ATTESTATIONS OVER TLS 1.3 AND ZKP

Sofía Celi (Brave)
https://eprint.iacr.org/2023/1063

Outline

The design of a complex/distributed system

- How to commit to TLS 1.3 encrypted data
- How to prove in zkp that said data has certain statements:
  - Follow the correct format
  - Prove over AES-GCM/Chacha-Poly as AEAD
  - Prove statements
- Discussion, limitation and applications

Outline

The design of a complex/distributed system

- How to commit to TLS 1.3 encrypted data
- How to prove in zkp that said data has certain statements:
  - Follow the correct format
  - Prove over AES-GCM/Chacha-Poly as AEAD
  - Prove statements
- Discussion and applications

This talk!
But points to future works

Mise-en-scene

- We want to be able to prove *something*
  - Prove so one accesses services
- We cannot send that *something* in the clear → use *zero knowledge*

In the web, we mostly rely on TLS for communication:

- Can we rely on it *to generate proofs*? How to prove *provenance*?
- Such channels commonly transmit trusted information about users behind clients such as proofs of age, social security statuses, and accepted purchase information
- Provenance and privacy concerns

Mise-en-scene

If we want to prove that the field “age” in a JSON-based TLS traffic has a value greater than 1 (a proof of a statement):

- How to commit to the transmitted data over TLS?
- How can be demonstrate that the data conforms to the JSON context-free grammar (*)?
- How can we demonstrate that the field “age” exists at the top level?
- How can we demonstrate that the value associated with “age” matches the grammar rule for integers?
- How can we demonstrate that the value of “age” satisfies a semantic constraint: While grammars handle structural validity, proving that the value of “age” is greater than 1 requires an additional semantic check beyond the grammar rules

Mise-en-scene

If we want to prove that the field “age” in a JSON-based TLS traffic has a value greater than 1 (a proof of a statement):

- **How to commit to the transmitted data over TLS?**
- How can be demonstrate that the data conforms to the JSON context-free grammar (*)?
- How can we demonstrate that the field “age” exists at the top level?
- How can we demonstrate that the value associated with “age” matches the grammar rule for integers?
- How can we demonstrate that the value of “age” satisfies a semantic constraint: While grammars handle structural validity, proving that the value of “age” is greater than 1 requires an additional semantic check beyond the grammar rules

Designated-Commitment TLS (DCTLS)

(or *three-party handshake protocols -3PS-* or *ZKTLS(*)*)

- Modified TLS handshakes that allow **exporting statements** over the **TLS channel** to a **designated verifier**
  - Perform handshakes that secret-share private session data amongst a client and a verifier, and compute the handshake and record-layer phases in two-party computation (2PC)
- Examples:
  - DECO, TLSNotary/PageSigner, TownCrier, Garble-then-Prove, and Janus
  - Zero-knowledge middleboxes
  - Multi-party TLS clients/servers: Oblivious TLS and MPCAuth

Designated-Commitment TLS (DCTLS)

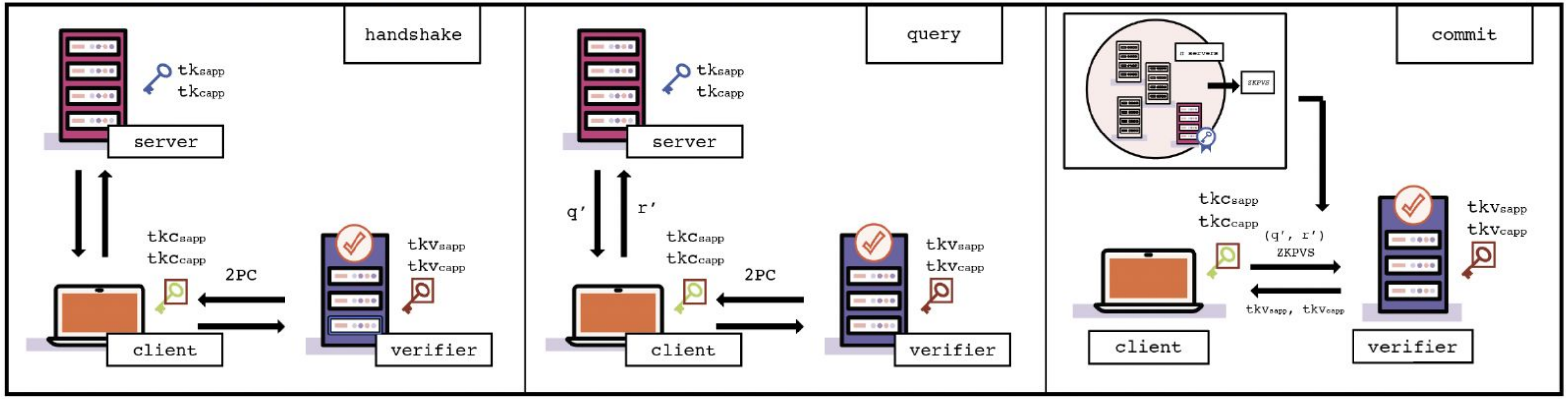
They have some limitations^{TM}:

- No explicit/detailed support for TLS 1.3
    - It is **not trivial** to extend something for TLS 1.2 for TLS 1.3
    - TLS 1.2 will be deprecated soon and the IETF has locked it for any new futures
    - Conversations on TLS 1.4 on the way
- Lack of formal security analysis
    - Close as possible to the TLS 1.3 one
- Efficiency concerns:
    - Unspecified usage of AES-GCM/ChachaPoly
- No client privacy: the identity of the server is revealed
- No open source implementations

- We introduce **DiStefano:** https://github.com/brave-experiments/distefano

Designated-Commitment TLS (DCTLS)

