EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Communication Networks

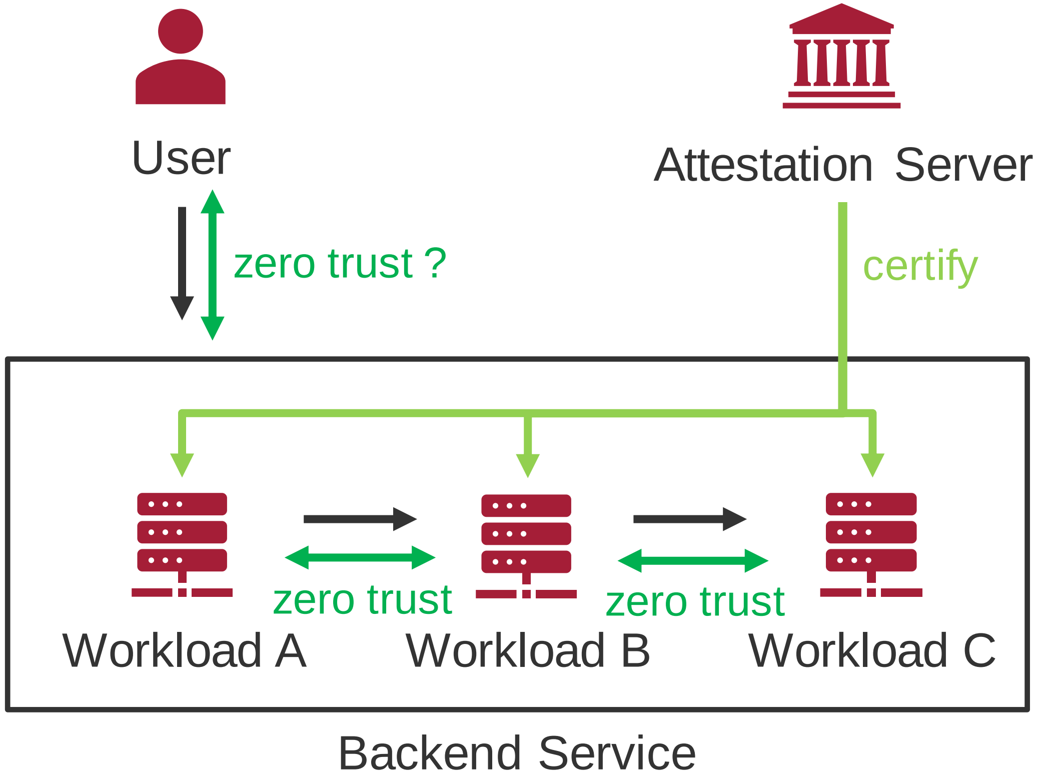
End-to-End Identities for Humans and Machines

By Jonas Primbs M.Sc., Chair of Communication Networks, University of Tübingen, Germany

http://kn.inf.uni-tuebingen.de

Workload Identities

- WIMSE WG is working on workload identities
- Goals of a workload identity
  - Enable zero trust in backends
  - Certify newly spawned workload instances
  - Uniquely identify workload instances
  - Point-to-point authentication of microservices
- Why stop at backends?
  - Zero trust also needs user authentication!
- **Goal**: enable end-to-end authentication between users and workloads!



