Submitted to USENIX Security 2015 for double-blind review (paper #495). Please do not cite or redistribute.

# Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation

David Kaloper Meršinjak, Hannes Mehnert, Anil Madhavapeddy and Peter Sewell
*University of Cambridge Computer Laboratory*
`first.last@cl.cam.ac.uk`

## Abstract

Transport Layer Security (TLS) implementations have a history of security flaws. The immediate causes of these are often programming errors, e.g. in memory management, but the root causes are more fundamental: the challenges of interpreting the ambiguous prose specification, the complexities inherent in large APIs and code bases, inherently unsafe programming choices, and the impossibility of directly testing conformance between implementations and the specification.

We present *nqsb-TLS*, the result of our re-engineered approach to security protocol specification and implementation that addresses these root causes. The same code serves dual roles: it is both a specification of TLS, executable as a test oracle to check conformance of traces from arbitrary implementations, and a usable implementation of TLS; a modular and declarative programming style provides clean separation between its components. Many security flaws are thus excluded by construction.

nqsb-TLS can be used in standalone applications, which we demonstrate with a messaging client, and can also be compiled into a Xen unikernel (a specialised virtual machine image) with a TCB that is 4% of a standalone system running a standard Linux/OpenSSL stack, with all network traffic being handled in a memory-safe language; this supports applications including HTTPS, IMAP, Git, and Websocket clients and servers.

## 1 Introduction

Current mainstream engineering practices for specifying and implementing security protocols are not fit for purpose: as one can see from many recent compromises of sensitive services, they are not providing the security we need. Transport Layer Security (TLS) is the most widely deployed security protocol on the Internet, used for authentication and confidentiality, but a long history of exploits shows that its implementations have failed to guarantee either property. Analysis of these exploits typically focusses on their immediate causes, e.g. errors in memory management or control flow, but we believe their root causes are more fundamental:

**Error-prone languages:** historical choices of programming language and programming style that tend to lead to such errors rather than protecting against them.

**Lack of separation:** the complexities inherent in working with large code bases, exacerbated by lack of emphasis on clean separation of concerns and modularity, and by poor language support for those.

**Ambiguous and untestable specifications:** the challenges of writing and interpreting the large and ambiguous prose specifications, and the impossibility of directly testing conformance between implementations and a prose specification.

In this paper we report on an experiment in developing a practical and usable TLS stack, *nqsb-TLS*, using a new approach designed to address each of these root-cause problems. This re-engineering, of the development process and of our concrete stack, aims to build in improved security from the ground up.

We demonstrate the practicality of the result in several ways: we show on-the-wire interoperability with existing stacks; we show reasonable performance, in both bulk transfer and handshakes; and we use it as part of a standalone instant-messaging client. In addition to use in such traditional executables, nqsb-TLS is usable in applications compiled into unikernels – type-safe, single-address-space VMs with small TCBs that run directly on a hypervisor [30]. This integration into a unikernel stack lets us demonstrate a wide range of working systems, including HTTPS, IMAP, Git, and Websocket clients and servers, while sidestepping a further difficulty with radical solutions in this area: the large body of legacy code (in applications, operating systems, and libraries) that existing TLS stacks are intertwined with.

We assess the security of nqsb-TLS also in several ways: for each of the root causes above, we discuss why our approach rules out certain classes of associated flaws, with reference to an analysis of flaws found in previous TLS implementations; and we test our authentication logic using the Frankencert fuzzer [8], which found flaws in several previous implementations. We have also made the system publically available for penetration testing, as a *Bitcoin Piñata*, an example unikernel using nqsb-TLS. This has a TCB size roughly 4% of that of a similar system using OpenSSL on Linux.

We describe our overall approach in the remainder of the introduction. In §2 we analyse flaws previously found in TLS implementations. The result of applying our approach to TLS, nqsb-TLS, is presented in §3. The duality of nqsb-TLS is demonstrated in §4 and §5, describing how we use its specification to validate recorded sessions, and how we execute its implementation to provide concrete services. We evaluate the interoperability, performance, and security (§6) of nqsb-TLS, describe related work (§7), and conclude (§8).

nqsb-TLS is freely available under a BSD license [1]. For the reader unfamiliar with TLS we include a brief description of the protocol in Appendix A.

## 1.1 Approach

**A precise and testable specification for TLS** In principle, a protocol specification should unambiguously define the set of all implementation behaviour that it allows, and hence also what it does not allow: it should be *precise*. This should not be confused with the question of whether a specification is loose or tight: a precise specification might well allow a wide range of implementation behaviour. It is also highly desirable for specifications to be *executable as test oracles*: given an implementation behaviour (perhaps a trace captured from a particular execution), the specification should let one compute whether it is in the allowed set or not.

In practice, the TLS specification is neither, but rather a series of RFCs written in prose [12, 13, 14]. An explicit and precise description of the TLS state machine is lacking, as are some security-critical preconditions of its transitions, and there are ambiguities in various semi-formal grammars. There is no way such prose documents can be executed as a test oracle to directly test whether implementation behaviour conforms to the specification. TLS is not unique in this, of course, and many other specifications are expressed in the same traditional prose style, but its disadvantages are especially serious for security protocols.

For nqsb-TLS, we specify TLS as a collection of pure functions over abstract datatypes. By avoiding I/O and shared mutable state, these functions can be considered in isolation and each is deterministic, with errors returned as explicit values. The top-level function takes an abstract protocol state and an incoming message, and calculates the next state and any response messages. To do so, it invokes subsidiary functions to parse the message, drive the state machine, perform cryptographic operations, and construct the response. This top-level function can be executed as a trace-checker, on traces both from our implementation and from others, such as OpenSSL, to decide whether they are allowed by our specification or not. In building our specification, to resolve the RFC ambiguities, we read other implementations and tested interoperability with them; we thereby capture the practical de facto standard. Readability of specifications is also important, but it is difficult to assess and we do not want to make a strong claim here; we contrast some extracts from the RFCs, and our specification in Appendix C for the interested reader.

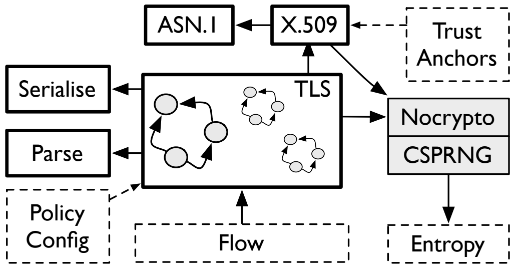

Figure 1: nqsb-TLS is broken down into strongly separated modules. The main part, in bold boxes, has pure value-passing interfaces and internals. The PRNG maintains internal state, while Nocrypto includes C code but has a pure effect-free interface.

**Reuse between specification and implementation** The same functions form the main part of our implementation, coupled with code for I/O and to provide entropy. Note that this is not an "executable specification" in the conventional sense: our specification is necessarily somewhat loose, as the server must have the freedom to choose a protocol version and cipher suite, and the trace checker must admit that, while our implementation makes particular choices.

Each version of the implementation (unix, unikernel) has a top-level Flow module that repeatedly performs I/O and invokes the pure functional core; the trace-checker has a top-level module of the same type that reads in a trace to be checked offline.

**Separation and modular structure** This focus on pure functional descriptions also enables a decomposition of the system (both implementation and specification) into strongly separated modules, with typed interfaces, that interact only by exchanging pure values, as shown in Fig. 1. These modules and their interfaces are arranged to ensure that localised concerns such as binary protocol formats, ASN.1 grammars and certificate validation are not spread throughout the stack, with no implicit dependencies via shared memory.

External resources are explicitly represented as modules, instead of being implicitly accessed, and each satisfies a module type that describes collections of operations over an abstract type, and that can be instantiated with any of several implementations. These include the `Nocrypto` cryptography layer and our PRNG, which depends on an external `Entropy` module type.

Communication with the outside world is factored out into an I/O component, `Flow`, that passes a byte sequence to the pure core, then transmits responses and handles timeouts, and is used by the top-level but not by the TLS engine itself. The pure TLS engine depends on some external data, such as the policy config and trust anchors.