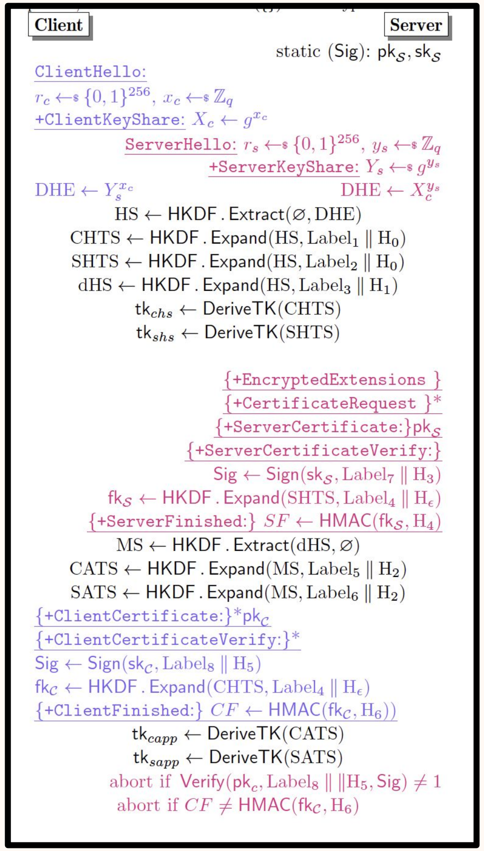
Figure 1. An overview of the DiStefano protocol. In the **handshake** and **query** phases, the client performs the TLS 1.3 handshake and record-layer phases in conjunction with the verifier using 2PC to secret-share traffic keys and other session data for establishing a secure session with the server (secret-shared keys are represented with a square over the key). In the **commitment** phase, the client authenticates the server to the verifier using a zero-knowledge proof of valid TLS signatures (denoted by ZKPVS, see Appendix C), and commits to some encrypted session data, before receiving the verifier's secret TLS session shares.



Designated-Commitment TLS (DCTLS)

TLS 1.3:

- Stronger security properties, the majority of handshake messages are encrypted
  - With the ability of having even more with ECH
- Division between handshake keys and record-layer keys
- Handshake Layer:
  - How can we secret-share session data with a third party?
  - How can we assure that the data is committed to?
- Record Layer:
  - How can we create proofs over committed queries and responses?



Client **Server**

static (Sig): $\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}$

`ClientHello:`
$r_c \leftarrow_\$ \{0,1\}^{256}$, $x_c \leftarrow_\$ \mathbb{Z}_q$
`+ClientKeyShare:` $X_c \leftarrow g^{x_c}$

`ServerHello:` $r_s \leftarrow_\$ \{0,1\}^{256}$, $y_s \leftarrow_\$ \mathbb{Z}_q$
`+ServerKeyShare:` $Y_s \leftarrow_\$ g^{y_s}$

$\mathrm{DHE} \leftarrow Y_s^{x_c}$ $\mathrm{DHE} \leftarrow X_c^{y_s}$

$\mathrm{HS} \leftarrow \mathsf{HKDF}.\mathsf{Extract}(\varnothing, \mathrm{DHE})$
$\mathrm{CHTS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_1 \,\|\, \mathrm{H}_0)$
$\mathrm{SHTS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_2 \,\|\, \mathrm{H}_0)$
$\mathrm{dHS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_3 \,\|\, \mathrm{H}_1)$
$\mathsf{tk}_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS})$
$\mathsf{tk}_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS})$

`{+EncryptedExtensions }`
`{+CertificateRequest }*`
`{+ServerCertificate:}`$\mathsf{pk}_{\mathcal{S}}$
`{+ServerCertificateVerify:}`
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{S}}, \mathrm{Label}_7 \,\|\, \mathrm{H}_3)$
$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{SHTS}, \mathrm{Label}_4 \,\|\, \mathrm{H}_\epsilon)$
`{+ServerFinished:}` $SF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

$\mathrm{MS} \leftarrow \mathsf{HKDF}.\mathsf{Extract}(\mathrm{dHS}, \varnothing)$
$\mathrm{CATS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{MS}, \mathrm{Label}_5 \,\|\, \mathrm{H}_2)$
$\mathrm{SATS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{MS}, \mathrm{Label}_6 \,\|\, \mathrm{H}_2)$

`{+ClientCertificate:}*`$\mathsf{pk}_{\mathcal{C}}$
`{+ClientCertificateVerify:}*`
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{C}}, \mathrm{Label}_8 \,\|\, \mathrm{H}_5)$
$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{CHTS}, \mathrm{Label}_4 \,\|\, \mathrm{H}_\epsilon)$
`{+ClientFinished:}` $CF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6))$

$\mathsf{tk}_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS})$
$\mathsf{tk}_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS})$

abort if $\mathsf{Verify}(\mathsf{pk}_c, \mathrm{Label}_8 \,\|\, \|\mathrm{H}_5, \mathsf{Sig}) \neq 1$
abort if $CF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

Designated-Commitment TLS (DCTLS)

In practice:

- Secret share session keys between **Client and Verifier:**
  - Verifier never has a “complete view” of the shares nor plaintext
  - Verifier commits to the encrypted data: both handshake-layer (from *ServerEncryptedExtensions* onwards) and record-layer messages
- Commit and reveal/prove system:
  - Commit to handshake messages
  - Commit to query/responses on the record layer

Designated-Commitment TLS (DCTLS)

1. Handshake phase:

Server learns the same secret session parameters as in standard TLS 1.3, while the client and verifier learn shares of the session parameters: engage in the core TLS 1.3 protocol using a series of 2PC functionalities.

2. Query phase:

The client sends queries to the server with help from the verifier. Since the session keys are secret-shared, the client and verifier jointly compute the encryptions of these queries in 2PC. The response can then be decrypted using a similar procedure.

3. Commitment phase:

After querying and receiving responses, the client commits to the session by forwarding the ciphertexts to the verifier, and receives the verifier’s session key shares in exchange. The client can verify the integrity of responses and later prove statements about them.

Designated-Commitment TLS (DCTLS): DiStefano

1. Handshake phase:

Server learns the same secret session parameters as in standard TLS 1.3, while the client and verifier learn shares of the session parameters: engage in the core TLS 1.3 protocol using a series of 2PC functionalities.

