Transport Layer Security purely in OCaml

Hannes Mehnert [*]
University of Cambridge

David Kaloper Meršinjak [†]
University of Cambridge

Abstract

Transport Layer Security (TLS) is probably the most widely deployed security protocol on the Internet. It is used to setup virtual private networks, secure various services such as web and email, etc. In this paper we describe our clean slate TLS implementation developed in OCaml. Our motivating goals are reliability, robustness, and API conciseness. While our implementation is still a work in progress and lacks some of the protocol features (such as client authentication and session resumption), it already interoperates with other existing TLS implementations[1]. Preliminary performance evaluation shows that our library is roughly five times slower compared to OpenSSL, but we expect to improve our performance.

1. Introduction

The Transport Layer Security (TLS) protocol has been standardized by the IETF 15 years ago, and provides communication privacy and authenticity: communication between applications which use this protocol is immune to eavesdropping, tampering, and message forgery. TLS is widely deployed for securing web services (HTTPS), email communication, virtual private networks, and wireless networks.

The variety of used TLS implementations is rather small: most programming language bind OpenSSL, an open source implementation written in C. There are three main reasons to interface an existing TLS library instead of developing one from scratch: deployment of different TLS versions (SSL 3, TLS 1.0, TLS 1.1, and TLS 1.2), protocol complexity (ASN.1 encoding, X509 verification, and crypto primitives), and numerous attacks (comprehensive list (Meyer and Schwenk 2013)). The disadvantage of a monoculture of TLS libraries is that an issue in the single library can lead to a catastrophe (as recently observed with the Heartbleed issue). OCaml-TLS is available under a BSD license on GitHub[2] and via OPAM.

The motivation for our implementation is manifold:

Reliability OCaml provides us with type safety and memory safety. Its unique module system is essential for development of robust and reliable implementations of security protocols: units of code can be developed and reasoned about in isolation, with type abstraction enforcing separation of concerns. Furthermore, we developed TLS using applicative style: the library core does not use mutable data structures[3], making it much easier to reason about. The code base of our current prototype is at least an order of magnitude smaller than OpenSSL[4]. We are aware of several attacks on the TLS protocol, and implement and discuss mitigations[5] in a transparent way. While OCaml's memory safety prevents certain vulnerabilities such as out-of-bounds memory accesses, it might open further attack surfaces (connected to timing and garbage collector) - this facet is still unexplored.

Conciseness of API Research on the certificate verification (Georgiev et al. 2012; Brubaker et al. 2014) discovered that badly designed APIs of existing TLS implementations often lead developers to use the libraries incorrectly, especially with respect to certificate verification. We develop from scratch, learn from their failures and strive to design APIs which are easy to use correctly.

Robustness Since the core is *pure*, it is much easier to test both against known cases and in an automated fashion. We developed the code with robustness in mind, and are looking forward to rigorously test it using execution traces checked by miTLS (Bhargavan et al. 2013) and other TLS implementations.

2. Description

In this section we describe the TLS protocol in more detail, followed by preliminary performance results of our implementation.

2.1 Protocol Description

The TLS protocol establishes a secure channel between a client and a server. It supports different key exchange methods, encryption algorithms and MAC algorithms. These are negotiated on a per-connection basis together with the protocol version within the initial handshake. Each handshake establishes shared secrets for the encryption algorithm and the MAC key.

Client and server can mutually authenticate each other, using X509 certificates. Usually only the server authenticates itself by presenting a certificate chain consisting of a server certificate (which contains the server name), some intermediate certificates, and the last intermediate must be signed by a certificate which is in the set of trust anchors on the client. The set of trust anchors is usually deployed with the client software.

Several attacks (Meyer and Schwenk 2013) exist on TLS, which fall into different categories: protocol deficiencies (key renegotiation not authenticated with previous handshake), weaknesses of cryptographic primitives (MD5, DES, RC4), and implementation issues (`goto fail`, Heartbleed).

[*] Partly funded by EPSRC grants EP/H005633/1 and EP/K008528/1

[†] Partly funded by FP7/2007-2013 under the UCN project, grant 611001

[1] Try it yourself at `https://tls.openmirage.org`

[2] `https://github.com/mirleft/ocaml-tls`

[3] We use the `cstruct` library to handle byte arrays with zero-copy, but we do not mutate these.

[4] `http://www.openbsd.org/papers/bsdcan14-libressl/mgp00026.html`

[5] both in an extended blog post (`http://openmirage.org/blog/ocaml-tls-api-internals-attacks-mitigation`) and our issue tracker (`https://github.com/mirleft/ocaml-tls/issues?labels=security+concern&page=1&state=open`)

2.2 Design Considerations

We dissect TLS into several parts:

- cryptographic operations[6]
- X.509 certificate verification[7] (ASN.1 encoding library[8])
- protocol implementation

We implemented cryptographic operations: block cipher primitives (AES, 3DES) and hash algorithms (MD5, SHA, SHA-224, SHA-256, SHA-384, SHA-512) by calling public domain C code (~4500 lines C code); RSA (with blinding) and DH (using `zarith`); and Fortuna (Ferguson and Schneier 2003, Chapter 10.3) (a pseudo random number generator). Our crypto library is smaller than 2000 lines of OCaml code, implementing high-level interfaces for the above crypto operations and some advanced block cipher modes (such as Galois counter mode), with ECC implementation in progress. The wrapper for C code are expressed entirely in OCaml, and all the memory management is done on the OCaml side.