**Choice of language and style** The structure we describe above could be implemented in many different programming languages, but guarantees of memory and type safety are desirable to exclude many common security flaws (lack of memory safety was the largest single source of vulnerabilities in various TLS stacks throughout 2014, as shown in our §2 vulnerability analysis), and expressive statically checked type and module systems help maintain the strongly separated structure that we outlined. Our implementation of nqsb-TLS uses OCaml, a memory-safe, statically typed programming language that compiles to fast native code with a lightweight, embeddable runtime. OCaml supports (but does not mandate) a pure programming style, and has a module system which supports large-scale abstraction via ML functors – modules that can depend on other modules' types. In OCaml, we can encode complex state machines (§3), with lightweight invariants statically enforced by the type checker (state machine problems were the second largest source of vulnerabilities). Merely using OCaml does not guarantee all the properties we need, of course (one can write imperative and convoluted code in any language); our specification and programming styles are equally important.

This is a significant departure from normal practice, in which systems software is typically written in C, but we believe our evaluation shows that it is viable in at least some compelling scenarios (§6).

**Non-goals** For nqsb-TLS we are focussed on the engineering of TLS specifications and implementations, not on the security protocol itself (as we recall in §2, some vulnerabilities have been found there). We are also not attempting to advance the state of the art in side-channel defence, though we do follow current best-practice. As we demonstrate in §6.1, nqsb-TLS can interoperate with many current TLS implementations, but we are not attempting to support legacy options or those of doubtful utility. We are also focussed on making a stack that is usable in practice and on security improvements achievable with better engineering processes, rather than trying to prove that a specification or implementation is correct or secure (see §7 for related work in that direction).

## 2 Vulnerability Analysis

In the past 13 months (January 2014 to January 2015), 54 CVE security advisories have been published for 6 widely used TLS implementations: 22 for OpenSSL, 6 for GnuTLS, 7 for NSS, 2 for SChannel, 2 for Secure Transport, 5 for JSSE, and 10 related to errors in their usage in the client software (these counts exclude vulnerabilities related to DTLS – TLS over UDP). Table 4 in the appendix details our full CVE classification.

These vulnerabilities have a wide range of causes. We classify them into broad families below, identifying root causes for each and discussing how nqsb-TLS avoids flaws of each kind.

**General memory safety violations** Most of these bugs, 15 in total, are memory safety issues: out-of-bounds reads, out-of-bounds writes and `NULL` pointer dereferences. A large group has only been demonstrated to crash the hosting process, ending in denial-of-service, but some lead to disclosure of sensitive information.

A now-notorious example of this class of bugs is Heartbleed in OpenSSL (CVE-2014-0160). Upon receiving a heartbeat record, a TLS endpoint should respond by sending back the payload of the record. The record contains the payload and its length. In Heartbleed, the TLS implementation did not check if the length of the received heartbeat matched the length encoded in the record, and responded by sending back as many bytes as were requested on the record level. This resulted in an out-of-bounds read, which lets a malicious client discover parts of server's memory. In April 2014, Cloudflare posed a challenge of exploiting this bug to compromise the private RSA key, which has been accomplished by at least four independent researchers.

nqsb-TLS avoids this class of issues entirely by the choice of a programming language with automated memory management and memory safety guarantees and by our pure-functional programming style: in the language, array bounds are checked automatically and it is not possible to access raw memory; and our style rules out reuse of mutable buffers.

**Certificate parsing** TLS implementations need to parse ASN.1, primarily for decoding X.509 certificates. While ASN.1 is a large and fairly complex standard, for the purposes of TLS, it is sufficient to implement one of its encodings (DER), and only some of the primitives. Some TLS implementations contain an ad-hoc ASN.1 parser, combining the core ASN.1 parsing task with the definitions of ASN.1 grammars, and this code operates as a part of certificate validation.

Unsurprisingly, ASN.1 parsing is a recurrent source of vulnerabilities in TLS and related software, dating back at least to 2004 (MS04-007, a remote code execution vulnerability), and up to 3 vulnerabilities in 2014 (CVE-2014-8275, CVE-2014-1568 and CVE-2014-1569). Two examples are CVE-2015-1182, the use of uninitialized memory during parsing in PolarSSL, which could lead to remote code execution, and CVE-2014-1568, a case of insufficiently selective parsing in NSS, which allowed the attacker to construct a fake signed certificate from a large space of byte sequences erroneously interpreted as the same certificate.

This class of errors is due to ambiguity in the specification, and ad-hoc parsers in most TLS implementations. nqsb-TLS avoids this class of issues entirely by separating parsing from the grammar description (§3.4).

**Certificate validation** Closely related to issues of ASN.1 parsing are the issues of certificate validation. X.509 certificates are nested data structures standardised in three versions and with various optional extensions, so validation involves parsing, traversing, and extracting information from complex compound data. This opens up the potential for errors both in the control-flow logic of this task and in the interpretation of certificates (multiple GnuTLS vulnerabilities are related to lax interpretation of the structures).

In 2014, there were 5 issues related to certificate validation. A prominent example in the control-flow logic is GnuTLS (CVE-2014-0092), where a misplaced `goto` statement lead to certificate validation being skipped if any of the intermediate certificates was an X.509 version 1 certificate.

Many implementations interleave the complicated X.509 certificate validation with parsing the ASN.1 grammar, leading to a complex control flow with subtle call chains. This illustrates another way in which the choices of programming language and style can lead to errors: the normal C idiom for error handling uses `goto` and negative return values, while in nqsb-TLS we return errors explicitly as values and have to handle all possible variants. The language typechecker and pattern-match exhaustiveness checking ensures at compile time that we have done this (§3.3).

**State machine errors** TLS consists of several sub-protocols that are multiplexed at the record level: *(i)* the handshake that initially establishes the secure connection and subsequently renegotiates it; *(ii)* alerts that signal out-of-band conditions; *(iii)* cipher spec activation notifications; *(iv)* heartbeats; and *(v)* application data. The majority of the TLS protocol specification covers the handshake state machine. The path to a successful negotiation is determined during the handshake and depends on the cipher-suite, protocol version, negotiated options, and configuration, such as client authentication. Errors in the handshake logic often lead to a security breach, allowing attackers to perform active man-in-the-middle (MITM) insertion, or to passively gain knowledge over the negotiated security parameters.

There were 10 vulnerabilities in this class. Some led to denial-of-service conditions caused (for example) by `NULL`-pointer dereferences on receipt of an unexpected message, while others lead to a breakdown of the TLS security guarantees.

A prominent example is Apple's "goto fail" (CVE-2014-1266), caused by a repetition of a `goto` statement targeting the cleanup block of the procedure responsible for verifying the digital signature of the `ServerKeyExchange` message. This caused the procedure to skip the subsequent logic and return the value registered in the output variable. As this variable was initialised to "success", the signature was never verified.

Another typical example is the CCS Injection in OpenSSL (CVE-2014-0626). `ChangeCipherSpec` is the message signalling that the just negotiated security parameters are activated. In the TLS state machine, it is legal only as the penultimate message in the handshake sequence. However, several implementations allowed a CCS message before the actual key exchange took place, which activated predictable initial security parameters. A MITM attacker can exploit this by sending a CCS during handshake, causing two parties to establish a deterministic session key and defeating encryption. Two implementations were independently vulnerable to this attack: OpenSSL (CVE-2014-0224) and JSSE (CVE-2014-6593).

Some of these errors are due to missing preconditions of state machine transitions in the specification. In nqsb-TLS, our code structure (§3.1) makes the need to consider each of these clear. We encode the state machine explicitly, while state transitions default to failure.

**Protocol bugs** In 2014, two separate issues in the protocol itself were described: POODLE and triple handshakes. POODLE is an attack on SSL version 3. SSL version 3 does not specify the value of padding bytes in CBC mode. Triple handshake [4] is a MITM attack where one negotiates sessions with the same security parameters and resumes. We do not claim to prevent nor solve those protocol bugs in nqsb-TLS, as we have not yet implemented session resumption. Furthermore, we focus on a modern subset of the protocol, not including SSL version 3, so neither attack is applicable.

**Timing side-channel leaks** Two vulnerabilities were related to timing side-channel leaks, where the observable duration of cryptographic operations depended on cryptographic secrets. These were implementation issues, related to the use of variable-duration arithmetic operations. The PKCS1.5 padding of the premaster secret is transmitted during an RSA key exchange. If the unpadding fails, there is computationally no need to decrypt the received secret material. But omitting this step leaks the information on whether the padding was correct through the time signature, and this can be used to obtain the secret. A similar issue was discovered in 2014 in various TLS implementations [32].

nqsb-TLS mitigates this attack by producing the fake PreMasterSecret upfront and always computing the RSA operation. To mitigate timing side-channels, which a memory managed programming language might further expose, we use C implementations of the low level primitives (see §3.2).

**Usage of the libraries** Of the examined bugs, 10 were not in TLS implementations themselves, but in the way the client software used them. These included the high-profile anonymization software Tor [15], the instant messenger Pidgin and the widely used multi-protocol data transfer tool cURL.

