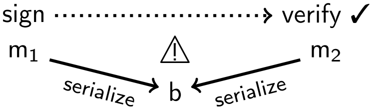
Comparse: Provably Secure Formats for Cryptographic Protocols



Théophile Wallez, *Inria Paris*
Jonathan Protzenko, *Microsoft Research*
Karthikeyan Bhargavan, *Inria Paris, Cryspen*

Message formats in cryptographic protocols

Message formats in MLS: the genesis of Comparse

TreeSync: Authenticated Group Management for Messaging Layer Security
USENIX Security '23

Message formats in MLS: the genesis of Comparse

TreeSync: Authenticated Group Management for Messaging Layer Security
USENIX Security '23

Developed Comparse to precisely study message formats in MLS. . .

Message formats in MLS: the genesis of Comparse

TreeSync: Authenticated Group Management for Messaging Layer Security
USENIX Security '23

Developed Comparse to precisely study message formats in MLS...
...and found an interesting attack exploiting these.

Signature ambiguity in MLS draft 12

TreeSync

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$

Signature ambiguity in MLS draft 12

TreeSync

$$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \underline{\mathsf{serialize}}_{T1}(\mathsf{msg}_1))$$

Signature ambiguity in MLS draft 12

TreeSync

$$\text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T1}(\text{msg}_1))$$
$$\text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T1}(\text{msg}_1))$$

Signature ambiguity in MLS draft 12

TreeSync

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$
$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$

TreeDEM

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$
$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$

Signature ambiguity in MLS draft 12

Same key

TreeSync

$$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$$
$$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$$

TreeDEM

$$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$$
$$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$$

Signature ambiguity in MLS draft 12

Same key

TreeSync

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$
$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$

TreeDEM

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$
$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$

Different types

Signature ambiguity in MLS draft 12

Same key

TreeSync

sig = sign(sk, $\text{serialize}_{T1}(\text{msg}_1)$)
verify(pk, sig, $\text{serialize}_{T1}(\text{msg}_1)$)

TreeDEM

sig = sign(sk, $\text{serialize}_{T2}(\text{msg}_2)$)
verify(pk, sig, $\text{serialize}_{T2}(\text{msg}_2)$)

Different types

What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?

Signature ambiguity in MLS draft 12

Same key

TreeSync

$\text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T1}(\text{msg}_1))$
$\text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T1}(\text{msg}_1))$

TreeDEM

$\text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T2}(\text{msg}_2))$
$\text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T2}(\text{msg}_2))$

Different types

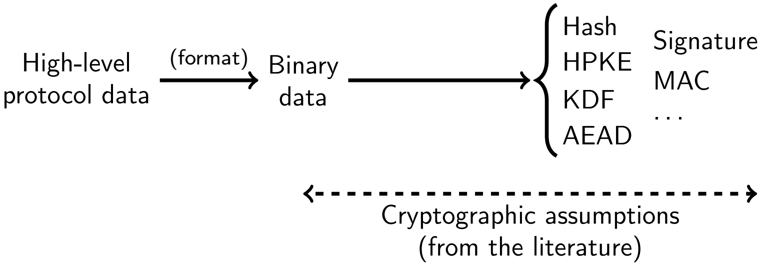
What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?

Possible attack:
TreeDEM signature could be used to forge a signature in TreeSync!

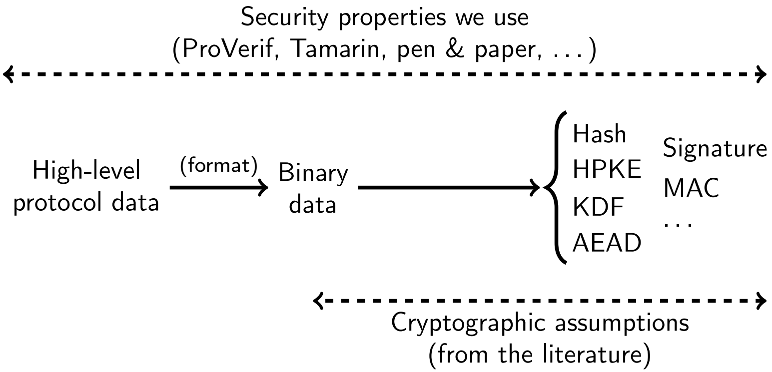
Security critical formats are omnipresent

$$\begin{array}{c}\text{High-level}\\\text{protocol data}\end{array} \xrightarrow{\text{(format)}} \begin{array}{c}\text{Binary}\\\text{data}\end{array} \longrightarrow \begin{cases} \text{Hash} & \text{Signature} \\ \text{HPKE} & \text{MAC} \\ \text{KDF} & \dots \\ \text{AEAD} & \end{cases}$$

