

# Cryptographically Verified Implementations for TLS

Karthikeyan Bhargavan  
Cédric Fournet

Microsoft Research, Cambridge  
{karthb, fournet}@microsoft.com

Ricardo Corin  
Eugen Zălinescu

MSR-INRIA Joint Centre, Orsay  
{ricardo.corin, eugen.zalinescu}@inria.fr

## ABSTRACT

We intend to narrow the gap between concrete implementations of cryptographic protocols and their verified models. We develop and verify a small functional implementation of the Transport Layer Security protocol (TLS 1.0). We make use of the same executable code for interoperability testing against mainstream implementations, for automated symbolic cryptographic verification, and for automated computational cryptographic verification. We rely on a combination of recent tools, and we also develop a new tool for extracting computational models from executable code. We obtain strong security guarantees for TLS as used in typical deployments.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification;  
C.2.0 [Computer-Communication Networks]: Security and Protection

## General Terms

Security, Verification

## 1. VERIFYING PROTOCOLS AND IMPLEMENTATIONS

There has been much recent 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 design, prototyping, and testing. Also, as a protocol or its implementations evolve, it is difficult to carry over security guarantees from past formal verification. Moreover, 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 enforce policy). For these reasons, we are interested in the integration of modern cryptographic verifiers into the arsenal of software testing and verification tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.  
Copyright 2008 ACM 978-1-59593-810-7/08/10 ...\$5.00.

*Symbolic vs Computational Cryptography.* Two complementary approaches have been successfully applied to protocol verification.

Symbolic models essentially treat cryptographic primitives as black boxes and focus on the logical properties of the protocol, as pioneered by Needham and Schroeder [1978] and formalized by Dolev and Yao [1983]. They have led to efficient automated tools [e.g. Blanchet, 2001], widely applied to the verification of large protocols.

Computational models tackle more concretely cryptographic assumptions; they treat primitives as probabilistic algorithms over concrete bitstrings, and reason about the advantage of an adversary with bounded computational capabilities. Computational models sometimes lead to long, delicate, hand-crafted proofs; automated tools are much more recent [Blanchet, 2006].

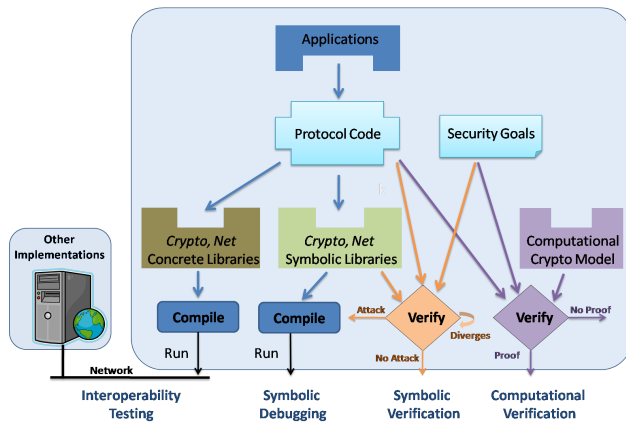
*Implementations vs Abstract Models.* Protocol specifications include many details; most (but not all) of them are of no importance for security. In the process of distilling a formal cryptographic model, most of these details are discarded. When is a protocol model oversimplified? In contrast with formal guarantees proved within the model, the relevance of the model relies on the experience of the formalist. The problem is compounded when considering protocol implementations. Thus, as far as possible, we propose to verify detailed protocol implementations and deployments, rather than simplified abstract models. Using automated tools based on sound proof techniques, the details can either be safely erased, or dealt with by brute-force analysis.

*From Implementations to Cryptographic Models.* More recent works [Goubault-Larrecq and Parrennes, 2005, Bhargavan et al., 2006] advocate the automatic extraction and verification of symbolic cryptographic models from executable code.

Bhargavan et al. [2006] verify protocol implementations written in F# [Syme, 2005], a dialect of ML, by compilation to symbolic models in ProVerif [Blanchet, 2001]. Their approach is to verify as much code as possible, while providing hand-written models for the rest, such as the core cryptographic libraries, which use bitstrings for concrete execution and symbolic terms for verification.

In this work, we rely on their tools for symbolic verification, and also experiment with direct computational cryptographic verifications of protocol implementations, by compilation to CryptoVerif, a recent tool for computational cryptography [Blanchet, 2006].

*Our Approach.* The picture below outlines our approach to protocol verification. It involves developing, testing, and verifying a small reference implementation of the protocol plus typical applications. In contrast with specialized modelling languages for protocols, our use of a standard development platform enables early testing, for instance to disambiguate the specification, experiment with potential attacks, and confirm functional correctness.



Verification consists of selecting a part of the implementation, writing additional “verification harness” code that specifies the attacker model, the cryptographic assumptions, and the target security properties, and then compiling their combination to some automated prover. Inasmuch as the verification tool chain is automated, one can easily re-verify the code base as it evolves.

In our experience, symbolic and computational verifications are complementary. Computational verification is more precise but more difficult to achieve; we obtain results only for the cryptographic core of the protocol implementation. Symbolic verification typically applies to the whole protocol, sometimes even including the application, but does not detect low-level cryptographic errors. Overall, we believe that the overhead of verification is getting affordable, in comparison with design, development, and testing. The next generation of tools could enable their integration in the development process.

**Implementing & Verifying TLS.** As an extended case study, this paper considers implementations of TLS 1.0, one of the most widely deployed communications protocols. Due to its popularity, many systems embed an implementation of TLS and rely on its security for communications.

As well as being of practical importance, TLS is a well-understood protocol, with a carefully written, self-contained specification, a series of successive versions, and a large body of related verification work, providing a detailed history of security vulnerabilities and improvements. Also, TLS is not an academic protocol, optimized (or designed) for verification purposes. This sometimes complicates its security analysis, but also provides a good benchmark for assessing verification techniques.

#### Contributions.

1. We program a small functional implementation of TLS. Using simple client and server code, we confirm that our implementation interoperates with mainstream implementations.
2. Relying on a combination of model-extraction and verification tools, we obtain a range of positive security results, covering both symbolic and computational cryptographic aspects of the protocol. We thus provide security guarantees for code as it is used in typical deployments of TLS.
3. To support computational verification, we develop a new tool for extracting cryptographic models from F# code. We believe this enables the first automated verification of executable code against standard cryptographic assumptions.
4. We review known weaknesses for various versions of TLS, and discuss their detection as part of our verification process, for the corresponding weakened implementations of TLS.

**Contents.** Section 2 recalls the main security features of TLS. Section 3 outlines our reference implementation and reports on interoperability. Section 4 presents our results using symbolic models; it also discusses formal security issues and related work. Section 5 describes a new tool for extracting computational models and relating computational security assumptions to concrete cryptography APIs. Section 6 presents our results using computational models; it also discusses cryptographic issues and related work.