TLS libraries typically have complicated APIs due to implementing a protocol with a large parameter space. For example, OpenSSL 1.0.2 documents 243 symbols in its protocol alone, not counting the cryptographic parts of the API. Parts of its API are used by registering callbacks with the library that get invoked upon certain events. A well-documented example of the difficulty in correctly using these APIs is the OpenSSL certificate validation callback. The library does not implement the full logic that is commonly needed (it omits name validation), so the client needs to construct a function to perform certificate validation using a mix of custom code and calls to OpenSSL, and supply it to the library. This step is a common pitfall: a recent survey [23] showed that it is common for OpenSSL clients in the wild to do this incorrectly. We counted 6 individual advisories stemming from improper usage of certificate validation API, which is a large number given that improper certificate validation undermines the authentication property of TLS and completely undermines its security.

The root cause of this error class is the large and complex legacy APIs of contemporary TLS stacks. nqsb-TLS does not mirror those APIs, but provides a minimal API with strict validation by default. This small API is sufficient for the various applications we developed. OpenBSD uses a similar approach with their `libtls` API.

## 3 The *nqsb-TLS* stack

We now describe how we structure and develop the nqsb-TLS stack, following the approach outlined in the introduction to avoid a range of security pitfalls.

### 3.1 TLS Core

The heart of our TLS stack is the core protocol implementation. By using pure, composable functions to express the protocol handling, we deal with TLS as a data-transformation pipeline, independent of how the data is obtained or transmitted.

Accordingly, our core revolves around two functions. One (`handle_tls`) takes the sequence of bytes seen on the wire and a value denoting the previous state, and produces, as new values, the bytes to reply with or to transfer to the application, and the subsequent state. Our `state` type encapsulates all the information about a TLS session in progress, including the state of the handshake, the cryptographic state for both directions of communication, and the incomplete frames previously received, as an immutable value. The other one (`send_application_data`) takes a sequence of bytes that the application wishes to send and the previous state, and produces the sequence ready to be sent and the subsequent state. Coupled with a few operations to extract session information from the state, these form the entire interface to the core protocol implementation.

Below the entry points, we segment the records, decrypt and authenticate them, and dispatch to the appropriate protocol handler. One of the places where OCaml helps most prominently is in handling of the combined state machine of handshake and its interdependent subprotocols. We use algebraic data types to encode each possible handshake state as a distinct type variant, that symbolically denotes the state it represents and contains all of the data accumulated so far. The overall `state` type is simply the discriminated union of these variants. Every operation that extracts information from `state` needs to scrutinize its value through a form of multi-way branching known as pattern match. This syntactic construct combines branching on the particular variant of the `state` present with extraction of components. The resulting dispatch leads to equation-like code: branches that deal with distinct states follow directly from the values representing them, process the state data locally, and remain fully independent in the sense of control flow and access to values they operate on. Finally, each separately materializes the output and subsequent state.

This construction and the explicit encoding of state-machine is central to maintaining the state-machine invariants and preserving the coherence of state representation. It is impossible to enter a branch dedicated to a particular transition without the pair of values representing the appropriate state and appropriate input message, and, as intermediate data is directly obtained from the state value, it is impossible to process it without at the same time requiring that the state-machine is in the appropriate state. It is also impossible to manufacture a state-representation ahead of time, as it needs to contain all of the relevant data.

The benefit of this encoding is most clearly seen in CCS-injection-like vulnerabilities. They depend on session parameters being stored in locations visible throughout the handshake code, which are activated on receipt of the appropriate message. In the OpenSSL case (CVE-2014-0224), the dispatch code failed to verify whether all of these locations were populated, which implies that the handshake progressed to the appropriate phase. In our case, the only way to refer to the session parameters is to deconstruct a state-value containing them, and it is impossible to create this value without having collected the appropriate session parameters.

All of core's inner workings adhere to a predictable, restricted coding style. Information is always communicated through parameters and result values. Error propagation is achieved exclusively through results, without the use of exceptions. We explicitly encode errors distinct from successful results, instead of overloading the result's domain to mean error in some parts of its range. The type checker verifies both that each code path is dealing with exactly one possibility, and – through the exhaustiveness checker – that both forms have been accounted for. The repetitive logic of testing for error results and deciding whether to propagate the error or proceed is then abstracted away in a few higher-order functions and does not re-appear throughout the code.

This approach has also proven convenient when maintaining a growing code-base: when we had to add significant new capabilities, e.g. extending the TLS version support to versions 1.1 and 1.2 or implementing client authentication, the scope of changes was localized and the effects they had on other modules were flagged by the type checker.

## 3.2 Nocrypto

TLS cryptography is provided by *Nocrypto*, a separate library we developed for that purpose. It supports basic modular public-key primitives like RSA, DSA and DH; the two most commonly used symmetric block ciphers, AES and 3DES; the most important hash functions, MD5, SHA and the SHA2 family; and an implementation of the cryptographically strong pseudorandom number generator, Fortuna [19]. One of the fundamental design decisions was to use block-level symmetric encryption cores and hash code written in C. There are several reasons for this. Firstly, symmetric encryption and hashing are the operations TLS spends most time performing, especially in bulk transfers. We wanted to be able to support high throughput rates, so we did not want to require such performance-critical code to be in the high-level, garbage-collected environment, and we wanted to retain optimizations that C compilers can perform at such a low level. We make use of public domain widely used C cores for these operations.

Secondly, the security impact of writing cryptography in a garbage-collected environment is unclear. Performing computations over secret material in a garbage collected environment is an unexplored potential attack vector. Given uniform allocations, any allocation pattern that depends on the secret material exhibits a probability of triggering a garbage collection cycle that is dependent on the secret data. Therefore, over a number or runs, the garbage collector strongly amplifies any timing signals that could lead to leakage of secret material. We sidestep this issue by confining the symmetric cipher and hash code, which deals with secret material, below the level of managed memory, and expose it to the OCaml runtime via opaque values.

Such treatment creates a potential safety problem in turn: even if we manage to prevent certain classes of bugs in OCaml, they could occur in our C code. Our strategy to contain this is to restrict the scope of C code: we employ simple control flow and never manage memory in the C layer. AES code, for example, has only two branches, deciding whether enough rounds have been performed for the key size, and contains no loops, while the SHA code has a single loop driving the compression function over the input buffer. C functions receive preallocated fixed-size buffers, tracked by the runtime, from the OCaml side, and write their results there. To further enhance safety on the OCaml-C boundary, instead of writing the foreign-function interface code directly, we use the `OCaml-CTypes` library to describe the interface in pure OCaml; this provides us with the appropriate invocation code and automatic marshalling [33].

More complex cryptographic constructions, like cipher modes (CBC, CTR, GCM and CCM) and HMAC are implemented in OCaml on top of C-level primitives. We benefit from OCaml's safety and expressive power in these more complex parts of the code, but at the same time preserve the property that secret material is not directly exposed to the managed runtime.

Public key cryptography is treated differently. It is not block-oriented and is not easily expressed in straight-line code, while the numeric operations it relies on are less amenable to C-level optimization. At the same time, there are known techniques for mitigating timing leaks at the algorithmic level [27], unlike in the symmetric case. We therefore implement these directly in OCaml using

GMP as our bignum backend and employ the standard blinding countermeasures to compensate both for timing problems inherent in these primitives and the fact that the secrets are exposed to the OCaml runtime.

Our Fortuna CSPRNG uses AES-CTR with a self-rekeying regime and a system of entropy accumulators. Instead of entropy estimation, it employs exponential lagging of accumulators, a scheme that has been shown to asymptotically optimally recover from state compromise under a constant input of entropy of unknown quality [16]. To retain purity of the system and facilitate deterministic runs, entropy itself is required from the system as an external service, as shown later in §5.

For the sake of reducing complexity in the upper layers, the API of *Nocrypto* is concise and retains the applicative style, mapping inputs to outputs. We did make two concessions to further simplify it, however: first, if it is possible for users of the API to ensure well-formedness of the input ahead of time, the library signals malformedness via OCaml exceptions. Secondly, we employ a global and changing RNG state, because operations involving it are pervasive throughout interactions with the library and the style of explicit passing would complicate the dependent code.

## 3.3 X.509

X.509 certificates are rich tree-like data structures whose semantics changes with the presence of several optional extensions. Although the core of the path-validation process is checking of the signature, a cryptographic operation, the correct validation required by the standard includes extensive checking of the entire data structure.