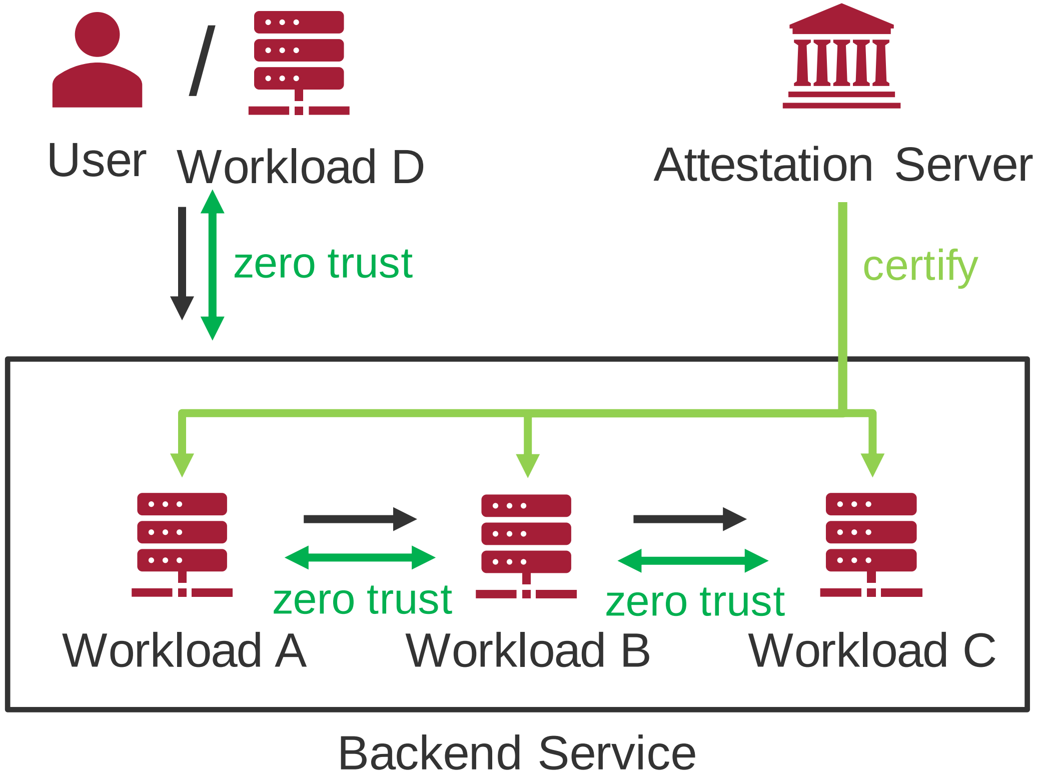
Workload Identities

- WIMSE WG is working on workload identities

- Goals of a workload identity
  - Enable zero trust in backends
  - Certify newly spawned workload instances
  - Uniquely identify workload instances
  - Point-to-point authentication of microservices

- Why stop at backends?
  - Zero trust also needs user authentication!

- **Goal**: enable end-to-end authentication between users and workloads!
- **Advantages**:
  - Same API for users and workloads
  - More security for users



Terms

- **Workload** = Backend (Micro)service, e.g., VM, Container, Serverless Function, …
  - Provides a network-faced interface, e.g., REST API, Web Socket, WebRTC, MQTT, Kafka, gRPC, …
- **Workload Identity** = Certificate for cryptographic authentication, e.g., X.509 cert, sender-constraint token, …

Workload

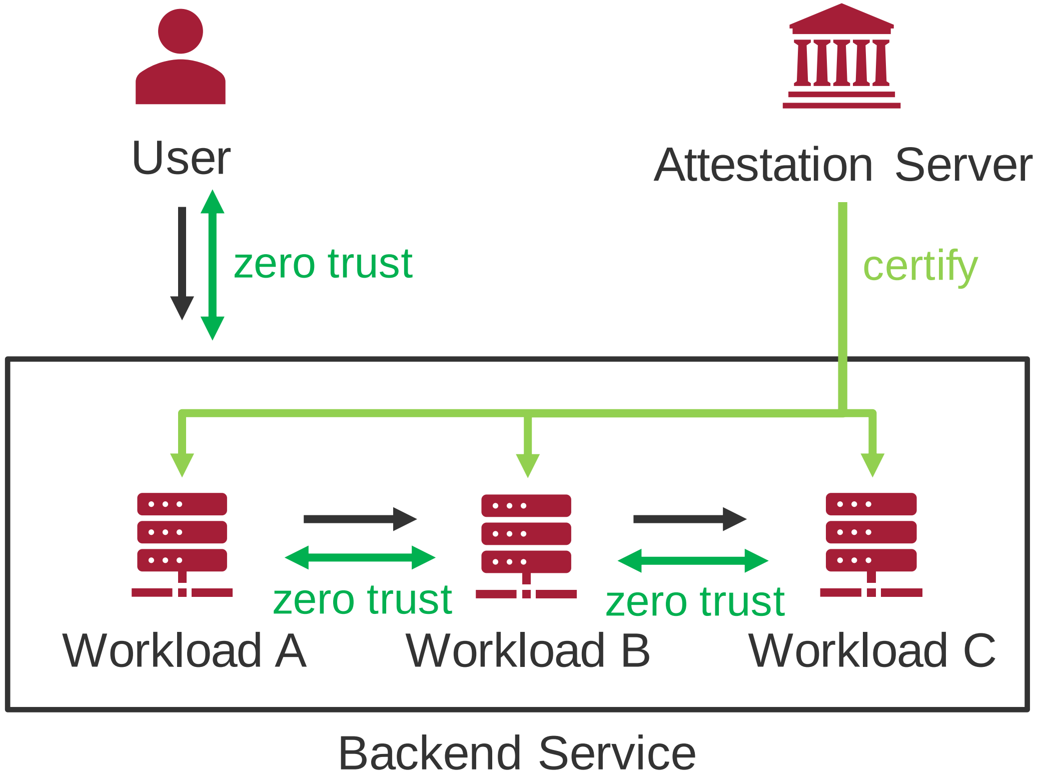
- **User** = Client, e.g., native app, web app, voice service, …
  - Communicates to the network-faced interface of a workload, e.g., REST API, Web Socket, …
- **User Identity** = Client certificate of the user, e.g., X.509 cert, sender-constraint Access Token, …

