Implementing TLS with Verified Cryptographic Security

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub

Implementing TLS with Verified Cryptographic Security

Karthikeyan Bhargavan*, Cédric Fournet†, Markulf Kohlweiss†, Alfredo Pironti*, Pierre-Yves Strub‡

*INRIA Paris-Rocquencourt, {karthikeyan.bhargavan,alfredo.pironti}@inria.fr
†Microsoft Research, {fournet,markulf}@microsoft.com
‡IMDEA Software, pierre-yves@strub.nu

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 typechecker. 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.

I. 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. 28, 21–23, 53, 43]. 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 [42]. Some exploit cryptographic design flaws, for instance using knowledge of the next IV to set up adaptive plaintext attacks [47]. Some, such as padding-oracle attacks, use a combination of protocol logic and cryptography, taking advantage of error messages to gain information on encrypted data [56, 17, 57]. Others rely on various implementation errors [14, 44, 38] or side channels [16]. 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 [52, 30]. 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 [51, 20, 32, 49, 48, 29, 36, 33]. 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 (§II), and a fingerprinting attack based on compression (§IV). 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: §IV explains how we reflect fragmentation and length-hiding, to offer some protection against traffic analysis; §VI 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.

(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. [33, fig. 3] and Gajek et al. [29, p. 4].

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.

A. 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. (§II 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 [28, 21–23]. We decided not to support SSL2 at all, since its usage is unsafe and now prohibited [55].

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. [15] 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. [1], 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 [56, 17] as a type-abstraction error. We also independently caught the truncated-MAC attack reported by Paterson et al. [50].

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.

B. 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. [27]. The main technical novelty is to keep track of conditional security using *type indexes* (see §III). 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.

Our presentation focuses on the main API and the interfaces of two core internal modules. The stateful authenticated encryption module (*StAE*), explained in §IV, implements record-layer cryptography. The handshake module (*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 contributions are to verify that the record layer securely implements the *StAE* interface for a range of authenticated encryption mechanisms (Theorem 3 in §IV); the handshake protocol implements the *HS* interface, with security guarantees when using RSA and DH (Theorem 4 in §V); 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 5 in §VI).

Prior Verification Work on TLS Implementations. We limit our discussion of related work to the verification of implemen-

tations; other works on formal aspects of TLS are discussed through the paper. To our knowledge, Bhargavan et al. [11] 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 [12] and CryptoVerif [13]. 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 [18] verify the SSL 2.0/3.0 handshake implementation in OpenSSL by model checking. Their analysis finds rollback attacks but applies only to fixed configurations, and they assume a symbolic model of cryptography. Others [35, 3] verify Java implementations of the handshake protocol using logical provers, also in the symbolic model.

Contents The paper is organized as follows. §II informally presents and evaluates our modular reference implementation. §III explains cryptographic verification by typing. §IV handles stream encryption. §V deals with the handshake. §VI presents our main API and theorems for TLS. §VII discusses limitations of our approach and future work.

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

II. A Modular Implementation of TLS

A. API Overview

Our application interface (see Fig. 4 in §VI) 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.

B. Modules and Interfaces

Our implementation is written in F#, a variant of ML for the .NET platform, and specified in F7 [9]. 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, §III 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 [19], an SMT solver. Our 'makefile' automates the process of verifying modules while managing their dependencies, with a target *<Module>.tc7* (representing a typing lemma) for typechecking each *<Module>*.

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. The numbers in Fig. 1(a) 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.
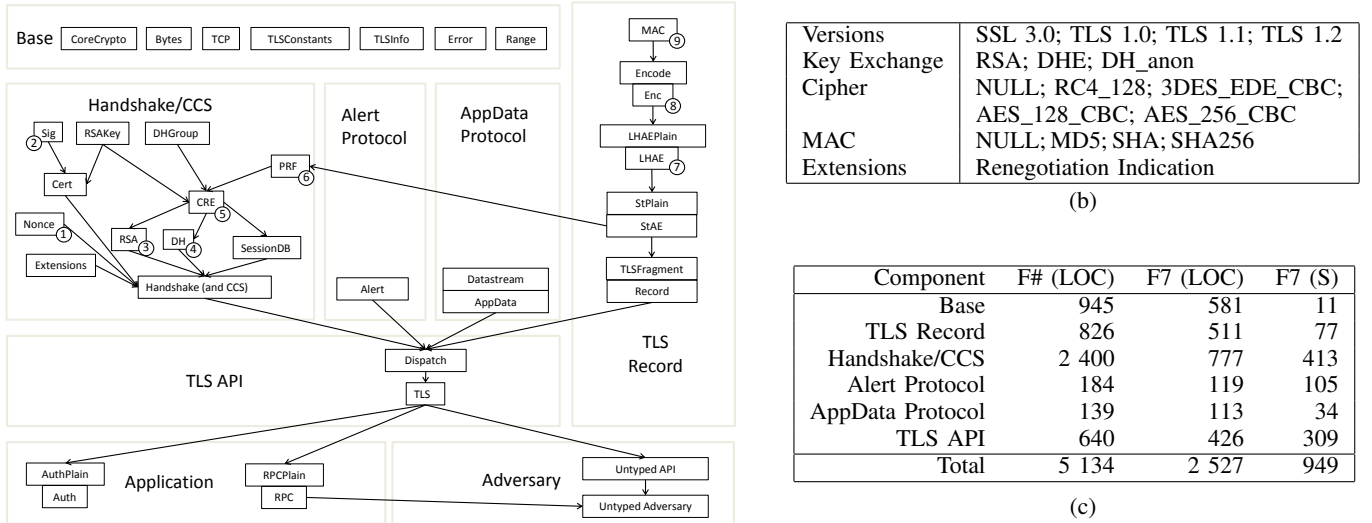
Figure 1. (a) Modular implementation of TLS, (b) Implemented features and algorithms, (c) Code size and verification time.

