AuCPace: Efficient Verifier-Based PAKE protocol tailored for the IIoT

Björn Haase, Benoît Labrique
Endress + Hauser Conducta GmbH & Co. KG.

Highly relevant topic in today's HMI authentication systems

Highly relevant topic in today's HMI authentication systems

Passwords ...

Highly relevant topic in today's HMI authentication systems

Passwords …

This Talk:

… In case that we are forced to accept that we can't avoid them:
How could we at least make their use as secure as possible …

even when facing tight resource constraints.

Highly relevant topic in today’s HMI authentication systems

Passwords …

This Talk:

… In case that we are forced to accept that we can’t avoid them:
How could we at least make their use as secure as possible …

even when facing tight resource constraints.

System-level approach

Examples for process industry installations and field devices

Examples for process industry installations and field devices

Many installations: critical infrastructure

Security should be mandatorily considered !

Security for industrial control equipment

- Security: A rather new topic for industrial control
- First step for security: focus on machine-to-machine interfaces and protocols.
- HMI interfaces often considered in a second step only.

- E+H: Remote HMI service access mostly provides an even larger attack vector!

- Most widespread authentication mechanism for HMI interfaces 2019: Passwords

Requirements derived when planning the E+H BlueConnect App Architecture

- In very important settings no PKI at the customer installation!
  => HMI security solution shall not rely on PKI.

- Network access to central authentication servers is not always available
  (Subnetworks “air-gapped” for security reasons / Devices integrated to legacy fieldbuses)
  => Support required for “offline” authentication with local storage of credentials

- Some devices have extremely tight resource constraints.
  (Intrinsically safe explosion protection by power and energy limits, See [HL17])

- Devices might become physically accessible for the adversary.

- We shall prepare the architecture for two-factor authentication, but need to accept that our customers will often stick to the concept of “passwords” for HMI authentication only.

Result of our assessment

We are forced to work with passwords?

Lets then do our **very** best to protect our customer's installations!

We need a combination of two elements:

- Verifier-based password authenticated key exchange (V-PAKE)
- State-of-the-art memory-hard password hashes

Astonishingly there is no established industry standard solution!

Our protocol proposals

- "Augmented Composable Password-Authenticated Connection Establishment"

AuCPace

- "Composable Password-Authenticated Connection Establishment"

CPace

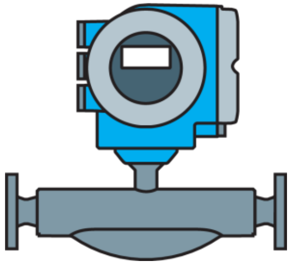
- Constructions were designed for allowing freely usable implementations avoiding patents in order to make it suitable for more widespread use and, possibly, standardization.
- Motivation for this paper: Security proof will be pre-condition for more widespread use.
- This talk also considers preliminary results from the second review round carried out in the context of the CFRG PAKE selection process.

Outline of this talk

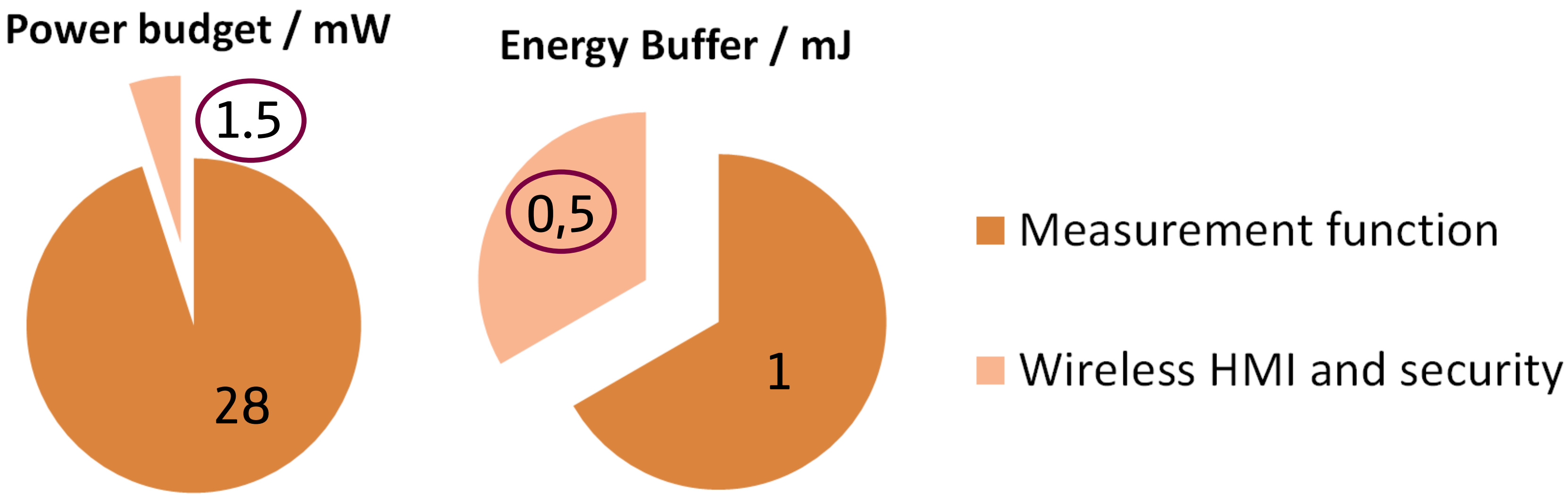
- AuCPace and CPace protocols and their security analysis
- Comparison with other V-PAKE nominations from current CFRG selection process
- Implementation strategy and results on ARM Cortex-M4 and Cortex-M0
- Summary

