

Verified Cryptographic Implementations for TLS

KARTHIKEYAN BHARGAVAN and CÉDRIC FOURNET, Microsoft Research, Cambridge
RICARDO CORIN and EUGEN ZĂLINESCU, Microsoft Research-INRIA Joint Centre, Orsay

We 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 automated computational cryptographic verification. We rely on a combination of recent tools and 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: C.2.0 [**Computer-Communication Networks**]: —*Security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Security, Verification

ACM Reference Format:

Bhargavan, K., Fournet, C., Corin, R., and Zălinescu, E. 2012. Verified cryptographic implementations for TLS. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 3 (March 2012), 32 pages.
DOI = 10.1145/2133375.2133378 <http://doi.acm.org/10.1145/2133375.2133378>

1. VERIFYING PROTOCOLS AND THEIR IMPLEMENTATIONS

There has been much recent progress in formal methods and tools for cryptography, enabling 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 protocol verifiers into the arsenal of software testing and verification tools.

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

—Symbolic models treat cryptographic primitives as black boxes and focus on the logical properties of a protocol, as pioneered by Needham and Schroeder [1978] and

An earlier version of this article appears in the *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*.

Author's address: K. Bhargavan, Centre de Recherche Commun INRIA-Microsoft Research, Parc Orsay Université, 28 rue Jean Rostand, 91893 Orsay Cedex, France; email: karthikeyan.bhargavan@inria.fr.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1094-9224/2012/03-ART3 \$10.00

DOI 10.1145/2133375.2133378 <http://doi.acm.org/10.1145/2133375.2133378>

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 cryptographic assumptions more concretely; they treat primitives as probabilistic algorithms over concrete bitstrings and reason about the advantage of an adversary with bounded computational capabilities. Computational verification methods do not scale up as well as symbolic ones, sometimes leading to long, delicate, hand-crafted proofs. Automatable proof methods are more recent (e.g., Laud [2005]; Backes and Laud [2006]), and practical verification tools are just emerging (Blanchet [2008]; Blanchet and Pointcheval [2006]; Tsahhurov and Laud [2007]).

Implementations vs. Abstract Models. Protocol specifications include many details, most of which (but not all) 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, we propose to verify detailed protocol implementations and deployments, as far as possible, rather than handwritten abstract models. By 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 advocate the automatic extraction and verification of symbolic cryptographic models from executable code [Bhargavan et al. 2006; Goubault-Larrecq and Parrennes 2005].

Bhargavan et al. [2006] verify protocol implementations written in F#, a dialect of ML for the .NET runtime, by compilation to symbolic models in ProVerif [Blanchet 2001]. Their approach verifies as much protocol code as possible while providing handwritten models for the rest, such as the core libraries that provide cryptographic primitives, using 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 2008].

Our Approach. Figure 1 illustrates our general approach to developing, testing, and verifying a reference implementation of a protocol, plus a typical application. In contrast with the use of specialized modeling languages for cryptographic protocols, our use of a standard development platform enables early testing to disambiguate the specification, confirm functional correctness, and experiment with potential attacks, among other advantages.

Verification consists of selecting a part of the implementation and 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. Since the verification tool chain is automated, one can easily re-verify the code base as it evolves, much like regression testing.

In our experience, symbolic and computational verifications are complementary. Computational verification is more precise but also 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 formal verification is getting affordable in comparison with

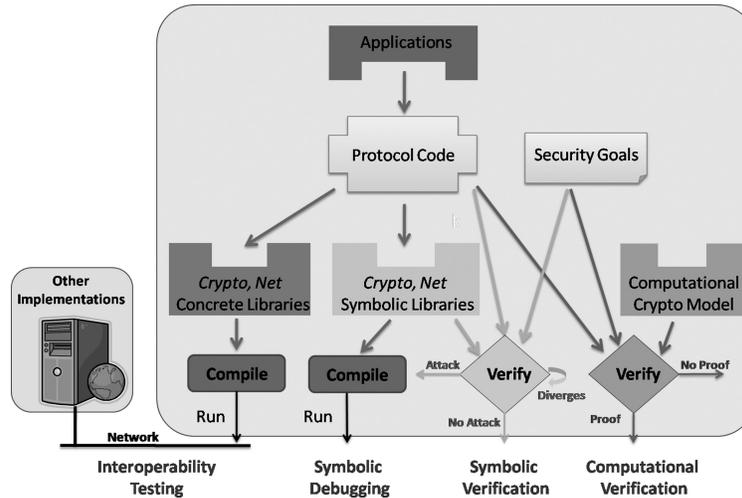


Fig. 1. Verification architecture and tools.

design, development, and testing. The next generation of tools could enable their integration in the development process.

Implementing and Verifying TLS. As an extended case study, this article 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. 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 in F#. Using simple client and server code, we confirm that it 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. To our knowledge, this enables the first automated verification of executable code under standard cryptographic assumptions.
- (4) We review known weaknesses for earlier variants of TLS, and confirm that they are exposed as we attempt to verify the corresponding weakened variants of our code.

Contents. Section 2 recalls the main security features of TLS. Section 3 discusses reference protocol implementations in ML. Section 4 outlines our implementation for TLS, describes its main modules, of TLS and reports on interoperability. Section 5 presents our results using symbolic models; it also discusses formal security issues

and related work. Section 6 describes a new tool for extracting computational models and relating computational security assumptions to concrete cryptography APIs. Section 7 presents our results using computational models of TLS; it also discusses cryptographic issues and related work. Section 8 concludes and suggests future work.

Our verification tools, the code for our TLS implementation, and an extended report with more details are available at <http://msr-inria.inria.fr/projects/sec/fs2cv>.

2. TRANSPORT LAYER SECURITY PROTOCOLS (REVIEW)

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]. Two more recent versions of the standard, TLS 1.1 and TLS 1.2 [Dierks and Rescorla 2006, 2008], include further improvements and clarifications, notably changes to thwart new cryptographic attacks. Since the three versions of TLS 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 most widely deployed version of the protocol. Next, we recall its main security features at the level of detail needed for the rest of the article. We follow the notations and terminology of the standard [Dierks and Allen 1999] and refer to it for a more general presentation.

TLS provides secure communications between a client and a server, with certificate-based authentication of the server and, optionally, of the client. The protocol distinguishes between *sessions* and *connections*; from an established session, each party can derive one or more connections and use them to send and receive sequences 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 the *Handshake protocol* for establishing sessions, the *Alert protocol* for communicating error messages, as well as application protocols.

2.1. Record Protocol

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

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 include a ciphersuite) and on the current *connection state* (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, prefixing the result with a record header. The record header has three fields: the content type of the subprotocol 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 [Bellare et al. 1996] (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 record 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.

2.2. 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 article, we consider only the most common RSA-based modes. We begin with the message flow for a handshake with an anonymous client.



For our discussion, it is convenient to decompose the protocol into the following four phases.

- (1) The client and server exchange connection parameters by means of the hello messages.
- (2) They establish an intermediate shared (pms). When using RSA, the client chooses pms, so the phase consists of a single ClientKeyExchange message.
- (3) They each compute a shared (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_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: a protocol version (version), a ciphersuite (cipher_suite), and a compression method (comp_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, whose

protection itself depends on these parameters. This circularity is a source of concern for TLS, as discussed in Section 5.

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 record format for backward compatibility). 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 pre-master secret `pms`, using the public key of the received certificate. In order to confirm the highest version supported by the client, the protocol version (`ver_max`) influences the padding before RSA encryption and is also embedded as the first two bytes of `pms`, such that

$$\text{pms} = \text{ver_max} \parallel \text{random},$$

where \parallel is bitstring concatenation and randomly consists of 46 random bytes. From the `pms` and exchanged random values, both parties compute the master secret using a pseudo-random function (PRF). This function takes as input, a secret, an identifying label, 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, such that

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

The materials for the Record protocol are generated similarly:

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

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

The two ChangeCipherSpec messages appear in brackets because they are not considered part of the Handshake protocol. 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, thereby providing key confirmation. These Finished messages contain a (hashed) transcript of the handshake to this point; their logical contents is

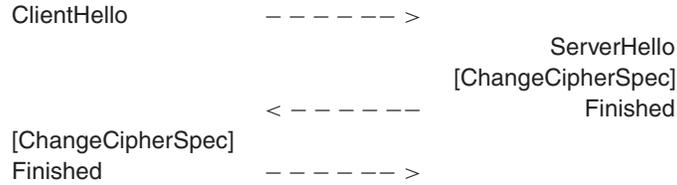
$$\text{verify_data} = \text{PRF}(\text{ms}, \text{finished_label}, \text{MD5}(\text{hsm}) \parallel \text{SHA1}(\text{hsm})),$$

where `finished_label` is a constant string (either "client finished" or "server finished," depending on the message sender) and `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 5. After a successful handshake, the parties can start exchanging application data in both directions.