For example, each extension must be present at most once, the key usage extension can further constrain which exact operations a certificate is authorized for, and a certificate can specify the maximal chain length which is allowed to follow. There are several ways in which a certificate can express its own identity and the identity of its signing certificate. After parsing, a correct validation procedure must take all these possibilities into account.

The ground encoding of certificates again benefits from algebraic data types, as the control flow of functions that navigate this structure is directed by the type-checker. On a level above, we separate the validation process into a series of functions computing individual predicates, such as the certificate being self-signed, its validity period matching the provided time or conformance of the present extensions to the certificate version. The conjunction of these is clearly grouped into single top-level functions validating certificates in different roles, which describe the high-level constraints we impose upon the certificates. The entire validation logic amounts to 314 lines of easily reviewable code.

This is in contrast to 7 000 lines of text in the RFC [9], which go into detail to explain extensions – such as policies and name constraints – that are rarely seen in the wild. For the typical HTTPS setting, the RFC fails to clarify how to search for a trust anchor, and assumes instead the presence of exactly one. Due to cross signing there can be multiple chains with different properties which are not covered by the RFC.

nqsb-TLS initially strictly followed the RFC, but was not able to validate many HTTPS endpoints on the Internet. It currently follows the RFC augmented with Mozilla's guidelines and provides a self-contained condensation of these which can be used to clarify, or even supplant, the specification. We created an extensive test suite with full code coverage, and the code has been evaluated with the Frankencert tool (see §6.2).

The interface to this logic is deterministic (it is made so by requiring the current time as an input). Our X.509 library provides operations to construct a full *authenticator*, by combining the validation logic with the current time at the moment of construction, which the TLS core can be parameterized with. We do not leave validation to the user of the library, unlike other TLS libraries [23]. Instead, we have full implementations of path validation with name checking [39] and fingerprint-based validation, and we use the type system to force the user to instantiate one of them and provide it to the TLS layer.

## 3.4 ASN.1

ASN.1 parsing creates a tension in TLS implementations: TLS critically relies on ASN.1, but it requires only a subset of DER encoding, and, since certificates are usually pre-generated, needs very little in the way of writing. For the purposes of TLS, it is therefore sufficient to implement just a partial parser.

When implementing ASN.1, a decision has to be made on how to encode the actual abstract grammar that will drive the parsing process, given by various TLS and X.509-related standards. OpenSSL, PolarSSL, JSSE and others, with the notable exception of GnuTLS, do not make any attempts to separate the grammar definition from the parsing process. The leaf rules of ASN.1 are implemented as subroutines, which are exercised in the order required by the grammar in every routine that acts as parser. In other words, they implement the parsers as ad-hoc procedures that interleave the code that performs the actual parsing with the encoding of the grammar to be parsed. Therefore the code that describes the high-level structure of data also contains details of invocation of low-lever parsers and, in the case of C, memory management. Unsurprisingly, ASN.1 parsers provide a steady stream of exploits in popular TLS implementations.

We retain the full separation of the abstract syntax representation from the parsing code, avoiding the complexity of the code that fuses the two. At the same time, we avoid parser generators which output source code that is hard to understand.

Instead, we created a library for declaratively describing ASN.1 grammars in OCaml, using a functional technique known as combinatory parsing [20]. It exposes an opaque data type that describes ASN.1 grammar instances and provides a set of constants (corresponding to terminals) and functions over them (corresponding to productions). Nested applications of these functions to create data that describes ASN.1 grammars follow the shape of the actual ASN.1 grammar definitions. Internally, this tree-like type is traversed at initialization-time to construct the parsing and serialization functions.

This approach allows us to create "grammar expressions" which encode ASN.1 grammars, and derive parsers and serializers. As the ASN.1-language we create is a fragment of OCaml, we retain all the benefits of its static type checking. Types of functions over grammar representations correspond to restrictions in the production rules, so type-checking grammar expressions amounts to checking their well-formedness without writing a separate parser for the grammar formalism. Moreover, type inference automatically derives the OCaml-level representation of the types defined by ASN.1 grammars.

Such an approach also makes testing much easier. The grammar type is traversed to generate random inhabitants of the particular grammar, which can be serialized and parsed back to check that the two directions match in their interpretation of the underlying ASN.1 encoding and to exercise all of the code paths in both.

A derived parsing function does not interpret the grammar data, but as its connections to component parsing functions are known only when synthesis takes place at run-time, we do not retain the benefit of inlining and inter-function optimization a truly compiled parser would have. Nonetheless, given that it parses roughly 50 000 certificates per second, this approach does not create a major performance bottleneck. The result is a significant reduction in code complexity: the actual ASN.1 parsing logic amounts to 620 lines of OCaml, and the ASN.1 grammar code for X.509 certificates and signatures is around 1 000 lines long. For comparison, PolarSSL 1.3.7 has around 7,500 lines of ASN.1 parsing code, while OpenSSL 1.0.1h has around 25 000 in its core ASN.1 parser alone.

## 4 Using nqsb-TLS as a test oracle

One use of nqsb-TLS is as an *executable test oracle*, an application which reads a recorded TLS session trace and checks whether it (together with some configuration

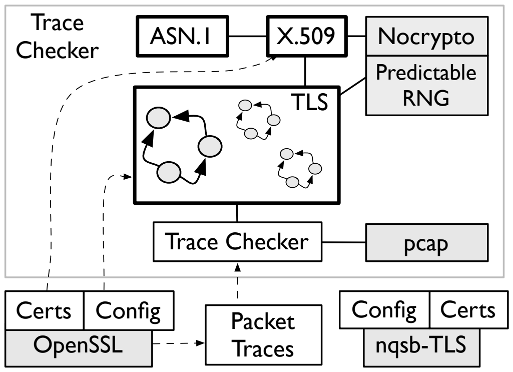

Figure 2: nqsb-TLS acts as a trace checker: the RNG is predictable, configuration and certificates are inputs, driven by packet traces from OpenSSL (or other stacks).

information) adheres to the specification that nqsb-TLS embodies. This recorded session can be a packet capture (using tcpdump) of a TLS session between various implementations (e.g. OpenSSL, PolarSSL, nqsb-TLS), or, for basic testing, a trace generated by nqsb-TLS itself.

To do this we must deal with the looseness of the TLS specification: a TLS client chooses its random nonce, set of ciphersuites, protocol version, and handshake extensions, while a TLS server picks its random nonce, the protocol version, the ciphersuite, possibly the DH group, and possibly extensions. Our test oracle does not make those decisions, but rather takes and checks the parameters recorded in the given session. To make this possible, given the on-the-wire encryption, some configuration information has to be provided to the trace checker, including private key material. In addition, both records and sub-protocols can be arbitrary fragmented; our test oracle normalises the records to not contain any fragmentation for comparison.

Figure 2 shows how nqsb-TLS can be used to build such a test oracle (note that it does not instantiate the entropy source for this usage). The test oracle produces its initial protocol state from the given session. It calculates `handle_tls` with its state and the record input of the given session, together with the particular selection of protocol version, etc., resulting in an output state, potentially an output record, and potentially decrypted application data. It then compares equality of the output record and the given session. If successful, it uses the output state and next recorded input of the given session to evaluate `handle_tls` again, and repeats to the end of the trace. It thus terminates either when the entire trace has been accepted, which means success; or with a discrepancy between the nqsb-TLS specification and the recorded session, which means failure and needs further investigation. Such a discrepancy might indicate an error in the TLS stack being tested, an error in the nqsb-TLS

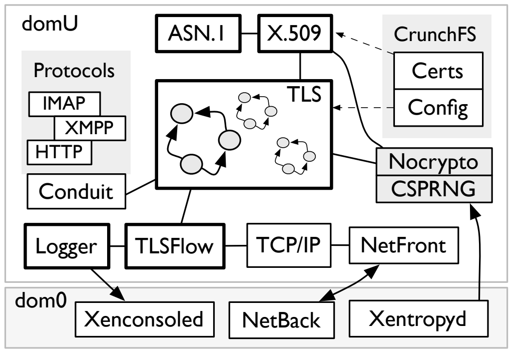

Figure 3: nqsb-TLS as a unikernel domU VM on Xen: a dom0 Xentropyd proxies host entropy, config and certificates are compiled in, various protocols run over TLS.

specification, or an ambiguity in what TLS actually is.

A first test of this infrastructure was to use a recorded session of the change cipher suite injection (CVE-2014-0224): our test oracle correctly denied this session, identifying an unexpected message. We ran our test oracle and validated some of our interoperability traces (see §6.1), and also recorded TLS sessions between various implementations (OpenSSL, PolarSSL, nqsb-TLS) using tcpdump, and validated those.

