# EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats

Tahina Ramananandro* Antoine Delignat-Lavaud* Cédric Fournet* Nikhil Swamy*
Tej Chajed† Nadim Kobeissi‡ Jonathan Protzenko*
*Microsoft Research †Massachusetts Institute of Technology ‡Inria Paris

## Abstract

We present EverParse, a framework for generating parsers and serializers from tag-length-value binary message format descriptions. The resulting code is verified to be safe (no overflow, no use after free), correct (parsing is the inverse of serialization) and non-malleable (each message has a unique binary representation). These guarantees underpin the security of cryptographic message authentication, and they enable testing to focus on interoperability and performance issues.

EverParse consists of two parts: LowParse, a library of parser combinators and their formal properties written in F⋆; and QuackyDucky, a compiler from a domain-specific language of RFC message formats down to low-level F⋆ code that calls LowParse. While LowParse is fully verified, we do not formalize the semantics of the input language and keep QuackyDucky outside our trusted computing base. Instead, it also outputs a formal message specification, and F⋆ automatically verifies our implementation against this specification.

EverParse yields efficient zero-copy implementations, usable both in F⋆ and in C. We evaluate it in practice by fully implementing the message formats of the Transport Layer Security standard and its extensions (TLS 1.0–1.3, 293 datatypes) and by integrating them into MITLS, an F⋆ implementation of TLS. We illustrate its generality by implementing the Bitcoin block and transaction formats, and the ASN.1 DER payload of PKCS #1 RSA signatures. We integrate them into C applications and measure their runtime performance, showing significant improvements over prior handwritten libraries.

## 1 Introduction

Because they are directly exposed to adversarial inputs, parsers are often among the most vulnerable components of security applications, and techniques to simplify their construction while ensuring their safety and correctness are valuable. Hence, developers prefer self-describing formats like JSON or XML (with universal implementations) or use automated tools and libraries to generate parsers from structured

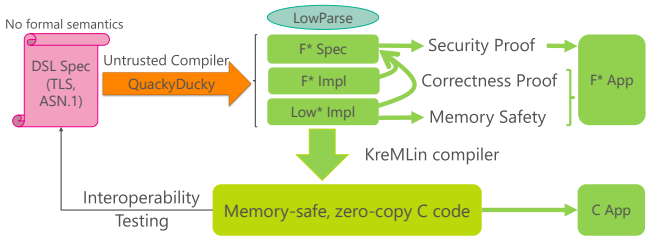


Figure 1: EverParse architecture

format specifications, or even from type declarations in Java or C++. However, when parsing is on the critical path of an application's performance, or because of requirements of the message format (such as compliance with a standard), developers may be forced to write and maintain their own parsers and serializers in low-level unsafe languages like C, increasing the risk of attacks triggered by malicious inputs.

More specifically, when the application authenticates messages in some way—using for instance cryptographic hashes, MACs, encryptions, or signatures—it is critical for security to ensure that the messages are verified, parsed, and interpreted by the receiver exactly as intended by the sender before serialization. This is often at odds with general-purpose formats and tools that may not provide such non-malleability guarantees.

This paper presents EverParse, a new framework to automatically generate efficient, low-level parsers and serializers in C from declarative descriptions of tag-length-value (TLV) binary message formats. The generated parsers and serializers are formally verified to be *safe* (no use-after-free, no buffer overruns, no integer overflows, ...), *functionally correct* (parsers and serializers are inverse of one another), and *non-malleable* (valid messages have unique representations). With EverParse, developers of low-level protocols can enjoy the ease of programming and maintenance expected from parser generators, and stop worrying about details of the message format and trade-offs between security and performance.

**Architecture Overview.** Figure 1 depicts the overall architecture of EverParse and its two main components: a simple, untrusted frontend (named QuackyDucky) for compiling message format descriptions; and a library of verified parsers and serializers (named LowParse).

Verification is based on F* [48], a programming language and proof assistant. Whereas F* is a high-level functional language, whose code extracts by default to OCaml or F#, it also embeds a language named Low* for writing verified low-level code that extracts to C using a tool named KReMLin [38]. EverParse uses Low* to program efficient, low-level parsers and serializers, proving them safe, correct and non-malleable in F*, and then extracting them to C. The resulting C code can be compiled using several off-the-shelf C compilers, including CompCert [28] for highest assurance, or more mainstream compilers like Clang or GCC.

The input of EverParse is a message format description for a collection of types, in the C-like language commonly used in RFCs and similar specifications. QuackyDucky translates this description into a collection of F* modules, one for each input type, and F* typechecks each of these modules to verify their safety and security. The modules produced by QuackyDucky include a formal parser specification (using high-level F* datatypes) and two correct implementations of this specification: one high-level and the other in Low*, suitable for extraction to safe C code. This lower-level implementation enables efficient message processing; it automatically performs the same input validation as the high-level parser, but it operates in-place using interior pointers to binary representations within messages. Its performance is similar to handwritten C code—but its safety, correctness, and security are automatically verified. Hence, rather than verifying existing, ad hoc C code by hand, which would require much effort and expertise even for small protocols, our toolchain automatically yields C code verified by construction.

The code generated by QuackyDucky consists of applications of *combinators* in LowParse, which are higher-order functions on parsers. For instance, given a integer parser, one can build a parser for pairs of integers by applying the concatenation combinator to two copies of the integer parser. While parser combinators are widely used in functional languages [22, 27], they are usually more difficult to apply in languages that do not easily support higher-order programming with closures, like C. However, by employing partial evaluation within F*, we specialize higher-order code to efficient, ad hoc, first-order C code.

We carry out all proofs on combinators once and for all within LowParse. Only the conditions for composing them must be checked (by typing) in the code produced by QuackyDucky. LowParse is split into three layers: one for *specifications*, where we prove non-malleability, one for *high-level functional implementations*, which are proved functionally correct with respect to specifications, and one for *low-level implementations*, which operate on positions within buffers and are proved functionally correct and memory safe.

EverParse code can be used in two ways. A verified F* application can use the formal specification for its security proof, and either the high-level or low-level implementation—this is the approach adopted for verifying protocols as part of the Everest project [6], and notably the MITLS [7] implementation of the TLS secure channel protocol. Alternatively, a native C/C++ application can use the interface extracted from the Low* implementation by the KReMLin compiler—this is the approach taken in this paper for performance evaluation.

**Our contributions** We present the following contributions:

- A definition of message-format security, motivated by a discussion of several vulnerabilities whose root cause is malleability (§2).
- QuackyDucky, a compiler from tag-length-value message format descriptions to their high-level datatype specifications and low-level implementations. It provides the first implemented zero-copy and secure message format language that captures several existing protocols and standards, including TLS, PKCS #1 signature payloads, and Bitcoin (§4).
- LowParse, a library of verified parser and formatter combinators, with support for non-malleability proofs (§5).
- A qualitative evaluation of EverParse: we present a complete case study of applying EverParse to the message formats of all TLS versions, featuring many improvements and corrections over the standard's descriptions. We integrate the generated high-level implementation into MITLS, an implementation of the TLS protocol in F*. For a few select types, we also replace the high-level implementation with the Low* one (§3).
- A quantitative evaluation of EverParse: we compare the performance of our extracted low-level parsers for Bitcoin blocks and the ASN.1 payload of PKCS #1 signatures with their counterparts in Bitcoin Core and mbedTLS. We find that our automatically generated code meets, and in some cases significantly exceeds, the performance of hand-written C/C++ reference code (§6).

All the components of EverParse and its dependencies are open-source and publicly available at https://github.com/project-everest/everparse

## 2 Parsing Security: Definitions & Attacks

In this paper, we focus on applications that authenticate the contents of serialized messages in some way. Cryptographic mechanisms provide (serialized) bytestring authentication, whereas applications rely on (parsed) message authentication. Hence, correctness and runtime safety are not sufficient to preserve authentication: a correct parser may accept inputs outside the range of the serializer, or multiple serializations of the same message, which may lead to subtle, and sometimes

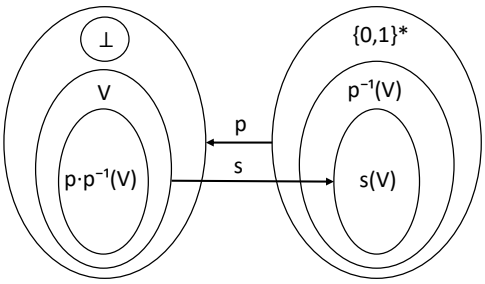


Figure 2: Parsing and serialization functions

devastating, vulnerabilities. We propose a security definition for authenticated message formats to prevent such vulnerabilities, and illustrate it using known high-impact attacks against popular applications.

## 2.1 What is a Secure Message Format?

We first set up notations for parsers and serializers, illustrated in Figure 2, and define their properties of interest. Given a set $\mathcal{V}$ of valid messages,

- a *parser* is a function $p : \{0,1\}^* \to \mathcal{V} \uplus \{\bot\}$ that returns either a message $m \in \mathcal{V}$ or a parsing error $\bot$;
- a *serializer*, or *formatter*, is a function $s : \mathcal{V} \to \{0,1\}^*$.

Informally, parsers and formatters are inverse of one another. A parser is *correct* with respect to a serializer when it yields back any formatted message: $\forall m \in \mathcal{V}, p(s(m)) = m$, and *exact* when it accepts only serialized messages: $p^{-1}(\mathcal{V}) = s(\mathcal{V})$.

Parsers may also be considered on their own. A parser is *non-malleable* (or *injective*) when it accepts at most one binary representation of each message: $\forall x, y \in \{0,1\}^*, p(x) = p(y) \Rightarrow (x = y \vee p(x) = \bot)$, and *complete* (or *surjective*) when it accepts at least one binary representation of each message: $p(\{0,1\}^*) \setminus \{\bot\} = \mathcal{V}$. If $p$ is a non-malleable parser for $\mathcal{V}$, then $p^{-1}$ is a serializer over $p(\{0,1\}^*) \setminus \{\bot\}$.

We say that $p$ is a *secure parser* for $\mathcal{V}$ if $p$ is non-malleable and complete. If $p$ is secure, then it is also correct and exact with respect to the (unique) serializer $p^{-1}$. We say a serializer $s$ is secure if there exists a secure parser $p$ such that $s = p^{-1}$. (§5 provides more general definitions that account for parsers that do not consume their whole input.)

In the rest of the paper, we only consider parsers that operate on strings of bytes $\mathcal{B} = \{0,1\}^8$.