User

- **Service** = Group of workloads

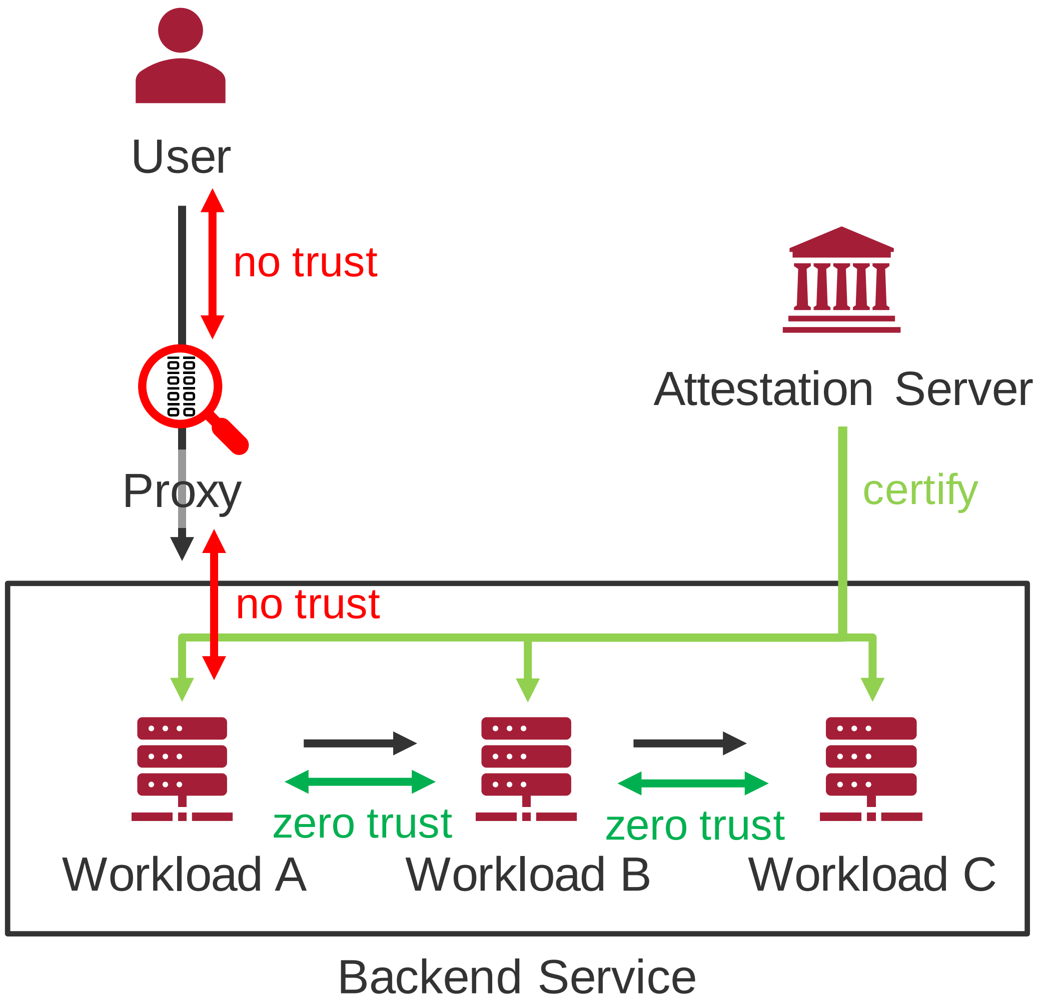
Zero Trust with Clients?

- What is wrong with client-to-server authentication?
  - We already have TLS, mutual TLS, HTTP Message Signatures, Bearer Tokens, sender-constraint tokens, FIDO2 / Passkeys, etc. !
- Do we really need more?



Zero Trust with Clients!

- What is wrong with client-to-server authentication?
  - We already have TLS, mutual TLS, HTTP Message Signatures, Bearer Tokens, sender-constraint tokens, FIDO2 / Passkeys, etc. !
- Do we really need more?
  - Yes!
- (Reverse) Proxies terminate TLS
  - Breaks client-to-server confidentiality
    - Proxy provider sees clear-text credentials
  - Breaks mutual TLS connections
    - Workload must trust the reverse proxy
  - Breaks some FIDO2 / Passkey features
    - TLS Channel binding of WebAuthN not possible