While running the test oracle we discovered interestingly varied choices in fragmentation and combination of messages among existing stacks, which may be useful in fingerprinting.

The test oracle opens up the prospect of extensive testing of the behaviour of different TLS implementations, especially if combined with automated test generation.

## 5 Using nqsb-TLS in applications

Another use of nqsb-TLS is as a TLS implementation in applications. We ported nqsb-TLS to two distinct environments and developed a series of applications, some for demonstration purposes, others for regular use.

### 5.1 Porting nqsb-TLS

To use nqsb-TLS as an executable implementation of TLS, we have to provide it with implementations of entropy and flow (see Figure 1), and a small, effectful piece of code that receives network traffic using the flow and drives the TLS core.

We pay special attention to prevent common client bugs which arise from complexity of configuring TLS stacks and correspondingly large APIs. In each instance, there is only one function to construct a TLS configuration which can be turned into an I/O interface, the function does extensive validation of the requested parameters, and the resulting configuration object is immutable. This restricts potentially error-prone interactions that configure TLS to a single API point.

**Unix** Porting nqsb-TLS to Unix was straightforward; we use the POSIX sockets API to build a flow and `/dev/urandom` as the entropy source. The exposed interface provides convenience functions to read certificates and private keys from files, and analogues of `listen`, `connect`, `accept`, `read`, `write`, and `close` for communication.

**MirageOS** The MirageOS variant allows nqsb-TLS to compile to a unikernel VM (see Figure 3). It uses the MirageOS OCaml TCP/IP library to provide the I/O flow, which is in turn coupled to Xen device drivers that communicate with the backend physical network device via shared memory rings [40]. The logger outputs directly to the VM console, and the certificates and the secret keys are compiled into OCaml data structures at build time and become part of the VM image. A key challenge when running in a virtualised environment is providing a suitable entropy source [17], especially in the common case of a VM having no access to physical hardware. Since specialised unikernels have very deterministic boot-sequences that make sources of entropy even scarcer, we had to extend MirageOS and Xen to avoid cryptographic weaknesses [25].

One way in which we solve this is by relying on dom0 to provide cross-domain entropy injection. We developed *Xentropyd*, a dom0 daemon which reads bytes from `/dev/urandom` and makes them available to VMs via an inter-VM shared memory channel. The entropy device is plugged in as a standard Xen device driver via Xenstore [22], and MirageOS has a frontend library that periodically injects entropy into the nqsb-TLS CSPRNG.

To avoid being fully reliant on dom0, we implement additional entropy harvesting within the unikernel itself. We do this by trapping the MirageOS event loop and using `RDTSCP` instruction to read the Time Stamp Counter (TSC) register on each external event. This provides us with the unpredictability inherent in the ambient events. This source is augmented with readings from the CPU RNG where available: we feed the results of `RDSEED` (or `RDRAND`) instruction into the entropy pool on each event.

To make the RNG more resilient, we do extra entropy harvesting at boot time. Following Whirlwind RNG [17], we employ a timing loop early in the boot phase, designed to take advantage of nondeterminism inherent in the CPU by way of internal races in the CPU state. This provides an initial entropy boost in the absence of *Xentropyd* and helps mitigate resumption-based attacks [17].

In an ideal scenario the unikernel entropy would be provided through both mechanisms, but we expect the usage to rely on one or the other, depending on deployment: on an ARM board lacking high-resolution timing and CPU RNG, the user is likely to have control over the hypervisor and be able to install *Xentropyd*. Conversely, in commercial hosting scenarios where the assistance of dom0 might not be available but the extra CPU features are, we expect the user to rely on the internal entropy harvesting.

## 5.2 Applications

An example application using the Unix interface is the terminal-based instant messaging client *jackline* using XMPP. The XMPP protocol negotiates features, such as TLS, over a plaintext TCP connection. Jackline performs an upgrade to TLS via the `STARTTLS` mechanism before authentication credentials are exchanged. The Unix port of nqsb-TLS contains an API that supports upgrading an already established TCP connection to TLS. Jackline uses both authentication APIs (either the path or the fingerprint validation) depending on user configuration.

The OCaml Conduit library (also illustrated in Fig. 3) supports communication transports that include TCP, inter-VM shared memory rings. It provides a high-level API that maps URIs into specific transport mechanisms. We added nqsb-TLS support to Conduit so that any application that links to it can choose between the use of nqsb-TLS or OpenSSL, depending on an environment variable. As of February 2014, 42 different libraries (both client and server) use Conduit and its provided API and can thus indirectly use nqsb-TLS for secure connections. The OPAM package manager uses nqsb-TLS as part of its mirror infrastructure to fetch 2 500 distribution files, with no HTTPS-related regressions encountered.

## 6 Evaluation

We now assess the interoperability, security, and performance of nqsb-TLS.

## 6.1 Interoperability

We assess the interoperability of nqsb-TLS in several ways: testing against OpenSSL and PolarSSL on every commit; successfully connecting to most of the Fortune 500 web sites; and by running a web server.

This web server, running since mid 2014, displays the live sequence diagram of a successful TLS session established via HTTPS [3]. A user can press a button on the website which let the server initiate a renegotiation. The server configuration includes all three TLS protocol versions and eight different ciphersuites, picking a protocol version and ciphersuite at random. Roughly 30 000 traces were recorded from roughly 350 different client stacks (6230 unique User-Agents).

Of these, around 27% resulted in a connection establishment failure (full breakdown is shown in the appendix in Table 5): Our implementation is strict, and does not allow e.g. duplicated advertised ciphersuites. Also, several accesses came from automated tools which evaluate the quality of a TLS server by trying each defined ciphersuite separately.

Roughly 50% of the failed connections did not share a ciphersuite with nqsb-TLS (160 of these would now, due to enhancements in nqsb-TLS). Another 20% started with bytes which were not interpretable by nqsb-TLS. Another 4% tried to negotiate SSL version 3. 12% of connections did not contain the secure renegotiation extension, which our server requires. 5% of the failed traces were attempts to send an early change cipher suite (CVE-2014-0224). Another 2.5% contained a ciphersuite with null (iOS6), and a further 2% send an unregistered alert type.

This three-fold evaluation shows that our TLS implementation is broadly interoperable with a large number of other TLS implementations, which also indicates that we are capturing the de facto standard reasonably well.

**Specification mismatches** While evaluating nqsb-TLS we discovered several inconsistencies between the RFC and other TLS implementations:

- Apple's SecureTransport and Microsoft's SChannel deny application data records while a renegotiation is in process, while the RFC allows interleaving.
- OpenSSL (1.0.1i) accepts X.509v3 certificates which contain either DigitalSignature or KeyEncipherment as key usage extension, independent of the used ciphersuite, whereas the RFC mandates DHE to contain DigitalSignature and RSA to contain KeyEncipherment.
- Some unknown TLS implementation inserts a length field into the padding data of the padding extension [28].
- Some unknown TLS implementation sends the unspecified alert type `0x80`, using only TLS 1.1.

## 6.2 Security

We assess the security of nqsb-TLS in several ways: the discussion of the root causes of many classic vulnerabilities and how we avoid them; mitigation of other specific issues; random testing with the Frankencert [8] fuzzing tool; a public integrated system protecting a bitcoin reward; and analysis of the TCB size of that compared with a similar system built using a conventional stack.

**Avoidance of classic vulnerability root causes** In

Sections 2 and 3 we described how the nqsb-TLS structure and development process exclude the root causes of many vulnerabilities that have been found in previous TLS implementations.

**Additional mitigations** The TLS version 1.2 RFC includes a section on implementation pitfalls. This is a list of known protocol issues and of common implementation failures regarding the cryptographic operations. nqsb-TLS mitigates all of these.

Further issues that nqsb-TLS addresses include:

- Interleaving of sub-protocols, except between change of cipher spec and finished.
- Each item of TLS 1.0 application data is prepended by an empty fragment to randomize the IV.
- Secure renegotiation [37] is required.
- SCSV extension is supported.
- Best practices against attacks arising from mac-then-encrypt in CBC mode are followed (no mitigation of Lucky13 [18])

**Frankencert** Frankencert is a fuzzing tool which generates syntactically valid X.509 certificate chains by randomly mixing valid certificates and random data. We generated 10 000 X.509 certificate chains, and compared the verification result of OpenSSL (1.0.1i) and nqsb-TLS The result is that nqsb-TLS accepted 120 certificates, a strict subset of the 192 OpenSSL accepted.