Versions	SSL 3.0; TLS 1.0; TLS 1.1; TLS 1.2
Key Exchange	RSA; DHE; DH_anon
Cipher	NULL; RC4_128; 3DES_EDE_CBC; AES_128_CBC; AES_256_CBC
MAC	NULL; MD5; SHA; SHA256
Extensions	Renegotiation Indication

(b)

Component	F# (LOC)	F7 (LOC)	F7 (S)
Base	945	581	11
TLS Record	826	511	77
Handshake/CCS	2 400	777	413
Alert Protocol	184	119	105
AppData Protocol	139	113	34
TLS API	640	426	309
Total	5 134	2 527	949

(c)

C. Modular Architecture for TLS

Fig. 1(a) 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; *CoreCrypto* 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 §V) 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 §IV. 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 §III. 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.

The handshake protocol, detailed in §V, 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.

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 §VI. Before evaluating our implementation, we discuss two attacks involving fragmentation and multiple epochs.

Renegotiating Peer Identities (an existing attack) Ray [52]

presents an attack exploiting the mis-attribution of application data to epochs. Until a a recent protocol extension [53], 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 [53]. 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 messages can be sent and received before completing the first handshake. Unlike handshake messages, 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 implementations 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.

D. Experimental Evaluation

Our implementation currently supports the protocol versions, algorithms, and extensions listed in Fig. 1(b), 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 1.0.1e and GnuTLS 3.1.4, and against the NSS 3.12.8 and the Oracle JSSE 1.7 libraries. We also implemented the .NET Stream interface on top of our TLS API, used it to program a multi-threaded HTTPS server, and tested it against Firefox 16.0.2, Safari 6.0.2, Chrome

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

Figure 2. Performance benchmarks (OpenSSL 1.0.1e as server).

23.0.1271.64 and Internet Explorer 9.0.5 web browsers, using different protocol versions and ciphersuites. Conversely, we programmed and tested an HTTPS client against an Apache 2.2.20-mod-ssl web server. Our implementation correctly interoperates, both in client and in server mode, with all these implementations, for all the protocol versions and ciphersuites we support. Of these, NSS only implements up to TLS 1.0 and Oracle JSSE does not support AES256. Our interoperability tests included session resumption, rekeying, and renegotiation.

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 one another, running on the same host to minimize network effects. Figure 2 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

5

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. 1(c) 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 implementation takes 15 minutes on a modern desktop.

III. 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 [9] has been extended in several directions [10, 54, 5–7]; in particular Swamy et al. [54] provide a mechanized theory for a language that subsumes F7. We follow the notations and results of its probabilistic variant [27], 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 $I_1, \ldots, I_\ell \vdash A \rightsquigarrow I$, when it correctly implements I using modules with interfaces $I_1, \ldots, I_\ell$. 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 ϕ range over first-order logical formulas on F# values. The refinement type $x{:}t\{\phi\}$ represents values x of type t such that formula ϕ holds (the scope of x is ϕ). For instance, $n{:}\mathrm{int}\{0 \leq n\}$ 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 $b{:}\mathrm{bytes}\{Length(b){=}16\}$, the type of 16-byte arrays, uses a logical function *Length* on bytes. and, to verify that byte arrays have this type, it may be enough to specify *Length* for empty arrays and concatenations. Refinements may specify data formats as above (for integrity) and also track runtime events (for authenticity). For instance, $c{:}cert\{Authorized(u,c)\}$ 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** ϕ) in F# code and F7 interfaces; conversely, they may appear in verification goal, expressed as assertions (**assert** ϕ). Logical specifications and assumptions must be carefully written and reviewed, since they condition our security interpretation of types [see e.g. 10, 54].