## 2.2 Attacks on Parsers

**Heartbleed** Unsurprisingly, the most common type of parser vulnerability is simply memory safety bugs. Indeed, one of the most impactful attacks in the past decade, Heartbleed (which is estimated to have affected up to 55% of the top internet websites [17]) is a simple buffer overrun caused by improper validation of the length field in the TLS messages defined in OpenSSL's implementation of the heartbeat protocol extension (shown in Figure 3). Interestingly, the specification of `HeartbeatMessage` is very unusual among TLS types (explained in detail in §3), because it contains a variable length field (`padding`) that is not prefixed by an explicit length (`padding_length` is not defined in the struct, but its value is defined semantically). Indeed, as specified, this type is not expressible in QuackyDucky because the padding length depends on a field of the parent `TLSPlaintext` type, and we only capture dependencies between fields that are concatenated. This forces applications to verify the `padding_length` semantically, increasing the risk of error. The Heartbleed disaster would likely have been averted if the format was specified using standard TLS constructors for variable length fields:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
The total length of a HeartbeatMessage MUST NOT exceed
2^14 or max_fragment_length when negotiated [RFC6066].
The padding is random content that MUST be ignored by
the receiver. The padding_length MUST be at least 16,
and equal to TLSPlaintext.length-payload_length-3 for
TLS and DTLSPlaintext.length-payload_length-3 for DTLS
The sender of a HeartbeatMessage MUST use a random
padding of at least 16 bytes. The padding of a
received HeartbeatMessage message MUST be ignored.
```

Figure 3: Specification of the Heartbeat message (fragment)

```
struct {
  HeartbeatMessageType type;
  opaque payload<0..2^14-21>;
  opaque padding<16..2^14-3>;
} HeartbeatMessage;
```

This example illustrates the benefits of writing message format descriptions in a constrained language: it encourages uniform patterns and enables automated analysis.

**PKCS #1 signature forgery** The PKCS #1 v1.5 signature format illustrates the risks of applying message parsing after a cryptographic operation (in this case, modular exponentiation). Given a public key $(N = pq, e)$ where $p$ and $q$ are large secret primes, the raw RSA signature $\sigma$ over a message $m$ is computed as $\sigma = m^d \bmod N$ where the secret exponent $d = e^{-1} \bmod (p-1)(q-1)$ is hard to compute without knowing $p$ and $q$. As written, this scheme is not usable because $m$ must be smaller than $N$, and it has the undesirable homomorphic property that the signature of the product of 2 messages is equal (modulo $N$) to the product of the signature of each message (thus, it is easy to forge new valid signatures from existing ones). To fix these shortcomings, PKCS #1 v1.5 defines a standard for hashing and padding the message to sign: given an arbitrary message $m$, it is first

hashed into a digest $h$, then stored together with the identifier $a$ of the hash algorithm into an ASN.1 DER [36] structure. The distinguished encoding rules are supposed to ensure that the serializer $\rho$ for this structure is secure. The final signature is obtained by applying raw RSA to $\rho(a,h)$ left-padded to the size of $N$ with padding of the form `\x00\x01(\xFF)*\x00`.

The security of the scheme relies heavily on (the integer interpretation of) the padding: it is computationally hard to forge a valid signature $\sigma$ because $\sigma^e \bmod N$ must be of the form $2^{\lceil \log_2(N) \rceil - 15} - 2^{\lceil \log_2(\rho(a,h)) \rceil + 1} + \rho(a,h)$ for some digest $h$ and hash identifier $a$. It is hard to find such a value by brute force because all but the $\lceil \log_2(\rho(a,h)) \rceil$ last bits are fixed, and inverting the modular exponentiation by $e$ is hard without knowing $d$. However, if the ASN.1 parser $\pi$ used after exponentiation is malleable (or non exact), there may exist a large class of inputs $x$ such that $\pi(x) = (a,h)$. If this class contains inputs that fill most of the $\lceil \log_2(N) \rceil$ bits of the message, the padding may be reduced to `\x00\x01\x00`. When $e$ is small ($e = 3$ is commonly used by legacy public keys), it may be easy to find a value $\sigma$ such that $\sigma^e \bmod N = 2^{\lceil \log_2(N) \rceil - 15} + x$ with $\pi(x) = (a,h)$ for any $h$. For instance, in Bleichenbacher's original description of the attack (retold by Finney [18]), the parser ignores the bytes that appear after the encoded ASN.1 structure, i.e. if $\pi(x) \neq \bot$, $\pi(x||z) = \pi(x)$ for all z. To forge a valid signature for $h$, one can simply compute the cubic root of $2^{\lceil \log_2(N) \rceil - \lceil \log_2(\rho(a,h)) \rceil - 16}(\texttt{\x0100}||\rho(a,h))$.

Ever since its publication, this attack has reappeared in dozens of implementations, including several recent examples (e.g. [9, 13, 37, 49]). Interestingly, the parser malleability bugs that cause the attack are diverse: unparsed extra bytes are tolerated at the end of the message [18, 49]; the parser accepts arbitrary parameters in the algorithm identifier [9]; and a length overflow causes only its last 4 bytes to be counted [13]. This diversity illustrates how difficult it is to write secure parsers and to detect malleability vulnerabilities—some of the attacks above have existed for years in popular libraries. All variants lead to universal signature forgery: an attacker can freely impersonate any client or server, sign malicious code updates, etc.

**Bitcoin transaction malleability** Another well-documented case of parser security attacks occurred against Bitcoin [34] transactions, which are signed by the sender, then hashed (after serialization) and stored in Merkle trees. Transactions are identified by their hash, which covers more data than what the sender signs (in particular, the hash includes the signature itself). The format of transactions is malleable in several ways: one example is the encoding of this signature, which originally did not mandate the ASN.1 DER rules for non-malleability. Another source comes from the ECDSA signing algorithm, which is a randomized scheme (hence, there are many valid signatures for the same message) that always has two valid representations: if $(r,s)$ is a valid signature, then $(r,-s)$ also is, and can be trivially computed without knowledge of the private key. Other sources of malleability are related to the `scriptSig` construct of the Bitcoin Script language,[1] inasmuch as the signature is passed to a stack-based script to authorize spending. In total, BIP62 [50] lists 9 different sources of malleability. Each of them allows an attacker to alter a valid transaction $t$ into a semantically-equivalent valid transaction $t'$ such that $h(t) \neq h(t')$. One way to exploit this is to try to fool someone into believing that a transaction they submitted was rejected by the network, although in reality, it was accepted under a different transaction hash. The Mt. Gox bitcoin exchange blamed this attack for the loss of over 850,000 bitcoins (worth $473M at the time of bankruptcy) and although this claim is heavily disputed, later forensic examination of the blockchain by Decker et al. [12] revealed that in total, 300,000 bitcoins were spent over 30,000 transactions confirmed under a different identifier than originally submitted between Feb 1, 2014 and Feb 28, 2014.

**Ambiguous TLS message** Sometimes, the message specifications themselves are ambiguous, and cannot be implemented by a secure parser. This is the case of the `ServerKeyExchange` message in TLS:

```
enum {dh_anon, dhe, ecdhe, rsa,(255)} KeyExchange;
struct {
  select (KeyExchange) {
    case dh_anon: DHAnonServerKeyExchange;
    case dhe: SignedDHKeyExchange;
    case ecdhe: SignedECDHKeyExchange;
    case rsa: Fail; /* Force error: no SKE in RSA */
  } key_exchange;
} ServerKeyExchange;
```

This message represents an untagged union: the struct is missing a field of type `KeyExchange` that clarifies which case to use in the union. A parser for an untagged union can only be secure if the format of all cases share no common prefix. The specification of TLS assumes that the key exchange algorithm is available from the context (in this case, it is part of the negotiation process). However, it turns out that the security of the TLS negotiation depends itself on the `ServerKeyExchange` message. This leads to a real practical attack reported by Mavrogiannopoulos et al. [32], where a `SignedDHKeyExchange` is interpreted as a bogus `SignedECDHKeyExchange`. Worryingly, two other TLS types use untagged unions: `ClientKeyExchange` (in TLS 1.2) and `CertificateEntry` (in TLS 1.3).

## 3 Case Study: the TLS Message Format

We choose the TLS message format as our main case study for several reasons: the message format description of TLS is reasonably specified; it is designed to be secure and extensible; it defines hundreds of types that exercise the full range

[1] https://en.bitcoin.it/wiki/Script

```
uint32 word; /* Type declaration */
word digest[16]; /* Fixed-length array of 4 words */
word phrase<0..2^8-1>; /* List of 0 to 16 words */
struct {
  opaque id[32]; /* Array of 32 bytes */
  uint16 payload<2..8>; /* List of 1 to 4 uint16 */
  digest payload_digest;
} body; /* Struct with 3 fields */
enum {
  request (0x2300), /* Constant tag */
  response (0x2301),
  (65535) /* Indicates 16 bit representation */
} header; /* Enum with 2 defined cases */
struct {
  header tag;
  select(tag) { /* Tagged union */
    case request: body;
    case response: phrase;
  } x; /* Enum-dependent field type */
} message;
struct {
  uint24 len; /* Explicit length */
  message data[len]; /* Ensures length(data)=len */
} batch; /* Length encapsulation */
```

Figure 4: Sample type descriptions in TLS message format.

of available combinators in LowParse; and there exists a verified F* implementation of TLS that we can use to test the integration of the generated parsers (including the integration of parser security lemmas into the protocol security proof).

**Language Description** IETF's RFC 2246 [14] specification of TLS 1.0, published in 1999, includes a section that describes the presentation language of its message format, inspired by C and XDR [46], and illustrated in Figure 4. A description consists of a sequence of type declarations. The base types are unsigned fixed-length integers `uint8`, `uint16`, `uint24`, and `uint32`, with `opaque` being used instead of `uint8` to indicate raw bytes. The type constructors are fixed-length arrays, variable-length lists, structs, enums, and tagged unions. The length boundaries of arrays and lists are all counted in bytes rather than in elements: for instance, type `digest` in Figure 4 is an array of elements of type `word` whose binary representation takes 16 bytes in total; since each `word` takes 4 bytes, this array holds exactly 4 elements. Arrays can be constructed only from fixed-length types, whereas lists can be defined for any types: as illustrated by `answer` and `payload`, their format declares the range of their length; and their binary representation embeds their actual length within that range.

Following the convention of RFCs, we interpret types in terms of the byte sequences that represent their elements. The representation of a struct is the concatenation of the representations of each of its fields in sequence, without any padding. Arrays are the concatenation of elements whose total length in bytes is the array's annotated size. Lists are represented by a length field encoded in a fixed number of bytes (determined by the maximum length of the list, encoded in big endian), followed by a concatenation of the elements. A special case of structs are length-dependent fields, e.g., the `batch` type in Figure 4. In these types the first field describes the length of a single (variable-length) element of the specified type of the second field represented adjacently. The interpretation of enumerations contains the big endian encodings of its elements in a constant number of bytes determined by the size descriptor of the enum type. Tagged unions (like `message` in the figure) are encoded as the concatenation of the tag's enum representation followed by the encoding of the corresponding case's type. TLS messages are more compact than many TLV formats: explicit tags only appear for tagged unions, and lengths only for lists (or when ascribed). All structural information is erased, in contrast with BSON [33] (which encodes field names) or Protocol Buffers [20] (which encodes field numbers).

We automatically extracted the data format descriptions from the RFCs for TLS 1.2 [15] (including descriptions also for TLS 1.0 and TLS 1.1), for TLS 1.3 [40], for TLS extensions (RFC 6066), and from the TLS IANA parameter assignments, which defines additional constants for enumerations. We then merged them together by hand, and edited some of them to fix minor mistakes, avoid name clashes in the original descriptions, and gain precision (e.g. by adding length dependencies documented in the RFC text).

**Extensibility** A difficult issue for any format description languages is extensibility: as new versions of the protocol are defined, it is often necessary to extend messages with new fields and cases while maintaining compatibility with older implementations. To address this problem, TLS was designed with extensibility through open enumerations. As a simple example, TLS has an enum type that defines the possible cipher suites to negotiate. Receiving a value that doesn't match any of the defined cases of the enum is not a parsing error—instead, the value should be treated as unknown but valid, and the receiver should ignore it in the rest of the negotiation. This also applies to enums that act as a tag for unions. For instance, the hello messages contain a list of extensions tagged with an extension type. Although this is implicit in the standard, it is possible to define a default type for unknown values in a `select`. The protocol is extended by defining new values for enums (such as new cipher suites or new group names), and new defined cases for tagged unions (for instance, new extensions). Interestingly, many TLS implementations fail to understand this concept, and incorrectly reject unknown values. To fight this problem, Google recently introduced GREASE [4], which causes Chrome to randomly include undefined values in all extensible fields of the protocol, thereby enforcing that implementations that interoperate with Chrome be extensible.

Unfortunately, the TLS standard does not clearly say which

