Implementing TLS with
Verified Cryptographic Security

Karthikeyan Bhargavan ∗1, Cédric Fournet †2, Markulf Kohlweiss †2, Alfredo Pironti §1, and Pierre-Yves Strub ¶3

1INRIA
2Microsoft Research
3IMDEA Software

Draft, March 2013

Abstract

TLS is possibly the most used protocol for secure communications, with a 18-year history of flaws and fixes, ranging from its protocol logic to its cryptographic design, and from the Internet standard to its diverse implementations.

We develop a verified reference implementation of TLS 1.2. Our code fully supports its wire formats, ciphersuites, sessions and connections, re-handshakes and resumptions, alerts and errors, and data fragmentation, as prescribed in the RFCs; it interoperates with mainstream web browsers and servers. At the same time, our code is carefully structured to enable its modular, automated verification, from its main API down to computational assumptions on its cryptographic algorithms.

Our implementation is written in F# and specified in F7. We present security specifications for its main components, such as authenticated stream encryption for the record layer and key establishment for the handshake. We describe their verification using the F7 type-checker. To this end, we equip each cryptographic primitive and construction of TLS with a new typed interface that captures its security properties, and we gradually replace concrete implementations with ideal functionalities. We finally typecheck the protocol state machine, and obtain precise security theorems for TLS, as it is implemented and deployed. We also revisit classic attacks and report a few new ones.

∗karthikeyan.bhargavan@inria.fr
†fournet@microsoft.com
‡markulf@microsoft.com
§alfredo.pironti@inria.fr
¶pierre-yves@strub.nu
Contents

1 Introduction 3
  1.1 Transport Layer Security ........................................ 4
  1.2 Compositional, Automated Verification .......................... 5

2 A Modular Implementation of TLS 6
  2.1 API Overview ...................................................... 6
  2.2 Modules and Interfaces .............................................. 7
  2.3 Modular Architecture for TLS .................................. 7
  2.4 Experimental Evaluation ........................................ 9

3 Cryptographic Security by Typing 11
  3.1 Games, Ideal Functionalities, and Typed Interfaces .......... 12
  3.2 Indexes for Multi-instance, Agility, and Corruption ....... 15

4 Authenticated Encryption for TLS Streams 17
  4.1 Traffic Analysis and Length Hiding ............................ 18
  4.2 Authenticated Encryption Schemes ............................... 19
  4.3 Related Work on Authenticated Encryption .................. 23

5 The Handshake Protocol 23
  5.1 The Handshake Interface .......................................... 24
  5.2 Handshake Security and Modular Verification ............... 27
  5.3 Key distribution for TLS master secrets ...................... 28
  5.4 Related Work on Key Exchange ................................. 36

6 Main API & Theorems for TLS 36
  6.1 TLS API ............................................................ 36
  6.2 TLS Security ........................................................ 38
  6.3 Security for 'untyped' adversaries .............................. 39
  6.4 Verified TLS Applications ...................................... 40

7 Limitations and Future Work 41
1 Introduction

Transport layer security (TLS) is possibly the most used security protocol; it is widely deployed for securing web traffic (HTTPS) and also mails, VPNs, and wireless communications. Reflecting its popularity, the security of TLS has been thoroughly studied, with a well-documented, 18-year history of attacks, fixes, upgrades, and proposed extensions [e.g. Freier et al., 2011, Dierks and Allen, 1999, Dierks and Rescorla, 2006, 2008, Rescorla et al., 2010, Langley and Moeller, 2010]. Some attacks target the protocol logic, for instance causing the client and server to negotiate the use of weak algorithms even though they both support strong cryptography [Langley, 2011]. Some exploit cryptographic design flaws, for instance using knowledge of the next IV to set up adaptive plaintext attacks [Moeller, 2004]. Others, such as padding-oracle attacks, use a combination of protocol logic and cryptography, taking advantage of error messages to gain information on encrypted data [Vaudenay, 2002, Canvel et al., 2003, Yau et al., 2005]. Others rely on various implementation errors [Bleichenbacher, 1998, Lawall et al., 2010, Klima et al., 2003] or side channels [Brumley and Boneh, 2003]. Further attacks arise from the usage or configuration of TLS, rather than the protocol itself, for instance exploiting poor certificate management or gaps between TLS and the application logic [Ray, 2009, Georgiev et al., 2012]. Overall, the mainstream implementations of TLS still require several security patches every year.

Meanwhile, TLS security has been formally verified in many models, under various simplifying assumptions [Paulson, 1999, Díaz et al., 2004, He et al., 2005, Ogata and Futatsugi, 2005, Morrissey et al., 2008, Gajek et al., 2008, Kamil and Lowe, 2008, Jager et al., 2012]. While all these works give us better confidence in the abstract design of TLS, and sometimes reveal significant flaws, they still ignore most of the details of RFCs and implementations.

To achieve provable security for TLS as it is used, we develop a verified reference implementation of the Internet standard. Our results precisely relate application security at the TLS interface down to cryptographic assumptions on the algorithms selected by its ciphersuites. Thus, we address software security, protocol security, and cryptographic security in a common implementation framework. In the process, we revisit known attacks and discover new ones: an alert fragmentation attack (§2), and a fingerprinting attack based on compression (§4). Our two main goals are as follows:

(1) Standard Compliance Following the details of the RFCs, we implement and verify the concrete message parsing and processing of TLS. We also support multiple versions (from SSL 3.0 to TLS 1.2) and ciphersuites, protocol extensions, sessions and connections (with re-handshakes and resumptions), alerts and errors, and data fragmentation.

The TLS standard specifies the messages exchanged over the network, but not its application programming interface (API). Since this is critical for using TLS securely, we design our own API, with an emphasis on precision—our API is similar to those provided by popular implementations, but gives more control to the application, so that we can express stronger security properties: §4 explains how we reflect fragmentation and length-hiding, to offer some protection against traffic analysis; §6 explain how we report warnings, changes of ciphersuites, and certificate requests.

We illustrate our new API by programming and verifying sample applications. We also implement .NET streams on top of it, and program minimal web clients and servers, to confirm that our implementation interoperates with mainstream implementations, and that it offers reasonable usability and performance. (In contrast, most verified models are not executable, which precludes even basic functionality testing.) Experimentally, our implementation also provides a convenient platform for testing corner cases, trying out potential attacks, and analyzing proposed extensions and security patches. In the course of this work, we submitted errata to the IETF\(^1\).

(2) Verified Security  Following the provable security approach of computational cryptography, we show the privacy and integrity of bytestreams sent over TLS, provided their connection keys were established using a strong ciphersuite between principals using secure long-term keys. Unavoidably, an active adversary may observe and disrupt encrypted network traffic below TLS. In brief, our main results show that a probabilistic, polynomial adversary cannot achieve more, except with a negligible probability: even with chosen adaptive plaintext and ciphertext bytestreams, it learns nothing about the content of their communication, and cannot cause them to accept any other content. These results are expressed using indistinguishability games, whereby the communication content is replaced with zeros before sending, and restored by table lookups after receiving.

Thus, we achieve the kind of cryptographic results traditionally obtained for secure channels, but on an unprecedented scale, for an executable, standard-compliant, 5,000-line functionality, rather than an abstract model of TLS—dozens of lines in pseudocode in Jager et al. [2012, fig. 3] and Gajek et al. [2008, p. 4]. In the process of verifying our implementation, we also establish functional properties, logical authentication goals, and state machine invariants.

In the rest of this section, we summarize the challenges involved in achieving our goals, namely accounting for the complexity of TLS, and automatically verifying a large implementation with precise cryptographic guarantees.

1.1 Transport Layer Security

TLS is an assembly of dynamically-configured protocols, controlled by an internal state machine that calls into a large collection of cryptographic algorithms. (§2 reviews the TLS architecture.) This yields great flexibility for connecting clients and servers, potentially at the cost of security, so TLS applications should carefully configure and review their negotiated connections before proceeding. Accordingly, we prove security relative to the choice of protocol version, ciphersuite, and certificates of the two parties.

Versions, Ciphersuites, and Algorithms  Pragmatically, TLS must maintain backward compatibility while providing some security. Indeed, 5 years after the release of TLS 1.2, which fixes several security weaknesses, RC4 remains the most popular cipher, most browsers still negotiate TLS 1.0, and many still accept SSL2 connections! It is thus crucial to assess the security of TLS as a whole, even if its usage of cryptography is outdated. As most implementations do, our codebase supports all protocol versions from SSL 3.0 till TLS 1.2 [Freier et al., 2011, Dierks and Allen, 1999, Dierks and Rescorla, 2006, 2008]. We decided not to support SSL2 at all, since its usage is unsafe and now prohibited [Turner and Polk, 2011].

Many algorithms, such as MD5, DES, or PKCS#1, are eventually broken or subsumed by others, so TLS features cryptographic agility, enabling users to choose at runtime between different methods and algorithms for similar purposes. Ciphersuites and extensions are its main agility mechanisms; together with the protocol version, they control the method and algorithms for the key exchange and the transport layer. Older ciphersuites can be very weak, but even the latest ciphersuites may not guarantee security: as a cautionary tale, Brumley et al. [2011] report, exploit and fix a “bug attack” in the implementation of elliptic-curve multiplication within OpenSSL, which left many advanced ciphersuites exposed to attacks for years. Accordingly, our formal development fully supports cryptographic agility, in the spirit of Acar et al. [2010], and provides security relative to basic cryptographic assumptions (say, IND-CPA or PRF) on the algorithms chosen by the ciphersuite. Thus, we obtain security for connections with strong ciphersuites running side-by-side with insecure connections with weak ciphersuites.

Side Channels and Traffic Analysis  Our API provides fine-grained control for fragmentation and padding; this enables applications to control the amount of information they leak via network traffic analysis. Our verification also explicitly handles many runtime errors, thus reflecting their potential use to leak secret information. Thus, our verification catches the padding oracle
attack of TLS 1.0 [Vaudenay, 2002, Canvel et al., 2003] as a type-abstraction error. We also
independently caught the truncated-MAC attack reported by Paterson et al. [2011].

On the other hand, our verification does not account for timing. Following the standard,
we only try to mitigate known timing channels by having a uniform flow, for instance ensuring
that the same cryptographic operations are performed, both in normal execution and in error
conditions.

1.2 Compositional, Automated Verification

To cope with the complexity of TLS and prove security on a large amount of code, we rely both
on compositionality and on automation. We extend the cryptographic verification by typing
approach of Fournet et al. [2011]. The main technical novelty is to keep track of conditional
security using type indexes (see §3.2). For instance, the index of a TLS connection includes the
algorithms and certificates used to establish the connection, so that we can specify the security
of each connection relative to this context. Cryptographically, indexes are similar to session
identifiers in the universal composability (UC) framework. Another central idea is to rely on
type abstraction to specify confidentiality and integrity, enabling us to express our main security
properties in just a few lines of typed declarations.

The core contribution of this work is our modular implementation of TLS and our com-
positional verification approach. Our presentation focuses on the main API and the interfaces
of two core internal modules. The stateful authenticated encryption module (StAE), explained in §4, implements record-layer cryptography. For modes based on block ciphers, we provide
length-hiding features as proposed and analyzed by Paterson et al. [2011]. The handshake mod-
ule (HS) implements the key exchange mechanisms of TLS. We specify ideal typed interfaces
for StAE and HS that suffice to prove application-level security for TLS. Our main formal con-
tributions are to verify that the record layer securely implements the StAE interface for a range
of authenticated encryption mechanisms (Theorem 4 in §4); the handshake protocol implements
the HS interface, with security guarantees when using RSA and DH (Theorem 5 in §5); and
the TLS protocol logic, dealing with application data, alerts, and multiple connections, securely
implements our main API, given any secure implementations of StAE and HS (Theorem 6 in
§6).

Prior Verification Work on TLS Implementations. We limit our discussion of related work to
the verification of implementations; other works on formal aspects of TLS are discussed through
the paper. To our knowledge, Bhargavan et al. [2012] present the only prior computational
security theorems for a TLS implementation. They conduct extensive verification of the protocol
logic by model extraction from F# to ProVerif [Blanchet, 2001] and CryptoVerif [Blanchet,
2006]. On the other hand, their Dolev-Yao models do not cover binary formats (excluding any
bytestream, fragmentation and padding issue), nor the properties of the underlying algorithms,
and their computational models cover only the cryptographic core of one ciphersuite. Their
results are less precise than ours (notably as regards secrecy) and blind to the cryptographic
weaknesses of TLS 1.0.

Chaki and Datta [2009] verify the SSL 2.0/3.0 handshake implementation in OpenSSL for
authentication and secrecy properties by model checking. Their analysis finds rollback attacks
but applies only to fixed configurations, and they assume a symbolic model of cryptography.
Others [Jürjens, 2006, Avalle et al., 2011] verify Java implementations of the handshake protocol
using logical provers, also in the symbolic model.

Contents The paper is organized as follows. §2 informally presents and evaluates our modular
reference implementation. §3 explains our approach to cryptographic verification by typing. §4
handles length-hiding stream encryption. §5 deals with the handshake. §6 presents our main
API and theorems for TLS. §7 discusses limitations of our approach and future work.

TLS is large and complicated, and so is any formal security statement on its implementa-
tion. We strive to give a precise description of our results using sample code and interfaces,
Figure 1: Modular implementation of TLS

but we necessarily omit many details. We refer to the standard for a complete protocol description, and to our full development at http://mitls.rocq.inria.fr/ for the annotated source code, a companion paper with additional cryptographic assumptions and proofs, and a discussion of attacks.

2 A Modular Implementation of TLS

2.1 API Overview

Our application interface (see Fig. 13 in §6) is inspired by typical APIs for TLS libraries and provides similar functionalities. It is thread safe, and does not allocate any TLS-specific thread, essentially leaving scheduling and synchronization in the hands of the application programmer. Cryptographically, we can thus treat our whole implementation as a probabilistic polynomial time (p.p.t.) module, to be composed with a main p.p.t. program representing the adversary.

Our reference implementation consists of a dynamically linked library (DLL) with an interface TLSInfo that declares various types and constants, e.g. for ciphersuites, and a main interface TLS for controlling the protocol. To use it, the application programmer provides a DataStream module that uses TLSInfo and defines the particular streams of plaintext application data he intends to communicate over TLS, and a main program that calls TLS. In addition, application code may use any other libraries and export its own interfaces.

Application code may create any number of TLS connections, as client or server, by providing some TCP connection and some local configuration that indicates versions, ciphersuites and certificates to use, and sessions to re-use. Our API returns a stateful connection endpoint (with an abstract type) that can then be used by the application to issue a series of commands, such as
read and write to communicate data once the connection is opened, rekey and rehandshake to trigger a new handshake, and shutdown to close the connection. Each command returns either a result, for instance the data fragment that has been read, or some event, for instance an alert, a certificate authorization request, or a notification that the current handshake is complete. At any point, the application can read the properties of its connection endpoints, which provide detailed local information about the current ciphersuites, certificates, and security parameters, bundled in a datatype named an epoch. A given connection may go through a sequence of different epochs, separated by complete handshakes, each with their own security parameters, so the application would typically inspect the new connection epoch when notified that the handshake is complete, and before issuing a write command for sending any secret data.

2.2 Modules and Interfaces

Our implementation is written in F#, a variant of ML for the .NET platform, and specified in F7 [Bengtson et al., 2011]. It is structured into 45 modules (similar to classes or components in other languages) each with an interface and an implementation. Each interface declares the types and functions exported by the module, copiously annotated with their logical specification.

We informally present the verification approach developed in the next sections. We use interfaces to specify the security properties of our modules and to control their composition. In particular, §3 explains how we use interfaces to express various cryptographic properties.

The F7 typechecker can verify each module independently, given as additional input a list of interfaces the module depends on. Assuming the specification in these interfaces, F7 verifies the module implementation and checks that it meets the specification declared in its own interface. Both tasks entail logical proof obligations, which are automatically discharged by calling Z3 [de Moura and Bjørner, 2008], an SMT solver. Our ‘makefile’ automates the process of verifying modules while managing their dependencies.

After verification, all F7 types and specifications are erased, and the module can be compiled by F#.

Our type-based cryptographic verification consists of a series of idealization steps, one module at a time, starting from nonces and the authentication of the public-key materials till the use of the record keys for authenticated encryption. The numbers in Fig. 1 indicate the order of idealization. Each step is conditioned by cryptographic assumptions and typing conditions, to ensure its computational soundness; it enables us to replace a concrete module implementation by a variant with stronger security properties; this variant can then be re-typechecked, to show that it implements a stronger ideal interface, which in turn enables further steps. Finally, we conclude that the idealized variant of our TLS implementation is both perfectly secure (by typing) and computationally indistinguishable from our concrete TLS implementation.

2.3 Modular Architecture for TLS

Fig. 1 gives our software architecture for TLS. Each box is an F# module, specified by a typed interface. These modules are (informally) grouped into components.

In the Base component, Bytes wraps low-level, trusted .NET primitive operations on byte arrays, such as concatenation; TCP handles network sockets, and it need not be trusted; Core-Crypto is our interface to trusted core algorithms, such as the SHA1 hash function and the AES block cipher; it can use different cryptographic providers such as .NET or Bouncy Castle. Other modules define constants, ciphersuite identifiers and binary formats; these modules are fully specified and verified. TLSInfo defines public data structures for sessions, connections and epochs (see §5) giving access for instance to the negotiated session parameters.

The TLS protocol is composed of two layers. The record layer is responsible for the secrecy and authenticity of individual data fragments, using the authenticated encryption mechanisms
described in §4. It consists of several modules: Record is TLS-specific and deals with headers and content types, whereas StAE, LHAE and ENC provide agile encryption functionalities, each parameterized by a plaintext module, as explained in §3. Finally, MAC provides various agile MAC functions on top of CoreCrypto and implements the ad hoc keyed hash algorithms of SSL 3.