2.3. Abbreviated Handshake Protocol (Resumption)

Instead of performing a full handshake, TLS offers the possibility of resuming a previously established session and even of duplicating an existing session in order to derive further connections.

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



Otherwise the server generates a fresh session identifier and the handshake continues, as in the general case. The standard does not mandate whether the original connection should be used to protect the resumption; experimentally, we observe that common server implementations of TLS accept resumption ClientHello messages, either protected with the current security parameters or sent in the clear.

3. PROTOCOL IMPLEMENTATIONS IN ML

We write our protocol implementations in the F# programming language. The structure and style of our programs reflect our goal to use the same code, as far as possible, for four different tasks, as depicted in Figure 1: concrete execution, for interoperability testing; symbolic execution, for debugging; symbolic verification; and computational verification.

For symbolic debugging and verification, every .NET function or operating system call that appears in protocol or application code must be given a symbolic model. Similarly, for computational verification, such function calls must be given a computational interpretation. We identify a set of commonly used functions and collect them in a set of core library modules written in F#. We require that protocol implementations use only these library functions to interact with the .NET runtime and operating system. For each library function, we define both symbolic and computational models that are used to extract full symbolic and computational models from protocol implementations.

Our model extraction tools only support a subset of rich language features of F#. In the remainder of this section, we describe the core library modules and language features used by our TLS implementations and supported by our tools.

Libraries for Networking and Cryptography. We use three core library modules: Net, Crypto, and Prins.

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 rcv to exchange messages on this connection.

The module Crypto defines standard cryptographic primitives. For example, our Record protocol implementation calls hmacsha1 and aes_encrypt to mac-then-encrypt messages, while the Handshake implementation creates a fresh pms using mkRandom and encrypts it using rsa_encrypt.

```

module Net
  type conn // TCP Socket
  val connect: string → conn
  val listen : string → conn
  val close: conn → unit
  val send: conn → bytes → unit
  val recv: conn → bytes

module Crypto
  val mkRandom: int → bytes
  val md5: bytes → bytes
  val sha1: bytes → bytes
  val aes_encrypt: symkey → bytes → bytes
  val aes_decrypt: symkey → bytes → bytes
  val hmacsha1: mackey → bytes → bytes
  val hmacsha1Verify: mackey → bytes → bytes → bool
  val rsa_encrypt: pubkey → bytes → bytes
  val rsa_decrypt: privkey → bytes → bytes

```

The module Prins (for principals) defines functions to create, retrieve, and check the validity of X.509 certificates. Each module has a public and a private interface. Public interfaces are those offered to applications and are also used for symbolic verification. Private ones are only used within our reference implementation. For example, the private interface of the Prins module is displayed, whereas the public interface does not have the `getPrivateKey` function. (Hence, the attacker may only obtain private keys through `leakX509`.) Unless mentioned explicitly, we assume that a module's public and private interfaces coincide.

```

module Prins
  type CertName // X.509 subject name
  val genX509Cert: CertName → unit
  val getX509Cert: CertName → bytes
  val getPrivateKey: CertName → privkey
  val checkX509Cert: bytes → (CertName * pubkey)
  val leakX509: CertName → privkey

```

Our concrete implementation 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 implements `connect` by using `System.Net.Sockets.TcpClient`.

Library Models for Debugging and Verification. Following an approach proposed by Bhargavan et al. [2006] (see Figure 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 `HMAC-SHA1(key,text)` that represents the keyed hash; Net models connections as communications on local channels between processes; and Prins models the X.509 store as a local private database.

For computational verification, we develop a third version of these library functions, encoding our computational cryptographic assumptions in the source language of `CryptoVerif`, as described in Section 6.2.

All three versions of the library modules implement the same interfaces. By compiling a protocol implementation and application 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

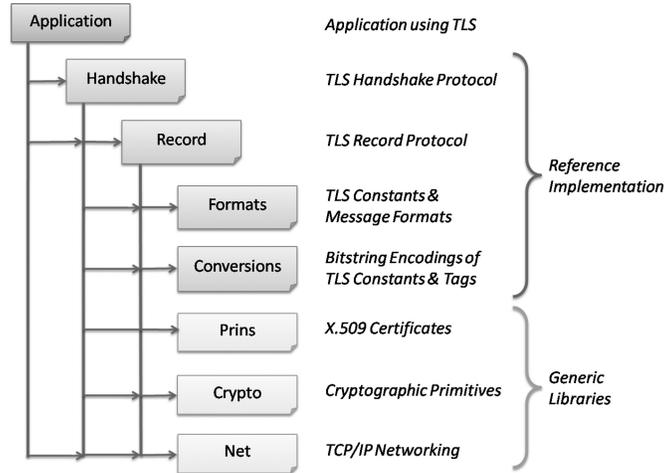


Fig. 2. Structure and module dependencies for our implementation.

follows their models; as such, these libraries represent the trusted computing base for our verification results.

Supported F# Language Features. The subset of F# supported by our verification tools 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, it avoids other features, such as mutable references, higher-order functions, and classes, because they are not fully supported by our verification tools. Moreover, as previously mentioned, every .NET function or operating system call that we use must be included in the libraries already described. For example, in order to use file I/O operations, we would need to extend our libraries.

Our code uses recursive functions sparingly, because they usually lead to nonterminating 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. Pure functions are efficiently translated to our target verification tools (for instance, into ProVerif reductions), whereas functions with side effects are translated to processes, which are more complex to verify.

Programming for verification imposes a coding discipline that can seem restrictive but is still rich enough to write fully functional protocol implementations, as we shall see next.

4. REFERENCE IMPLEMENTATION

We now describe our reference implementation of TLS 1.0. Although the standard does not specify any particular API, the TLS protocol is usually implemented as a library linked to Web-based applications, such as browsers, proxies, and servers. Figure 2 gives the structure of our reference TLS implementation; each box represents an F#

module, and each arrow represents a direct dependency between modules. Hence, the Handshake and Record modules implement the Handshake and Record protocols, respectively, and their interfaces enable some Application module to send and receive messages over TLS. The Formats module contains functions to build and parse formatted TLS messages. It relies on the Conversions module for low-level encodings of strings and TLS-specific tags. In the rest of this section, we outline the implementations of these modules.

4.1. Record Module

The Record public interface shown here consists of only two functions that enable applications to send and receive messages over TLS connections.

```

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

type Connection = {
  net_conn: Net.conn;
  entity: ConnectionEnd;
  sessid: bytes;
  crt_version: ProtocolVersion;
  read: ConnectionState;
  write: ConnectionState; }

```

The module maintains a private database of active connections indexed by ConnectionIds and populated by the Handshake protocol as new connections are established. The record type Connection represents an established TLS connection. It contains the underlying TCP connection net_conn, the entity (Client or Server) which owns the connection, the identifier of the session from which the connection is derived, the protocol version used for 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 secret mk used by the mac algorithm, the current sequence number seq_num, and the security parameters sparams established by the Handshake protocol, which includes the cipher type (i.e., stream or block cipher), the encryption algorithm (like AES), and the mac algorithm (like HMAC-SHA1).

```

type ConnectionState = {
  cipher_state: CipherState;
  mk: bytes;
  seq_num: int;
  sparams: SecurityParameters; }

type SecurityParameters = {
  cipher_type: CipherType;
  bulk_cipher_algorithm: BulkCipherAlgorithm;
  mac_algorithm: MACAlgorithm; }

```

Given an established TLS connection conn, the send and recv functions write and read payloads over the connection in accordance with the Record protocol. As they process messages, they log two security events,

$$\text{Send}(\text{entity}, \text{payload}, \text{conn}) \quad \text{and} \quad \text{Recv}(\text{entity}, \text{payload}, \text{conn}),$$

where entity is a label with value either Client or Server. The first event logs that entity sends message payload over connection conn. The second event logs that entity accepts message payload as valid over connection conn (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`. 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` to decrypt the payload `msg` and verify the mac; it calls `log` to log a `Recv` event as just described; it calls `updateReadSeqNum` to increment the sequence number of the read connection state; and finally, it calls `storeConnection` to update the connection database with new connection parameters before returning `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
  log tr (Recv(conn.entity, msg, conn));
  let conn = updateReadSeqNum conn in
  storeConnection connid conn;
  msg
```

The cryptographic checks are all performed in `verifyPayload`. 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`, and yields plaintext and a new connection state 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 secret in `connst`, thereby authenticating the sequence number, content type, protocol version, ciphertext length, and payload. Finally, `checkContentType` (besides checking the content type of the received record) handles the case when an alert is received.