Abstract Types An interface may declare a type as abstract (e.g. **type** *key*) and keep its representation 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. 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 dependent function type $x{:}t\{\phi\} \rightarrow y{:}t'\{\phi'\}$ represents functions with pre-condition ϕ and post-condition ϕ' (the scope of x is ϕ, t' and ϕ'), 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. [27] formalize a probabilistic variant of F7 and develop a framework for the modular cryptographic verification of protocols coded in F#. (Küsters et al. [41] adopt a similar approach for programs in Java.) We recall their main theorems.

A program is *safe* if, in every run, every **assert** logically follows from prior **assume**s. The main property of the type system is that well-typed expressions are always safe.

Theorem 1 (Type Safety [27]): If $\varnothing \vdash A : t$, 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_α with an abstract type α 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. With suitable restrictions on I_α, the result of the program then does not depend on secrets and their operations:

Theorem 2 (Secrecy by Typing [27]): If $\varnothing \vdash P_b \rightsquigarrow I_\alpha$ for $b = 0, 1$ and $I_\alpha \vdash A : bool$, then $P_0 \cdot A \approx P_1 \cdot A$.
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 α 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 [54] 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, [27] also defines asymptotic notions of safety and secrecy for expressions A_η parameterized by a *security parameter* η, 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_η 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 η) are *asymptotically equivalent*, written $A_0 \approx_\epsilon A_1$, when the statistical distance $\frac{1}{2}\sum_{M=true,false}|Pr[A_0 \Downarrow M] - Pr[A_1 \Downarrow M]|$ is negligible. 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 [41] for a detailed discussion of polynomial-time notions for code-based simulation-based security.)

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 §IV and §VI 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*. 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. Decryption returns an option, that is, either some plaintext or none, in case of decryption error.

val *ENC*: *p:plain* $\rightarrow$ *c:cipher*
val *DEC*: *c:cipher* $\rightarrow$ *o:plain 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, this may be defined as $C \cdot G_0 \cdot A \approx_\epsilon C \cdot G_1 \cdot A$.

Typical games for modeling the authenticity and confidentiality of AE are *INT-CTXT* and *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. (See [27] for 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. For primitives such as AE, we may design F so that C itself is a valid simulator, i.e. $C \cdot A \approx_\epsilon C \cdot F \cdot A$. 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.

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 (*INT-CTXT*):

val *ENC*: *p:plain* $\rightarrow$ *c:cipher* {*ENCrypted(p,c)*}
val *DEC*: *c:cipher* $\rightarrow$ *o:(plain option)*
$$\{\forall p.\ o=Some(p) \Leftrightarrow ENCrypted(p,c)\}$$

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

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

Definition 1: C is I_C^i-secure if there exists a module C^i with $\vdash C^i \leadsto I_C^i$ such that, for all p.p.t. expressions A with $I_C^i \vdash A$, we have $C \cdot A \approx_\epsilon C^i \cdot A$.

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

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

Definition 2: A plain interface I_{Plain} is of the form

type *repr* = *b*:bytes {*Length(b) =plainsize*}
type *plain*
val *repr*: *plain* $\rightarrow$ *repr*
val *plain*: *repr* $\rightarrow$ *plain*

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

Definition 3: C is $I_{Plain}^i \leadsto I_C^i$-secure when there exists a module C^i with $I_{Plain}^i \vdash C^i \leadsto I_C^i$ such that, for all p.p.t. modules P with $\vdash P \leadsto I_{Plain}^i$, $\vdash P \leadsto I_{Plain}$, and A with $I_{Plain}, I_C^i \vdash A$, 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** *plain* = *m:repr{Msg(m)}* where *Msg* is the protocol specification of an authentic plaintexts and then rely on typing to ensure authenticity of decrypted plaintexts.

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^i-secure, and C' is $I_{C'}^i$-secure, then $C \cdot C'$ is $I_C^i, I_{C'}^i$-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 *GEN*: *id:index* $\rightarrow$ *(;id)k*. The user provides the index, and type safety guarantees that materials with different indexes are not mixed. For example, an ideal multi-instance interface for AE is:

type *(;id:index)key*
val *GEN*: *id:index* $\rightarrow$ *(;id)key*
val *ENC*: *id:index* $\rightarrow$ *(;id)key* $\rightarrow$ *p:(;id)plain* $\rightarrow$
  c:cipher {*ENCrypted(id,p,c)*}
val *DEC*: *id:index* $\rightarrow$ *(;id)key* $\rightarrow$ *c:cipher* $\rightarrow$
  o:(;id)plain option { $\forall p.\ o = Some(p) \Leftrightarrow ENCrypted(id,p,c)$ }

This interface is parameterized by a plain module that defines an indexed abstract type $(;id{:}index)plain$, and uses an *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 4: A program A is a restricted user of I_C^i when $I_C^i \vdash A$ and A calls *GEN* with pairwise distinct indexes.