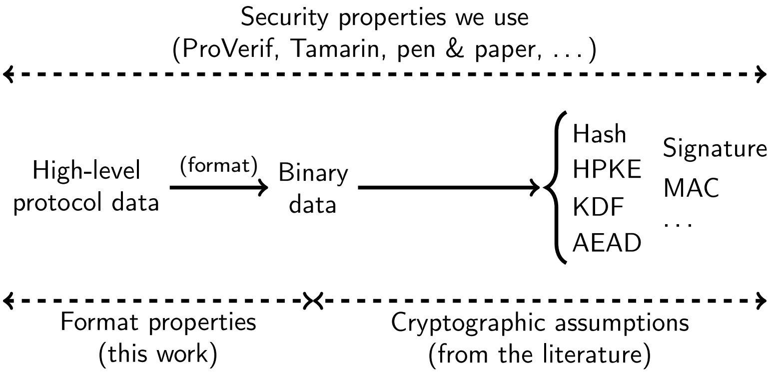
Security critical formats are omnipresent



Security critical formats are omnipresent



Security critical formats are omnipresent



Messages formats play a crucial role in cryptographic protocols security.

We study their impact in two steps:

1. study properties of message formats
2. show how format properties compose with cryptographic assumptions to obtain the security properties we use

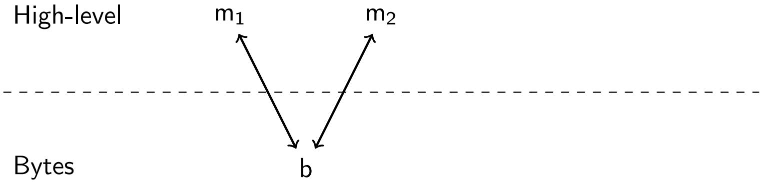
Running example: signatures.

