Not-quite-so-broken TLS 1.3 mechanised conformance checking

David Kaloper-Meršinjak
University of Cambridge

Hannes Mehnert
University of Cambridge

Abstract

We present a set of tools to aid TLS 1.3 implementors, all derived from a single TLS implementation/model. These include an automated offline TLS protocol conformance checker, a test oracle validating recorded sessions, a tool replicating recorded sessions with other implementations, and an interactive online handshake visualisation.

The conformance checker repeatedly runs a user-provided TLS implementation, attempting to establish TLS sessions with it; the checker explores the TLS parameter space to determine feature coverage of the provided implementation. The test oracle takes a recorded session between two endpoints and decides whether the session was conformant with the specification. The replication utility re-runs one side of a recorded session against another TLS implementation, and observes its behaviour. The online visualisation accepts connections from clients and presents the TLS session as an interactive sequence diagram.

All of these tools are based on our clean-slate nqsb-TLS implementation/model. It already supports TLS 1.0-1.2, and interoperates with a broad range of other TLS implementations. We are currently extending nqsb-TLS with TLS 1.3 support, and tracking the progress of the TLS 1.3 draft, adapting our implementation/model accordingly.

We invite the community to use our tools while implementing the TLS 1.3 RFC, and provide feedback on deviations in the interpretation thereof. This process enables the community to converge to a single, mechanically checkable TLS 1.3 model, as implemented by nqsb-TLS.

1 Introduction

TLS has a long history of implementation flaws [5]. These originate from the difficulties in interpreting the loose prose it is specified in; from the complexity of the protocol and the challenges in implementing it; and from the fact that there is no mechanised way of checking that implementations conform to the written specification.

The draft TLS 1.3 specification does not improve this situation. It still relies solely on the prose specification to convey the definition of the protocol, and provides no ways of mechanically checking the conformance to it. We

anticipate early implementations will continue to suffer from interoperability and conformance problems.

Building on nqsb-TLS [4], our TLS implementation/model, we started implementing the TLS 1.3 draft. Our contribution is a set of concrete tools aimed at filling the gap of automated conformance checking of TLS 1.3 implementations.

Initial usage might uncover discrepancies between nqsb-TLS and the RFC. Since nqsb-TLS is written in a high-level functional language, we are able to quickly adapt to such findings and fix discrepancies. In this way, we can work with the community on refining our interpretation of the draft standard, making nqsb-TLS the common agreed-upon executable model of TLS 1.3.

nqsb-TLS supports earlier TLS protocol versions and interoperates well with widely deployed TLS implementations. Our TLS 1.3 support is a work in progress, as the 1.3 protocol draft is still under development, and there are no available implementations to check interoperability against.

Our tools are still a work in progress, but we expect to be able to present them fully working at TRON. Some of the tools are already available for earlier versions of TLS.

nqsb-TLS is available from `https://nqsb.io`.

2 Description

One might think that a suite of static test vectors would be sufficient for TLS. Due to the stateful nature of TLS, and being a cryptographic protocol, its protocol state space is enormous and not easily exhaustively coverable with test vectors. The TLS 1.3 draft includes new features, such as version downgrade protection, certificate selection methodology, post-handshake authentication, and caching of server configuration for 0-RTT and PSK identities for resumption. These features cause the state space to grow even more. Based on our nqsb-TLS implementation/model we developed several tools, intended to aid TLS 1.3 developers to assess the conformance to the specification:

- a *dynamic conformance checker*, which automatically explores the state space of a given TLS executable;
- a *test oracle*, which validates whether a recorded TLS session conforms to the nqsb-TLS model;
- a *replication* tool, which re-runs a recorded TLS session against another implementation; and
- a *visualisation*, which renders interactive sequence diagrams of TLS sessions on a website.

Conformance checker The first tool is a dynamic testing tool. This tool is run against a stand-alone executable under test, as illustrated in Figure 1. The executable will receive a file descriptor as an argument, and must attempt to

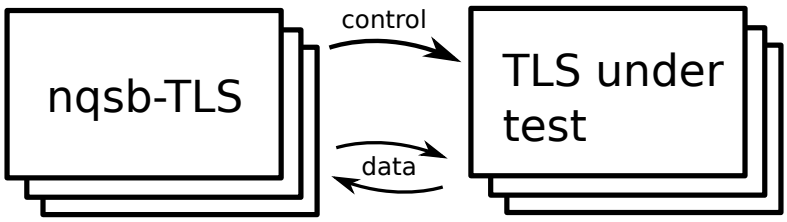

Figure 1: *Conformance checker* testing an executable. The checker can explore multiple execution paths, inducing the tested executable to fork, in order to preserve and roll-back its state.

establish a TLS session over it, acting either as client or server. The executable can do so by internally running an arbitrary TLS implementation, which does not need to be modified beforehand. The tool will run the executable under test and proceed to perform a TLS handshake with it. Once it reaches a *choice point*, a place in the protocol where an implementation's behaviour is not uniquely constrained by the standard, it generates possible responses. It will then take a snapshot of the process running the executable under test, and proceed with the TLS handshake. After completion, it will return to the choice point, restart the tested process from the snapshot, and pick the next possible response. This is repeated until the set of possible reponses is exhausted. As there can be more than one choice point in succession, the checker in effect enumerates the tree of computations generated by possibilities in the initial configuration state and subsequent underconstrained choices.