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, *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 *StrongAuth* for authenticity and *Strong* for authenticated encryption. Hence, our postcondition of *DEC* now is $\{StrongAuth(id) \Rightarrow (\forall p.o = Some(p) \Leftrightarrow ENCrypted(id,p,c))\}$

We also generalize our ideal plain interface, leaving the *plain* and *repr* functions available, but with preconditions that restrict their usage to weak algorithms:

val *plain*: $id{:}index\{\ not(StrongAuth(id))\ \} \to repr \to (;id)plain$
val *repr*: $id{:}index\{\ not(Strong(id))\ \} \to (;id)plain \to repr$

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.

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, *Corrupt(id)*, that holds when keys are corrupted. To provide the adversary with the possibility to compromise keys we extend our indexed interfaces I_C^i with functions

val *LEAK*: $id{:}index\ \{Corrupt(id)\} \to (;id)key \to \text{bytes}$
val *COERCE*: $id{:}index\ \{Corrupt(id)\} \to \text{bytes} \to (;id)key$

and we adapt our ideal interfaces to provide security guarantees conditioned by the predicate *not(Corrupt(id))*, e.g., for AE, the postcondition of *DEC* becomes $\{\ not(Corrupt(id)) \wedge StrongAuth(id) \Rightarrow (\forall p.\ o = Some(p) \Leftrightarrow ENCrypted(id,p,c))\}$.

As noted, e.g., by [4, 40], an idealized module C^i 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. To avoid the commitment problem, we require *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 in-

stance about the ciphersuite and certificates used in the handshake to generate the keys. In §IV, we will use two main predicate on indexes, *Safe* that guarantees both authenticity and secrecy for the transport layer, and *Auth* that guarantees authenticity but not necessarily secrecy, logically defined as $Auth(id) \overset{\triangle}{=} not(Corrupt(id)) \wedge StrongAuth(id)$ and $Safe(id) \overset{\triangle}{=} not(Corrupt(id)) \wedge Strong(id)$. For simplicity, we do not model the independent corruption of connections after key establishment, so the *Corrupt* predicate will be determined by the handshake, as the negation of its *Honest* predicate on long-term keys.

IV. 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 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; (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.

Traffic Analysis and Fingerprinting Even with perfect cryptography, traffic analysis yields much information about TLS applications [25]. For example, compression may reveal redundancy in the plaintext when both plaintext and ciphertext lengths are known [37]; 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 subsequences of ciphertext lengths. The attack is most effective against RC4 ciphersuites, but also succeeds against block ciphers with minimal padding.

Accordingly, our implementation disables compression, and our formal results apply only to connections where TLS-level compression is disabled.

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; 1|], …, [|255;…; 255|] as long as the resulting plaintext is block-aligned. Most implementations use minimal padding; others, such as GnuTLS [45], randomly select any of the correct paddings, but per-fragment padding schemes are often statistically ineffective [25].

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. Extending this idea, we implement a deterministic fragmentation and padding algorithm for block ciphers based only on ranges.

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

Length-Hiding Authenticated Encryption (LHAE) We define $I^i_{LHAEPlain} \leadsto I^i_{LHAE}$ security for the plaintext interface $I^i_{LHAEPlain}$ outlined below.

type (;*id*:*index*,*ad*:(;*id*)*data*,*r*:*range*) *plain*
type (;*r*:*range*) *rbytes* = *b*:bytes{ $fst(r) \leq Length(b) \leq snd(r)$ }
val *plain*: *id*:*index*{*not*(*Auth*(*id*))} →
   r:*range* → *ad*:(;*id*)*data* → (;*r*)*rbytes* → (;*id*,*ad*,*r*) *plain*
val *repr*: *id*:*index*{*not*(*Safe*(*id*))} →
   r:*range* → *ad*:(;*id*)*data* → (;*id*,*ad*,*r*) *plain* → (;*r*)*rbytes*

Each plaintext is indexed by an instance *id*, its additional data *ad*, and its range *r*. We use the refined type (;*r*)*rbytes* for concrete representation of plaintexts that fit in range *r*. The functions *plain* and *repr* translate between concrete and abstract plaintexts. As explained in §III, their precondition states that LHAE can use them only on weak *id*s (e.g. for weak ciphersuites or corrupt keys).

