Comparse: Provably Secure Formats for Cryptographic Protocols

Théophile Wallez
theophile.wallez@inria.fr
Inria
Paris, France

Jonathan Protzenko
protz@microsoft.com
Microsoft Research
Redmond, Washington, USA

Karthikeyan Bhargavan
karthikeyan.bhargavan@inria.fr
Inria and Cryspen
Paris, France

ABSTRACT

Data formats used for cryptographic inputs have historically been the source of many attacks on cryptographic protocols, but their security guarantees remain poorly studied. One reason is that, due to their low-level nature, formats often fall outside of the security model. Another reason is that studying *all* of the uses of *all* of the formats within one protocol is too difficult to do by hand, and requires a comprehensive, automated framework.

We propose a new framework, "Comparse", that specifically tackles the security analysis of data formats in cryptographic protocols. Comparse forces the protocol analyst to systematically *think* about data formats, formalize them precisely, and show that they enjoy strong enough properties to guarantee the security of the protocol.

Our methodology is developed in three steps. First, we introduce a high-level cryptographic API that lifts the traditional game-based cryptographic assumptions over bitstrings to work over high-level messages, using formats. This allows us to derive the conditions that secure formats must obey in order for their usage to be secure. Second, equipped with these security criteria, we implement a framework for specifying and verifying secure formats in the F* proof assistant. Our approach is based on format combinators, which enable compositional and modular proofs. In many cases, we relieve the user of having to write those combinators by hand, using compile-time term synthesis via Meta-F*. Finally, we show that our F* implementation can replace the symbolic notion of message formats previously implemented in the DY* protocol analysis framework. Our newer, bit-level precise accounting of formats closes the modeling gap, and allows DY* to reason about concrete messages and identify protocol flaws that it was previously oblivious to.

We evaluate Comparse over several classic and real-world protocols. Our largest case studies use Comparse to formalize and provide security proofs for the formats used in TLS 1.3, as well as upcoming protocols like MLS and Compact TLS 1.3 (cTLS), providing confidence and feedback in the design of these protocols.

1 INTRODUCTION

Modern software applications rely on a variety of cryptographic protocols to protect sensitive data as it is transmitted over or stored on insecure media. They use Transport Layer Security (TLS) or Noise when they need secure channels, FileVault or Bitlocker for disk encryption, Signal or Messaging Layer Security (MLS) for secure messaging, Bitcoin or Ethereum for distributed ledgers.

Each of these *protocols* can be described as a sequence of (one or more) high-level *messages* sent between (one or more) *participants*. At each step, a sender takes a message that has a particular meaning in the context of the protocol and encodes it into a bitstring by following a protocol-specific *format*. This bitstring is then protected using some *cryptographic construction*. For example, the sender may encrypt the bitstring to guarantee the *confidentiality* for the message content (and the *privacy* of its metadata); or they may add signatures, message authentication codes (MACs), or zero-knowledge proofs to guarantee the *integrity* and *authenticity* of the message. The output bitstring is then serialized according to a *wire format* before it is sent over the network or stored on some disk. The recipient follows the protocol in reverse, parsing the received bitstring, applying its own sequence of cryptographic operations, and decoding the result to obtain the high-level protocol message that the sender (hopefully) intended to send.

Attacks on Cryptographic Protocols. As a classic example, consider the core of the Needham-Schroeder public-key protocol [33]:

$$\begin{aligned} A \longrightarrow B: &\quad \{N_A \| A\}_{PK(B)} \\ B \longrightarrow A: &\quad \{N_A \| N_B\}_{PK(A)} \\ A \longrightarrow B: &\quad \{N_B\}_{PK(B)} \end{aligned}$$

Each participant generates a nonce (N_A, N_B), formats it along with some identity information ($N_A \| A$), and encrypts the resulting bitstring using a public-key. Note that the formatting of the message is done *within* the encrypted payload; any wire-formatting that is done outside the encryption (for example, a header mentioning the sender and recipient) is considered to be under the control of the network adversary, and so is ignored in the analysis of the protocol.

The protocol aims to authenticate two participants (A and B) to each other, and to establish a shared secret (N_B) between them. However, it has a famous attack, originally found by Gavin Lowe [27], which exploits the fact that the second message does not mention its recipient's name, allowing an attacker to mix messages across two sessions, breaking the security of the protocol. Adding B inside the encryption of the second message prevents the attack.

Interestingly, however, there is another, less-well-known attack on this protocol that relies on a message format ambiguity. Meadows [30] observed that an attacker could take the second message (from B to A) from one session and pass it off as a first message in a new session (seemingly from someone named N_B to A). In essence, the formats of the first two messages are *not disjoint*, since a priori, N_B may well be a valid identifier for a protocol participant.

Meadows finds two attacks that exploit this format ambiguity, and Heather et al. [20] show that Lowe's fix to the protocol does not prevent this attack. To fix the protocol against such format confusion attacks, it is necessary to change the internal formats and remove the ambiguity, say by systematically *tagging* each internal message by a distinct bitstring.

Formats in Real-World Protocols. Format specifications are ubiquitous in real-world protocols, since agreeing on formats is a necessary precondition to enabling multiple interoperable implementations. Internet standards like TLS 1.3 [38] and MLS [8] devote 20% of their text to describing message formats, relying on a custom language called the TLS presentation language. Other protocols rely on a variety of format description languages to encode cryptographic inputs, including XML [1, 2, 15], JSON [23, 24], CBOR [41], Protocol Buffers [40], and ASN.1 [22].

Each format description language aims to make it easier for developers and protocol designers to design new formats and correctly implement serializers and parsers for them. However, the sheer number of these languages should serve as a warning sign of the diversity of formatting requirements and constraints in mainstream protocols. Binary formats like CBOR and Protocol Buffers prioritize conciseness, while text formats like XML and JSON aim for Web-friendly interoperability. The TLS presentation language is specialized for a single family of protocols, while ASN.1 aims to be generic and self-describing. Furthermore, proprietary protocols often use their own custom formats according to their own needs.

Unfortunately, despite the great deal of attention given to describing formats, and although the dangers of format confusions have long been known, cryptographic protocols still get them wrong, resulting in high-profile attacks that continue to be found on a regular basis, both in published standards and in proprietary software.

Format Confusion Attacks. Mavrogiannopoulos et al. [29] describe an attack on TLS 1.2 that relies on a format confusion between the signature inputs used in two kinds of Diffie-Hellman handshakes; the attacker takes a server signature produced in one handshake and uses it to impersonate the server in another handshake. Wallez et al. [47] describe a vulnerability in MLS draft 12, where the inputs to signatures in the TreeSync and TreeDEM components of MLS can be confused for each other.

More recently, Paterson et al. [35] describe a series of attacks on Threema, including a format confusion attack between different encrypted messages in the C2S and E2E sub-protocols. Another attack was recently found on the Matrix protocol, where the inputs to MACs used in two different messages could be confused [6].

These attacks involve different kinds of protocols, and different cryptographic constructions, but in all cases, the problem can be traced to the incorrect or ambiguous use of formats within cryptographic inputs. These flaws are in the protocol itself, and so cannot be fixed by a clever implementation.

Even finding such format confusion attacks in a large standard like TLS 1.3 or MLS can be a real challenge, since the attacks often involve messages in different sub-protocols, sometimes described in different documents altogether. To systematically find and prevent format confusion attacks in real-world protocols, we need a formal framework for specifying and reasoning about secure formats.

Analyzing Crypto Protocols with Precise Formats. Lowe's attack on Needham-Schroeder and the subsequent fix have been very influential, serving as a motivating example for a whole line of protocol verification tools based on symbolic (or Dolev-Yao) analysis [7], including modern tools like ProVerif [14], Tamarin [31], and DY* [10] that have been applied to real-world cryptographic protocols like TLS, Signal, and Noise.

However, the analysis of precise formats remains poorly supported by protocol verification tools and techniques. For example, the default model of symbolic concatenation used in ProVerif, Tamarin, and DY*, fails to find the format confusion attack on the fixed Needham-Schroeder-Lowe protocol even today. While it is possible to extend the algebra of terms to account for some format confusion attacks, one cannot be sure that all the low-level details of the format have been captured.

Pen-and-paper cryptographic proofs (and their mechanized variants in tools like CryptoVerif [13] and EasyCrypt [9]) technically reason about bitstring messages, but in terms of practical format analysis, they do even worse than symbolic tools. Proofs in the computational model of cryptography are much harder than symbolic analyses, so in order to make a security proof feasible, papers routinely disregard any formatting concerns and focus only on the cryptographic steps. As we will see in §4, even comprehensively studied protocols like TLS 1.3 have not been properly analyzed for format confusion attacks.

Our Work and Contributions. As we have seen, there is a large gap between the academic proofs for cryptographic protocols and real-world protocols with complex internal formats. To close this gap, we need a framework for protocol analysis that can specify and prove bit-level precise formats suitable for all the cryptographic inputs used in a protocol. To ensure that we got the formats correctly, we need to be able to test these formats against published test vectors and interoperable implementations. We then need to be able to automatically, or semi-automatically, search for and prove the absence of format confusion attacks across cryptographic inputs in all the message in all related sub-protocols. Finally, we need to be able to embed our formatting proofs within a protocol analysis framework that can verify the security of protocols.

