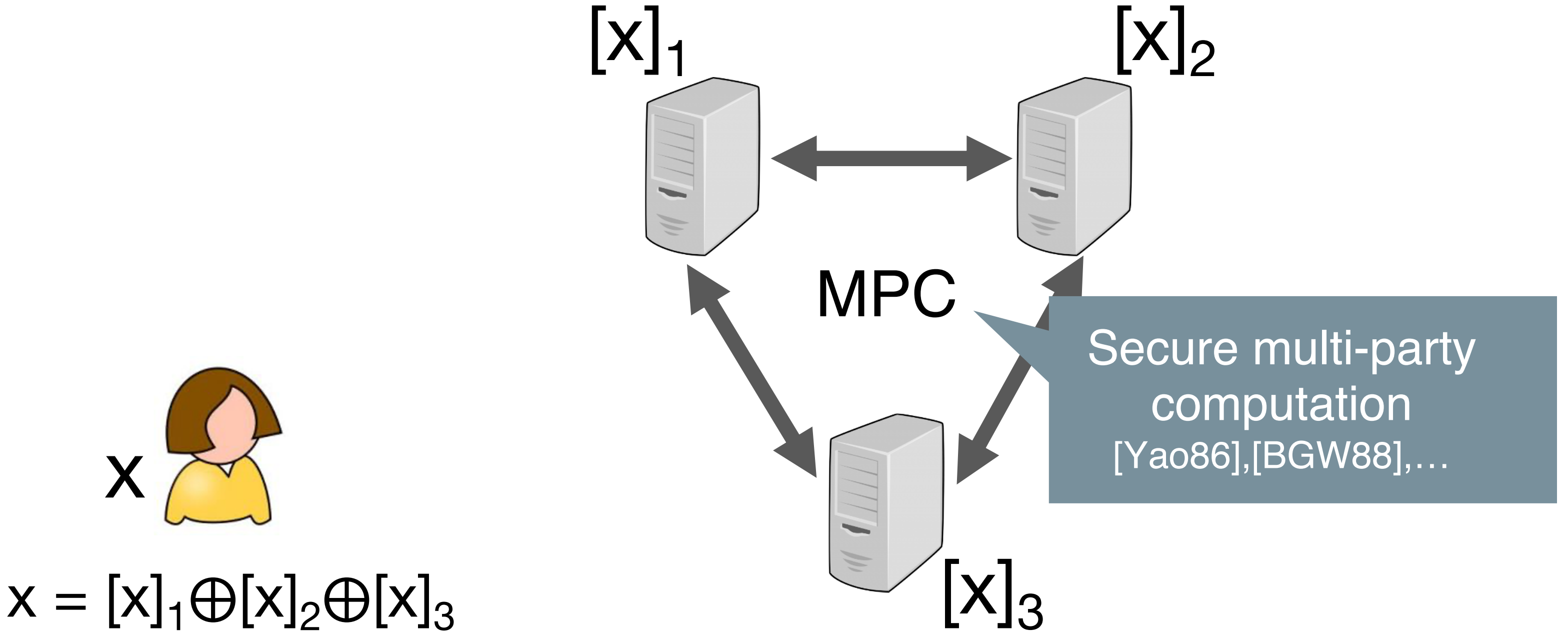
MPCAuth: Multi-factor Authentication for Distributed-trust Systems

Sijun Tan Weikeng Chen Ryan Deng Raluca Popa

UC Berkeley

Appeared at IEEE S&P 2023

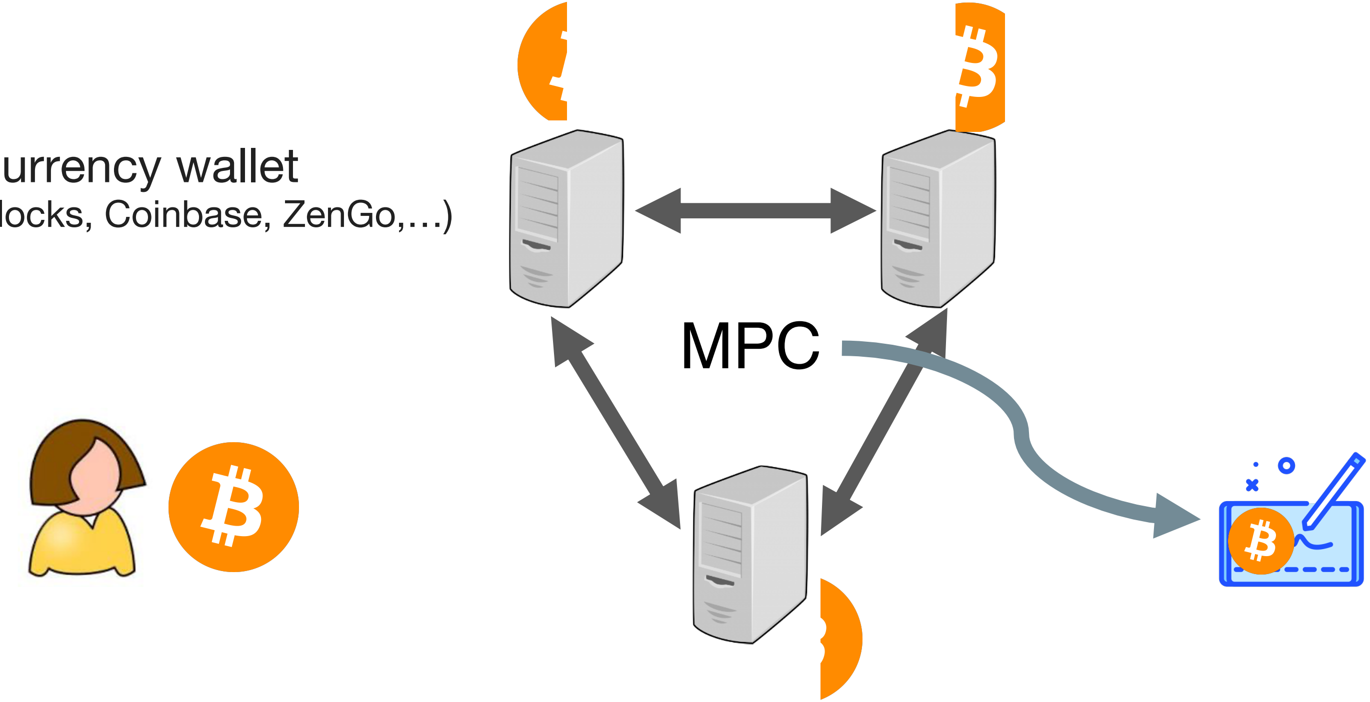
Overview of distributed-trust systems



The attacker needs to compromise all servers to recover the client's secrets.