2. Query phase:

The client sends queries to the server with help from the verifier. Since the session keys are secret-shared, the client and verifier jointly compute the encryptions of these queries in 2PC. The response can then be decrypted using a similar procedure.

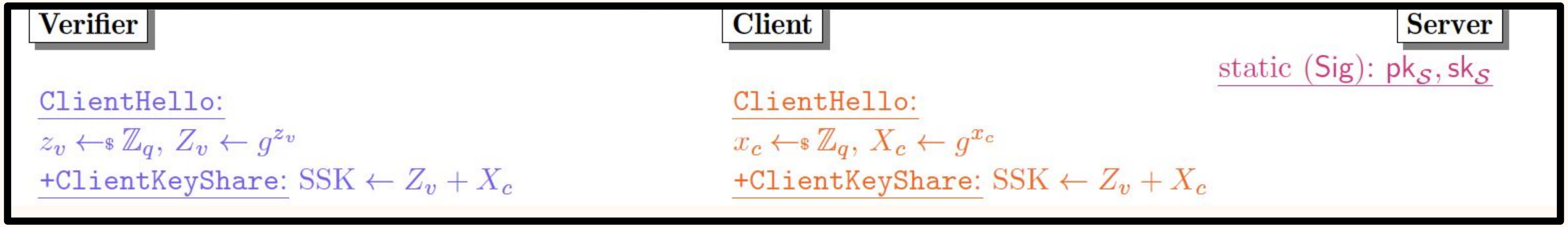
3. Commitment phase:

After querying and receiving responses, the client commits to the session by forwarding the ciphertexts to the verifier, and receives the verifier's session key shares in exchange. The client can verify the integrity of responses and later prove statements about them.

DCTLS: DiStefano

Handshake phase: preprocessing phase

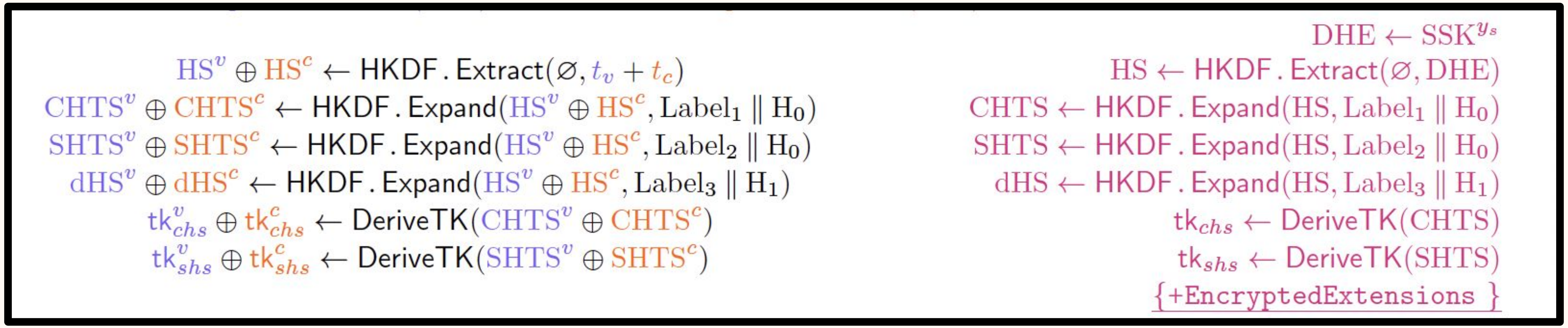
- Client and Verifier act as a single entity
- Server remains the same
- We "reverse" the traditional flow of the handshake so that a key share is computed in advance


Verifier
Client
Server
static (Sig): $\mathsf{pk}_S, \mathsf{sk}_S$
ClientHello:
$z_v \leftarrow_\$ \mathbb{Z}_q,\ Z_v \leftarrow g^{z_v}$
+ClientKeyShare: $\mathsf{SSK} \leftarrow Z_v + X_c$
ClientHello:
$x_c \leftarrow_\$ \mathbb{Z}_q,\ X_c \leftarrow g^{x_c}$
+ClientKeyShare: $\mathsf{SSK} \leftarrow Z_v + X_c$

DCTLS: DiStefano

Handshake phase: key agreement

- Client and Verifier have additive shares: they operate on the "x" coordinate
  - Operate on the field instead of the curve
- A binary-garbled circuit for the 2PC functionality:
  - Handshake keys derivation
  - Encryption and Decryption