Of these 72 accepted by OpenSSL but not by nqsb-TLS, 57 certificate chains contain arbitrary data in X.509v3 extensions where our implementation allows only restricted values. An example is the KeyUsage extension, which specifies a sequence of OIDs. In the RFC, 9 different OIDs are defined. Our X.509v3 grammar restricts the value of the KeyUsage extension to those 9 OIDs. 12 certificate chains included an X.509v3 extension marked critical but not supported by nqsb-TLS.

Two server certificates are certificate authority certificates. While not required by the path validation, best practices from Mozilla recommend to not accept a server certificate which can act as certificate authority. The last certificate is valid for a Diffie-Hellman key exchange, but not for RSA. Our experimental setup used RSA, thus nqsb-TLS denied the certificate appropriately.

**Exposure to new vulnerabilities** Building nqsb-TLS in a managed language potentially opens us up to vulnerabilities that would not affect stacks written in C. Algorithmic complexity attacks are a low-bandwidth class of denial-of-service attacks that exploit deficiencies in many common default datastructure implementations [10]. The modular structure of nqsb-TLS makes it easy to audit the implementations used within each component. The security of the OCaml runtime was assessed by the French computer security governmental office [35], which introduced several changes. We are also careful to avoid timing side channels due to garbage collection pauses, as described in §3.2.

| | Linux/OpenSSL | Unikernel/nqsb-TLS | |
|---|---|---|---|
| Kernel | 1600 | 48 | (36) |
| Runtime | 689 | 25 | (6) |
| Crypto | 230 | 23 | (14) |
| TLS | 41 | 6 | (0) |
| Total | 2560 | 102 | (56) |

Table 1: TCB (in kloc); portion of C code in parens

**The Bitcoin Piñata** To demonstrate the use of nqsb-TLS in an integrated system based on MirageOS, and to encourage external code-review and penetration testing, we set up a public bounty, the *Bitcoin Piñata* [2]. This is a standalone MirageOS unikernel containing the secret key to a bitcoin address, which it transmits upon establishing a successfully authenticated TLS connection. The service exposes both TLS client and server on different ports, and it is possible to bridge the traffic between the two and observe a successful handshake and the encrypted exchange of the secret.

The attack surface encompasses the entire system, from the the underlying operating system and its TCP/IP stack, to TLS and the cryptographic level. The system will only accept connections authenticated by the custom certificate authority that we set up for this purpose. Reward is public and automated, because if an attacker manages to access the private bitcoin key, they can transfer the bitcoins to an address of their choosing, which is attestable through the blockchain.

While this setup cannot prove the absence of security issues in our stack, it motivated several people to read through our code and experiment with the service. To date, there have been 16 600 failed and 6 700 successful TLS connections. Although we cannot directly verify that all successful connection resulted from the service being short-circuited to connect to itself, there have been no outgoing transactions registered in the blockchain.

**Trusted computing base** The TCB size is a rough quantification of the attack surface of a system. We assess the TCB of our Piñata, compared to a similar traditional system using Linux and OpenSSL. Both systems are executed on the same hardware and the Xen hypervisor, which we do not consider here. The TCB sizes of the two systems are shown in Table 1 (using `cloc`).

The traditional system contains the Linux kernel (without drivers and assembly code), glibc, and OpenSSL. In comparison, our Piñata uses a minimal operating system, the OCaml runtime, and several OCaml libraries. The cryptographic library depends on gmp. While the traditional system uses a C compiler, our Piñata additionally uses the OCaml compiler (roughly 40 000 lines of code).

The trusted computing base of the traditional system is 25 times larger than ours. Both systems provide the same service to the outside world and are hardly distinguishable for an external observer.

## 6.3 Performance

We evaluate the performance of nqsb-TLS, using OpenSSL version 1.0.2, known as a high-performance implementation, as baseline. We use a single machine to avoid the effects of network unpredictability and its limiting influence on throughput. The test machine has Intel i7-2660M CPU and runs Linux 3.18 and glibc 2.21. We measured throughput by running the command-line interface of OpenSSL as client, the tested implementation as server, and transferring 100 MB of data. We measured handshakes by running 20 parallel processes which continuously establish connections to the test server (at this point the servers' capacity to accept connections becomes saturated) and measuring the maximum nuber of negotiated connections during one second. As the tests have large variance, we repeated each 50 times and took the best result.

We present two separate series of tests. In the first scenario, labelled as "Same" in the tables, both programs run in the same operating system instance and communicate over the loopback network interface. To better gauge the effects of virtualization, we booted Xen 4.4 under KVM virtualization using Qemu 2.2.0 to provide virtual hardware, and performed measurements between the native Linux host and a virtual machine on top of Xen (running as dom0 in the case of OpenSSL and as domU in the case nqsb-TLS compiled into a unikernel), this second scenario being labelled as "VM".

OpenSSL has several implementations of AES capable of exploiting processor features (SSE and AES-NI), and our own cryptographic backend contains only portable code which runs on both x86 and ARM. To isolate the overheads of higher-level logic, we ran a separate series of throughput tests with OpenSSL's CPU acceleration disabled.

Throughput results are summarized in Table 2, and handshakes in Table 3. OpenSSL without CPU acceleration is 2 times faster than nqsb-TLS in the "Same" scenario. In the "VM" scenario, OpenSSL is only 1.8 times faster than nqsb-TLS, which is likely related to the single address space and absence of context switches in the MirageOS virtual machine. nqsb-TLS can sustain slightly higher handshake rates, apart from the RSA exchange case in the "VM" scenario which needs further inspection.

| | nqsb-TLS | OpenSSL | OpenSSL AES-NI |
|---|---|---|---|
| Same | 50 MB/s | 103 MB/s | 285 MB/s |
| VM | 33 MB/s | 61 MB/s | 158 MB/s |

Table 2: Throughput comparison between nqsb-TLS and OpenSSL without and with CPU acceleration, using AES256-CBC-SHA.

| | | nqsb-TLS | OpenSSL |
|---|---|---|---|
| RSA | Same | 639 hs/s | 573 hs/s |
| RSA | VM | 305 hs/s | 462 hs/s |
| DHE-RSA | Same | 374 hs/s | 360 hs/s |
| DHE-RSA | VM | 305 hs/s | 270 hs/s |

Table 3: Handshake performance of nqsb-TLS and OpenSSL

Due to effects of virtualization, networking overhead, and – in the case of handshake measurements – concurrency, these numbers should be taken only as a rough estimate of relative performance. The overall picture is that, while nqsb-TLS is slower than OpenSSL in sustained throughput, when used across the internet its speed would not be the limiting factor.

Furthermore, the nqsb-TLS process spends 79% of the running time in the cryptographic backend, with 55% of the running time in the C layer. Performance can be improved by employing CPU acceleration in the C core, and using lower-level structures in the OCaml code that implements block cipher modes, but we defer such optimizations to future work.

## 7 Related Work

**Security proofs** Several research groups [34, 21, 26, 24, 11, 36] have modelled and formally verified security properties of TLS. Because TLS is a complex protocol, most models use a simplified core, and formalising even these subsets is challenging work which is not very accessible to an implementor audience. Additionally, these models need to be validated with actual implementations, to relate to the de facto standard, but this is rarely done (to do so, some kind of trace checker or executable needs to be developed). Some of these models formalised the handshake protocol, but omitted renegotiation, in which a security flaw was present until discovered in 2009.

A noteworthy exception is miTLS [5], developed in F7 with the possibility to extract executable F# code. It is both a formalization of the TLS protocol and a runnable implementation. This formalization allowed its developers to discover two protocol-level issues: alert fragmentation and the triple handshake. As an implementation, it depends on the Common Language Runtime for execution, and uses its services for cryptographic operations and X.509 treatment (including ASN.1 parsing). In contrast, nqsb-TLS cannot be used for verifying security properties of TLS, but provides a runnable implementation and a test oracle. It also compiles to native code and implements the entire stack from scratch, making it self-contained. It can be used e.g. in MirageOS, which only provides the bare TCP/IP interfaces and has no POSIX-like layer or cryptographic services. There are differences in feature coverage too: miTLS supports session resumption, while nqsb-TLS provides AEAD ciphersuites and supports Server Name Indication.

**Language-oriented approach to security** Mettler et al. propose Joe-E [31], a subset of Java designed to support the development of secure software systems. They strongly argue in favor of a particular programming style to facilitate ease of security reviews.