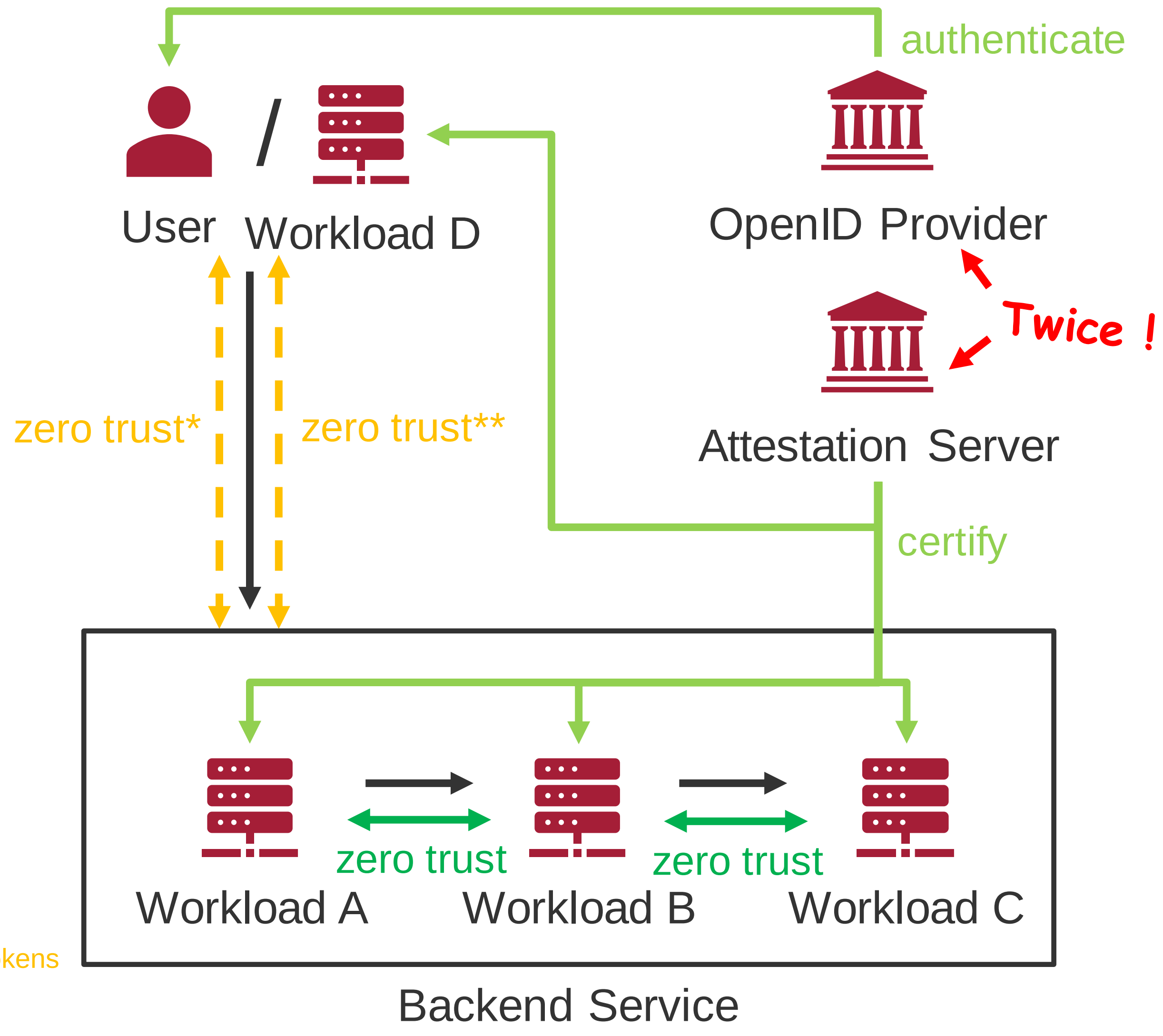


Zero Trust with Clients and Servers!

- What is wrong with client-to-server authentication?
  - We already have TLS, mutual TLS, HTTP Message Signatures, Bearer Tokens, sender-constraint tokens, FIDO2 / Passkeys, etc. !
- Do we really need more?
  - Yes!

Fourth !

- ~~Twice~~ the effort for user and workload usage
  - Users' clients must be authenticated via OIDC or authorized via OAuth 2
    - Client-to-server authentication via bearer or sender-constraint token
  - Workloads must be certified by Attestation Server
    - Workload identity as X.509 cert (mTLS) or bearer Token (JWT)