The upper layer consists of four sub-protocols, respectively dealing with the handshake, change-cipher-spec signals (CCS), alerts, and application data. The Dispatch module interleaves the outgoing messages sent by these sub-protocols into a single stream of fragments, tagged with their content type, possibly splitting large messages into multiple fragments, and conversely dispatches incoming fragments to these protocols, depending on their content type. Not all possible message interleavings are valid; for instance application data should never be sent or accepted before the first handshake successfully completes (establishing a secure channel), and no data should be delivered after receiving a fatal alert. Except for these basic rules, the RFC does not specify valid interleavings; this complicates our verification and, as illustrated below, enables subtle attacks when combined with fragmentation. Dispatch relies on a state machine to enforce the safe multiplexing of sub-protocols; to this end, each sub-protocol signals any significant change in its own internal state. For instance, the handshake protocol signals the availability of new keys, the sending of its Finished message, and its successful completion. To our knowledge, our model is the first to account for this important aspect of TLS implementations. (In contrast, most handwritten cryptographic models cover a single, sequential trace of inputs and outputs for each role of the protocol—for instance an initial session establishment followed by a single data connection.)

The handshake protocol, detailed in §5, negotiates the connection parameters (such as protocol versions, ciphersuites, and extensions) and establishes the shared keys for the record layer. To this end, it relies on generic PRF modules and key exchange algorithms (e.g. RSA-based encryption and Diffie-Hellman exchange). In the TLS terminology, a session identifies a set of security parameters, the peers, and a shared master secret. Each full handshake yields a new session, with its own master secret. Instead, an abbreviated handshake resumes an existing session, retrieving its master secret from a local database. In both cases (full or abbreviated), a new epoch begins, with keys derived from the master secret together with some fresh random values. The same connection may rely on several successive epochs to refresh keys, or to achieve stronger peer authentication, possibly with different security properties. Conversely, several connections may resume from the same session. (For example, most web browsers open concurrent connections for getting the files that constitute a page.)

The alert protocol handles warnings and fatal errors; it tells the dispatcher when to close a connection.

The application-data protocol handles messages on behalf of the TLS application; it is parameterized by a DataStream module provided by the application.

At the toplevel, TLS implements our main API, described in §6. Like other mainstream APIs, it is designed to hide most internal details, while providing enough control to the application. The API is event based, meaning that each time the user invokes a function, the returned value can notify the user that an event occurred. It signals any security-relevant out-of-band events, for example explicitly asking for certificate authorization, or notifying a change of epoch. Simpler, more abstract interfaces may be programmed on top of it, for instance to implicitly handle (or reject) re-handshakes. Before evaluating our implementation, we discuss two attacks involving fragmentation and multiple epochs.

Renegotiating Peer Identities (an existing attack) Ray [2009] presents an attack exploiting the mis-attribution of application data to epochs. Until a a recent protocol extension [Rescorla et al., 2010], TLS did not cryptographically link successive epochs on the same connection: as each handshake completes, the two parties agree on the new epoch, but not necessarily on prior epochs. Their man-in-the-middle attack proceeds as follows: when a target client tries to connect to a server, the attacker holds the client connection, performs a handshake with the
server, sends some (partial) message to the server, then forwards all client-server traffic. As
the client completes its first handshake, the server instead enters its second epoch. If the server
ignores the change of epoch, then it will treat the message injected by the attacker concatenated
with the first message of the client as a genuine message of the client.

Surprisingly, existing TLS APIs have no reliable mechanism to notify epoch changes, even
when the peer identity changes. Instead, the extension implicitly authenticates prior epochs
in Finished messages [Rescorla et al., 2010]. We implement this extension, and in addition,
our API immediately notifies any epoch change, and separately tracks application data from
different epochs.

**Alert fragmentation (a new attack)** We discovered another, similar interleaving attack, against
all versions of TLS, this time involving the alert protocol. Unlike application data, alert mes-
sages can be sent and received before completing the first handshake. Unlike handshake mes-
sages, alert messages are not included in the Finished message computation. Alert messages
are two bytes long, hence they can also be fragmented by the attacker. Our attack proceeds as
follows: when a client-server connection begins, the attacker injects a one-byte alert fragment
\(x\) during the first handshake; according to the standard, this byte is silently buffered; any time
later, after completion of the handshake, as the first genuine 2-byte alert message \(yz\) is sent on
the secure connection, the alert \(xy\) is received and processed instead. This clearly breaks alerts
authentication.

Experimentally, we confirmed that at least OpenSSL is subject to this attack, transforming
for instance a fatal error or a connection closure into an ignored warning, while other imple-
mentations reject fragmented alerts—a simple fix, albeit against the spirit of the standard. Our
implementation simply checks that the alert buffer is empty when a handshake completes, and
otherwise returns a fatal error.

### 2.4 Experimental Evaluation

Our implementation currently supports the protocol versions, algorithms, and extensions listed
in Fig. 2(a), and hence all the ciphersuites obtained by combining these algorithms. Conversely,
our implementation does not yet support elliptic curve algorithms, AEAD ciphers such as AES-
GCM, most TLS extensions, or TLS variants such as DTLS.

**Interoperability** We tested interoperability against the command line interface of OpenSSL
### Ciphersuite Performance

<table>
<thead>
<tr>
<th>Ciphersuite</th>
<th>KEX</th>
<th>Enc</th>
<th>MAC</th>
<th>F# (BC)</th>
<th>OpenSSL</th>
<th>Oracle JSSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HS/s</td>
<td>MiB/s</td>
<td>HS/s</td>
<td>MiB/s</td>
<td>HS/s</td>
<td>MiB/s</td>
</tr>
<tr>
<td>RSA RC4 MD5</td>
<td>305.25</td>
<td>30.17</td>
<td>292.04</td>
<td>226.51</td>
<td>431.66</td>
<td>53.34</td>
</tr>
<tr>
<td>RSA RC4 SHA</td>
<td>291.37</td>
<td>27.85</td>
<td>288.74</td>
<td>232.42</td>
<td>446.69</td>
<td>39.65</td>
</tr>
<tr>
<td>RSA 3DES SHA</td>
<td>267.09</td>
<td>8.40</td>
<td>283.04</td>
<td>22.95</td>
<td>421.59</td>
<td>8.34</td>
</tr>
<tr>
<td>RSA AES128 SHA</td>
<td>278.71</td>
<td>18.54</td>
<td>285.35</td>
<td>234.41</td>
<td>419.20</td>
<td>27.58</td>
</tr>
<tr>
<td>RSA AES128 SHA256</td>
<td>278.71</td>
<td>16.50</td>
<td>281.92</td>
<td>128.33</td>
<td>432.70</td>
<td>23.69</td>
</tr>
<tr>
<td>RSA AES256 SHA</td>
<td>291.37</td>
<td>16.86</td>
<td>282.89</td>
<td>204.47</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RSA AES256 SHA256</td>
<td>267.09</td>
<td>15.16</td>
<td>307.72</td>
<td>119.42</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DHE 3DES SHA</td>
<td>20.16</td>
<td>8.37</td>
<td>58.07</td>
<td>22.99</td>
<td>45.72</td>
<td>8.21</td>
</tr>
<tr>
<td>DHE AES128 SHA</td>
<td>20.41</td>
<td>18.59</td>
<td>57.06</td>
<td>244.30</td>
<td>46.08</td>
<td>27.72</td>
</tr>
<tr>
<td>DHE AES128 SHA256</td>
<td>19.99</td>
<td>16.45</td>
<td>58.33</td>
<td>128.34</td>
<td>45.03</td>
<td>23.84</td>
</tr>
<tr>
<td>DHE AES256 SHA</td>
<td>20.29</td>
<td>16.72</td>
<td>56.83</td>
<td>203.01</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DHE AES256 SHA256</td>
<td>20.16</td>
<td>14.86</td>
<td>59.52</td>
<td>120.96</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3: Performance benchmarks (OpenSSL 1.0.1e as server).

### Performance Evaluation

We evaluate the performance of our implementation, written in F# and linked to the Bouncy Castle C# cryptographic provider, against two popular TLS implementations: OpenSSL 1.0.1e, written in C and using its own cryptographic libraries, and Oracle JSSE 1.7, written in Java and using the SunJSSE cryptographic provider. Our code also consistently outperforms the rudimentary TLS client distributed with Bouncy Castle.

We tested clients and servers for each implementation against another, running on the same host to minimize network effects. Figure 3 reports our results for different clients and ciphersuites with OpenSSL as server. We measured (1) the number of Handshakes completed per second; and (2) the average throughput provided on the transfer of a 400 MB random data file. (Server-side results are similar.) For RSA key exchange, our implementation has a handshake rate similar to that of OpenSSL but slower than Oracle JSSE. Our throughput is significantly lower than OpenSSL and is closer to Oracle JSSE. The numbers for throughput and for DHE key exchanges are closely linked to the underlying cryptographic provider, and we pay the price of using Bouncy Castle’s managed code. (Using instead the .NET native provider increases the throughput by 20% but hinders portability.)

Our reference implementation is designed primarily for modular verification, and has not (yet) been optimized for speed. Notably, our code relies on naïve data structures that facilitate their specification. For example, we represent bytes using functional arrays, which involve a lot of dynamic allocation and copying as record fragments are processed. A trusted library implementing infix pointers to I/O buffers with custom memory management would improve performance, with minimal changes to our verified code, but we leave such optimizations as future work.

### Code Size and Verification Time

Compared with production code, our implementation is smaller; it has around 5 KLOC excluding comments, compared with about 50 KLOC for OpenSSL (only TLS code) and 35 KLOC for Oracle JSSE. This different is due partly to the fact that we support fewer ciphersuites and extensions; the rest can be attributed to the brevity of F# code. Still, we believe ours is the first cryptographic verification effort at this scale. Fig. 2(b)
gives the size of each component in our implementation, the size of its F7 specification, and the
verification time for the typechecked components. Overall, typechecking the whole implemen-
tation takes 15 minutes on a modern desktop.

3 Cryptographic Security by Typing

We verify TLS using F7, a refinement typechecker for F#. In addition to ordinary type safety
(preventing e.g. any buffer overflow) it enables us to annotate types with logical specifications
and to verify their consistency by typing. Its core type system Bengtson et al. [2011] has been
extended in several directions Bhargavan et al. [2010], Swamy et al. [2011], Backes et al. [2009,
2010, 2012]; in particular Swamy et al. [2011] provide a mechanized theory for a language
that subsumes F7. We follow the notations and results of its probabilistic variant Fournet et al.
[2011], presented below.

F7 Types A program is a sequential composition of modules, written \( A_1 \cdot A_2 \cdot \ldots \cdot A_n \). Each
module has a typed interface that specifies the types, values, and functions it exports. A module
is well-typed, written \( I_1, \ldots, I_l \vdash A \sim I \), when it correctly implements \( I \) using modules with
interfaces \( I_1, \ldots, I_l \). A program is well-typed when its modules are well-typed in sequence.

The core typing judgment \( I \vdash e : t \) states that expression \( e \) has type \( t \) in typing environment \( I \).

Types \( t \) include standard F# types like integers, references, arrays and functions, plus refinement
types and abstract types.

Logical refinements Let \( \phi \) range over first-order logical formulas on F# values. The refinement
type \( x : \{ \phi \} \) represents values \( x \) of type \( t \) such that formula \( \phi \) holds (the scope of \( x \) is \( \phi \)). For
instance, \( n : \text{int}\{0 \leq n\} \) is the type of positive integers. Formulas may use logical functions and
predicates, specified in F7 interfaces or left uninterpreted. For instance, let ‘bytes’ abbreviate
the type of byte arrays in F#; its refinement \( b : \text{bytes}\{\text{Length}(b) = 16\} \), the type of 16-byte arrays,
uses a logical function \( \text{Length} \) on bytes. and, to verify that byte arrays have this type, it may be
enough to specify \( \text{Length} \) for empty arrays and concatenations. Refinements may specify data
formats as above (for integrity) and also track runtime events (for authenticity). For instance,
\( c : \text{cert}\{\text{Authorized}(u,c)\} \) may represent an X.509 certificate that user \( u \) has accepted by clicking on
a button. Formally, such security events are introduced as logical assumptions (assume \( \phi \)) in F#
code and F7 interfaces; conversely, they may appear in verification goal, expressed as assertions
(assert \( \phi \)). Logical specifications and assumptions must be carefully written and reviewed, since
they condition our security interpretation of types [see e.g. Bhargavan et al., 2010, Swamy et al.,
2011].

Abstract Types An interface may declare a type as abstract (e.g. type key) and keep its repre-
sentation private (e.g. 16-byte arrays); typing then ensures that any module using this interface
will treat key values as opaque, thereby preserving their integrity and secrecy. Conversely,
the module that implements key would include a concrete type declaration, e.g. type key = b:bytes
\( \{\text{Length}(b) = 16\} \), and use it to implement the rest of the interface. Besides, abstract types may
themselves be indexed by values, e.g. type (id:t)key is the type of keys indexed by a value id
of type \( t \), which may indicate the usage of those keys; typing then guarantees that any module
using the interface won’t mix keys for different usages.

The rest of the type system tracks refinements and abstract types. For example, the de-
dependent function type \( x : \{ \phi \} \rightarrow y:t \{ \phi' \} \) represents functions with pre-condition \( \phi \) and post-
condition \( \phi' \) (the scope of \( x \) is \( \phi \), \( t' \) and \( \phi' \)), and both \( t \) and \( t' \) may be indexed abstract types. We will see various examples in the types for authenticated encryption below.

Safety and Perfect Secrecy in F7 (Review) Fournet et al. [2011] formalize a probabilistic
variant of F7 and develop a framework for the modular cryptographic verification of protocols
coded in F#. (Küsters et al. [2012] adopt a similar approach for programs in Java.) We recall
their main theorems.
A program is safe if, in every run of the program, every assert logically follows from prior assumes. The main property of the type system is that well-typed expressions are always safe.

**Theorem 1 (Type Safety [Fournet et al., 2011])** If \( \varnothing \vdash A : t \), then \( A \) is safe.

Perfect secrecy is specified as probabilistic equivalence: two expressions \( A_0 \) and \( A_1 \) are equivalent, written \( A_0 \approx A_1 \), when they return the same distribution of values. We use abstract types to automatically verify secrecy, as follows. Suppose a program is written so that all operations on secrets are performed in a pure (side-effect free) module \( P \) that exports a restrictive interface \( I_\alpha \) with an abstract type \( \alpha \) for secrets (concretely implemented by, say, a boolean). By typing, the rest of the program can still be passed secrets, and pass them back to \( P \), but cannot directly access their representation. For instance, the rest of the program can never branch on a secret value. With suitable restrictions on \( I_\alpha \), the result of the program then does not depend on secrets and their operations:

**Theorem 2 (Secrecy by Typing [Fournet et al., 2011])** If \( \varnothing \vdash P_b \rightsquigarrow I_\alpha \) for \( b = 0, 1 \) and \( I_\alpha \vdash A : \text{bool} \), then \( P_b \cdot A \approx P_1 \cdot A \).

Intuitively, the program \( A \) interacts with different secrets, kept within \( P_0 \) or \( P_1 \), but it cannot distinguish between the two.

Theorem 2 generalizes from single types \( \alpha \) to families of indexed types, intuitively with a separate abstract type at every index. The formal details are beyond the scope of this paper; we refer to Swamy et al. [2011] for a similar development.

In Theorems 1 and 2, the module \( A \) may be composed of libraries for cryptographic primitives and networking, protocol modules, and the adversary. This adversary can be treated as an untrusted ‘main’ module, simply typed in F#, without any refinement or abstract type. In contrast, the internal composition and verification of the other modules of the program can rely on and are in fact driven by typed F7 interfaces.

**Asymptotic Safety and Secrecy** To model computational security for cryptographic code, [Fournet et al., 2011] also defines asymptotic notions of safety and secrecy for expressions \( A_\eta \) parameterized by a security parameter \( \eta \), which is treated as a symbolic integer constant and is often kept implicit, writing \( A \) instead of \( (A_\eta)_{\eta \geq 0} \). Asymptotic safety states that the probability of an assertion failing in \( A_\eta \) is negligible. The corresponding secrecy notion is stated in terms of asymptotic equivalence: two closed boolean expressions \( A_0 \) and \( A_1 \) (implicitly indexed by \( \eta \)) are asymptotically equivalent, written \( A_0 \approx_\epsilon A_1 \), when the statistical distance

\[
\frac{1}{2} \sum_{M = \text{true, false}} |Pr[A_0 \triangle M] - Pr[A_1 \triangle M]| \leq \epsilon
\]

is negligible. A trace property of a protocol \( C \) can be expressed as the asymptotic safety of the composition \( C \cdot A \) of the protocol with any p.p.t. adversary \( A \). These asymptotic notions apply only to modules that meet polynomial restrictions, so that all closed programs resulting from their composition always terminate in polynomial time. (See Küsters et al. [2012] for a detailed discussion of polynomial-time notions for code-based simulation-based security.)

### 3.1 Games, Ideal Functionalities, and Typed Interfaces.

We now explain how to use F7 typing to model cryptographic primitives and protocols, using authenticated encryption (AE) as a running example—see §4 and §6 for its TLS elaborations. Let \( C \) be a module that implements a cryptographic functionality or protocol. We may define security for \( C \) in three different styles: using games, ideal functionalities, or ideal interfaces. In this section, we assume that \( C \) is keyed, but our approach also applies to more complex, stateful functionalities. To begin with, we suppose that \( C \) manages a single key internally and does not allow for key compromise.

We define an interface \( I_C \) with two functions for encryption and decryption, for now assuming that plaintexts and ciphers are fixed-sized byte arrays. The key is kept implicit, so encryption...
takes a plaintext and returns a cipher; Conversely, decryption takes a cipher and returns a plaintext option, that is, either some plaintext or none, in case of decryption error.

\[
\text{type cipher } = \text{bytes} \\
\text{val ENC: p:plaintext } \rightarrow c:\text{cipher} \\
\text{val DEC: c:cipher } \rightarrow o:\text{plaintext option}
\]

**Games** Games provide oracle access to \(C\); this may be programmed as a module \(G\) with an interface \(I_G\) that exports oracle functions. Games come in two flavors: (1) Games with a winning condition, which can be expressed by the adversary breaking a safety assertion, (2) Left-or-right games, in which the adversary has to guess which of the two variants \(G_0\) or \(G_1\) of the game it is interacting with. In our framework these two variants are defined as follows:

**Definition 1** (1) \(C\) is \(G\)-game-secure if for all p.p.t. expressions \(A\) with no assume or assert such that \(I_G \vdash A : \text{unit}\), the expression \(C \cdot G \cdot A\) is asymptotically safe. (2) \(C\) is \((G_0, G_1)\)-game-secure if for all p.p.t. expressions \(A\) with \(I_G \vdash A : \text{bool}\), we have \(C \cdot G_0 \cdot A \approx_C C \cdot G_1 \cdot A\).