API design for certificate verification has to be done carefully (Georgiev et al. 2012) to ensure that clients correctly verify certificates. Our API requires a client to provide a X.509 validator, a hostname and a port to connect to. The certificate verification takes the hostname and the list of certificates received from the server, and returns either `‘Ok` or a `‘Fail of certificate_failure`. The certificate verification code is roughly 400 lines, plus some helpers for encoding and decoding PEM etc. We currently do not support certificate revocations.

The protocol implementation itself consists of byte array parsing and unparsing (600 lines, internally using exceptions to signal failures, which are signaled in a monadic style upwards) and the handshake state machines (1000 lines together for client and server, written using a monadic style to propagate errors).

TLS is not stateless: messages must be received and sent in order, some messages are only allowed when specific key exchange methods are negotiated, and - especially during the `handshake` phase - some incoming messages elicit more messages to be sent. This poses an architectural problem: how should the accumulated state and reactivity be represented in a pure context?

The TLS implementation in HOL (Lochbihler and Züst 2014) solves this problem using *reactive resumptions* (Harrison and Procter 2005), a CPS-like monadic abstraction that interleaves the pure TLS computation with an effectful one (used for network IO), largely storing the internal protocol state as local variables the continuation is closed over. We opt for a much simpler approach - our main handler function is typed:

```
handle : state * bytes → state * bytes * bytes.
```

This function maps state and the input into the new state, possible output and possible application-level data decoded from the input. The `state` type explicitly encapsulates all the state accumulated during the TLS session.

To actually use our pure implementation we provide a `Lwt` frontend (and a `Mirage` frontend) that connect the engine to the network IO layer.

So far we are developing unit tests for the entire implementation, and use code coverage tools. The network packet parser and unparser already has 100% code coverage (using `OUnit2` for test cases and `Bisect` for code coverage).

2.3 Preliminary Performance Results

We ran our implementation on both client and server side, used 3DES CBC as the encryption algorithm and SHA-1 as the hashing algorithm; and in a second setup ran OpenSSL on the same machine with the same algorithms. We measured the time of transferring 100 MB of data after an initial handshake. The preliminary results show that our OCaml implementation is roughly five times slower than OpenSSL. Roughly 60%-70% of the time is spent in the C-level crypto primitives (3DES and SHA), and not on the OCaml side. Replacing the current primitives is likely to speed up our library significantly. On top of that, the OCaml code was largely not tuned for speed, so we expect further improvements after we reach feature completeness and start optimizing.

3. Related Work

The miTLS project (Bhargavan et al. 2013) verifies the cryptographic security of TLS using F# and F7 for specification.

Both Erlang and Haskell have a TLS library. Recently a partial TLS library was developed in Isabelle (Lochbihler and Züst 2014) with a focus on correctness. Also, a partial specification has been developed in Coq, leading to the discovery of an issue in OpenSSL when a change cipher suite packet was sent prematurely.

Parsifal[9] is a OCaml-based parsed engine. It provides a domain specific language to write binary protocols in a concise way, and generates a parser and unparser. The reason we are not using parsifal is its license (which is rather GPL than BSD) and that we wrote the parser and unparser before we became aware of parsifal.

4. Conclusion and Future Work

We have implemented a complete TLS stack in OCaml, further description is written in our blog series[10]. It can be used to communicate with a large number of services deployed on the Internet, running different TLS implementations. The code base is more than an order of magnitude smaller than OpenSSL. We support all major TLS versions (1.0, 1.1, 1.2), but not SSL version 3. Several TLS extensions are supported, such as server name indication (SNI) to enable virtual hosting, and secure renegotiation.

We are working on missing TLS features such as client authentication, elliptic curve and GCM cipher suites, as well as other frontends to our pure library.

References

K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE Security and Privacy*, 2013.

C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Security and Privacy*, 2014.

N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 047122894X.

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 *ACM CCS*, pages 38–49, 2012.

W. L. Harrison and A. Procter. Cheap (but functional) threads. *Submitted to Journal of Functional Programming*, 2005.

A. Lochbihler and M. Züst. Programming TLS in Isabelle/HOL, 2014.

C. Meyer and J. Schwenk. Lessons learned from previous ssl/tls attacks - a brief chronology of attacks and weaknesses. Cryptology ePrint Archive, Report 2013/049, 2013.

[6] `https://github.com/mirleft/ocaml-nocrypto`

[7] `https://github.com/mirleft/ocaml-x509`

[8] `https://github.com/mirleft/ocaml-asn1-combinators`

[9] `https://github.com/ANSSI-FR/parsifal`

[10] `http://openmirage.org/blog/introducing-ocaml-tls`