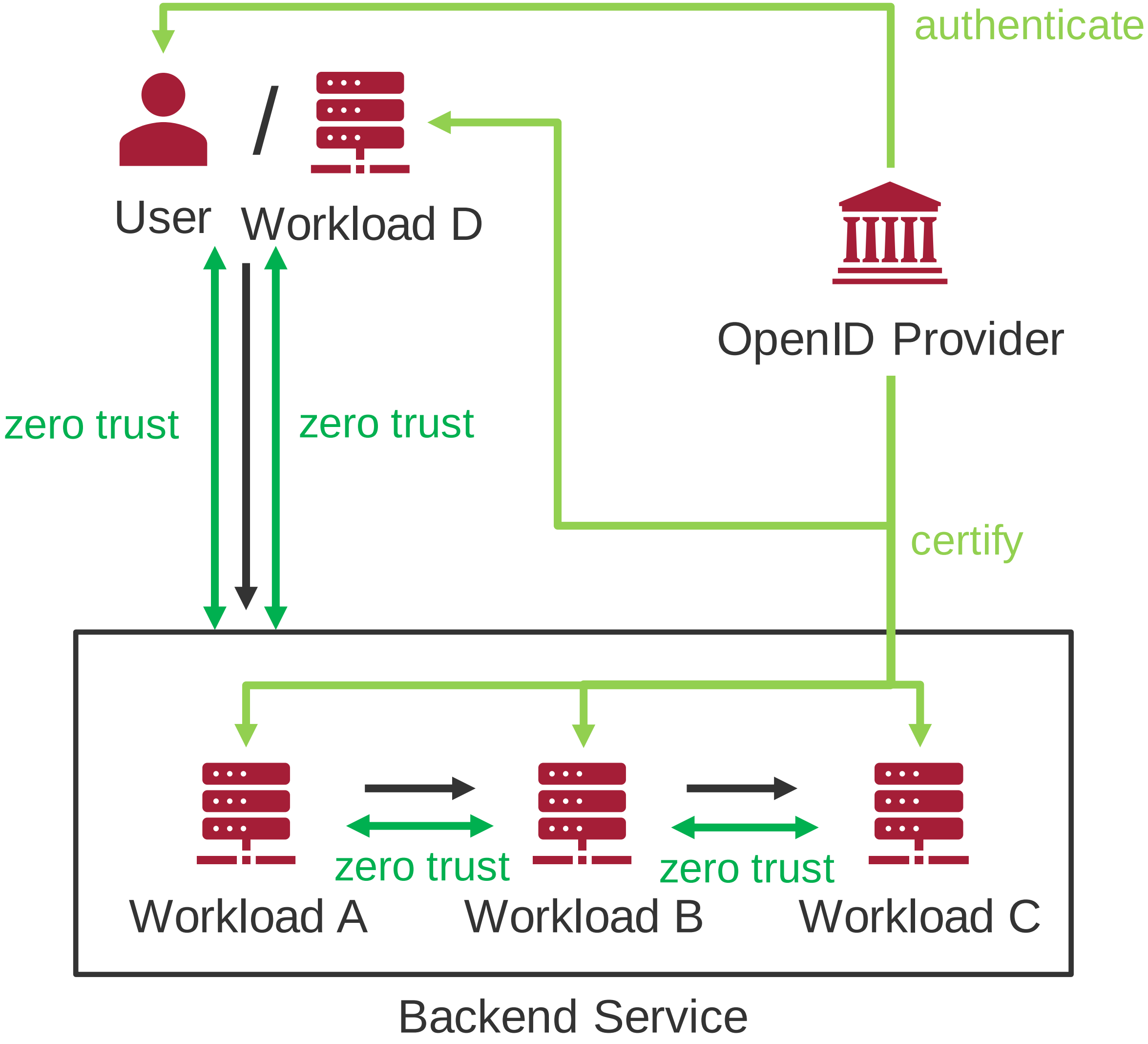


* only with sender-constraint tokens

** only with mTLS

Simple Zero Trust with Clients and Servers!

- ▶ What is wrong with client-to-server authentication?
  - We already have TLS, mutual TLS, HTTP Message Signatures, Bearer Tokens, sender-constraint tokens, FIDO2 / Passkeys, etc. !
- ▶ Do we really need more?
  - Yes!
- ▶ Once ! ~~Twice~~ the effort for user and workload usage
  - Users’ clients must be authenticated via OIDC or authorized via OAuth 2
    - – Client-to-server authentication via ~~bearer or~~ sender-constraint token
  - Workloads must be certified by Attestation Server
    - – Workload identity as ~~X.509 cert (mTLS) or bearer~~ sender-constraint Token (JWT)



Identity Certification Token (ICT)

- Sender-constraint JWTs are the solution!
  - JWT-equivalent for X.509 certificates on the application layer
    - Works through (reverse) proxies!
  - Standardized in RFC 7800
    - Library and OpenID Provider implementations already exist!
  - Flexible data structure (JSON) for payload
    - “cnf” claim contains user’s / workload’s public key
    - “iss” claim contains OpenID Provider’s / Attestation Server’s base URL
    - “exp” contains expiration date
    - Other standardized claims from OAuth 2, JWT, OIDC, etc. available!
- Certifies user / workload identity
  - Called “**Identity Certification Token (ICT)**”

- Header:

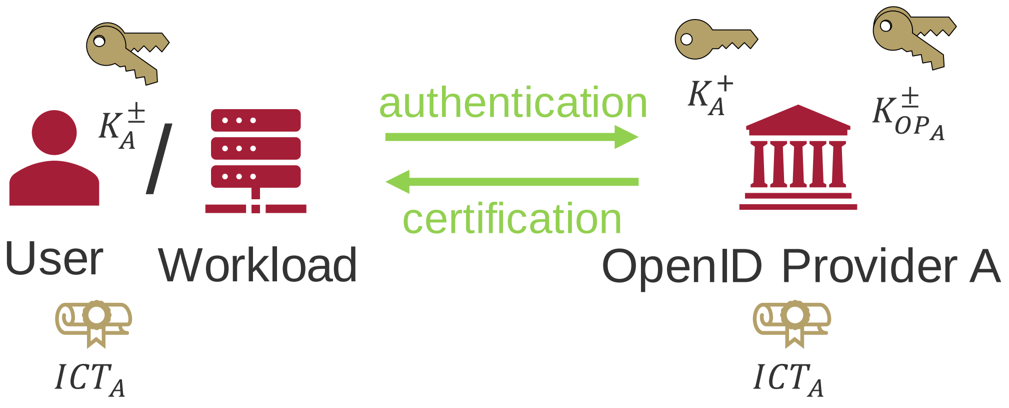
```
{
  "alg":"RS256",
  "kid":"2C8ECC453BE4B0F5E4F58D9653E1E259",
  "typ":"ict+jwt"
}
```