Our approach shares some of the ideas in Joe-E. By disallowing mutable static fields of Java classes, they effectively prohibit globally visible mutable state and enforce explicit propagation of references to objects, which serve as object capabilities. They also emphasize immutability to restrict the flows of data and achieve better modularity and separation of concerns.

The difference in our approach is that we use immutability and explicit data-passing not only on the module (or class) boundaries but pervasively throughout the code, aiming to facilitate code-review and reasoning on all levels. A further difference is that Joe-E focuses the proposed changes in style on security reviews only, aiming to help the reader of the code ascertain that the code does not contain unforeseen interactions and faithfully implements the desired logic. In contrast, we employ a fully declarative style. Our goals go beyond code review, as large portions of our implementation are accessible as a clarification to the specification, and we have an executable test oracle.

Finally, there is the difference between host languages. Java lacks some of the features we found to be most significant in simplifying the implementation, chiefly the ability to encode deeply nested data structures and traverse them via pattern-matching, and to express local operations in a pure fashion.

**TLS implementations in high-level languages** Several high-level languages contain their own TLS stack. Oracle Java ships with JSEE, a memory-safe implementation. However its overall structure closely resembles the C implementations. For example, the state machine is built around accumulating state by mutations of shared memory locations, the parsing and validation of certificates are not clearly separated, and the certificate validation logic includes non-trivial control flow. This resulted in high-level vulnerabilities similar in nature to the ones found in C implementations, such as CCS Injection (CVE-2014-0626), and its unmanaged exception system led to several further vulnerabilities [32].

There are at least two more TLS implementations in functional languages, one in Isabelle [29] and one in Haskell. Interestingly, both implementations experiment with their respective languages' expressivity to give the implementations an essentially imperative formulation. The Isabelle development uses a coroutine-like monad to directly interleave I/O operations with the TLS processing, while the Haskell development uses a monad stack to both interleave I/O and to implicitly propagate the session state through the code. In this way they both lose the clear description of data-dependencies and strong separation of layers nqsb-TLS has.

**Protocol specification and testing** There is an extensive literature on protocol specification and testing in general (not tied to a security context). We build in particular on ideas from Bishop et al.'s work on TCP [6, 38], in which they developed a precise specification for TCP and the Sockets API in a form that could be used as a trace-checker, characterising the de facto standard. TCP has a great deal of internal nondeterminism, and so Bishop et al. resorted to a general-purpose higher-order logic for their specification and symbolic evaluation over that for their trace-checker. In contrast, the internal nondeterminism needed for TLS can be bounded as we describe in §4, and so we have been able to use simple pure functional programming, and to arrange the specification so that it is simultaneously usable as an implementation. We differ also in focussing on an on-the-wire specification rather than the endpoint-behaviour or end-to-end API behaviour specifications of that work. In contrast to TCP with its Sockets API, TLS does not specify any API.

## 8 Conclusion

We have described an experiment in engineering critical security-protocol software using what may be perceived as a radical approach. We focus throughout on structuring the system into modules and pure functions that can each be understood in isolation, serving dual roles as test-oracle specification and as implementation, rather than traditional prose specifications and code driven entirely by implementation concerns.

Our evaluation suggests that it is a successful experiment: nqsb-TLS is usable in multiple contexts, as test oracle and in Unix and Unikernel applications, it has reasonable performance, and it is a very concise body of code. Our security assessment suggests that, while it is by no means guaranteed secure, it does not suffer from several classes of flaws that have been important in previous TLS implementations. In this sense, it is at least not quite so broken as some secure software has been.

In turn, this indicates that our *approach* has value. As further evidence of that, we applied the same approach to the *off-the-record* [7] security protocol, used for end-to-end encryption in instant messaging protocols. We engineered a usable implementation and reported several inconsistencies in the prose specification. The XMPP client mentioned earlier uses nqsb-TLS for transport layer encryption, and our OTR implementation for end-to-end encryption.

The approach cannot be applied everywhere. The two obvious limitations are (1) that we rely on a language runtime to remove the need for manual memory management, and (2) that our specification and implementation style, while precise and concise, is relatively unusual in the wider engineering community. But the benefits suggest that, where it can be applied, it will be well worth doing so.

## References

[1] `http://tinyurl.com/mr5c3bs`. (the target of this link is not anonymised).

[2] `http://tinyurl.com/msu98to`. (the target of this link is not anonymised).

[3] `http://tinyurl.com/nvg87s5`. (the target of this link is not anonymised).

[4] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., and Strub, P.-Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *IEEE Symposium on Security & Privacy (S&P 14)* (2014), p. 98–113.

[5] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., and Strub, P.-Y. Implementing TLS with Verified Cryptographic Security. In *IEEE Symposium on Security & Privacy (Oakland)* (San Francisco, United States, 2013), pp. 445–462.

[6] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: the ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review* (Aug. 2005), pp. 265–276.

[7] Borisov, N., Goldberg, I., and Brewer, E. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2004), WPES '04, ACM, pp. 77–84.

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

[9] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[10] Crosby, S. A., and Wallach, D. S. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association, pp. 3–3.

[11] Díaz, G., Cuartero, F., Valero, V., and Pelayo, F. Automatic verification of the tls handshake protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (New York, NY, USA, 2004), SAC '04, ACM, pp. 789–794.

[12] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465.

[13] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465.

[14] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465.

[15] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 21–21.

[16] Dodis, Y., Shamir, A., Stephens-Davidowitz, N., and Wichs, D. How to eat your entropy and have it too – optimal recovery strategies for compromised rngs. Cryptology ePrint Archive, Report 2014/167, 2014.

[17] Everspaugh, A., Zhai, Y., Jellinek, R., Ristenpart, T., and Swift, M. Not-so-random numbers in virtualized linux and the whirlwind rng. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 559–574.

[18] Fardan, N. J. A., and Paterson, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540.

[19] Ferguson, N., and Schneier, B. *Practical Cryptography*, 1 ed. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[20] Frost, R., and Launchbury, J. Constructing natural language interpreters in a lazy functional language. *Comput. J. 32*, 2 (Apr. 1989), 108–121.

[21] Gajek, S., Manulis, M., Pereira, O., reza Sadeghi, A., and Schwenk, J. Universally Composable Security Analysis of TLS. In *Provable Security* (2008), pp. 313–327.

[22] Gazagnaire, T., and Hanquez, V. Oxenstored: An efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2009), ICFP '09, ACM, pp. 203–214.

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

[24] He, C., Sundararajan, M., Datta, A., Derek, A., and Mitchell, J. C. A modular correctness proof of ieee 802.11i and tls. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 2–15.

[25] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 205–220.

[26] Jager, T., Kohlar, F., Schäge, S., and Schwenk, J. On the security of TLS-DHE in the standard model. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 273–293.

[27] Kocher, P. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology CRYPTO 96*, N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996, pp. 104–113.

[28] Langley, A. A TLS ClientHello padding extension, Feb. 2015.

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

[30] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 461–472.

[31] Mettler, A., Wagner, D., and Close, T. Joe-e: A security-oriented subset of java.

[32] Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., and Tews, E. Revisiting ssl/tls implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 733–748.

[33] Minsky, Y., Madhavapeddy, A., and Hickey, J. *Real World OCaml*, 1 ed. O'Reilly Associates, Nov. 2013, ch. 19.

[34] Morrissey, P., Smart, N. P., and Warinschi, B. A Modular Security Analysis of the TLS Handshake Protocol. In *International Conference on the Theory and Application of Cryptology and Information Security* (2008), pp. 55–73.

[35] nationale de la sécurité des systèmes d'information, A. Étude de la sécurité intrinséque des langages fonctionnels, 2013.

[36] Paulson, L. C. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security 2* (1999), 332–351.

[37] Rescorla, E., Ray, M., Dispensa, S., and Oskov, N. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), Feb. 2010.

[38] Ridge, T., Norrish, M., and Sewell, P. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proceedings of FM 2008: the 15th International Symposium on Formal Methods (Turku, Finland), LNCS 5014* (May 2008), pp. 294–309.

[39] Saint-Andre, P., and Hodges, J. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), Mar. 2011.

[40] Warfield, A., Hand, S., Fraser, K., and Deegan, T. Facilitating the development of soft devices. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 22–22.

## A TLS Background

TLS provides the twin features of authentication and confidentiality. Clients typically verify the server's identity, the server can optionally verify the client's identity, followed by the two endpoints establishing an encrypted communication channel. This channel should be immune from eavesdropping, tampering and message forgery.