We define the interface I^i_{LHAE} parametrized by $I^i_{LHAEPlain}$; we omit its *COERCE* and *LEAK* functions for brevity.

type (;*id*:*index*) *key*
val *GEN*: *id*:*index* → (;*id*) *key*

val *ENC*: *id*:*index* → *k*:(;*id*) *key* → *d*:(;*id*) *data* → *r*:*range* →
  p:(;*id*,*d*,*r*) *plain* → (*k'*:(;*id*)*key* ∗ *c*:*cipher*)
  {*CipherRange*(*id*,*r*,*c*) ∧ *ENCrypted*(*id*,*d*,*p*,*c*)}
val *DEC*: *id*:*index* → *k*:(;*id*) *key* → *d*:(;*id*) *data* → *c*:*cipher* →
  o:(*k'*:(;*id*) *key* ∗ *r* :*range* {*CipherRange*(*id*,*r*,*c*)} ∗
    p :(;*id*,*d*,*r*) *plain*) *option*
  {*Auth*(*id*) ⇒ !*k'*,*r*,*p*, (*o* = *Some*(*k'*,*r*,*p*) ⇔ *ENCrypted*(*id*,*d*,*p*,*c*))}

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.

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).

ENCrypted(*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 *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 *Auth*(*id*) and *Safe*(*id*) hold. For instance, when *Safe* (*id*) holds, the user (including the adversary) may learn the values of their indexes *id*, *d*, *r*, but cannot call the *repr* function to read their content, nor call the *plain* function to forge their content.

Our implementation supports many protocol versions and ciphersuites, but provides security only for *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 [39, Theorem 2] that states that *IND-CPA* security of encryption and combined *INT-CTXT* security of MAC-then-encrypt afford secure channels. Krawczyk also shows that stream ciphers as used in TLS provide combined *INT-CTXT* security. We use the result of Paterson et al. [50] to show that the block-cipher-based schemes implemented by our LHAE module are combined *INT-CTXT* secure, despite the unauthenticated padding, for strong block ciphers and MAC algorithms.

Our concrete implementation of LHAE is a sequence of modules $C \triangleq MAC \cdot Encode \cdot ENC \cdot LHAE$. Under the combined *INT-CTXT* assumption, we prove by typing that C is $I^i_{LHAEPlain} \leadsto I^i_{LHAE}$-secure for *IND-CPA* secure modules ENC and for restricted users (using *LHAE* keys linearly with pairwise-distinct additional data).

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 $I^i_{StPlain}$ introduces a fourth index: a *log* that records the sequence of preceding plaintexts and additional data.

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

We omit its *plain* and *repr* declarations similar to those of I^i_{LHAE}. The ideal interface I^i_{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*') = (*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:(*r*:*range* {*CipherRange*(*id*,*r*,*c*)} ∗ *p*:(;*id*,*Log*(*rd*),*d*,*r*) *stplain* ∗
    rd':(;*id*) *reader*{*Log*(*rd*') = (*d*,*p*)::*Log*(*rd*)}) *option*
    {*Auth*(*id*) ⇒ (!*rd*',*r*,*p*. *o* = *Some*(*rd*',*r*,*p*) ⇔
        (∃*wr*. *ENCrypted*(*id*,*wr*,*d*,*p*,*c*) ∧ *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 TLS, the additional data for *StAE* contains the protocol version and content type; *StAE* 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 $I^i_{LHAEPlain}$ using $I^i_{StPlain}$. Then, for instance, *StAE.ENC* simply adds a sequence number then invokes *LHAE.ENC*. By typing (Lemma *StAE.tc7*), 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 3 (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 = MAC · Encode · ENC · LHAE and S = LHAEPlain · C · StAE.

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

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

V. 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.

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}.

Data Structures We give below the public datatypes of the API that expose information about sessions and epochs to the application. Our main integrity goal for the handshake is that clients and servers agree on their content.

type *SessionInfo* = {
  init_crand: *random*;
  init_srand: *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_rand*: *random*;
  id_in: *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 *ConnectionInfo*s are pairwise distinct. 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.

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.

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 (see Fig. 4 for their counterparts in the main TLS API): *accept* creates a server instance (possibly resuming an existing session, at the client's initiative); *connect* creates a client instance (with a fresh session); *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. 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.

The *Complete* predicate in the postcondition of connection establishment (see *Handshaken* in Fig. 4) 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. Provided that (1) both the ciphersuite and all its algorithms in *si* are strong (predicate *StrongHS*(*si*), explained shortly); and (2) the long-term keys recorded in *si* are honest (predicate *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 *SentCCS*(*ci.id_in*)); and (c) the handshake was actually secure (predicate *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 §VI-D.

Handshake Security We define security for the ideal handshake interface I_{HS}^i used in our verification, and parameterized by I_{StAE}^i, the ideal interface for StAE in §IV that defines the type of keys established by the handshake.