In this paper, we propose a new framework that addresses these requirements. Our framework, called Comparse, uses game-based cryptographic assumptions to establish the set of requirements that formats must obey for their usage to be secure. Because our usage restrictions encompass several classes of attacks, we come up with criteria over formats that rule out not only confusion attacks, but also other well-known attacks. Comparse is implemented and embedded within the F* programming language and verification framework. We demonstrate its expressiveness by using it to specify and analyze all the formats used in large Internet standards like TLS 1.3 and MLS, as well as classic protocols like Needham-Schroeder. We show how our framework allows us to guide the design of new variants of TLS like Compact TLS 1.3 (cTLS). Finally, we show how our framework is embedded within the DY* protocol verification framework and can be used to verify the security of cryptographic protocols while accounts for bit-level precise formats.

Ours is the first formatting framework that is embedded within a protocol security analysis tool. We provide the first formal proof of correctness for the formats of cTLS, an emerging standard for IoT, and we close an important gap in prior analyses of TLS 1.3. Although our framework does produce reference implementations of serializers and parsers for our formats, this is primarily meant for testing our specification, not as production-ready code. Producing efficient, zero-copy parsers for protocol formats is not our goal.

Outline. Section 2 provides a high-level overview of secure formats and establishes the properties they should obey via a new flavor of cryptographic games. Section 3 describes a formalization and implementation of our format analysis framework, Comparse in F*. Section 4 describes TLS 1.3, cTLS, examines the gap between the published security proofs of these protocols when deployed in parallel with each other, and shows how Comparse can address those. Section 5 shows how Comparse is integrated into DY*. Section 6 discusses our results, their impact, and their limitations. Section 7 briefly describes related work, and Section 8 concludes.

2 THE ESSENCE OF SECURE FORMATS

Real-world formats in protocols like TLS are described in a combination of custom language, comments, and English prose. This has made their comprehensive study difficult, and has resulted in several protocol attacks that leverage *design* flaws in these formats.

To address these shortcomings, this section introduces a formal notion of *message formats* and properties over those message formats, which form the foundation of our security analysis. In §3, we shall see how this notion is formalized in a proof assistant.

Throughout this section, we define formats for objects that are represented as bytes. This means that our formats are not just for messages sent over the wire, but also apply to cryptographic inputs (for signatures, MACs, transcript hashes, etc) or any protocol session state or key material that is stored in some binary format.

2.1 Formally Defining Message Formats

We use $\mathbb{B}$ to denote the set of byte sequences (also known as “bytestrings”), and a byte is a value in $[0, 255]$. We write ε for the empty bytestring, and $b_1 + b_2$ for the concatenation of two bytestrings. We write literal bytestrings in hexadecimal notation; for example, `c0ffee` is a bytestring of length 3.

Message Format. A message format for a type M is a relation $\rightleftarrows^M$ between M and $\mathbb{B}$. The index M over $\rightleftarrows$ disambiguates relations when there are several types M, M' involved. There may be several relations for a given M, but we only ever manipulate one such $\rightleftarrows^M$ in any given context.

If we pick $M = \mathbb{B}^2$, and define the message format $(b_1, b_2) \rightleftarrows^M b$ to be e.g. $\exists b_0 \in \mathbb{B} . b_0 + b_1 + b_2 = b$, then the following three properties hold: $(\texttt{c0}, \texttt{ffee}) \rightleftarrows^M \texttt{c0ffee}$, but also $(\texttt{c0ff}, \texttt{ee}) \rightleftarrows^M \texttt{c0ffee}$, and $(\texttt{c0ff}, \texttt{ee}) \rightleftarrows^M \texttt{feedc0ffee}$.

Albeit simplistic, this example warrants two observations. First, we define our notion of format without any reference to a parser or a serializer. In our view, a format is defined as a relation independently of any concrete encodings. Second, this sample format enjoys almost no property of interest: among the many issues with this format, we remark that a given bytestring may correspond to multiple elements in M, and conversely, that an element in M may be represented by several bytestrings.

Serializers and Parsers. Naturally, designers and implementers do not think in terms of logical predicates relating M and $\mathbb{B}$, but rather in terms of concrete formats defined by parsers and serializers. A serializer for a message format $\rightleftarrows^M$ is a function $\text{serialize}_M : M \to \mathbb{B}$ such that (correctness) $\forall m \in M . m \rightleftarrows^M \text{serialize}_M(m)$. A parser for a message format $\rightleftarrows^M$ is a function $\text{parse}_M : \mathbb{B} \to M \cup \{\bot\}$, such that (completeness) $\forall b \in \mathbb{B} . (\exists m \in M . m \rightleftarrows^M b) \implies \text{parse}_M(b) \neq \bot$, and (correctness) $\forall b \in \mathbb{B} . \text{parse}_M \neq \bot \implies \text{parse}_M(b) \rightleftarrows^M b$.

We note that from completeness, it follows that a parser returns $\bot$ (meaning the input bytestring b is malformed) if and only if there no element of M is in relation with b (meaning b cannot be parsed).

Induced Message Format. Given two functions $\text{serialize}_M : M \to \mathbb{B}$ and $\text{parse}_M : \mathbb{B} \to M \cup \{\bot\}$, we define their induced message format on M as: $m \rightleftarrows^M b := m = \text{parse}_M(b) \vee \text{serialize}_M(m) = b$. This is the smallest message format for which parse_M is a parser and serialize_M is a serializer.

The induced message format relates the programmer-centric, concrete view (a format is defined by its parser and serializer) with the security-centric, more abstract view (a format is a relation between messages and bytestrings). Hence, it shows that our abstract formats can still be defined using a concrete parser and serializer.

An advantage of our format-centric view is that it allows us to state properties on a single concept (the format, i.e. a mathematical relation), rather than stating two separate properties (on the parser, and the serializer), and having to account for implementation details (such as the possibility of parsing failure).

Furthermore, as we will see shortly, we show that security properties on an induced message format can be turned into more familiar properties over the underlying parser and serializer, as is done in our implementation of Comparse. There is therefore no lack of expressivity, nor awkwardness, in adopting formats as our central concept for which core notions are defined.

2.2 Properties of Message Formats

Leveraging the definitions above, we start with two *security* properties: non-ambiguity, and representation unicity. Failure to exhibit these properties when needed typically indicates a protocol weakness. We follow with two intermediary properties: non-extensibility, and non-emptiness. These are typically established as lemmas towards a functional correctness result, or a larger security proof.

Non-ambiguity. A message format $\rightleftarrows^M$ is non-ambiguous if it relates at most one high-level message to each bytestring. Formally: $\forall m_1, m_2 \in M, b \in \mathbb{B} . m_1 \rightleftarrows^M b \wedge m_2 \rightleftarrows^M b \implies m_1 = m_2$.

A non-ambiguous format is one for which the parser can only make a single choice for a given bytestring. It is usually easy to find and prevent ambiguity in message formats sent over the wire. When designing the message parser, the implementer will usually notice that multiple choices can be made, or they will typically find the issue during interoperability testing or by fuzzing.

The format-confusion attack on Needham-Schroeder mentioned in §1 relies on passing the contents of the second message as a valid format for the first message. Let us see how this can be seen as a violation of our format security properties. Since all three messages in the protocol use the same cryptographic construction (public-key encryption) with potentially the same keys, the three message payloads should conservatively be treated as three sub-cases of a single message format. However, we would then find that this shared format violates the non-ambiguity property above, since the formats of the first and second message overlap. To restore non-ambiguity for the payload format, we would need to change the protocol, by tagging each message with a distinct label, for example. Such format confusion issues are widespread in real-world protocols, as we shall later see in the context of TLS 1.2 (§2.3).

However, formats used in cryptographic constructions, such as signatures, do not always involve parsing messages, and so non-ambiguity is not naturally enforced or systematically tested. This has resulted in high-profile attacks, as we saw in §1.

As we mentioned earlier, the induced message format allows carrying properties of the *format* onto a parser and serializer.

Lemma. *Given a parser and serializer for M, the induced message format* $\rightleftarrows^M$ *is non-ambiguous if and only if:*
$\forall m \in M.\, \mathrm{parse}_M(\mathrm{serialize}_M(m)) = m$.

Representation unicity. A message format $\rightleftarrows^M$ is said to enjoy representation unicity when at most one bytestring can be in relation with each high-level message. Formally:
$\forall m \in M, b_1, b_2 \in \mathbb{B}\,.m \rightleftarrows^M b_1 \wedge m \rightleftarrows^M b_2 \implies b_1 = b_2$.

Representation unicity has sometimes been referred to as non-malleability [36]. We prefer the term "representation unicity", to avoid confusion with the standard notion of cryptographic malleability [48]. Just like non-ambiguity, representation unicity is mandatory in many security contexts, such as signed content. For instance, transaction malleability is a serious concern in Bitcoin, as described in BIP 62 [49]. Representation unicity rules out such attacks, by imposing a unique bytestring representation for each signed high-level message or transaction. We can also state representation unicity as a property of parsers and serializers.

Lemma. *Given a parser and serializer for M, the induced message format* $\rightleftarrows^M$ *has representation unicity if and only if* $\forall b \in \mathbb{B}\,.\, \mathrm{parse}_M(b) \neq \bot \implies \mathrm{serialize}_M(\mathrm{parse}_M(b)) = b$.

We now look into additional properties that are not directly security-critical, but are oftentimes required in the course of proving functional correctness, or the security of a complete protocol.