Typical games for modeling the authenticity and confidentiality of AE are \(\text{INT-CTX}\) and \(\text{IND-CPA}\). The former requires that the adversary forge a valid ciphertext; the latter requires that an adversary that freely chooses \((x_0, x_1)\) cannot distinguish between encryptions of \(x_0\) and encryptions of \(x_1\). In our formalism these game-based security properties are expressed as \(G_{\text{CTX}}\)-game-security and \((G_0, G_1)\)-game-security, using the following games coded in F# (\(b \in \{0, 1\}\)):

\[
\begin{align*}
\text{let log} & = \text{ref} [\text{}] \\
\text{let dec c } & = \text{match DEC c with} \\
& | \text{None } \rightarrow \text{None} \\
& | \text{Some(x)} \rightarrow \text{assert(List.mem c !log); Some(x)}
\end{align*}
\]

\[
G_{\text{CTX}} \triangleq \begin{cases} 
\text{let log } = \text{ref}[\text{}] & \text{let enc p } = \text{let c=} \text{ENC p in} \\
& \text{log } : = \text{c} :: !\text{log} ; c \\
\end{cases}
\]

\[
G_0 \triangleq \begin{cases} 
\text{let enc } x_0 x_1 = \text{ENC } x_0 \\
\end{cases}
\]

For \(G_{\text{CTX}}\) the interface \(I_G\) exports \(\text{enc}\) and \(\text{dec}\) (but not \(\text{log}\)). For \(G_0\) and \(G_1\) it exports only \(\text{enc}\). (See Fournet et al. [2011] for further examples of games coded in F#.)

**Ideal Functionalities with Simulators** An ideal functionality \(F\) for \(C\) implements the same interface \(I_C\) but provides nicer properties. \(F\) only needs to implement \(C\) partially; the rest of the implementation that is not security critical may be provided by a simulator \(S\), which is only required to exist.

**Definition 2** \(C\) is \(F\)-functionality-secure if there is a simulator \(S\) such that, for all p.p.t. expressions \(A\) with \(I_C \vdash A\), we have \(C \cdot A \approx_{\epsilon} S \cdot F \cdot A\).

In the spirit of conditional reactive simulatability [Backes et al., 2008], we may specify conditional emulation, by demanding that \(A\) be well typed with respect to an ideal interface \(I_C^I\) annotated with pre-conditions. This allows us, e.g., to give an ideal functionality for CPA secure encryption.

For primitives such as AE, we may design \(F\) so that \(\check{C}\) itself is a valid simulator, i.e. \(\check{C} \cdot A \approx_{\epsilon} C \cdot F \cdot A\). We refer to this as self-simulation.

**Definition 3** \(C\) is \(F\)-functionality-secure in self-simulation if for all p.p.t. expressions \(A\) with \(I_C \vdash A\) we have \(C \cdot A \approx_{\epsilon} C \cdot F \cdot A\).

Intuitively, emulating such a functionality corresponds to being secure with respect to a left-or-right game, in which the left game just does forwarding and the right game applies the filter \(F\).

Within our TLS implementation we define such left-right variants using a compile flag \#if ideal, as in the following example for authenticated encryption:
Definition 4 for all \( p.p.t. \) expressions \( A \)

\[
G_0, G_1 \triangleq \text{let } \log = \text{ref } [] \\
\text{let } \text{ENC}(p: \text{plain}) = \\
\quad \#\text{if } \text{ideal} \\
\quad \text{let } c = \text{ENC zero in} \\
\quad \log := (c:p) ::! \log \\
\quad \#\text{else} \\
\quad \text{let } c = \text{ENC } p \\
\quad \#\text{endif}
\]

\( G_0 \) is the code compiled with \( \text{ideal} \) unset and \( G_1 \) is the same code with \( \text{ideal} \) set.

Ideal Interfaces Instead of code, we may use types to express perfect security properties. For AE, for instance, the ideal interface below specifies ciphertext integrity (\( \text{INT-CTXT} \)):

\[
\text{val } \text{ENC}: p: \text{plain} \to c: \text{cipher} \{ \text{ENCrypted}(p,c) \} \\
\text{val } \text{DEC}: c: \text{cipher} \to \alpha(\text{plain option}) \\
\{ \langle p, o=\text{Some}(p) \Rightarrow \text{ENCrypted}(p,c) \rangle \}
\]

This interface is more precise than \( I_C^*: \text{ENC} \) now has a post-condition \( \text{ENCrypted}(p,c) \) stating that its result \( c \) is an encryption of its argument \( p \). (\( \text{ENC} \) may assume this as an event.) Hence, the postcondition of \( \text{DEC} \) states that decryption succeeds (that is, returns \( \text{Some } p \) for some plaintext \( p \)) only when applied to a cipher produced by \( \text{ENC} \).

A module is secure with respect to an ideal interface \( I_C^* \) when it asymptotically implements it, in the following sense:

**Definition 4** \( C \) is \( I_C^* \)-interface-secure if there exists a module \( C^i \) with \( \vdash C^i \rightsquigarrow I_C^* \) such that, for all \( p.p.t. \) expressions \( A \) with \( I_C^* \vdash A \), we have \( C \cdot A \approx_{\epsilon} C^i \cdot A \).

For instance, one may use an ideal functionality \( F \) such that \( F \rightsquigarrow I_C^* \). The advantage of type-based security is that one can then automatically continue the proof on code that uses \( I_C^* \).

As indicated above, there are obvious connections between games, ideal functionalities, and ideal interfaces, and under certain conditions one can prove these definitions equivalent. When it is clear from the context whether we talk about games, functionalities, or interfaces, we simply write \( G \)-secure, \( F \)-secure, and \( I \)-secure. \( I_C^* \)-security implies \( F \)-security if the typing properties of \( I_C^* \) is sufficient to guarantee that \( C^i \cdot A \approx_{\epsilon} C^i \cdot F \cdot A \).

**Proof:** \( I_C^* \)-security gives us \( C \cdot A \approx_{\epsilon} C^i \cdot A \). From the second premise we conclude that \( C^i \cdot A \approx_{\epsilon} C^i \cdot F \cdot A \). So \( C^i \) is a valid simulator. For a proof of self-simulation, one can use \( I_C^* \)-security a second time, now with \( A' = F \cdot A \) to conclude that \( C \cdot A \approx_{\epsilon} C \cdot F \cdot A \).

Secrecy using Ideal Interfaces To define confidentiality using types, we introduce concrete and ideal interfaces for the module that defines plaintexts for encryption:

**Definition 5** A plain interface \( I_{\text{Plain}} \) is of the form

\[
\text{type } \text{repr} = b: \text{bytes} \{ \text{Length}(b) = \text{plainsize} \} \\
\text{type } \text{plain} \\
\text{val } \text{repr}: \text{plain} \to \text{repr} \\
\text{val } \text{plain}: \text{repr} \to \text{plain}
\]

The type \( \text{repr} \) gives the representation of plaintexts, whereas the type \( \text{plain} \) is abstract, with functions \( \text{repr} \) and \( \text{plain} \) to convert between the two. (These may be implemented as the identity function.) The ideal plain interface \( I_{\text{plain}}^i \) is \( I_{\text{Plain}} \) without these two functions. Intuitively, removing them makes the interface parametric in type \( \text{plain} \), so that we can apply Theorem 2. Using ideal plain interfaces, we give an interface-based definition of secrecy.

**Definition 6** \( C \) is \( I_{\text{Plain}}^i \rightsquigarrow I_C^* \)-secure when there exists a module \( C^i \) with \( I_{\text{plain}}^i \vdash C^i \rightsquigarrow I_C^* \) such that, for all \( p.p.t. \) modules \( P \) with \( \vdash P \rightsquigarrow I_{\text{plain}}^i \), and \( A \) with \( I_{\text{plain}}^i \vdash A \), we have \( P \cdot C \cdot A \approx_{\epsilon} P \cdot C^i \cdot A \).
Parametricity guarantees both plaintext secrecy and integrity (but not ciphertext integrity). For example, a protocol using AE may define type \( \text{plain} = m : \text{repr}\{\text{Msg}(\text{m})\} \) where \( \text{Msg} \) is the protocol specification of an authentic plaintexts and then rely on typing to ensure authenticity of decrypted plaintexts.

**Equivalence of games, ideal functionalities, and ideal interfaces (illustrated for AE).** For authenticated encryption one can show that all three definitional styles—game based, functionality based and interface based—are equivalent. In particular, \( C \cdot F \) has the required typing properties. \( F \) corrects false-decryptions and encrypts zeros instead of plaintexts. This guarantees that it is parametric and meets its refinement typing.

Conversely these typing properties are sufficient to guarantee that \( C_i \cdot F \cdot A \approx C_i \cdot A \). Parametricity guarantees equivalence for the module in which \( C_i \) is handed zeros instead of plaintexts, and the refinement types statically ensure that \( F \) makes a correction if and only if the same correction is made by \( C_i \).

Moreover, \( F \)-functionality-security is equivalent to \( (G_0, G_1) \)-game-security where the left game only does forwarding and the right game applies the filter \( F \). This combined authenticated encryption game which simultaneously models authenticity and secrecy (see, e.g., Paterson et al. [2011]) can in turn be shown equivalent to being secure w.r.t. both the INT-CTXT and IND-CPA games described above.

### 3.2 Indexes for Multi-instance, Agility, and Corruption

**Multi-instance functionalities** Ideal functionalities and interfaces compose in the following intuitive sense: if the interfaces \( I_C \) and \( I_C' \) are disjoint, \( C \) is \( I_C \)-secure, and \( C' \) is \( I_C' \)-secure, then \( C \cdot C' \) is \( I_C \cdot I_C' \)-secure, and similarly with functionalities.

Rather than a fixed number of modules, we may use a module that support multiple, dynamic instances, via a code transformation that adds an `index` value (plus e.g. a key) to every call. (Software libraries are typically multi-instance.) For a keyed primitive, this module may generates a key at each call to some function \( \text{GEN}: \text{id}: \text{index} \rightarrow (\text{id})k \). The user provides the index, and type safety guarantees that materials with different indexes are not mixed. Instances may also differ, e.g. in their choice of plaintext lengths. For example, an ideal multi-instance interface for AE is:

```plaintext
type (\text{id}:\text{index})\text{key}
val \text{GEN}: \text{id}:\text{index} \rightarrow (\text{id})\text{key}
val \text{ENC}: \text{id}:\text{index} \rightarrow (\text{id})\text{key} \rightarrow \text{p:(\text{id})plain} \rightarrow 
cipher \{\text{ENCrypted}(\text{id},p,c)\}
val \text{DEC}: \text{id}:\text{index} \rightarrow (\text{id})\text{key} \rightarrow \text{cipher} \rightarrow 
o(\text{id})\text{plain option} \{ \forall p, o = \text{Some}(p) \Leftrightarrow \text{ENCrypted}(\text{id},p,c) \}
```

This interface is parameterized by a plain module that defines an indexed abstract type \( (\text{id}:\text{index})\text{plain} \), and uses an \( \text{ENCrypted} \) predicate with an extra index argument. Some multi-instance interfaces rely on usage restrictions that cannot be enforced by typing. We document these restrictions as side conditions. For instance, to achieve CTXT, we would usually require that users never generate two keys with the same index.

**Definition 7** A program \( A \) is a restricted user of \( I_C \) when \( I_C \vdash A \) and \( A \) calls \( \text{GEN} \) with pairwise distinct indexes.

As an important technicality, it is often sufficient to prove security against adversaries \( A \) that generate a single key.

To establish a self composition result for multi-instance cryptographic primitives we consider what it means for it to be secure with respect to a single session adversary which meets the following restriction.
Definition 8 A program $A$ is a single-instance user of $I^i_C$ if $I^i_C \vdash A$ and $A$ generates a single key.

The two restrictions on $A$ give rise to the following two definitions of security.

Definition 9 A multi-instance $C$ primitive is multi-instance (single-instance) $F$-secure, when, for all p.p.t. restricted (respectively single-instance) users $A$ with $I^i_C \vdash A$ there exists a simulator $S$ such that

$$C \cdot A \approx_S F \cdot A.$$  

Single-session security implies multi-instance security under some isolation conditions on $C$ and $F$. This follows from single instance security being a sufficient condition for universal composability [Canetti, 2001].

Theorem 3 (Multi-instance composition) If for each $i$, $C$ behaves like some isolated module $C_{id}$ and for the same $id$, $F$ behaves like some isolated module $F_{id}$ that calls $C_{id}$, then single-instance $F$-security implies multi-instance $F$-security.

Weak cryptographic algorithms Since indexed types keep different instances separated, we may as well use different algorithms, as long as they meet the same interface. For example, the index may include the name of the algorithm. Interestingly, this provides support for dealing with weak cryptographic algorithms, that is, algorithms that do not meet their specified security property. To this end, we introduce a predicate on indexes, $\text{Strong}(id)$, that holds when the algorithm is cryptographically secure, and we refine our ideal interface so that it offers security guarantees only at strong indexes.

For AE, we have two security properties, so we introduce predicates $\text{StrongAuth}$ for authenticity and $\text{Strong}$ for authenticated encryption. Hence, our postcondition of DEC now is $\{\text{StrongAuth}(id) \rightarrow (\forall p.o = \text{Some}(p) \Leftrightarrow \text{ENCrypted}(id,p,c))\}$. We also generalize our ideal plain interface, leaving the $\text{plain}$ and $\text{repr}$ functions available, but with preconditions that restrict their usage to weak algorithms:

$$\text{val plain: id:index}\{\ \text{not(StrongAuth(id))}\ \} \rightarrow \text{repr} \rightarrow (\text{id})\text{plain}$$

$$\text{val repr: id:index}\{\ \text{not(Strong(id))}\ \} \rightarrow (\text{id})\text{plain} \rightarrow \text{repr}$$

Intuitively, this enables AE to forge ciphertexts (or access plaintexts) at weak indexes, reflecting the fact that we do not have cryptographic security for their concrete algorithms. For programming ideal functionalities, we also introduce a specification function $\text{strong: id:index} \rightarrow b:bool\{b=\text{true} \Leftrightarrow \text{Strong(id)}\}$. For AE, for instance, the ideal functionality would encrypt zeros and decrypt by table lookups when $\text{strong id}$, and use the concrete algorithms otherwise. Of course, concrete implementations do not rely on this function.

Key compromise Cryptographic keys can be corrupted. As a further refinement of our interfaces, we consider two forms of key compromises: the leakage of honestly generated keys, and adversarially chosen keys. To this end we introduce a predicate on indexes, $\text{Corrupt(id)}$, that holds when keys are corrupted. To provide the adversary with the possibility to compromise keys we extend our indexed interfaces $I^i_C$ with functions

$$\text{val LEAK: id:index}\{\ \text{Corrupt(id)}\ \} \rightarrow (\text{id})\text{key} \rightarrow \text{bytes}$$

$$\text{val COERC: id:index}\{\ \text{Corrupt(id)}\ \} \rightarrow \text{bytes} \rightarrow (\text{id})\text{key}$$

and we adapt our ideal interfaces to provide security guarantees conditioned by the predicate $\text{not(Corrupt(id))}$, e.g., for AE, the postcondition of DEC becomes $\{\ \text{not(Corrupt(id))} \land \text{StrongAuth}(id) \rightarrow (\forall p.o = \text{Some}(p) \Leftrightarrow \text{ENCrypted}(id,p,c))\}$. We also introduce a specification function $\text{corrupt: id:index} \rightarrow b:bool\{b=\text{true} \Leftrightarrow \text{Corrupt(id)}\}$ for programming ideal functionalities.

As noted, e.g., by Backes and Pfitzmann [2004], Küsters and Tuengerthal [2009], an idealized module $C'$ that first encrypts a message and then leaks a key cannot be both indistinguishable from a real encryption scheme $C$ and parametric in the message. Given a ciphertext that is
independent of the message, efficient encryption schemes simply do not add enough ciphertext entropy to allow the simulation of adaptive corruptions. Schemes based on interaction, keys of the size of the message, and random oracles are the notable exception [Nielsen, 2002]. To avoid the commitment problem, we require \textit{Corrupt} and \textit{corrupt} to be monotonic, and fixed after the first encryption of a secret message.

In our TLS formal development, indexes are similar, but they keep track of more detailed information, for instance about the ciphersuite and certificates used in the handshake to generate the keys. In §4, we will use two main predicate on indexes, \textit{Safe} that guarantees both authenticity and secrecy for the transport layer, and \textit{Auth} that guarantees authenticity but not necessarily secrecy, logically defined as \textit{Auth}(id) = \textit{not}(\textit{Corrupt}(id)) ∧ \textit{StrongAuth}(id) and \textit{Safe}(id) = \textit{not}(\textit{Corrupt}(id)) ∧ \textit{Strong}(id). For simplicity, we do not model the independent corruption of connections after key establishment, so the \textit{Corrupt} predicate will be determined by the handshake, as the negation of its \textit{Honest} predicate on long-term keys.

**Lemmas established by typechecking** In the rest of the paper, we rely on numerous typing lemmas, established by running \texttt{F7} on the corresponding series of files—this task is automated by the main \texttt{Makefile} in our source distribution.