- Payload:

```
{
  "iss": "https://issuer.example.com",
  "aud": "https://workload.example.org",
  "exp": 1361398824,
  "cnf":{
    "jwk":{
      "kty": "EC",
      "use": "sig",
      "crv": "P-256",
      "x": "18wHLeIgW9wVN6VD1Txgpqy2L…8njVAibvhM",
      "y": "-V4dS4UaLMgP_4fY4j8ir7cgc…x535o7TkcSA"
    }
  }
}
```

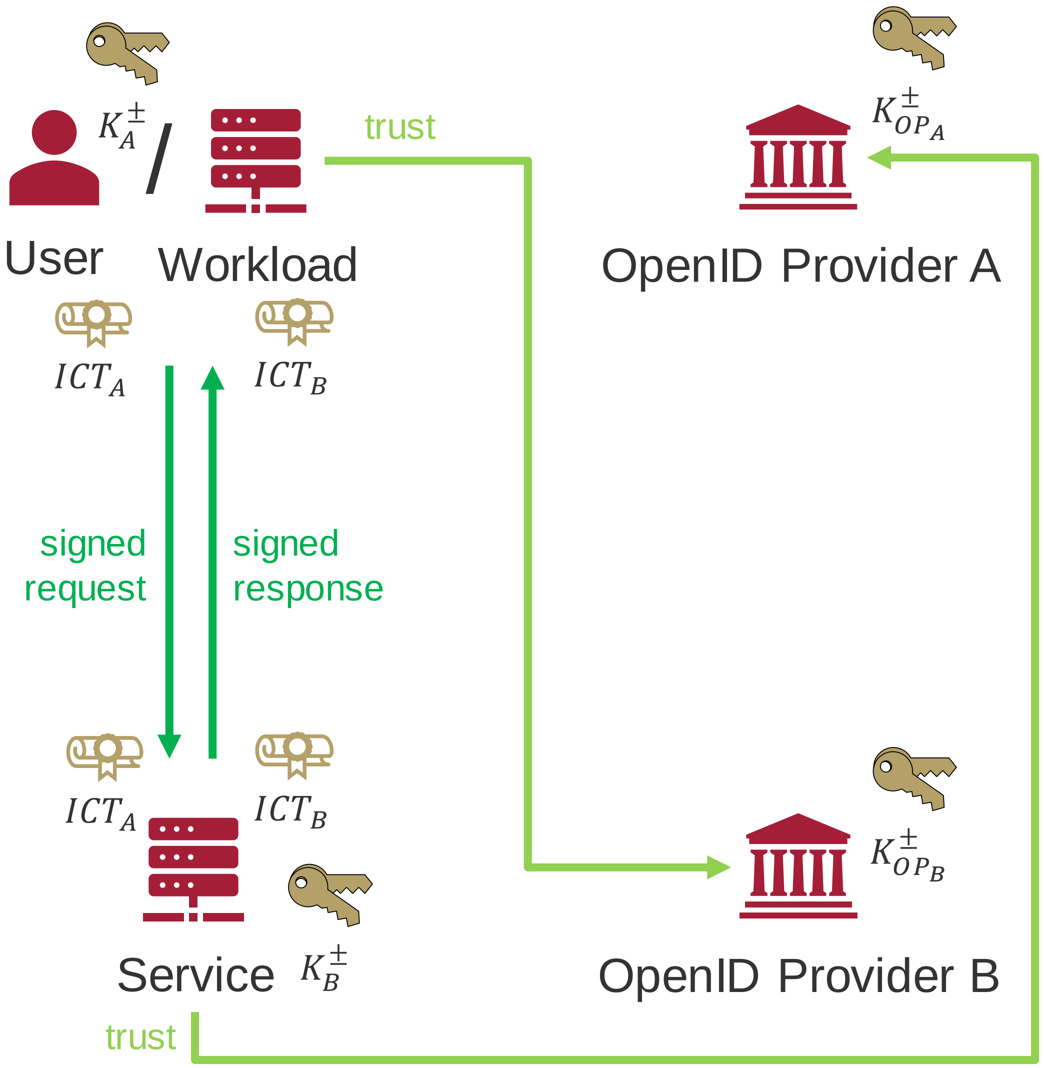
Certification

1. User's client / workload generates asymmetric key pair $K_A^{\pm}$
   - E.g., elliptic curve, RSA, …
2. User / workload authenticates themselves to the OpenID Provider
   - **User**: login with credentials / Passkey / …
   - **Workload**: remote attestation, API key, …
   - **Both**: public key + proof of possession
3. OpenID Provider verifies credentials and proof of possession, and issues an **Identity Certification Token ($\boldsymbol{ICT_A}$)**
4. OpenID Provider issues ICT_A to user / workload
   - Contains public key as confirmation (cnf) claim
   - Contains other claims about the user's / workload's identity