Message formats properties

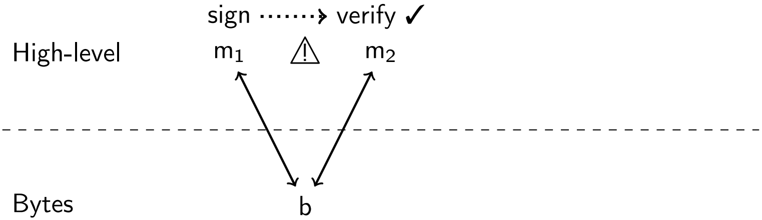
High-level

Bytes

Message formats properties

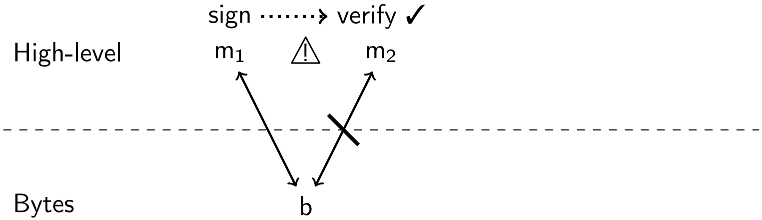


Message formats properties

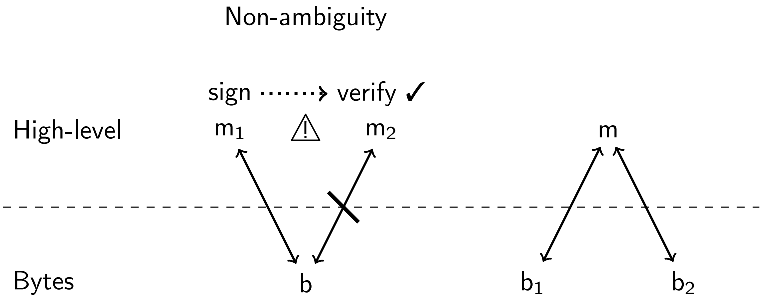


Message formats properties

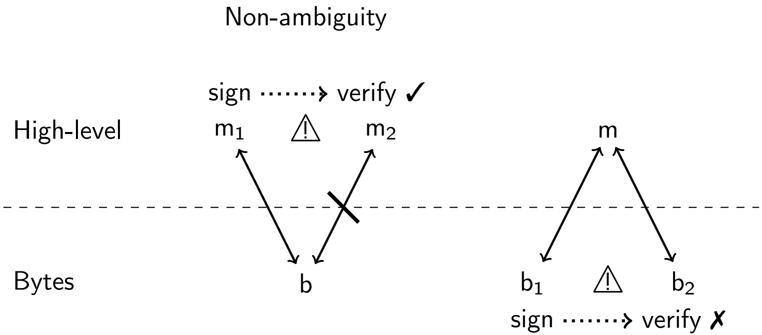
Non-ambiguity



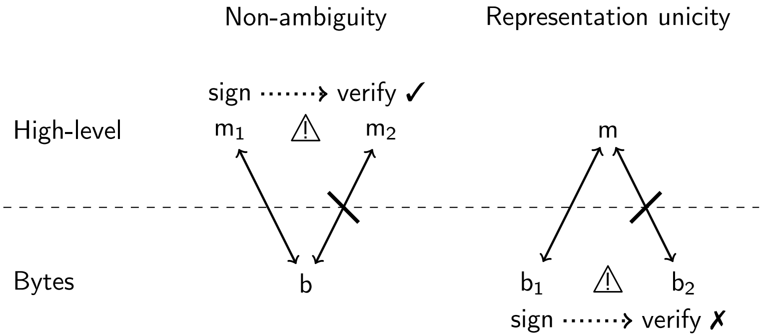
Message formats properties



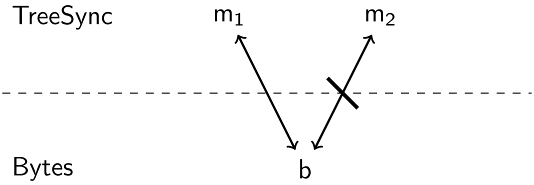
Message formats properties



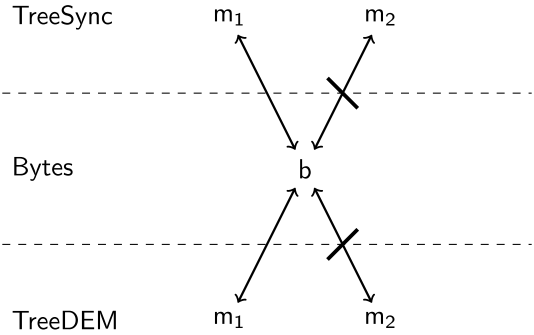
Message formats properties



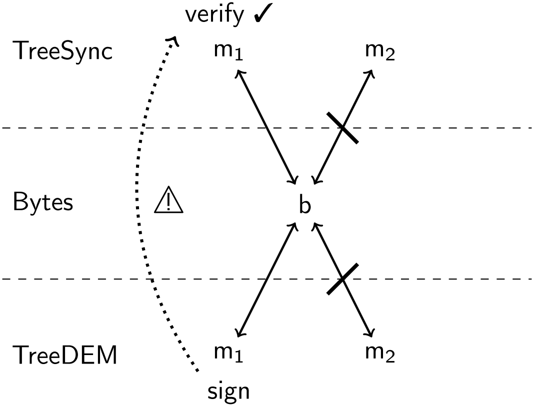
Message formats properties across protocols



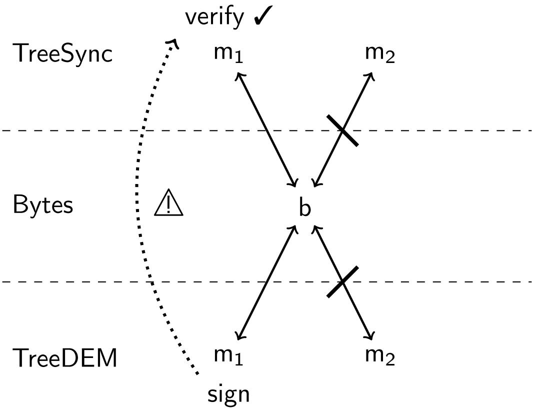
Message formats properties across protocols