Definition 5: A module *HS* is a secure handshake when it is $I_{StAE}^i \leadsto I_{HS}^i$-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 *not*(*Auth*(*id*)), the pre-condition of *COERCE*. Thus, Definition 5 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_{HS}^i uses a predicate *SafeHS* on SessionInfo to indicate the secure runs of the handshake, such that *Auth*(*id*) implies *SafeHS*(*SI*(*id*)). To type the handshake, we let *SafeHS* (*si*) $\triangleq$ *StrongHS*(*si*) $\land$ *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 [34] and the *PRF-ODH* assumption of [33] respectively (see the full paper for details). Thus we define

$$StrongHS(si) \triangleq StrongSig(si) \land StrongCRE(si) \land StrongPRF(si) \land \\ (StrongRSAPMS(si) \lor StrongDHPMS(si))$$

For example, if the ciphersuite of *si* matches `TLS_DHE_DSS_WITH_*`, *StrongHS*(*si*) holds if the signature scheme *DSS* is *INT-CMA* secure [31], *CRE* and *DH* are jointly *DH-PMS* secure, *CRE* is a computationally strong randomness extractor [26], and *PRF* is a pseudo-random function, and similarly for RSA-based ciphersuites.

Theorem 4 (Handshake): If Nonce is I_{Nonce}^i-secure, Sig is I_{Sig}^i-secure, CRE is $I_{PRF}^i \leadsto I_{CRE}^i$-secure, PRF is $I_{StAE}^i \leadsto I_{PRF}^i$-secure, and we have

(RSA-PMS) $\quad$ $RSAKey \cdot CRE \cdot RSA \approx_\epsilon RSAKey \cdot CRE \cdot RSA^i$,
(DH-PMS) $DHGroup \cdot CRE \cdot DH \approx_\epsilon DHGroup \cdot CRE \cdot DH^i$,
then $HS \triangleq Nonce \cdot Sig \cdot RSAKey \cdot Cert \cdot PRF \cdot DHGroup \cdot CRE \cdot RSA \cdot DH \cdot TLSExt \cdot Handshake$ is $I_{StAE}^i \leadsto I_{HS}^i$ 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 4 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 which for the ciphersuites `TLS_DHE_DSS_WITH_*` is very close to Jager et al. [33]. Their proof 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 [46].

After idealization, we apply typing Lemmas to verify by typing that the idealized handshake meets I_{HS}^i. The *Handshake* module itself, the largest and most complex in our codebase,

type (;*id*:*epoch*) *stream*
type (;*id*:*epoch*, *h*:(;*id*)*stream*, *r*:*range*) *data*
val *data*:
 id:*epoch*{*not*(*Auth*(*id*))} → *s*:(;*id*) *stream* → *r*:*range* →
 b:(;*r*) *rbytes* → *c*: (;*id*,*s*,*r*) *data*
val *repr*:
 id:*epoch*{*not*(*Safe*(*id*))} → *s*:(;*id*) *stream* → *r*:*range* →
 c: (;*id*,*s*,*r*) *data* → (;*r*) *rbytes*
val *split*: *id*:*epoch* → *s*:(;*id*) *stream* →
 r0:*range* → *r1*:*range* → *d*:(;*id*,*s*,*Sum*(*r0*,*r1*)) *data* →
 d0:(;*id*,*s*,*r0*) *data* ∗ *d1*:(;*id*,*ExtendStream*(*id*,*s*,*r0*,*d0*),*r1*) *data*

Figure 3. DataStream interface towards TLS (excerpt).

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.

VI. MAIN API & THEOREMS FOR TLS

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

A. TLS API

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

- *Auth*(*id*), defined as *SafeHS*(*SI*(*id*)) ∧ *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*)) ∧ *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 §IV), 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. 3 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 §IV, *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 *r0* and *r1* 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.

Main TLS Interface Fig. 4 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 §V). 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 §V).

- *read* takes the current state and returns an *ioresult_i*, with different cases: *Read*(*c*,*d*) returns an updated connection 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 *ioresult_o* 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): *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*).