An extended version of this paper and sample code appear at <http://msr-inria.inria.fr/projects/sec/fs2cv>.

## 2. TRANSPORT LAYER SECURITY

The Secure Session Layer (SSL) protocol was promoted by Netscape as a means of providing privacy over the Internet, by securing HTTP connections between web browsers and servers. Its first public version, SSL 2.0 [Hickman, 1995], was released in 1994. Its successor, SSL 3.0 [Frier et al., 1996], includes major changes and addresses serious security flaws. It then evolved into an Internet standard, named Transport Layer Security (TLS 1.0) [Dierks and Allen, 1999]. The latest standard, TLS 1.1 [Dierks and Rescorla, 2006] and the latest draft, TLS 1.2 [2008] include further improvements and clarifications, notably changes designed to thwart new cryptographic attacks. Since the three TLS versions are relatively similar, we refer to them generically as the TLS protocol(s).

To facilitate interoperability tests, our code targets mostly TLS 1.0, the latest largely deployed version of the protocol. Next, we briefly recall its main security features. We follow the notations and terminology of the RFC [Dierks and Allen, 1999]; we refer to this document for a more general presentation.

**TLS 1.0.** TLS provides secure communications between a client and a server, with certificate-based server authentication and, optionally, client authentication. The protocol distinguishes between *sessions* and *connections*; from an established session, each party can derive one or more connections, and use them to send series of messages. The protocol has two layers. The lower layer consists of the *record protocol*, for exchanging messages using current connection parameters. The upper layer includes a *handshake protocol* for establishing sessions, as well as application protocols.

**Record Protocol.** The record protocol receives uninterpreted data from the upper layer. This data is first (possibly) split and compressed, then formatted into a series of records, and passed to a lower, unprotected transport protocol.

Both parties independently maintain state for the read and write directions of the connection. Each record is protected depending on the *security parameters* negotiated by the handshake protocol, which mostly include a ciphersuite, and on the current connection states (e.g. keys and IVs). A ciphersuite specifies a key exchange mode (either Diffie-Hellman- or RSA-based), an encryption algorithm, and a hash algorithm. The encryption and hash algorithms are relevant only to the record protocol, while the key exchange mode is relevant only to the handshake protocol.

Initially, the ciphersuite is set to null, indicating no security transformations. Thus, the messages of the handshake protocol are not protected by the record protocol, until shared security parameters can be established.

After the handshake, each fragment is protected using the mac-then-encrypt technique, and prefixing the result with a record header. The record header has three fields: the content type of the sub-protocol the fragment belongs to, the version of the protocol used for processing this record, and the fragment length. The mac is computed by applying HMAC (with the hash algorithm and hash secret given by the security parameters and current state) to the

concatenation of the fragment, the record header, and the fragment sequence number. The fragment and the resulting mac are then fed to the encryption algorithm, in cipher block chaining (CBC) mode, after padding to a length that is a multiple of the block size.

**Handshake Protocol.** The handshake protocol authenticates the server, optionally authenticates the client, establishes a shared *master secret*, derives cryptographic materials for their connections, and confirms that both parties agree on their exchanged parameters. In this paper, we consider only RSA-based modes. We begin with the message flow for a handshake with an anonymous client:

```

ClientHello      ----->
                                     ServerHello
                                     Certificate
                                     ServerHelloDone
<-----
ClientKeyExchange
[ChangeCipherSpec]
Finished        ----->
                                     [ChangeCipherSpec]
                                     Finished
<-----

```

For our discussion, it is convenient to decompose the protocol into four phases, explained in more detail below.

1. the client and the server exchange connection parameters by means of the hello messages;
2. they establish an intermediate shared *pre\_master\_secret* (*pms*); when using RSA, the client chooses *pms*, so the phase consists of a single *ClientKeyExchange* message;
3. they each compute a shared *master\_secret* (*ms*); this enables the record layer to derive fresh cryptographic materials for each direction of the record protocol;
4. they exchange *ChangeCipherSpec* messages, immediately followed by *Finished* messages, to confirm that they share matching keys, check server authentication, and ensure integrity of the handshake messages.

The Hello messages include fresh nonces, a session identifier picked by the server, and session parameters; their logical content is

```

ClientHello(ver_min,ver_max,cr,rsid,cipher_suites,comp_methods)
ServerHello(version, sr, sid, cipher_suite, comp_method)

```

The *Certificate* message carries the server's X.509 certificate; the *ServerHelloDone* message has no payload.

TLS enables the negotiation of some connection parameters, that is, a protocol version, a ciphersuite, and a compression method. These parameters are passed unprotected in the Hello messages: the client expresses its preference as a range of parameters, then the server sets the session parameters within that range. The negotiation is authenticated later by the *Finished* messages, which themselves depend on these parameters. This circularity is a source of concerns for TLS, discussed in Section 4.

The client announces its highest supported version in the *ver\_max* field of *ClientHello* and includes its lowest supported version *ver\_min* in the version field of the record header that encloses *ClientHello* (provisionally using this version's record format). The server announces its choice in the version field of *ServerHello*, and also includes it in the enclosing record header.

The *ClientKeyExchange* message includes the RSA encryption of a fresh random *pms*, using the public key of the received certificate. In order to confirm the highest version supported by the client, the protocol version byte *ver\_max* is embedded in *pms*, and also influences the padding before RSA encryption:

$$pms = ver\_max \parallel random$$

where  $\parallel$  is bitstring concatenation and *random* consists of 46 random bytes.

From the *pms* and exchanged random values, both parties compute the master secret using an ad-hoc pseudo-random function (PRF). This function takes as input a secret and a seed, and generates a stream of bytes, using HMAC (with two hash algorithms, MD5 and SHA1) as base primitive. For generating *ms*, the secret is *pms* and the seed is the concatenation of a fixed bitstring and the two nonces exchanged in the hello phase:

$$ms = PRF(pms, \text{"master secret"} \parallel cr \parallel sr)$$

The materials for the record protocol are generated similarly:

$$key\_block = PRF(ms, \text{"key expansion"} \parallel sr \parallel cr)$$

is truncated and split into six secrets for the initial read and write connections: two encryption keys, two mac keys, and two IVs.

The two *ChangeCipherSpec* messages appear in brackets because they are not considered part of the handshake itself; they signal the use of the newly-negotiated algorithms and keys, so the *Finished* messages are the first to be maced-and-encrypted by the record protocol. These *Finished* messages contain a (hashed) transcript of the handshake to this point; their logical contents is

$$verify\_data = PRF(ms, finished\_label \parallel MD5(hsm) \parallel SHA1(hsm))$$

where *hsm* is the concatenation of the sequence of handshake messages (including handshake subprotocol headers, but not the outer TLS record headers). The resulting authentication guarantees are detailed in Section 4. After a successful handshake, the parties can start exchanging application data in both directions.