CHES2017: Typical budget constraints for Ex-ia field devices

- Ignition by hot surfaces → Limit peak supplied electrical power
- Ignition by Sparks → Limit size of energy buffers (e.g. capacitors)



Add-on feature "HMI interface and security" will be granted only a small fraction of the available power / transient buffer budget!



Optimization strategy

- Protocol level
  - Allow for fast curves: X25519 Diffie-Hellman
  - “x-coordinate-only” solution avoids need for point compression
  - Secure quadratic twist of Curve25519: AuCPace simplified point verification
  - No hash over full protocol transcripts required
  - Refer the password hash to the powerful client
- Curve25519 group element operations
  - Optimization of Elligator2 in comparison to [HL17] by using method from [BDL+11]
- Fe25519 field operations
  - Optimized assembly-level code using register-allocating code-generator tool

The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

AuCPace Augmentation layer

$x \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$		
$X = B^{x\, c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
$W, \text{salt} = \text{lookup}W(\text{user})$		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_\sigma(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_\$ \{0,1\}^{k2}$,		abort if X invalid
else $PRS = W^{x\, c_{\mathcal{J}}}$		$PRS = X^{w\, c_{\mathcal{J}}}$

CPace substep

$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$		$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$		$y_b \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\, c_{\mathcal{J}}}$		$Y_b = G^{y_b\, c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a\, c_{\mathcal{J}}}$		$K = Y_a^{y_b\, c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \Vert K)$		$sk_1 = \mathsf{H}_2(ssid \Vert K)$

Explicit mutual authentication

$T_a = \mathsf{H}_3(ssid \Vert sk_1)$		$T_a = \mathsf{H}_3(ssid \Vert sk_1)$
$T_b = \mathsf{H}_4(ssid \Vert sk_1)$		$T_b = \mathsf{H}_4(ssid \Vert sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid \Vert sk_1)$		$sk = \mathsf{H}_5(ssid \Vert sk_1)$

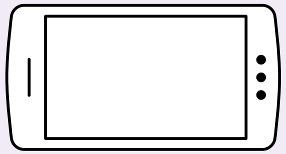
Endress+Hauser

The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

- Client side (e.g. tablet PC): Clear-text password ("pw") available

AuCPace Augmentation layer

$x \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$		
$X = B^{x \cdot c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
$W, \text{salt} = \text{lookup}W(\text{user})$		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_\sigma(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_\$ \{0,1\}^{k2}$,		abort if X invalid
else $PRS = W^{x \cdot c_{\mathcal{J}}}$		$PRS = X^{w \cdot c_{\mathcal{J}}}$

CPace substep

$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$		$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$		$y_b \leftarrow_\$ \{1 \dots m_{\mathcal{J}}\}$
$Y_a = G^{y_a \cdot c_{\mathcal{J}}}$		$Y_b = G^{y_b \cdot c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a \cdot c_{\mathcal{J}}}$		$K = Y_a^{y_b \cdot c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \Vert K)$		$sk_1 = \mathsf{H}_2(ssid \Vert K)$