Applications of distributed-trust systems

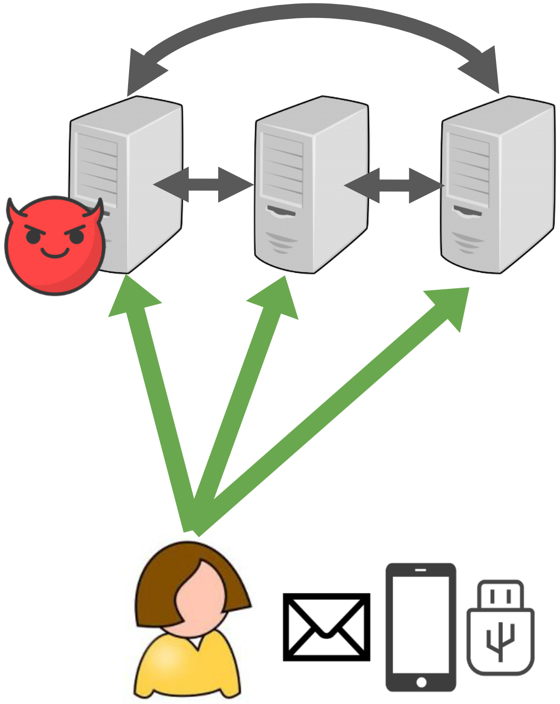
Cryptocurrency wallet
(e.g. Fireblocks, Coinbase, ZenGo,…)



Lots of other applications: Collaborative ML (e.g. Meta, Ant group), Secret key recovery (e.g. Signal) .

How to authenticate to distributed-trust systems?

Strawman 1: Authenticate to one master server.

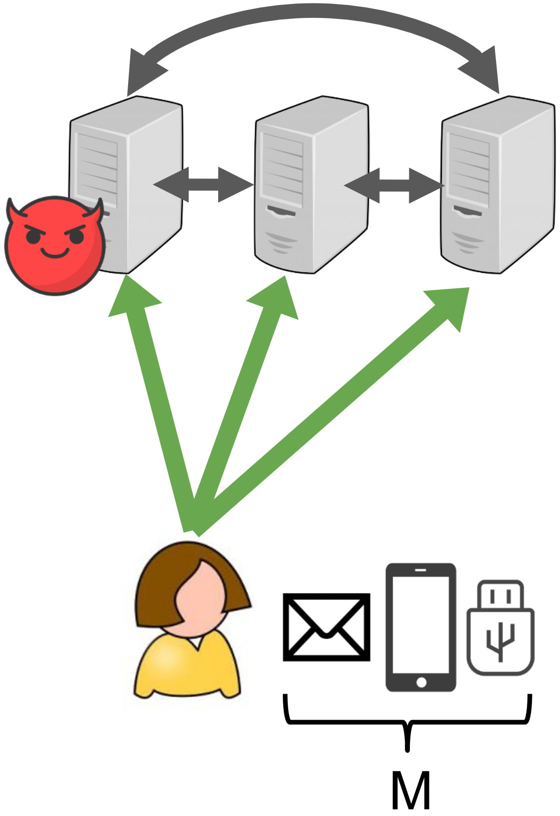


Other servers trust the master server.

A malicious attacker can compromise this one server to recover the secrets.

The client needs to authenticate to all servers to ensure security.

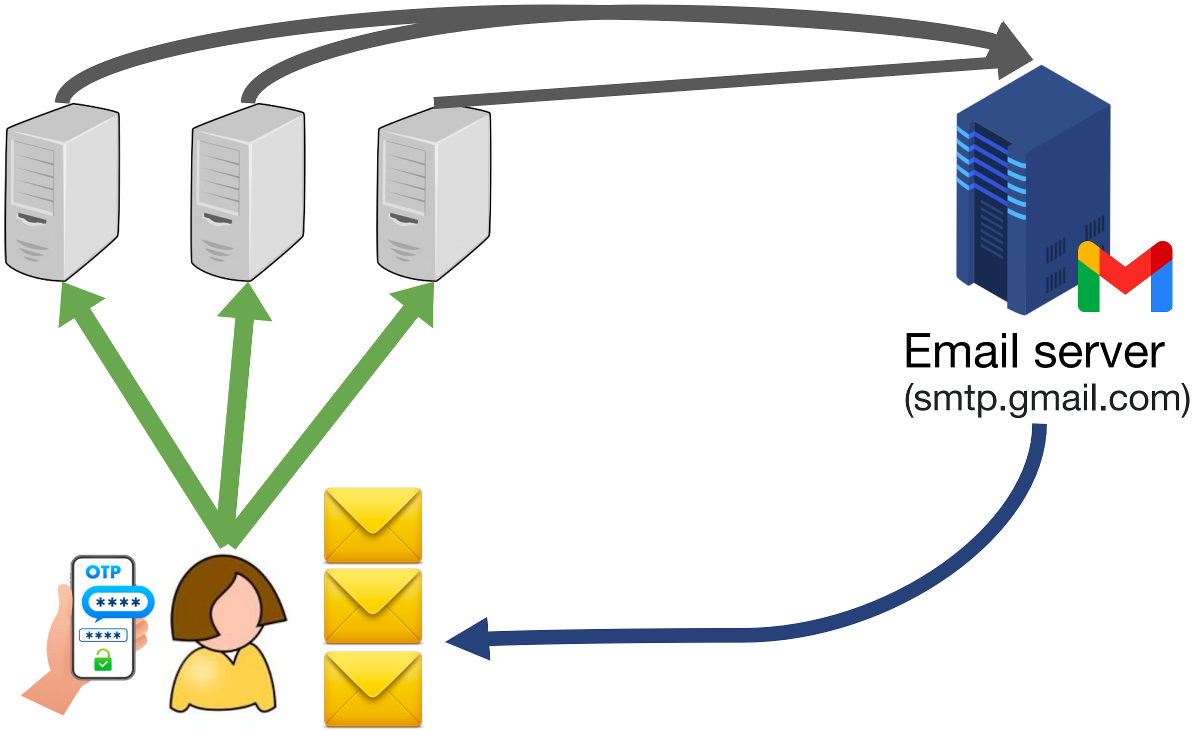
Strawman 2: Authenticate to each of N servers



Avoids a central point of attack.

Problem: The client needs to authenticate to N servers NxM times, one for each of the M factors.

Problem: Burdensome user experience



The client needs to receive N emails and enter passcodes N times!

Our system: MPCAuth

An authentication system for distributed-trust applications in which the user authenticates only ***once***.