```
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 failwith "bad version" else
    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 = concat5 bseq bct bver len payload in
    let mres = hmacVerify connst maced recvmac in
    if mres = false then failwith "bad record mac" else
      checkContentType ct rct payload;
      let conn = updateReadConnectionState conn connst in
      (conn, 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 and payload.

4.2. 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

```

The second parameter for close is a Boolean that indicates whether to wait for the TCP connection to be terminated or not. The usage of the other functions is detailed next.

The module maintains a private 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 messages, and the pre-master secret pms. All these fields are exchanged during the full handshake that establishes 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 a full handshake to establish a new session and a new connection in each direction. Upon completion of a full handshake, both accept and connect construct their own Session and Connection records and populate them with all the values authenticated by the handshake, including, for instance, the session identifier, ciphersuite, security parameters, and computed keys. To indicate completion of the protocol and agreement on these values, the two functions log the following events.

```

SendFinished(entity,session,conn)   AcceptFinished(entity,session,conn)

```

Each event contains the entity that logs it and full Session and Connection records. These are the records that are stored in the private databases and of which indexes are returned by the connect and accept functions.

A client can call resume for stored sessions to start an abbreviated handshake. A server executing accept can also perform an abbreviated handshake if the clients asks for resumption and the requested session is stored in the server's database of active sessions. (Our server always resumes a stored session when asked.) Upon completion of the Resumption protocol, both accept and resume construct their own new Connection records and populate them with the new negotiated values: the version and the freshly computed keys. To indicate completion of an abbreviated handshake and agreement on these values, the two functions log the events SendFinished-Res(entity,session,conn,ch,sh) and AcceptFinishedRes(entity,session,conn,ch,sh). The parameters of these events have the same meaning as those of a full handshake. In

addition, the client and server Hello records are tracked. In these events, 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.

4.3. Sample Applications and Interoperability

Using our reference implementation, we write three applications:

- A client that connects to an HTTPS URI and retrieves a webpage over a TLS connection.
- A server that listens at an HTTPS URI and returns a webpage over a TLS connection.
- A mutually authenticated client-server application where the client authenticates to the server using a password over a TLS connection.

Next, we outline the code for the two client applications and describe how we test them symbolically and concretely.

Simple HTTPS Client. Our basic HTTPS client (shown here) uses functions from the `Handshake` and `Record` modules.

```

let clientHttps uri =
  let connid, sessionid =
    tlsconnect uri in
  httpGet connid uri
let tlsconnect uri =
  let conn = Net.connect uri in
  let shost = Net.get_host uri in
  Handshake.connect conn shost
let httpGet connid uri =
  let req = buildHttpGet uri in
  Record.send connid req;
  let resp = Record.recv connid in
  parseHttpGetResponse resp connid

```

The function `clientHttps` fetches a single page from a server by calling the auxiliary functions `tlsconnect` and `httpGet`. The function `tlsconnect` first establishes a TCP connection (`conn`) with the server by calling `Net.connect` and then calls `Handshake.connect` to perform a TLS handshake over this unprotected connection. If the handshake succeeds, it returns a connection identifier (`connid`) and a session identifier (`sessionid`). The function `httpGet` performs an HTTP exchange over the TLS connection. It builds an HTTP/Get request, sends the request over the TLS connection, waits for a response, and returns its parsed contents.

Password-Based Client Authentication. To illustrate how applications with additional security properties may be built on top of our TLS implementation, we develop a client-server application where the client sends its user name and password over a TLS connection, and the server checks these values to authenticate the client. We assume that the client passwords are already shared between clients and servers, and are stored in a private database on the server side. The code for the client is as follows.