Explicit mutual authentication

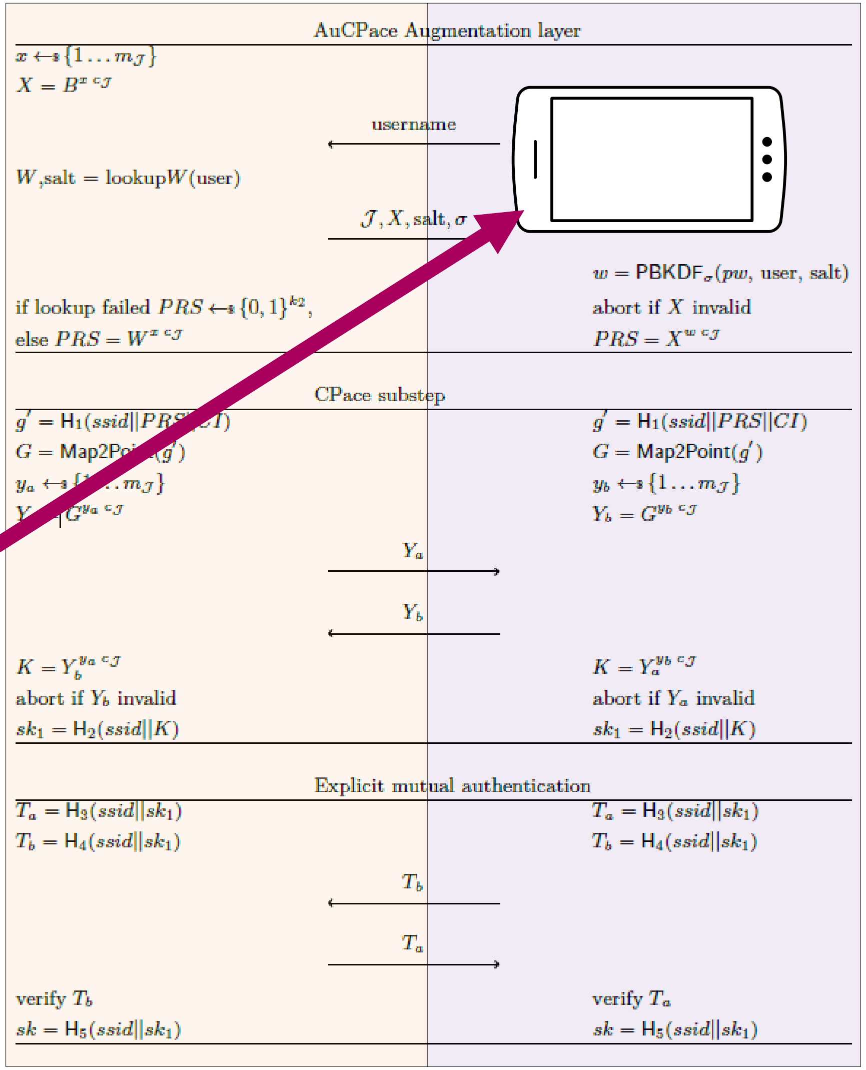
$T_a = \mathsf{H}_3(ssid \Vert sk_1)$		$T_a = \mathsf{H}_3(ssid \Vert sk_1)$
$T_b = \mathsf{H}_4(ssid \Vert sk_1)$		$T_b = \mathsf{H}_4(ssid \Vert sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid \Vert sk_1)$		$sk = \mathsf{H}_5(ssid \Vert sk_1)$

The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

- Client side (e.g. tablet PC):
  Clear-text password ("pw") available

Typically large memory,
powerful computation capabilities.
(scrypt/Argon2)