For each typing entry in this Makefile, with target of the form \texttt{<Module>\.tc7} when typechecking \texttt{<Module>}, with the \#ideal flag set, we refer to the resulting typing lemma by the name of the target. Taking the example of \textit{MAC}, a module that depends on the declarations of \textit{TLSInfo} and that implements message authentication, with an ideal typed interface \(I^\text{ideal}_{\text{MAC}}\) that expresses INT-CMA and a simpler typed interface \(I^{\text{unsafe}}_{\text{MAC}}\) that only enforces key abstraction, such a lemma may be explicitly stated as:

\begin{equation}
\text{Lemma 1 (MAC.tc7) } I_{\text{TLSInfo}} \vdash MAC_i \sim I^\text{ideal}_{\text{MAC}} \text{ and } I_{\text{TLSInfo}} \vdash MAC \sim I^{\text{unsafe}}_{\text{MAC}}.
\end{equation}

and is proved by successfully running \texttt{make MAC.tc7}. (The second, simpler typing judgment follows by considering the case where \textit{Safe}(id) is defined as \textit{false}).

As informally explained in §2.2, the concrete typing lemmas may be used to justify idealization steps (e.g. just checking that keys are abstract), while the ideal lemmas may be used to type larger idealized constructions. These typing lemmas can be systematically composed as we build larger functionalities; for instance, the lemma below may be obtained as corollaries of typing Lemmas \texttt{M1.tc7} till \texttt{Mn.tc7} with the appropriate chaining of interfaces.

\begin{equation}
\text{Lemma 2 (Ideal M}_i^1 \cdot M^i_n) I_{M_0} \vdash M^i_1 \cdot \ldots \cdot M^i_n \sim I^i_M.
\end{equation}

4 Authenticated Encryption for TLS Streams

We briefly describe the record layer, explain the new length-hiding features of our API, then outline our results for authenticated encryption in TLS.

**Fragment; Compress; MAC; Pad; then Encrypt** For each connection epoch, the transport layer runs two independent instances of stateful authenticated encryption (StAE) for communicating sequences of data fragments in both directions. The handshake creates these instances according to the suffix of the negotiated ciphersuite (after \texttt{WITH}), and provides them with adequate keying materials. In this section, we consider only the usual MAC-then-encrypt ciphersuites, parameterized by a symmetric encryption algorithm (3DES, AES, or RC4) and a MAC algorithm (e.g., HMAC with SHA1); our implementation also supports all authentication-only ciphersuites and has a placeholder for GCM encryption.

From protocol messages down to network packets, StAE proceeds as follows: (1) the message is split into fragments, each containing at most \(2^{14}\) bytes; (2) each fragment is compressed using the method negotiated during the handshake, if any; (3) each fragment is appended with a MAC over its content type, protocol version, sequence number, and contents; (4) when using a block cipher, each fragment is padded, as detailed below; (5) the resulting plaintext is encrypted;
Figure 4: Wire size of compressed-then-encrypted TLS fragments for 10 different mp3 files. X-axis is the fragment number for each file; Y-axis is the size, in bytes, of the compressed-then-encrypted fragment observed on the network. The 288 bytes first fragment of each file, fingerprinting the file type, is not plotted because it is out of range for this graph.

(6) the ciphertext is sent over TCP, with a header including the protocol version, content type, and length.

The details of fragmentation and padding are implementation dependent, but those details matter inasmuch as they affect cryptographic security and network traffic analysis.

4.1 Traffic Analysis and Length Hiding

Traffic Analysis and Fingerprinting

Even with perfect cryptography, traffic analysis yields much information about TLS applications [Dyer et al., 2012]. For example, compression may reveal redundancy in the plaintext when both plaintext and ciphertext lengths are known [Kelsey, 2002]; this suffices to break any IND-CPA based notion of secrecy. More surprisingly, TLS first fragments then compresses, hence sequences of ciphertext lengths may leak enough information to identify large messages being transferred. Thus, we implemented a new attack showing that an eavesdropper can uniquely identify JPG images and MP3 songs selected from a database, simply by observing short sub-sequences of ciphertext lengths. The attack is most effective against RC4 ciphersuites, but also succeeds against block ciphers with minimal padding.

We display in Fig. 4 the fingerprints of 10 MP3 files downloaded using Chrome from an OpenSSL server, a client-server configuration that negotiates compression by default. Although each file transfer involves more than 150 fragments, just 3 successive fragment lengths suffice to uniquely identify each song. Accordingly, our implementation disables compression, and our formal results apply only to connections where TLS-level compression is disabled.

Length Hiding.

TLS is not designed to prevent traffic analysis, but it does provide countermeasures when using a block cipher: padding before encryption hides the actual plaintext length and, by inserting extra padding beyond the minimal required to align to the next block boundary, one can hide a larger range of plaintext lengths. The padding may be any of the following 256 arrays \([0], [1], \ldots, [255], \ldots, [255]\) as long as the resulting plaintext is block-aligned. Most implementations use minimal padding; others, such as GnuTLS [Mavrogiannopoulos and Josef-
nnon, 2011], randomly select any of the correct paddings, but per-fragment padding schemes are often statistically ineffective Dyer et al. [2012].

A Length-Hiding TLS API Our API lets applications hide the length of their messages by indexing them with a range $m..n$ where $0 \leq m \leq n$. Intuitively, an observer of the encrypted connection may learn that the plaintext fits within its range, while its actual length remains secret.

Consider for example a website that relies on personalized cookies, containing between 100 and 500 bytes. The website may give cookies the indexed abstract type $(100,500)$data, hence requesting that their actual length be hidden. The range $(100,500)$ is treated as public, and suffices to determine fragmentation and padding. If the connection uses a block cipher, say AES_{128}_CBC_SHA, then any value of this type can be uniformly split, MACed, encoded, and encrypted into two fragments of 36 blocks each.

Our implementation follows a simple fragmentation and padding algorithm: given a range $m..n$, we compute first the minimal number of fragments needed to include up to $n - m$ bytes of additional padding, then the maximal length $p$ of the first fragment. Then, unless $p = n$, we use let $(f,rest)=DataStream.split \text{p} (m-1) (n-p) \text{text}$ then send the fragment $f$ and iterate on rest. The actual sizes of text, f, rest, and the padding added to obtain an encoded fragment of size $p$ remain provably secret (thanks to type abstraction) and do not influence the size of the fragment on the wire. Any implementation of the split function in DataStream must satisfy the following interface:

\[
\begin{align*}
\text{val concat: } m0:\text{nat} \rightarrow n0:\text{nat} \{ m0 <= n0 \} & \rightarrow \\
m1:\text{nat} \rightarrow n1:\text{nat} \{ m1 <= n1 \} & \rightarrow \\
b0: (m0,n0) \text{data} \rightarrow b1: (m1,n1) \text{data} & \rightarrow \\
b:(m0+m1, n0+n1) \text{data} \{ b0 @| b1 = b \}
\end{align*}
\]

where @ is byte array concatenation.

On the receiving end of a connection, the same length-hiding specification applies: as incoming TCP packets are processed, the application is notified of the arrival of “some” bytes, with a public range size that depends only on the ciphersuite and the size of those wire packets. Continuing with our example, the receiver would get two data chunks, each with a size range of $0..250$.

Applications built on top of this LH mechanism are responsible for specifying sensible ranges and thus control the shape of the network conversation. (They can still trivially leak the plaintext length, for instance by emitting a separate network event for every sent or received byte.) How to program secure applications on top of our API that do not leak privacy sensitive length-information is an interesting question outside the scope of this paper.

### 4.2 Authenticated Encryption Schemes

We present the two modules that implement multi-instance authenticated encryption for TLS fragments: first LHAEP, featuring indexes, ranges, and additional data (AD) to be authenticated with the plaintext; then StAE, implementing stateful encryption on top of LHAEP and organizing fragments into streams.

**Length-Hiding Authenticated Encryption (LHAEP)** We define $I^1_{\text{LHAEP}} \sim I^1_{\text{LHAEP}}$ security for the plaintext interface $I^1_{\text{LHAEP}}$ outlined below.

\[
\begin{align*}
type (\langle id:\text{index}, ad:\langle id\rangle data, r, range \rangle) \text{plain} & \\
type (\langle r, range \rangle) \text{bytes} = b:\text{bytes} \{ \text{fst}(r) \leq \text{Length}(b) \leq \text{snd}(r) \} & \\
\text{val plain: } id:\text{index} (\langle \text{notAuth(id)} \rangle) & \rightarrow \\
r, range \rightarrow ad:\langle id\rangle data \rightarrow (\langle r \rangle)\text{bytes} & \rightarrow (\langle id, ad, r \rangle) \text{plain} & \\
\text{val repr: } id:\text{index} (\langle \text{notSafe(id)} \rangle) & \rightarrow
\end{align*}
\]
Each plaintext is indexed by an instance \(id\), its additional data \(ad\), and its range \(r\). We use the refined type \((; id) data \rightarrow (; id, ad, r) plain \rightarrow (; r) bytes\)

We define the interface \(LHAEPlain\) parametrized by \(LHAE\); we omit its \textsc{Coerce} and \textsc{Leak} functions for brevity.

\[
\begin{align*}
\text{type} & \quad (; id, index) \rightarrow \text{key} \\
\text{val GEN}: id, index \rightarrow (; id, key) \\
\text{val ENC}: id, index \rightarrow k, (; id) key \rightarrow d, (; id) data \rightarrow r, range \rightarrow p, (; id, d, r) plain \rightarrow (; k', (; id) key \ast \text{c}\text{ipher}) \\
& \quad \{\text{CipherRange}(id, r, c) \ast \text{ENCRYPT}(id, d, p, c)\} \\
\text{val DEC}: id, index \rightarrow k, (; id) key \rightarrow d, (; id) data \rightarrow c, \text{cipher} \rightarrow o, (; k', (; id) key \ast r, range \{\text{CipherRange}(id, r, c)\} \ast \not\equiv \{p, (; id, d, r) plain\} \ast \text{option} \\
& \quad \{\text{Auth}(id) \Rightarrow k', r, p, (o = \text{Some}(k', r, p) \iff \text{ENCRYPT}(id, d, p, c))\}
\end{align*}
\]

The index \(id\) determines the algorithms to use. Keys for a particular index are created by calling \(GEN\); they encapsulate the full encryption state, typically an encryption key, a MAC key, and (when necessary) an IV or stream cipher state.

Encryption \(ENC\) takes a plaintext, executes the MAC-Encode-Encrypt sequence, and returns a cipher and (potentially) updated key. Decryption \(DEC\) takes a cipher, decrypts, decodes, and verifies the MAC; if every check succeeds, it returns a plaintext and updated key; otherwise it returns an error. Their logical specification is explained below.

\(\text{CipherRange}(id, r, c)\) is a predicate asserting that the length of ciphertext \(c\) reveals at most that the length of the plaintext is in the range \(r\). The secret length of the plaintext is authenticated, but its range at encryption is not: the range at decryption may be wider (unless \(id\) prescribes a stream cipher and all three lengths coincide).

\(\text{ENCRYPT}(id, d, p, c)\) is an abstract predicate specified as the postcondition of encryption, stating that \(c\) is an authenticated encryption of \(p\) with additional data \(d\). Its appearance also as a postcondition of decryption expresses \textit{ciphertext integrity}: only correctly-generated ciphertexts successfully decrypt.

Authenticity and confidentiality of plaintexts follow from parametricity for values of the \((; id, d, r, plain)\) type when the predicates \(\text{Auth}(id)\) and \(\text{Safe}(id)\) hold. For instance, when \(\text{Safe}(id)\) holds, the user (including the adversary) may learn the values of their indexes \(id, d, r\), but cannot call the \textsc{repr} function to read their content, nor call the \textsc{plain} function to forge their content.

Our implementation supports many protocol versions and ciphersuites, but provides security only for \textit{strong} indexes that use TLS 1.2 with secure ciphersuites, e.g. AES_CBC with fresh IVs. Our formal development mirrors a well known result of Krawczyk [2001, Theorem 2] that states that \textsc{IND-CPA} security of encryption and combined \textsc{INT-CTX} security of MAC-then-encrypt afford secure channels. Krawczyk also shows that stream ciphers as used in TLS provide combined \textsc{INT-CTX} security. We use the result of Paterson et al. [2011] to show that the block-cipher-based schemes implemented by our LHAE module are combined \textsc{INT-CTX} secure, despite the unauthenticated padding, for strong block ciphers and MAC algorithms.

Our concrete implementation of LHAE is a sequence of modules \( C \triangleq \text{MAC} \cdot \text{Encode} \cdot \text{ENC} \cdot \text{LHAE}\). Under the combined \textsc{INT-CTX} assumption, we prove by typing that \(C\) is \(LHAEPlain\)\textsuperscript{\textsc{secure}} for \textsc{IND-CPA} secure modules ENC and for restricted users (using LHAE keys linearly with pairwise-distinct additional data). This is expressed by the following lemma which leads up to our proof for stateful length-hiding authenticated encryption.

We let \(LHAE_F\) be the variant of LHAE that filters out ciphertexts that were not logged during encryption (see File \(LHAE,F,fs\) with flag \#ideal,F and its typing in Lemma \(LHAE,F,tc7\)); and let \(LHAE_F\) be the variant that always decrypts using table lookup and fails otherwise (File \(LHAE_F\)
with flag #ideal).

**Lemma 3 (Length-hiding AE)** Let $I_{LHAEP\text{plain}}$ and $I^{\prime}_{LHAEP\text{plain}}$ be the concrete and ideal plain interfaces for LHAE, respectively. Let $I_{LHAE}$ be the ideal LHAE interface. Let $C = MAC \cdot Encode \cdot ENC \cdot LHAE$. If $P \cdot C \cdot A \approx_{c} P \cdot MAC \cdot Encode \cdot ENC \cdot LHAE^{F} \cdot A$ for restricted users, MAC is $I_{MAC}$-secure, and ENC is $I^{\prime}_{Encode} \approx_{a} I^{\prime}_{ENC}$-secure, then $C$ is $I_{LHAEP\text{plain}} \approx \neg I_{LHAE}$-secure for restricted users.

In the lemma, the equation $P \cdot C \cdot A \approx_{c} P \cdot MAC \cdot Encode \cdot ENC \cdot LHAE^{F} \cdot A$ captures the combined INT-CTXT assumption on LHAE proved by Krawczyk [2001] for stream ciphers and by Paterson et al. [2011] for length-hiding CBC-mode encryption.

Internally, we also decompose MAC into a core MAC module and individual non-agile modules for each strong MAC algorithm, e.g., MAC_SHA1 and MAC_SHA256. This allows to prove that MAC is $I_{MAC}$-secure based on classical INT-CMA assumptions on these modules.

**Proof Outline** The proof proceeds as follows, each step mostly relying on automated typechecking.

\[
P \cdot C \cdot A \triangleq P \cdot MAC \cdot Encode \cdot ENC \cdot LHAE \cdot A
\]

(1)

\[
\approx_{c} P \cdot MAC \cdot Encode \cdot ENC \cdot LHAE^{F} \cdot A
\]

(2)

\[
\approx P \cdot MAC \cdot Encode \cdot ENC \cdot LHAE \cdot A
\]

(3)

\[
\approx_{c} P \cdot MAC \cdot Encode \cdot ENC^{i} \cdot LHAE \cdot A
\]

(4)

\[
\approx P \cdot MAC \cdot Encode^{i} \cdot ENC^{i} \cdot LHAE \cdot A
\]

(5)

\[
\approx P \cdot MAC^{i} \cdot Encode^{i} \cdot ENC^{i} \cdot LHAE \cdot A
\]

(6)

where $P$ ranges over all p.p.t. modules with $\vdash P \approx I_{LHAEP\text{plain}}, \vdash P \approx I_{LHAEP\text{plain}}$, and $A$ ranges over all p.p.t. modules with $\vdash I_{LHAEP\text{plain}}, \vdash I_{LHAE} \vdash A$.

Step (1) is by definition. Step (2) applies the joint INT-CTXT computational assumption on $C$; it requires no specific typing, since the keys of LHAE are treated abstractly (at safe indexes) in its exported interface. Step (3) relies on the (typed) functional correctness of ENC and Encode, which implies that logged ciphertexts always decrypt to their original plaintext (Lemma LHAE,$F$,tc7). Step (4) applies IND-CPA security for ENC (see ENC,$F$s with flag # ideal and the ideal interface ENC,$F$s7). It demands that the keys of ENC be treated abstractly and that only logged ciphertexts are ever decrypted—this is enforced by typechecking LHAE$^{F}$ (which does not decrypt anymore at safe indexes) against ENC,$F$s7. Step (5) is by parametricity (Theorem 2) for the types defined by Encode, after typechecking that ENC$^{i}$ · LHAE$^{F}$ · A never accesses their representations at safe indexes. Encode$^{i}$ is idealized to only encode strings of zeros. Step (6) applies INT-CMA security for MAC (see MAC,$F$s with flag #ideal and the ideal interface MAC,$F$s7). We obtain that $C$ is $I_{LHAEP\text{plain}} \approx I_{LHAE}$-secure by showing that the type combined $C^{i} = MAC^{i} \cdot Encode^{i} \cdot ENC^{i} \cdot LHAE^{F}$ is such that $I_{LHAEP\text{plain}} \vdash C^{i} \approx \neg I_{LHAE}$, which follows from the automated typechecking of MAC$^{i}$, Encode$^{i}$, ENC$^{i}$, and LHAE$^{F}$ (lemmas MAC$^{i}$,tc7, Encode$^{i}$,tc7, ENC$^{i}$,tc7, and LHAE,tc7).

**Stateful Length-hiding Authenticated Encryption (StAE)** Programmed and verified on top of LHAE, StAE authenticates the position of each plaintext within a stream of messages. To this end, its ideal plaintext interface $I^{\prime}_{StPlain}$ introduces a fourth index: a log that records the sequence of preceding plaintexts and additional data. Hence, in a sequence of stateful plaintexts, the first is indexed by the empty log, the second by a log containing the first plaintext, and so on.