**Resumption Protocol.** Instead of performing a full handshake, TLS offers the possibility of resuming a previously established session, or even duplicate an existing session to derive further connections.

Assume parties have already performed a successful handshake, thus establishing a session. The client can propose an abbreviated handshake by sending an Hello message that includes a fresh nonce and the old session identifier. If the server accepts this session identifier, both parties skip phases 2–3, immediately derive fresh cryptographic materials, and exchange *Finished* messages. Thus, the message flow for the abbreviated handshake is:

```

ClientHello      ----->
                                     ServerHello
                                     [ChangeCipherSpec]
<-----
[ChangeCipherSpec]
Finished        ----->
                                     Finished

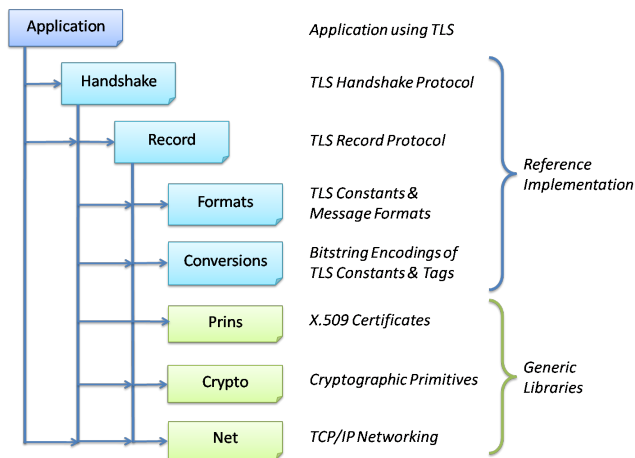
```

Otherwise, a handshake continues as in the general case.

### 3. REFERENCE IMPLEMENTATION

Our code is entirely written in F# [Syme, 2005], a dialect of ML, and executed on the .NET runtime. Its structure and programming style reflects our goal to use the same code, as far as possible, for four different tasks: symbolic execution for debugging; concrete execution for interoperability testing; symbolic verification; and computational verification.

TLS is usually implemented as a library, linked to web-based applications such as browsers or web servers. The figure below gives the structure of our reference TLS implementation; each box represents an F# module; each arrow represents a direct dependency between modules. Hence, the modules *Handshake* and *Record* implement the handshake and record protocols, respectively; and their



interfaces enable some Application module to send and receive messages over TLS. Module `Formats` contains functions to build and parse formatted TLS messages; it relies on `Conversions` for low-level encodings of strings and TLS-specific tags. Modules `Crypto`, `Net`, and `Prins` provide library functions that are not specific to TLS.

In the rest of this section, we outline the interfaces and implementations of these modules. (The full paper gives more details.)

**Libraries for Networking and Cryptography.** The module `Net` defines functions to set up and use TCP connections. For example, by calling `connect` with a URI  $u$ , a client application can create a TCP socket to a server listening at  $u$ . It can then call the functions `send` and `recv` to exchange messages on this connection. The module `Crypto` defines standard cryptographic primitives. For example, the `Handshake` module creates a fresh `pms` using `mkNonce` and encrypts it using `rsa_encrypt`, while the `Record` module calls `hmacsha1` and `aes_encrypt` to mac-and-encrypt messages. The module `Prins` (for principals) defines functions to create and retrieve X.509 certificates.

Our *concrete* implementation of these libraries relies on various classes in the .NET Framework; for instance, the `Crypto` module implements `hmacsha1` by calling the `ComputeHash` method in `System.Security.Cryptography.HMACSHA1`, and `Net` uses `System.Net.Sockets.TcpClient` to implement `connect`.

Following an approach proposed by Bhargavan et al. [2006] (see the figure in Section 1), we also develop a *symbolic* implementation of these libraries, for use in symbolic verification and debugging. In this version, the `Crypto` module models hashing and encryption as algebraic datatype constructors for an abstract type; for instance, `hmacsha1(key,text)` simply returns a term `HMACSHA1(key,text)` representing the keyed hash; `Net` models connections as communications on local channels between processes; `Prins` models the X.509 store as a local private database.

Both versions of the library modules implement the same interfaces. By compiling our reference TLS implementation and applications against the concrete libraries, we obtain an executable that can be deployed on the network and tested against remote clients and servers. By compiling against the symbolic libraries, we obtain an executable that can be used for generating symbolic traces for local debugging. For symbolic (and computational) verification, we assume that the concrete implementation of these libraries follows their models; as such, these libraries represent the trusted computing base for our verification results.

**Record Module.** The `Record` module exports two functions:

```
val send: ConnectionId → bytes → unit
val recv: ConnectionId → bytes
```

It uses a database of active connections, indexed by `ConnectionIds`. The handshake protocol populates this database with new connections as they are established.

The type `Connection` represents an established TLS connection:

```
type Connection = {
  net_conn: Net.conn;
  crt_version: ProtocolVersion;
  read: ConnectionState;
  write: ConnectionState; }
type ConnectionState = {
  cipher_state: CipherState;
  mackey: bytes;
  seq_num: int;
  sparams: SecurityParameters; }
```

It is a record type storing the underlying TCP connection `net_conn`, the version used during this connection, and the read and write connection states. Each read or write connection state is a record containing the cipher state (represented in our case, i.e. for block ciphers, by the encryption key and the current initialization vector), the key used for macing `mackey`, the current sequence number `seq_num`, and the security parameters `sparams` established by the handshake protocol (including, for example, the encryption and hash algorithms).

Given an established TLS connection with identifier `id`, the `send` and `recv` functions write and read payloads over the connection, in accordance with the record protocol. As they process messages, they log the following security events:

```
Send(id,entity,payload)
Recv(id,entity,payload)
```

where `entity` is either `Client` or `Server`. The event `Send(id,entity,payload)` logs that `entity` sends message `payload` over the connection `id`. The event `Recv(id,entity,payload)` logs that `entity` accepts message `payload` as valid over the connection `id` (after cryptographic record processing, before passing it to the application). These events have no effect at runtime; they are used only to specify our security goals for verification.