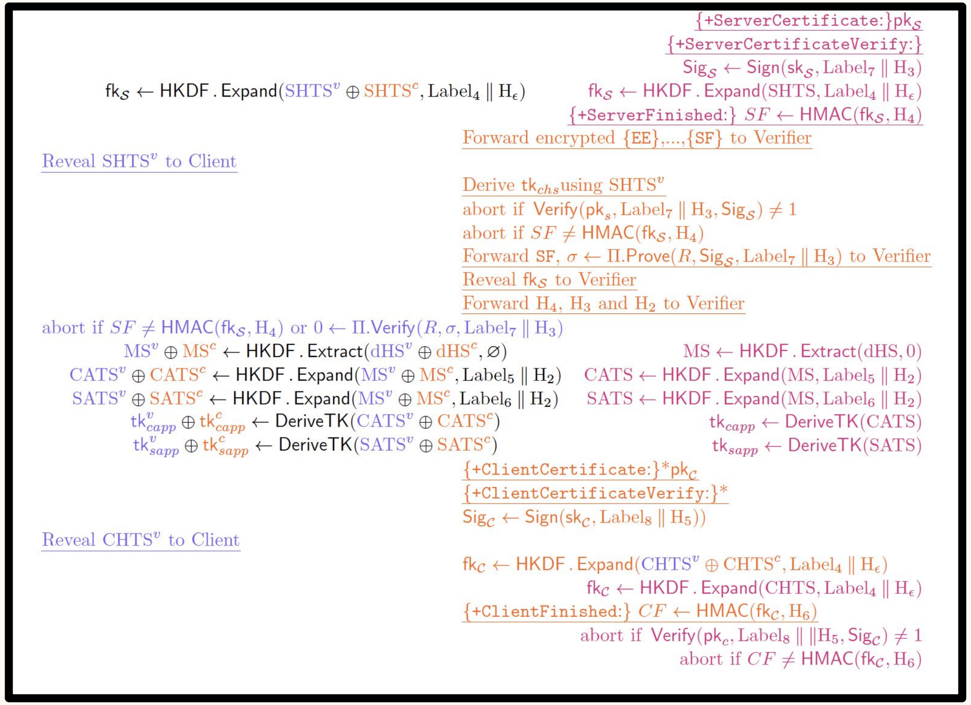
$$\mathrm{HS}^v \oplus \mathrm{HS}^c \leftarrow \mathsf{HKDF.Extract}(\varnothing, t_v + t_c)$$
$$\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_1 \parallel \mathrm{H}_0)$$
$$\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_2 \parallel \mathrm{H}_0)$$
$$\mathrm{dHS}^v \oplus \mathrm{dHS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_3 \parallel \mathrm{H}_1)$$
$$\mathsf{tk}^v_{chs} \oplus \mathsf{tk}^c_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c)$$
$$\mathsf{tk}^v_{shs} \oplus \mathsf{tk}^c_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c)$$

$$\mathrm{DHE} \leftarrow \mathrm{SSK}^{y_s}$$
$$\mathrm{HS} \leftarrow \mathsf{HKDF.Extract}(\varnothing, \mathrm{DHE})$$
$$\mathrm{CHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_1 \parallel \mathrm{H}_0)$$
$$\mathrm{SHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_2 \parallel \mathrm{H}_0)$$
$$\mathrm{dHS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_3 \parallel \mathrm{H}_1)$$
$$\mathsf{tk}_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS})$$
$$\mathsf{tk}_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS})$$

{+EncryptedExtensions }

DCTLS: DiStefano

Handshake phase: authentication and integrity



$\{\texttt{+ServerCertificate:}\}\mathsf{pk}_{\mathcal{S}}$

$\{\texttt{+ServerCertificateVerify:}\}$

$\mathsf{Sig}_{\mathcal{S}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{S}}, \mathrm{Label}_7 \parallel \mathrm{H}_3)$

$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$ $\quad$ $\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$

$\{\texttt{+ServerFinished:}\}\ SF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

Forward encrypted $\{\texttt{EE}\}$,...,$\{\texttt{SF}\}$ to Verifier

Reveal SHTS^v to Client

Derive tk_{chs}using SHTS^v

abort if $\mathsf{Verify}(\mathsf{pk}_s, \mathrm{Label}_7 \parallel \mathrm{H}_3, \mathsf{Sig}_{\mathcal{S}}) \neq 1$

abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

Forward $\mathtt{SF}$, $\sigma \leftarrow \Pi.\mathsf{Prove}(R, \mathsf{Sig}_{\mathcal{S}}, \mathrm{Label}_7 \parallel \mathrm{H}_3)$ to Verifier

Reveal $\mathsf{fk}_{\mathcal{S}}$ to Verifier

Forward H_4, H_3 and H_2 to Verifier

abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$ or $0 \leftarrow \Pi.\mathsf{Verify}(R, \sigma, \mathrm{Label}_7 \parallel \mathrm{H}_3)$

$\mathrm{MS}^v \oplus \mathrm{MS}^c \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}^v \oplus \mathrm{dHS}^c, \varnothing)$ $\quad$ $\mathrm{MS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}, 0)$

$\mathrm{CATS}^v \oplus \mathrm{CATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_5 \parallel \mathrm{H}_2)$ $\quad$ $\mathrm{CATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_5 \parallel \mathrm{H}_2)$

$\mathrm{SATS}^v \oplus \mathrm{SATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_6 \parallel \mathrm{H}_2)$ $\quad$ $\mathrm{SATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_6 \parallel \mathrm{H}_2)$

$\mathsf{tk}^v_{capp} \oplus \mathsf{tk}^c_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS}^v \oplus \mathrm{CATS}^c)$ $\quad$ $\mathsf{tk}_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS})$

$\mathsf{tk}^v_{sapp} \oplus \mathsf{tk}^c_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS}^v \oplus \mathrm{SATS}^c)$ $\quad$ $\mathsf{tk}_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS})$

$\{\texttt{+ClientCertificate:}\}^*\mathsf{pk}_{\mathcal{C}}$

$\{\texttt{+ClientCertificateVerify:}\}^*$

$\mathsf{Sig}_{\mathcal{C}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{C}}, \mathrm{Label}_8 \parallel \mathrm{H}_5))$

