# OCaml inside: a drop-in replacement for `libtls`

Enguerrand Decorne (speaker), Jeremy Yallop, David Kaloper-Meršinjak
University of Cambridge Computer Laboratory

## Introduction: `openssl` to `libtls` to `libnqsb-tls`

The C programming language pervades systems software. An operating system in the Unix tradition consists of a kernel, written in C, and a collection of libraries and executables, also written in C, which communicate in large part via APIs defined as C types and functions. Systems built in C typically suffer from a number of problems, ranging from buffer overflows and other violations that follow inevitably from unrestricted access to memory, to awkward APIs that result from an inexpressive type system and a lack of automatic memory management.

The `openssl` library, which implements the cryptographic protocols TLS and SSL, suffers from both these problems. The lack of bounds checking in C led to the notorious Heartbleed bug in 2014; a study two years earlier found that almost no clients of `openssl` use the library correctly, apparently due to its unhelpful interface (Georgiev et al. 2012).

In response to the Heartbleed bug, the OpenBSD team created `libressl`, a fork of `openssl` with the aim of correcting the deficiencies. One early fruit of the `libressl` project is `libtls`, a much simpler, more disciplined interface to the TLS protocol implementation in `libressl`.

However, `libtls` is still built in C, and so is still vulnerable to potential buffer overflows, type errors, and similar defects. In this talk we describe one approach to avoiding these problems, namely replacing `libtls` with a fully compatible library written in OCaml. Our library, `libnqsb-tls`[1], matches the `libtls` function-for-function, but the implementation contains no C; instead, it it wraps the pure OCaml TLS implementation ocaml-tls (Kaloper-Meršinjak et al. 2015).

## Exposing OCaml to C (without writing any C)

The implementation of `libnqsb-tls` is based on two existing OCaml libraries. As described above, the ocaml-tls library forms the core of `libnqsb-tls`. The interface between OCaml and C is defined by another library, `ocaml-ctypes` (Yallop et al. 2016) (typically shortened to plain *ctypes*).

The ctypes library is widely used to expose C functions to OCaml. However, it also supports the inverse arrangement, i.e. wrapping a set of OCaml functions as a library that can be called from C, which is precisely what is needed to implement `libnqsb-tls`. Ctypes supports both regular and inverted bindings in a similar fashion: in both cases the user constructs a set of OCaml values which describe the types and functions of the cross-language interface, and then ctypes uses the description to generate code. (See Listing 1 for an example, which is expounded in more detail in the next section.) For inverted bindings, the generated code consists of a C header file (Listing 2), an OCaml source file (not shown), and a C source file (Listing 3). The header file contains declarations for the OCaml functions exposed to C — in our case, these are declarations of the functions in the `libtls` interface, which our library exports. The source files contain definitions of those functions, which mediate between the `libtls` interface and the `ocaml-tls` implementation, and which can be compiled and linked together with `ocaml-tls`, the OCaml runtime, and the OCaml code that implements `libnqsb-tls` to build a shared library.

[1] Available here: `https://github.com/mirleft/libnqsb-tls/`

```
let tls_server () =
  let tls_server =
   { error = None; config = None; fd = None;
     state = ‘NotConfigured; linger = None } in
  Root.create tls_server |> from_voidp tls

let () = I.internal "tls_server"
   (void @-> returning (ptr tls)) tls_server
```

Listing 1: Exposing an OCaml function to C

```
struct tls *tls_server(void);
```

Listing 2: Generated header file, matching libtls's header

```
struct tls *tls_server(void) {
   enum { nargs = 1 };
   CAMLparam0();
   CAMLlocalN(locals, nargs);
   locals[0] = Val_unit;
   value x239 = functions[fn_tls_server];
   value x240 = caml_callbackN(x239, nargs, locals);
   struct tls *x241 = CTYPES_ADDR_OF_FATPTR(x240);
   caml_local_roots = caml__frame;
   return x241;
}
```

Listing 3: Generated C code

## Replicating `libtls`: challenges and techniques

Exposing a C interface to an OCaml library involves several challenges, including converting between OCaml and C views of data, harmonising the different styles of memory management, and bridging two programming styles.

***Converting between OCaml and C values*** Values in OCaml are represented as tagged blocks, described by a rich type system with support for parameterised and abstract types. Values in C are represented as flat blocks, described by a simple type system which closely corresponds to the concrete representation of values.

The ctypes library exposes a set of typed combinators which can be used to describe the type of a C function. The type description determines a corresponding OCaml type for the C function, and generates code which converts between the OCaml and C data representations. This approach works in both directions: the same type description can be used to expose a C function to OCaml, or (as in our `libtls` replacement) to expose an OCaml function to C.

Listing 1 shows a ctypes type description for a function tls_server, which wraps an OCaml function of the same name. The ctypes function I.internal takes three arguments: the name of the generated C function, a description of the type of the function, and an OCaml function which is wrapped by the generated code.