```
/* All TLS versions*/
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
 } Extension;

/* From RFC 5246, section 7.4.2 */
opaque ASN.1Cert<1..2^24-1>;
struct {
  ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;

/* From RFC 8446, section 4.4.2 */
enum { X509(0), RawPublicKey(2), (255)} CertType;
opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
struct {
  select (certificate_type) {
    case RawPublicKey: ASN1_subjectPublicKeyInfo;
    case X509: ASN.1Cert; } cert;
  Extension extensions<0..2^16-1>;
} CertificateEntry;
struct {
  opaque certificate_request_context<0..2^8-1>;
  CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

Figure 5: The `Certificate` type for TLS 1.2 vs TLS 1.3.

enums and which tagged unions are extensible in the message format, and instead explain their intended semantics in text. In QuackyDucky, we add an explicit annotation to mark which enums are extensible, and we extend the syntax of `select` to support `default` cases. For instance, we mark the tag of extensions `ExtensionType` with the `/*@open*/` attribute, but not the tag of messages `HandshakeType`, as the RFC states that receiving any unknown message is an error.

**Protocol versions** Another complication stems from version differences not captured by extensibility. Consider the Certificate message in TLS 1.2 [15] and TLS 1.3 [40]. Its format, listed in Figure 5, illustrates several problems with the TLS specification. First, the two definitions of the `Certificate` type are mutually incompatible, even though the `Certificate` message is defined in both versions using the same `handshakeType` tag. Second, the `CertificateEntry` type of TLS 1.3 uses an untagged union, where the value of the tag depends on the context rather than on a value sent over the network (as in `ServerKeyExchange` and `ClientKeyExchange` where the key exchange algorithm is omitted). Third, the `Extension` type is under-specified: there are complex rules and tables about which extension may appear in which message (see [40, §4.2]), and the type of each extension contents depends on which message it appears in. None of these constraints are currently captured in type definitions.

To address the issue of conflicting definitions across versions, we split the definitions of incompatible types such as

| TLS 1.2 | TLS 1.3 |
|---|---|
| `struct {`<br>`  HandshakeType type;`<br>`  uint24 length;`<br>`  select (type) {`<br>`  case hello_request:`<br>`  Empty;`<br>`  case certificate:`<br>`  Certificate12;`<br>`  /* ... */`<br>`  case finished:`<br>`  Finished;`<br>`  } m[length];`<br>`} Handshake12;` | `struct {`<br>`  HandshakeType type;`<br>`  uint24 length;`<br>`  select (msg_type) {`<br>`  case eoed:`<br>`  Empty;`<br>`  case certificate:`<br>`  Certificate13;`<br>`  /* ... */`<br>`  case key_update:`<br>`  KeyUpdate;`<br>`  } m[length];`<br>`} Handshake13;` |

Figure 6: Specialized `Handshake` types for TLS 1.2 and 1.3

`Certificate` and we define version-specific variants of the handshake message type, shown in Figure 6. Before version negotiation, hello messages are parsed using a third, version-agnostic `Handshake` type. We then switch to parsers for the negotiated version, thus ensuring that the following messages are parsed using precise, version-specific types.

To address the problem of untagged unions, we introduce an `/*@implicit*/` attribute for tags, which instructs QuackyDucky to generate an interface where the value of the tag is passed as an additional argument to the parser and formatter. This approach is not compositional, and comes with a restriction: if a type that contains an implicit tag appears in another type, it must appear at a location that includes an explicit length. The parsing will be *staged*: when it reaches the surrounding length, it skips its contents, leaving it uninterpreted. The application needs to manually call the parser for the untagged union by providing the tag value. The presence of untagged unions is a clear mistake in message formats, as the application is responsible for providing the correct tag when it calls the parser, and thus, we only provide a conditional security guarantee in this case.

Similarly to what we did with message types, we split and specialize the definition of extension types for each message that may include them (hellos, hello retry, encrypted extensions, certificate, certificate request, new session ticket). This also reveals some interesting mistakes in RFCs. For instance, in [41, Appendix A], the authors fail to understand the purpose of the explicit length around extensions, and incorrectly believe it is redundant with the length of the list in the extension contents. They claim merging the two lengths makes the extension less ambiguous; in reality, their change makes the format *more ambiguous*: it is no longer possible to distinguish between receiving no ticket and receiving an empty ticket. To handle such corner cases, we add support in QuackyDucky to coerce `opaque` arrays with explicit lengths to `opaque` lists after parsing.

# 4 Compiling Message Format Descriptions

We now present our compiler from the message formats of §3 to parsers and serializers for processing these messages in Low⋆ [38]. We briefly review F⋆ and Low⋆ (§4.1), then explain the code generated by QuackyDucky in three parts: datatypes and parser specifications for verification purposes (§4.2); high-level functional parsers and serializers (§4.3); and lower-lever code for reading (§4.4) and writing (§4.5) messages.

QuackyDucky recursively descends through the structure of the format description, generating parsers and serializers for compound types from those previously generated, while keeping track of their properties (notably their length boundaries). QuackyDucky mostly composes the combinators provided by LowParse, described in §5. In contrast with this library, which involves complex proofs for a few generic combinators, the generated code is verbose but shallow, enabling us to automatically verify its safety, correctness, and non-malleability using the properties verified in LowParse.

## 4.1 Verified Programming in Low⋆ (Review)

**The F⋆ language and proof assistant** We carry out our specification, implementation and proofs using F⋆ [48], a functional language and proof assistant based on dependent types. A simple form of dependent types supported in F⋆ is *refinement types*, to represent types of values satisfying additional properties: whereas int is the F⋆ type of mathematical integers, the type nat of non-negative mathematical integers is defined as the refinement type (x: int { x ≥ 0 }). F⋆ supports types that depend on other types and values. It also supports functions where argument types can depend on the values of the previous arguments and the type of the return value of a function can depend on the values of its arguments. For instance, the integer division function, which would have a simple type int → int → int in a functional language such as OCaml, can be given a more precise type, such as (a: int) → (b: int {b > 0}) → (q: int { 0 ≤ a − b ∗ q < b }) meaning that F⋆ will reject, at typechecking time, any call to such a function if it cannot prove the second argument is strictly positive (b > 0). Conversely, the caller can use the postcondition on its return value 0 ≤ a − b ∗ q < b to prove further goals.

F⋆ is not just a proof assistant: it is also a programming language enjoying automatic translation (*extraction*) into executable languages such as OCaml or C; in this extraction process, the user can mark some of their F⋆ functions as *ghost*, to have them erased at extraction time, such as lemmas and proofs, but also auxiliary functions that need not, or cannot, be executed at run time. To this end, F⋆ equips its function types with an *effect* system. By default, given two types t and u, all functions from type t to u, in the type t → u, will be extracted, one says that they are in the Tot effect, so their type can be written as t → Tot u; to mark a function ghost, the user needs to use the GTot effect, hence using the function type t → GTot u instead. Stateful functions that operate on mutable objects (such as a mutable array of bytes, i.e. `uint8_t*` in C) live in the ST effect and are extracted to stateful OCaml or C code.

All F⋆ proofs rely on automatic encoding of proof obligations to first-order logic, which are then solved by automatic theorem provers such as Z3, leading to reduced overall proof effort. F⋆ also has a tactic system [31] that can be used in cooperation with, or in replacement of, Z3-based proofs.

**The Low⋆ subset of F⋆ and its extraction to C code** Users who wish to use our parsers and serializers not only demand performance, but also the ability to integrate our code into their codebases. Therefore, OCaml is oftentimes not a viable option; instead, we compile our F⋆ code to C.

As shown in Figure 1, specifications are first implemented in F⋆. This implementation can already be compiled to C using a dedicated compiler, KReMLin. However, lacking further restrictions, this F⋆ implementation relies on lists and bytes as values, meaning that compiling the F⋆ implementation through KReMLin generates code that (i) is inefficient, because it uses functional byte copies and linked lists; (ii) is not idiomatic: no C programmer would write such code; and (iii) requires a garbage collector, since lists and bytes are persistent values with no lifetime (as in, say, OCaml).

To avoid these shortcomings, the Low⋆ [38] subset of F⋆ defines a C-like memory model that talks about the stack and the heap, along with corresponding abstractions and libraries, e.g. for mutable arrays, machine integers and in-place loops. Using Low⋆ requires the programmer to rewrite their code and reason about spatial and temporal safety; in exchange for these added constraints, the F⋆ code is compiled by KReMLin to idiomatic, readable, efficient C code that does not require a garbage collector. The Low⋆ restrictions only apply to executable code; computationally-irrelevant portions of the program, such as proofs and ghost code, retain the full power of F⋆, since they are eliminated when compiled to C.