Reveal CHTS^v to Client

$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$

$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$

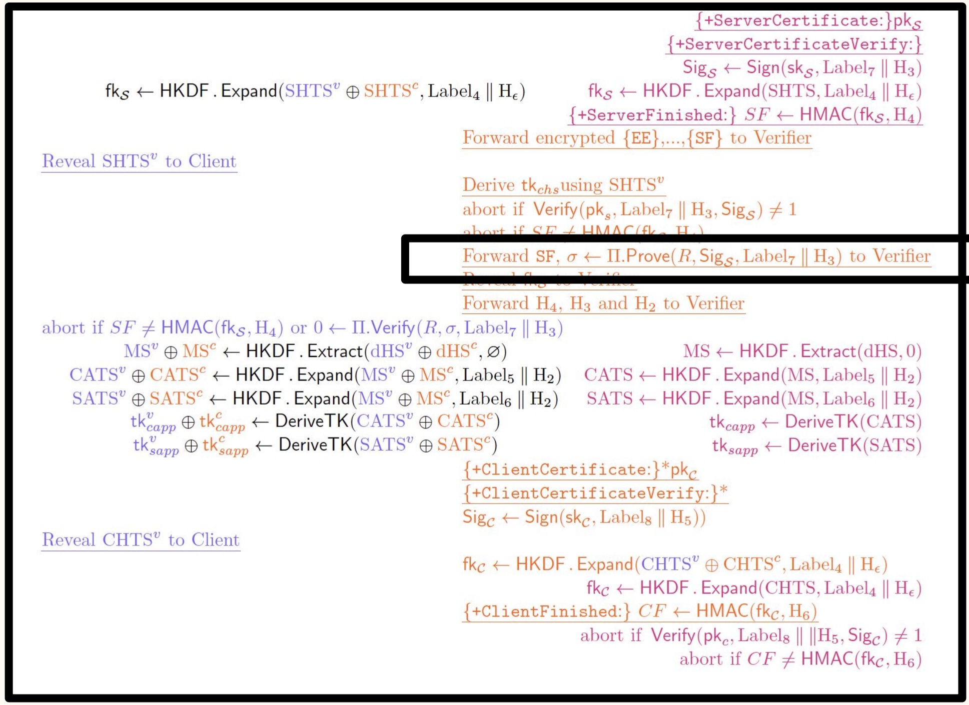
$\{\texttt{+ClientFinished:}\}\ CF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

abort if $\mathsf{Verify}(\mathsf{pk}_c, \mathrm{Label}_8 \parallel \parallel \mathrm{H}_5, \mathsf{Sig}_{\mathcal{C}}) \neq 1$

abort if $CF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

DCTLS: DiStefano

Handshake phase: authentication and integrity



$\{\texttt{+ServerCertificate:}\}\mathsf{pk}_{\mathcal{S}}$

$\{\texttt{+ServerCertificateVerify:}\}$

$\mathsf{Sig}_{\mathcal{S}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{S}}, \mathrm{Label}_7 \| \mathrm{H}_3)$

$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$ $\quad$ $\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$

$\{\texttt{+ServerFinished:}\}$ $SF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

Forward encrypted {EE},...,{SF} to Verifier

Reveal SHTS^v to Client

Derive tk_{chs}using SHTS^v

abort if $\mathsf{Verify}(\mathsf{pk}_s, \mathrm{Label}_7 \| \mathrm{H}_3, \mathsf{Sig}_{\mathcal{S}}) \neq 1$

abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

Forward SF, $\sigma \leftarrow \Pi.\mathsf{Prove}(R, \mathsf{Sig}_{\mathcal{S}}, \mathrm{Label}_7 \| \mathrm{H}_3)$ to Verifier

Reveal $\mathsf{fk}_{\mathcal{S}}$ to Verifier

Forward H_4, H_3 and H_2 to Verifier

abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$ or $0 \leftarrow \Pi.\mathsf{Verify}(R, \sigma, \mathrm{Label}_7 \| \mathrm{H}_3)$

$\mathrm{MS}^v \oplus \mathrm{MS}^c \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}^v \oplus \mathrm{dHS}^c, \varnothing)$ $\quad$ $\mathrm{MS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}, 0)$

$\mathrm{CATS}^v \oplus \mathrm{CATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_5 \| \mathrm{H}_2)$ $\quad$ $\mathrm{CATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_5 \| \mathrm{H}_2)$

$\mathrm{SATS}^v \oplus \mathrm{SATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_6 \| \mathrm{H}_2)$ $\quad$ $\mathrm{SATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_6 \| \mathrm{H}_2)$

$\mathsf{tk}^v_{capp} \oplus \mathsf{tk}^c_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS}^v \oplus \mathrm{CATS}^c)$ $\quad$ $\mathsf{tk}_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS})$

$\mathsf{tk}^v_{sapp} \oplus \mathsf{tk}^c_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS}^v \oplus \mathrm{SATS}^c)$ $\quad$ $\mathsf{tk}_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS})$

$\{\texttt{+ClientCertificate:}\}^*\mathsf{pk}_{\mathcal{C}}$

$\{\texttt{+ClientCertificateVerify:}\}^*$

$\mathsf{Sig}_{\mathcal{C}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{C}}, \mathrm{Label}_8 \| \mathrm{H}_5))$

Reveal CHTS^v to Client

$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$

$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$

$\{\texttt{+ClientFinished:}\}$ $CF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

abort if $\mathsf{Verify}(\mathsf{pk}_c, \mathrm{Label}_8 \| \|\mathrm{H}_5, \mathsf{Sig}_{\mathcal{C}}) \neq 1$

abort if $CF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

Prove that the signature is valid in regards to a public set of keys, without revealing which one: *ZKPVS*

Sofia Celi, Shai Levin, Joe Rowell, *CDLS: Proving Knowledge of Committed Discrete Logarithms with Soundness*, International Conference on Cryptology in Africa 2024 (AfricaCrypt2024): https://eprint.iacr.org/2023/1595

DCTLS: DiStefano → encryption/decryption

TLS 1.3: **AES-GCM** and ChachaPoly in a AEAD setting (*focusing in AES-GCM*):

- Remember: AES is a block cipher with an authenticity tag
- Both Client and Verifier learn all AES-GCM ciphertext blocks $C_i = M_i \oplus AES.Enc(k, IV + i)$ produced by the server, where the session key k is secret-shared
- Problem: the client has to create a commitment [1] to the received ciphertext blocks without revealing their key share:
    - Non-committing attack: $C_i = M'_i \oplus AES.Enc(k', IV + i)$
    - Validate the tag in 2PC: (C_i, τ)
    - Commit to masks (b_i), and send commitments to them and the key (in *2PC AES-Decrypt*):
        - e_i = AES.Enc(k, IV + i) and $E_i = e_i + b_i$
        - Verifier reveals their k
        - $M_i = C_i \oplus \mathbf{AES.Enc(k_c + k_v, IV_c + i)}$)$i \in [n]$ to the Client
        - ($E_i = \mathbf{AES.Enc(k_c + k_v, IV_c + i)_i} \oplus b_i$)$i \in [n]$ to the Verifier → commitment
        - Client can forward now blocks e_i