Non-extensibility. A message format $\rightleftarrows^M$ is non-extensible when for every bytestring, at most one of its prefixes is in relation with a high-level message. Formally:
$\forall m_1, m_2, b_1, b_2 . m_1 \rightleftarrows^M b_1 \wedge m_2 \rightleftarrows^M (b_1 + b_2)) \implies b_2 = \varepsilon$.

We remark that our format-based approach allows us to separate this property from non-ambiguity, i.e. non-extensibility does *not* imply $m_1 = m_2$. In practice, most of the formats we care about are both non-extensible and non-ambiguous, with the notable exception of the extensible TLS transcript.

The non-extensibility property is not desirable in itself, but constitutes an important intermediary lemma in order to establish the non-ambiguity theorem for the dependent pair combinator (§3.2).

Non-emptiness. A message format on M is non-empty when the empty string is associated with no high-level message. Formally:
$\forall m, b.m \rightleftarrows^M b \implies b \neq \varepsilon$.

Like non-extensibility, non-emptiness does not directly serve any security purpose, but is a crucial property required of a format in order to derive non-ambiguity of its list combinator (§3.4).

Lemma. *Given a parser and serializer for M, the induced message format* $\rightleftarrows^M$ *is non-empty if and only if* $\forall b \in \mathbb{B}\,.\, \mathrm{serialize}_M(b) \neq \varepsilon$

Self-contained and data-dependent message formats. A message format is said to be data-dependent when it is parameterized over a piece of data not contained in the bytestring. A message format that is not data-dependent is said to be self-contained. For instance, TLS12SignatureInput (Figure 3), is data-dependent, over KeyExchangeAlgorithm. A concrete consequence is we cannot make sense of a bytestring for a data-dependent message format, until we know all of the data dependencies. Self-contained message formats are crucial in cryptographic protocol design (§2.3), and protect against protocol-confusion attacks such as [29].

2.3 A Systematic Approach to Format Security

Having defined what we mean by message format (§2.1), and having stated properties of interest for such message formats (§2.2), we now connect cryptographic primitives and secure formats.

First, a remark: almost every cryptographic primitive implicitly relies on a message format for its input. For instance, hashing an object implicitly relies on converting the object to a bytestring. The format must not introduce collisions in the process. Similarly, signatures are implicitly carried around as bytestrings; for functional correctness, the format must allow for a successful verification.

We now set out to review the standard toolkit of cryptographic primitives; we lift each primitive to a *high-level* primitive operating on high-level messages (instead of bytestrings) by relying on a message format. We then proceed by reduction: we state a high-level security assumption (for the high-level primitive operating on messages in M), and determine which properties the format should enjoy in order for this assumption to reduce down to a standard security assumption on the low-level (bytestring-based) primitive. This allows proofs of protocol security to work off of these high-level security assumptions, and abstract message formats away.

In order to be meaningful, the security assumptions we come up with during reduction apply to all usages of a given primitive, across the entire protocol. This means that we can identify design weaknesses, such as lack of disambiguation of signature inputs, because we consider all the signatures in the entire protocol. We explain below with the example of signatures.

Additional primitives can be added to this list, taking care to equip them with suitable restrictions on formats that enforce correct cryptographic usage, following the methodology we describe here.

Our security conditions are somewhat opinionated: they are sound with respect to standard cryptographic assumptions, however there exist protocols that don't satisfy our conditions, yet remain secure; after all, the protocol where two parties can never communicate is always secure, regardless of the cryptographic operations that are performed underneath. But in reality, protocols that violate these assumptions would raise red flags in the cryptographic community, and would most likely be shunned.

Tracking all uses of all primitives across an entire protocol is non-trivial, and difficult to perform by hand; the next section (§3) shows how proof assistants can help scale our comprehensive format security analysis to real-world protocols.

Signature. We begin with signatures, whose security we consider in some detail; other primitives use a similar argument.

Each signature key must be used to sign messages (of high-level type M) with the same self-contained, non-ambiguous, representation-unique message format $\rightleftarrows^M$. If a signature key is used with two different message formats, or the format is ambiguous or data-dependent, this could lead to a signature confusion attack, such as the one exploited in TLS 1.2 [29] (as explained in §4). This condition therefore ensures the signed bytestrings correspond to the same unique message, and thus rule out signature confusion attacks.

This invariant on the whole protocol can then be exploited in security proofs. For example, we can lift the standard existential unforgeability under chosen message attack (EUF-CMA) assumption and specialize it with the message format: the challenger generates a pair of keys and give the public key to the attacker, then the attacker can ask the challenger for signatures of messages M, and succeeds if it manages to output $x \in M$ not queried to the challenger, along with a valid signature for it. This lifted EUF-CMA game security can then be reduced to the standard EUF-CMA security assumption, using the non-ambiguity and self-contained properties of $\rightleftarrows^M$.

We observe that this lifted EUF-CMA game doesn't say anything about a signature key used to sign two different message formats; this means that in order for the game to apply, we must have a way to ensure we only operate over signatures of messages in M. In other words, we enforce the absence of format confusion across different uses of signature within the protocol.

The input format for signatures must also enjoy representation unicity for functional correctness, so as to rule out the scenario where a message corresponds to two possible bytestrings, one used by the signer and the other by the verifier, which would lead to a verification failure for valid signatures.

Enforcing these requirements means that one must track every usage of a signature key; notably, in the case of TLS, this requires not only tracking signatures across TLS 1.2 and TLS 1.3, but also anticipating future extensions or versions of the protocol.

Symmetric MACs. The same precautions as for signatures must be used, for the exact same reasons. However, in practice, MAC keys are short-lived and used only a few times, which makes tracking all their uses easier. We recommend that all messages that may be MACed with the same key must conform to the same non-ambiguous, self-contained message format.

Authenticated Encryption with Associated Data (AEAD). As with MACs, we recommend that all encryption inputs that might use the same key must use the same non-ambiguous, self-contained message format for encrypted data and additional data. In some cases, this is stricter than necessary, and we can allow the format for encrypted data to be data-dependent on the associated data. It allows additional data to fulfill its duty: providing context in which the encryption was performed, and disambiguating identical inputs stemming from two different context. For functional correctness, the format of additional data must enjoy representation unicity (to prevent two parties from disagreeing on the serialization).

We note that the Threema messaging protocol fails these conditions, and uses ambiguous inputs to authenticated encryption, resulting in the attack [35] described earlier (§1).

Key Derivation Functions (KDFs). Given a secret key, a salt, and an info field, a KDF like HKDF generates a pseudo-random output of any desired length. In the cryptographic literature, a KDF is typically modeled as a pseudo-random function (PRF), where we assume that the attacker cannot distinguish the output of the KDF from a fresh random value. Protocols like TLS typically use a KDF to generate multiple keys for different purposes. To guarantee key independence for these keys, which is often an important precondition in security proofs, it is necessary to ensure that all the info fields used by a KDF with the same key and salt use the same non-ambiguous and self-contained message format. Furthermore, to preserve functional correctness, the format of these KDF inputs must enjoy representation unicity.

Hashing Messages and Transcripts. Many protocols use hash functions to compute digests of high-level data, including messages and protocol transcripts. A common requirement for this usage is collision resistance – two different inputs should yield two different hashes, except with negligible probability. However, even when using a collision-resistant hash function, this property may not hold for two high-level messages if they serialize to the same bitstring. Hence, we recommend that all inputs to hash functions must use non-ambiguous message formats. Furthermore, if possible, we advise that protocol authors use a single, self-contained message format for hash functions when two hashes must be collision-resistant in security proofs. Protocols might fail to obey this restriction, but might still be secure, for instance if the data dependency is authenticated elsewhere; such a situation will call for more sophisticated proofs. For functional correctness, the hash input format must often also satisfy representation unicity.

Summary. All of the cryptographic operations considered above require non-ambiguity and representation unicity. The former rules out confusion; the latter is required for functional correctness. As we will see shortly, our format framework imposes, by construction, that every format must satisfy these two properties.

3 VERIFIED FORMATS IN F*

We now put our ideas in practice, and formalize secure formats within the F* proof assistant. F* is a dependently-typed programming language that supports program proof using a mixture of automatic (SMT-based) and manual (tactic-based) proofs. F* supports a wide array of programming patterns, including compile-time term synthesis, which we leverage in this section. Throughout this paper, we only ever use the pure fragment of F* and need not rely on its effect system.

Studying formats as complex as those of TLS in a monolithic fashion is unrealistic; any reasonable programmer will decompose formats into basic blocks that can each be studied in isolation, then composed together to form larger formats. To support this modular approach, this section introduces a set of format combinators that allow assembling complex message formats from simpler ones. We show how security properties of complex formats (the application of a combinator) can be deduced from the security properties of the simpler formats they are built upon (the arguments to the combinator). Authoring these formats by hand is tedious; we show how to automate the process using Meta-F*.