## 4.2 Datatypes and Parser Specifications

Continuing from §2.1, we use F⋆ types to represent sets of parsed values. Hence, QuackyDucky generates a parsed type *t* for each named format description in its input.

Our F⋆ types for parser and serializer specifications are listed below.

```
type parser (t:Type) (k:meta) =
  p: (input: seq uint8 → GTot (option (t * l: nat{l ≤ length input})))
  { parser_prop k p }

type serializer (#t:Type) (#k:meta) (p: parser t k) =
  s: (t → GTot (seq uint8))
  { ∀ (x:t) . p (s v) == Some (v, length (s v)) }
```

The parser type definition is parameterized by a parsed type *t* and some metadata *k* (explained below) that records verified properties of the parser. It states that a parser is a pure function

that takes as input a sequence of bytes (of arbitrary length) and returns an optional result that consists of some parsed value of type $t$ and the number of input bytes consumed. GTot states that this parser is ghost, that is, used only for verification; option indicates that it may return nothing in case parsing fails; the refinement $l \leq$ length input ensures that it consumes at most a prefix of its input; the refinement parser_prop $k$ p states that $p$ is *non-malleable* (see formal definition in §5) and that it satisfies the properties recorded in $k$.

The metadata $k$ includes a verified range of lengths of input that the parser may consume. This range provides useful bounds for programming with this parser, for instance to wait for a minimal number of input bytes before parsing, or to allocate I/O buffers of adequate sizes. Internally, QuackyDucky also relies on this information to select more efficient implementations (for example when the input length is fixed) and to require that some parsers consume at least one byte (for example to compute the length of a list from its binary format). The metadata may include additional properties, indicating for instance that the parser always fails; or that it always succeeds given enough input bytes; or that it is no-lookahead (see §5).

The serializer type definition is indexed by a *parser* for $t$, not just by $t$ (the # makes this parameter an implicit argument, as it can be inferred from p). It states that a serializer specification is a pure, total, ghost function from values of type $t$ to sequences of bytes, such that parsing its output for any value $v$ of type $t$ succeeds and yields back $v$ and its binary length.

**Running Example** Consider the following excerpt of the TLS 1.3 wire format description [40] for the body of its first ClientHello message.

```
struct {
  ProtocolVersion version;
  opaque random[32];
  opaque session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  Compression compressions<1..2^8-1>;
  ClientHelloExtension extensions<0..2^16-1>;
} ClientHello;
```

This message includes a 2-byte protocol version, a 32-byte random nonce, a variable-length session identifier for resumption, and variable-length list of proposed cipher suites, compression methods, and extensions. QuackyDucky translates it to a corresponding F⋆ record type:

```
let in_range x min max = min ≤ x ∧ x ≤ max
type clientHello = {
  version : protocolVersion;
  random : (b:bytes {Bytes.length b == 32});
  session_id : (b:bytes{in_range (Bytes.length b) 0 32});
  cipher_suites : (l:list cipherSuite{in_range (List.length x) 1 32767});
  compressions : (l: list compression{in_range (List.length l) 1 255});
  extensions : (l: list extension{in_range (extensions_bytesize l) 0 65535});}
```

This high-level type includes precise length information, coded as refinements. Since the elements of the first three lists have constant binary lengths, QuackyDucky computes precise bounds on their numbers of elements. Conversely, the extensions in the last list are themselves of variable lengths, hence QuackyDucky captures the bounds on its total binary size using an auxiliary function extension_list_bytesize previously defined from the extension serializer. The main benefit of capturing these constraints is to ensure that *all* messages of type clientHello can be serialized to a standard-compliant bytestring.

QuackyDucky also generates the corresponding metadata, parser and serializer specifications.

```
let clientHello_meta = {low=43;high=131396; ...}
val clientHello_parser: parser clientHello_meta clientHello
val clientHello_serializer: serializer clientHello_parser
```

The parser metadata is exposed in the generated interface (indicating, e.g., that the shortest TLS clientHello body message takes 43 bytes) whereas the parser and serializer specs are kept abstract—the interface gives their types, but hides the details of their wire format. Thus, the three lines above state the abstract, joint properties of our generated parser and serializer specs (including non-malleability and round-trip properties) and typechecking these specs ensures these properties hold.

Anticipating on the combinators defined in §5, we give below an outline of the generated definition of clientHello_parser, which parses our sample message by successively calling the parsers corresponding to each of its fields:

```
let clientHello_parser =
  ((protocolVersion_parser × clientHello_random_parser) × ...)
    synth (fun ((pv, r), ...) → { protocolVersion = pv; random = r; ... })
```

These definitions are used only as reference implementations and are not extracted to C. In a second stage, QuackyDucky generates actual parsers and serializers, and typechecking their code ensures they safely implement these specs.

## 4.3 Functional Parsers and Serializers

QuackyDucky generates high-level functional parsers and serializers, with the following interface. (The "32" suffix indicates that their code uses 32-bit machine integers instead of the unbounded integers in their specs.)

```
val clientHello_parser32: parser32 clientHello_parser
val clientHello_serializer32: serializer32 clientHello_serializer
```

We systematically index our implementations by their specifications. Here, for instance, the type definition parser32 $p$ used above simply states that a high-level parser for the parser-specification $p$ is a pure function that takes a (bounded, immutable) F⋆ bytestring and returns the same result as $p$ on the corresponding sequence of bytes except that the consumed length is an unsigned 32-bit integer.

We give below an F⋆ code sample illustrating their use: a 'reader' function that counts the number of cipher suites in a given ClientHello message in wire format, and a 'writer' function that builds a message given a configuration.

```
let count_ciphersuites (input: bytes): UInt32.t =
  match clientHello_parser32 input with
  | None → 0ul
  | Some ch → List.length ch.ciphersuites

let compute_extensions config: (l: list extension {...} ) = ...

let hello (cfg: config) : bytes =
  clientHello_serializer32 {
    version = TLS_1p3;
    random = config.random;
    ...;
    extensions = compute_extensions config; }
```

This code and its supporting parsers and formatters operate on immutable bytestrings. Although it can be safely extracted to C, it is inefficient, and the implicit allocations and copies mandate the use of a garbage-collector. For example, clientHello_parser32 allocates 4 lists and briefly uses only one.

## 4.4 Low-Level Accessors and Readers

To provide more efficient implementations, QuackyDucky also generates code for a lower-level API that enables in-place processing of messages in their binary formats.

We begin with a low-level alternative to parsing. For each parser specification $p$, QuackyDucky provides functions that operate on an input buffer. A *validator* reads the input buffer and returns either the number of bytes that $p$ would consume by successfully parsing its contents, or an error code. Thus, successful validation ensures the existence of a high-level message in binary format, but does not construct it. Assuming the input buffer is valid, then, for each field of the message, an *accessor* computes the position of the first byte in its binary representation. This guarantees in particular that this representation of this element of the message is also valid. Accessor computations are similar to pointer arithmetic in C, or "get element pointer" computations in LLVM, but they sometimes require reading the lengths of intermediate parts in order to skip them. For each base type (e.g. 16-byte unsigned integers), a *reader* takes an input buffer and position and actually parses and returns a value of that type.

Continuing with our example, QuackyDucky produces a validator for clientHello and an accessor for every field (shown below only for its cipherSuites).

```
val clientHello_validator : validator clientHello_parser
val accessor_clientHello_cipherSuites :
  accessor
    clientHello_parser
    clientHello_cipherSuites
    clientHello_cipherSuites_parser
```

The type definitions validator and accessor are still parameterized by parser specifications, but they are more complex, since they describe functions that operate on pointers to mutable buffers. We represent their input as a *slice*, that is, a Low$^\star$ buffer (§4.1) and its length, and a *position* within this slice. (Experimentally, computations on integer positions based on a single pointer are simpler to verify, and better optimized by C compilers.) Accordingly, our validators return either the final position in the slice after successful validation, or an error coded as a large integer. We illustrate their use by re-implementing the count_cipherSuites example of §4.3 in Low$^\star$.

```
let count_ciphersuites_inplace (input:slice) (pos:UInt32.t) =
  let pos_final = clientHello_validator input pos in
  if max_length < pos_final then 0ul (* invalid input *)
  else
    let pos_ciphersuites = accessor_clientHello_ciphersuites input pos in
    clientHello_cipherSuites_count input pos_ciphersuites
```

The last line calls another QuackyDucky-generated function that returns the length of a list of cipher suites in wire format. In this case, since each cipher suite takes exactly two bytes, this length is computed without actually reading the list content, by dividing its binary length by two.

Unsurprisingly, this function yields C code of the form:

```
// A slice is the pair of a byte array and its size
typedef struct {uint8_t * base; uint32_t len; } slice;
uint32 count_ciphersuites_inplace(slice input, uint32 pos) {
  uint32 pos_final = clientHello_validator(input,pos);
  if (max_length < pos_final) return 0;
  else {
    uint32 pos_ciphersuites = accessor_clientHello_ciphersuites(input,pos);
    return clientHello_cipherSuites_count(input,pos_ciphersuites);
  }
```

Once compiled by Clang, we can check on the resulting machine code that the 'else' branch eventually boils down to (1) adding 34 to pos to skip the first two fields; (2) reading and adding the one-byte length of the third field; (3) reading the two-byte length of the fourth field and shifting it by one.

Validators, accessors, jumpers and readers are specified using a *validity* predicate, $\mathsf{valid}(p,m,b,i)$ stating that the parser $p$ succeeds when provided the bytes in buffer $b$ starting from offset $i$ in memory state $m$. If this predicate holds, then, thanks to the injectivity property of $p$, there is a unique value $\mathsf{contents}(p,m,b,i)$ returned by the parser, and an offset $\mathsf{getpos}(p,m,b,i)$ within $b$ one past the end of the representation of that value. This validity predicate is the post-condition of validators when they succeed, and is the precondition of jumpers and readers; accessors for struct fields have the validity predicate for the struct (resp. field) parser as a precondition (resp. postcondition). We give below the type definitions for validators and readers:

```
type validator (#k: meta) (#t: Type) (p: parser k t) =
  (input: slice) → (pos: U32.t) → ST U32.t
  (requires (fun m → live_slice input pos ∧ input.len ≤ max_length))
  (ensures (fun m pos' m' → m ≡ m' ∧
    (valid(p, m, input, pos) ⇔ pos' ≤ max_length) ∧
    (pos' ≤ max_length ⇒ pos' == getpos(p, m, input, pos))))

type reader (#k: meta) (#t: Type) (p: parser k t) =
  (input: slice) → (pos: U32.t) → ST t
  (requires (fun m → valid(p, m, input, pos)))
  (ensures (fun m res m' → m ≡ m' ∧ res == contents(p, m, input, pos)))
```

### 4.5 Low-Level Writers

We finally describe our low-level API for serializing a message in an output buffer. Our goal is to avoid intermediate allocations and copies; for example, our high-level hello function constructs the whole message before serializing. To this end, QuackyDucky generates families of low-level writers and auxiliary functions that take as parameter an output slice (that is, a buffer and a length) and a write position, modify the buffer between this position and the end of the buffer, and return a new write position. These functions either require that the output buffer is large enough (based on parser metadata) or may also return an error in case the buffer is too small.