[1] P. Grubbs, J. Lu, and T. Ristenpart, *Message franking via committing authenticated encryption* in CRYPTO 2017

DCTLS: DiStefano

Implementation

- Implementation in C++ in BoringSSL:
  - 14k lines of code
- Evaluated in a LAN and WAN setting
  - LAN: we use a consumer-grade device (a Macbook air M1 with 8GB of RAM) for the Client, and a server-grade device (an Intel Xeon Gold 6138 with 32GB of RAM) for the Verifier and Server. All communication used in TLS 1.3 was carried out using a single thread over a 1 Gbps network with a latency of around *16 ms*.
  - WAN: we use two AWS EC2 “t2.2xlarge” machines: one for the Client located in Spain, and one for the Verifier and the Server in Ohio, USA, where the median latency is estimated at *100 ms*.
  - Timings and bandwidth measurements are computed as the mean of 50 samples, and are represented in milliseconds and mebibytes, respectively (1 MiB is 220 bytes).

DCTLS: DiStefano

Implementation

- < 1 s and ≤ 80 KiB to execute the *online phase:*
  - Less than TLS handshake timeout times which, while configurable, are typically between 10 and 20 seconds

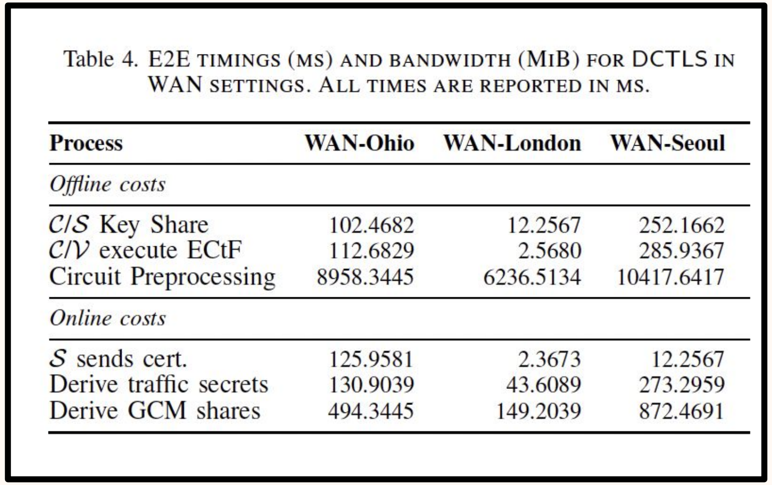


Table 4. E2E TIMINGS (MS) AND BANDWIDTH (MIB) FOR DCTLS IN WAN SETTINGS. ALL TIMES ARE REPORTED IN MS.

Process	**WAN-Ohio**	**WAN-London**	**WAN-Seoul**
Offline costs			
$\mathcal{C}/\mathcal{S}$ Key Share	102.4682	12.2567	252.1662
$\mathcal{C}/\mathcal{V}$ execute ECtF	112.6829	2.5680	285.9367
Circuit Preprocessing	8958.3445	6236.5134	10417.6417
Online costs			
$\mathcal{S}$ sends cert.	125.9581	2.3673	12.2567
Derive traffic secrets	130.9039	43.6089	273.2959
Derive GCM shares	494.3445	149.2039	872.4691

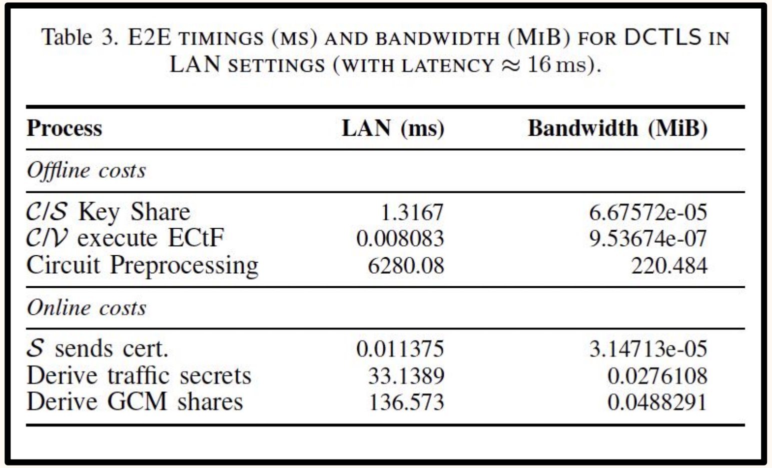


Table 3. E2E TIMINGS (MS) AND BANDWIDTH (MIB) FOR DCTLS IN LAN SETTINGS (WITH LATENCY $\approx$ 16 ms).

Process	**LAN (ms)**	**Bandwidth (MiB)**
Offline costs		
$\mathcal{C}/\mathcal{S}$ Key Share	1.3167	6.67572e-05
$\mathcal{C}/\mathcal{V}$ execute ECtF	0.008083	9.53674e-07
Circuit Preprocessing	6280.08	220.484
Online costs		
$\mathcal{S}$ sends cert.	0.011375	3.14713e-05
Derive traffic secrets	33.1389	0.0276108
Derive GCM shares	136.573	0.0488291

Note

- This does require changes at the TLS library level
  - For security reasons, those libraries do not allow direct access to a majority of functions
- RFC9421(https://datatracker.ietf.org/doc/html/rfc9421) (Signed HTTP headers):
  - Context is CDNs: the TLS connection from the client (eyeball) will be terminated at the edge, and new HTTP headers can be added for the internal CDN connections will be added
  - In order to preserve the truthness of the HTTP messages sent by the client, the messages are also signed so eventually the origin server is assured it is a "true" message from the client (CDN did not removed/added any needed header)
- JSON Web Signature tokens (https://datatracker.ietf.org/doc/html/rfc7515)

- Restriction: any kind of signature can be used to track a user across sessions (like third-party cookies)
  - Signature as as a cross-site tracking identifier

Designated-Commitment TLS (DCTLS): DiStefano

1. Handshake phase:

Server learns the same secret session parameters as in standard TLS 1.3, while the client and verifier learn shares of the session parameters: engage in the core TLS 1.3 protocol using a series of 2PC functionalities.