```

let clientPasswd uri =
  let connid, sessionid = tlsconnect uri in
  Printf.printf "\nType your user name\n";
  let client = read_line() in
  sendPasswd connid (str client) uri
let sendPasswd connid client server =
  let pwd = get_password client server in
  log tr (SendPasswd(Client, pwd, client, server));
  let msg = concatvar (utf8 client) pwd in
  Record.send connid msg

```

As in the simple HTTPS application, the client calls `tlsconnect` to establish a TLS connection. It then reads the user name from a console and calls the function `sendPasswd` that performs a secure message exchange with the server. The `get_password` function returns a bitstring and may be implemented in several ways. For instance, it may ask

the user to enter her password, or it may query a password database. (Our symbolic verification does the latter.) The client logs the password with the identities of the involved parties and sends its username and password over the Record protocol. (The function `concatvar` prepends a length field to the concatenated bitstring so that the server may unambiguously parse the record payload.)

Concrete Executions and Interoperability Testing. As a first experiment, we run our applications concretely over a network and test them against one another. For instance, our HTTPS client and server are successfully able to perform both handshake and resumption protocols and to perform HTTP exchanges on top of the established connections.

Then to test interoperability, we run our applications concretely over a network against popular client/server applications. In our experiments, our simple HTTPS client application successfully fetches pages from Web servers running Microsoft IIS 7.0 or Apache 2.2.9, and our HTTPS server application successfully serves pages to clients running Microsoft Internet Explorer 7 or Mozilla Firefox 3.0.6. Both applications successfully resume sessions when asked, for instance, to refresh a webpage displayed in a previous session. Hence, at least for these applications, our reference implementation interoperates with the mainstream TLS implementations used by these browsers and Web servers, including OpenSSL and the Windows CryptoAPI.

In practice, most TLS implementations expose only a selection of cryptographic suites, and sometimes slightly deviate from the standard, for instance in the handling of the versions in `ClientHello` and `ClientKeyExchange` messages. Our experiments also helped us explore these issues and disambiguate details of the standard.

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 and provide partial support for SSL 3.0 and TLS 1.1. In this mode, we support all ciphersuites using AES, DES, RC4, SHA, and MD5 algorithms, thus, for example, `TLS_RSA_WITH_AES_128_CBC_SHA` and `TLS_RSA_WITH_RC4_128_MD5`. Our implementation does not support renegotiation, data compression, or fragmentation. It does not send alerts, and it silently fails upon receiving a bad message. Despite these limitations, our experiments show that it is adequate for writing simple client and server applications.

5. 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 [Blanchet 2001], that analyzes such models automatically. Although our symbolic verification problem is undecidable in general, for many protocol implementations, the verifier either proves the security goals or produces a counterexample. In some cases, the verifier may not terminate; in others, it may take several gigabytes of memory.

To use this tool chain, we write symbolic implementations for selected low-level libraries, as described in Section 3. We define the attacker model in terms of the public interface exposed by these libraries and by our reference implementation, and we write our authentication and secrecy goals as correspondences 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 public interface. Our results rely on the correctness of the core translations and algorithms underpinning the model extractor and ProVerif [Bhargavan et al. 2006; Blanchet 2001].

Attacker Model. The attacker capabilities are given by the public interfaces of the modules Net, Crypto, Prins Handshake, and Record. Thus, we let attackers range over arbitrary programs that use the functions given in these interfaces. This yields a rich symbolic threat model à la Dolev and Yao [1983] with an active attacker that can perform the following activities.

- Control the network (Net) and perform cryptographic operations (Crypto).
- Create any number of servers by generating certificates and, optionally, compromise any server by reading its private key. Whenever this occurs, an event Leak(subj) is emitted to record the subject subj of the compromised certificate (Prins).
- Run any number of sessions between clients and servers of its choice—obtaining their connection and session identifiers—and trigger the Resumption protocol for any session identifier (Handshake).
- Send and receive messages over the record layer (Record).

We let *System* range over programs that consist of the symbolic implementations of the Net, Crypto, and Prins modules; of the implementations of the Handshake, Record, Formats, and Conversions modules; and of arbitrary code with access to the public interfaces of Net, Crypto, Prins, Handshake, and Record. All the security results of this section hold for any run of any such program.

5.1. Handshake Protocol

We first present symbolic results for the Handshake protocol. For the Full Handshake, 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 System, for any AcceptFinished event logged by an entity (client or server), either there is a SendFinished event logged by the opposite entity (server or client respectively) with matching connection and session parameters, or the opposite entity is corrupted.*

In these statements, the server identity is just its certificate subject; it is left to the application to correlate this authenticated identity with the intended peer server, using, for instance, the target URL. (Experimentally, neither Internet Explorer nor Firefox sends the client Finished message when this correlation fails.)

The proof is by automated verification of the following two queries on the extracted model.

query ev: AcceptFinished(Client, sess, conn)
 \implies principals: sess, subj, pms & correctParam: sess & connections: conn, conn'
 & (ev: SendFinished(Server, sess, conn') | ev: Leak(subj)).

query ev: AcceptFinished(Server, sess, conn)
 \implies principals: sess, subj, pms & correctParam: sess & connections: conn, conn'
 & (ev: SendFinished(Client, sess, conn') | attacker: pms).

In the queries, lowercase parameters are variables, whereas Client and Server are symbolic constants. Queries relate facts using logical conjunction (&), disjunction (|), and temporal implication (\implies): $e \implies e'$ means that whenever e is true, e' must have been true before. Of the three connectives, & has the highest precedence and \implies the lowest. All variables in a query are universally quantified, except for the variables

on the right-hand side of \implies that do not appear in the left-hand side, which are existentially quantified. Each fact has the form $p : m_1, \dots, m_n$, where p is a predicate and m_1, \dots, m_n are symbolic terms.

The predicates `ev` and `attacker` are predefined in ProVerif. `ev: e` means that the event `e` has been logged, and `attacker: m` means that the attacker knows message `m`.

The predicates `principals`, `correctParam`, and `connections` are defined by us. `principals: sess, subj, pms` holds when the subject of the certificate `sess.server_cert` and `sess.pms` equal, respectively, `subj` and `pms`. `correctParam: sess` holds when `sess.sid` equals the `sid` field of the `ServerHello` message `sess.sh`. `connections: conn, conn'` holds when the version field `crt_version` of the two connections are equal and the read connection state of `conn` equals the write connection state of `conn'` (and vice-versa).

The two queries require that, at the end of the handshake, the server and client agree on the important fields of the session. These fields 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 of `conn` and `conn'`) are also correlated.

As a minor technical point, there is no formal agreement on the client's lowest supported version for TLS (`ver_min`): the attacker may change this value, together with the message record format for `ClientHello`, without detection. However, this is innocuous as long as (1) the client checks that `version \geq ver_min` and (2) the server's choice of version does not depend on `ver_min`. Indeed, the standard says that the server should pick version to be either `ver_max` or its highest supported version, whichever is lower.

Session Resumption. We obtain a similar theorem for the Resumption protocol.

THEOREM 2 (RESUMPTION AUTHENTICATION). *In any run of System, for any AcceptFinishedRes event logged by an entity (client or server), either there is a SendFinishedRes event logged by the opposite entity (server or client) with matching connection and session parameters, or the opposite entity is corrupted.*

Moreover, within each AcceptFinishedRes and SendFinishedRes event, the new ServerHello message has the same session identifier and ciphersuite as the old session.

The proof is by automated verification of the following two queries.

query `ev: AcceptFinishedRes(Client, sess, conn, ch, sh)`
 \implies `principals: sess, subj, pms & correctParam: sess & connections: conn, conn'`
 $\&$ `matchRes: sess, ch, sh`
 $\&$ `((ev: SendFinishedRes(Server, sess, conn', ch, sh)`
 \implies `ev: SendFinished(Server, sess, conn')`) | `ev: Leak(subj)`).

query `ev: AcceptFinishedRes(Server, sess, conn, ch, sh)`
 \implies `principals: sess, subj, pms & correctParam: sess & connections: conn, conn'`
 $\&$ `matchRes: sess, ch, sh`
 $\&$ `((ev: SendFinishedRes(Client, sess, conn', ch, sh)`
 \implies `ev: SendFinished(Client, sess, conn')`) | `attacker: pms`).

These queries use a new predicate `matchRes` that correlates session parameters from the original handshake (recorded in `sess`) with the connection parameters established by the new abbreviated handshake (recorded in the Hello messages `ch` and `sh`). In particular, `matchRes` requires that the old `sess.sid` equals both the `rsid` requested in the new `ch` and the `sid` returned in the new `sh`; it also requires that the old `cipher_suite` (within `sess.sh`) equals the new `cipher_suite` (within `sh`).

Conversely, `matchRes` does not ask for equality between the version fields of the old and new `ServerHello` message, and with good reason: The queries do not hold if `matchRes` also requires agreement on the protocol version. As explained in Section 5.5, this reflects a potential version rollback attack on TLS resumption.

In the two queries, the second \implies is a requirement that a full handshake has been executed before resumption. For instance, when verifying the first query, whenever an `AcceptFinishedRes` event occurs, some `SendFinishedRes` event (with matching parameters) must have occurred before, and moreover, this `SendFinishedRes` event can only have occurred after some `SendFinished` event. These queries encode a weak form of client authentication: during resumption, the new connection is associated with the same client as the initial connection. Hence, an adversary cannot use resumption to hijack a session.

Handshake Secrecy. We prove secrecy for all secrets generated during full and abbreviated handshakes, including `pms`, `ms`, the two encryption keys, the two mac secrets, and the two IVs. Our theorem asserts syntactic secrecy; it requires that the secret values are not obtained by the attacker unless he has compromised the server or controls the client.

THEOREM 3 (HANDSHAKE SECRECY). *In any run of System, for any connection in any session between two entities, either the values of `pms`, `ms`, all encryption and mac keys, and initial IVs are syntactically secret, or one of the entities is corrupted.*

To encode this secrecy requirement, we extend the protocol interface with a function that allows an attacker to guess the value of a secret and, if the guess matches the value stored in the connection or session database, logs an event `LeakedSecret` to indicate that the attacker has compromised the session and connection. Then, the theorem is established by proving the following query.

query `ev: LeakedSecret(entity, sess, conn)`
 \implies principals: `sess, subj, pms` & (`ev: Leak(subj) | attacker:pms`).

This states that if any of the secret values stored in the database is obtained by the attacker, then one of the two entities in the session must be corrupted.

5.2. Record Protocol

For the Record protocol, authentication is specified as a correspondence between the `Recv` events emitted when a receiver accepts a message and the `Send` events emitted when messages are sent.

THEOREM 4 (RECORD AUTHENTICATION). *In any run of System, for any `Recv` event logged by an entity (client or server), either there is a `Send` event logged by the opposite entity (server or client) with matching payload and connection parameters, or one of the two entities is corrupted.*

The proof is by verification of the following two queries.

query `ev: Recv(Client, payload, conn)`
 \implies principals: `sess, subj, pms` & connections: `conn, conn'`
 & `connInSess: conn, sess` & `ev: AcceptFinished(Client, sess, conn')`
 & (`ev: Send(Server, payload, conn')` | `ev: Leak(subj) | attacker: pms`).

query `ev: Recv(Server, payload, conn)`
 \implies principals: `sess, subj, pms` & connections: `conn, conn'`
 & `connInSess: conn, sess` & `ev: AcceptFinished(Server, sess, conn')`
 & (`ev: Send(Client, payload, conn')` | `ev: Leak(subj) | attacker: pms`).

The queries say that if a Recv event is logged for a connection `conn` that is part of a session `sess`, then either the corresponding Send event was logged, or one of the entities in `sess` is corrupted. In contrast with handshake authentication, our record authentication theorem allows for the corruption of either entities. This is because record authentication depends upon the secrecy of session and connection secrets (specifically the authentication keys and IVs), and these are secret only if neither entity is corrupted (see Theorem 3). Note also that the Recv event only mentions the connection `conn` and not the session; so both queries use the AcceptFinished event to choose some valid session `sess` and then call `connInSess: conn, sess` to check that the session identifier in `sess` matches the one in `conn`.

Record authentication implies that the client and server have the same sequence number (`read.seq_num`). In TLS, sequence numbers are incremented each time a record is processed to ensure connection integrity: if a sequence of records is received by some party on a connection, then the corresponding party has sent (at least) the same sequence of message on the same connection. Although Theorem 3 is stated in terms of a single message, it implies connection integrity in the following sense. It guarantees that whenever a record payload is accepted, both parties agree on its sequence number. Hence, if the accepted payloads and sent payloads are arranged in order of their sequence numbers, either the accepted sequence is a prefix of the sent sequence, or the sending entity is corrupted. This argument holds for arbitrarily large sequence numbers. We support it in our symbolic model by allowing the adversary to choose sequence numbers (the sender reads the sequence number for the next record from a public channel).

Next, we show that the Record protocol preserves payload secrecy. More precisely, we prove that if a freshly generated (secret) nonce is used as a record payload over an established connection, then it remains syntactically secret as long as both client and server are uncorrupted. To specify this property, we extend the Record module with two new functions. First, we add a function `send'` that only sends fresh nonces as record payloads, such that