This behaviour is made possible by extending nqsb-TLS with backtracking behaviour and protocol knowledge about underconstrained features in the protocol. The executable under test is dynamically instrumented by injecting an ELF initialiser into the process during the linking phase, which allows the checker to externally cause it to fork itself at choice points.

After enumerating the space of choices, the testing tool presents sequences of choices which led to protocol failures. We can optionally visually present the failing conversations (see below).

In this way, the checker serves the role that would traditionally be served by a larger suite of (positive) test vectors. Instead of saving a large corpus of tests which an implementation can be tested against, our checker can instead explore its behaviour dynamically. This allows us to side-step the basic problem of testing a cryptographic protocol, the fact that the data observably exchanged on the wire is encrypted and different between separate executions, even if they take logically the same path. That arrangements allows us to provide the tests in a more compact form, as a single executable, and allows us to provide additional interactive features.

Test oracle The second tool is an offline test oracle, which validates recorded TLS sessions against the nqsb-TLS model. The test oracle can validate the client

endpoint, the server endpoint, or both endpoints. It has to be provided with a recorded TLS session (as the data of the underlying TCP stream), and the secrets that were used within that session. An implementation under test need to be modified to dump its per-connection secrets, such as the PEM-encoded private part of the keyshare, in order to use the test oracle. This is the *only* change needed to an implementation under test. The test oracle will re-run the (nqsb) protocol logic, committing to the same choices that the recorded implementations choose, and decide whether the conversation as a whole can be accepted as valid.

The test oracle starts by deriving its configuration from the recorded session and secrets. For each recorded received byte sequence, it executes the protocol logic, and compares the output with the recorded sent byte sequence. The comparison does not check for a byte equality, but normalises regarding fragmentation, ordering of extensions, and other varieties which do not affect the semantics. Selection of choice points, such as server/client-random, protocol version, ciphersuite, etc., is done by forward-lookup in the recorded session.

The outcome can either be that the recorded session and the test oracle agree on a result, or they disagree. A discrepancy between nqsb-TLS and the recorded session indicates an error, either in the tested TLS stack or in nqsb-TLS, or an ambiguity in what TLS actually is.

The test oracle complements the conformance checker, in a sense: while the conformance checker actively probes an implementation to decide whether its behaviour is conformant, the test oracle is the offline equivalent, capable of validating prerecorded TLS sessions after the fact.

Replication The third tool is a replication utility. It requires the same input as the test oracle: a recorded session and the secrets. The replication utility executes either the client side by connecting via a socket, or the server side by listening on a socket. It re-runs the recorded session against the implementation under test, comparing the recorded output against the received output.

This tool enables a developer to test a specific behaviour observed between two stacks with other stacks. If this spots behaviour that is only loosely specified in the RFC, the recorded session and its rendered sequence diagram can be used to clearly communicate the observed and desired behaviour to the standards committee.

The replay utility contemplates both the test oracle and the dynamic testing by executing a single recorded session against another implementation. It does not need to be a valid session in order to be replayed, thus it opens the possibility for developing a corpus of negative test sessions.

Trace visualisation The fourth tool is a sequence diagram visualisation. nqsb-TLS can internally record traces of its behaviour, and the visualisation is capable of converting those traces into sequence diagrams.

A live visualisation for earlier TLS versions is available at `https://tls.openmirage.org`, a screenshot shown in Figure 2. This runs as a stand-alone