To illustrate our coding style, we detail the code for the `recv` function, which takes one argument, a connection identifier `connid`, and returns a record payload `msg`.

```
let recv (connid:ConnectionId) =
  let conn = getConnection connid in
  let conn, input = recvRecord conn in
  let conn, msg = verifyPayload conn CT_application_data input in
  let id,entity = connid in Pi.log tr (Recv (id,entity,msg));
  storeConnection connid conn;
  msg
```

The function is written as a sequence of function calls. It first calls `getConnection` to retrieve the connection record `conn`; it then calls `recvRecord`, which blocks until the next message input is received on the connection; it calls `verifyPayload` (detailed below) to decrypt the payload `msg` and verify the mac; it calls `Pi.log` to log a `Recv` event as described above; and finally calls `storeConnection` to update the connection database with the new connection parameters before returning `msg`. The cryptographic checks are all performed in the `verifyPayload` function:

```
let verifyPayload (conn:Connection) (ct:ContentType) (input:bytes)=
  let (bct, bver, blen, ciphertext) = parseRecord input in
  let rct, rver, rlen = getAbstractValues bct bver blen in
  let ver = conn.crt_version in
  if rver = ver then
    let connst = conn.read in
    let connst, plaintext = decrypt ver connst ciphertext in
    let payload, recvmac = parsePlaintext ver connst plaintext in
    let len = bytes_of_int 2 (length payload) in
    let bseq = bytes_of_seq connst.seq_num in
```

```

let maced = append5 bseq bct bver len payload in
let conn = updateConnection_read conn connst in
checkContentType ct rct payload;
if hmacVerify connst maced recvmac = true then
  (conn.payload)
else failwith "bad record mac"
else failwith "bad version"

```

The function takes three arguments: a connection record `conn`, an expected content type, and a message input received over `conn`, and returns an updated connection `conn` and the received message payload. Most of the function prepares materials for calling the two cryptographic functions `decrypt` and `hmacVerify`. The call to `decrypt` decrypts ciphertext using the algorithm, key, and initialization vector stored in the connection read state `connst`, to yield plaintext and a new `connst` with an updated initialization vector. The decrypted plaintext consists of a payload and a mac `recvmac`. The call to `hmacVerify` verifies the mac, using the algorithm and mac key in `connst`, thereby authenticating the sequence number, content type, protocol version, ciphertext length, and payload. The function fails with an exception if the mac is incorrect, if the version, content type, or sequence number do not match the expected values, if the message is an alert, or if any parsing function fails. In all other cases, it returns an updated connection state and payload.

**Coding for Verification.** As illustrated in these two functions, our subset of F# is rich enough to write modular code that accounts for detailed message formats, cryptographic operations, and security events. Our code uses typical functional language features such as nested function applications, tuples, records, algebraic datatypes, pattern matching, exceptions, and modules. However, we avoid other features such as references, higher-order functions, and classes, because they are not presently supported by our verification tools. Moreover, since every library function or operating system call that we use must be given a symbolic model, we define and use only the minimal set of libraries described above. (We do not use, for example, any file I/O operation.)

We use recursive functions sparingly, because they usually lead to non-terminating runs of ProVerif, and are difficult to verify in CryptoVerif. For example, we have a recursive list membership function for lists of publicly known elements, but no list concatenation over private data. Moreover, we tend to separate purely functional code from code that has side effects, such as events or networks operations. Purely functional code like `verifyPayload` translates to reductions in ProVerif and equations in CryptoVerif, whereas functions like `recv` translate to processes, which are more complex to verify.

**Handshake Module.** The Handshake module exports four functions that enable client and server applications to set up new sessions, resume old sessions, and close connections.

```

val connect: Net.conn → ServerName → ConnectionId * SessionId
val resume: Net.conn → SessionId → ConnectionId * SessionId
val accept: Net.conn → CertName → ConnectionId * SessionId
val close: ConnectionId → bool → unit

```

It maintains a database of active sessions indexed by `SessionIds`. The type `Session` characterizes a TLS session:

```

type Session = {
  sid: bytes;           ch: ClientHello;
  ms: bytes;           sh: ServerHello;
  server_cert: Certificate; pms: bytes; }

```

It is a record of a session identifier `sid`, the master secret `ms`, the server certificate, the client and server Hello message and the `pms`. All these fields are exchanged during the full handshake which established the session. The fields in the left column suffice to run the protocol; the other fields are included only for the security analysis.

A server calls `accept` to listen on a TCP connection for a TLS connection request; when a client calls `connect` over the same TCP connection, the client and server engage in the full handshake protocol to establish a new session and a new connection in each direction. Upon completion of the handshake protocol, both `accept` and `connect` construct their own `Session` record and `Connection` record and populate them with all the values authenticated by the handshake protocol, including, for instance, the session identifier, ciphersuite, security parameters, and computed keys. To indicate the completion of the protocol and agreement on these values, the two functions log the following event

```
SendFinished(id,entity,subject,pms,session,read,write)
```

and the `AcceptFinished` event with the same parameters. Each event contains the connection id, the entity which logs it, the subject of the server's certificate, the `pre_master_secret`, a full `Session` record, and read and write connection states for the `Connection` records in the indicated direction.

Clients can call `resume` for such sessions to trigger the resumption protocol. Upon its completion, the `resume` function logs the following event:

```
SendFinishedRes(id,entity,subject,pms,session,read,write,ch,sh)
```

and the `AcceptFinishedRes` event with the same parameters, which have the same meaning as for the full handshake. In addition the client and server Hello records are tracked. Note that all fields in `session` (including for example `session.ch` and `session.sh`) refer to the initial full handshake, while the last two parameters `ch` and `sh` refer to the Hello messages in the abbreviated handshake. The servers executing `accept` also log these two events if a resumption protocol has been completed.

**Sample Applications and Interoperability.** Using our reference implementation, we write three applications:

- a client that connects to an arbitrary HTTPS URI and retrieves a web page over a TLS connection;
- a server that listens at an HTTPS URI and serves up a single web page;
- a mutually authenticated client-server application where the client authenticates to the server using a password over a TLS connection.

In our experiments, our client application can access basic pages from a variety of web servers running IIS or Apache. Our server application successfully serves pages to clients running Internet Explorer or Firefox. Both applications successfully resume sessions, when triggered for instance to refresh a web page displayed in a previous session. Hence, we experimentally establish that our reference implementation is interoperable with the mainstream TLS implementations used by these browsers and web servers, including OpenSSL and the Windows CryptoAPI. Moreover, as our client-server application demonstrates, applications with additional security properties can be built on top of our TLS implementation.

In comparison to mainstream implementations, our reference implementation supports a smaller subset of the standard. We focus on TLS 1.0 in RSA mode for the key exchange. In this mode, we support all ciphersuites using AES, DES, RC4, SHA, and MD5 algorithms. Our implementation does not support data compression, nor fragmentation; it does not send alerts and fails silently upon receiving a bad message. Despite these limitations, our experiments show that it is adequate for writing simple client and server applications. We find that having an implementation to experiment with helped to understand disambiguate details of the specification. It also enabled us to try out attacks on our and other implementations.