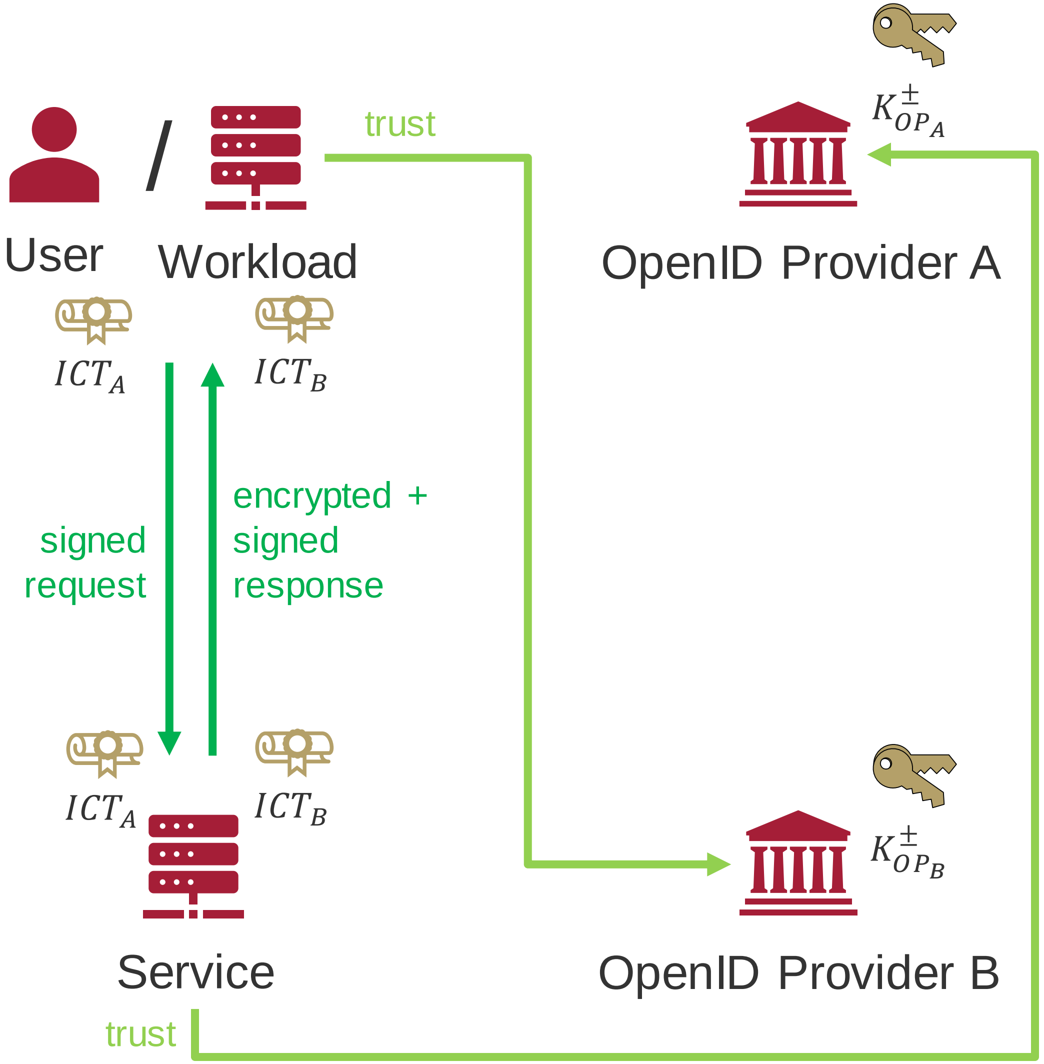
Authentication

1. User's client / workload adds ICT_A to the request and signs it with its private key K_A^-
   - ICT as sender-constraint in Authorization header (RFC 9449)
   - HTTP Message Signatures (RFC 9421)
2. Service validates ICT_A and signature
   - ICT_A issuer trusted?
   - ICT_A valid and user / workload authorized?
   - HTTP Message Signature valid?
3. Service (= workload) adds its own ICT_B to the response and signs it with its private key K_B^-
   - ICT_B in header
   - HTTP Message Signatures (RFC 9421)
4. User / workload verifies service's ICT_B and response signature
   - Requires trust in service's OpenID Provider