```
let send' (id:ConnectionId) = let nonce = mkRandom 16 in send id nonce.
```

Second, we add a function that allows the attacker to guess the value of a payload and, if the guess matches one of these fresh nonces, logs an event `LeakedPayload` to indicate that a secret payload has been obtained by the attacker. We add both these functions to the Record public interface and remove the function `recv`; otherwise, `recv` acts as a decryption oracle allowing the attacker to trivially obtain the secret payload. We let *System'* be the variant of *System* with this modification.

THEOREM 5 (RECORD PAYLOAD SECRECY). *In any run of System', for any record payload sent over a connection between two entities, either the payload is syntactically secret, or one of the entities is corrupted.*

The proof is by verification of the following query.

```
query ev: LeakedPayload(entity, payload, conn)
  => principals: sess, subj, pms
    & connInSess: conn, sess & ev: AcceptFinished(entity, sess, conn')
    & (ev: Leak(subj) | attacker: pms).
```

This query is structurally similar to the authentication queries. It retrieves a valid session for the connection and states that either the client or the server must be corrupted.

Verified Parts of TLS	Security Goals	F# Code	Queries	Time	Memory
Full Handshake	Authentication	1418 lines	2	27 s	60 MB
Full Handshake	Secrecy	1418 lines	2	25 s	80 MB
Full Handshake & Resumption	Authentication	2194 lines	2	8 min	460 MB
Full Handshake & Resumption & Record	Authentication (Record Only)	3344 lines	2	11 min	700 MB
Full Handshake & Resumption & Record	Authentication & Secrecy	3344 lines	10	3.5 h	4.5 GB
Full TLS & Password-based Application	User Authentication	3855 lines	1	1.5 h	1.2 GB

Fig. 3. Summary of experimental results for symbolic verification.

5.3. Password-Based Client Authentication

Section 4.3 describes an application that uses shared passwords for client authentication on top of TLS connections. Clients are identified by a username and password; servers are identified, as usual, by the subject name of their X.509 certificates. The application is implemented in the `Passwd` module, and has the following interface.

```

val generatePasswd: UserName → ServerName → unit
val clientPasswd: UserName → ServerName → unit
val serverPasswd: ServerName → unit

```

The function `generatePasswd` creates a new username-password record for use at a particular server and stores it in a protected password database. The functions `clientPasswd` and `serverPasswd` start new instances of the client and server application, respectively.

We let $System''$ be $System$ augmented with the `Passwd` module and arbitrary code with access to the interface of `Passwd`, plus all the public interfaces of $System$.

THEOREM 6 (PASSWORD-BASED USER AUTHENTICATION). *In any run of $System''$, for any `AcceptPasswd` event logged by a server, either there is a `SendPasswd` event logged by a client with matching username, password, and server name, or the password is known to the attacker, or the server is corrupted.*

The proof is by verification of the following query.