Although minimalistic (we only use, and describe, a mere 4 combinators), our approach is expressive enough to describe all message formats in TLS, cTLS and MLS. Our combinators guarantee that the formats they produce are secure. This proof is done once and for all, which relieves the programmer of the bulk of the proof effort. Users may also opt out of combinators and write message formats directly, but then are required to prove their correctness by hand, which is more onerous.

```
struct {
    ProtocolVersion legacy_version;
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16−2>;

    opaque legacy_compression_methods<1..2^8−1>;

    Extension extensions<8..2^16−1>;

} ClientHello;
```

(a) ClientHello as defined in the TLS 1.3 RFC [38]

```
type client_hello = {
    legacy_version: protocol_version;
    random: random;
    legacy_session_id: tls_bytes {min=0; max=32};
    cipher_suites:
        tls_list cipher_suite {min=2; max=(pow2 16)−2};
    legacy_compression_methods:
        tls_bytes {min=1; max=(pow2 8)−1};
    extensions:
        tls_list extension {min=8; max=(pow2 16)−1};
}
```

(b) Equivalent client_hello type in F*

Figure 1: Translation of TLS 1.3 ClientHello in F*. Note how the F* type is precise: for example, the tls_bytes type represents bytes of bounded length, precisely corresponding to the opaque x<n..m> notation used in the TLS 1.3 RFC.

3.1 Defining Secure Message Formats in F*

Definitions, lemmas, proofs of reductions. We follow §2.1 and define formats as logical predicates in F*. We transcribe definitions from §2.2 and prove all of the corresponding lemmas.

Connection with parsers & serializers. Presenting formats as relations allowed us to capture the essence of formats, along with their security properties, free of the implementation-centric notions of parsers and serializers.

However, in our F* library, we define our formats using parsers and serializers. This design decision is a pragmatic one. First, this makes life easier for the programmer and doesn't require them to write a logical predicate $\rightleftarrows^{M}$ by hand. Second, this makes our formats *executable* which is crucial for authoring reference implementations that can serve for interoperability testing, but also for building further security proofs (§5). Third, we write and prove (in F*) the connection between induced formats and parsers and serializers (§2.1). Because this connection is verified, we not only do not lose any expressive power, but also provably know that our security properties over parsers and serializers are equivalent to the original security properties over the induced format.

A type definition for secure formats. Next, we devise a "user interface" for Comparse, that is, we write a concrete type definition in F* for what we mean by *format*.

As we saw earlier (§2.3), regardless of the context in which the formats appear, we always want them to enjoy non-ambiguity and have a unique representation. Therefore, our type of formats, below, takes not only a parser and a serializer, but also *proofs* that those two crucial properties always hold. Per the lemmas from §2.1, those two properties boil down to stating that the parser and serializer are inverses of each other. Because of this design choice, our library will refuse to handle ambiguous or non-unique formats: it is our position that formats that fail to exhibit these properties indicate a design weakness in the protocol.

```
type message_format_for (a:Type) = {
  parse: bytes ⟶ option a;
  serialize: a ⟶ bytes;
  // Non-ambiguity
  parse_serialize_inv: x:a ⟶ Lemma ( parse (serialize x) == Some x );
  // Representation unicity
  serialize_parse_inv: buf:bytes ⟶ Lemma (
    match parse buf with
    | Some x ⟶ serialize x == buf
    | None ⟶ ⊤); }
```

Non-extensible message formats. We define a separate type, called prefix_message_format_for, to represent non-extensible secure message formats, where the parser only consumes a prefix of the input bytestring, and returns the parsed element and the remaining suffix. In this type, the parse_serialize_inv property is adapted to allow for the suffix, as follows:

```
parse ((serialize x) ++ suffix) == Some (x, suffix)
```

Non-empty message formats. We say a message format is non-empty when all its serializations have non-zero lengths (Lemma 2.2). In F*, we offer this as a refinement over the earlier types.

3.2 The dependent pair combinator

We begin with our first combinator for pairs. Repeated applications of this combinator allow encoding pairs of several elements (known as tuples): we write $A \times B \times \ldots \times D$ for $A \times (B \times (\ldots \times D))$.

Tuples naturally occur in the wild, such in the ClientHello message of TLS (Figure 1a), which is simply the combination of all of its subfields. Because our combinators are generic, they cannot produce a specific user-defined type such as ClientHello; rather, they produce a tuple of ProtocolVersion × Random × . . .. We show in §3.5 how to convert a structural type (the tuple) into a nominal type suitable for the rest of the protocol definition (the user-provided client_hello, Figure 1b).

Sometimes, the message format for the second element of a pair depends on the contents of the first one. This is a *dependent* pair, which generalizes to dependent tuples. A dependent tuple occurs in the Handshake message of TLS (Figure 2), where the format of the third field depends on the value of the first one (via select), as well as the value of the second one (via the comment referring to length). Our combinator supports general dependent pairs, of which non-dependent pairs are a special case; this allows to encode, notably, the tagged union pattern of messages such as Handshake.

Message format combinator. The message format for the dependent pair $X \times Y$ is defined as a simple concatenation. Formally: $(x, y) \rightleftarrows^{X \times Y} b := \exists b_1, b_2 . b = b_1 + b_2 \wedge x \rightleftarrows^X b_1 \wedge y \rightleftarrows^{Y(x)} b_2$.

We write the dependency explicitly ($\rightleftarrows^{Y(x)}$), which captures the fact that the format of the second element depends on the first. We use a lightweight notation $X \times Y$ for dependent pairs to avoid cluttering the formulas, as opposed to the traditional $\sum_{x:X} Y(x)$. In practice, the dependent pair combinator allows turning a data-dependent format (the second element of the pair, §2.2) into a self-contained format (if the dependency is only over the first element). A common instance of this pattern is for tagged unions.

Formally proven security properties.

- $\rightleftarrows^{X \times Y}$ is non-ambiguous if $\rightleftarrows^X$ is non-extensible and non-ambiguous, and $\rightleftarrows^{Y(x)}$ is non-ambiguous for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ has representation unicity if $\rightleftarrows^X$ and $\rightleftarrows^{Y(x)}$ have representation unicity for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ is non-extensible if $\rightleftarrows^X$ is non-extensible and non-ambiguous, and $\rightleftarrows^{Y(x)}$ is non-extensible for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ is non-empty if $\rightleftarrows^X$ is non-empty or $\rightleftarrows^{Y(x)}$ are non-empty for every $x \in X$.

Role of non-extensibility. In the non-ambiguity theorem, non-extensibility of the first element of the pair is crucial: for example, consider the trivial message format on $\mathbb{B}$, defined as $b_1 \rightleftarrows^{\mathbb{B}} b_2 := b_1 = b_2$. This message format is non-ambiguous, but a non-dependent pair of two such formats is, for the same reason as the message format on $\mathbb{B}^2$ studied in §2.1.

Formalization in F*. Two flavors exist for the dependent pair combinator: although the message format for the first element of the pair must always be non-extensible, there is no such restriction on the second element of the pair. Furthermore, the result is non-extensible if and only if the second element of the pair is non-extensible. We reflect this with two separate F* functions.

```
// When both mf_a and mf_b have the non-extensibility property
val prefix_message_format_for_dep_pair:
  #a:Type → #b:(a → Type) →
  mf_a:prefix_message_format_for a →
  mf_b:(x:a → prefix_message_format_for (b x)) →
  prefix_message_format_for (x:a & b x)

// When only mf_a has the non-extensibility property
val message_format_for_dep_pair:
  #a:Type → #b:(a → Type) →
  mf_a:prefix_message_format_for bytes a →
  mf_b:(x:a → message_format_for bytes (b x)) →
  message_format_for bytes (x:a & b x)
```

Encoding Handshake with dependent pairs. We illustrate the usage of the dependent pair combinator on the type of handshake (Figure 2). Assuming we have message formats for T (HandshakeType), U_{24} (uint24, unsigned 24-bit integers) and $M(t, l)$ (data-dependent handshake content of type t and serialized length l), Handshake can be encoded as $T \times (U_{24} \times M)$ (or more precisely: $\sum_{t:T} \sum_{l:U_{24}} M(t, l)$). We note that the resulting dependent triple is no longer data-dependent.

We cannot yet show the definition of M; is it a dependent type, along with an added restriction over its length. To express the latter, we need a new format: the refinement combinator.

```
struct {
    HandshakeType msg_type; /* handshake type */
    uint24 length; /* remaining bytes in message */
    select (Handshake.msg_type) { /* handshake content */
        case client_hello: ClientHello;
        case server_hello: ServerHello;
        /* ... */
    };
} Handshake;
```

Figure 2: The Handshake message format, as defined in TLS 1.3 [38]. The msg_type determines the format to use for the handshake content (via select). Furthermore, the comment for field length encodes a semantic restriction: the total length (in bytes) of the select ... field is equal to length.

3.3 The refinement combinator

Message formats are sometimes described as subsets of other message formats. For example, we can define a boolean as a byte restricted to the value 0 or 1.

Message format combinator. If $Y \subset X$, and we have a message format $\rightleftarrows^X$, then we can define $m \rightleftarrows^Y b := m \rightleftarrows^X b$.

Length restriction. A particularly useful usage of the refinement combinator is to enforce exact length restrictions on high-level messages. Given a set of messages M, we define its subset $\text{RestrictLen}(M, l) = \{m \in M \mid \forall b . m \rightleftarrows^M b \implies \text{length}(b) = l\}$. This refinement, when used in conjunction with a dependent pair, allows encoding length-prefixed messages, wherein the first element of the pair is a (bounded) unsigned integer that stands for the length of the second element.

Formally proven security properties. The refinement combinator preserves non-ambiguity, non-extensibility, non-emptiness and representation unicity. When used with RestrictLen, it is unconditionally non-extensible.