***Harmonising memory management*** Memory management is a second fundamental difference between C and OCaml.

In OCaml, memory is managed by the garbage collector (GC). Allocated values are collected (freed) after the GC has determined that they are no longer reachable, and the GC may move values from one part of memory to another in a compaction phase. Since OCaml programs do not manipulate addresses, collection and compaction are not generally visible to a program.

In C, memory is managed by the programmer. Allocated values must be freed by the programmer when he has determined that they are no longer reachable. Since almost all C programs involve significant address manipulation, the C runtime has no freedom to collect or move allocated values.

While both these views of memory management are internally consistent, care is needed in a program or library (such as `libnqsb-tls`) that attempts to combine them. Exposing the address of an OCaml value to C could have disastrous consequences, since the OCaml runtime assumes that the addressed value can be safely moved, while the C program assumes that it will remain where it is.

Ctypes encourages a programming style that avoids many of these difficulties, by making it easy to pass C values to OCaml (which is generally safe) and difficult to pass OCaml values to C (which is often dangerous). However, when reimplementing an existing C interface there is less control over memory management. For example, the tls_server function (Listing 2) involves returning a representation of the internal state of the tls library; in our case this is an OCaml value.

The solution, as often, is an extra level of indirection. Rather than returning the address of the OCaml value from tls_server, our implementation returns a pointer to a small block which holds the address of the OCaml value. Allocating the block in C-managed memory ensures that its address remains stable. Registering the block as a root with the OCaml runtime ensures that it is updated if the value is moved during compaction.

***Bridging programming styles*** Replicating the exact behaviour of a C library in OCaml can involve additional challenges. For example, ocaml-tls takes great care to avoid misconfiguration, and uses OCaml's rich type system to ensure that parameters are correct. LibreSSL's libtls, on the other hand, uses an incremental approach, where the user sets parameters one by one; the parameters may be inconsistent until the point where the TLS context is finally generated. In order to bridge the gap between these two styles, we stage each parameter in a large record which stores parameters as options until the generation point, where we check the whole configuration set and convert it to ocaml-tls's more structured representation.

## Integration with system services

A number of services in OpenBSD, including `httpd`, `spamd`, `ntpd` and `ftp`, rely on `libtls` for communications security. For example, the `httpd` web server uses `libtls` to implement the HTTPS protocol.

In order to check that our library closely matches the behaviour of `libtls`, we linked `httpd` against `libnqsb-tls` in place of `libtls` (Figure 1).

Integrating `libnqsb-tls` with `httpd` exposed a number of disparities in behaviour with `libtls`. For example `httpd` uses the `pledge` system call, part of OpenBSD's capabilities system, to relinquish the privileges needed to read a certificate once reading is no longer necessary. Our initial implementation was written with the assumption that certificates could be read at any point, which led to a runtime failure.

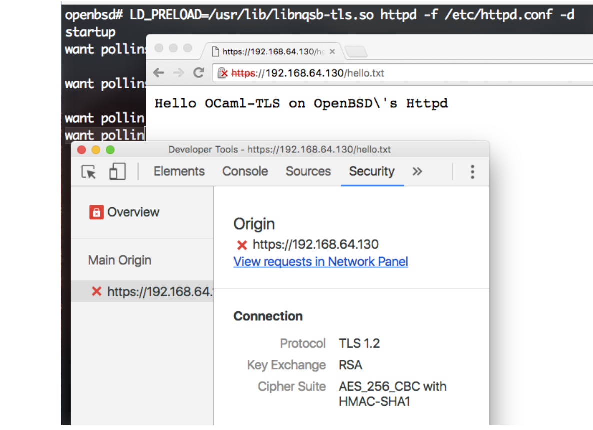


**Figure 1.** `httpd` running `libnqsb-tls`.

## Performance

Performance is a common consideration when deciding whether to replace a low-level component with a high-level alternative. There are two possible causes for concern. First, moving to a high level language typically means giving up some control over data layout and other low-level details, which can make code difficult to optimize by hand. Second, the FFI layer that integrates the high level component with other parts of the system introduces a layer of indirection with potentially significant overheads.

The results from preliminary performance tests with `libnqsb-tls` are promising. Our tests involved transferring a large file (over 1GB) over HTTPS using `httpd`, and measuring the transfer speed when either `libtls` or `libnqsb-tls` was used to encrypt the payload. In this setting the performance of `libnqsb-tls` was within a factor of two of the performance of `libtls`. Since we have not yet made any attempts at optimization, we hope to see significant improvements to this figure in the future.

## Conclusion

This submission describes some reusable lessons from our experience in building a replacement for a core OpenBSD component (`libtls`) based on an existing OCaml library (`ocaml-tls`). Our replacement library, `libnqsb-tls`, displays promising performance, integrates with various OpenBSD components, and is implemented without any C code (except for a tiny reusable stub that initializes the OCaml runtime). If our proposal is accepted, we plan to include a live demonstration of `libnqsb-tls` during the talk.

## References

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49, 2012.

David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 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., USA, August 12-14, 2015.*, pages 223–238, 2015.

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. Declarative foreign function binding through generic programming. In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 198–214, 2016.