2. Query phase:

The client sends queries to the server with help from the verifier. Since the session keys are secret-shared, the client and verifier jointly compute the encryptions of these queries in 2PC. The response can then be decrypted using a similar procedure.

3. Commitment phase:

After querying and receiving responses, the client commits to the session by forwarding the ciphertexts to the verifier, and receives the verifier's session key shares in exchange. The client can verify the integrity of responses and later prove statements about them.

DCTLS: DiStefano ⟶ queries

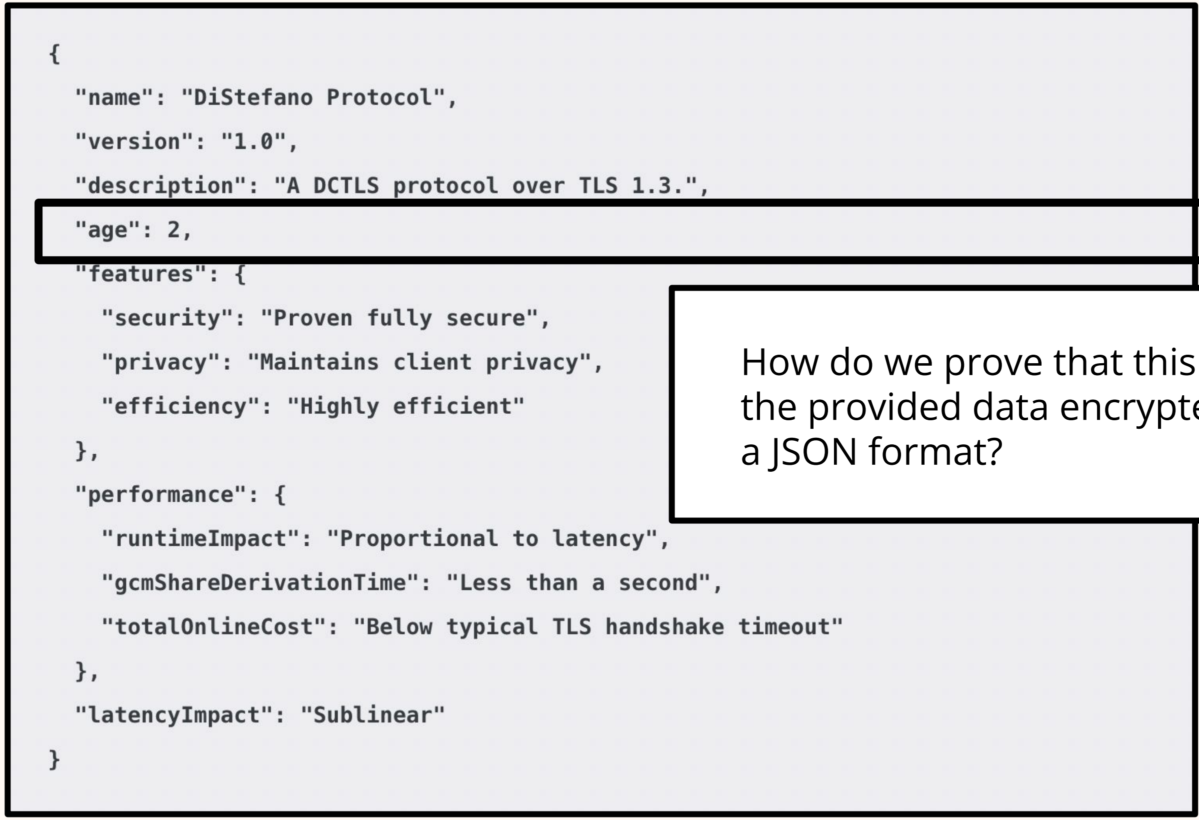
Record phase: proofs over transmitted data

- DiStefano:
  - Ability to create commitments to AES-GCM encrypted data transmitted over TLS 1.3:
    - Allows for *selective opening:* we consider this intrusive and hence will not use it
  - Can easily be extended to ChachaPoly ⟶ Future work!
- TLS 1.3: works mostly over HTTP with HTML/JSON types
- Other formats like XML, binary data, or proprietary protocols can also be transmitted securely over TLS
- HTTP method: query and response

DCTLS: DiStefano ⟶ queries

```
{
  "name": "DiStefano Protocol",
  "version": "1.0",
  "description": "A DCTLS protocol over TLS 1.3.",
  "age": 2,
  "features": {
    "security": "Proven fully secure",
    "privacy": "Maintains client privacy",
    "efficiency": "Highly efficient"
  },
  "performance": {
    "runtimeImpact": "Proportional to latency",
    "gcmShareDerivationTime": "Less than a second",
    "totalOnlineCost": "Below typical TLS handshake timeout"
  },
  "latencyImpact": "Sublinear"
}
```

DCTLS: DiStefano ⟶ queries



```
{
  "name": "DiStefano Protocol",
  "version": "1.0",
  "description": "A DCTLS protocol over TLS 1.3.",
  "age": 2,
  "features": {
    "security": "Proven fully secure",
    "privacy": "Maintains client privacy",
    "efficiency": "Highly efficient"
  },
  "performance": {
    "runtimeImpact": "Proportional to latency",
    "gcmShareDerivationTime": "Less than a second",
    "totalOnlineCost": "Below typical TLS handshake timeout"
  },
  "latencyImpact": "Sublinear"
}
```

How do we prove that this field exist in the provided data encrypted via AES with a JSON format?