Formalization in F*. The refinement combinator also comes in two flavors, depending on whether the input format is extensible or not. We show the extensible version here:

```
val refine:
  #a:Type → message_format_for a → pred:(a → bool) →
  message_format_for (x:a{pred x})
```

We provide a dedicated combinator that captures the fact that an extensible format can be turned into a non-extensible one, via a length restriction.

```
val fixed_length_format_to_non_extensible:
  #a:Type → len:nat →
  mf_a:message_format_for a{∀ x. length (mf_a.serialize x) == len} →
  prefix_message_format_for a
```

Encoding Handshake content with refinement. Now that we have refinements, we can revisit our earlier Handshake example (§3.2) and use RestrictLen to encode the constraint on the length of the third field. As mentioned above, this means the third field is unconditionally non-extensible, which in turn makes the whole Handshake message non-extensible. Our format for handshake is now of the form $\sum_{t:T} \sum_{l:U_{24}} \text{RestrictLen}(M'(t), l)$.

3.4 The list combinator

With the dependent pair combinator (§3.2), we can encode fixed-sized lists as n-tuples, but cannot represent lists whose length is not known at compile-time. For this, we need a new list combinator.

Message format combinator. Given a message format on M, we define a message format on M^*, the type of lists of Ms (with any number of elements), as: $[m_1, \ldots, m_n] \rightleftarrows^{M^*} b := \exists b_1, \ldots, b_n . b = b_1 + \cdots + b_n \wedge \forall i. m_i \rightleftarrows^{M} b_i$. Our format does not require that the list be prefixed by its length, although if the protocol mandates it, we can always encode it using a combination of refinement, dependent pair, and list combinator.

Formally proven security properties.

- $\rightleftarrows^{M^*}$ is non-ambiguous if $\rightleftarrows^{M}$ is non-ambiguous, non-extensible, non-empty.
- $\rightleftarrows^{M^*}$ has representation unicity if $\rightleftarrows^{M}$ has representation unicity.

Role of non-emptiness. Requiring non-emptiness rules out degenerate cases, such as the unit format $() \rightleftarrows^{unit} \varepsilon$. Lists of units all serialize to a single empty bytestring, meaning lists of unit are ambiguous.

Formalization in F*. To be secure, the list combinator takes as input a non-extensible, non-empty secure message format. It returns an extensible message format. Non-ambiguity and representation unicity are carried over automatically, since they are bolted into the two message format types.

```
val message_format_for_list:
  #a:Type → mf_a:prefix_message_format_for a{is_non_empty mf_a} →
  message_format_for (list a)
```

Encoding the TLS 1.3 transcript. The TLS transcript is a list of Handshake messages. Because Handshakes are non-ambiguous, non-extensible, non-empty, and have representation unicity, the TLS 1.3 transcript is non-ambiguous and has representation unicity, which are crucial properties that guarantee the correct behavior of transcript hashes in the security proof of TLS.

3.5 The isomorphism combinator

Given a message format defined in a document, such as in the TLS 1.3 RFC [38], we write a type in F* precisely capturing the expressivity of the message format. This corresponds to the type M of messages we saw earlier. We give an example for the TLS 1.3 ClientHello message in Figure 1.

The three combinators we have seen so far can parse ClientHello and Handshake, but return tuples that are isomorphic to, but not equal, to the type that the user would write in F* for Handshake (Figure 1).

We thus need one final combinator that goes from a *generic* representations (base types, lists, and dependent pairs) into the "original", user-defined message type. This is the isomorphism combinator. The isomorphism combinator is typically the final building block used to create a message format.

Message format combinator. Given a bijective function $f : T \rightarrow E$, which maps a high-level type T to an encoding E, and a message format for E, we define a message for T as $m \rightleftarrows^{T} b := f(m) \rightleftarrows^{E} b$.

Formally proven security properties. Because f is a bijection, the isomorphism combinator preserves non-ambiguity, representation unicity, non-extensibility, non-emptiness.

Formalization in F*. We rely on two functions for the bijection, which we require to be inverse of each other. We do this by adding a precondition to the isomorphism combinator: we can use it only if we prove that the two bijection functions are inverse of each other. This precondition is crucial to prove that the resulting message format is secure. The isomorphism combinator comes in two flavors, depending on whether the message format is non-extensible or not. Because they are so similar we only show the signature of the non-extensible version.

```
val prefix_message_format_for_isomorphism:
  #a:Type → #b:Type → mf_a:prefix_message_format_for a →
  a_to_b:(a → b) → b_to_a:(b → a) →
  Pure (prefix_message_format_for b)
  (requires (∀ x. a_to_b (b_to_a x) == x) ∧ (∀ x. b_to_a (a_to_b x) == x))
```

Finalizing our Handshake format. In §3.2, we obtained an encoding of Handshake as a dependent tuple of 3 elements. However, Handshake is not a dependent tuple, it is a nominal type in F*:

```
type handshake = {
  msg_type: handshake_type;
  length: uint24;
  msg: fixed_length_handshake_content msg_type length; }
```

We use the isomorphism combinator to link the nominal type (handshake) and its encoding (the dependent tuple of 3 elements).

3.6 Automating Combinator Synthesis

Writing combinator applications by hand quickly becomes repetitive. We now present a facility that allows the user to write only their top-level type, such as ClientHello (Figure 1). With a few strategically placed annotations, our facility inspects the type definition and automatically generates the combinators that will parse and serialize elements of that type.

Our facility relies on Meta-F* [28], a general-purpose compile-time metaprogramming framework. Using a technique known as elaborator reflection, Meta-F* essentially allows the programmer to "script" the compiler, to resolve proof obligations, or in our case, inspect terms and generate fresh definitions.

We authored a meta-program that takes an annotated type definition and produces a corresponding format, complete with proofs of non-ambiguity and representation unicity.

Inner workings. When processing a type such as client_hello (Figure 1b), the meta-program proceeds in two steps: first, it derives a message format using anonymous types (dependent tuples), then it uses the isomorphism combinator (§3.5) to produce a message format for the user-defined type (client_hello, a record with user-provided field names).

For each field, a corresponding format is looked up in the environment. If this corresponds to a user-defined type for which a format was previously generated, or to a base type for which we provide a hand-written format (such as uint24), all is well. Otherwise, the meta-program fails and the user must annotate the type by hand to indicate which format ought to be used for the given field.

Once again, this could be done entirely by hand: our automation relieves the user of a repetitive task. Importantly, it also makes the program easier to maintain: if an internet draft is updated to a new revision, the programmer just needs to change the type definition, and the formats automatically follow.

Handshake example. In the case of the handshake, the dependency of the msg field over msg_type and length is handled naturally: the format found in the environment for fixed_length_handshake_content takes two parameters, so the tactic instantiates that format with its two arguments brought in scope by the dependent tuple. Thanks to a judicious choice for our default formats, handshake serializes exactly per the TLS 1.3 RFC [38].

Other supported types. F* sum types are also handled by our tactic, which picks a tagged union scheme. By default, the tag occupies the minimum number of bytes required to encode all cases; the user can override that choice, and specify explicitly which type should hold the tag (including its size). This allows the user to obey a precise format specified e.g. in an RFC.

3.7 Implementation

The implementation described in this section occupies a total of about 2,500 lines of code. This includes the combinators and base types (942 lines), the derived types (776 lines) and the tactic (742 lines). The tactic is among the three largest Meta-F* programs written to date. We estimate that the total effort for formalizing Comparse in F* took a few person-months.

The overall conciseness of our development is explained by the judicious choice of a notion of *format* as our base abstraction, rather than parsers and serializers. Furthermore, the judicious choice of combinators limits the amount of work we need to perform. Finally, the fact that we do not need to rely on effects, along with careful crafting of definitions, allow us to maximize the amount of proofs performed automatically by SMT.

4 VERIFIED FORMATS FOR TLS AND CTLS

The Transport Layer Security (TLS) protocol, standardized by the IETF, is used to secure the vast majority of Web traffic. The most recent version, TLS 1.3, was standardized in 2018 [38] and is the the most commonly used version of TLS, followed by TLS 1.2 [39].

```
struct {
    select (KeyExchangeAlgorithm) {
        case dhe_rsa, dhe_dss, // From TLS 1.2
             dhe_rsa_export, dhe_dss_export: // From TLS 1.0
            opaque client_random[32];
            opaque server_random[32];
            ServerDHParams params;
        case ec_diffie_hellman: // From TLS 1.2 ECC
            opaque client_random[32];
            opaque server_random[32];
            ServerECDHParams params;
    };
} TLS12SignatureInput;
```

Figure 3: A common format for TLS 1.0-1.2 signature inputs [34, 39].

More recently, the IETF TLS working group has been working on standardizing Compact TLS 1.3 (cTLS) [37].

In this section, we will discuss how we implemented all the formats used in TLS and cTLS using Comparse, and show how our work provides crucial missing properties needed for the security analyses of these protocols, both in isolation and when they are deployed in parallel with each other.

4.1 Format Confusion Attacks in TLS 1.0-1.2

The TLS protocol establishes a secure channel between a client and a server. It works in two phases: first, a *handshake* phase performs an authenticated key exchange to establish shared keys between the client and server, then a *transport* phase allows them to exchange application data protected with these shared keys. Typical TLS implementations support multiple versions and ciphersuites for backwards compatibility and to support maximum interoperability.