TLS does not guarantee synchronization between input and output streams; for instance, the client may write three messages d_0, d_1, d_2 then read d'_0, then initiate rekeying, while

```
type Cn
type (;g:config) Cn0 = c0:Cn{InitCn(g,c0)}
type (;c:Cn) nextCn = c':Cn{NextCn(c,c')}
type (;c:Cn) msg_i = r:range * (;CI(c).id_in, Stream_i(c), r) data
type (;c:Cn) msg_o = r:range * (;CI(c).id_out, Stream_o(c), r) data
type (;c:Cn) ioresult_i =
| Read of c':(;c) nextCn * d:(;c) msg_i
  {Extend_i(c,c',d) ∧ (Auth(CI(c).id_in) ⇒ Write(CI(c).id_in, Bytes_i(c'))) }
| Close of TCP.Stream{Auth(CI(c).id_in) ⇒ Close(CI(c).id_in, Bytes_i(c))}
| Fatal of a:alertDescription
  {Auth(CI(c).id_in) ⇒ Fatal(CI(c).id_in,a,Bytes_i(c))}
| CertQuery of c':(;c) nextCn * (;c') query {Extend(c, c')}
| Handshaken of c':Cn {Complete(CI(c'),Cfg(c')) ∧ ...}
| ...
val read : c:Cn → (;c) ioresult_i

type (;c:Cn,d:(;c) msg_o) ioresult_o =
| WriteComplete of c':(;c) nextCn {Extend_o(c,c',d)}
| WritePartial of c':(;c) nextCn * d':(;c') msg_o
  { ∃d0. Extend_o(c,c',d0) ∧ Split_o(c, d, d0, c', d') }
| WriteError of alertDescription option
| MustRead of c':Cn {...}
val write: c:Cn → d:(;c) msg_o → (;c,d) ioresult_o

val connect: TCP.Stream → g:config → c0:(;g)Cn0{CI(c0).role = Client}
val accept: TCP.Stream → g:config → c0:(;g)Cn0{CI(c0).role = Server}
val shutdown: c:Cn → c':Cn{...}
val rekey: c:Cn {CI(c).role=Client} → c':(;c)nextCn{Extend(c,c')}
val resume: TCP.Stream → g:config → sessionID → c0:(;g)Cn0{...}
val rehandshake: c:Cn {CI(c).role=Client} → c':(;c)nextCn{...}
val request: c:Cn {CI(c).role=Server} → c':(;c)nextCn{...}
val authorize: c:Cn → (;c) query → (;c) ioresult_i
val refuse: c:Cn → (;c) query → unit
```

Figure 4. Main TLS interface (excerpt).

the server reads d_0, write d'_0 and d'_1, then reads d_1. On the other hand, when notified of a Close or that 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.

B. TLS Security

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. 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.

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, §VI-C 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. Let I_{DS}^i be the dataStream interface (Fig. 3) and I_{TLS}^i be our main TLS interface (Fig. 4), including auxiliary interfaces such as I_{Cert} to give the adversary control over long-term key management.

Definition 6: A module C is TLS-secure when it is $(I_{DS}^i, I_{TCP}) \rightsquigarrow I_{TLS}^i$-secure for restricted users.

Intuitively, the definition means that TLS 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 5 (TLS Security): For any StAE and HS that are $I_{StPlain}^i \rightsquigarrow I_{StAE}^i$-secure and $I_{StAE}^i \rightsquigarrow I_{HS}^i$-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 *SafeHS(SI(id))* ∧ *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, $I_{DS}^i \vdash StPlain \rightsquigarrow I_{StPlain}^i$ (Lemma *StPlain.tc7*) and $I_{TCP}, I_{DS}^i, I_{StAE}^i, I_{HS}^i \vdash TLS \rightsquigarrow I_{TLS}^i$ (Lemmas *Dispatch.tc7* and *TLS.tc7*, where *Dispatch.fs* is an auxiliary module of TLS that multiplexes between content types.).

We combine Theorems 3, 4, and 5 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.

C. Security for 'untyped' adversaries

Theorem 5 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 5 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.
- A ranges over all p.p.t. programs such that we have $I_{TCP}, I_{UTLS} \vdash A$; 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 6 (Game-Based Security): Let T be TLS-secure. *For all p.p.t. adversaries A with access to the oracles defined*

by the challenger UTLS and TCP, we have (1) $DS_0 \cdot TCP \cdot T \cdot UTLS \cdot A \approx_\epsilon DS_1 \cdot TCP \cdot T \cdot UTLS \cdot A$ and (2) $DS_0 \cdot TCP \cdot T \cdot UTLS \cdot A$ is asymptotically safe.

D. 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 → *u*:*username* ∗ *c*:*Connection*
  { ∃*token*. *Valid*(*u*,*token*) ∧ *Login*(*CI*(*c*).*id_in*,*u*,*token*) } *option*

To model (1), the client assumes the event *Login*(*CI*(*c*).*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 *coerce*d (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:

(;*id*,*emptyStream*,(*minTkLen*,*maxTkLen*)) *data* =
    tk:*token*{∃*u*. *Valid*(*u*,*tk*) ⇒ *Login*(*id*,*u*,*tk*)}

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 [24].

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 5, 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.

VII. 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 cipher-suites (e.g. ECDH, AES-GCM) and we still have to optimize our code for performance (see §II-D). Its security also relies on a large, unverified TCB: the F7 typechecker, 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. [2]).

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 4) that are not established by typing, but could be verified using more advanced affine type systems [54]. We focus on the standard model of cryptography, resulting in rather strong assumptions for the Handshake, similar to those of Jager et al. [33] for the DHE key exchange. Relaxing these assumptions and developing concrete security bounds [8] for our implementation is left as important future work.