```

query ev: AcceptPasswd(Server, pwd, user, subj)
  ⇒ ev: SendPasswd(Client, pwd, user, subj) | attacker: pwd | ev: Leak(subj).

```

5.4. Experimental Results

All the queries in this section are automatically verified on the systems just described. In some cases, we had to hand-tune the protocol code extensively in order to make verification feasible; otherwise `ProVerif` either ran out of memory or did not terminate for several days. To this end, we also verified parts of the protocol in isolation during early development. For comparison, Figure 3 gives the number of queries and verification times for these simpler experiments, as well as for the full systems. All experiments are performed on a computer with Intel Xeon Dual Quad-core processors at 3 GHz and 32GB RAM. We used `FS2PV` version 1.1 and `ProVerif` version 1.14pl5.

The first row of Figure 3 describes the verification of handshake authentication (Theorem 1) on just the Handshake module, with Record and Resumption protocols

disabled. (These protocols are disabled by removing their functionality from the public interface.) The second row verifies handshake secrecy (Theorem 3), again, for just the Handshake protocol. The third row verifies the Handshake module for resumption authentication (Theorem 2) but with the Record protocol disabled. The fourth row verifies our full reference implementation of TLS for record authentication (Theorem 4). The fifth row verifies the full implementation for all the queries in this section, establishing Theorems 1–5. The final row verifies our password-based client-server application (Theorem 6).

5.5. Related Work: Symbolic 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 the client and server both prefer to use strong cryptography but are also willing to use weak cryptography, then an attacker may convince them both to establish a session with weak cryptography. This is called a *ciphersuite rollback* attack. In our model, this attack appears as a failure of handshake authentication (Theorem 1). We experimented with an SSL 2.0-like variant of our client implementation that sends application data before receiving a valid Finished message. ProVerif rightfully failed to prove an authentication query, and instead, produced a counterexample indicating the attack.

Recent versions of TLS (since SSL 3.0) guarantee full handshake integrity by authenticating all previous handshake messages in the final Finished messages. In particular, Hello messages are integrity-protected, thus preventing ciphersuite rollback attacks. However, if a client and server still support SSL 2.0 for backwards compatibility, then a new *version rollback* attack becomes possible: an attacker may convince both parties to use SSL 2.0 and then exploit any flaws (including ciphersuite rollback) of the earlier version. Version rollback is a general problem faced by protocols that have multiple versions deployed at the same time. SSL 3.0 and TLS include two mechanisms to address this problem, both in the ClientKeyExchange message, so that a server can authenticate `ver_max`, the highest protocol version supported by the client. In particular, TLS clients embed `ver_max` within `pms`, but SSL 2.0 clients do not. Hence, a server can identify clients that support versions higher than SSL 2.0 and protect itself from version rollback. Indeed, Theorem 1 asserts that our Handshake implementation authenticates the ClientHello, which includes `ver_max`.

The version rollback protection mechanisms of TLS does not suffice for the Resumption protocol, since the abbreviated handshake does not contain a ClientKeyExchange message. Hence, every implementation of TLS, including ours, is vulnerable to version rollback during resumption. In particular, resumption authentication (Theorem 2) guarantees only that the new connection parameters excluding `ServerHello.version` are correlated with the old session parameters, not that the new version is higher than the old one. Experimentally, we found that deployed server implementations of TLS are vulnerable to version rollback attacks from TLS 1.0 to SSL 3.0 during resumption. However, we could not exhibit the potentially more dangerous rollback from TLS 1.0 to SSL 2.0, partly because, in practice, the length of session identifiers differs between these two versions. (The standard does not impose a particular length.)

In addition to protocol-level attacks, our method also finds common implementation-level attacks. A common error in many TLS implementations and applications is the incorrect validation of received server certificates (e.g., see [Cisco 2007]). In an early

version of our implementations, we found a certificate validation error (as a failure of Theorem 1). We found errors in other early versions, such as allowing the null ciphersuite to be selected or not checking that the received sequence number is correct. In each case, our verification tools generated counterexamples to our secrecy and authentication queries.

More recent vulnerabilities in OpenSSL exposed over 400 applications that relied on OpenSSL for its cryptographic and TLS/SSL functionality [OpenSSL 2009]. In essence, these applications and other functions within OpenSSL incorrectly handled the return values of the main signature verification functions, treating erroneous signatures as valid. In particular, the vulnerability allowed certificate verification in the OpenSSL TLS implementation to be bypassed in some conditions. Since our implementation does not use return values to signal errors, and instead, raises exceptions, such vulnerabilities do not directly appear in our code. However, our verification tools would find those errors, as well.

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

5.6. Related Work: Previous Symbolic Analyses

In a long line of works starting from Dietrich [1997], researchers have used various techniques to verify models and, more recently, implementations of different versions of SSL and TLS. We describe only those most closely related to our work.

Wagner and Schneier [1996] carry out an informal analysis of SSL 3.0. They point out an ambiguity in the specification which allows application data to be exchanged before the confirmation of security parameters with the Finished messages. Thus, in this case ciphersuites rollback attacks could be successful. This ambiguity was corrected in the TLS 1.0 protocol. They also fear possible anomalies in the Resumption protocol related to the version rollback issue we have mentioned.

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 protocols of increasing complexity, including a version of SSL 3.0 with both full and abbreviated handshakes 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 full 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. Díaz et al. [2004] show the correct message flow of the Handshake protocol in the absence of the intruder using the UPPAAL model checker. Ogata and Futatsugi [2005] show secrecy of the pre-master secret and liveness properties for the Handshake protocol 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 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 [2009]

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

6. A COMPUTATIONAL VERIFIER FOR PROTOCOL IMPLEMENTATIONS

This section describes our computational verification approach and tools. Section 7 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).

6.1. 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, referring to Blanchet [2008] and 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, modeled 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), while the game-based equivalences are used to state minimal negative assumptions. Section 7 gives some examples.

Proofs, Games, and Indistinguishability. The input script can be seen as an initial game, modeling 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* [Blanchet and Pointcheval 2006; Corin and den Hartog 2006].

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.

In a recent case study, CryptoVerif is used to verify a model of the Basic and Public-Key Kerberos protocol [Blanchet et al. 2008].

Process and Variable Instances. Processes in CryptoVerif can be replicated polynomially in a given security parameter, enabling multiple parallel executions. Hence, every replica has access to its own instance of each variable; in addition, a special find command gives read access to all other instances of each variable.

Target Security Properties. CryptoVerif can verify authentication and secrecy properties. Authentication goals are written as correspondences [Blanchet 2007], much as in our symbolic models. Computationally, correspondences assert that if some

event is executed, then other events must also have been executed (at least once) with matching parameters, and at least with overwhelming probability. This last bit reflects the computational nature of the model. (CryptoVerif can deal with more general properties expressed as implications logical formulas and can verify both injective and non-injective properties.)

Secrecy goals are expressed as 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 any instance of a given variable from a random value. The stronger notion (query `secret`) states that the adversary cannot distinguish the vector of values of all the instances of a given variable from a vector of independent random values.

6.2. A Compiler from ML Programs to CryptoVerif Scripts

We outline the design of our new model extractor that translates protocol implementations written in F# to CryptoVerif scripts. (See Bhargavan et al. [2009] for additional details.) 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. It then 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` treats bytes as concrete bitstrings and defines cryptographic primitives as uninterpreted functions over bitstrings. For each primitive, it defines equations, inequations, and equivalences that encode specific cryptographic assumptions for the protocol. Functions for generating fresh values, such as `mkRandom`, are written using the CryptoVerif primitive `new` that chooses a random bitstring uniformly from a type, such as the set of all 16-byte nonces. Our model of `Prins` maintains a private database of public-private keypairs as a CryptoVerif array.

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 7.

Compiling Protocol Code to CryptoVerif. The compiler first applies a series of code transformations to generate a smaller, more specialized source program. These transformations include aggressive inlining of nonrecursive functions, partial evaluation of functions and patterns, and dead-code elimination.

The compiler then converts all public functions to processes and normalizes the resulting code to fit in a restricted ML syntax that is very close to that of CryptoVerif. The final CryptoVerif script is obtained by translating these processes and the protocol security goals into CryptoVerif syntax, and by inlining abstract models for the core libraries.

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.

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

Top-Level Process. Each variable binding $\text{let } x = e$ 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 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 7)—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.

Translation Correctness and Expressivity. To gain more confidence on the translation, we show the soundness of the translation in ongoing work [Bhargavan et al. 2009]. We define a probabilistic reduction semantics for $F\#$ and use it to prove the soundness of a series of source-to-source transforms that bring our programs to a fragment of $F\#$ closer to CryptoVerif. For this fragment, we relate our $F\#$ semantics to a lower-level abstract machine closely related to the probabilistic polynomial semantics of CryptoVerif.

We also define abstractions for private databases and compromised keys in CryptoVerif. Private databases allow, for instance, the modeling of principal libraries (e.g., the Prins library, previously described) where key materials are stored. It also enables the modeling of the compromise of a principal, where its keys get leaked to the adversary.

7. COMPUTATIONAL VERIFICATION