Mise-en-scene

If we want to prove that the field “age” in a JSON-based TLS traffic has a value greater than 1 (a proof of a statement):

- How to commit to the transmitted data over TLS?
- **How can be demonstrate that the data conforms to the JSON context-free(*) grammar?**
- **How can we demonstrate that the field “age” exists at the top level?**
- **How can we demonstrate that the value associated with “age” matches the grammar rule for integers?**
- How can we demonstrate that the value of “age” satisfies a semantic constraint: While grammars handle structural validity, proving that the value of “age” is greater than 1 requires an additional semantic check beyond the grammar rules

Putting it all together

The design of a complex/distributed system

- *DiStefano*: provides already commitments (via AEAD) to encrypted blocks
- Prove over the AES/ChachaPoly blocks/stream:
  - That the JSON is well-formed
  - That the key exists and it is in the correct “path”
  - That the value associated with the key has a property:
    - is an integer
    - is bigger than

Putting it all together

The design of a complex/distributed system

- *DiStefano*: provides already comm
- Prove over the AES/ChachaPoly b
  - That the JSON is well-formed
  - That the key exists and it is i
  - That the value associated wi
    - is an integer
    - is bigger than

Upcoming future work!
Notice that:

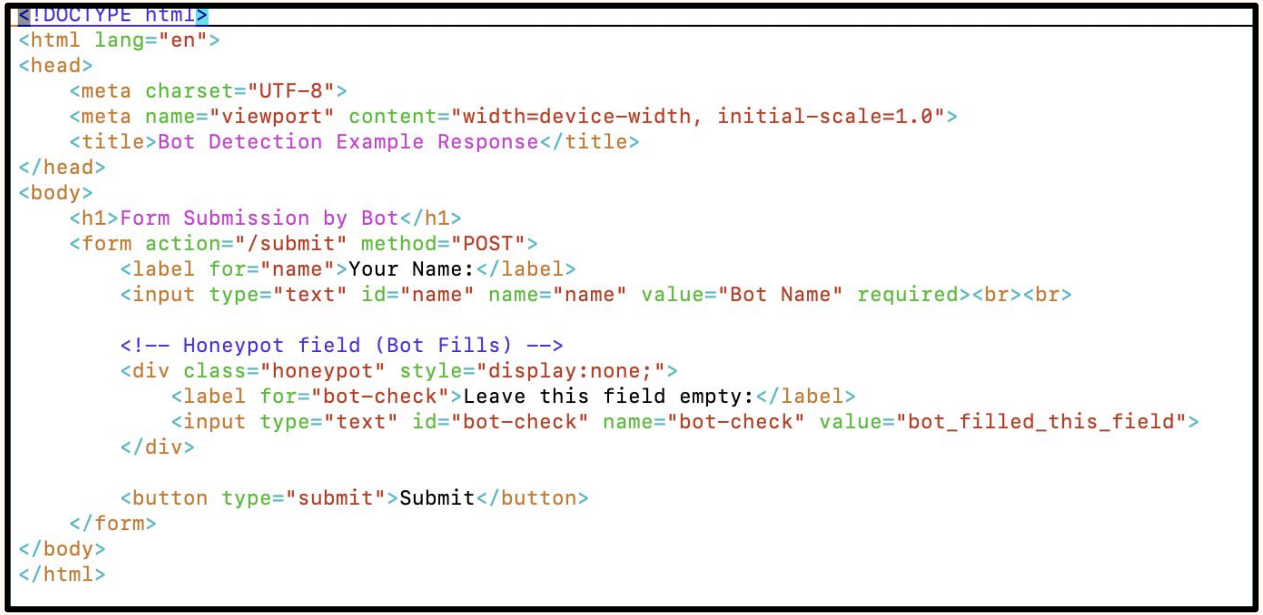
- We can leverage the block structure of AES

Why is this useful for the web?

Anti-fraud checks:

- Attest that the honeypot field was filled
- Do not reveal the honeypot details

- Bank statements, identity documents



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Bot Detection Example Response</title>
</head>
<body>
    <h1>Form Submission by Bot</h1>
    <form action="/submit" method="POST">
        <label for="name">Your Name:</label>
        <input type="text" id="name" name="name" value="Bot Name" required><br><br>

        <!-- Honeypot field (Bot Fills) -->
        <div class="honeypot" style="display:none;">
            <label for="bot-check">Leave this field empty:</label>
            <input type="text" id="bot-check" name="bot-check" value="bot_filled_this_field">
        </div>

        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Why is this useful for the web?

Request MTLS or bilateralTLS (https://datatracker.ietf.org/meeting/122/materials/slides-122-tls-request-mtls-00)

- Many bots come from public known clouds
- Bots distinguish themselves by setting a special user agent (user-agent=”web crawler”) → very easy to forge
- Ask for a certificate (send the “request certificate” message) from the client
    - But not from all clients, as some honest clients will choque if they don’t have a certificate
    - Might not cover all cases: AI-agents, remote isolated-browsers

Why is this useful for the web?

Attestation of device’s age (https://github.com/antifraudcg/proposals/issues/15, https://docs.google.com/presentation/d/1IgNRhtV_jEM7KHRO4P0QAzgvjyYWqYBnzSj_0ToTIpY/edit?usp=sharing)

- An attacker will create many headless browsers
  - Those browsers are young
- However, disclosing the age of a device can end in cross-site tracking
  - Use “signed” interactions that allow to create proofs of the lifetime
  - However, each proposal has to be carefully analysed

Why is this useful for the web?

- No “easy solution”
  - Catching bots depends on many signals:
    - User-agent and identity
    - User-agent and device lifetime
    - Honey-pot fields
    - Signals of “humanity”

However...

- Attesting fraud is never easy
- Restrictive systems:
    - Requirements to show financial liquidity
    - Attestation of humanity → anti-fraud systems
    - Create a system of rules where the ones without it are limited in access
    - This is not a *one-size-fits-all* solution
- Can enable surveillance:
    - How can we enforce that verifiers only attest certain statements?

THANK YOU!

@claucece
https://sofiaceli.com/