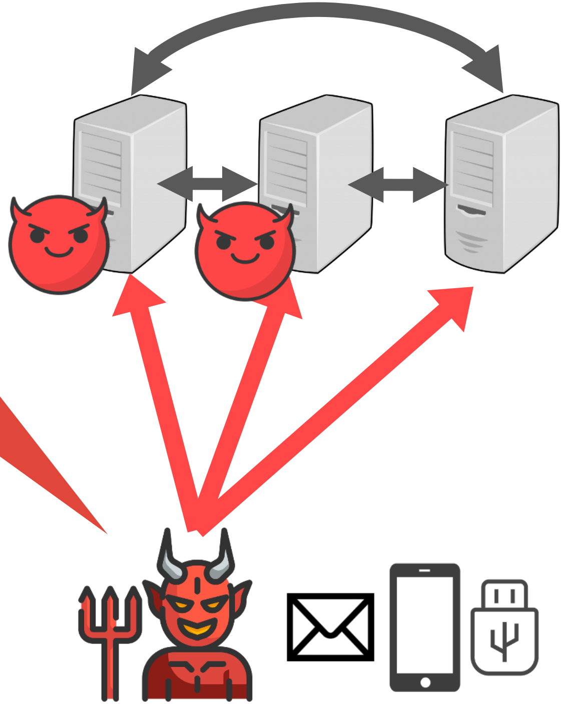
Type	**Factors**
Possession	Email, SMS, U2F
Knowledge	Passcode, Pin, Security Questions
Inherence	Biometrics

In addition, hides the user's authentication profiles. (e.g. email username, phone number, passwords, biometric features)

Threat model

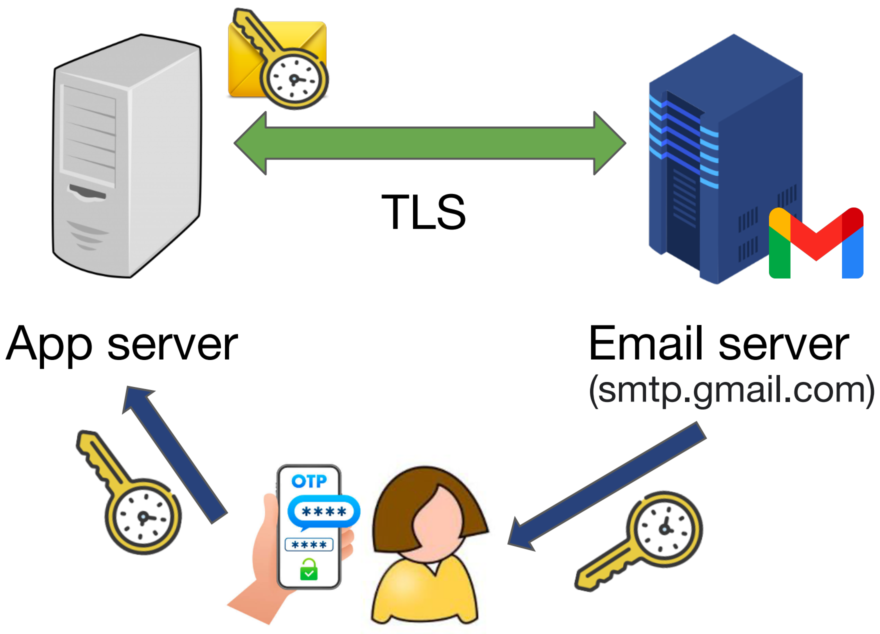
- An attacker can corrupt up to N-1 out of N servers.
- The attacker tries to impersonate a client.

The attacker cannot successfully authenticate as an honest user, if at least one server and one authentication factor is not compromised.