There have been three standardized versions of TLS, 1.0, 1.1 and 1.2, while the last SSL version (3) is still seeing wide usage. A key feature of TLS is algorithmic agility: it allows the two endpoints to negotiate the key exchange method, symmetric cypher and the message authentication mode upon connecting. This triple is called a cipher suite, and between versions 1.0 and 1.2, there are around 160 cipher suites standardized and widely supported. Together with a number of standardized extensions to the protocol that can be negotiated, this creates a large possible space of session parameters. This large variation in configuration options is a marked characteristic of TLS, significantly contributing to the complexity of its state-machine.

Being a relatively large protocol, TLS has only a handful of implementations in wide use. The three major free or open-source implementations are OpenSSL, GnuTLS and Mozilla's NSS. Microsoft supplies SChannel with their operating systems, while Apple supplies Secure Transport with theirs, and Oracle Java runtime comes bundled with JSSE.

Structurally, TLS is a two-layered protocol. The outer layer, record protocol, preserves message boundaries and provides framing. It encapsulates one of five subprotocols: handshake, change cipher spec, alert, application data or heartbeat.

A TLS session is initiated by the client, which uses the handshake protocol to signals the highest protocol version, possible extensions, and a set of ciphersuites it supports. The server picks the highest protocol version it shares with the client and a mutually supported ciphersuite, or fails the handshake. The ciphersuite determines whether the server authenticates itself, and server's configuration determines whether it requires the client to do the same. After the security parameters for the authenticated encryption scheme are negotiated, the handshake can be finished by sending a Change Cipher Spec message which activates them, and finally authenticated in the last handshake message. Either party can renegotiate the session over the established channel by initiating another handshake.

The handshake protocol contains a complex state machine, which must be successfully traversed at least once. Handshake messages are independent of other protocols, but the other protocols are not independent of handshakes. For instance, it is impossible to exchange data intended for the application before a session is established, and it should be impossible to affect the use of negotiated session parameters while the negotiation is still in progress.

Server and client authentication is performed by means of X.509 certificates. When using standard path validation, after one party presents a sequence of certificates called the certificate chain, the other party needs to verify that a) each certificate in the chain is signed by the next certificate; b) the last certificate is signed by one of the trust anchors independent of connection; and c) that the first party has the knowledge of the private key associated with the first certificate in the chain. For correct authentication, the authenticating party also needs to verify general semantic well-formedness of the involved certificates, and be able to deal with three version of X.509 and a number of extensions.

X.509 certificates are described through ASN.1, a notation for describing the abstract syntax of data, and encoded using Distinguished Encoding Rules (DER), one of the several standard encodings ASN.1 defines. A particular description in the ASN.1 language coupled with a choice of encoding defines both the shape the the data-structures and their wire-level encoding. ASN.1 provides a rich language for describing structure, with a number of primitive elements, like `INTEGER` and `BIT STRING`, and combining constructs, like `SEQUENCE` (a record of sub-grammars) and `CHOICE` (a node joining alternative grammars). The ASN.1 formalism can be used with a compiler that derives parsing and serialization code for the target language, but TLS implementations more typically contain custom parsing code for dealing with X.509 certificates. As X.509 exercises much of ASN.1, this parsing layer is non-trivial and significantly adds to the implementation complexity.

## B Tables

| Count | Reason |
|---|---|
| 3984 | no shared cipher |
| 1650 | bad first five bytes |
| 916 | no secure renegotiation |
| 394 | early change cipher suite |
| 322 | ssl version 3 |
| 210 | null cipher (iOS 6) |
| 159 | other side: alert 0x80 |
| 67 | pedantic parser |
| 54 | bad record mac |
| 25 | version too high |
| 23 | other side: bad certificate |
| 15 | other side: close notify |
| 13 | other side: protocol version |
| 7 | unsolicited client certificate |
| 7 | bad padding extension |
| 6 | bad content type |
| 4 | record overflow |
| 3 | invalid secure reneg |
| 1 | trailing bytes |
| 1 | underflow |

Table 5: Breakdown of failed traces.

| Product | CVE ID | Issue source |
|---|---|---|
| OpenSSL | 2013-4353, 2015-0206, 2014-[3567, 3512, 3569, 3508, 3470, 0198, 0160] | Memory management |
| | 2015-0205, 2015-0204, 2014-3572, 2014-0224, 2014-3568, 2014-3511 | State machine |
| | 2014-8275 | Certificate parsing |
| | 2014-2234 | Certificate validation |
| | 2014-3509, 2010-5298 | Shared mutable state |
| | 2014-0076 | Timing side-channel |
| | 2014-3570 | Wrong sqrt |
| GnuTLS | 2014-8564, 2014-3465, 2014-3466 | Memory management |
| | 2014-1959, 2014-0092, 2009-5138 | Certificate validation |
| NSS | 2014-1544 | Memory management |
| | 2013-1740 | State machine |
| | 2014-1490 | Shared mutable state |
| | 2014-1569, 2014-1568 | Certificate parsing |
| | 2014-1492 | Certificate validation |
| | 2014-1491 | DH param validation |
| SChannel | 2014-6321 | Memory management |
| Secure Transport | 2014-1266 | State machine |
| JSSE | 2014-6593, 2014-0626 | State machine |
| | 2014-0625 | Memory exhaustion |
| | 2014-0411 | Timing side-channel |
| Applications | 2014-2734 | Memory management |
| | 2014-3694, 2014-0139, 2014-2522, 2014-8151, 2014-1263 | Certificate validation |
| | 2013-7373, 2014-0016, 2014-0017, 2013-7295 | RNG seeding |
| Protocol-level | 2014-1771, 2014-1295, 2014-6457 | Triple handshake |
| | 2014-3566 | POODLE |

Table 4: Vulnerabilities in TLS implementations in 2014.

## C nqsb-TLS readability

The ASN.1 grammar of a to-be-signed certificate, defined in RFC 5280 [9], is illustrated in Figure 4. Next to it in Figure 5 is the syntax as written in nqsb-TLS, which evaluates to both a parser and a writer of such a structure (DER encoding).

Another artifact is the handler of handshake packet in nqsb-TLS in Figure 6. This receives a handshake state and a byte sequence as input, and first matches whether the parser can successfully decode the byte sequence. A nested match is done on the server handshake state and the parsed data, where each specific state accepts exactly the matching data type, otherwise an unexpected message failure is signalled.

```
TBSCertificate ::= SEQUENCE  {
     version         [0]  Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
     extensions      [3]  Extensions OPTIONAL
}
```

Figure 4: ASN.1 grammar from RFC

```
let tBSCertificate =
  sequence (
      (optional (explicit 0 version))
    @ (required certificate_sn)
    @ (required Algorithm.identifier)
    @ (required Name.name)
    @ (required validity)
    @ (required Name.name)
    @ (required PK.pk_info_der)
    @ (optional (implicit 1 unique_identifier))
    @ (optional (implicit 2 unique_identifier))
   -@ (optional (explicit 3 Extension.extensions_der)) )
```

Figure 5: OCaml code for TBSCertificate

```
let handle_handshake ss hs buf =
  let open Reader in
  match parse_handshake buf with
  | Or_error.Error re -> fail (`Fatal (`ReaderError re))
  | Or_error.Ok handshake ->
       match ss, handshake with
       | AwaitClientHello, ClientHello ch ->
          answer_client_hello hs ch buf
       | AwaitClientCertificate_RSA (session, log), Certificate cs ->
          answer_client_certificate_RSA hs session cs buf log
       | AwaitClientCertificate_DHE_RSA (session, dh_sent, log), Certificate cs ->
          answer_client_certificate_DHE_RSA hs session dh_sent cs buf log
       | AwaitClientKeyExchange_RSA (session, log), ClientKeyExchange kex ->
          answer_client_key_exchange_RSA hs session kex buf log
       | AwaitClientKeyExchange_DHE_RSA (session, dh_sent, log), ClientKeyExchange kex ->
          answer_client_key_exchange_DHE_RSA hs session dh_sent kex buf log
       | AwaitClientCertificateVerify (session, sctx, cctx, log), CertificateVerify ver ->
          answer_client_certificate_verify hs session sctx cctx ver buf log
       | AwaitClientFinished (session, log), Finished fin ->
          answer_client_finished hs session fin buf log
       | Established, ClientHello ch ->
          answer_client_hello_reneg hs ch buf
       | AwaitClientHelloRenegotiate, ClientHello ch ->
          answer_client_hello_reneg hs ch buf
       | _, hs -> fail (`Fatal (`UnexpectedHandshake hs))
```

Figure 6: nqsb-TLS server handshake dispatcher