**type** (id:index, l:id) log, ad:(id) data, r:range stplain

We omit its plain and repr declarations similar to those of $I^{\prime}_{LHAE}$. The ideal interface $I^{\prime}_{StAE}$ for StAE is as follows:
val GEN: id:index →
  w:(id) writer {Log(w) = []} + r:(id) reader {Log(r) = []}
val ENC: id:index → wr:(id) writer → d:(id) data → r:range →
  p:(id,Log(wr),d,r) stplain → c:cipher + wr’:(id) writer
  {Log(wr’r) = (d,p):Log(wr) ∧ ENCrypted(id,wr’d,p,c)}
  ∧ CipherRange(id,r,c))
val DEC: id:index → rd:(id) reader → d:(id) data → c: cipher →
  o:(range \{CipherRange(id,r,c)\}) + p:(id,Log(rd),d,r) stplain *
  rd’:(id) reader{Log(rd') = (d,p):Log(rd)} option
  {Auth(id) = (3rd’,r,p, o = Some(rd’,r,p) ⇔
    (∃wr. ENCrypted(id,wr,d,p) ∧ Log(wr) = Log(rd))}

It uses the same Safe and Auth predicates as LHAE.

Keys and sequence numbers for STAE are encapsulated into linear writer and reader capabilities that hold the local state of the encryption and (for specification purposes only) the log of messages written or read so far. Encryption adds a log entry into the writer, containing the plaintext and its additional data. If a sequence of plaintexts was encrypted using STAE, then decryption guarantees that the returned plaintexts arrive in the right order (unless not(Auth(id))), since each plaintext must be indexed by the preceding log.

In particular, we define an embedding of STAE plaintexts into LHAE plaintexts.

val toLHAEPlain: id:index → l:(id) log → ad:(id) data →
  r:range → (id,lad,r) stplain → (id,MakeAD(id,Length(l),ad),r) plain
val fromLHAEPlain: id:index → l:(id) log → ad:(id) data →
  r:range → (id,MakeAD(id,Length(l),ad),r) plain → (id,lad,r) stplain

In TLS, the additional data for STAE contains the protocol version and content type; to implement STAE on top of LHAE, makeAD (and its specification function MakeAD) adds an 8-byte prefix representing the sequence number to form the additional data for LHAE. To program STAE using LHAE, we first write an LHAEPlain module that implements $I_{LHAEPlain}$ using $I_{StPlain}$. Then, for instance, STAE.ENC simply adds a sequence number then invokes LHAE.ENC.

(* StLHAE *)

let ENC id wr data rg plain =
  let text = toLHAEPlain id wr.log data plain
  let seqn = List.length wr.log
  let data’ = makeAD id seqn data
  let key’.cipher = LHAE.ENC id wr.key data’ rg text
  let log’ = addToLog id wr.log data rg text
  (⟨key=key’,log=log’⟩,cipher)

let DEC id rd data cipher =
  let seqn = List.length rd.log
  let data’ = makeAD id seqn data
  match LHAE.DEC id r.key data’ cipher with
  | Correct(key’,rg,text) →
    let plain = fromLHAEPlain id rd.log data rg text
    let log’ = addToLog id rd.log data rg text
    Correct(⟨key=key’,log=log’⟩, rg, plain)

| Error(x,y) → Error(x,y)

This code casts $I_{StPlain}^*$ to $I_{LHAEPlain}^*$ computes the sequence number as the current length of the log, creates a new additional data, calls LHAE.ENC, updates the key and log and returns. Decryption performs these actions in the reverse order, raising an Error if LHAE decryption fails. By typing (Lemma STAE.to7), we show that our STAE code meets its ideal interface, assuming restricted users (using readers and writers linearly) and given that LHAE meets its ideal interface.
Theorem 4 (Stateful AE) Let $I^i_{LHAEPlain}$ and $I^i_{LHAE}$ be the ideal plain interface and ideal interface of LHAE. Let $I^i_{StPlain}$ and $I^i_{StAE}$ be the ideal plain interface and ideal interface of StAE. Let $C = \text{MAC} \cdot \text{Encode} \cdot \text{ENC} \cdot \text{LHAE}$ and $S = \text{LHAEPlain} \cdot C \cdot \text{StAE}$.

If $C$ is $I^i_{LHAEPlain}$-secure for restricted users, then $S$ is $I^i_{StPlain}$-secure for restricted users.

Proof: The proof is mostly by typing; we still need to check that $StAE$ satisfies the unsafe restriction of $LHAE$ by inspecting its code. The proof proceeds as follows:

\[ P \cdot S \cdot A \triangleq P \cdot \text{LHAEPlain} \cdot C \cdot \text{StAE} \cdot A \]

(7)

\[ \approx_{e} P \cdot \text{LHAEPlain} \cdot C^i \cdot \text{StAE} \cdot A \]

(8)

where $P$ ranges over all p.p.t. modules with $\vdash P \leadsto I^i_{StPlain}$, $\vdash P \leadsto I^i_{StPlain}$, and $A$ ranges over p.p.t. modules such that $I^i_{StPlain}, I^i_{StAE} \vdash A$. We conclude that $S$ is $I^i_{StPlain}$-secure by typing the combined $S^i = \text{LHAEPlain} \cdot C^i \cdot \text{StAE}$ as $I^i_{StPlain} \vdash S^i \leadsto I^i_{StAE}$, which follows from the typing assumption on $C^i$ and the automated typechecking lemmas $\text{LHAEPlain}.tc7$ and $\text{StAE}.tc7$.

4.3 Related Work on Authenticated Encryption

Provable security cast doubt at the security of early authenticated encryptions [An and Bellare, 2001, Bellare and Namprempre, 2008] and proposed provably-secure modes [Rogaway et al., 2003, McGrew and Viega, 2004, Kohno et al., 2004, Bellare et al., 2004b]. Others looked specifically at the authenticated encryption techniques used by SSH [Bellare et al., 2004a, 2002], or IPSEC [Degabriele and Paterson, 2010]. Krawczyk [2001], Paterson et al. [2011] look at the MAC then encode and encrypt (MEE) authenticated encryption mode used by TLS. The latter [Paterson et al., 2011] considers the case of length-hiding encryption. Like Jager et al. [2012] we build on the work of Paterson et al. [2011] but, in addition, we establish the security of the stateful encryption of TLS based on length-hiding AEAD. Besides, our result applies to an implementation, not just a model.

Authenticated encryption has been studied in the context of simulation-based security [Küsters and Tuengerthal, 2009]. Küsters and Tuengerthal [2011b] provide a large cryptographic library amenable to the verification of realistic protocols. Maurer and Tackmann [2010] look at MEE in the model of constructive cryptography.

Paterson et al. [2011] provide the cryptographic result closest to the TLS record protocol; in particular they also explicitly address the relative lengths, alignments, and encoding of the plaintexts and MACs. They report an attack when using short MACs, which we independently discovered as part of this project. They give a concrete bound on MAC, Encode, then Encrypt (MEE) using CBC. On the other hand, their model does not consider the integration of LHAE within TLS.

Hence, we obtain security for TLS Record streams, under the cryptographic assumptions discussed for LHAE.

5 The Handshake Protocol

This section discusses the ‘control’ part of our TLS API for managing sessions and connections. Our implementation delegates these tasks to a component that entirely hides the Handshake protocol from the rest of our code. We verify it against a typed interface $I^i_{HS}$ that specifies key-establishment, and we independently verify the rest of TLS for any key-establishment functionality that implements $I^i_{HS}$. We discuss the main features of the Handshake, but we refer to
the online materials for its 750-line F7 specification and the details of the underlying cryptographic assumptions in its auxiliary modules (see the typed interfaces Sig.fs7, RSA.fs7, DH.fs7, CRE.fs7, PRF.fs7 and their implementations with flag #ideal).

**Ciphersuites** The Handshake protocol depends on both the TLS version and the prefix of the ciphersuite (before WITH). It has two main mechanisms for establishing a shared pre-master secret (PMS): (1) the client samples a fresh value and encrypts it using the server public key; or (2) the client and server exchange Diffie-Hellman exponentials $g^x$, $g^y$ and use their private exponents x and y to compute the value $g^{xy}$. The Diffie-Hellman exponentials can be either static, meaning that for authenticated ciphersuites they have to be included in certificates, or ephemeral, meaning that they need to be signed with certified keys.

**Data Structures** We give below the public datatypes of the API that expose information about sessions and epochs to the application. (These types are defined in TLSInfo.fs.) Our main integrity goal for the handshake is that clients and servers agree on their content.

```plaintext
type SessionInfo = {
    init_clrand: random;
    init_sr rand: random
    version: version;
    cipherSuite: cipherSuite;
    compression: compression;
    pms_data: bytes;
    clientID: cert list;
    serverID: cert list;
    sessionID: sessionID
}

type Role = Client | Server

type ConnectionInfo = {
    role: Role; id_clrand: random;
    id_sr: epoch;
    id_out: epoch
}

type epoch =
    Init of Role
    Next of random * random
        * SessionInfo
        * epoch
```

SessionInfo records information for a given session: the initial client and server random values (used in the full handshake that generated the session); the protocol version, ciphersuite, and compression algorithm; the exchanged data for the PMS; the certificates used for authenticating each role, if any; and the session identifier (used for resumption). ConnectionInfo holds the current epochs, for reading and writing, the local role, and the local random value, to guarantee that ConnectionInfos are pairwise distinct. (The role field can be computed as a function from the writing epoch, and is duplicated in ConnectionInfo for ease of access.) Each epoch is unidirectional and initially records just the role of the writer (Client or Server); for each complete handshake, it also records the SessionInfo and client and server randoms used for key derivation.

### 5.1 The Handshake Interface

The handshake interface is divided into three parts: the long-term key interface and the control interface for modeling the creation of honest and corrupted keys and initiating and controlling runs of the handshake respectively are both outlined in Fig. 5; the network interface for driving the progress of the handshake and for receiving keys and notifications is outlined in Fig. 6.

**Long-term Key Interface** The handshake makes use of long-term keys, which may be either honestly generated and used, or compromised. The certification of long-term keys is outside the TLS standard, but is crucial for modeling its security. For this reason, we implement basic certificate management in the Cert module, but we leave the interpretation of certificates to the TLS application. From the protocol viewpoint, we only require a function (certkey) to extract public keys from exchanged certificate chains, and a predicate (Honest) to specify which of the long-term keys used by TLS are honest.

**Control Interface** We now outline the handshake interface. There is one instance of the Handshake protocol at each TCP connection, each able to perform a sequence of handshakes for that connection. At each end of the connection, the local state has an abstract type (:ci)state indexed by the current connectionInfo ci. We require that connection states be treated linearly: each call to the interface takes the current state and returns the next state.
(*) Long—term key Interface *)

predicate val Honest : pk → bool
val create : template → (pk:pk {Honest(pk)}) option
val coerce : template → bytes → pk option
function val CertKey : certs → pk option
val certkey : c:certs → o:pk option {o = CertKey(c)}

(* Control Interface *)

predicate Authorize of Role ∗ SessionInfo
type ConnectionInfo = CI
private type (ci:CI) state
function val Config : ci:CI ∗ s:(ci)state → config
val init : r:Role → c:config → (ci:CI ∗ s:(ci)state){Config(ci,s) = c ...}
val resume : nextSID:sessionID → c:config → (ci:CI ∗ s:(ci)state){ Config(ci,s) = c ...}
val rehandshake : ci:CI → s:(ci)state → c:config → (b:bool ∗ s’:(ci)state){ ...}
val rekey : ci:CI → s:(ci)state → c:config → b:bool ∗ s’:(ci)state{ ...}
val request : ci:CI → s:(ci)state → c:config → b:bool ∗ s’:(ci)state{ ...}
val authorize : r:Role → si:SessionInfo → unit {Authorize(r,si)}

Figure 5: Ideal Handshake interface (Key and Control interface excerpt).

The interface first provides functions to create new instances of the protocol, as client or server, possibly resuming existing sessions, and to initiate re-handshakes on established connections:

* init creates a client instance (with a fresh session) or a server instance (possibly resuming an existing session, at the client’s initiative);

* resume creates a client instance from some existing session. For all of these functions, an event Config(ci,c) records the configuration chosen by the user.

* With request the server asks the client to start a renegotiation;

* rehandshake or rekey let the client start a renegotiation, using a full or abbreviated handshake (with the same ciphersuite).

Network Interface Once configured and started, the handshake progresses by sending and receiving fragments of content types Handshake and CCS. Calls to next fragment may yield an outgoing fragment to be sent using the current record, if any; conversely, calls to recv fragment and recv CCS process incoming fragments. In response to these calls, the handshake updates its internal state and notifies progress gradually, first by delivering the new index and cryptographic materials, independently for each direction (using event SentCCS(id) for each epoch) then, after both (1) accepting the correct Finished message from its peer and (2) sending its own Finished message, by confirming that the handshake is complete (using predicate Complete(ci,cfg) for the full ConnectionInfo) and thus that the new keys can be used to send and receive application data. In TLS, whether (1) or (2) above happens first depends both on the role and whether we are resuming a prior session or not. (To support accelerated handshakes, one may even decide to start sending data immediately after (2), and before key confirmation; This is the gist of the false start extension proposed by Langley and Moeller [2010].)

The Complete predicate in the postcondition of connection establishment states that the incoming and outgoing epochs in the new ConnectionInfo are synchronized, and relates their common SessionInfo (written si for SI(ci.id.out) below) to the local and remote configurations.

2This corresponds to accept and connect in the main TLS API in Fig. 13 in §6.
predicate val Complete: CI → config → bool
predicate EvSentFinishedFirst of CI = bool
predicate val SentCCS: epoch → bool

\[
def \text{type} (\pi:CI, hs:(ci) state) \text{ outgoing } = \\
(\text{OutIdle of } s':(ci) state) \mid \\
(\text{OutSome of } (rg:\text{range} \ast f:(ci.id\text{,out},rg)\text{Fragment} \ast s':(ci)state)) \mid \\
(\text{OutCCS of } (rg:\text{range} \ast f:(ci.id\text{,out},rg)\text{Fragment} \ast \text{cs}\!:(CI) \ast cs'(ci)\text{id\text{,out}}\text{StatefulLHAE}\!:(state) \ast s':(ci)state) \mid \text{ci.write } = \text{Pred}(ci'.write) \land \\
\text{ci.read } = ci'.read \ldots ) \\
(\text{OutFinished of } (rg:\text{range} \ast f:(ci.id\text{,out},rg)\text{Fragment} \ast s':(ci)state) \mid \text{EvSentFinishedFirst}(ci, \text{true})) \\
(\text{OutComplete of } (rg:\text{range} \ast f:(ci.id\text{,out},rg)\text{Fragment} \ast s':(ci)state) \mid \text{Complete}(ci,\text{Config}(ci hs)))
\]

val next_fragment: ci:CI → s:(ci)state → (\text{x,c})\text{outgoing}

\[
def \text{type} (\pi:CI, c,config)\text{ incoming } = \\
(\text{InAck of } (ci,c)\text{nextState}) \mid \\
(\text{InVersionAgreed of } (ci,c)\text{nextState } \ast \text{ProtocolVersion}) \mid \\
(\text{InQuery of } \text{Cert.certchain } \ast \text{advice:bool } \ast \text{(ci)state}) \mid \\
(\text{InFinished of } (\text{ci})\text{state } \{\text{EvSentFinishedFirst}(ci, \text{false})\}) \mid \\
(\text{InComplete of } (\text{ci})\text{state } \{\text{Complete}(ci,c)\}) \mid \\
(\text{InError of } \text{alertDescription } \ast \text{string } \ast \text{(ci)state})
\]

val recv_fragment: ci:CI → s:(ci)state → (\text{rg:range} \rightarrow (\text{ci.id\text{,in},rg})\text{Fragment} \ast \text{fragment} \rightarrow (ci,\text{Config}(ci,s))\text{incoming}

\[
def \text{type} (\pi:CI, c,config)\text{ incomingCCS } = \\
(\text{InCCSack of } \text{c}\!:(CI) \ast (\text{ci}\!:\text{id\text{,in}})\text{StatefulLHAE}\!:(state) \ast (ci')\text{state } \{\text{ci.write } = ci'.write \land \text{ci.read } = \text{Pred}(ci'.read)\}) \mid \\
(\text{InCCSError of } \text{alertDescription } \ast \text{string } \ast (ci,c)\text{nextState})
\]

val recv_ccs: ci:CI → s:(ci)state → (\text{rg:range} \rightarrow (\text{ci.id\text{,in},rg})\text{Fragment} \ast \text{fragment} \rightarrow (ci,\text{Config}(ci,s))\text{incomingCCS}

\]

Figure 6: Ideal Handshake interface (Network interface excerpt).

Provided that (1) both the ciphersuite and all its algorithms in \( si \) are strong (predicate \text{StrongHS} (si), explained shortly); and (2) the long-term keys recorded in \( si \) are honest (predicate \text{Honest}), then we have that (a) the negotiated content of the session \( si \) is compatible with the two initial configurations; (b) the peer sent a CCS with a matching epoch (event \text{SentCCS}(ci.id.in)); and (c) the handshake was actually secure (predicate \text{SafeHS}(si)), thereby enabling secure transport.

By definition, for connections with an anonymous client, the server obtains no such guarantees, but the connection may still provide server authentication, and then be used to run application-level client authentication—see §6.4.

Modularity with Finished Messages (Discussion) Most modern security definitions for key establishment require that the resulting key be indistinguishable from a fresh random key. In contrast, TLS uses the new epoch before Handshake completion, to encrypt and decrypt the Finished messages, and thus does not meet this requirement. To address this issue, Datta et al. [2006] introduce a weaker notion of key usability for a given cryptographic task. The main drawback of key usability is that it breaks modularity and must be re-established for each task. Interestingly, our type discipline already restricts the usage of keys, so we entirely avoid the Finished message controversy, and achieve both modularity and TLS compliance. We guarantee indistinguishability from fresh keys only as the Handshake passes the keys to the Record layer. These keys can be used at once to process the Finished messages, and later (after completion) to secure application data.
5.2 Handshake Security and Modular Verification

We define security for the ideal handshake interface $I_{\text{HS}}$, outlined in Fig. 5, and parameterized by $I_{\text{SAE}}$, the ideal interface for StAE in §4 that defines the type of keys established by the handshake. As in §4, we demand that the users of the handshake interface use its state linearly; this is easily checked by inspection of the code of Dispatch.fs.

