erase code that is irrelevant to the proof of a given property. More theoretically, it would be interesting to characterize our target security properties in F#.

After implementing the compilers outlined in this paper, we would like to consider more serious case studies, such as a reference implementation of a standard protocol. We would also like to develop verified cryptographic libraries in F#, in order to encapsulate the standard usage of selected cryptographic primitives.

Acknowledgements. Many thanks to Bruno Blanchet for his help in applying CryptoVerif to the password-based authentication protocol, and to Andy Gordon for his comments.

References