In contrast to accessors, which enable random access to validated input in binary format, it is not generally possible to know in advance where to write data before writing any preceding variable-length data. Thus, our API assumes that data will be written sequentially, with the flexibility for the programmer to use an intermediate buffer whenever they choose to write data out of order. A notable exception is for encoding the lengths of variable-length data, which is usually known only after writing the raw data itself. To this end, QuackyDucky provides a *finalizer* that takes two positions in the output buffer, requires as a precondition that the buffer contain a placeholder for the length followed by a valid binary representation of the raw data, computes and writes its length, and ensures as a post-condition that the buffer now contains a valid variable-length representation of this data.

We illustrate these different cases on a low-level variant of the hello function, whose Low⋆ and extracted C code are shown in Figure 7. The first field, version, is an enumeration formatted in a fixed two-byte format: it is directly written using the QuackyDucky writer for protocolVersions. The second field, random, is a fixed-length, previously-allocated bytestring that can be copied from the configuration. Omitting intermediate fields, which may be handled similarly, the last field is a complex list of extensions. The list itself is written by repeatedly calling the extension writer on each element, after skipping the 2 bytes required for their total length. The computed length of the list is finally written by calling a QuackyDucky finalizer.

As a cumulative post-condition of all these steps, we know that the output buffer now contains the concatenation of a valid binary format for each of its fields, and we can conclude that it thus also contains a valid representation of a clientHello message by calling the *validity lemma* clientHello_valid also generated by QuackyDucky and verified by F⋆. The call to this lemma is erased before extraction to C code.

As an important simplification, our sample code requires as static precondition that the output buffer be large enough to hold any valid clientHello message. In more realistic code, one would need to dynamically check this length. (Each of these writer functions are fail-safe, but their errors still need to be propagated.)

```
let write_extension_list cfg output pos = ...
  (* write a list of extensions computed from the configuration *)

let write_hello cfg output pos =
  (* write 2 bytes of protocol version *)
  let pos_after_protocol_version =
    write_protocolVersion output pos TLS_1p3 in
  (* copy 32 bytes from the configuration *)
  memcpy cfg.random 0ul out.base pos_after_protocol_version 32ul;
  let pos_after_random = pos_after_protocol_version + 32ul in
  (* similarly write or copy the other fields *)
  let pos_after_session_id = ... in
  let pos_after_ciphersuites = ... in
  let pos_after_compressions = ... in
  (* leave two bytes for the total length of the extension list *)
  let pos_list = pos_after_compressions in
  (* calls an auxiliary function to write the extension list in-place *)
  let pos_after_extensions =
    write_extension_list cfg output pos_list in
  (* computes and writes the extensions length at pos_after_compressions *)
  finalize_clientHelloExtensions
    output pos_after_compressions pos_after_extensions;
  (* call the validity lemma for the clientHello message *)
  let m = get () in clientHello_valid m output pos;
  (* return the final position *)
  pos_after_extensions
```

```
uint32_t write_extension_list(config cfg, slice output, uint32_t pos);
uint32_t write_hello(config cfg, slice output, uint32_t pos)
{
  uint32_t pos_after_protocol_version =
    write_protocolVersion(output, pos0, TLS_1p3);
  memcpy(cfg.random + 0, output.base + pos_after_protocol_version, 32);
  uint32_t pos_after_random = pos_after_protocol_version + 32;
  ...
  uint32_t pos_list = pos_after_compressions + 2;
  uint32_t pos_after_extensions =
    write_extension_list(cfg, output, pos_after_compressions);
  finalize_clientHelloExtensions
    (output, pos_after_compressions, pos_after_extensions);
  return(pos_after_extensions);
}
```

Figure 7: Sample Low⋆ code for writing a TLS client hello (above) and its translation to C (below).

## 5 LowParse: Secure Parser Combinators

As we have seen in §4.2, QuackyDucky produces parser implementations by composing basic parsers using *combinators*, which are higher-order functions on parsers. For example, a combinator for pairs may take parsers for types $t$ and $u$ and yield a parser for type $t \times u$. Its implementation may first parse a message of type $t$, then parse a message of type $u$.

LowParse is our library of parser combinators, based on the long tradition of *monadic* parser combinators [22] in the functional programming community. However, LowParse is unique in that it is tailored to support the verification of non-malleable, correct parsers. We focus on combinators at the

| QuackyDucky Syntax | Data Type | Parser Combinator |
|---|---|---|
| uint$N$, $N \in \{8,16,32,64\}$ | Unsigned integer within $0..2^N-1$ | parse_u$N$ |
| t[$N$], $N \in \mathbb{N}$ | Fixed-size array of $t$s of length $N$ | plist[p] truncN |
| t<$M$..$N$> | List of $t$s, of variable length $M..N$ | vldata(plist[p], M, N) |
| t{$M$..$N$} | List of $t$s of variable element count $M..N$ | $(\mathsf{parse\_u}k \mathsf{\ filter\ } (n \mapsto M \le n \le N)) \triangleright (n \mapsto p^n)$ where $k = 8 \times \log_{256} N$ |
| struct{$t_1\ x_1$; …; $t_n\ x_n$;} | Record with $n$ fields named $(x_i)$ of type $(t_i)$ | $(p_1 \times \cdots \times p_n)$ synth $((v_1,\ldots,v_n) \mapsto \{x_1 = v_1; \ldots; x_n = v_n\})$ |
| struct {…; uint$N$ $x$; $t$ $y$[$x$]; …} | Variable-length field $y$ prefixed by its length $x$ | $\mathsf{vldata}(p, 0, 256^{N/8}-1)$ |
| enum {$E_1(N_1)$, …, $E_n(N_n)$, ($M$)} | Constant integer enumeration (with maximal value $M = 2^N-1$) | $\mathsf{penum}(\mathsf{parse\_u}N, [(E_1,N_1); \ldots; (E_n,N_n)])$ |
| struct {$t$ $x$; select($x$) { case $E_1$: $t_1$; …; case $E_n$: $t_n$ } $y$} | Tagged union ($t$ must be an enum type) | $p \triangleright_f q$ where $f(E_i,x) = E_i$ and $q(E_i) = p_i$ synth $(y \mapsto (E_i,y))$ |

Figure 8: The QuackyDucky input language and the corresponding LowParse combinators: everywhere in this table, $p_i$ is the parser for type $t_i$. All lengths are counted in bytes except otherwise mentioned.

specification level and their security properties, then discuss more briefly their implementations. For each specification combinator, we prove non-malleability and inverse properties; for each implementation combinator, we prove both safety and correctness. All properties are verified by typing the library.

Figure 8 summarizes the QuackyDucky input language and the corresponding LowParse combinators. We designed QuackyDucky and LowParse in a modular way, making it easy to extend the surface syntax of QuackyDucky by providing additional combinators. For instance, the t x{$M$..$N$} syntax for variable-size lists prefixed with their number of elements is a late addition to support the Bitcoin application in §6.2 but is not required for TLS.

We first define the properties attached to the specifications of §4.2. We prove a stronger version of non-malleability than the one given in §2.1, extending the definition there to handle parsers that may not consume all their input.

**Definition 1** *A parser $p$ for type $t$ is* non-malleable *if, whenever it succeeds and returns the same parsed value on two inputs, it also returns the same number of consumed bytes, and the two inputs coincide of these bytes.*

We also rely on the following no-lookahead property:

**Definition 2** *A parser $p$ has the* strong prefix property *when, if it succeeds on an input and consumes $\ell$ bytes, then it returns the same result on any inputs with the same first $\ell$ bytes.*

For a serializer to exist for a format that requires concatenating two value representations valid with respect to two parsers $p_1, p_2$ (such as pairs, lists, tagged unions, or variable-length data), $p_1$ is required to have the strong prefix property. Consider for instance serializing a pair of two pieces of data $x_1, x_2$ using serializers $s_1, s_2$ correct with respect to parsers $p_1, p_2$. We would like to prove that the serialization $s(x_1,x_2) = s_1(x_1) \cdot s_2(x_2)$ obtained by concatenating the two serializations, is correct with respect to the parser for pairs. By correctness of $s_1$, $p_1(s_1(x_1))$ succeeds and returns $(x_1, |s_1(x_1)|)$, but this is not enough to know that $p_1(s_1(x_1) \cdot s_2(x_2))$ succeeds and also returns $(x_1, |s_1(x_1)|)$ (so that we can cut the input after $|s_1(x_1)|$ and apply $p_2$ on the remainder, $s_2(x_2)$), unless $p_1$ has the strong prefix property.

These properties are included in the definition of parser_prop on the metadata generated by QuackyDucky, hence enforced by typing for all its parser specifications.

## 5.1 Specification Combinators

LowParse is an extensible library of combinators. For each parser specification combinator, we attach a corresponding metadata combinator; then, we define, when possible, a serializer combinator.

**Parser combinators** We define primitive parser combinators below. For each of them, we prove injectivity and any relevant additional properties indicated in their metadata, such as the strong prefix property. We also define derived combinators; in contrast, all their properties are established automatically as the result of their definitions (by type unification and matching on their metadata). The code produced by QuackyDucky only inserts annotations to prove their composability conditions, for instance, by computing the length boundaries of the derived metadata, which are then verified by F$^\star$.

We start by defining primitive parser combinators: fail, which consumes no input and fails; ret[$x$], which consumes no input and succeeds returning $x$; read_byte, which consumes and returns a single byte of input; and and_then, which sequentially composes two parsers where the second parser depends on the value parsed by the first parser (i.e., monadic composition). For each of these basic combinators, we prove non-malleability and/or the strong prefix property under suitable conditions. For instance, $p$ and_then $q$ has the strong prefix property provided that $p$ has it, $q[x]$ has it for all $x$, and, moreover, if $q[x_1]$ and $q[x_2]$ succeed on inputs $b_1$ and $b_2$, respectively, and return the same value, then $x_1 = x_2$. (Otherwise, consider for example $p =$ read_byte and $q =$ ret[0].)