In TLS versions up to TLS 1.2, each handshake may use one of multiple key exchange modes. Depending on the mode, each TLS handshake consists of between 6 and 13 messages, which include various cryptographic constructions: 2 MACs, 3 key derivations, 2 encryptions, and up to 2 signatures. The formats for all these messages and cryptographic inputs are described in the custom TLS presentation language, which looks like C structs.

For example, the input formats for server signatures used in implementations of TLS 1.0-1.2 is depicted in Figure 3. It includes the specification of Ephemeral Diffie-Hellman signatures (DHE) from TLS 1.2, the signature inputs for export ciphersuites in TLS 1.0, and the format for Elliptic Curve Diffie Hellman (ECDHE) in TLS 1.2. These formats are actually defined in three different documents, and we have brought them together for illustration.

We note that the format depends on a value (the key exchange algorithm) that is external to it, which is not authenticated by the signature itself. Indeed, in the absence of this external input, the signatures used in DHE key exchanges can be confused for those in ECDHE key exchanges, which leads to a concrete cross-protocol attack [29] on TLS 1.2. Note also that the format used in DHE is the same as the one used in Export DHE, which is one of the factors exploited by the Logjam attack [5].

With our methodology (§2.3), we impose that every signature key be associated to a single, self-contained, non-ambiguous message format. These properties are violated here, because the format is not self-contained, owing to the key exchange algorithm which is an external input.

We formalized the signature input format for TLS 1.2 (DHE, ECDHE) in Comparse, and proved that given the data-dependency (KeyExchangeAlgorithm), the format has non-ambiguity and representation unicity. We were not able to do the same without the data-dependency, hinting at the cross-protocol attack. We once again re-emphasize that we see *all* the inputs to a *single* primitive as a *single* format, meaning that both TLS 1.2 and TLS 1.3 signature formats are seen as sub-cases of the general "signature input format". This systematic approach forces us to reason globally about *all* the signatures in the protocols.

4.2 Verified Formats for TLS 1.3

TLS 1.3 fixes many of the attacks in TLS 1.2, including the format confusion attacks described above. In particular, it defines a uniform format for all signature inputs, MAC inputs, and key derivations, by using the handshake transcript in all three cases. TLS 1.3 also encrypts handshake messages for privacy, hence the format of encryption inputs now includes 8 handshake messages.

The security of TLS 1.3 relies crucially on the non-ambiguity of the format of the handshake transcript at each stage of the protocol. It also relies on the non-ambiguity of each handshake message format. Furthermore, if TLS 1.3 is to be safely deployed alongside TLS 1.2, we need to prove non-ambiguity across both protocols.

Despite its importance, this property is not accounted for in many published proofs of TLS 1.3. The mechanized proofs of TLS 1.3 in ProVerif [12], CryptoVerif [12], and Tamarin [16] all assume that the message formats are injective and disjoint, and that the transcript can be treated unambiguously as a tuple of handshake messages. The pen-and-paper security proofs of TLS 1.3 (e.g. [19, 25, 26]) abstract away all formatting details; they typically assume distinct labels for the different cryptographic inputs in the protocol to simplify their analysis. Consequently, none of the published proofs of the TLS 1.3 handshake actually apply to the bit-level formats used in the protocol.

To close this gap in the literature, we formalized all the handshake messages of TLS 1.3, the handshake transcript, and all inputs to signatures, MACs, encryption, and key derivation, in Comparse, and proved that these formats were non-ambiguous. This means that any proof of TLS 1.3 that only considers a high-level abstraction of each message still applies to the concrete protocol that uses low-level bitstrings within the cryptographic inputs. Furthermore, we combined the TLS 1.2 and TLS 1.3 signature inputs to prove that the combined format is non-ambiguous. This result shows that it is safe to deploy TLS 1.3 and TLS 1.2 in parallel.

Some prior works do prove injectivity for TLS 1.2 transcripts [42] but they do not consider TLS 1.3, or both in parallel. There is also prior work on specifications and efficient implementations of TLS 1.3 and TLS 1.2 message formats [36], but they do not model TLS and do not prove non-ambiguity across TLS 1.3 and TLS 1.2 signatures.

```
enum {
  profile(0),
  version(1),
  cipher_suite(2),
  dh_group(3),
  signature_algorithm(4),
  random(5),
  mutual_auth(6),
  handshake_framing(7),
  client_hello_extensions(8),
  server_hello_extensions(9),
  encrypted_extensions(10),
  certificate_request_extensions(11),
  known_certificates(12),
  finished_size(13),
  optional(65535)
} CTLSTemplateElementType;

struct {
  CTLSTemplateElementType type;
  opaque data<0..2^32−1>;
} CTLSTemplateElement;

struct {
  uint16 ctls_version = 0;
  CTLSTemplateElement elements<0..2^32−1>
} CTLSTemplate;
```

Figure 4: Compression templates for Compact TLS 1.3

4.3 Verified Formats for cTLS

The handshake messages exchanged in a TLS handshake can get very large, primarily because TLS is designed to be interoperable across a large range of devices and so the messages contain information that may be needed in different scenarios.

Compact TLS 1.3 is a new proposal that aims to reduce the message size by agreeing out-of-band on a *compression template*. For example, if the client and server can agree beforehand on the ciphersuite and server certificate, several elements in the handshake can be eliminated, and others can be treated as fixed-length values (without length prefixes).

Other than the message formats being compressed, the cryptographic steps in cTLS are identical to TLS 1.3. Consequently, one might hope that all the TLS 1.3 security proofs will apply to cTLS. However, this only holds if we can prove that the cTLS messages and transcripts are unambiguous, and if we can show that the cTLS transcript is equivalent (for each template) to the TLS 1.3 transcript. Finally, since cTLS is likely to be deployed in parallel with TLS 1.3, we need to prove that the joint formats between the two protocols are unambiguous.

Compressing TLS 1.3 using Templates. The cTLS protocol defines a compression template that the client and server must agree to in advance. CTLSTemplate (Figure 4) depicts all the elements that a certain template can fix. Given such a template, the protocol describes how each message in TLS 1.3 can be compressed at a fine-grained level. For example, in the keyshare extension of the client hello, a length field can be omitted if the template specifies a single Diffie-Hellman group. Consequently, each message format in cTLS depends on the compression template.

We follow the methodology described in §3.1 by writing types that precisely capture what is representable by the message format, and depend on the compression template. On the example of the cipher suite compression in ClientHello, we define a new type for the cipher_suites field of client_hello. Although these new types are more complex than the ones used in TLS 1.3, they are fully handled by the meta-program.

```
type ctls_cipher_suites (t:ctls_template) =
  match get_template_element cipher_suite t with
  | Some _→ unit
  | None → tls_list cipher_suite ({min=2; max=(pow2 16)−2})

type ctls_client_hello (t:ctls_template) = {
  (* ... *)
  cipher_suites: ctls_cipher_suites t;
  (* ... *)
}
```

Proofs for cTLS message formats and transcripts. We prove that given a template, each cTLS message has non-ambiguity and representation unicity. We do this by representing each cTLS message with a dependent type (on the template), and use Comparse to derive and prove a format on each cTLS message.

The cTLS transcript includes the compression template as a first (dummy) message and then continues with the list of cTLS handshake messages. We show that the first handshake message of the transcript carries enough information to deduce what modification was applied on the transcript, whether it is hashing the first ClientHello, or it is doing a cTLS compression with a template.

We prove that all the cTLS messages are non-ambiguous with unique representations, and that cTLS transcripts and TLS 1.3 transcripts are non-ambiguous with respect to each other. This means that despite all the format changes and optimizations, cTLS is free from format confusion attacks, and that it is safe to deploy cTLS in parallel with TLS 1.3 (using the same server certificates.)

Equivalence between TLS 1.3 and cTLS transcripts. We also prove that each cTLS compressed transcript correspond to a unique TLS 1.3 transcript, up to extensions re-ordering. Each TLS 1.3 type that is modified by cTLS is associated with a cTLS compressed type, that depends on the compression template. We then write two compression and decompression functions, that convert between the TLS 1.3 type and the cTLS dependent type.

The compression function can fail, for instance if the compression template enforces the some ciphersuite, but the TLS 1.3 message tries to negociate the wrong ciphersuite. The decompression function can also fail, because compression removes some length tags, meaning that there exist valid compressed transcripts that contain elements whose length exceeds what is admissible by TLS 1.3. This is a novel insight that seems to not have been noticed by the cTLS designers before, and might be addressed in future drafts.

We prove round-trip properties on compression and decompression: if compression succeeds, then decompression on its result succeeds and returns something equal to the compression input (up to extension re-ordering); also, if decompression succeeds, then we can compress back its result, it will succeed and return something equal to the decompression input.

By proving these compression/decompression guarantees between TLS 1.3 and cTLS, we show that the messages and transcripts in the two protocols are interchangeable. Consequently, any proof of security for TLS 1.3 that relies on high-level message formats still holds when the messages are formatted (i.e. compressed) according to cTLS. This provides a foundation upon which future proofs of cTLS can build, since they now know that all of the formats are safe, and are in correspondence with those of TLS 1.3.