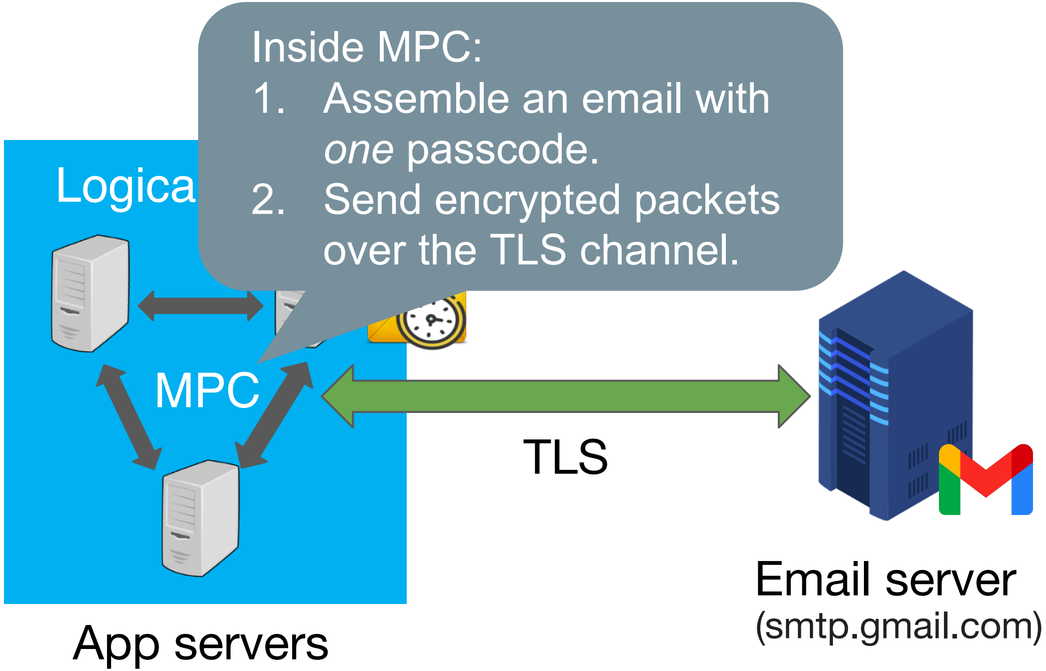


MPCAuth's Email Authentication

Traditional email authentication

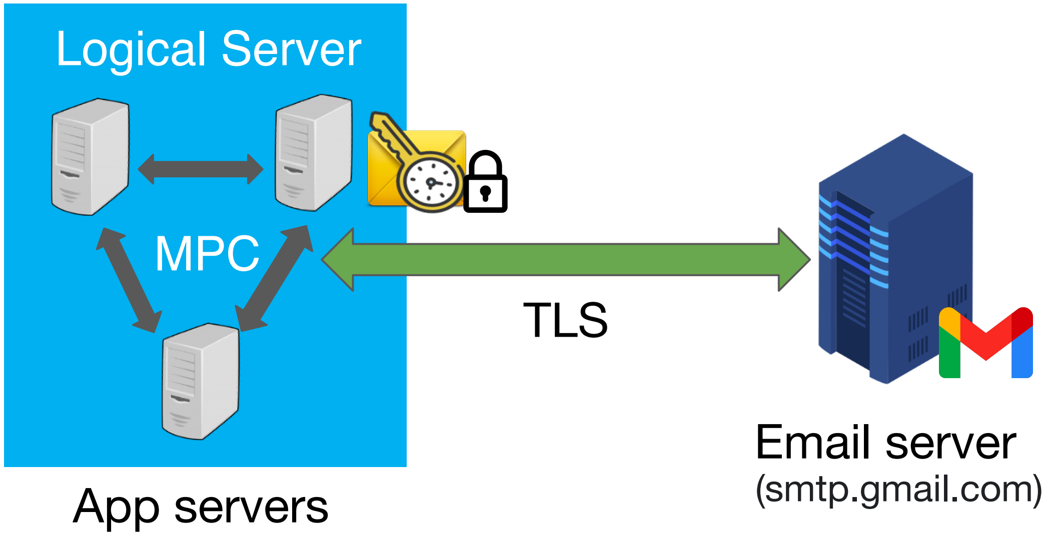


Email authentication for distributed-trust systems



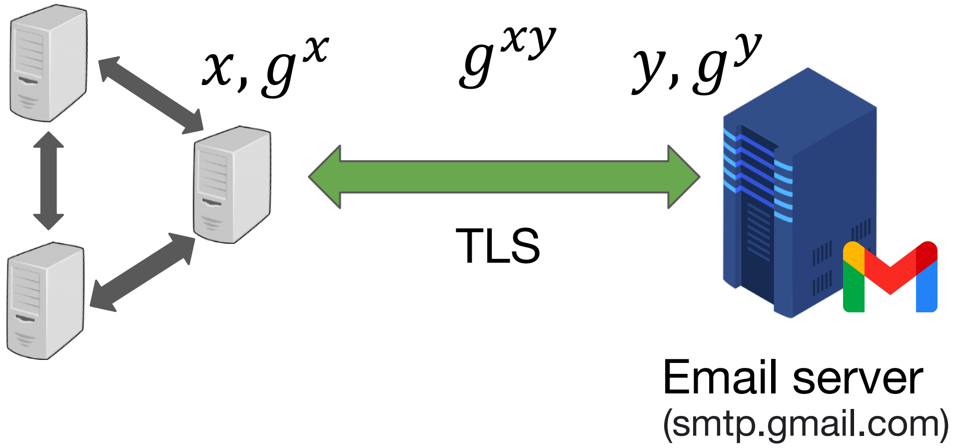
The N servers jointly act as one logical server to interact with the email server.

Email authentication for distributed-trust systems



The N servers jointly act as one logical server to interact with the email server.

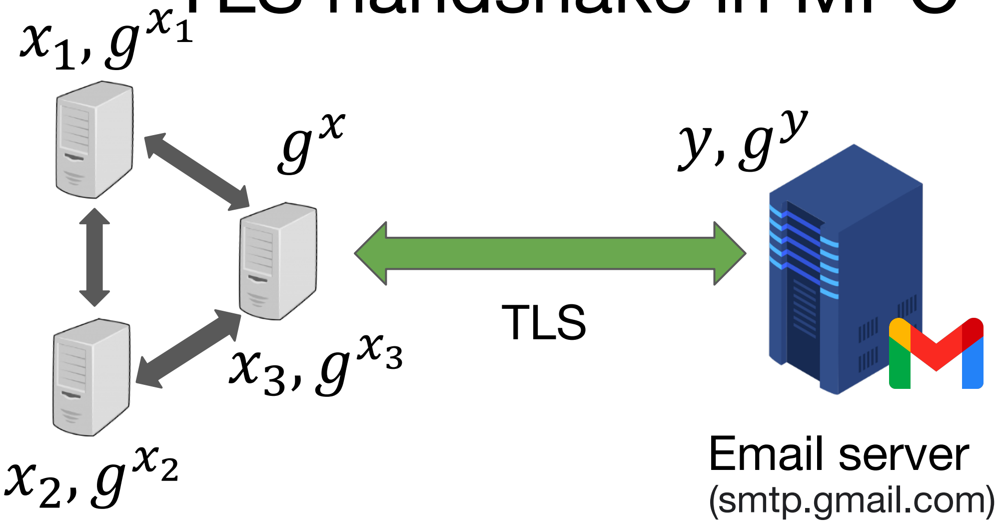
TLS-in-MPC



TLS Handshake: Jointly perform Diffie-Hellman key exchange.

Data transmission: Jointly run an authenticated encryption scheme to encrypt messages and transmit them over the network.

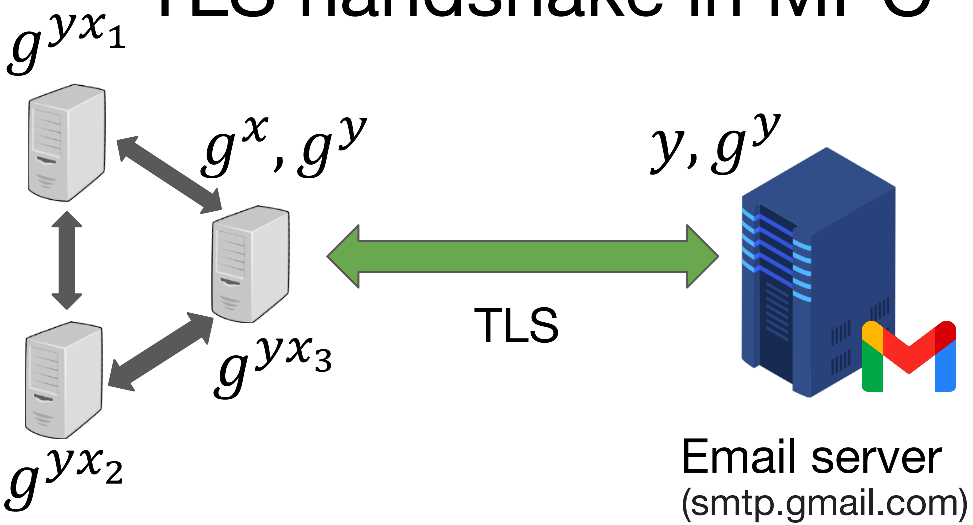
TLS handshake in MPC



$$g^x := g^{x_1+x_2+x_3}$$

Each party locally samples x_i, computes g^{x_i} and sends it to the relaying party.