Message formats properties across protocols

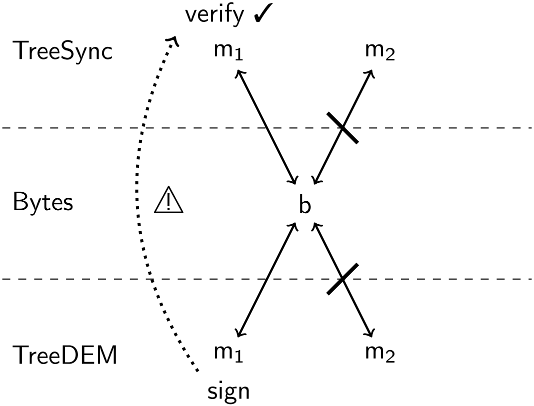


Message formats properties across protocols



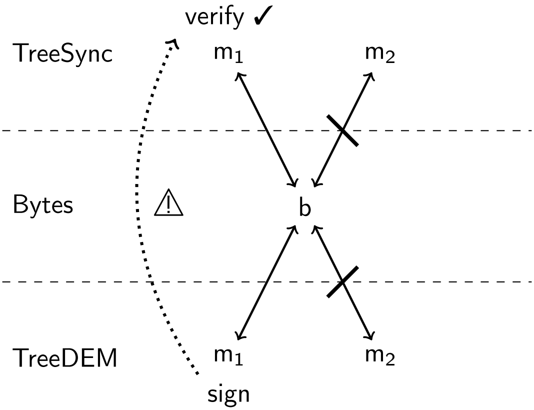
The problem: the meaning of b **depends** on the sub-protocol.

Message formats properties across protocols



The problem: the meaning of b **depends** on the sub-protocol.
One solution: add a tag in b to disambiguate the sub-protocol in use.

Message formats properties across protocols



The problem: the meaning of b **depends** on the sub-protocol.
One solution: add a tag in b to disambiguate the sub-protocol in use.
The result: the meaning of b becomes **self-contained**.

A rigorous approach to domain separation

Bytes	High-level
sign	
verify	
EUF-CMA	

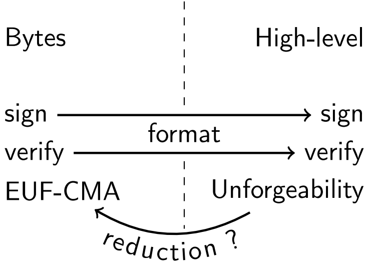
A rigorous approach to domain separation

Bytes	High-level

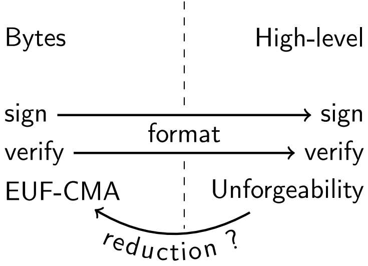
$$\begin{tikzcd}
\text{sign} \arrow[r] & \text{sign} \\
\text{verify} \arrow[r, "\text{format}"] & \text{verify}
\end{tikzcd}$$

EUF-CMA	Unforgeability

A rigorous approach to domain separation

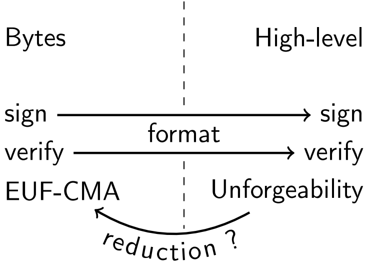


A rigorous approach to domain separation



Reduction if: this format is self-contained and non-ambiguous.

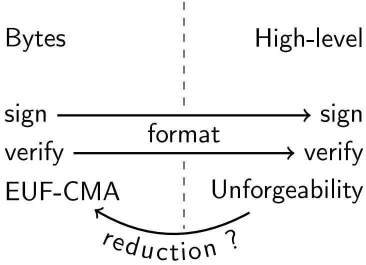
A rigorous approach to domain separation



Design discipline: Each signature key is used with a single format, and

Reduction if: this format is self-contained and non-ambiguous.

A rigorous approach to domain separation

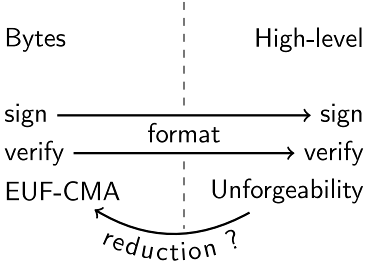


Design discipline: Each signature key is used with a single format, and

Reduction if: this format is self-contained and non-ambiguous.