**Definition 10** A module HS is a secure handshake when it is $I_{\text{SAE}} \leadsto I_{\text{HS}}$-secure for restricted users.

The StAE keys have abstract types, so the module HS in the definition can obtain them only by calling GEN and COERCE, and it can turn bytes into key materials using the latter only for epochs $id$ such that $\text{not(Auth(id))}$, the pre-condition of COERCE. Thus, Definition 10 entails that, whenever Auth (and a fortiori Safe) holds, a secure handshake establishes ideal, fresh random key materials (as created by GEN).

More precisely, $I_{\text{HS}}$ uses a predicate SafeHS on SessionInfo to indicate the secure runs of the handshake, such that $\text{Auth(id)}$ implies $\text{SafeHS(SI(id))}$. To type the handshake, we let

$$\text{SafeHS(si)} = \text{StrongHS(si)} \land \text{HonestPMS(si)}$$

where HonestPMS(si) means that the pre master secret was securely generated between compliant endpoints using honest long-term keys, and where StrongHS(si) collects our cryptographic assumptions on the algorithms selected by the protocol version and ciphersuite indicated in the SessionInfo si. For the handshake, these algorithms are provided by the modules Sig implementing all signatures used by TLS, RSA and DH implementing the two sub-protocols for exchanging the PMS, CRE a computational randomness extractor for deriving master secrets, and PRF implementing pseudo-random functions for deriving keys and authenticating finish messages.

We obtain the security of the pre-master secret exchange by making strong cryptographic assumptions (RSA-PMS) and (DH-PMS) on the combined modules CRE-RSA and CRE-DE. These assumptions are similar to the tagged key-encapsulation security of Jonsson and B. S. Kaliski [2002] and the PRF-ODH assumption of Jager et al. [2012] respectively (see §5.3 for details). Thus we define

$$\text{StrongHS(si)} = \text{StrongSig(si)} \land \text{StrongCRE(si)} \land \text{StrongPRF(si)} \land (\text{StrongRSAPMS(si)} \lor \text{StrongDHPMS(si)})$$

For example, if the ciphersuite of si matches TLS_DHE_DSS_WITH_*+, StrongHS(si) holds if the signature scheme DSS is INT-CMA secure [Goldwasser et al., 1988], CRE and DH are jointly DH-PMS secure, CRE is a computationally strong randomness extractor [Fouque et al., 2008], and PRF is a pseudo-random function, and similarly for RSA-based ciphersuites. The theorem below states the security of our handshake implementation relative to the strength of the algorithms it uses.

**Theorem 5 (Handshake)** If $\text{Nonce}$ is $I_{\text{Nonce}}$-secure, $\text{Sig}$ is $I_{\text{Sig}}$-secure, $\text{CRE}$ is $I_{\text{CRE}}$-secure, $\text{PRF}$ is $I_{\text{PRF}}$-secure, and we have

- (RSA-PMS) $\text{RSAKey} \cdot \text{CRE} \cdot \text{RSA} \approx \text{RSAKey} \cdot \text{CRE} \cdot \text{RSA}'$,  
- (DH-PMS) $\text{DHGroup} \cdot \text{CRE} \cdot \text{DH} \approx \text{DHGroup} \cdot \text{CRE} \cdot \text{DH}'$,  

then $\text{HS} = \text{Nonce} \cdot \text{Sig} \cdot \text{RSAKey} \cdot \text{Cert} \cdot \text{PRF} \cdot \text{DHGroup} \cdot \text{CRE} \cdot \text{RSA} \cdot \text{DH} \cdot \text{TLSExt} \cdot \text{Handshake}$ is $I_{\text{SAE}} \leadsto I_{\text{HS}}$-secure.

Intuitively, the theorem states that HS is secure provided its cryptographic building blocks are INT-CMA, CRE, PRF, RSA-PMS, and DH-PMS secure for all strong handshake ciphersuites.
Proof outline To be able to complete the proof of Theorem 5 by typing, we replace each concrete implementations of the underlying cryptographic modules by their typed, ideal counterparts. The order of idealizations in our proof corresponds to the sequence of games in ordinary security proofs. For example, consider the ciphersuites TLS_DHE_DSS_WITH_* analyzed by Jager et al. [2012]. Compare the numbers in the modular structure of handshake in Fig.1 with the sequence of games in the proof of their Theorem 1. Each game corresponds to the idealization of one module in our architecture. Game 1 corresponds to idealizing Nonce to avoid collisions; (Games 2 and 4 capture losses in the reduction due to the lack of multi-instance secure cryptographic primitives.) Game 3 to idealizing Sig to guarantee that the attacker cannot replace ephemeral Diffie-Hellman exponentials\(^1\); Game 5 to idealizing DH to seed CRE with randomness unknown to the attacker and to idealizing CRE to output truly random master secrets; Game 6 and Game 7 correspond to idealizing PRF to generate both random keys and ideal authentication of all handshake messages. Note that PRF can be a standard pseudo-random function, whereas module CRE needs to be a computational randomness extractor [Fouque et al., 2008] as it is seeded with an exponential. The proof of Jager et al. [2012] only considers a particular ciphersuite in isolation and only for the initial handshake; the proof for our implementation requires more work to handle full and abbreviated handshakes and re-handshakes with different key exchange methods, and thus heavily relies on automation, e.g., because of the potential for cross-protocol attacks [Mavrogiannopoulos et al., 2012].

Let \( S \) abbreviate our implementation of StAE from § 4 and let \( A \) ranges over all p.p.t. adversaries that meet the interface of StAE and HS. Following the game sequence above, we have the following equations, where each step is justified by a cryptographic assumption (or a reordering of independent modules) and typechecking.

\[
\begin{align*}
S\text{-Nonce} \cdot Sig \cdot RSAKey \cdot Cert \cdot PRF \cdot DHGroup \cdot CRE \cdot RSA \cdot DH \cdot TLSExt \cdot Handshake \cdot A & \quad (9) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot RSAKey \cdot Cert \cdot PRF \cdot DHGroup \cdot CRE \cdot RSA \cdot DH \cdot TLSExt \cdot Handshake \cdot A & \quad (10) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot RSAKey \cdot Cert \cdot PRF \cdot DHGroup \cdot CRE \cdot RSA \cdot DH \cdot TLSExt \cdot Handshake \cdot A & \quad (11) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot DHGroup \cdot RSAKey \cdot CRE \cdot RSA \cdot DH \cdot Cert \cdot TLSExt \cdot Handshake \cdot A & \quad (12) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot DHGroup \cdot RSAKey \cdot CRE \cdot RSA' \cdot DH \cdot Cert \cdot TLSExt \cdot Handshake \cdot A & \quad (13) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot RSAKey \cdot Cert \cdot DHGroup \cdot CRE \cdot DH \cdot RSA' \cdot TLSExt \cdot Handshake \cdot A & \quad (14) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot RSAKey \cdot Cert \cdot DHGroup \cdot CRE \cdot DH \cdot RSA' \cdot TLSExt \cdot Handshake \cdot A & \quad (15) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot RSAKey \cdot Cert \cdot DHGroup \cdot CRE \cdot DH \cdot RSA' \cdot TLSExt \cdot Handshake \cdot A & \quad (16) \\
\approx, \quad S\text{-Nonce} \cdot Sig \cdot PRF \cdot RSAKey \cdot Cert \cdot DHGroup \cdot CRE \cdot DH \cdot RSA' \cdot TLSExt \cdot Handshake \cdot A & \quad (17)
\end{align*}
\]

After idealization, we apply typing Lemmas to verify by typing that the idealized handshake meets \( I_{HS} \). The Handshake module itself, the largest and most complex in our codebase, implements the handshake internal state machine, but does not implement cryptography. It is verified by typing using the ideal interfaces of the cryptographic modules (Lemma Handshake.tc7). For this task, we carefully specify the content of the message log eventually verified in the Finished messages, and we rely on the safe renegotiation extension to provide authentication of the whole chain of epochs extended by each successive handshake on the connection.

5.3 Key distribution for TLS master secrets

Next, we investigate the RSA-PMS and DH-PMS assumptions in more detail. Consider the key distribution mechanisms invoked by the TLS handshake for establishing TLS master secrets. As part of the key distribution, the master secret is generated by a computational randomness extractor (implemented in our CRE module) from pre-master secrets that are either communicated using RSA encryption or established by a Diffie-Hellman protocol. The corresponding modules
are RSAKey, RSA for RSA and DHGroup, DH for Ephemeral Diffie Hellman. For simplicity, we do not currently support static Diffie Hellman ciphersuites, which are not used much in practice.

The master-secret distribution component of the handshake composes these modules together with the CRE module. In this section we refer to this subfunctionality of Handshake as the master secret module MS. The cryptographic context $A$ (often called environment in simulation-based security) of MS consists of the remaining handshake, the rest of TLS and, as usual, potential applications and their adversaries. This module composition is of the form

$$\text{RSAKey} \cdot \text{DHGroup} \cdot \text{CRE} \cdot \text{RSA} \cdot \text{DH} \cdot \text{MS} \cdot A$$

with the dependency graph depicted in Fig. 7.

The module MS relies on the following declarations exported by the interface of RSAKey $\cdot$ DHGroup $\cdot$ CRE $\cdot$ RSA $\cdot$ DH:

$$( \ast \text{CRE} \ast)$$

open DHGroup
open TLSInfo
type rsapms
type dhpms
val genRSA: pk:RSAKeys.pk $\rightarrow$ pv:ProtocolVersion $\rightarrow$ (pk,pv) rsapms
val coerceRSA: pk:RSAKeys.pk $\rightarrow$ pv:ProtocolVersion $\rightarrow$ bytes $\rightarrow$ (pk,pv)rsapms
val genDH: p:p $\rightarrow$ g:(p)g $\rightarrow$ gx:(p)elt $\rightarrow$ gy:(p)elt $\rightarrow$ (p,g,gx,gy) dhpms
val extractRSA: SessionInfo $\rightarrow$ ProtocolVersion $\rightarrow$ rsapms $\rightarrow$ masterSecret
val extractDH: SessionInfo $\rightarrow$ dhpms $\rightarrow$ masterSecret

$$( \ast \text{RSA} \ast)$$

val encrypt: RSAKeys.pk $\rightarrow$ ProtocolVersion $\rightarrow$ CRE.rsapms $\rightarrow$ bytes
val decrypt: RSAKeys.sk $\rightarrow$ ProtocolVersion $\rightarrow$ bool $\rightarrow$ bytes $\rightarrow$ CRE.rsapms

$$( \ast \text{DH} \ast)$$

type pp = p*g

type spp = pp:pp { StrongPP(pp) }

type (p,p,g,g,elt) secret
val genPP : unit $\rightarrow$ spp
val defaultPP : unit $\rightarrow$ spp
val genKey: pp $\rightarrow$ elt $\ast$ secret
val exp: p:p $\rightarrow$ g:(p)g $\rightarrow$ gx:(p)elt $\rightarrow$ gy:(p)elt $\rightarrow$ x:(p,g,gx) secret $\rightarrow$ (p,g,gx,gy) CRE.dhpms

The types RSAKey.sk, DHE.rsapms, DHE.dhpms, and DH.secret are abstract. The type RSAKeys.pk is bytes, but if not(Honest(pk)), then encrypt and decrypt operate only on corrupted rsapms. Similarly, the type pp is public, but if not(StrongPP(pp)), genKey and genDH produce
corrupted secret and $dhpms$ values. The $SessionInfo$ values passed to $extractRSA$ and $extractDH$ are public, but required to contain unique client and server randoms for each pre-master secret.

We do not describe the $MS$ implementation in detail, but instead consider a large class of possible $MS$ implementations that respect these constraints. The ultimate security property of master key distribution is the establishment of master secrets that are indistinguishable from random values, even given the adversaries possibility to query all functions in the interface subject to the parametricity and uniqueness constraints described above. As an intermediate security goal, i.e., as a proof step, we show that $rsapms$ and $dhpms$ are treated abstractly by an idealized version of these modules.

The **RSA-PMS assumption** For RSA key transport to provide security directly requires chosen ciphertext attack secure encryption or key encapsulation. This is however wishful thinking, as PKCS#1 v1.5 simply is not secure against chosen ciphertext attacks. Instead, we rely on a joint cryptographic assumption about the combined modules $RSA \cdot CRE$ stating that our real $RSA$ module which implements the Bleichenbacher countermeasures can be replaced with an ideal $RSA$ module that encrypts a fake $rsapms$ and recovers the real $rsapms$ during decryption by idealized table lookup. Please consult our source code for this idealized version. To state the assumption more concisely, we define this assumption without typing requirement using a cryptographic game. The RSA-PMS game implies the above idealization up to a loss in the reduction incurred because it considers only a single honest RSA public key, i.e., for p.p.t. $A$ and p.p.t. $A'$, it holds that

$$\forall A. RSAKey \cdot CRE \cdot RSA \cdot A \quad \forall A'. RSAKey \cdot CRE \cdot RSA \cdot RSAGame_0 \cdot A'$$

$$\approx_c RSAKey \cdot CRE \cdot RSA^i \cdot A \quad \iff \quad \approx_c RSAKey \cdot CRE \cdot RSA \cdot RSAGame_1 \cdot A'. $$

We say that the RSA-PMS assumption holds if $RSAKey \cdot CRE \cdot RSA$ is $(RSAGame_0, RSAGame_1)$-secure. We give the code for $RSAGame_0$ and $RSAGame_1$ in Fig. 5.3.

Crucially, $decrypt$ returns a random $pms$ in case of decryption error to prevent the Bleichenbacher [1998] attack. The adversary’s goal is to guess the side $b$ of the game he is interacting with. When $b = 0$, we have a single key version of the concrete TLS modules. When $b = 1$, the ciphertext is independent of the honestly generated PMSs.

Following Jonsson and B. S. Kaliski [2002] we can reduce RSA-PMS to the one-wayness of PKCS#1 v1.5 under existence of a plaintext checking oracle [Okamoto and Pointcheval, 2001].

Figure 8: The code for $RSAGame_0$ and $RSAGame_1$; the latter is obtained by setting the ideal flag. The interface $I_{RSAgame}$ grants the adversary access to the value $pk$ as well as the two oracle functions $genRSA\_encrypt\_extract$ and $decrypt\_extract$. 

```plaintext
let sk, pk = RSAKeys\_keyGen()
let genRSA\_encrypt\_extract si pv =
  let CrSr = si.\_init\_rand, si.\_init\_rand
  if mem CrSr \!\langle log, CrSr\_enc \rangle \text{ then None }
  else
    log, CrSr\_enc := CrSr::!log, CrSr\_enc
    let pms = CRE,genRSA pk pv
    #if ideal
    let fake = CRE, genRSA pk pv
    log := (fake,pms)::!log
    let c = encrypt pk pv fake
    #else
    let c = encrypt pk pv pms
    #endif
    let ms=extractRSA si pv pms
    Some (c,ms)

let decrypt\_extract si pv c=
  let CrSr = si.\_init\_rand, si.\_init\_rand
  if mem CrSr \!\langle log, CrSr\_dec \rangle \text{ then None }
  else
    log, CrSr\_dec := (CrSr)::!log, CrSr\_dec
    let pms = decrypt sk pv true c
    #if ideal
    let pms =
      match assoc el \!log with
      | Some(ideal_pms) \rightarrow ideal_pms
      | None \rightarrow pms
    #endif
    Some (extractRSA si pv pms)
```

30
This, however, requires the random oracle model. More precisely, extractRSA needs to be modeled as a random oracle, and RSAGame is extended with such an oracle to idealize master secret extraction. The one-wayness of PKCS#1 v1.5 under existence of a plaintext checking oracle can in turn be further related to the one-wayness of RSA given a partial RSA decision oracle [Jonsson and B. S. Kaliski, 2002]. Next, we outline this reduction proof in our setting.

One-wayness states that for $x = \text{random } 46$, $sk, pk = \text{keyGen}()$, and $c = \text{encrypt}_{PKCS1v15} pk pv@x$ an adversary given $pk, c$, and access to let $pco$ plain cipher $= (\text{decrypt } sk \text{ cipher } = \text{plain})$ cannot efficiently output $x$ (with high enough probability in short enough time). The specific form of the challenge plaintext, $pv@x$, models a TLS detail, namely that the minimum supported TLS version is appended in front of the $pms$ value.

PROVING THE RSA-PMS ASSUMPTION FOR TLS Given a successful RSAGame attacker $A$, we build a PKCS#1 v1.5 one-wayness attacker $B$. The reduction $B$ uses an auxiliary list $L$ that is originally empty and will store failed or delayed decryption attempts. $B$ is given a value $c$ for $pv$ for which it should find the plaintext $pms$. When $A$ queries genRSA_encrypt_extract $si pv$, $B$ picks a random value $ms$ and returns $(c, ms)$.

We first describe how $B$ simulates calls to extractRSA which, as this proof is in the random oracle model, are translated into calls to the random oracle $H$.

$B$ responds to extractRSA $si pv pms$ queries as follows. If the query is old, the output is already defined in $H$. Otherwise $B$ queries the $pco$ on $pms.c$. If $pco$ outputs true, $B$ wins by outputting $pms$ and is done. Next, $B$ queries $pco$ on $pms.c^*$ for all $(pv, c^*, Cr, Sr) \rightarrow ms$ in $L$. If $pco$ outputs true, $B$ will use this $ms$ as the result of extractRSA. Otherwise $B$ generates a random value for a fresh $ms$. Finally $B$ programs $H$ with $(pv, pms, Cr, Sr) \rightarrow ms$ to return $ms$.

$B$ answers a decrypt_extract $si pv c$ query by first sending $pms, c$ to $pco$ for every $pms$ string such that $(pv, pms, Cr, Sr) \rightarrow ms$ is in $H$. If $pco$ returns true for some $pms$ then $B$ returns the corresponding $ms$. If the $pco$ oracle returns false, it generates a random value $ms$ and adds $(pv, c, Cr, Sr) \rightarrow ms$ in $L$. It sends $ms$ as the response to the query.