## 4. SYMBOLIC VERIFICATION

Our symbolic verification is based on an existing tool chain consisting of a model extractor [Bhargavan et al., 2006], that compiles code written in F# to process models in an applied pi calculus [Abadi and Fournet, 2001], and the state-of-the-art verifier ProVerif, which analyzes such models automatically. For many protocol implementations, the verifier either proves the security goals or produces a counter-example. In some cases, however, the analysis may not terminate; in others, it may take several gigabytes of memory. To use this tool chain, we write symbolic implementations for low-level libraries as described in Section 3, we define the attacker model in terms of the interface exposed by these libraries and by our reference implementation, and we write our authentication and secrecy goals as correspondence between events logged by functions in the interface. Then we can extract a symbolic model from the reference implementation and verify security queries automatically. If the tool proves a query, we obtain a security theorem about the protocol implementation, against all attackers that use its interface. Our results rely on the correctness of the core translations and algorithms underpinning the model extractor and ProVerif [Blanchet, 2001, Bhargavan et al., 2006].

**Attacker Model.** The attacker capabilities are given by the interfaces of the modules: Net, Crypto, Prins, Handshake, and Record. Access to the functions given in these interfaces yields a (standard) symbolic threat model, where the attacker can

- control the network (Net), and perform cryptographic operations (Crypto),
- create any number of servers by generating certificates and compromise any server by reading its private key (Prins),
- open arbitrarily many sessions between clients and servers of its choice, obtaining connection and session ids, and trigger the resumption protocol for any session id (Handshake),
- send and receive messages over the record layer (Record).

Let *Sys* denote the program consisting of the symbolic implementations of the libraries Net, Crypto, Prins, and Conversions, along with our reference implementations for Handshake, Record, and Formats. We prove authentication and secrecy properties for this full program, although in the following, we describe the verification results separately for each part of the protocol.

**Handshake Protocol.** We first present symbolic authentication results for the handshake protocol. *Authentication* is specified as a correspondence from events triggered when a party accepts the peer’s Finished message to prior events triggered when the party sends that message. The more information these events record, the stronger the property. We say that a server has been corrupted when its private key has been leaked to the attacker. We say that a client is corrupted if its pms is known to the attacker.

**THEOREM 1 (FULL HANDSHAKE AUTHENTICATION).**

*In any run of Sys, for any AcceptFinished event, either there is a SendFinished event with opposite entities (client/server or server/client) and matching connection and session parameters, or one of the entities is corrupted.*

(The precise ProVerif queries for all theorems in this section are included in the full version.) Here, almost all fields of the session records at the client and server are correlated. These include sid, pms, ms, server\_cert, and also cr, sr, version, and cipher\_suite from ch and sh in session. The derived cryptographic materials (within the read and write connection states, from the client’s viewpoint)

are also correlated. These variables provide complete coverage of the current state of both parties as they switch to the application protocol, as well as additional parameters used in the negotiation.

As a minor, technical point, there is no formal agreement on the client’s lowest supported version for TLS (*ver\_min*): the adversary may change this value, together with the first message record format, in ClientHello, without detection. However, this seems innocuous as long as (1) the client checks that version  $\geq$  *ver\_min* and (2) whenever the server accepts the first message, version does not depend on *ver\_min*.

**Resumption Protocol.** We obtain a similar authentication theorem for the resumption protocol.

**THEOREM 2 (RESUMPTION AUTHENTICATION).** *In any run of Sys, for any AcceptFinishedRes event, either there is a SendFinishedRes event with opposite entities (client/server or server/client) and matching connection and session parameters, or one of the entities is corrupted. Moreover, within each AcceptFinishedRes and SendFinishedRes event, the new ServerHello message has the same session id and ciphersuite as the old session.*

We also prove secrecy queries for all the secrets generated during the handshake, including the pms, ms, and all four keys. These queries assert *syntactic secrecy*; they show that the secret values are not obtained by the attacker, unless he has compromised the server or controls the client. For brevity, we omit the formal security theorem which can be found in the full version.

**Record Protocol.** For the record protocol, authentication is specified as a correspondence between the Recv events emitted when a receiver accepts a message to prior Send events emitted when messages are sent.

**THEOREM 3 (RECORD AUTHENTICATION).** *In any run of Sys, for any Recv event, either there is a Send event with opposite entities (client/server or server/client) and matching connection identifiers and records or one of the entities is corrupted.*

In addition, we prove the syntactic secrecy of record payloads, when the payloads are freshly generated values. As usual, the payload is secret only if the server is uncorrupted and the client is not controlled by the attacker. The formal secrecy theorem appears in the full paper.

**Reconstructing Known Attacks on TLS.** Previous formal and informal analyses of SSL and TLS have uncovered a range of vulnerabilities and attacks. For instance, SSL 2.0 does not guarantee integrity of many elements of the handshake negotiation, including the ciphersuite. Hence, if both client and server prefer to use strong cryptography but also allow weak cryptography, then an attacker may convince them both to establish a session with weak cryptography. In our model, implementations of SSL 2.0 do not satisfy handshake authentication (Theorem 1). Our tools fail to prove our authentication queries for such implementations, and instead generate counter-examples indicating the attack.

Recent versions (since SSL 3.0) provide more extensive integrity guarantees for the handshake. Still, Wagner and Schneier [1996] found several attacks on the handshake and resumption protocols of SSL 3.0. They found that if both parties also support SSL 2.0 for backward compatibility, then *version rollback* attacks become possible: an attacker can convince them to use SSL 2.0, and then exploit any cryptographic flaws of the earlier version. TLS includes two mechanisms to address this problem, both in the ClientKeyExchange message, so that a server can identify SSL-only clients. For instance, TLS clients embed *ver\_max* within pms, but SSL 2.0 clients do not.

However, these mechanisms still do not suffice with the resumption protocol, since the abbreviated handshake does not contain the ClientKeyExchange message. Hence, resumption authentication (Theorem 2) guarantees only that the new connection parameters *excluding* ServerHello.version are correlated with the old session parameters. Experimentally, we found that deployed server implementations of TLS are vulnerable to this version rollback attack from TLS 1.0 to SSL 3.0 during resumption. However, we did not find the more dangerous rollback from TLS 1.0 to SSL 2.0, partly because the length of the session id parameter fortunately differs between these two versions.

Our method also catches common errors in TLS implementations, such as not verifying server certificates, or not checking that the received sequence number is correct. Such errors result in counter-examples to our authentication queries.

On the other hand, it is worth pointing out that several well-known attacks on TLS are outside the scope of our symbolic (and computational) model. These include cryptanalyses on the underlying cryptographic functions, traffic analyses, and padding attacks.

**Previous Symbolic Analyses.** In a long line of works, researchers have used various techniques to verify models, and more recently, implementations, of different versions of SSL and TLS. Here, we describe only those most closely related to our work.

Mitchell et al. [1998] study a model of SSL 3.0 using the Murphi tool. They use model-checking to perform a finite-state exploration of a sequence of simple protocols with increasing complexity, including a version of SSL 3.0 with both handshake and resumption protocols, but limited to finite configurations consisting of, for example, two clients and a server.

Paulson [1999] develops formal, machine-checked proofs for a model of TLS 1.0 in Isabelle, with authentication and secrecy theorems that, like ours, apply to more general configurations of clients and servers. His model includes both handshake and resumption but does not address version rollback issues within resumption.