Note 1: MLS draft 12 failed to obey this design discipline!
This weakness can be used in an attack.

A rigorous approach to domain separation



Design discipline: Each signature key is used with a single format, and

Reduction if: this format is self-contained and non-ambiguous.

Note 1: MLS draft 12 failed to obey this design discipline!
This weakness can be used in an attack.

Note 2: similar design discipline for MAC, AEAD, KDF, ...

Analyzing message formats in a given protocol

Comparse: a proof framework for message formats

We define:

- 4 safe message format combinators (e.g. pairs, lists),
- prove their security properties once and for all

(full details in the paper!)

Comparse: a proof framework for message formats

We define:

- 4 safe message format combinators (e.g. pairs, lists),
- prove their security properties once and for all

(full details in the paper!)

To study a specific format, we can

- define it using the safe combinators,
- obtain its security properties (almost) for free!

Comparse: a proof framework for message formats

We define:

- 4 safe message format combinators (e.g. pairs, lists),
- prove their security properties once and for all

(full details in the paper!)

To study a specific format, we can

- define it using the safe combinators,
- obtain its security properties (almost) for free!

Remaining problem: protocols define many formats, many checks to do :(

Comparse: a proof framework for message formats

We define:

- 4 safe message format combinators (e.g. pairs, lists),
- prove their security properties once and for all

(full details in the paper!)

To study a specific format, we can

- define it using the safe combinators,
- obtain its security properties (almost) for free!

Remaining problem: protocols define many formats, many checks to do :(

Solution: in a proof assistant,

- automate message format generation,
- prove security conditions automatically!

Defining formats with Comparse in F*

TLS 1.3 RFC

```
struct {
  ProtocolVersion legacy_version;
  Random random;
  opaque legacy_session_id<0..32>;


  // ...

  Extension extensions<8..2^16-1>;


} ClientHello;
```

Defining formats with Comparse in F*

TLS 1.3 RFC ⟶ F*

```
struct {
  ProtocolVersion legacy_version;
  Random random;
  opaque legacy_session_id<0..32>;


  // ...

  Extension extensions<8..2^16-1>;


} ClientHello;
```

```
type client_hello = {
  legacy_version: protocol_version;
  random: random;
  legacy_session_id:
    tls_bytes {min=0; max=32};

  // ...

  extensions:
    tls_list extension
    {min=8; max=(pow2 16)-1};
}
```

Defining formats with Comparse in F*

TLS 1.3 RFC ⟶ F*

```
struct {
  ProtocolVersion legacy_version;
  Random random;
  opaque legacy_session_id<0..32>;

  // ...

  Extension extensions<8..2^16-1>;

} ClientHello;
```

```
type client_hello = {
  legacy_version: protocol_version;
  random: random;
  legacy_session_id:
    tls_bytes {min=0; max=32};

  // ...

  extensions:
    tls_list extension
    {min=8; max=(pow2 16)-1};
}
```

Call the Comparse meta-program:

```
%splice [mf_client_hello] (gen_format_for (`%client_hello));
```

and prove automatically non-ambiguity and representation unicity.

Defining formats with Comparse in F*

TLS 1.3 RFC ⟶ F*

```
struct {
  ProtocolVersion legacy_version;
  Random random;
  opaque legacy_session_id<0..32>;

  // ...

  Extension extensions<8..2^16-1>;

} ClientHello;
```

```
type client_hello = {
  legacy_version: protocol_version;
  random: random;
  legacy_session_id:
    tls_bytes {min=0; max=32};

  // ...

  extensions:
    tls_list extension
    {min=8; max=(pow2 16)-1};
}
```

Call the Comparse meta-program:

```
%splice [mf_client_hello] (gen_format_for (`%client_hello));
```

and prove automatically non-ambiguity and representation unicity.
We support several fallbacks if the meta-program fail.

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.
Problem: TLS 1.3 messages are big!