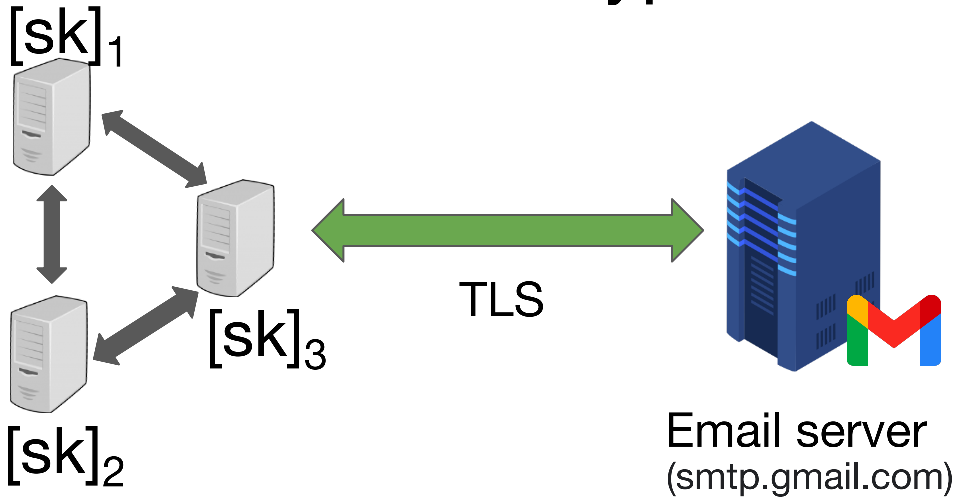
TLS handshake in MPC



[sk] :=KDF([g^{xy}])

Each party locally computes g^{yx_i} , which forms the secret share $[g^{xy}]$

Authenticated encryption in MPC

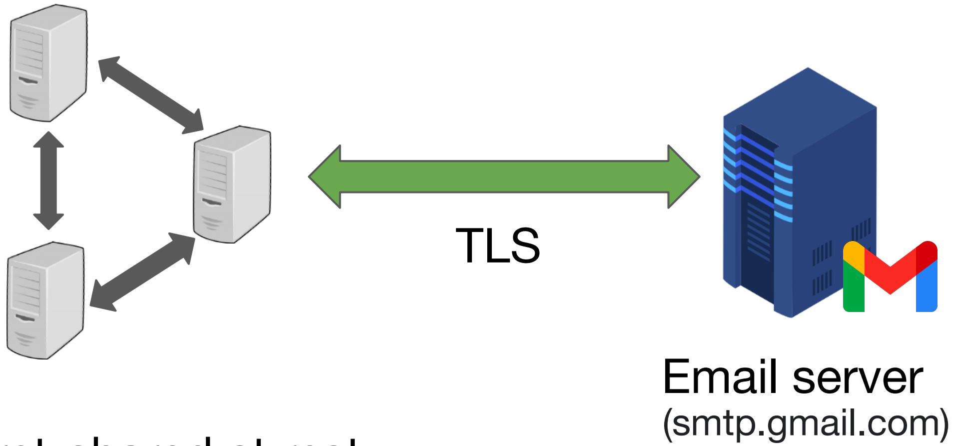


[sk] :=KDF([g^{xy}])

enc,mac:=AES-GCM([sk], [msg])

Compute authenticated encryption in MPC with secret-shared sk and message.

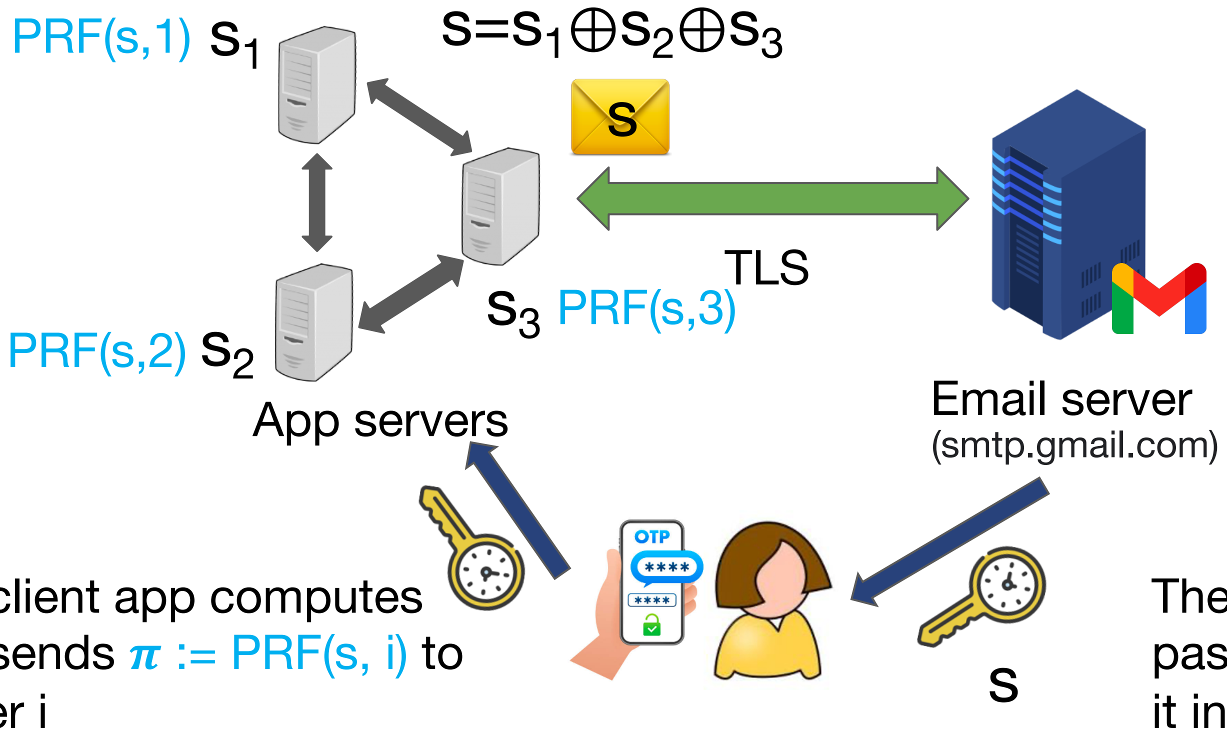
Implication of TLS-in-MPC



- Data is secret-shared at rest.
- During transmission, data is encrypted in MPC with a secret-shared encryption key.
- None of the server sees any plaintext data during the whole process.

The protocol itself is extendable to use cases beyond authentication.