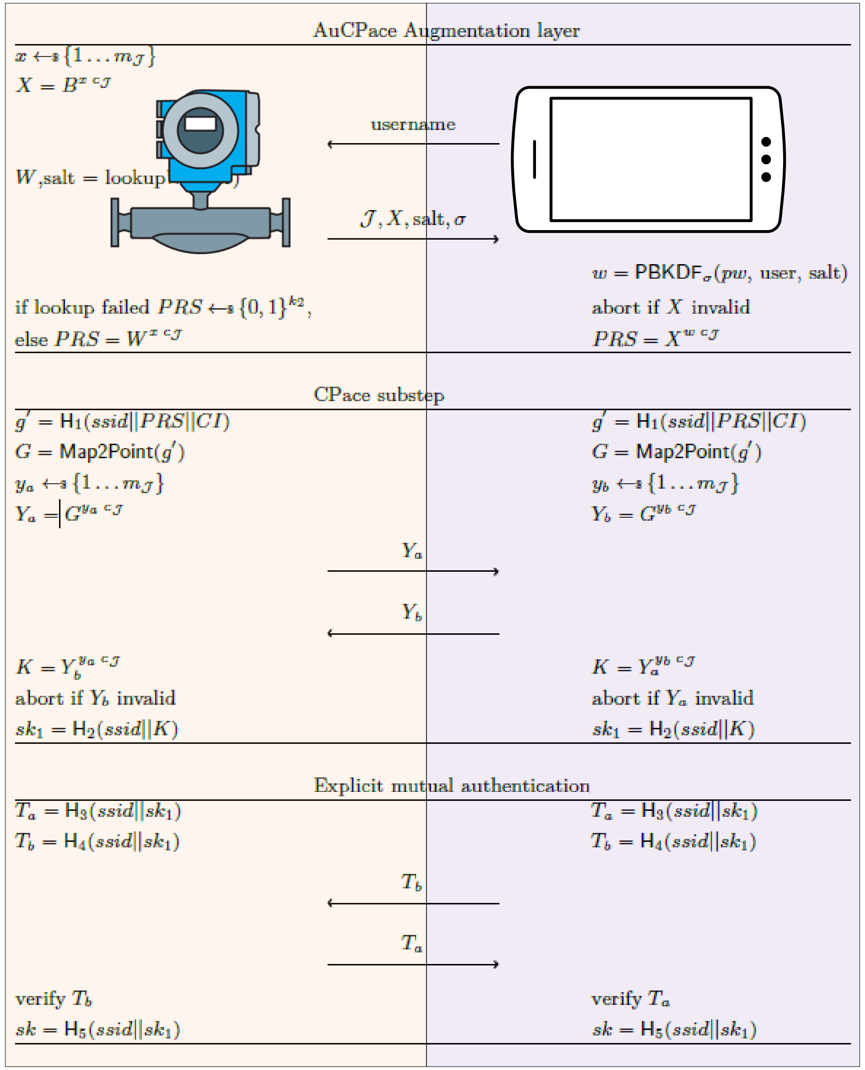
Endress+Hauser

The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

- Client side (e.g. tablet PC):
  Clear-text password (“pw”) available

- Server side (e.g. field device)
  Password verifier (“W”)

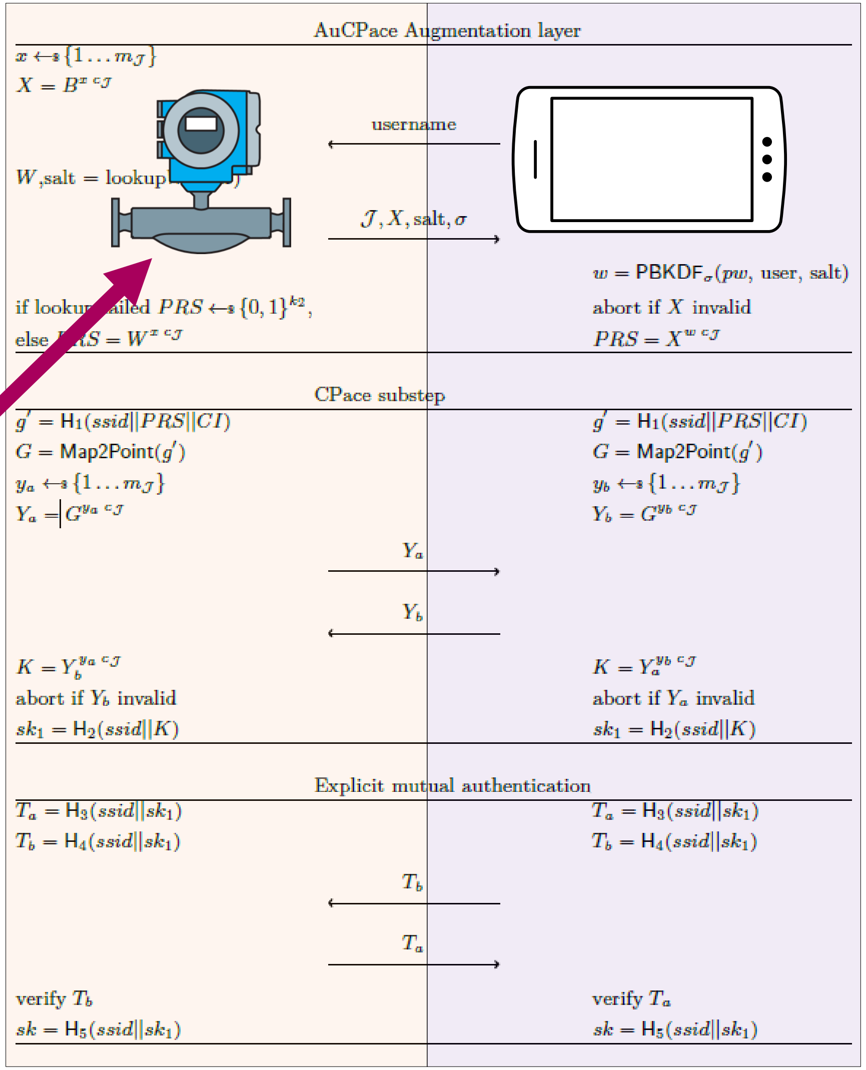


The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

- Client side (e.g. tablet PC): Clear-text password ("pw") available
- Server side (e.g. field device) Password verifier ("W")