Correct simulation. We check that all oracle functions provided to $A$ return the expected distributions:

- The simulation of genRSA_encrypt_extract $si pv$ returns a PKCS#1 v1.5 ciphertext encrypting a random $pms$ and a random value $ms$.
- The simulation of extractRSA $si pv pms$ returns random values, except when queried on the same values before, or when the value was returned by decrypt_extract $si' pv c$ where $si'.init.rand = si.init.rand$ and $si'.init.rand = si.init.rand$. In the latter case, the simulation guarantees that $pms$ is the decryption of the ciphertext $c$ provided to decrypt_extract.
- The simulation of decrypt_extract $si' pv c$ returns $ms$ when queried on the challenge ciphertext and challenge $si pv$ such that $si'.init.rand = si.init.rand$ and $si'.init.rand = si.init.rand$. Otherwise it either returns a value $ms$ such that either $(pv, pms, Cr, Sr) \rightarrow ms$ in $H$ where $pms$ is a decryption of $c$. Or a random value $ms$ which will be returned in calls to extractRSA $si pv pms$, if $pms$ is a decryption of $c$ and $si'.init.rand = si.init.rand$ and $si'.init.rand = si.init.rand$.

In the implementation a random $ms$ could also result from a random $pms$ being returned by decrypt on decryption failure. This value is however unpredictable to $A$ and does not allow him to test for any inconsistencies in the simulation. Note that crucially here we rely on the fact that decrypt_extract $si pv c$ can only be queried once per $(Cr Sr)$ pair.

Probability analysis. We now compare the success probability of $A$ to that of $B$. We assume that $A$ runs in less than $t$ time, makes at most $q_{sd}$ and $q_{br}$ decrypt_extract and extractRSA queries respectively, and has success probability at least $\epsilon$, i.e. $Adv_A = |Pr[RSAGame_0 \cdot A \downarrow 1] - Pr[RSAGame_1 \cdot A \downarrow 1]| = \epsilon$. 

31
The games differ only on whether the \textit{pms} values in the calls to \textit{encrypt} and \textit{extractRSA} in function \textit{genRSA\_encrypt\_extract} are the same or different:

\begin{verbatim}
# if ideal
let fake = CRE\_genRSA pk pv
log := (fake, pms)::!log
let c = encrypt pk pv fake
# else
let c = encrypt pk pv pms
# endif
extractRSA si pv pms
\end{verbatim}

Stated otherwise, unless \textit{A} calls \textit{extractRSA} on values \((v, pms, Cr, Sr)\) which we denote by the event \(E\), he has no way of distinguishing real from ideal and thus

\[
\Pr[RSAGame_0 \cdot A \downarrow 1|\neg E] - \Pr[RSAGame_1 \cdot A \downarrow 1|\neg E] = 0
\]

We split the success probability of \(A\) by \(E\), and get \(\epsilon \leq 0 \cdot \Pr[\neg E] + 1 \cdot \Pr[E]\), and so \(A\) queries \textit{extractRSA} for \textit{pms} within \(q_R\) queries at least with probability \(\epsilon\). Comparing the running time of \(A\) and \(B\) we observe that \(B\) makes at most \(3 \cdot q_R\) queries to \textit{pco} and does \(O(q_R + q_D)\) elementary table lookups. This would contradict the one-wayness of RSA in presence of a plaintext checking oracle and thus provides evidence against the feasibility of the RSA-PMS adversary \(A\).

This proof relies on random oracles, and would require to switch our whole analysis of TLS into a relativized complexity theoretic world in which random oracles exist. The RSA-PMS assumption however is independent of random oracles. This can be seen by the following proof in the standard model which assumes that future TLS standards will employ proper chosen ciphertext secure encryption schemes. A disconnect from reality very different from random oracles.

\textbf{Proof outline of RSA-PMS assuming CCA2 encryption} \quad If we, hypothetically, assume that \textit{encrypt} and \textit{decrypt} use a chosen ciphertext secure encryption scheme instead of a malleable one, the proof can be substantially simplified.

As the first step we idealize encryption, i.e. we encrypt a zero string, and decrypt by table lookup using the ciphertext. Now the ciphertext \(c\) returned by \textit{genRSA\_encrypt\_extract} does not depend on the bit \(b\) anymore. Thus, the only chance \(A\) has in distinguishing \(RSAGame_0\) from \(RSAGame_1\) is by guessing \(fake\) which can only happen with probability \(q_D/2^{30.8}\).

To get closer to a proof by typing we can define the plain interface of the CCA encryption scheme to be \(CRE\_rsapms\), which we can then idealize with some probability loss to:

\begin{verbatim}
val genRSA: pk:RSAKeys.pk \rightarrow pv:ProtocolVersion \rightarrow (pk,pv) rsapms\{Gen(pk,pms)\}
val coerceRSA: pk:RSAKeys.pk \rightarrow pv:ProtocolVersion \rightarrow bytes \rightarrow (pk,pv)rsapms\{'pms'\_id'|id\'".
\end{verbatim}

The postcondition of \textit{coerceRSA} which is used by \textit{decrypt} when successfully decrypting ciphertexts generated by the adversary, guarantees that the adversary is unable to guess \textit{rsapms} values generated by \textit{genRSA}. This in turn guarantees that the lookup in \textit{decrypt\_extract} only succeeds for the challenge ciphertext, which makes the real and the ideal version of \textit{RSAGame} perfectly indistinguishable. Note that the loss in the reduction is covered by the idealization of \textit{genRSA} and \textit{coerceRSA} in \textit{CRE}. We do not yet idealize \textit{extractRSA} as this relies on good pre-master secrets and thus relies on the RSA-PMS assumption.

\textbf{The DHE-PMS assumption} \quad For Diffie-Hellman, it may seem that pre-master secret security relies simply on the DDH assumption, which states that \(g^{xy}\) is indistinguishable from a random \(g^x\). As noted by Jager et al. [2012] this is however not the case. TLS clients output keys for \(g^{xy}\)
while the attacker is still able to impersonate the client to query the server on a $g^x$ of his choice, for which the server will then compute $g^x$. Given such a DH oracle the DDH assumption simply does not hold anymore. We can only restore security by considering the larger key exchange protocol in which the attacker can only learn $extractDH si g^x$, and $extractDH si g^y$.

Our joint cryptographic assumption about the combined modules $DH \cdot CRE$ states that the real $DH$ module can be replaced with an ideal $DH$ module that for pairs of honest $gx$ and $gy$ values replaces $g^x$ by $g^*$. Please consult our source code for this idealized version.

To state the assumption concisely, we define it without typing requirements using a cryptographic game. The RSA-PMS game implies the above idealization up to a loss in the reduction incurred because it only considers a single honest server DH value $gy$, i.e., for p.p.t. $A$ and p.p.t. $A'$, it holds that

$$\forall A. \ DHGroup \cdot CRE \cdot DH \cdot A \approx^e DHGroup \cdot CRE \cdot DH \cdot DHEGame_0 \cdot A'$$

$$\forall A'. \ DHGroup \cdot CRE \cdot DH' \cdot A \iff \approx^e DHGroup \cdot CRE \cdot DH \cdot DHEGame_1 \cdot A'$$

The $DH$-PMS assumption holds if $DHGroup \cdot CRE \cdot DH$ is $(DHEGame_0, DHEGame_1)$-secure. The code for module $DHEGame_1$ is depicted in Fig. 9.

The adversary’s goal is to guess the side $b$. When $b = 0$, we have a single server $gy$ version of the concrete TLS modules. When $b = 1$, then $gx$ and $gy$ are independent of the honestly generated PMSs. This game does not model all constraints on $Cr$ and $Sr$ enforced by TLS and thus pessimistically gives the adversary more power than he has. This facilitates composition and may afford some extra protection in case nonces are abused, as discussed by Rogaway and Shrimpton [2006] for authenticated encryption, but requires a stronger assumption.

To allow comparison with Jager et al. [2012] we also consider a more restricted game $DHEGame'$ shown in Fig. 10, which is sufficient for our analysis if, like them, we consider only ciphersuites with client authentication. In this case we can declare all epochs with anonymous ciphersuites to be unsafe.

Under the restriction that $genKey, si$ is called only once and is called before $exp_extract$, we show that our game-based assumption is equivalent to the $PRF\cdotODH$ assumption of Jager et al. [2012] conditioned on $extractDH$ being a good computational randomness extractor. In our formalism, $PRF\cdotODH$ is defined by the game in Fig. 11.

The function $wrap m'$ converts the representation of $(Cr, Sr)$ in the PRF-ODH assumption into its corresponding $SessionInfo$. This allows reuse of our TLS code in the defini-
let \( pp = \text{genPP}() \)

let \( \text{private} \ gy, \text{secret} = \text{genKey} \ pp \)

(* gy: fixed, honest server keys *)

let \( \text{private} \ logs \ gx \)

(* a log of honest gx \( \rightarrow \) Cr, Sr *)

let \( \text{private} \ log \)

(* a log of honest (gx,gy) \( \rightarrow \) honest PMS *)

let \( \text{genKey} \ si \ si = \)

let \( \text{CrSr} = \text{si}.\text{ini} \text{.crand}, \text{si}.\text{ini} \text{.srand} \)

let \( \text{gx,gy} = \text{genKey} \ pp \)

log.gx:= (gx,CrSr)::!log.gx

gx, gy

(* gx, gy is released after fixing (Cr, Sr) *)

let \( \text{exp} \text{.extract} \ si \ gx \)

let \( \text{CrSr} = \text{si}.\text{ini} \text{.crand}, \text{si}.\text{ini} \text{.srand} \)

let \( \text{pms} = \)

match assoc gx |!log.gx with
| Some(CrSr') when CrSr=CrSr' \rightarrow
  #if ideal
  let pms =
    match assoc (gx,gy) |!log with
    | None \rightarrow
      let pms=CRE sampleDH p g gx gy
      log := ((gx,gy),pms)::!log; pms
    | Some pms \rightarrow pms
    #else
    let pms = exp p g gx gy secret
    #endif
    Some (extractDH si pms)
  #else
  let pms = Some (extractDH si (exp p g gx gy secret))
  (* we only return the resulting ms *)

Figure 10: The code for \( \text{DHGame}'_0 \) and \( \text{DHGame}'_1 \); the latter is obtained by setting the \( \text{ideal} \) compile flag. The interface \( I_{\text{DHGame}'} \) gives the adversary access to \( pp, \text{genKey}_si \) and \( \text{exp} \text{.extract} \).

let \( pp = \text{genPP}() \)

let \( \text{private} \ log.gx = \text{ref} [] \)

let \( \text{challenge} \ m = \)

let \( \text{gx,gy} = \text{genKey} \ pp \)

let \( \text{gy, secret} = \text{genKey} \ pp \)

(* gx: fixed, honest server keys *)

#if ideal
let \( \text{ms}=\text{sampleMS} \ (\text{wrap} \ m) \)
#else
let \( \text{pms} = \text{exp} \ p \ g \ gx \ gy \text{ secret} \)
let \( \text{ms} = \text{extractDH} \ (\text{wrap} \ m') \ pms \)
#endif

gx,gy,ms

(* we only return the resulting ms *)

Figure 11: The code for \( \text{PRF-ODH}'_0 \) and \( \text{PRF-ODH}'_1 \); the latter is obtained by setting the \( \text{ideal} \) compile flag. The interface \( I_{\text{PRF-ODH}} \) gives the adversary access to \( pp, \text{challenge} \) and \( \text{queryDHO} \).

34
The proof uses the fact that, if $extractDH$ is a good computational randomness extractor, then $extractDH$ is indistinguishable from the uniformly distributed master secret produced by $sampleMS$.

**Proof Outline of PRF-ODH Assuming $(DHGame_0', DHGame_1')$-Security and CRE.**

A successful adversary $A$ against PRF-ODH can be used to build an algorithm $B_1$ that wins the $DHGame'$ game or an algorithm $B_2$ that wins the CRE game.

- **Game 1.** The same as the PRF-ODH$_0$ game.

- **Game 2.** The same as Game 1 except that instead of returning $ms = extractDH \text{(wrap } m) g^{x^y}$ it returns $ms = extractDH \text{(wrap } m) g^z$ for a randomly sampled $z$.

- **Game 3.** The same as Game 2 except that instead of returning $ms = extractDH \text{(wrap } m) g^z$ for a randomly sampled $z$, it returns $ms = sampleMS$.

**Lemma 4 (Game 1-2)** The distinguishing probability of $A$ in Game 2 is bounded by the success probability of adversary $B_1$ against $DHGame'$.

If we have a successful attacker $A$ distinguishing Game 1 and Game 2 we build a successful adversary $B_1$ against $DHGame'$.

$B_1$ receives $p$ and simulates the environment of $A$ which outputs $m$. $B_1$ uses $wrap$ to build some $si$. It then queries $genKey, si$ to learn $gy$ and $exp, extract, si, gy$ to complete it’s challenge $gx, gy, ms$ for $A$. If $b = 0$, these values are distributed like $extractDH \text{ si } g^{x^y}$, otherwise like $extractDH \text{ si } g^z$. $B_1$ forwards the guess of $A$ to break PRF-ODH.

**Lemma 5 (Game 2-3)** Any algorithm that distinguishes between Game 2 and Game 3 can be used to build adversary $B_2$ that wins the CRE game.

This follows from the definition of computational randomness extraction.

**Proof Outline of $(DHGame_0', DHGame_1')$-Security Assuming PRF-ODH and CRE.**

A successful adversary $A$ against $DHGame'$ can be used to build an algorithm $B_1$ that breaks the PRF-ODH game or an algorithm $B_2$ that wins the CRE game.

- **Game 1.** The same as the $DHGame_0'$ game.

- **Game 2.** The same as Game 1, except that instead of returning $ms = extractDH \text{ si } g^{x^y}$ it returns $ms = sampleMS$.

- **Game 3.** The same as Game 2, except that instead of returning $ms = sampleMS$ it returns $ms = extractDH \text{ si } g^z$ for a randomly sampled $z$.

This game is the same as $DHGame_1'$.

**Lemma 6 (Game 1-2)** The distinguishing probability between Game 1 and Game 2 is bounded by the success probability of $B_1$ against PRF-ODH.

If we have a successful attacker $A$ distinguishing Game 1 and Game 2 we build a successful adversary $B_1$ against PRF-ODH. $B_1$ receives $p$ and simulates the environment of $A$ which first calls $genKey, si$. $B_1$ provides $si, init srand@si, init srand$ as $m$ to it’s challenger to learn $g^z$, $x^y$, and $ms$. It returns the first two values as $gx$ and $gy$ to $A$. When $A$ queries for $exp, extract$, $B_1$ uses $ms$ when queries on $gx$ and a call to the ODH oracle for all other values. $B_1$ forwards the guess of $A$ to break PRF-ODH.

**Lemma 7 (Game 2-3)** The distinguishing probability between Game 2 and Game 3 is bounded by the success probability of $B_2$ against a CRE game.

This follows from the definition of computational randomness extraction.
5.4 Related Work on Key Exchange

Cryptographic research on secure key exchange usually follows either a game-based approach or a simulation-based approach, as pioneered by Bellare and Rogaway [1993] and Canetti and Krawczyk [2002]. Indeed, Gajek et al. [2008] outline an ambitious proof of TLS in the simulation-based model of Canetti [2001]. However, Küsters and Tuengerthal [2011a] point out that their use of the UC joint state theorem to obtain multi-session security relies on pre-established identifiers not available in TLS, and suggest how to overcome this limitation. We share with simulation-based definitions that we rely on indistinguishability to model both authenticity and secrecy.

Morrissey et al. [2008] analyze a variant of the TLS handshake protocol. Fouque et al. [2008] study the key extraction function of TLS. Jager et al. [2012] perform a game-based security analysis of TLS relying on Paterson et al. [2011].

6 Main API & Theorems for TLS

We are now ready to explain our ideal interface for TLS and give our main theorems.

6.1 TLS API

The main API depends on two predicates on epochs, logically derived from those defined in §4 and §5:

- **Auth(id)**, defined as SafeHS(SI(id)) \(\wedge\) StrongAuth(id), indicates that data exchanged over a connection with epoch id is expected to be authentic in an ideal TLS implementation. Our types prevent the forgery of such data.

- **Safe(id)**, defined as SafeHS(SI(id)) \(\wedge\) Strong(id), indicates that data exchanged over id is expected to be both authentic and secret in an ideal implementation. Our types prevent all access to such data outside the application.

Both these predicates rely on the honesty of the pre master secret, and hence of the long-term keys used in id. For simplicity, our API does not enable the compromise of StAE keys once they have been safely generated by the handshake. However, since these keys are also typed using interfaces with LEAK functions (see §4), it would be straightforward to formally supplement our APIs with explicit functions that let the adversary generate corrupt keys. Similarly, we do not currently model forward secrecy, which can in any case only be achieved for ephemeral Diffie-Hellman ciphersuites.

**DataStream** The API is parameterized by an application-level plaintext module DataStream. Fig. 12 provides its main interface towards TLS. (It may export a richer interface to other application-level modules.) The indexed abstract type data represents messages exchanged over
TLS connections; stream is the type of specification-level sequences of data fragments, used to index the messages sent (or received) at a particular position in the data stream. DataStream may define data concretely e.g. as bytes, and stream as a list of bytes.

To send the next message over an established connection indexed by id, after sending the stream s, the application may provide any value of type (id,s,rg)data. As explained in §4, data is also indexed by a range rg, so that the application may shape the traffic by hiding secret data lengths within a given public range. Both data and stream are abstract types indexed precisely by positions and epochs, thus only the application may access raw data or move data between positions and epochs. The DataStream interface exports three functions to TLS. The functions data and repr let TLS read the concrete binary representation of application data at un-Safe indexes, and forge application data at un-Auth indexes. In addition, the split function enables TLS to fragment data without looking at its contents, by providing two sub-ranges that add up to the index range r; the function returns two data values that logically come one after the other in their data stream. The application may disallow data from being split at certain ranges, to prevent small fragments, for example.