MPCAuth's email authentication protocol

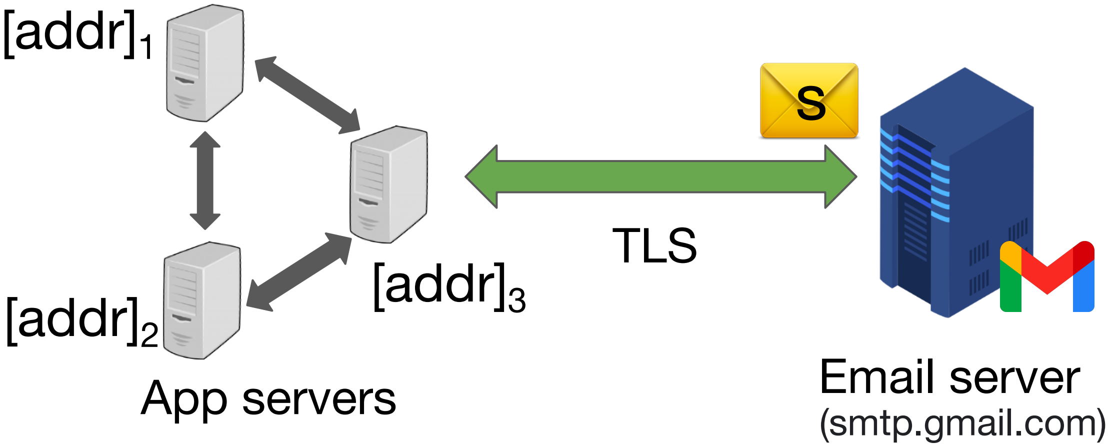


The client app computes and sends π := PRF(s, i) to server i

The client receives passcode s and enters it into the app.

The passcode s is hidden from all servers.

MPCAuth's email authentication protocol



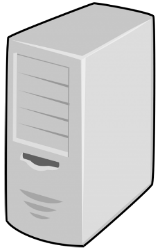
- The client only enters the passcode *once* on the client app.
- The client's email username is hidden from all servers.

MPCAuth's U2F Authentication

Traditional U2F authentication

sk

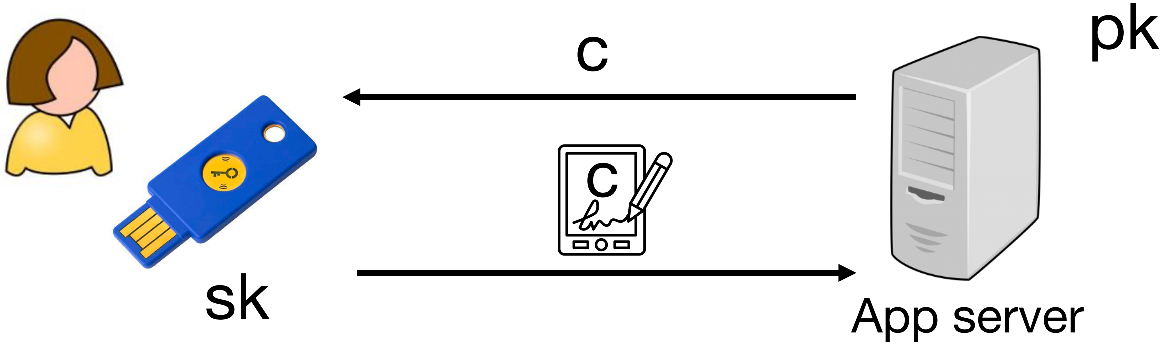
pk



App server

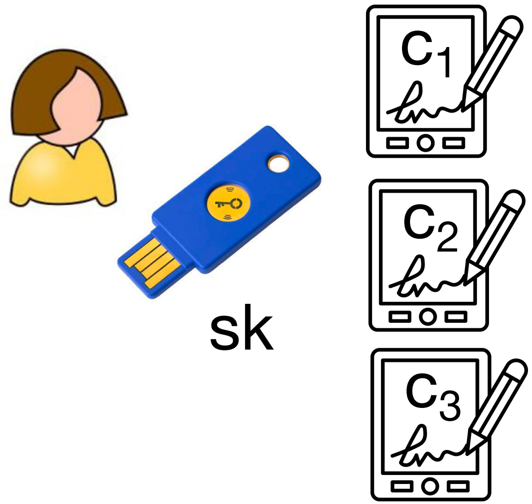
During registration, U2F generates a key pair and stores the public key to the server.

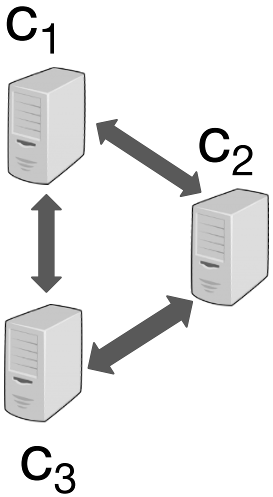
Traditional U2F authentication



During authentication, U2F produces a signature over the app server's challenge. The app server verifiers the signature.

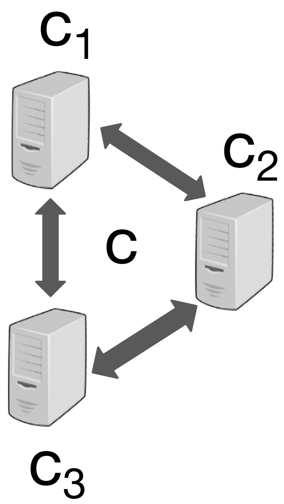
U2F authentication under distributed trust





Naively, the user needs to tap the U2F button N times.

Strawman 2: Negotiate a joint challenge



Idea: Negotiate a joint challenge, verify individually.

Does not prevent against replay attacks.

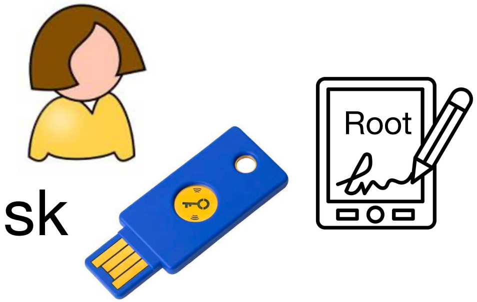
Designing an authentication protocol

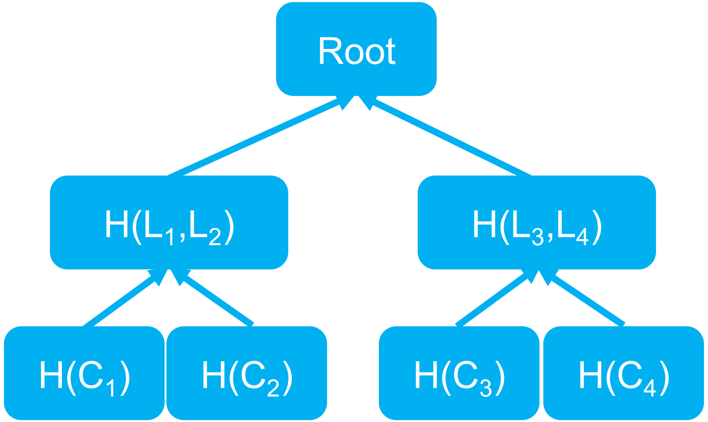
Main Takeaway:

1) The U2F signs a single joint challenge.
2) Each server needs to verify its local challenge.
3) Each server's local challenge needs to be kept secret.