ProtocolVersion legacy_version;
Random random;
opaque legacy_session_id<0..32>;
uint16 cipher_suites_length;
CipherSuite cipher_suites[cipher_suites_length];
opaque legacy_compression_methods<1..2^8-1>;
uint16 extensions_length;
ExtensionType extension type = supported groups;
uint16 extension_length;
NamedGroup supported_groups[extension_length];
ExtensionType extension_type = key_share;
uint16 extension_length;
uint16 client_shares_length;
NamedGroup group = x25519;
uint16 key_size;
opaque key[key_size];
... (other extensions, omitted)

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.
Problem: TLS 1.3 messages are big!

```
ProtocolVersion legacy_version;
Random random;
opaque legacy_session_id<0..32>;
uint16 cipher_suites_length;
CipherSuite cipher_suites[cipher_suites_length];
opaque legacy_compression_methods<1..2^8-1>;
uint16 extensions_length;
ExtensionType extension type = supported groups;
uint16 extension_length;
NamedGroup supported_groups[extension_length];
ExtensionType extension_type = key_share;
uint16 extension_length;
uint16 client_shares_length;
NamedGroup group = x25519;
uint16 key_size;
opaque key[key_size];
... (other extensions, omitted)
```

cTLS compression steps:

1. Trim legacy

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.
Problem: TLS 1.3 messages are big!

```
ProtocolVersion legacy_version;
Random random;
opaque legacy_session_id<0..32>;
uint16 cipher_suites_length;
CipherSuite cipher_suites[cipher_suites_length];
opaque legacy_compression_methods<1..2^8-1>;
uint16 extensions_length;
ExtensionType extension type = supported groups;
uint16 extension_length;
NamedGroup supported_groups[extension_length];
ExtensionType extension_type = key_share;
uint16 extension_length;
uint16 client_shares_length;
NamedGroup group = x25519;
uint16 key_size;
opaque key[key_size];
... (other extensions, omitted)
```

cTLS compression steps:

1. Trim legacy
2. Agree off-band on ciphersuites

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.
Problem: TLS 1.3 messages are big!

ProtocolVersion legacy_version;
Random random;
opaque legacy_session_id<0..32>;
uint16 cipher_suites_length;
CipherSuite cipher_suites[cipher_suites_length];
opaque legacy_compression_methods<1..2^8-1>;
uint16 extensions_length;
ExtensionType extension type = supported groups;
uint16 extension_length;
NamedGroup supported_groups[extension_length];
ExtensionType extension_type = key_share;
uint16 extension_length;
uint16 client_shares_length;
NamedGroup group = x25519;
uint16 key_size;
opaque key[key_size];
... (other extensions, omitted)

cTLS compression steps:

1. Trim legacy
2. Agree off-band on ciphersuites
3. Trim redundant length tags

Case study: cTLS

Use-case: using TLS 1.3 between IoT devices.
Problem: TLS 1.3 messages are big!

ProtocolVersion legacy_version;
Random random;
opaque legacy_session_id<0..32>;
uint16 cipher_suites_length;
CipherSuite cipher_suites[cipher_suites_length];
opaque legacy_compression_methods<1..2^8-1>;
uint16 extensions_length;
ExtensionType extension type = supported groups;
uint16 extension_length;
NamedGroup supported_groups[extension_length];
ExtensionType extension_type = key_share;
uint16 extension_length;
uint16 client_shares_length;
NamedGroup group = x25519;
uint16 key_size;
opaque key[key_size];
... (other extensions, omitted)

cTLS compression steps:

1. Trim legacy
2. Agree off-band on ciphersuites
3. Trim redundant length tags
4. ...

Case study: cTLS format properties

cTLS modifies the message formatting of TLS 1.3.

Questions:

- is it still secure?
- can it be deployed in parallel to TLS 1.3?

Case study: cTLS format properties

cTLS modifies the message formatting of TLS 1.3.

Questions:

- is it still secure?
- can it be deployed in parallel to TLS 1.3?

Short answer:

- we proved that the formats still follow the protocol design disciplines

Long answer:

- §4 in the paper

Conclusion

Our contributions:

- shed light on the importance of formatting in cryptographic protocols
- show our approach on large case studies (TLS 1.3, MLS, cTLS)
- prove security of cTLS formats, pave the way to a full security proof
- theoretically integrates in proofs in the computational model
- concretely integrate with the DY* symbolic proof framework, a core component of an MLS security proof

</> `https://github.com/Inria-Prosecco/comparse-artifact`

`theophile.wallez@inria.fr`

`https://www.twal.org/`

@twallez

Case studies

Protocol	Nb. formats	RFC LoC	F^* LoC	Verif. time
TLS 1.3	51	311	452	3min15s
MLS	82	482	624	2min45s
cTLS	30	623	608	2min45s