Using those primitive combinators, we define derived combinators, for which verification of non-malleability and metadata correctness automatically follows by typing. Given parsers $p_0$ and $p_1$ for $t_0$ and $t_1$, respectively, we can derive a parser for pairs of type $t_0 \times t_1$ using $p \times q$; mapping functions over parsed results using $p\,\mathsf{synth}\,f$; filtering parsed results by some predicate using $p\,\mathsf{filter}\,f$; etc. proving non-malleability and the strong prefix property for them under suitable conditions.

More specifically, we derive parsers for fixed-length machine integers, and we prove their non-malleability for both endiannesses. For instance, we define little-endian 16-bit parsing as $(\mathsf{read\_byte} \times \mathsf{read\_byte})\,\mathsf{synth}\,((x,y) \mapsto x + 256 \times y)$.

Our next combinators support variable-length data and lists:

- Given a parser $p$ for type $t$, the parser $\mathsf{plist}[p]$ is defined by repeatedly applying $p$ to its input. It fails as soon as $p$ fails or consumes zero bytes. If succeeds when $p$ eventually consumes its whole input and then returns the resulting list of values.
- Given a parser $p$ for type $t$ and $n > 0$, the parser $p\,\mathsf{trunc}\,n$ succeeds when $p$ succeeds on its input truncated to its first $n$ bytes and consumes exactly $n$ bytes.

The parser $\mathsf{plist}[p]$ does not have the strong prefix property, but it consumes all its input. The parser $p\,\mathsf{trunc}\,n$ always has the strong prefix property, even if $p$ does not. If $s$ is a correct serializer for $p$ at type $t$, then $p\,\mathsf{trunc}\,n$ is a parser for type $x : t\{|s(x)| = n\}$ and $s$ is its correct serializer at that type.

We finally present further derived combinators, whose properties are automatically verified by construction:

*Tagged unions:* if $p$ is a parser for type $t$ and $f : u \to t$ and $q[x]$ is a parser for type $(y : u\{f(y) = x\})$ for every $x : t$, then:

$$p \triangleright_f q = p\,\mathsf{and\_then}\,(x \mapsto q[x]\,\mathsf{synth}\,(y \mapsto y))$$

is a parser for type $u$. This combinator is a strengthening of and_then that enforces non-malleability of $q$ by making its codomain dependent: $u$ is the union type, and $t$ is the tag type, and $f$ gives the tag of an element of the union type. From there, we define a combinator for *sum types*, which can be used for tagged unions.

*Enum types:* if $l$ is a list of key-value pairs where each key and each value only appear once, then it defines both a closed enum type (whose elements are the keys that appear in $l$) and an open enum type (whose elements are the known keys that appear in $l$ and the unknown values that do not appear in $l$). We define parsers for both variants $\mathsf{penum}(p,l)$ (where $p$ is the value parser), using filter, synth and the dictionary function on key-value pair lists.

*Variable-sized data:* formats such as TLS often specify variable-length data as a payload prefixed by its size in bytes. If $p$ is a parser for the payload, and if $s$ is a serializer correct with respect to $p$, then we define

$$\begin{aligned}\mathsf{vldata}(p,l,h) = \quad & \mathsf{parse\_u}_\ell \circ \mathsf{filter}(n \mapsto l \le n \le h) \\ & \triangleright_f (n \mapsto p\,\mathsf{trunc}\,n)\end{aligned}$$

as a parser for the refined type $(x : t\{l \le |s(x)| \le h\})$, where $\ell = 8 \times |\log_{256}(h)|$, is the bit size of the size integer prefix, and $f(x) = |s(x)|$. Such parsers inherit the strong prefix property from the parser for the prefix size, regardless of whether it holds for $p$.

**Correct Serializers** Not all parser specifications have correct serializers. For instance, $\mathsf{ret}[x]$ and and_then do not have a generic serializer. So, in LowParse, we provide serializer combinators for read_byte, fail, plist, synth, and $\triangleright$, for each of which we prove correctness with respect to its corresponding parser combinator (i.e., that they are inverse of one another). We also easily prove that a correct serializer for $p$ is also correct for $p\,\mathsf{trunc}\,n$ and $p\,\mathsf{filter}\,f$ (once its domain is restricted accordingly). From there, we derive correct serializers for $\times$, nlist, vldata, etc. for which the correctness proof automatically follows by typing.

## 5.2 Implementation Combinators

For each parser-specification combinator, LowParse provides combinators for its high-level parser and for its low-level validators and jumpers (and similarly for serializers). For primitive combinators, we implement their corresponding validators jumpers and serializers; for each of them we prove memory safety and functional correctness with respect to their specification. We implement most derived combinators by following the same construction as for their specs, by assembling the corresponding implementation combinators. Thus, their memory safety and functional correctness automatically follow by typing. We also define accessor combinators for synth and tagged unions, and accessors for pair elements, from which QuackyDucky derives accessors for struct fields and sum types.

By design, our combinators are inherently higher-order and so they cannot directly be extracted to C. Instead, we rely on meta-programming features of F$^\star$ and KReMLin, based on source code annotations, to ensure that all combinator code is inlined and specialized before extraction. In most cases, this is achieved by annotating our source code. In other cases, we extend LowParse with F$^\star$ tactics [31], pieces of F$^\star$ metaprograms written once and for all and evaluated at typechecking time to automatically generate Low$^\star$ validators from some type definitions. For example, our validators for enum values and tagged unions are specified using constant key-value lists. Instead of programming a loop on these lists, we meta-program their unrolling at compile-time, which yields a cascade of `if`s automatically turned into a `switch` by many C compilers. In rare cases, such as unions tagged with an enum value, we write additional validator combinators to more precisely control their inlining by F$^\star$ and KReMLin.

In addition, metadata allow us to provide some generic validator combinators that apply regardless of the actual parser combinator. For example, if we know that a parser consumes a constant $n$ bytes and always succeeds, then we can use a

| | QD | F* LoC | Verify | Extract | C LoC | Obj. |
|---|---|---|---|---|---|---|
| TLS | 1601 | 69,534 | 46m | 25m | 192,229 | 717KB |
| Bitcoin | 31 | 1,925 | 1m56s | 1m14s | 1,344 | 8KB |
| PKCS #1 | 117 | 4,452 | 2m14s | 2m39s | 3,368 | 26KB |
| LowParse | N/A | 32,210 | 3m5s | 1m5s | 185 | 739 B |

Table 1: Overview of EverParse Applications

validator that just jumps $n$ bytes. QuackyDucky selects these combinators based on the metadata it computes.

## 6 Integration and Evaluation

We evaluate the integration of EverParse-generated parsers for three applications: the TLS message format, integrated into MITLS; the Bitcoin block and transaction format, integrated into the Bitcoin Core benchmark; and the ASN.1 payload of PKCS #1 signatures, integrated into mbedTLS.

Table 1 shows for each application the lines of QuackyDucky input specification, the amount of F* code generated, the time required for verification and KReMLin extraction, and the size of the C code and compiled objects. The Bitcoin evaluation was performed on a 28-core Xeon E5-2680 v4 CPU with 128GB of RAM, running with turbo boost and all but one core disabled. The rest of the figures were collected on a 10-core Xeon W-2155 CPU with 128GB of RAM, running F* commit 7b6d77 with Z3 4.5.1 and GCC 7.4.

### 6.1 TLS Message Format

As described in §3, we have specified the TLS message format for all versions of TLS from 1.0 to 1.3. However, integrating the generated parsers presents some major challenges: implementations tend to define their own representations of messages, with field and tag names that differ from the RFC, and some of them like mbedTLS interleave the parsing and processing of messages. MITLS [7] uses functional, high-level parser implementations and types, operating on values. Most of the basic data types (such as cipher suite names) are defined in a module called `TLSConstants`, while some specialized ones scattered in other modules (e.g. group names in `CommonDH`). Extension types and parsers are in the `Extensions` module, while message types and parsers are in `HandshakeMessage`. We noticed that these files contain many assumptions and incomplete proofs, many of which have been completed for earlier drafts of TLS 1.3, but not updated as the formats changed (with EverParse, such updates and extensions only require a few changes to the format description).

In total, in order to switch to the high-level implementation produced by QuackyDucky, we update or rename over 200 types (and propagate these changes), which requires 2,865 additions and 3,266 deletions over 38 files (according to our Github pull request). Unlike LowParse, MITLS individually proves the non-malleability of each parser as a lemma separate from parser definitions instead of a refinement; the MITLS proofs for such lemmas are lengthy and intricate. So, we define a `LowParseWrappers` module to replace such proofs with a uniform call to LowParse parser property lemmas. Our changes do not break other existing proofs, but several generated types are more precise than the handwritten ones (notably, all lists are refined to ensure they can be serialized), which leads to additional conditions to prove in many functions. The generated parsers are also a lot stricter: for instance, we now check at parsing which extensions can appear in a message, and which messages can appear for the negotiated version.

To test the impact of EverParse parsers, we run the simple HTTP client and server tool distributed with MITLS to compare how many requests can be served, using the default algorithm choices. This tool is not optimized for production and processes requests sequentially. We compare the time to process 500 requests between the original MITLS parsers and EverParse high-level parser implementations.

| | MITLS | MITLS-EverParse |
|---|---|---|
| HTTP requests | 49.8 req/s | 53.3 req/s |

Integrating the low-level Low* implementations into MITLS requires a large effort, as many functions that are currently pure (operating on values such as lists) become stateful (the buffer that contains the valid positions matching each value must be live). To anticipate the benefits of this effort, we run a synthetic benchmark that validates all messages from a public dataset of TLS handshakes published by Lumen [39]. This dataset contains handshake produced by a wide range of clients and servers, and contains over 13GB of data (including the BSON overhead). As a baseline, we compare in-place validation time with the cost of checking the message length, allocating a buffer of the message size, and copying the contents of the message in the buffer.

| Memcpy | | EverParse | |
|---|---|---|---|
| 1,864 MB/s | 1.761 cy/B | 2,684 MB/s | 1.177 cy/B |

### 6.2 Bitcoin Blocks and Transactions

To show that EverParse is extensible and evaluate the performance of its low-level parsers, we implement the Bitcoin block and transaction format, listed in Figure 9. We do not implement Segregated Witness ("segwit"), an extension that overloads the semantics of a length in the block format to conditionally add a new field to the block structure, because it requires a very ad-hoc combinator. Bitcoin requires two LowParse extensions: one for the encoding of "compact integers" (`bitcoin_varint`), and one for lists prefixed by their size in elements rather than in bytes.