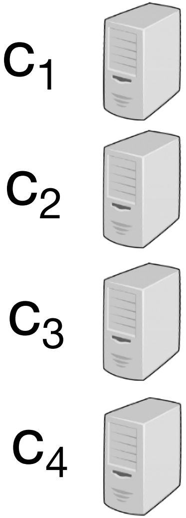
MPC works but there is an even simpler solution.

MPCAuth's U2F authentication protocol





$C_i := \text{Commit}(c_i, r_i)$



The client app:

1) Commits to each local challenge c_i with r_i.
2) Builds a Merkle tree over C_i.
3) Produce a signature over the root hash.

MPCAuth's U2F authentication protocol

sk

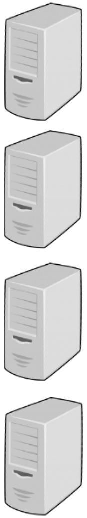
(sig, root, π_i, r_i)

(sig, root, π_1, r_1) c_1

(sig, root, π_2, r_2) c_2

(sig, root, π_3, r_3) c_3

(sig, root, π_4, r_4) c_4



The client app sends to server *i*:

1) The Merkle root hash.
2) The Merkle opening proof for leave *i* π_i.
3) The signature over the root hash.
4) The randomness r_i.

MPCAuth's U2F authentication protocol

sk

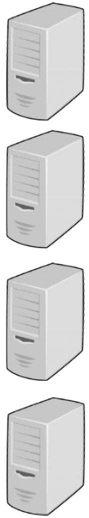
(sig, root, $\pi_{i,}$ r_i)

(sig, root, $\pi_{1,}$ r_1) c_1

(sig, root, $\pi_{2,}$ r_2) c_2

(sig, root, $\pi_{3,}$ r_3) c_3

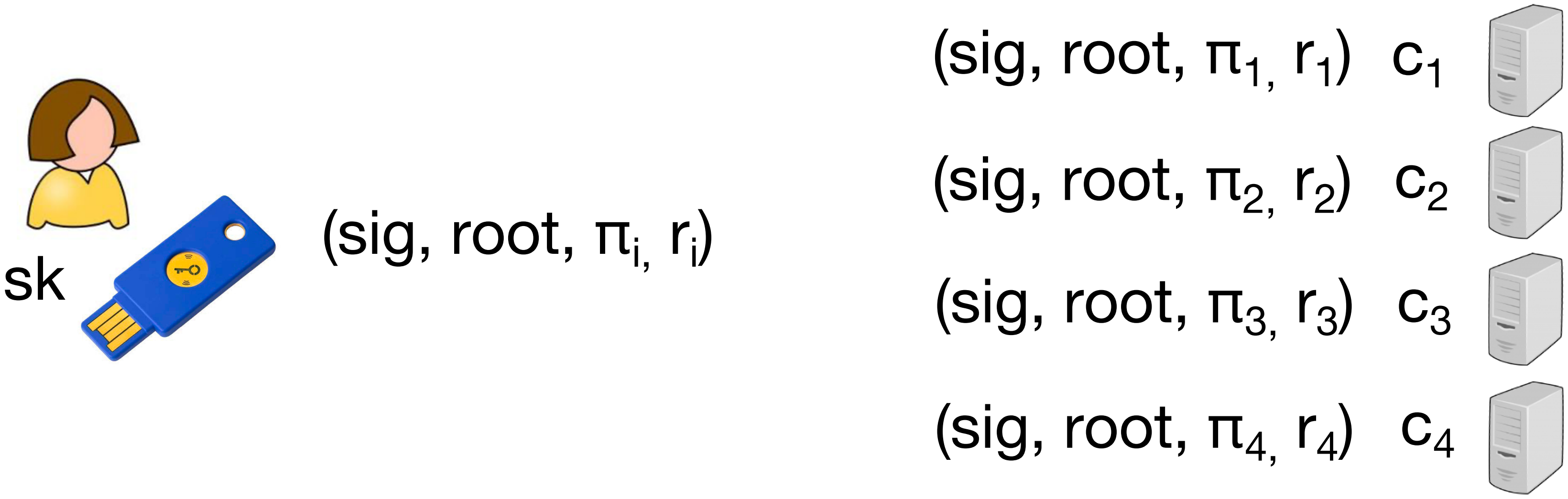
(sig, root, $\pi_{4,}$ r_4) c_4



Each server *i* checks:

1) The signature is over the root hash.
2) C_i is included in the Merkle tree.
3) C_i is a commitment of c_i.

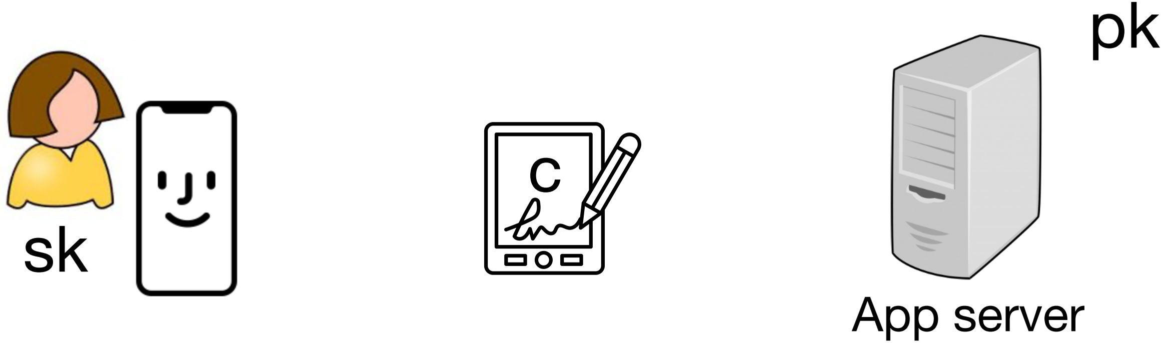
MPCAuth's U2F authentication protocol



- The user only signs one signature over the joint challenge.
- Each server receives a different response.
- Each server verifies both the joint challenge, as well as their local challenge (by checking commitment opening)

MPCAuth's Biometrics Authentication

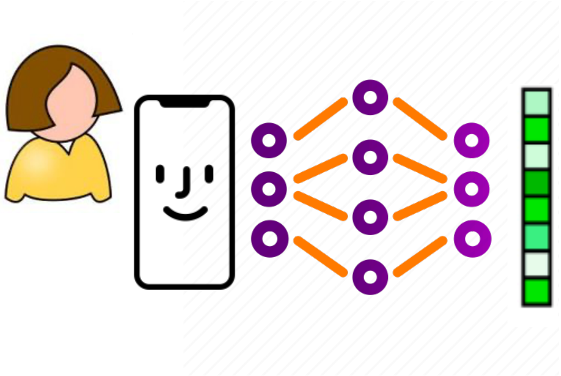
Client-side biometric authentication

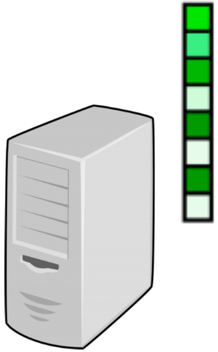


- Alice scans her biometrics to unlock her secret key.
- Alice signs a signature over the verifier's challenge.
- The server verifies that the signature is correct.