He et al. [2005] apply logic-based proof techniques to the IEEE 802.11i protocol, and include a simple model of TLS as a subprotocol. Using PCL, they prove agreement on all exchanged messages and secrecy of the pre-master secret.

Ogata and Futatsugi [2005] show secrecy of the pre-master secret and liveness properties for the handshake protocol with resumption using the OTS/CafeOBJ tool.

Kamil and Lowe [2008] report on an analysis of a detailed strand spaces model of the handshake and record protocols. They prove authentication and secrecy theorems similar to ours, and also show that the record protocol provides two authenticated streams and satisfies session independence.

Jürjens [2006] verifies a Java implementation of the TLS handshake protocol for secrecy and authentication properties. His analysis works on the control-flow graph and does not account for multiple versions or low-level message formats.

Chaki and Datta [2008] apply software model checking on OpenSSL code to verify secrecy and authentication for configurations of up to three servers and clients.

## 5. A COMPUTATIONAL VERIFIER FOR PROTOCOL IMPLEMENTATIONS

This section describes our computational verification approach and tools; Section 6 applies them to TLS. Compared with symbolic models, computational models adopt a less optimistic approach to cryptography: rather than giving the adversary essentially the same capabilities as ordinary protocol participants, they specify both minimal positive assumptions (guaranteeing, for instance,

that the correct decryption of an encrypted message yields the original plaintext) and minimal negative assumptions (bounding, for instance, the probability that a polynomial adversary may break a particular usage of encryption).

### 5.1 CryptoVerif (Review)

The CryptoVerif verifier can prove the security of a given protocol under a set of security assumptions for its cryptographic primitives, within a probabilistic polynomial-time (PPT) model of computation. We briefly present the tool; we refer to Blanchet [2006], Blanchet and Pointcheval [2006] for an explanation of CryptoVerif syntax and semantics.

CryptoVerif takes as input a script, written in a variant of the pi calculus with an explicit polynomial bound for every replicated process. Thus, processes represent PPT Turing machines that exchange finite bitstrings through an adversary, modelled as an (unknown) PPT machine. In the script, cryptographic assumptions are introduced through type and function declarations, equations, inequations, and game-based equivalences. The equations and inequations are typically used to describe minimal positive assumptions (the functional correctness of the primitive), whilst the game-based equivalences are used to state minimal negative assumptions. Section 6 gives some examples.

**Proofs, Games, and Indistinguishability.** The input script can be seen as an initial game, modelling the protocol, to which CryptoVerif applies transformations, until a final game that satisfies target security conditions is reached—this proof technique is known as *game-hopping*.

Each transformation between two consecutive games preserves PPT indistinguishability, that is, the adversary cannot distinguish the games before and after the transformation. Example transformations include the application of an equivalence stating the security of a cryptographic primitive, and the semantics-preserving rearrangement of code, such as inlining and partial evaluation. CryptoVerif runs either automatically or interactively, in which case it receives guidance from the user for selecting transformations.

**Process and Variable Instances.** Processes in CryptoVerif can be replicated polynomially in a given security parameter, enabling multiple parallel executions. A special find command permits to access the different variable instances of each process replica.

**Target Security Properties.** CryptoVerif can deal with authentication and secrecy properties, specified as follows.

Authentication is expressed using correspondences, much as in our symbolic models. Correspondences typically assert that, if some event is executed, then other events must also have been executed, with matching parameters, *at least with overwhelming probability*; this last bit reflects the computational nature of the model.

Secrecy is expressed using indistinguishability between two configurations. (It is often called *strong* secrecy in symbolic models, in contrast with the weaker notion of syntactic secrecy.) CryptoVerif has two notions of secrecy. The weaker notion (query `secret1` in CryptoVerif) states that the adversary cannot distinguish the value of a variable in a specific instance from a random value; the stronger notion (query `secret`) states that the adversary cannot distinguish the sequence of all the values of variable instances from a sequence of independent random values.

### 5.2 Compiling to CryptoVerif Scripts

We describe the design and implementation of a new model extractor that translates protocol implementations written in F# to CryptoVerif scripts. The extractor takes three inputs: protocol modules written in F#, such as the modules in our reference implementation, a computational model of the cryptographic libraries expressed as

CryptoVerif assumptions, and security goals for the protocol expressed as CryptoVerif queries. Given these inputs, it generates a CryptoVerif script that can be verified either automatically or interactively. In the rest of this section, we outline the various steps of the translation.

**Computational Models for Libraries.** We first define models for all the functions in the library modules, such as `Net`, `Crypto`, and `Prins`. Our model of `Net` treats connections as public channels; hence, calls to `Net.send` and `Net.recv` send and read messages from a single public channel that is controlled by the attacker. Our model of `Crypto` defines cryptographic primitives as uninterpreted functions in CryptoVerif. For each primitive, we then add equations, inequations, and equivalences to encode the specific cryptographic notions we wish to use. Functions for generating fresh values, such as `mkNonce`, are written using the CryptoVerif primitive `new` that chooses a random bitstring uniformly from a type, such as the set of all nonces. Our model of `Prins` maintains a private array of public-private keypairs; it allows a polynomial number of such keypairs but does not model key compromise. In contrast with `Net` and `Prins`, which are generic, `Crypto` must be written specifically for the protocol at hand; the specific definitions used for TLS are described in Section 6.

**Code Transformations.** The model extractor applies a series of code transformations to generate a smaller, more specialized source program. These transformations include aggressive inlining of non-recursive functions, partial evaluation of functions and patterns, and dead-code elimination. Data structures such as records are translated to simpler forms such as tuples, and all type abbreviations are inlined. All functions that do not appear in the interfaces are eliminated, and all modules are flattened into a single module by suitably qualifying the names of functions, variables, and types. This single module then consists of datatype definitions, function definitions, and top-level code that evaluates expressions and binds variables.

**Algebraic Datatypes.** For each algebraic datatype, the translation produces a CryptoVerif type declaration with transparent constructors, thereby reflecting our assumption that constructors are used only for tagging data, not for hiding information. For instance, the F# declaration `type t = A of bytes` yields a CryptoVerif type `t` and a type constructor `A` that is declared to be invertible; hence it is always possible to obtain the bitstring `x` from `A(x)`.

**Functions as Processes.** For each function definition `let f x = e`, the translation first transforms the body expression `e` in continuation-passing style, into a sequence of imperative commands `e'`: each line in `e'` is either a function call or a pattern match. Each line is then translated to CryptoVerif: function calls become processes that call CryptoVerif function symbols; pattern matches become `let` processes. Some function calls are specially translated: calls to fork spawn parallel processes; calls to log yield primitive event recording processes. Finally, the whole function definition is translated to a process of the form `let f = in(callf, x);...; out(resultf, r)` that takes its arguments on channel `callf` and returns its result on channel `resultf`. Since these channels are public, the opponent may call any of the functions in the public interface, as oracles.