This section describes the computational security properties that we verified for our implementations of the Record and Handshake protocols. We verify various

configurations of these protocols in detail, but in contrast with symbolic verification, our results do not extend to the full TLS protocol.

7.1. Record Protocol

To verify the Record protocol (module Record in Section 4) in isolation, we write a module Connected that abstractly simulates the Handshake protocol by populating a global database with preestablished connections. The module provides two functions to the adversary: the first generates a fresh private connection for a given connection identifier and adds it to the database; the second leaks all the contents of a connection to the adversary, but only after issuing a ConnLeak event that marks the connection as corrupted.

We use the tool described in Section 6 to automatically extract a computational model from the modules Record and Connected. This yields polynomially replicated communicating sender and receiver processes, wrapped up in a context that sets up shared connections, including encryption keys and mac secrets. To obtain the final script for verification, we compose this generated code with a series of handwritten CryptoVerif declarations that model the Crypto module, gathered in the CryptRecord.cv script. This script includes the following declarations.

- Type definitions for the cryptographic computations.
- Game equivalences that embed our security assumptions on cryptographic primitives.
- Target security properties in the form of correspondence assertions and secrecy queries.

Next, we detail these declarations.

Types for Cryptography. We introduce specific types for cryptographic computations. For instance, we have types for macs (macs), mac secrets (mkey), and the seeds for generating mac keys (mkeyseed), and we declare a function `fun mkgen(mkeyseed):mkey` for generating a mac secret from a mac seed (derived from the connection state).

Security Assumptions for Mac, Symmetric Encryption, and PRF. We present our assumptions for the security primitives of the Record protocol. We refer to the libraries included in the CryptoVerif distribution for the corresponding formulations as game equivalences; see also Blanchet [2008] for other protocols that rely on these assumptions.

- *MAC.* The message authentication code scheme has three functions—`mkgen`, `mac`, and `check`—for generating a mac secret from a seed, applying the mac algorithm on a message, and verifying a mac, respectively. For this scheme, we assume *unforgeability under chosen message attacks* (UF-CMA), stated as an equivalence that replaces all calls to `mac` and `check`, such that `check` performs a table lookup on any previously generated macs instead of recomputing it. Blanchet [2008] also relies on this equivalence and relates it to the usual formulation of UF-CMA (Proposition 2).
- *Symmetric Encryption.* The symmetric encryption scheme has three functions—`kgen`, `enc`, and `dec`—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. We assume the notion of *super pseudorandom permutation* (SPRP), introduced by Phan and Pointcheval [2004], entailing that encryption is a random permutation, at least when used with randomly chosen keys. (The “super” qualifier indicates that the adversary also has access to a decryption oracle.) SPRP is modeled as a CryptoVerif equivalence that

replaces every call to encryption and decryption operations by lookups (via the `CryptoVerif find` command) on a table that associates previous encryption and decryption queries with randomly generated values.

- *PRF*. We specify security for each of the pseudo-random functions `prf` and `mprf` in the Random Oracle model [Bellare and Rogaway 1993], as an equivalence that replaces every call to the function by a table lookup, so that the first call generates a fresh random value and subsequent calls return the same value. (Blanchet and Pointcheval [2006] use a similar equivalence for proving the security of a signature scheme.)

Authentication. We consider events `Send'` and `Recv'` variants of `Send` and `Recv` used for symbolic verification in Section 5, where (for technical reasons), instead of the full connection, we only record the connection identifier. We embed our authenticity property as a correspondence query in `CryptoRecord.cv`. The query also relies on `ConnLeak` events that record the identifiers of (possibly) leaked connections. Let *System* range over the script that consists of `CryptoRecord.cv` and the translation of `Connected`, `Record`, and their auxiliary modules, composed with an arbitrary polynomial adversary. `CryptoVerif` automatically proves record authentication on the script through 15 game transformations.

THEOREM 7 (RECORD AUTHENTICATION). *With overwhelming probability, in any run of System, for any Recv' event there is a Send' event with matching record payload, or the connection is corrupted (that is, there is a ConnLeak event for the connection).*

Secrecy. As in Section 5.2, we extend the `Record` module with a function `send'` that generates and sends a freshly generated `record_payload` over a connection. We add this function to the `Record` public interface; we exclude the `recv` function (which would otherwise act as a decryption oracle); and we disable the corruption of connections, obtaining a variant *System'* of the system *System* of Theorem 7. `CryptoVerif` verifies the secrecy of `record_payload` through 14 game transformations.

THEOREM 8 (RECORD SECRECY). *In any run of System', the sequence of values of record_payload is computationally indistinguishable from a sequence of independent, random values.*

In the theorem, computational indistinguishability means that a polynomial adversary can not, with overwhelming probability, distinguish the real secret from a random value. Thus, the theorem asserts that polynomial adversaries gain no information from `record_payloads` sent by client instances of the `Record` protocol.

Differences with the Symbolic Model. Symbolically, it is possible to show secrecy not only for the record payloads, but also for the encryption keys and mac secrets used to protect the payloads. Computationally, however, we show key secrecy only before they are actually used. This is the case for the session keys of the `Record` protocol and for the pre-master secret of the `Handshake` protocol. To prove secrecy after the key is used, for example, as a session key, we would use a weaker notion, such as key usability [Datta et al. 2006]. `CryptoVerif` does not support this notion at present, although some first models [Blanchet et al. 2008] have been tried via a combination of `CryptoVerif` code and then manual reasoning.

Another difference can be seen on the security notion for macs. With `CryptoVerif`, the UF-CMA equivalence only states that the integrity of the maced message is guaranteed, and says nothing on its secrecy. In contrast, symbolic models usually treat macs as perfect one-way constructors, thereby implicitly guaranteeing message secrecy. To

highlight this difference, consider a variant of the Record protocol where, instead of mac-then-encrypting, we encrypt only the payload and keep the mac in the clear.

As we attempt to verify this variant, CryptoVerif fails to prove Theorem 8, since nothing guarantees that the payload remains secret. In contrast, symbolic verification still succeeds.

7.2. Handshake Protocol

To verify the Handshake protocol (module Handshake), we set up an environment containing a certified trusted server. More specifically, we write F# wrapper code in a module Certified that sets up a public/private keypair of a trusted server.

Module Handshake is composed with Certified, as well as with the modules on which Handshake depends upon, most notably the Record module (albeit in null mode, as prescribed by the standard for the initial key setup). We consider two stages which roughly correspond to the phases 2–4 of the Handshake protocol.

First stage. Initially, we consider a client that sends both a ClientKeyExchange (encrypting the pre-master secret) and a Finished message (which macs the message with a derived key from the pre-master secret). In this case, we prove secrecy of the pre-master secret despite it being (1) encrypted for the server and (2) used as part of the key derivation in the subsequent Finished message (we rely both on IND-CCA2 for encryption, as well as a Random Oracle assumption on the PRF, detailed next).

Second stage. Subsequently, we consider a client sending the Finished message and a server processing it. In this case, we prove that the (newly established) master secret is agreed between the client and server.

As done for the Record protocol, using the tool of Section 6, we extract polynomially replicated processes from Certified, Handshake, and Record.

Security Notions for PRF and Asymmetric Encryption. We manually craft CryptoHandshake.cv with our cryptographic declarations and assumptions. We use the same cryptographic types as with the Record protocol and include secrecy and correspondence queries. Finally, CryptoHandshake.cv includes CryptoVerif equivalences embedding our cryptographic assumptions, detailed next.

- *PRF.* To prove secrecy of the master secret we need the PRF to behave as a primitive that completely hides the pre-master secret from the later derived master secret. For this, as we just did with the Record protocol, we specify security of the PRF in the Random Oracle model. However, the PRF is also used as a mac primitive in order to provide integrity of the Finished messages. In order to prove authentication of the derived master secret via the Finished messages, we also assume that the PRF primitive is UF-CMA (as just previously explained for the Record protocol).
- *Asymmetric Encryption.* We have functions skgen and pkgen for creating private and public keys. We also have functions enca and deca 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 of the CryptoVerif libraries, which replaces encrypted plaintexts with a message consisting of only zeroes (of the appropriate length), and replaces decryptions by table lookups.

Verification Result	F# Code	Crypto Assumptions	Games	Time
Record Authentication (Theorem 7)	1967 lines	18 lines	15	1.9 s
Record Secrecy (Theorem 8)	1967 lines	25 lines	14	0.3 s
PMS Random Secrecy (Theorem 9)	2497 lines	33 lines	18	1.1 s
MS Authentication (Theorem 10)	2497 lines	23 lines	8	24 s

Fig. 4. Summary of computational experimental results.

Secrecy for 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 *System''* be the script that consists of CryptoHandshake.cv (embedding the IND-CCA2 and PRF assumptions) and the translation of Certified and Handshake (including the sending of the ClientKeyExchange and Finished messages), composed with an arbitrary polynomial adversary. CryptoVerif verifies the secrecy of random, through 18 game transformations.

THEOREM 9 (PMS RANDOM SECRECY). *In any run of System'', the sequence of random values within pre-master secrets is computationally indistinguishable from a sequence of independent random values.*

Authentication of the Finished messages. By assuming UF-CMA for the PRF when used as a mac, we can also prove the integrity of the master secret agreed upon between the client and server by checking the Finished messages. We let *System'''* be the variant of *System''*, embedding the UF-CMA assumption and including the sending and receipt of the Finished messages. CryptoVerif verifies the necessary correspondence in eight game transformations.

THEOREM 10 (MASTER SECRET AUTHENTICATION). *With overwhelming probability, in any run of System''', for any AcceptFinished event, there is a SendFinished event with a matching master secret.*

Differences with the Symbolic Model. Our secrecy property is close to the symbolic notion of strong secrecy but is finer than syntactic secrecy. For instance, symbolically we have syntactic secrecy for the full pms with the embedded protocol version constant, not just its random part. In future work, it may be possible to computationally verify that the full pms satisfies a weaker secrecy property than ours, following Morrissey et al. [2010].