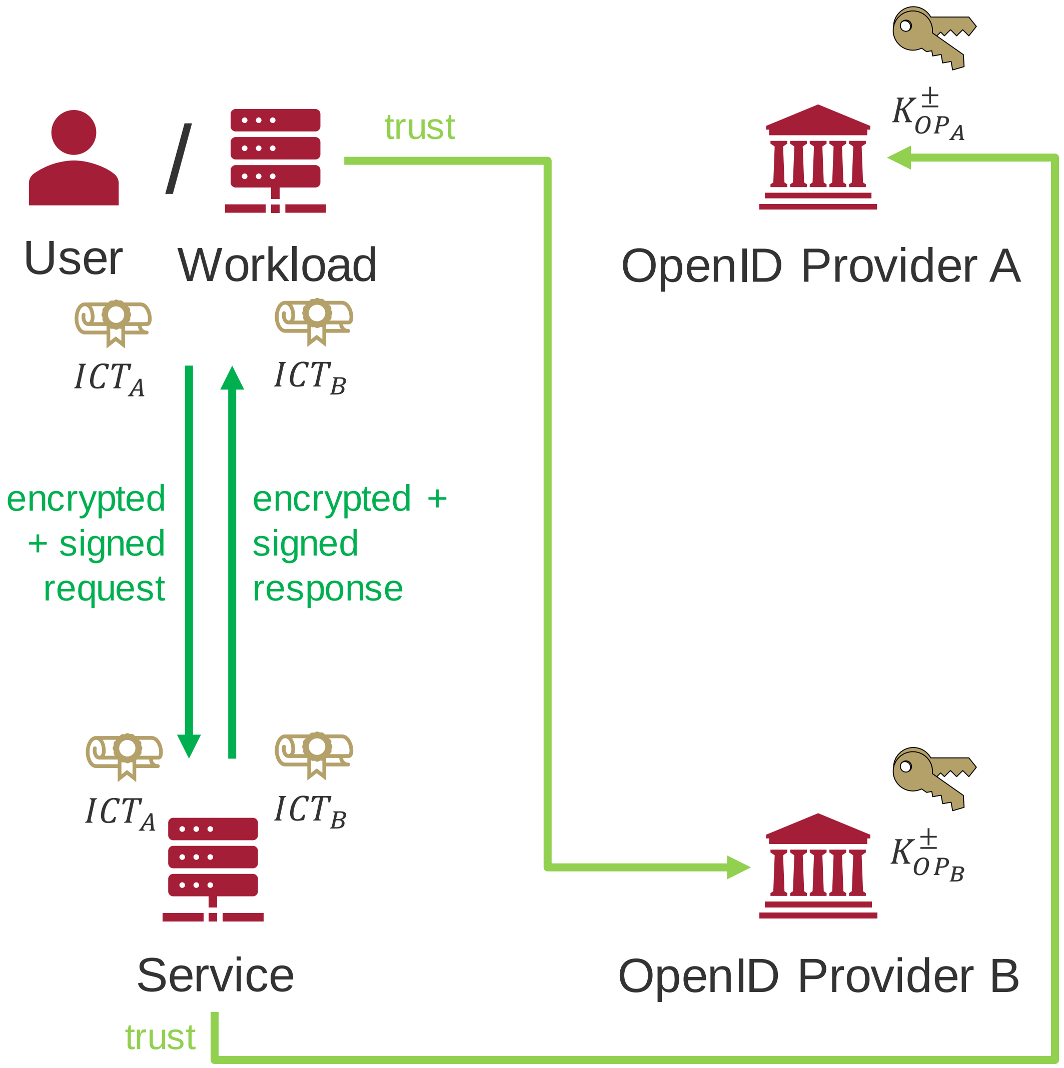
Encryption + Authentication

1. User's client / workload adds ICT_A **and Diffie-Hellman request parameters** to the request and signs it with its private key
   - Initializes a signed Diffie-Hellman key exchange
2. Service validates ICT_A and signature **and generates own Diffie-Hellman parameters**
   - Service can already compute shared secret
3. Service (= workload) adds its own ICT_B **and Diffie-Hellman parameters** to the response, **encrypts the payload with the shared secret** and signs it with its private key
4. User / workload verifies service's ICT_B and response signature**, computes shared secret and decrypts payload**



Advanced Encryption

- Exchanged shared secrets can be reused when creating a session
  - Only one initial key exchange required
  - Allows encrypted requests
- Keys can be rotated
  - Timed, e.g., every 10 minutes
  - In each request/response
    - Implements a Diffie-Hellman ratchet, see Signal
- Works stateless with session tokens
  - Session token is a JWT which contains the current state
  - Session token is symmetrically encrypted, MAC-ed, and issued by the service
  - Prevents synchronization errors in parallel requests



Further Steps

- We call the underlaying technology **Open Identity Certification with OpenID Connect (OIDC²)**
  - Peer-reviewed paper available on IEEE OJCOMS: https://doi.org/10.1109/OJCOMS.2024.3376193

- Demo available on GitHub: https://github.com/JonasPrimbs/oidc2-demo
  - Also contains demo for email with Google Mail, instant messaging with Matrix (soon), and video conferencing with WebRTC (soon)

- Questions, suggestions, cooperation requests?
  - Email to jonas.primbs@uni-tuebingen.de
  - LinkedIn: https://www.linkedin.com/in/jonasprimbs/
  - X: https://x.com/JonasPrimbs