**Top-level Process.** Each variable binding `let x = e` in the source code translates to a process context that binds `x` to the result of evaluating `e`. The expression `e` is translated to a process, as expressions in function definitions, possibly spawning processes using fork. Hence, the top-level process that represents the full system consists of bindings for all variables, parallel threads for all spawned processes, and `N` replicas for each function process, where `N` is a polynomial in the security parameter.

**Verification.** The full CryptoVerif script consists of the computational models of the libraries, the type definitions in the protocol implementation, and the top-level process representing the oracle interface provided by the implementation to the attacker. We then write security goals as CryptoVerif queries for this process, and proceed with verification.

Besides the scripts obtained from our reference TLS implementation (Section 6), our largest case-study so far, we have extracted CryptoVerif scripts from the code of several sample protocols, including the Otway-Rees protocol and a password-based authentication protocol; we could verify both authentication properties, expressed as non-injective correspondences between events, and strong secrecy properties for keys and payloads. Although all our scripts are currently automatically verified by CryptoVerif, manual guidance may be required in general, in the form of advice.

**Challenges (Future Work).** We mention two challenges with script extraction. First, as detailed in Section 3, libraries such as `Prins` rely on private databases to store local state and cryptographic materials for principals. This programming style is delicate to translate to CryptoVerif, which does not support private channels. We are considering models that combine local variable bindings (for data writes) and find commands (for data lookups).

More theoretically, we would like to show the correctness of our translation, with respect to a probabilistic, polynomial-time semantics for F#. This would enable us to carry over the computational properties verified by CryptoVerif to our source programs, in terms of PPT adversaries with access to selected F# interfaces.

## 6. COMPUTATIONAL VERIFICATION

This section provides computational security properties for code that implements two stages of TLS: the full record layer and the `ClientKeyExchange` phase of the handshake stage, respectively.

### 6.1 Record protocol

In addition to our code for the record protocol (module `Record` in Section 3), we write F# wrapper code that sets up a secure connection (module `Connected`). From these modules, we automatically extract the computational model using the tool of Section 5. This yields polynomially-replicated communicating sender and receiver processes, wrapped up in a context that sets up a shared connection including a master secret `ms` and random values `cr` and `sr`. We assume that neither the client nor the server is corrupted hence these values are not known to the attacker.

In order to obtain the final CryptoVerif script, we include `CryptoRecord.cv` to the above processes; it contains a hand-written implementation of module `Crypto` that embeds our cryptographic assumptions described below.

**Security Notions for PRF, MAC and Symmetric Encryption.** We present our assumptions for the security primitives of the record protocol. (The full paper lists the corresponding CryptoVerif definitions.)

**PRF.** We specify security in the Random Oracle model [Bellare and Rogaway, 1993], as an equivalence that replaces every call to PRF by a table lookup, such that the first call generates a fresh random value. Blanchet and Pointcheval [2006] use a similar equivalence for proving the security of a signature scheme.

**MAC.** The message authentication code scheme has three functions, `mkgen`, `mac`, and `check` for generating a mac key from a seed, macing a message, and verifying a mac, respectively. We assume unforgeability under chosen message attacks (UF-CMA), stated as an equivalence that replaces every call to `mac` and `check`, so that `check` performs instead a table lookup on any previously-generated

macs. Blanchet [2006, Proposition 2] also relies on this equivalence, and shows that it is indeed implied by UF-CMA.

**SPRP.** The symmetric encryption scheme has three functions `kgen`, `symenc`, and `symdec` for generating a symmetric key from a seed, encrypting a message, and decrypting a message, respectively. Since the ciphersuites we consider use AES and DES, we model this scheme as a block cipher, and assume the usual notion of super pseudo-random permutation (SPRP) [Phan and Pointcheval, 2004], entailing that encryption is a random permutation, at least for randomly chosen keys. (The “super” qualifier indicates that the adversary has also access to a decryption oracle.) We model SPRP in CryptoVerif as an equivalence that replaces every call to encryption and decryption operations by lookups (via the CryptoVerif `find` command) on a table that relates previous encryption and decryption queries with freshly generated random values.

**Record Authentication.** Relying on the same events `Send` and `Recv` as in Section 4, we express our security property as a correspondence query within `CryptoRecord.cv`. Let `Sys` be the script composed of `CryptoRecord.cv` (embedding PRF, MAC, and SPRP assumptions) and the translation of `Connected` and `Record`. CryptoVerif automatically proves record authentication, through 8 game transformations in less than a second.

**THEOREM 4 (RECORD AUTHENTICATION).** *In any polynomial run of `Sys`, with overwhelming probability, for any `Recv` event, there is a `Send` event with opposite entities (client/server or server/client) and matching connection identifiers and records.*

**Record Secrecy.** In order to check secrecy of the communicated payloads, we extend `Record` with a new function definition

```
let send' (id:ConnectionId) =
  let payload = mkNonce() in send id payload
```

This function is translated to a CryptoVerif process that, instead of inputting a payload from the adversary, generates and sends a freshly generated payload. In order to prove secrecy of payload, we exclude the `recv` function (which would otherwise act as an oracle) and obtain a variant `Sys'` of the system `Sys` of Theorem 4. CryptoVerif verifies the secrecy of payload, through 26 game transformations.

**THEOREM 5 (RECORD SECRECY).** *In any polynomial run of `Sys'`, the sequence of sent payload values is indistinguishable from a sequence of independent random values.*

**Attacks and Differences with the Symbolic Model.** Symbolically, it is possible to show secrecy not only for the record payloads, but also for the used keys. Computationally, however, one can only show key secrecy *before* they are actually used; this is the case for the session keys of the record protocol and the pre-master secret of the handshake protocol (Section 6.2).

Another difference can be seen in the following variant of the record protocol, where instead of mac-then-encrypting, we only encrypt the payload and keep the mac in the clear. In this case, CryptoVerif fails to find a proof of Theorem 5, and with reason: nothing in the equivalence of macs ensures that the maced payload should be kept secret. Interestingly, this variant protocol remains secure in the symbolic model, as the mac function (modelled as a non-invertible `hmacsha1` function) leaks nothing. This is another evident illustration of the difference in abstraction levels between symbolic and computational models.

The protocol model we verify here is more limited in comparison to the symbolic one: we do not consider resumption, nor the composition of the record and handshake protocols. We do not model

server corruption, and we verify only a single established connection between an honest sender and an honest receiver (however, using this single connection, a polynomial number of messages can be concurrently exchanged between the sender and receiver). These restrictions stem from limitations in the current version of our compiler and of CryptoVerif. As future work, we foresee no difficulty in reflecting computationally all the symbolic results of Section 4.