References

[1] T. Acar, M. Belenkiy, M. Bellare, and D. Cash. Cryptographic agility and its relation to circular encryption. In *EUROCRYPT*, pages 403–422, 2010.

[2] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, pages 297–307, 2010.

[3] M. Avalle, A. Pironti, D. Pozza, and R. Sisto. JavaSPI: A framework for security protocol implementation. *International J. of Secure Software Engineering*, 2:34–48, 2011.

[4] M. Backes and B. Pfitzmann. Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In *CSFW*, pages 204–218, 2004.

[5] M. Backes, C. Hritcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. In *FCS*, 2009.

[6] M. Backes, M. Maffei, and D. Unruh. Computational sound verification of source code. In *CCS*, 2010.

[7] M. Backes, C. Hrițcu, and M. Maffei. Union and intersection types for secure protocol implementations. In *TOSCA'11*, pages 1–28, 2012.

[8] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, pages 394–403, 1997.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 33(2):8, 2011.

[10] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, pages 445–456, 2010.

[11] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified Cryptographic Implementations for TLS. *ACM TISSEC*, 15(1): 1–32, 2012.

[12] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW*, pages 82–96, 2001.

[13] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE S&P*, pages 140–154, 2006.

[14] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *CRYPTO'98*, pages 1–12, 1998.

[15] B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *CT-RSA*, 2011.

[16] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security*, pages 1–14, 2003.

[17] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a ssl/tls channel. In *CRYPTO*, pages 583–599, 2003.

[18] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF 2009*, pages 172–185, 2009.

[19] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963, 2008.

[20] G. Díaz, F. Curtero, V. Valero, and F. Pelayo. Automatic verification of the TLS handshake protocol. In *SAC*, pages 789–794, 2004.

[21] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, 1999.

[22] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, 2006.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.

[24] T. Duong and J. Rizzo. The CRIME attack. 2012. Ekoparty.

[25] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE S&P*, pages 332–346, 2012.

[26] P.-A. Fouque, D. Pointcheval, and S. Zimmer. Hmac is a randomness extractor and applications to tls. In *ASIACCS*, pages 21–32, 2008.

[27] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *ACM CCS*, pages 341–350, 2011.

[28] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (SSL) protocol version 3.0 – 1996. RFC 6101, 2011.

[29] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk. Universally composable security analysis of TLS. In *ProvSec*, pages 313–327, 2008.

[30] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS*, pages 38–49, 2012.

[31] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[32] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *CCS'05*, pages 2–15, 2005.

[33] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293, 2012.

[34] J. Jonsson and J. B. S. Kaliski. On the security of RSA encryption in TLS. In *CRYPTO*, pages 127–142, 2002.

[35] J. Jürjens. Security analysis of crypto-based java programs using automated theorem provers. In *ASE'06*, pages 167–176, 2006.

[36] A. Kamil and G. Lowe. Analysing TLS in the strand spaces model. Technical report, Oxford University Computing Laboratory, 2008.

[37] J. Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption*, pages 95–102. IACR, 2002.

[38] V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

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

[40] R. Küsters and M. Tuengerthal. Universally composable symmetric encryption. In *CSF*, 2009.

[41] R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of java-like programs. In *CSF*, pages 198–212, 2012.

[42] A. Langley. Unfortunate current practices for HTTP over TLS, 2011. http://www.ietf.org/mail-archive/web/tls/current/msg07281.html.

[43] N. M. Langley, A. and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, 2010.

[44] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in OpenSSL using coccinelle. In *EDCC'10*, 2010.

[45] N. Mavrogiannopoulos and S. Josefsson. GnuTLS documentation on record padding, 2011. http://www.gnutls.org/manual.

[46] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the tls protocol. In *CCS*, pages 62–72, 2012.

[47] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. http://www.openssl.org/~bodo/tls-cbc.txt, 2004.

[48] P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. In *ASIACRYPT'08*, pages 55–73, 2008.

[49] K. Ogata and K. Futatsugi. Equational approach to formal analysis of TLS. In *ICSCS*, pages 795–804, 2005.

[50] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT 2011*, pages 372–389, 2011.

[51] L. C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM TISSEC*, 2(3):332–351, 1999.

[52] M. Ray. Authentication gap in TLS renegotiation. http://extendedsubset.com/Renegotiating_TLS.pdf, 2009.

[53] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. RFC 5746, 2010.

[54] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

[55] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176, 2011.

[56] S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In L. R. Knudsen, editor, *EUROCRYPT*, pages 534–546, 2002.

[57] A. K. L. Yau, K. G. Paterson, and C. J. Mitchell. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In *FSE*, pages 299–319, 2005.