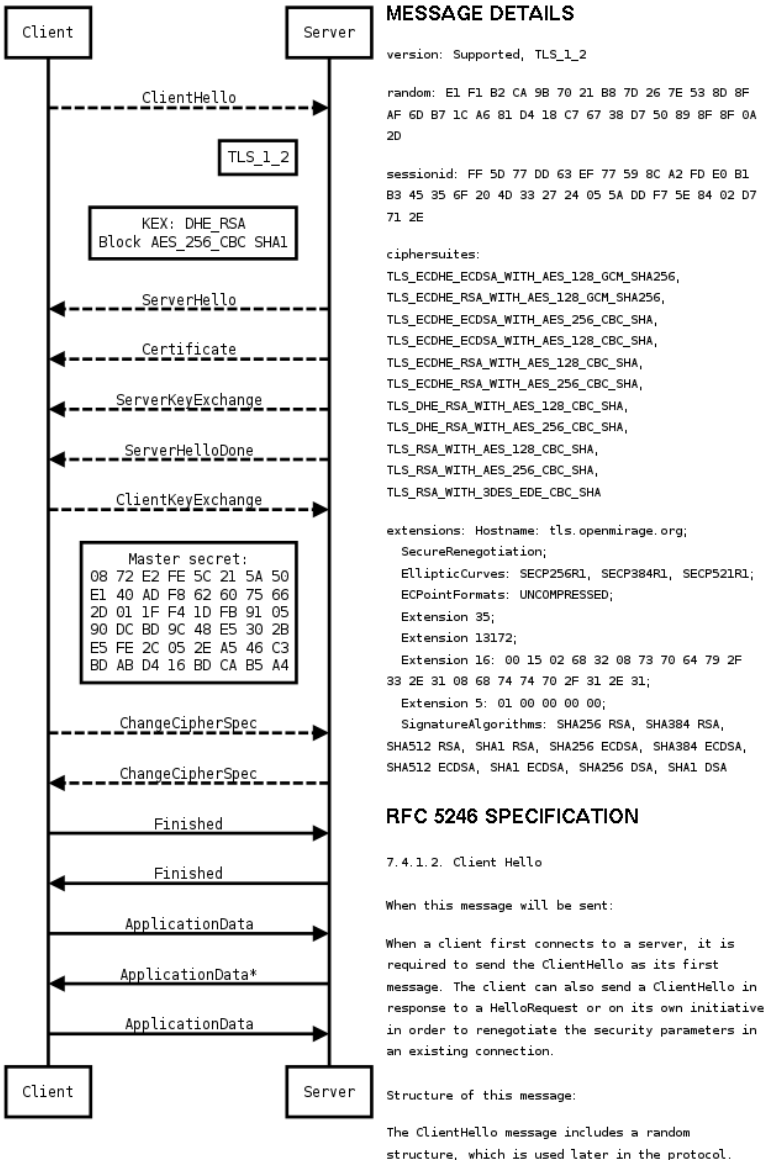

Figure 2: Screenshot of the visualisation at https://tls.openmirage.org.

server: it accepts a TLS client, records the handshake, and renders a sequence diagram of the just established TLS handshake in the web browser. Users can explore details of the messages by clicking on the messages.

We plan to develop a visualisation utility which receives a recorded TLS session and the secrets as input, and renders the sequence diagram as a pdf using LaTeX.

The trace visualisation complements the previous tools by presenting sequence diagrams of TLS sessions, which are easier to understand and follow than packet dumps.

3 Related Work

Testing of TLS implementations is done by fuzzing tools which send random input to a listening server. Recently several researchers [1, 3] investigated state machine exploration of TLS implementations. Their motivation is to find illegal state transitions which destroy the security properties of TLS. In contrast to these approaches, we provide a tool suite to automatically explore the features of a TLS implementation by covering the space of allowed transitions.

Websites such as `https://sslabs.com` test the protocol conformance and assess the security of a specific configuration. Our goal is to provide tools which aid TLS implementors, rather than tools which test a specific site configuration.

miTLS [2] is a *reference implementation* of TLS. Its primary focus is providing a formalised security proof of TLS. Instead, we aim to provide testing tools to aid the implementors of the upcoming TLS 1.3 specification. In future, we want to compare the behaviour of our nqsb-TLS model with the miTLS implementation, once a version with TLS 1.3 support is released.

4 Conclusion

TLS has long suffered from the absence of any means to mechanically check conformance to the written standard. We propose a way to bridge this gap by repurposing our model/implementation as a set of conformance testing tools, which TLS implementors can use to detect interoperability flaws early on. By discussing the sources of such discrepancies, the community can clarify the meaning of the RFC, and we will adapt nqsb-TLS if needed. Through this process, the community can help to make nqsb-TLS the agreed-upon executable reference model of TLS 1.3.

References

[1] Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., and Bhargavan, K. Flextls: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Washington, D.C., Aug. 2015), USENIX Association.

[2] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., and Strub, P.-Y. Implementing TLS with verified cryptographic security. In *Security and Privacy* (2013).

[3] de Ruiter, J., and Poll, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 193–206.

[4] Kaloper-Meršinjak, D., Mehnert, H., Madhavapeddy, A., and Sewell, P. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 223–238.

[5] Meyer, C., and Schwenk, J. Lessons learned from previous SSL/TLS Attacks - a brief chronology of attacks and weaknesses. Cryptology ePrint Archive, Report 2013/049, 2013.