## 6.2 Handshake protocol

We consider the code for the first stage of the handshake protocol, up to the sending of the `ClientKeyExchange` message, and show the secrecy of the encrypted pms inside the `ClientKeyExchange` message. For this proof, we disable all the code in `Handshake` module for the subsequent stages. Similarly to the record protocol, we write F# wrapper code in `Certified` that sets up a public/private keypair of a trusted server. Using the tool of Section 5, we extract the polynomially replicated processes from `Certified` and `Handshake`.

**Security Notions for Asymmetric Encryption.** We manually craft `CryptoHandshake.cv` with our cryptographic declarations and assumptions. We have functions `skgen` and `pkgen` for creating private and public keys; we also have functions `enc` and `dec` to encrypt and decrypt messages. (We assume a probabilistic scheme, so the encryption function inputs a seed as well.)

We use a strong notion of security for asymmetric encryption, namely indistinguishability against chosen-ciphertext attacks (IND-CCA2). We use the standard equivalence from the CryptoVerif libraries, which replaces encrypted plaintexts with a message consisting of only zeroes (of the appropriate length), and replaces decryptions by table lookups.

**Secrecy of the Pre-master Secret.** As specified in TLS 1.0, the pre-master secret is the concatenation of a two-byte constant `TLS1p0` plus 46 bytes of random. Let `Sys''` be the script composed of `CryptoHandshake.cv` (embedding the IND-CCA2 assumption) and the translation of `Certified` and `Handshake` (up to the sending of the `ClientKeyExchange` message). CryptoVerif verifies the secrecy of random, through 6 game transformations.

**THEOREM 6 (PMS RANDOM SECRECY).** *In any polynomial run of `Sys''`, the sequence of random values within pre-master secrets is indistinguishable from a sequence of independent random values.*

**Attacks and Differences with Symbolic Model.** Our secrecy property is close to the symbolic notion of strong secrecy but is finer than syntactic secrecy. For instance, symbolically, we can establish syntactic secrecy for the full pms with the embedded protocol version constant, not just random. Even computationally, it may be possible to prove computational secrecy of the whole pms, as shown in independent ongoing work [Morrissey et al., 2008], if we model asymmetric encryption using a weaker one-wayness property that allows the adversary to recover some parts of pms.

We were unable to prove full handshake and resumption authentication computationally, due to limitations of our verification tools. We leave these as future work.

## 6.3 Previous Computational Analyses

There are many analyses of TLS in computational settings; we focus on the positive results, although we still mention important negative results.

Krawczyk [2001] shows that the mac-then-encrypt operation (as used in the computational analysis of our record protocol) is safe when the mac is UF-CMA and the encryption scheme is used in CBC mode and is IND-CPA. Phan and Pointcheval [2004] describe

notions of PRP and SPRP (which we model in CryptoVerif) and their equivalences to standard semantic security and security against lunchtime attacks.

More recently, Fouque et al. [2008] argue for the suitability of the HMACSHA1 construction as a PRF as used in TLS (whereas our model assumes a random oracle). Related to this, Gajek et al. [2008] study randomness extraction from pre-master secret to master secret in the standard model (something we do not address computationally as we focus on the pre-master secret exchange and on the record stages only).

Jonsson and B. S. Kaliski [2002] give a security reduction for the security of TLS/SSL when instantiated with RSA-PKCS-1v1\_5 (modelling the PRF as a random oracle). This contrasts with our work in which the encryption primitive is not explicitly considered but assumed to be IND-CCA2.

Padding attacks have been exploited for TLS both for asymmetric encryption using PKCS #1 [Bleichenbacher, 1998] and for symmetric encryption in CBC mode [Yau et al., 2005], when the adversary is given an oracle that says whether plaintexts are correctly padded or not. In our model, we do not consider this oracle; further we assume that an application encrypts only one block at a time.

Finally, Klima et al. [2003] use the version check in the ClientKeyExchange to construct timing attacks over RSA-based sessions. In our model we do not consider side channel attacks.

## Acknowledgments

We thank Martín Abadi, Bruno Blanchet, Andy Gordon, and Bogdan Warinschi for their helpful comments.

## References

- M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.
- M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security (CCS'93)*, pages 62–73, 1993.
- K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.
- B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.
- B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.
- B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *26th Annual International Cryptology Conference (CRYPTO'06)*, pages 537–554, 2006.
- D. Bleichenbacher. Chosen ciphertext attacks against protocols based on rsa encryption standard PKCS #1. In *18th Annual Cryptology Conference (CRYPTO'98)*, pages 1–12, 1998.
- S. Chaki and A. Datta. Automated verification of security protocol implementations. Technical Report CMU-CyLab-08-002, Carnegie Mellon University, 2008.
- T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, IETF, 1999.
- T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, IETF, 2006.
- T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. Internet Draft, IETF, 2008.
- D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
- P.-A. Fouque, D. Pointcheval, and S. Zimmer. HMAC is a randomness extractor and applications to TLS. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 21–32, 2008.
- A. Frier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet Draft, IETF, 1996.
- S. Gajek, M. Manulis, A.-R. Sadeghi, and J. Schwenk. Provably secure browser-based user-aware mutual authentication over TLS. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 300–311, 2008.
- J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, pages 363–379, 2005.
- C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *12th ACM conference on Computer and Communications Security (CCS'05)*, pages 2–15, 2005.
- K. E. Hickman. The SSL protocol. Draft specification, Netscape, 1995.
- J. Jonsson and J. B. S. Kaliski. On the security of RSA encryption in TLS. In *22nd Annual International Cryptology Conference (CRYPTO'02)*, pages 127–142, 2002.
- J. Jürjens. Security analysis of crypto-based java programs using automated theorem provers. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 167–176, 2006.
- A. Kamil and G. Lowe. Analysing TLS in the strand spaces model. Technical report, Oxford University Computing Laboratory, 2008.
- V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems (CHES'03)*, pages 426–440, 2003.
- H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *21st Annual International Cryptology Conference (CRYPTO'01)*, pages 310–331, 2001.
- J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *7th conference on USENIX Security Symposium (SSYM'98)*, pages 201–216, 1998.
- P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. Cryptology ePrint Archive: Report 2008/236, 2008.
- R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12): 993–999, 1978.
- K. Ogata and K. Futatsugi. Equational approach to formal analysis of TLS. In *25th IEEE International Conference on Distributed Computing Systems (ICSCS'05)*, pages 795–804, 2005.
- L. C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
- D. H. Phan and D. Pointcheval. About the security of ciphers (semantic security and pseudo-random permutations). In *Selected Areas in Cryptography*, pages 182–197, 2004.
- D. Syme. *F#*, 2005. <http://research.microsoft.com/fsharp/>.
- D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce (WOEC'96)*, pages 29–40, 1996.
- A. K. L. Yau, K. G. Paterson, and C. J. Mitchell. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In *Fast Software Encryption*, pages 299–319, 2005.