Main TLS Interface Fig. 13 outlines our main F7 interface, omitting most refinements for simplicity. The API provides abstract TLS connections using two main types: indexes (ConnectionInfo, written CI for brevity) and states (Cn). An index is an immutable data structure detailing connection parameters (see §5). A state is an abstract type, representing a handle c to a running client or server TLS connection; its index is written CI(c). Initial states (Cn0) are returned by connect or accept; they must then be used linearly; next states that leave the index unchanged are written nextCn. The interface provides two main functions to operate on TLS connections, read and write, plus a series of functions to initiate them and control their successive handshakes (explained in §5).

- read takes the current state and returns an iostreamz, with different cases: Read(c,d) returns an updated state c and some received data d; the index of d states that it extends the input stream of the current epoch, and a postcondition states that if Auth holds for this epoch, then the peer has sent that data; similarly Fatal and Close, report genuine alerts from the peer if Auth holds; CertQuery notifies the application that the current handshake requests some certificate authorization (either by resuming the handshake with authorize or aborting it with refuse); Handshaken signals the completion of the current handshake; the application can then inspect the new epoch before proceeding.

- write takes the current state and some data, and similarly returns an iostreamz with different cases, e.g., WritePartial returns an updated state and the rest of the message, after sending its first fragment; and MustRead notifies the application that it should read until the ongoing handshake completes before writing again.

For instance, a client application that implements data as strings may interact with TLS with a (call → result) sequence as follows (with an implicit state threaded through the calls):

```plaintext
connect t g; read → CertQuery(q); authorize q → Handshaken;
write 6..30 "Hello world\n" → WriteComplete;
read → Read(0..24,"404\n"); read → Close(t).
```

A sample matching server trace may be

```plaintext
accept t' g' read → Handshaken;
read → Read(0..128,"Hello World\n");
write 4..4 "404\n" → WriteComplete;
shutdown; read → Close(t').
```

TLS does not guarantee synchronization between input and output streams; for instance, the client may write three messages d0, d1, d2 then read d0, then initiate rekeying, while the server reads d0, write d'0 and d'1, then reads d1. On the other hand, when notified of a Close or that
type \((x:\text{Cl})\) \text{query}

\begin{align*}
\text{type} \; \text{Cn} \\
\text{type} \; (g:\text{config}) \; \text{Cn}0 = c0 : \text{Cn} \{\text{InitCn}(g,c0)\} \\
\text{type} \; (x:\text{Cn}) \; \text{nextCn} = c' : \text{Cn} \{\text{NextCn}(c,c')\} \\
\text{type} \; (x:\text{Cn}) \; \text{msg}, i = r : \text{range} \times (x:\text{Cl}.\text{id}_{\text{in}}, \text{Stream}_{\text{f}}(c), r) \; \text{data} \\
\text{type} \; (x:\text{Cn}) \; \text{msg}, o = r : \text{range} \times (x:\text{Cl}.\text{id}_{\text{out}}, \text{Stream}_{\text{o}}(c), r) \; \text{data} \\
\text{type} \; (x:\text{Cn}) \; \text{ioresult}, i = \\
\text{Read of} \; c' : (x) \; \text{nextCn} + d : (x) \; \text{msg}, i \\
\{ \text{Extend}_{d}(c,c',d) \land (\text{Auth}(c,c') \Rightarrow \text{Write}(c,c',\text{Bytes}_{j}(c'))) \} \\
\text{Close of TCP.Stream} \{\text{Auth}(c,c') \Rightarrow \text{Close}(c,c',\text{Bytes}_{j}(c'))\} \\
\text{Fatal of} \; \text{alertDescription} \\
\{ \text{Auth}(c,c') \Rightarrow \text{Fatal}(c,c',a,\text{Bytes}_{j}(c'))\} \\
\text{CertQuery of} \; c' : (x) \; \text{nextCn} + (x') \; \text{query} \{\text{Extend}(c, c')\} \\
\text{Handshaken of} \; c' : \text{Cn} \{\text{Complete}(c,c'),\text{Cfg}(c')\} \land \ldots \\
\ldots \\
\text{val} \; \text{read} : c : \text{Cn} \rightarrow (x) \; \text{ioresult}, i \\
\end{align*}

\begin{align*}
\text{type} \; \text{Cn}, d : (x) \; \text{msg}, o) \; \text{ioresult}, o = \\
\text{WriteComplete of} \; c' : (x) \; \text{nextCn} \{\text{Extend}_{o}(c,c',d)\} \\
\text{WritePartial of} \; c' : (x) \; \text{nextCn} + d' : (x) \; \text{msg}, o \\
\{ \text{Extend}_{o}(c,c',d) \land \text{Split}_{o}(c, d, d', c', d') \} \\
\text{WriteError of} \; \text{alertDescription option} \\
\text{MustRead of} \; c' : \text{Cn} \{ \ldots \} \\
\text{val} \; \text{write} : c : \text{Cn} \rightarrow d : (x) \; \text{msg}, o \rightarrow (x,d) \; \text{ioresult}, o \\
\text{val} \; \text{connect} : \text{TCP.Stream} ightarrow g : \text{config} \rightarrow c0 : (g)\text{Cn}0 \{\text{Cl}(c0),\text{role} = \text{Client}\} \\
\text{val} \; \text{accept} : \text{TCP.Stream} \rightarrow g : \text{config} \rightarrow c0 : (g)\text{Cn}0 \{\text{Cl}(c0),\text{role} = \text{Server}\} \\
\text{val} \; \text{shutdown} : c : \text{Cn} ightarrow c' : \text{Cn} \{ \ldots \} \\
\text{val} \; \text{rekey} : c : \text{Cn} \{\text{Cl}(c),\text{role} = \text{Client}\} \rightarrow c' : (c)\text{nextCn} \{\text{Extend}(c,c')\} \\
\text{val} \; \text{resume} : \text{TCP.Stream} \rightarrow g : \text{config} \rightarrow \text{sessionID} \rightarrow c0 : (g)\text{Cn}0 \{ \ldots \} \\
\text{val} \; \text{rehandshake} : c : \text{Cn} \{\text{Cl}(c),\text{role} = \text{Client}\} \rightarrow c' : (c)\text{nextCn} \{ \ldots \} \\
\text{val} \; \text{request} : c : \text{Cn} \{\text{Cl}(c),\text{role} = \text{Server}\} \rightarrow c' : (c)\text{nextCn} \{ \ldots \} \\
\text{val} \; \text{authorize} : c : \text{Cn} \rightarrow (x) \; \text{query} \rightarrow (x) \; \text{ioresult}, i \\
\text{val} \; \text{refuse} : c : \text{Cn} \rightarrow (x) \; \text{query} \rightarrow \text{unit}
\end{align*}

Figure 13: Main TLS interface (excerpt).

a new handshake is complete, our interface guarantees that all previous fragments have been received; so, the client knows that \(d_2\) was received, and the server knows that \(d'_1\) was received.

### 6.2 TLS Security

**Adversarial Network** As usual with communications protocols, the adversary is in full control of the network. This is modelled by a trivial TCP implementation, written TCP below, that reads and writes into buffers shared with the adversary. For instance, we define TCP.write as

\begin{align*}
\text{let} \; \text{write ns (b:bytes)} = \text{buffer o} := (\text{ns b}) :: \text{buffer o}
\end{align*}

The application and its adversary may repeatedly set the input buffer, call the TLS interface, and read the output buffer, thereby scheduling any number of parallel connections.

**TLS Security, using the Typed API** Our main theorem is stated for a class of adversaries that range over restricted programs well-typed against the TLS API. As illustrated below, such programs include TLS applications composed with their own adversaries, and our theorem enables the automated security verification of these applications by typechecking. In addition, §6.3 gives a corollary, stated more cryptographically as security for a class of adversaries with oracle access to functions over plain datatypes (bytes, pairs, and integers) rather than those of our API.

38
Let $I_{DS}$ be the dataStream interface (Fig. 12) and $I_{TLS}$ be our main TLS interface (Fig. 13), including auxiliary interfaces such as $I_{Cert}$ to give the adversary control over long-term key management.

**Definition 11** A module $C$ is TLS-secure when it is $(I_{DS}, I_{TCP}) \rightsquigarrow I_{TLS}$-secure for restricted users.

Intuitively, the definition means that TLS, used to communicate application data streams provided by $(DS, A)$, treats data sent over connections with Safe indexes as if it were abstract—only the application is able to create and read them. Moreover, the whole streams are authenticated, interleaved with occurrences of TLS events about the handshake and alerts.

**Theorem 6 (TLS Security)** For any $StAE$ and $HS$ that are $I_{StPlain} \rightsquigarrow I_{StAE}$-secure and $I_{StAE} \rightsquigarrow I_{HS}$-secure for restricted users, the module $StAEPlain \cdot StAE \cdot HS \cdot TLS$ is TLS-secure.

**Proof outline** Recall the definition of Safe$(id)$ as Safe$(HS,(SI(id))) \land$ Strong$(id)$; thus indexes safe for $HS$ and $StAE$ are also safe with regards to our TLS implementation. The main step of the proof is by typechecking our implementation code, that is, $I_{DS} \vdash StPlain \rightsquigarrow I_{StPlain}$ (Lemma $StPlain.tc7$) and $I_{TCP}, I_{DS}, I_{StAE}, I_{HS} \vdash TLS \rightsquigarrow I_{TLS}$ (Lemmas $Dispatch.tc7$ and $TLS.tc7$, where $Dispatch.fs$ is an auxiliary module of TLS that multiplexes between content types.).

We combine Theorems 4, 5, and 6 and summarize them in cryptographic terms as follows: If the cryptographic building blocks of TLS are IND-CPA, INT-CMA, SPRP, and PRF secure for strong record cipher-suites and INT-CMA, CRE, PRF, RSA-PMS, and DH-PMS secure for strong handshake cipher-suites, then TLS is secure when used safely through our API. As illustrated by our sample applications, the safe use of our API can easily be controlled by typing.

### 6.3 Security for ‘untyped’ adversaries

Theorem 6 holds for any composition of applications and their adversaries well-typed against our TLS API. To show that the adversary power is not unduly constrained by typing, we give another, simply-typed API that exports only functions on basic types such as int and bytes and we typecheck its implementation against the main typed API. Cryptographically, this amounts to proving game-based security for adversaries $A$ with oracle access to the TLS API. We apply Theorem 6 to restricted TLS users $(DS_b, UTLS \cdot A)$ defined as follows:

- $DS_b$ is a fixed, typed implementation of DataStream that defines data as an abstract type with oracle functions for creating data from ranges $rg$ and bytes $v$ within that range, and extracting bytes from data, and that, for Safe indexes, passes to TLS either $v$ (when $b = 0$) or a max-sized array of zero bytes (when $b = 1$).

- $UTLS$ is a fixed, typed implementation of our basic TLS API $I_{UTLS}$ that maintains a private table from integers to current states of TLS connections and that exports the same functionalities as the TLS API with base types (see files $UTLS.fs$, $UTLS.fs7$ and lemma $UTLS.tc7$).

For instance, $UTLS$ defines a TLS write oracle of the form

```plaintext
let write (i:int) rg (v:bytes) : int =
match findCn i with
| Some(cn) -> let b = truncate rg v in
  match (TLS.write cn (rg, data rg b)) with
  | WriteComplete(cn') -> updateCn i (Some(cn')); [0 0] | . . .
  | None -> [1 1]
```

39
A ranges over all p.p.t. programs such that we have $I_{TCP} \vdash A$; although we still formally require that $A$ be typed, this does not restrict its power, inasmuch as $I_{UTLS}$ only exports functions on plain data types.

We arrive at a usual cryptographic game (on a large amount of code) in which (1) $A$ needs to distinguish between real encryptions and encryptions of zero; and (2) $A$ attempts to break application integrity.

**Theorem 7 (Game-Based Security)** Let $T$ be TLS-secure.

1. For all p.p.t. adversaries $A$ with access to the oracles defined by the challenger UTLS and TCP: $DS_0 \cdot TCP \cdot T \cdot UTLS \cdot A \approx_{\epsilon} DS_1 \cdot TCP \cdot T \cdot UTLS \cdot A$.

2. For all p.p.t. adversaries $A$ with access to the oracles defined by the challenger UTLS and TCP: $DS_0 \cdot TCP \cdot T \cdot UTLS \cdot A$ is asymptotically safe.

Informally, this means that $A$ cannot win a game defined in terms of matching conversations, for instance by making an honest client apparently open a connection with an honest server and a strong ciphersuites without that server having a matching conversation. 3

### 6.4 Verified TLS Applications

**Ad hoc client authentication** Our first sample application illustrates a typical pattern: an anonymous client and a server establish a TLS connection, then proceed with client-authentication at the application level, relying on shared secret bytes, which may represent a username–password pair, a token, or a secure cookie.

Our sample application security is that, whenever the client sends the authenticator and whenever the server accepts an authenticator as valid, (1) the client and server share a secure session; and (2) the adversary gains no information about the authenticator (hence the client identity). For simplicity, in contrast with our general theorem, we use a strong ciphersuite, a single honest server certificate, and a secure token repository with tokens that fit in a single fragment, so we can specify our application code as:

```>
val client : url → username → token → c:Connection option
val server : unit → username → c:Connection
   { ∃ u. Valid(u,token) ∧ Login(CI(id.in,u,token)) } option
```

To model (1), the client assumes the event $Login(CI(id.out,username, token))$ before sending out his token, and the post-condition of server guarantees that the user is registered and authenticated. Application-level authentication holds only inasmuch as the adversary does not guess the authenticator, with a probability that depends on its min-entropy. We capture this assumption by coding an ideal token functionality that guarantees that honestly generated and coerced (guessed) authenticators never collide.

```>
type token
val create : unit → tk:token{Honest(tk)}
val register : u:string → tk:token{Honest(tk)} → unit{Valid(u,tk)}
val verify : u:string → tk:token → b:bool{b ⇒ Valid(u,tk)}
val coerce : bytes → tk:token{not(Honest(tk))}
```

We define a **DataStream** module that sends tokens (within a given length range) as data at the beginning of the stream:

```>
(val.emptyStream,minTkLen,maxTkLen)) data =
   tk:token{∃ u. Valid(u,tk) ⇒ Login(id.u,tk)}
```
**predicate** Request of epoch * bytes
**predicate** Response of epoch * bytes * bytes
**function** val StreamToBytes: id:epoch * (id) stream → bytes
**type** (id:epoch, h:(id) stream, r:range) data = {
  contents: b:(r) rbytes{
    (∃rq: Request(id, rq))
    ∧ (∃s: StreamToBytes(id, h) @ (b @ s)))
  } ∨ (∃rp: Response(id, rq, rp)
    ∧ (∃s: StreamToBytes(id, h) @ (b @ s)))
}

**val** createRequest:
  id:epoch → s:(id) stream
  { EmptyStream(id, s) }
  → r:range → b:(r) rbytes
  { Request(id, b) } → (id, s, r) data

Figure 14: RPCDataStream interface (excerpt).

so that type abstraction ensures both (1) and (2). F7 shows that our DataStream and application code modules are well typed, using the TLS API and the ideal token interface. This suffices to show that our application is secure, except for the (small) probability that an adversary guesses the authenticator, and the negligible probability that an adversary can break our TLS idealization. Using our length hiding TLS API for authenticators enables us to get this simple guarantee; without it traffic analysis might help guessing attacks, for example, if the token were a compressed HTTP session cookie Duong and Rizzo [2012].

**Secure RPC** Our second application is an RPC library that relies on TLS to exchange multiple requests and responses after mutual authentication. By typechecking our code and applying Theorem 6, we easily obtain secrecy, authenticity, and correlation between requests and responses. The full paper presents an RPC DataStream module that defines data concretely as bytes, with a refinement that says that it must be a fragment of either a serialized request or a serialized response (to handle fragmentation if their size exceeds 16K). By type abstraction, TLS guarantees that RPC will handle and deliver message fragments in accordance with the DataStream interface: messages will be kept secret and will arrive in the right order with strong authentication.

Fig. 14 gives an excerpt of RPCDataStream, and shows how to define the type data to protect the two message exchange. The predicate Request(id, rq) represents a valid request rq in the session id, while Response(id, rq, rp) represents a valid response rp, for the session id and the request rq. The abstract type data is concretely implemented as a byte array with a refinement that says it must be a prefix of either a request or a response. TLS guarantees that it will handle and deliver message fragments in accordance with the DataStream interface: fragments will arrive in order with strong authentication and secrecy guarantees. By typechecking our sample application and applying Theorem 6, we show that RPC over TLS provides integrity and confidentiality against p.p.t. adversaries with access to TCP traffic and oracle access to the RPC interface.

7 Limitations and Future Work

We implemented, tested, and cryptographically verified a reference implementation of TLS. By writing a few hundred lines of F# and F7 code on top of our API, we also confirmed that applications can rely on our theorems to prove end-to-end security while ignoring the low-level details of the RFCs.

Still, our implementation and security theorems come with caveats. We do not yet support some algorithms and ciphersuites (e.g. ECDH, AES-GCM) and we still have to optimize our code for performance (see §2.4). Its security also relies on a large, unverified TCB: the F7 type-
checker, the F# compiler, the .NET runtime, and the core cryptographic libraries. Besides, we do not formally account for side channels attacks based e.g. on timing, even though our implementation tries to mitigate them; proving the absence of such attacks would require specific tools (see e.g. Askarov et al. [2010]).

Our verification method enabled us to develop modular security proofs for a 5KLOC program, based on precise cryptographic assumptions on core primitives. Most proofs are by automatic typechecking, but writing type annotations requires attention and care, and the resulting interfaces amount to 2.5KLOC. Some proofs also rely on usage restrictions (e.g. Definition 7) that are not established by typing, but could be verified using more advanced affine type systems Swamy et al. [2011]. We focus on the standard model of cryptography, resulting in rather strong assumptions for the Handshake, similar to those of Jager et al. [2012] for the DHE key exchange. Relaxing these assumptions and developing concrete security bounds [Bellare et al., 1997] for our implementation is left as important future work.

References


H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *CRYPTO'01*, 2001.