For compact integers, the representation may either use 1, 3, 5, or 9 bytes depending on whether the value of the first byte is respectively less than 252, 253, 254, or 255. It is not clear from the Bitcoin documentation and wiki that the format of compact integer is not malleable (e.g. 4636

```
opaque sha256[32];
struct {
  sha256 prev_hash; uint32_le prev_idx;
  opaque scriptSig<0..10000 : bitcoin_varint>;
  uint32_le seq_no;
} txin;
struct {
  uint64_le value;
  opaque scriptPubKey<0..10000 : bitcoin_varint>;
} txout;
struct {
  uint32_le version;
  txin inputs{0..1000 : bitcoin_varint};
  txout outputs{0..11110 : bitcoin_varint};
  uint32_le lock_time;
} transaction;
struct {
  uint32_le version;
  sha256 prev_block; sha256 merkle_root;
  uint32_le timestamp;
  uint32_le bits; uint32_le nonce;
  transaction tx{0..2^16 : bitcoin_varint};
} block;
```

Figure 9: QuackyDucky specification of Bitcoin blocks

could be represented as `fd121c`, or `fe0000121c`). However, we checked that the Core implementation enforces the shortest representation in the `ReadCompactSize` function. Additionally, we allow list types to specify in their range the type of integer used to encode the prefix length or size (e.g. `txin inputs{0..2^14 : bitcoin_varint}`). A drawback of prefixing lists by their number of elements is that the theoretical maximum length of the formatted list can get extremely large. For instance, the maximal size of a well-formed Bitcoin block is over $2^{320}$ bytes (in practice, it is well-known that non-segwit blocks are at most 1MB). To avoid overflowing OCaml's 63-bit arithmetic in the parser metadata length computations in QuackyDucky, we must write more conservative boundaries. Scripts are known to be at most 10,000 bytes. Historically, all non-segwit blocks in the main chain contain less than $2^{16}$ transactions (although the maximum is higher). It is more difficult to bound the number of inputs and outputs of a transaction. If we assume a transaction is standard (at most 100,000 bytes) and all inputs are signed (their script is at least 64 bytes), there are less than 1000 inputs. Since outputs can be as short as 9 bytes, a transaction can have over 11000. Our test data is blocks 100,000 to 110,000 of the Bitcoin blockchain, totaling 21MB. To experimentally check those assumptions, we parsed all of these blocks and confirmed they are accepted by our validator.

For benchmarking, we measure: first, the performance of our zero-copy block validator compared with the built-in deserialization function of the Bitcoin Core client (commit cbe7efe); second, the performance variations of our zero-copy block validator across compilers and optimization levels; third, the performance impact of fine-grained code-generation options passed to KReMLin (Figure 10). For each one of those benchmarks, we report numbers in kiB/s, i.e. the throughput; we only occasionally report cycles per byte since most of our validators run at less than 1 cy/B.

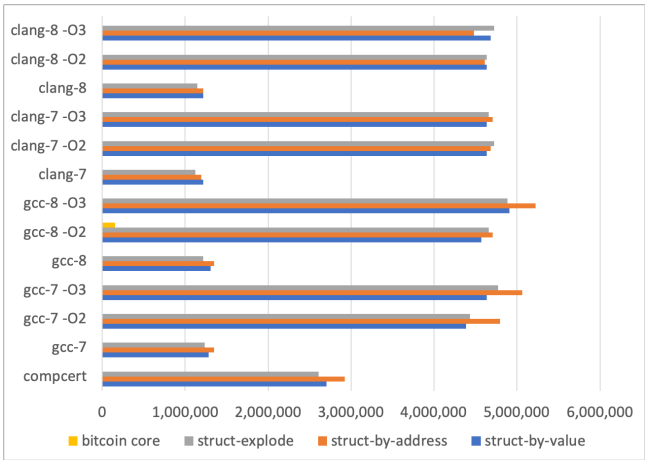


Figure 10: Synthetic performance comparison for validating 10,001 Bitcoin blocks. Throughput in kiB/s, higher is better.

The first measurement compares the performance of our code against a reference implementation, namely, Bitcoin Core. Bitcoin uses a custom template for serializing C++ objects. This template is well-optimized and tries to rely on casts and the in-memory representation of base types as much as possible. However, it is not zero-copy: parsing relies on the memory allocated for the C++ object, and serialization requires a copy to the output buffer. The benefit is that the data can be accessed using standard data structure libraries such as `std::vector` for lists. Bitcoin provides a built-in benchmarking tool for many of its features, including block deserialization and validation in `src/bench/checkblock.cpp`.

We modify this benchmark to use our test data of 10,001 blocks and deserialize all of them in each run. The benchmark deserializes 130 times, and reports the median over 5 runs. We keep the default compiler options (gcc-8, O2 optimization level). The measured throughput is 152,786 kiB/s, which translates to 15 cy/B on the test machine used for the Bitcoin measurements. We then validate the same 21MB of data using our validators, with the same compiler and optimization levels. We obtain a throughput of 4,568,632 kiB/s, which is less than a cycle per byte, for the default KReMLin configuration.

While the validation performance of our code is excellent, we do not claim that this benchmark is representative of real application usage, as it doesn't account for the overhead of accessor functions to read the block and transaction contents. Nevertheless, this shows that our verified low-level implementation is competitive with hand-optimized formatters.

Second, we measure performance variations across compiler versions. The performance is comparable between the two most recent versions of Clang and GCC, for optimization levels O2 and O3. Unsurprisingly, the default setting without optimization yields much slower code, but even then, we remain considerably faster than the original Bitcoin code. We also measure the performance of our code compiled with CompCert [28]. We find that CompCert is consistently 42% slower than GCC and Clang with optimizations, but still more than twice as fast as GCC or Clang without optimizations. We conclude that our code lends itself well to optimizations by modern compilers, and that users do not need to enable the (risky) O3 performance level to get maximum performance out of Clang or GCC.

Third, we experiment with various compilation schemes of the KReMLin compiler for the core type `LowParse.slice.slice`, a two-field C struct (representing a byte buffer through its base pointer and length) which in the default configuration is passed by value (§4.4). Two alternate compilation schemes are considered. First, passing both the base pointer and length as separate arguments to functions; this is the "struct-explode" category, and yields no performance improvement. Second, we pass those structures by address, relying on an unverified transformation in the KReMLin compiler, similar to CompCert's `-fstruct-passing` feature. This yields modest performance improvements for GCC 7 and GCC 8 at the higher optimization levels (3% to 9%). We conclude that our generated C code is satisfactory and that we don't need to either rewrite our code to pass slices by address (a substantial proof burden) or instruct KReMLin to perform this transformation (which would increase the trusted computing base).

Finally, we perform fuzz testing on the X64 machine code of our generated bitcoin-block validator as compiled with gcc-8 -O2. (Although our verification results ensure memory safety for all inputs, fuzzing may still, in principle, detect bugs in our toolchain and the C compilers we use.) We use SAGE [19], a fuzzer specialized to parsers, which generates random input, valid or not, and feeds them to the validator which SAGE automatically instruments to check for buffer overflows. As expected, SAGE reported no bugs after 21,664,448 inputs tested at an average rate of 599 inputs per minute.

### 6.3 ASN.1 Payload of PKCS #1 Signatures

Our last example is the payload of PKCS #1 signatures introduced in § 2.2. We extend LowParse with a combinator for the encoding of ASN.1 DER lengths. This encoding is particularly convoluted: if a length is less than 127, it is represented over a single byte. Otherwise, the 7 least significant bits of the first byte encode the length in byte of the shortest big endian representation of the length. This means the length can be at most $2^{2^{1016}} - 1$. To avoid overflows, we only support values of the first byte less than 132 (i.e. 32-bit lengths). An issue with the specification is the lack of dependency between the object identifier of the hash algorithm and the octet string of the actual digest: the application is required to check the digest is of the correct length if it tries to parse the signature contents. We capture this dependency by only making the outermost sequence variable length, and by parsing the object identifier as a constant tag of an union of fixed-length arrays. (Note that this is for illustration only, the recommended approach is to serialize the computed hash, and use a constant time comparison with the un-padded signature contents instead).

We integrate our code into `pkcs1_v15_verify` function of mbedTLS, and modify the built-in benchmarking tool to measure the PKCS #1 signature verification time instead of the raw public key and private key operation time measured by default. In addition, we also export the internal function to format the ASN.1 payload of the signature (`pkcs1_v15_encode`), and compare it with our extracted formatter functions. The following table compares the amount of operations per second and cycles per operation for complete signature verification, and for the encoding of the ASN.1 payload:

| Operation | mbedTLS | | EverParse | |
|---|---|---|---|---|
| Verify | 79K op/s | 5,700 cy/op | 79K op/s | 5,649 cy/op |
| Encode | 31M op/s | 14 cy/op | 134M op/s | 3 cy/op |

As expected, the verification time is dominated by the cost of the RSA exponentiation: even though our validator is over 4 times faster and avoids the allocation of a modulus-sized intermediate buffer to compare the expected and computed digests, the impact on overall validation performance is negligible. For signing and encoding, the constant constant parts of the signature payload must be written manually, and separate finalizers must be called for to write the bytes we depend on for the algorithm choice and the outermost ASN.1 length.

We tested our implementation against all variants of the Bleichenbacher's attack listed in §2.2 and confirmed they are properly rejected.

## 7 Related work

Parsing combinators are widely used in functional programming languages, and there exist several libraries for network protocols [29], including TLS and X.509 [30].

For well-behaved language classes (e.g. regular, context-free), there is a long history on verification of parser correctness with respect to simple specifications (regular expressions, grammars). Jourdan et al. [25] propose a certifying compiler for LR(1) grammars, which translates the grammar into a pushdown automaton and a certificate of language equivalence between the grammar and the automaton. The certificate is checked by a validator verified in Coq [1], while the automaton is interpreted by a verified interpreter. Barthwal et al. [3] propose a verified grammar compiler and automaton interpreter for the simpler class of SLR languages, verified in HOL [42]. For regular languages, Koprowski et al.

introduced TRX [26], an interpreter for regular expressions verified in Coq. All of these works require runtime interpretation, which greatly degrades the performance compared to compilation. Furthermore, they target garbage-collected functional language runtimes like OCaml, which cannot easily be integrated into high-performance, native C applications.

For TLV languages, there have been some attempts [2] to create context-free or even regular specifications for X.509. However, due to the context-sensitive nature of ASN.1, these efforts rely on discretizations of some fields (such as variable-length integers) and drastic simplifications of the format (such as limiting the choice of extensions to a known subset). The combinatorial explosion required to achieve interoperability makes these approaches impractical for real implementations, although some authors claim otherwise [21].

For runtime safety, fuzzing techniques [19, 43] are widely deployed and often included into test suites for cryptographic libraries. Although best practice, fuzzing is by nature incomplete, and may be difficult to apply to authenticated messages (as fuzzing invalidates hashes, signatures and MACs). Dynamic analysis tools like Valgrind [35] or AddressSanitizer [44] are widely used but also incomplete, while static analysis tools like Frama-C [11] require higher expertise, a significant time investment, and tend to scale poorly with large codebases. Because of past attacks, specific tools have been created for TLS and cryptographic libraries, including TLS-Attacker [45], FlexTLS [5], and Wycheproof [8], but their focus is to uncover known vulnerability patterns in protocol implementations rather than prove formal guarantees on their message formats.