5 EMBEDDING COMPARSE IN DY*

Existing protocol verification tools often miss format confusion attacks since they do not account for bit-level precise formats. We now show how to close this gap, by integrating Comparse into the DY* symbolic protocol analysis framework. In so doing, we make it possible to apply symbolic protocol analysis to low-level message formats. Despite this additional precision, rather than incurring an additional proof burden, Comparse actually significantly reduces the proof effort for DY* proofs because of its support for automation.

5.1 Background: Symbolic verification with DY*

DY* [10] extends the F* [42] proof system with a symbolic verification framework for cryptographic protocols. In comparison to symbolic provers like Tamarin and ProVerif, DY* proofs require more manual annotations and are less automated. Conversely, DY* uses a more scalable proof technique, based on typechecking, and can exploit the full expressiveness of the F* proof assistant. Consequently, DY* is particularly well-suited to the formal analysis of large, complex cryptographic protocols with recursive protocol flows and inductive data structures. Indeed, DY* has been used to obtain state-of-the-art verification results for advanced protocols like Signal [10], ACME [11], Noise [21] and MLS [47].

Symbolic Message Formats. Like other symbolic provers, DY* relies on the Dolev-Yao model [18], where messages are treated as algebraic terms that can be constructed and destructed using abstract functions that obey simple symbolic equations. These constructors and destructors are used to model cryptographic primitives like encryption/decryption and signature/verification.

Most symbolic protocol analyses in tools like ProVerif and Tamarin ignore message formatting, and simply use tuples to represent message contents. DY* does a little better, by defining an abstract type and interface for bytestrings that include ASCII strings, freshly generated random values, cryptographic elements, and can furthermore be concatenated and split to implement specific message formats.

However, the underlying model is still symbolic, and hence does not precisely model concrete bytestrings or their lengths. For example, in this model, concat(a,concat(b,c)) is different from concat(concat(a,b),c). Consequently, it is possible that verified DY* code written against the *symbolic bytes* API may potentially still be vulnerable to format confusion attacks that exploit such inconsistencies. To counter this, we need to prove that the the bytestring API is also sound with respect to a concrete model of bytestrings.

Verifying Message Formats. Since DY* lacks a framework for automatic format analysis, DY* programmers are expected to write, by hand, serialization and parsing functions for all the message formats, cryptographic inputs, and session states used in the protocol and then prove non-ambiguity for all these formats, to use as lemmas within the security proof.

DY* tracks the secrecy and authenticity of each bytestring using secrecy labels and logical refinements, allowing users to reason about the security guarantees of their protocol code. Hence, the programmer also needs to prove that secrecy labels are preserved by formatting. For example, before sending a message on the network, we need to prove that the serialization of this message is "publishable", so that revealing it to the attacker will not leak secret values. To prove this, the programmer needs to reason about the format, and show that a serialized message is publishable if and only if every field of the high-level message is publishable.

These formatting proofs are currently done by hand in DY*, inducing a significant proof burden even for simple message formats. While this is feasible for small protocols, it quickly becomes tiresome or even impossible for real-world protocols like TLS. In the rest of this section, we show how Comparse can help alleviate this proof burden and close a gap in DY* by providing a concrete model for bytestrings. Furthermore, by combining DY* and Comparse, we are able, for the first time, to execute DY* applications *concretely* to create and process wire-compatible message bytestrings.

5.2 Plugging DY* and Comparse together

Handling multiple bytes types. Until now, we assumed that Comparse worked on one defined type for byte sequences. However, DY* defines its own type to do symbolic protocol verification, the aforementioned *symbolic bytes*. To cover all the bytes types one might want to use, we now parameterize Comparse over a typeclass for bytes, which contains the minimal set of properties and lemmas that work for *any* instantiation of the type class (i.e. for both symbolic and concrete bytes). This means that the entire Comparse development is carefully crafted to never rely on any *extensionality* hypothesis, namely that two bytestrings are equal if they have the same length and coincide on every index. Comparse does not rely on concatenation associativity either; none of these properties hold for symbolic bytes.

To the best of our knowledge, no realistic parser framework was ever devised before to work without the use of extensionality or associativity. This is a key contribution of our work, and a core difference compared to other frameworks such as EverParse [36].

We can therefore use Comparse both for *concrete* bytes, to execute cryptographic protocols, and *symbolic* bytes, to prove security of cryptographic protocols. We now give more details.

The typeclass. To simplify the instantiation of the typeclass with various bytes types, we make it as minimal as we can. In the typeclass, we require bytes to have a length, and an empty sequence of bytes whose length is zero. We also require concat and split functions, which are well-behaved with respect to length. The split function is allowed to fail, for example if the index is out-of-bounds (in the concrete world) or if the index is not exactly at the right position (in the symbolic world). We furthermore require split and concat to be well-behaved with respect to each other. We are as liberal as we can about this, so that it is easily implementable with any symbolic bytes implementation. In particular, we know a split succeeds only when the index is at the boundary of a concatenation.

Writing message formats combinators. We ensure the combinators in §3 only rely on lemmas provided by the type class. This means our proofs are more difficult to conduct, since we cannot assume concat to be associative, we refer the reader to the supplementary material [4] for the full details.

5.3 Improving message formats in DY*

Precise message formats. We are now able to precisely model message formats as they are written in the RFCs (§3.1) by plugging Comparse into DY*. Combined with Comparse's automation, this guarantees that users of DY* can easily and accurately model real-world formats as opposed to sketches of formats.

Non-ambiguity. Non-ambiguity is a built-in feature of the formats in Comparse (§3.1), meaning we immediately satisfy the DY* requirement after integrating Comparse.

Information flow. Given a predicate pred on bytes which is preserved by concatenation and splitting, we define a predicate on high-level messages is_well_formed pred, capturing the fact that every sequence of bytes in the high-level message satisfies the predicate pred. Moreover, we have the property that if is_well_formed pred msg then pred (serialize msg) and if pred buf and parse buf = Some msg then is_well_formed pred msg. This allows us to satisfy the DY* requirements regarding labeling of message, and compose them with Comparse while retaining a high degree of automation.

Impact on DY* examples. We adapted two examples of DY* to use Comparse for their formatting, as shown in Table 1. Before, the protocols relied on hand-written definitions of parsers and serializers, along with manual non-ambiguity proofs, resulting in 148 lines of code devoted to message formats in the NSL example and 141 in the ISO-DH example. With Comparse, this proof effort can be reduced by an order of magnitude (respectively 19 and 24 lines of code) by boiling it down to writing the type of messages, letting Comparse generates combinators and proofs automatically.

Furthermore, with the use of Comparse, we are now able to execute each DY* example both symbolically and concretely, providing two independent forms of debugging. The symbolic traces show the high-level protocol flow, while the concrete traces display the low-level bytestrings obtained by applying all the specified cryptographic and formatting functions.

Impact on MLS verification. We also used Comparse to formalize all the formats of the MLS RFC. In so doing, we prove that the signature confusion attack on MLS draft 12 [47] is now fixed, and provide strong security guarantees for all the formats used in the MLS. Indeed, our formal security proofs form part of a larger formal verification effort for MLS, and were used in a recently published work on the TreeSync sub-protocol [47]. Our proofs enjoy the automation provided by Comparse (§3.6) to write and verify the 82 formats defined in the RFC [8].

Protocol	Nb. formats	RFC LoC	F* LoC	Lemmas	Verif. time
NSL	7	–	19	16	1min
ISO-DH	9	–	24	21	45s
TLS 1.3	51	311	452	105	3min15s
MLS	82	482	624	164	2min45s
cTLS	30	623	608	110	2min45s

Table 1: Evaluation over a set of protocol case studies. Lemmas include non-ambiguity, representation uniqueness lemmas, and disjointness. TLS 1.3 proofs include non-ambiguity with TLS 1.2; cTLS proofs include non-ambiguity with TLS 1.3, and properties of compression/decompression.

6 DISCUSSION

To evaluate the effectiveness of Comparse, we applied it to several case studies, as shown in Table 1, including two examples from DY* (NSL, ISO-DH), TLS 1.3, cTLS and MLS.

Lessons. During the course of this work, we learned several lessons. Our approach is focused on formats used in cryptographic inputs and is justified by the cryptographic assumptions typically used in protocol analysis. We believe this principled approach yields more precise security conditions than other works that focus on parsers and serializers.

Second, our work makes it apparent why one needs to study all of the usages of a given primitive across an entire protocol, in order to rule out the entire class of format confusion attacks.

Third, proof assistants are crucial in making the analysis above tractable. With pen and paper, tracking every usage of a given key across all of TLS 1.3 (and TLS 1.2, because of backwards compatibility) would be impossible. With a carefully crafted library, combinators take care of the bulk of the work, and the library need not grow into a massive software artifact: four well-chosen combinators suffice.

Finally, the format analysis does not need to live in isolation as a separate development. It can be successfully integrated into a general-purpose symbolic security analysis framework (in our case, DY*), paving the way for future evolutions of other tools (e.g. Tamarin or ProVerif) that might make them, too, format-aware.

Limitations. We also identified several limitations throughout our journey in the land of formats. First, there are some real-world, non-ambiguous messages formats with unique representation that cannot be expressed using our combinators. One example is TLSInnerPlaintext, which must be parsed starting from the end. Fortunately, these are few such cases, and we can use the escape hatch we mentioned earlier: it suffices to write the formats by hand, without the distinguished combinators from §3. Because such hand-written formats still use the types from Comparse, they compose with the rest of the framework.