MPCAuth's U2F authentication protocol works.

Server-side biometric authentication



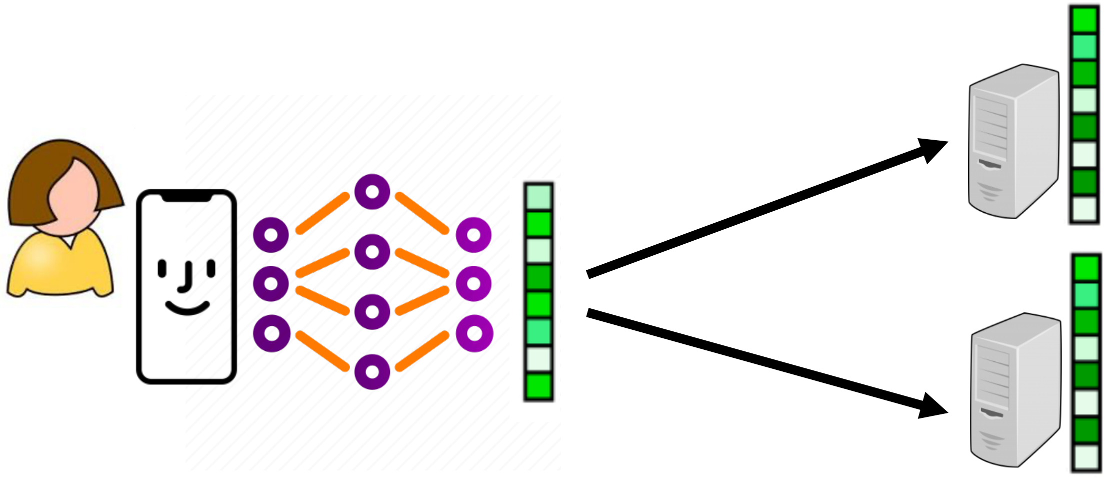


App server

- Alice scans her biometrics, the client device locally process it, and sends the feature vector to the server.
- The server verifies that the feature vector is closed to the registered one.

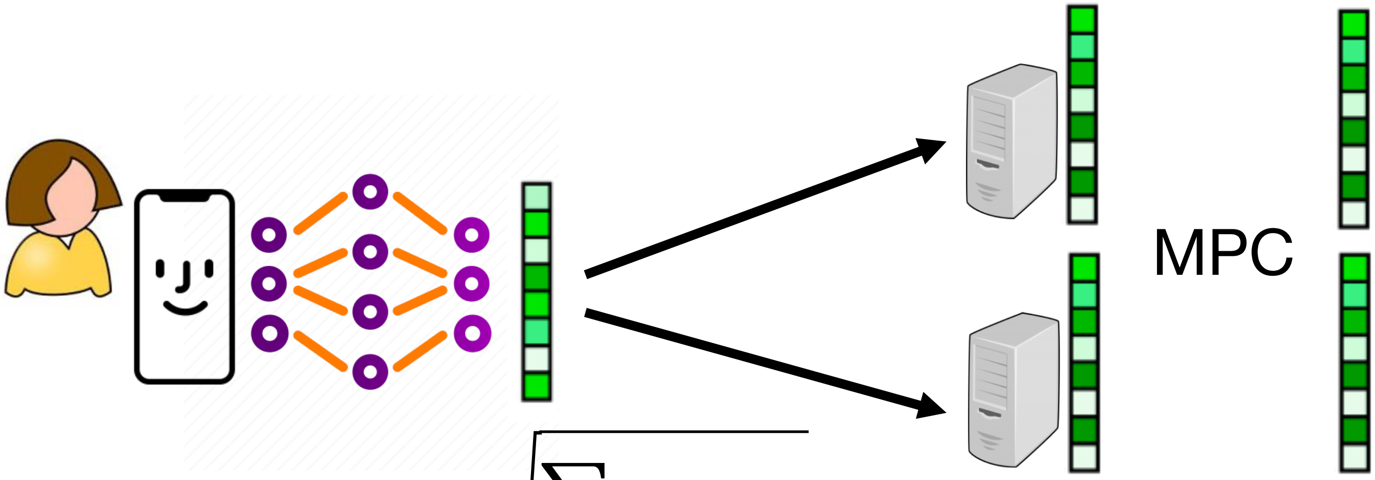
Poses huge privacy risks as the app server needs to store the feature vector in plaintext!

MPCAuth's biometric authentication



During registration, Alice secret-shares the feature vector v_1 to the servers.

MPCAuth's biometric authentication



$$L_2(\vec{x},\vec{y}) = \sqrt{\sum (x_i - y_i)^2}$$

During authentication, Alice produces a feature vector v_2. The server performs an L2 distance check between v_1 and v_2.

Implementation & Evaluation

Implemented the system using MP-SPDZ, EMP-AGMPC, and WolfSSL.

Evaluated the system on 2-5 AWS c5n.2xlarge 3.0GHz 8 core CPU.

Server-to-server bandwidth: 2Gbit/s
Client-to-server bandwidth: 100Mbit/s.

Without established TLS

3PC	**Offline**	**Online**	**Total**
Email Auth	10.9s	1.3s	12.2s

With established TLS

3PC	**Offline**	**Online**	**Total**
Email Auth	2.9s	0.4s	3.3s

Works with existing email provider (Gmail) with no timeout.

Evaluation of TLS-in-MPC

Offline latency of TLS-in-MPC

	N=2	N=3	N=4	N=5
Offline	7.4s	8.1s	11.1s	14.8s

Online latency of TLS-in-MPC

	N=2	N=3	N=4	N=5
Offline	0.7s	0.9s	1.1s	1.4s

Given the low online latency, TLS-in-MPC can be scaled to a larger number of parties with no TLS timeout (15s).

Summary of MPCAuth

An authentication system for distributed trust applications.

- Enables a client to authenticate independently to N servers by doing the work of only *one* authentication.
- Design secure, practical, and profile-hiding protocols for multiple authentication factors.

Email: sijuntan@berkeley.edu

Paper: https://eprint.iacr.org/2021/342.pdf

Thank you!