Our computational theorems are more limited in comparison with the symbolic results. They do not account for resumption and server compromise and only partly handle the composition of the Record and Handshake protocols. Our results are at the limit of what CryptoVerif can achieve, and hence reflect its technical limitations at the time of writing. To give an example, when we compose the Record and Handshake protocols, we find that the resulting model grows far too large for CryptoVerif in just a few transformations. We anticipate that as the tool evolves, future analyses will be able to improve on our results.

Experimental Results. Figure 4 gives the verification times for our theorems. All results are automatically proved by CryptoVerif (with no user interaction). We use FS2CV version 1.1 and CryptoVerif version 1.7. These experiments are performed on a computer with an Intel Pentium D at 3 GHz processor.

7.3. Related Work: 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 use for modeling in CryptoVerif) and their relation to standard semantic security and to security against active attacks.

Our treatment of (symmetric) encryption, encapsulated in the SPRP notion, assumes that exactly one block is encrypted at a time. Plaintext bitstrings are padded toward a (single, large) block that is then encrypted; the IV is assumed to be part of the key and is never updated. Hence, attacks that rely on the CBC mode, or the implementation of padding, or the choice of IVs, do not appear in our model. In practice, this means that the SPRP property may be too strong to be met by concrete implementations of the encryption primitive (e.g., AES). We leave, as future work, a less abstract model of symmetric encryption that accounts for the standard usages of cipher block chaining, padding, and initialization vectors.

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). They study randomness extraction from pre-master secret to master secret in the standard model.

Jonsson and Kaliski [2002] give a security reduction for the security of TLS/SSL when instantiated with RSA-PKCS-1v1.5 (modeling 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. Gajek et al. [2008] present an extended random-oracle model to analyze a mutual authentication protocol built on top of TLS.

Klima et al. [2003] use the version check in the Client-KeyExchange to construct timing attacks over RSA-based sessions. In our model we do not consider side channel attacks. Padding attacks have also been exploited for the TLS protocol, both for asymmetric encryption using PKCS #1 [Bleichenbacher 1998] and for symmetric encryption in CBC mode [Yau et al. 2005]. In both cases the adversary is given an oracle that says whether plaintexts are correctly padded or not. We do not consider padding in our work.

8. CONCLUSIONS AND FUTURE WORK

We have presented the first symbolic and computational verification results for an interoperable TLS implementation. Using an existing model extractor and a mature verification tool, ProVerif, we proved symbolic authentication and secrecy properties for the full implementation of the Record, Handshake, and Resumption protocols. Using a novel model extractor and a recent verification tool, CryptoVerif, we also proved authentication and secrecy properties (in the more demanding computational model) for the core of the Record and Handshake protocols.

The main open area for future work is to extend computational verification to a full TLS implementation. We see three main challenges toward this goal. First, our model extractor and CryptoVerif need to be more scalable to handle the code for all three TLS protocols. Second, to compose the verification of different phases of the protocol, we would need to use a weaker notion of key secrecy, such as key usability [Datta et al. 2006], as used in recent verifications of Kerberos [Blanchet et al. 2008]. Third, our model of symmetric encryption needs to be refined to account for padding, cipher block chaining, and initialization vectors.

Another area of future work is the verification of more security properties and larger applications. Our symbolic results can be extended to verify strong secrecy of keys and

session secrets. Although ProVerif supports such properties, they typically take more time and memory and may require further model abstractions. TLS implementations are used within complex applications, such as Web browsers and Web servers, and the security of a session depends upon user actions, such as the inspection and approval of server certificates. An open problem is to extend our symbolic and computational theorems to account for such applications, including their user interfaces.

Our results show that if a protocol implementation is carefully written, automatically verifying strong security properties against a powerful adversary is technically feasible. Verifying legacy implementations is a natural next step and offers its own new challenges.

ACKNOWLEDGMENTS

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

REFERENCES

- ABADI, M. AND FOURNET, C. 2001. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*. ACM, 104–115.
- BACKES, M. AND LAUD, P. 2006. Computationally sound secrecy proofs by mechanized flow analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. ACM, 370–379.
- BELLARE, M., CANETTI, R., AND KRAWCZYK, H. 1996. Keying hash functions for message authentication. In *Proceedings of the 16th Annual Cryptology Conference on Advances in Cryptology (CRYPTO'96)*. Springer, 1–15.
- BELLARE, M. AND ROGAWAY, P. 1993. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS'93)*. ACM, 62–73.
- BHARGAVAN, K., CORIN, R., FOURNET, C., AND ZĂLINESCU, E. 2009. Automated computational verification for cryptographic protocol implementations. MSR-INRIA Tech. rep. <http://msr-inria.inria.fr/projects/sec/fs2cv>.
- BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. 2006. Verified interoperable implementations of security protocols. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE Computer Society, 139–152.
- BLANCHET, B. 2001. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*. IEEE Computer Society, 82–96.
- BLANCHET, B. 2007. Computationally sound mechanized proofs of correspondence assertions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*. IEEE Computer Society, 97–111.
- BLANCHET, B. 2008. A computationally sound mechanized prover for security protocols. *IEEE Trans. Depend. Secur. Comput.* 5, 4, 193–207.
- BLANCHET, B. AND POINTCHEVAL, D. 2006. Automated security proofs with sequences of games. In *Proceedings of the 26th Annual International Cryptology Conference (CRYPTO'06)*. Lecture Notes in Computer Science, vol. 4117. Springer, 537–554.
- BLANCHET, B., JAGGARD, A. D., SCEDROV, A., AND TSAY, J.-K. 2008. Computationally sound mechanized proofs for basic and public-key Kerberos. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*. ACM, 87–99.
- BLEICHENBACHER, D. 1998. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *Proceedings of the 18th Annual Cryptology Conference on Advances in Cryptology (CRYPTO'98)*. Lecture Notes in Computer Science, vol. 1462. Springer, 1–12.
- CHAKI, S. AND DATTA, A. 2009. Aspier: An automated framework for verifying security protocol implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF'09)*. IEEE Computer Society, 172–185.
- CISCO. 2007. SSL/TLS certificate and SSH public key validation vulnerability. <http://www.cisco.com/warp/public/707/cisco-sa-20070118-certs.shtml>.

- CORIN, R. AND DEN HARTOG, J. 2006. A probabilistic hoare-style logic for game-based cryptographic proofs. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming Part II (ICALP'06)*. Lecture Notes in Computer Science, vol. 4052. Springer, 252–263.
- DATTA, A., DEREK, A., MITCHELL, J. C., AND WARINSCHI, B. 2006. Computationally sound compositional logic for key exchange protocols. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE Computer Society, 321–334.
- DÍAZ, G., CURTERO, F., VALERO, V., AND PELAYO, F. 2004. Automatic verification of the TLS handshake protocol. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC'04)*. ACM, 789–794.
- DIERKS, T. AND ALLEN, C. 1999. The TLS protocol version 1.0.
- DIERKS, T. AND RESCORLA, E. 2006. The Transport layer security (TLS) protocol version 1.1.
- DIERKS, T. AND RESCORLA, E. 2008. The Transport layer security (TLS) protocol version 1.2.
- DIETRICH, S. 1997. A formal analysis of the secure sockets layer protocol. Ph.D. thesis, Adelphi University.
- DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Trans. Inform. Theory* *IT-29*, 2, 198–208.
- FOUQUE, P.-A., POINTCHEVAL, D., AND ZIMMER, S. 2008. HMAC is a randomness extractor and applications to TLS. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*. ACM, 21–32.
- FRIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL protocol version 3.0. Internet Draft, IETF.
- GAJEK, S., MANULIS, M., SADEGHI, A.-R., AND SCHWENK, J. 2008. Provably secure browser-based user-aware mutual authentication over TLS. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*. ACM, 300–311.
- GOUBAULT-LARRECQ, J. AND PARRENNES, F. 2005. Cryptographic protocol analysis on real C code. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*. Lecture Notes in Computer Science, vol. 3385. Springer, 363–379.
- HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. 2005. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM, 2–15.
- HICKMAN, K. E. 1995. The SSL protocol. Draft specification, Netscape.
- JONSSON, J. AND KALISKI, J. B. S. 2002. On the security of RSA encryption in TLS. In *Proceedings of the 22nd Annual International Cryptology Conference (CRYPTO'02)*. Lecture Notes in Computer Science, vol. 2442. Springer, 127–142.
- JÜRJENS, J. 2006. Security analysis of crypto-based java programs using automated theorem provers. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE Computer Society, 167–176.
- KAMIL, A. AND LOWE, G. 2008. Analysing TLS in the strand spaces model. Tech. rep., Oxford University Computing Laboratory.
- KLIMA, V., POKORNY, O., AND ROSA, T. 2003. Attacking RSA-based sessions in SSL/TLS. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*. Lecture Notes in Computer Science, vol. 2779. Springer, 426–440.
- KRAWCZYK, H. 2001. The order of encryption and authentication for protecting communications (or How secure is SSL?). In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'01)*. Lecture Notes in Computer Science, vol. 2139. Springer, 310–331.
- LAUD, P. 2005. Secrecy types for a simulatable cryptographic library. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM, 26–35.
- MITCHELL, J. C., SHMATIKOV, V., AND STERN, U. 1998. Finite-state analysis of SSL 3.0. In *Proceedings of the 7th USENIX Security Symposium (SSYM'98)*. USENIX Association, 201–216.
- MORRISSEY, P., SMART, N. P., AND WARINSCHI, B. 2010. The TLS handshake protocol: A modular analysis. *J. Cryptology* *23*, 2, 187–223.
- NEEDHAM, R. AND SCHROEDER, M. 1978. Using encryption for authentication in large networks of computers. *Comm. ACM* *21*, 12, 993–999.
- OGATA, K. AND FUTATSUGI, K. 2005. Equational approach to formal analysis of TLS. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICSCS'05)*. IEEE Computer Society, 795–804.
- OPENSSL. 2009. EVP_VerifyFinal function signature verification vulnerability. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-5077>.
- PAULSON, L. C. 1999. Inductive analysis of the Internet protocol TLS. *ACM Trans. Inf. Syst. Secur.* *2*, 3, 332–351.

- PHAN, D. H. AND POINTCHEVAL, D. 2004. About the security of ciphers (semantic security and pseudo-random permutations). In *Proceedings of the 11th International Workshop on Selected Areas in Cryptography (SAC'04)*. Lecture Notes in Computer Science, vol. 3357. Springer, 182–197.
- TSAHHIROV, I. AND LAUD, P. 2007. Application of dependency graphs to security protocol analysis. In *Proceedings of the 3rd Symposium on Trustworthy Global Computing*. Lecture Notes in Computer Science, vol. 4912. Springer, 294–311.
- WAGNER, D. AND SCHNEIER, B. 1996. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (WOEC'96)*. USENIX Association, 29–40.
- YAU, A. K. L., PATERSON, K. G., AND MITCHELL, C. J. 2005. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In *Fast Software Encryption*, Springer, 299–319.

Received March 2009; revised April 2011; accepted June 2011