Second, there exist formats which intentionally do not enjoy representation unicity, such as protocol buffers [46]. We cannot account for such formats with Comparse, perhaps suggesting that they should not be used for secure protocols.

7 RELATED WORK

Table 2 recaps the capabilities of the various tools we describe here.

Generic verified formats. EverParse with the QuackyDucky frontend [36] generates efficient C implementation of validators and serializers, proves non-ambiguity and representation unicity. QuackyDucky has limited support for data-dependency, e.g. a union must have its tag immediately preceding it. The work introduces the idea that non-ambiguity and representation unicity are important properties for cryptographic protocols, but does not provide a theoretical justification for it, and does not exhibit a concrete set of recommendations like we do (§2.3).

EverParse3D [43] generates efficient C validators from an expressive format description with good support for data-dependency. They prove validator injectivity, which is related to representation unicity, but is more implementation-centric, since it is a property of the validator, as opposed to the underlying format (the relation).

Narcissus [17] is a Coq library for writing non-ambiguous and non-extensible message formats. It does not consider representation unicity. Narcissus formats are defined using both a state and a relation, whereas we avoid state by relying on our powerful dependent pair combinator. Narcissus also uses combinators, but does not prove general results regarding (say) the non-ambiguity of combinator applications.

[45] defines a deep embedding of data formats into Agda. They relate high-level data types to low-level formats by composing a series of transformations. Lacking support for any sort of frontend format or automatic combinator generation, their format descriptions are not as concise as ours. They reason about non-ambiguity and non-extensibility, but not representation unicity. They also rely on combinators such as dependent pairs. Owing to the nature of the Agda proof assistant, it is unclear to what extent this work can achieve a high degree of proof automation; furthermore, their choice of combinators seems geared towards their IPv4 example and we do not know if their work would scale to e.g. all of the formats of TLS 1.3.

Verification of specific formats. Cheerios [3] is a serialization library for Coq types, relying on combinators. They prove non-ambiguity and non-extensibility with an equation similar to ours (§3.1). They serialize to a Cheerios-specific custom binary data format that is not user-defined, meaning they cannot target real-world, RFC-prescribed formats.

[44] prove a C implementation of a specific ASN.1 format used in the automotive industry. They prove non-ambiguity and representation unicity, using an equation similar to ours. The work does not tackle the general question of proving those properties generically, for a certain class of ASN.1 formats.

[50] builds upon Narcissus [17] to prove verified parsers and serializers for Protocol Buffer 3. They prove non-ambiguity, but not representation unicity, which Protocol Buffer 3 does not enjoy.

Message formats and symbolic security. [32] propose sufficient criteria for secure message formats in a protocol, and prove that an attack on a protocol using concrete formats implies an attack on the protocol with abstract formats. The criterion is that all message formats must be non-ambiguous, moreover they must be pairwise disjoint. In our vocabulary, it means that the protocol must rely on a single, non-ambiguous message format.

Paper	Expressiveness					Properties			Execution		
	Structs	Internal dependency	External dependency	Support extensible formats	Textual format (e.g. XML)	Non-ambiguity	Representation unicity	Disjoint formats	Reference implementation	Efficient implementation	Symbolic execution
Comparse	●	●	●	●	○	●	●	●	●	○	●
Everparse+QD [36]	●	●	○	●	○	●	●	○	●	●	○
Everparse+3D [43]	●	●	◐	●	○	●	●	○	●	●	○
Narcissus [17]	●	◐	○	○	○	●	○	○	●	○	○
vGS17 [45]	●	●	●	○	○	●	○	○	●	○	○
MK14 [32]	●	◐	○	●	●	●	○	◐	○	○	◐
Cheerios [3]	Custom format					●	○	○	●	○	○
V2V [44]	ASN.1					●	●	○	○	●	○
Verified Protobuf [50]	Protocol Buffer v3					●	○	○	●	○	○

Table 2: Related features of other *verified* parser frameworks. We intentionally omit unverified systems.

These criteria are significantly stricter than the ones we propose in §2.3, and we believe they are too strict for most real-world protocols. In particular, composing two unrelated protocols in parallel (e.g. TLS and SSH) violates this property, even though running both at the same time does not impact the security of each protocol. Our criteria (§2.3) are more likely to be true on real-world protocols, are preserved by parallel composition, and can be used in DY* proofs.

Furthermore, their approach is less expressive than ours: they do not support data-dependency such as tagged union (which explains their strict disjointness conditions), and they cannot modularly analyze message formats. Finally, they neither provide a formal analysis tool nor a reference implementation for their formats.

8 CONCLUSION

Comparse is a framework to study the security of formats in cryptographic protocols, allowing the user to prove crucial properties such as non-ambiguity, representation unicity and (absence of) data dependency. We demonstrate the expressiveness and effectiveness of Comparse by using it to specify and verify all the formats in TLS 1.3, MLS, and cTLS. Our formats and their guarantees are compatible with both symbolic and computational cryptographic proofs. We integrate Comparse with DY* and show how the format proofs of Comparse can be composed with symbolic security proofs for a variety of protocols. In particular, our framework has been used as part of a security proof for a key component of MLS [47]. Comparse encodes a strong yet flexible discipline that can help protocol designers easily write formats that are provably secure. Our case study on cTLS shows that cTLS formats are secure, and paves the way to a complete security proof.

ACKNOWLEDGMENTS

This work received funding from the French Government, managed by the ANR under grant agreements ANR-22-PECY-006 and ANR-19-P3IA-0001.

REFERENCES

[1] XML Encryption Syntax and Processing Version 1.1. W3C Recommendation, April 2013.
[2] XML Signature Syntax and Processing Version 2.0. W3C Recommendation, July 2015.
[3] Cheerios, 2016. https://github.com/uwplse/cheerios.
[4] Comparse: Supplementary material, 2023. https://github.com/Inria-Prosecco/comparse-artifact.
[5] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.
[6] Martin R. Albrecht, Sofia Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in matrix. Cryptology ePrint Archive, Paper 2023/485, 2023. https://eprint.iacr.org/2023/485.
[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy (S&P)*, pages 777–795, 2021.
[8] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.
[9] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO*, pages 71–90, 2011.
[10] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.
[11] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. An in-depth symbolic security analysis of the ACME standard. In *ACM SIGSAC Conference on Computer and Communications (CCS)*, pages 2601–2617. ACM, 2021.
[12] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.
[13] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.
[14] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.
[15] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0, 2005.
[16] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 1773–1788, 2017.
[17] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.
[18] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.
[19] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *J. Cryptol.*, 34(4):37, 2021.

[20] James Heather, Gavin Lowe, and Steve A. Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[21] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *IEEE Symposium on Security and Privacy (S&P)*, pages 107–124. IEEE, 2022.

[22] ITU-T. Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks. Recommendation ITU-T X.509, October 2019.

[23] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). IETF RFC 7515, May 2015.

[24] Michael B. Jones and Joe Hildebrand. JSON Web Encryption (JWE). IETF RFC 7516, May 2015.

[25] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (de-)constructing tls 1.3. In *Progress in Cryptology – INDOCRYPT 2015*, pages 85–102, 2015.

[26] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. Multiple handshakes security of tls 1.3 candidates. In *IEEE Symposium on Security and Privacy (SP)*, pages 486–505, 2016.

[27] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer Berlin Heidelberg, 1996.

[28] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F* : Proof automation with SMT, tactics, and metaprograms. In Luís Caires, editor, *Programming Languages and Systems - European Symposium on Programming, ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 30–59. Springer, 2019.

[29] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 62–72, New York, NY, USA, 2012. Association for Computing Machinery.

[30] Catherine A. Meadows. Analyzing the needham-schroeder public key protocol: A comparison of two approaches. In *Computer Security — ESORICS*, pages 351–364. Springer Berlin Heidelberg, 1996.

[31] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[32] Sebastian Mödersheim and Georgios Katsoris. A sound abstraction of the parsing problem. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 259–273, 2014.

[33] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, dec 1978.

[34] Yoav Nir, Simon Josefsson, and Manuel Pégourié-Gonnard. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. RFC 8422, August 2018.

[35] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from Threema: Analysis of a secure messenger. In *Proceedings of the 32th USENIX Conference on Security Symposium*, SEC'23, USA, 2023. USENIX Association.

[36] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1465–1482, USA, 2019. USENIX Association.

[37] E. Rescorla, R. Barnes, H. Tschofenig, and B. Schwartz. Compact TLS 1.3. IETF Internet Draft version 8, March 2023.

[38] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[39] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[40] J. Schaad and August Cellars. Protocol Buffers (proto 3). https://protobuf.dev, July 2008.

[41] J. Schaad and August Cellars. CBOR Object Signing and Encryption (COSE). IETF RFC 8152, July 2017.

[42] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[43] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 31–45, New York, NY, USA, 2022. Association for Computing Machinery.

[44] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. Formal verification of a vehicle-to-vehicle (v2v) messaging system. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 413–429, Cham, 2018. Springer International Publishing.

[45] Marcell van Geest and Wouter Swierstra. Generic packet descriptions: Verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, page 30–40, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72:23, 2008.

[47] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, August 2023.

[48] Wikipedia contributors. Malleability (cryptography) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 4-May-2023].

[49] Pieter Wuille. Dealing with malleability. BIP 62, 2014.

[50] Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 222–233, New York, NY, USA, 2019. Association for Computing Machinery.