Another related line of work [10, 16] applies abstract interpretation and symbolic execution to study the properties of parsers, such as whether two implementations of a format accept the same message. These techniques can be applied to existing implementations, but cannot generate new ones.

Narcissus [47] also constructs correct binary parsers from a verified library of combinators written in Coq. There are two major differences with EverParse: first, Narcissus only proves the correctness of its parsers, while we also prove parser security; second, Narcissus only generates higher-order, functional implementations while our compiled approach means that our parsers are entirely specialized at F⋆ extraction, and can be compiled in zero-copy mode. Building on Narcissus, Ye and Delaware [51] build a verified compiler in Coq for parsers and formatters described using Protocol Buffers [20]. Like EverParse, their parsers and formatters are proven to be correct. Their library produces high-level functional code, which is memory-safe by construction—in contrast, EverParse produces low-level C code, together with memory safety proofs. Further, due to the inherent structure of the Protocol Buffers format, their work does not consider non-malleability.

Jim and Mandelbaum [23, 24] have formalized and developed parser generators for a wide class of context-free grammars extended with data dependency, including tag-length-value encodings, tagged unions, and other forms of dependence supported by QuackyDucky. They also provide tooling, like QuackyDucky, to automatically extract message format descriptions from RFCs and have applied their work to network message formats like IMAP, the popular mail protocol. While the input language of their framework is significantly more expressive than ours, EverParse, in contrast, produces provably safe, secure and functionally correct parsers. Jim and Mandelbaum also do not address message formatting.

## 8 Limitations and Future Work

**Trusted computing base**: we statically guarantee at the F⋆ source level memory safety, functional correctness, and non-malleability for all code generated by QuackyDucky. Preserving non-malleability down to machine code requires only preserving functional correctness, since non-malleability is a specification-level guarantee. All our verification results, including preservation of memory safety and functional correctness down to machine code, relies on a trusted computing base (TCB) that includes:

- the F⋆ proof assistant and the Z3 theorem prover, although work by Swamy et al. [48] provides a model of a subset of F⋆ and proves its soundness;
- the KReMLin compiler from Low⋆ to C, although work by Protzenko et al. [38] provides a model of a subset of Low⋆, its compilation to CompCert Clight, and proofs (on paper) that compilation to C preserves memory safety and functional correctness;
- the C compiler, although one can use the CompCert [28] verified C compiler, which ensures the preservation of memory safety and functional correctness, at the expense of some performance.

This trusted base is comparable to Coq-based verified implementations, which trust Coq, the Coq extraction to OCaml, and the OCaml compiler and runtime. Ongoing research aims to reduce this TCB by verifying Coq extraction; similar efforts could, in principle, be applied to F⋆ and KReMLin.

Conversely, neither LowParse nor QuackyDucky are in the TCB. LowParse is fully verified. The input format specification of QuackyDucky is trusted for liveness, but not for security: if there is a mistake in the format specification, the worse that can happen is that the generated messages are incompatible with implementations of the correct format. We rely on interoperability testing to detect such mistakes. Conversely, EverParse can be used during the standardization of a new message format, as it can prove that the specification is secure regardless of the generated implementation.

**Expressiveness** QuackyDucky currently focuses on supporting tag-length-value encodings of non-malleable data formats. We show that the message formats of several important protocols and standards, including TLS, PKCS #1 signature payloads and Bitcoin, fall into this class. LowParse,

being the target language of QuackyDucky's translation, is also currently restricted to supporting non-malleable data formats. However, it would be straightforward to make non-malleability conditional on a flag set in the parser metadata in order to define combinators for zero-copy malleable formats, including MessagePack, CBOR, Apache Arrow, Cap'n proto, and Protocol Buffers which are malleable at least by default (some have canonical representation rules). Generalized to support malleable formats, LowParse, being a library of verified monadic parser combinators, would support parsing with arbitrary data dependence and lookahead, beyond the class of context-free languages—however, coming up with *efficient* verified implementations of parsers for such language classes is an open question. In the future, we will also consider generalizing QuackyDucky to target the class of languages supported by LowParse.

**Side-channel attacks**: the implementation produced by EverParse branches on values read from the input buffer, which may leak (through timing side-channels) information when used on confidential data. We may in principle verify properties such as constant-time execution for the processing of simple message formats, reusing F$^\star$ and KReMLin techniques and libraries for side-channel protection of cryptographic algorithms. For example, we may provide constant-time combinators for fixed-length secret bytestrings. We leave such extensions for future work.

**Fuzzing**: since we expect our extracted C code to be compiled by unverified toolchains (such as GCC and LLVM, with optimizations), fuzz testing can provide additional assurance that the compilation from F$^\star$ to binary does not break our verified safety properties. We started using fuzzers optimized for parsers, such as SAGE [19], to fuzz the generated bitcoin block validator; we plan to extend their use to fuzz application code that uses generated validators and accessors.

**Integration**: we have integrated the high-level implementation of EverParse TLS parsers into MITLS, but our goal is to transition to the low-level implementation, thus avoiding many unnecessary heap allocations and copies. This is a major step towards making MITLS practical in performance-sensitive deployments.

## 9 Conclusion

Developers should prefer the convenience and robustness of writing high-level format specifications compiled by parser generation tools to programming tedious and error-prone custom parsers, although the latter is sometimes required for performance reasons. EverParse offers a unique combination of high performance, zero-copy implementations and high-assurance formal verification of the generated parsers.

**Acknowledgments** We thank the anonymous reviewers and Prateek Saxena for their helpful comments, which improved the writing of this paper. We thank Barry Bond, Christoph Wintersteiger and the Everest team for their help in testing EverParse. We thank Clément Pit-Claudel and Benjamin Delaware for insightful discussions on the goals of verified parsing. Tej Chajed and Nadim Kobeissi completed their work during internships at Microsoft Research.

## References

[1] The Coq proof assistant. http://coq.inria.fr, 1984–2019.

[2] A. Barenghi, N. Mainardi, and G. Pelosi. Systematic parsing of X.509: eradicating security issues with a parse tree. *CoRR*, abs/1812.04959, 2018.

[3] A. Barthwal and M. Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

[4] D. Benjamin. Applying GREASE to TLS extensibility. IETF Draft, 2016.

[5] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan. FLEXTLS: A tool for testing TLS implementations. In *Usenix Workshop on Offensive Technologies (WOOT15)*, 2015.

[6] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pages 1:1–1:12, 2017. https://project-everest.github.io.

[7] K. Bhargavan, C. Fournet, and M. Kohlweiss. miTLS: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, Nov 2016. https://github.com/project-everest/mitls-fstar.

[8] D. Bleichenbacher, T. Duong, E. Kasper, and Q. Nguyen. Project Wycheproof: Scaling crypto testing. In *Real World Crypto Symposium, New York, USA*, 2017.

[9] S. Y. Chau. The OID parser in the ASN.1 code in GMP allows any number of random bytes after a valid OID. Available from MITRE CVE-2018-16151, 2018.

[10] P. Cousot and R. Cousot. *Grammar Analysis and Parsing by Abstract Interpretation*, pages 175–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[12] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[13] A. Delignat-Lavaud. RSA signature forgery attack in NSS due to incorrect parsing of ASN.1 encoded DigestInfo. MITRE CVE-2014-1569, 2014.

[14] T. Dierks and C. Allen. The TLS 1.0 protocol. IETF RFC 2246, 1999.

[15] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. IETF RFC 5246, 2008.

[16] K.-G. Doh, H. Kim, and D. A. Schmidt. *Abstract LR-Parsing*, pages 90–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] Z. Durumeric, F. Li, J. Kasten, J. Amann, et al. The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference*, pages 475–488. ACM, 2014.

[18] H. Finney. Bleichenbacher's RSA signature forgery based on implementation error, 2006.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[20] Google. Protocol buffers. github.com/protocolbuffers.

[21] R. D. Graham and P. C. Johnson. Finite state machine parsing for internet protocols: Faster than you think. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 185–190. IEEE, 2014.

[22] G. Hutton. Higher-order functions for parsing. *Journal of functional programming*, 2(3):323–343, 1992.

[23] T. Jim and Y. Mandelbaum. Efficient earley parsing with regular right-hand sides. *Electr. Notes Theor. Comput. Sci.*, 253(7):135–148, 2010.

[24] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Programming Languages and Systems - 20th European Symposium on Programming (ESOP)*, pages 378–397, 2011.

[25] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

[27] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. 2001.

[28] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[29] O. Levillain. Parsifal: A pragmatic solution to the binary parsing problems. In *2014 IEEE Security and Privacy Workshops*, pages 191–197, May 2014.

[30] A. Madhavapeddy and D. J. Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[31] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming*, 2019.

[32] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 62–72, 10 2012.

[33] MongoDB. BSON. http://bsonspec.org/.

[34] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[35] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[36] G. Neufeld and S. Vuong. An overview of ASN.1. *Computer Networks and ISDN Systems*, 23(5):393–415, 1992.

[37] Y. Oiwa, K. Kobara, and H. Watanabe. A new variant for an attack against RSA signature verification using parameter field. In J. Lopez, P. Samarati, and J. L. Ferrer, editors, *Public Key Infrastructure*, pages 143–153, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[38] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, Sept. 2017.

[39] A. Razaghpanah, A. Akhavan Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Tls handshake data collected by Lumen, Sept. 2017. https://haystack.mobi/datasets.

[40] E. Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 8446, 2018.

[41] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport layer security (TLS) session resumption without server-side state. IETF RFC 5077, 2008.

[42] N. Schirmer. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[43] K. Serebryany. OSS-Fuzz: Google's continuous fuzzing service for open source software. 2017.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Usenix Annual Technical Conference (ATC12)*, pages 309–318, 2012.

[45] J. Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[46] R. Srinivasan. XDR: External data representation. IETF RFC 1832, 1995.

[47] S. Suriyakarn, B. Delaware, A. Chlipala, et al. Narcissus: Deriving correct-by-construction decoders and encoders from binary formats. *arXiv preprint arXiv:1803.04870*, 2018.

[48] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM Symposium on Principles of Programming Languages*, pages 256–270, 2016. https://www.fstar-lang.org.

[49] F. Valsorda. Bleichenbacher'06 signature forgery in Python-RSA, 2016.

[50] P. Wuille et al. BIP62: Dealing with malleability, 2014.

[51] Q. Ye and B. Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 222–233, 2019.