Strongly constrained device

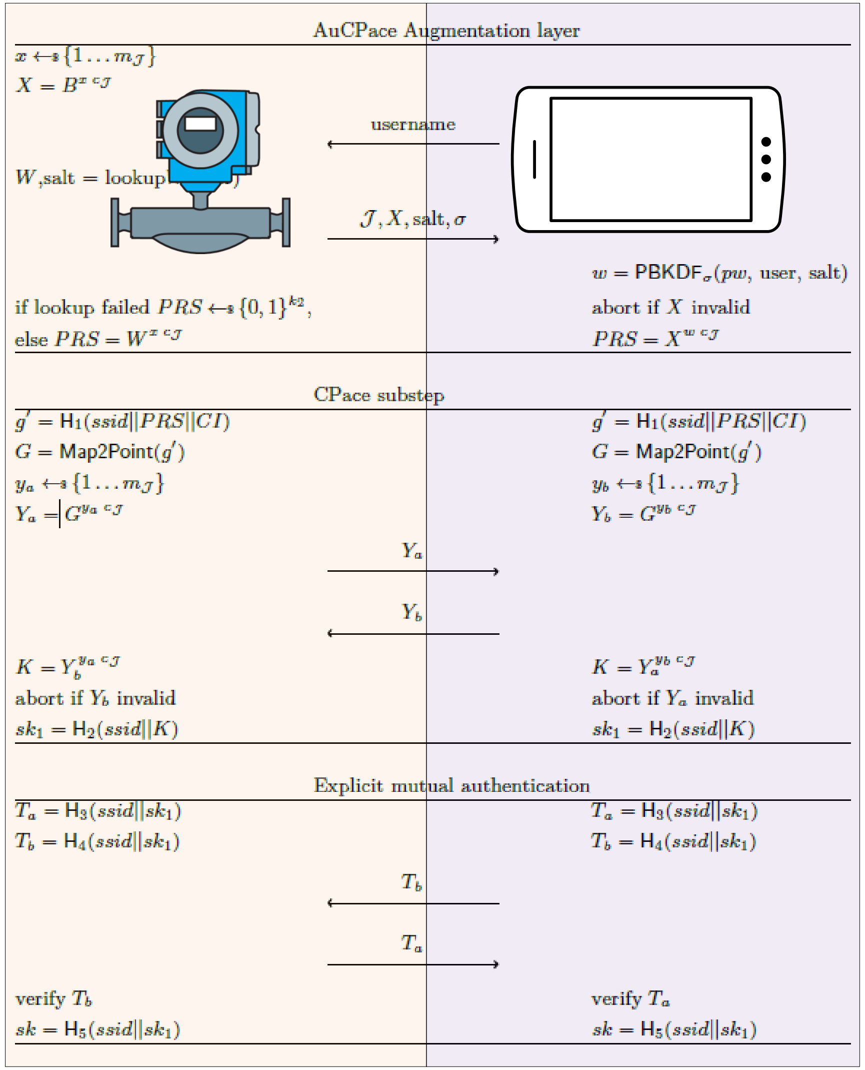


The modular AuCPace protocol construction

AuCPace is a two-party *verifier-based* Password-Authenticated Key Exchange (PAKE) protocol

- Client side (e.g. tablet PC):
  Clear-text password ("pw") available
- Server side (e.g. field device)
  Password verifier ("W")

V-PAKE: Knowledge of password verifier W does not allow for taking over the client role.



The modular AuCPace protocol construction

Three subcomponents within AuCPace

- AuCPace augmentation layer
- CPace balanced PAKE protocol
- Optional explicit mutual authentication

AuCPace Augmentation layer

$x \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$		
$X = B^{x \cdot c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
$W, \text{salt} = \text{lookup}W(\text{user})$		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_\sigma(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_\$ \{0,1\}^{k2}$,		abort if X invalid
else $PRS = W^{x \cdot c_{\mathcal{J}}}$		$PRS = X^{w \cdot c_{\mathcal{J}}}$

CPace substep

$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$		$g' = \mathsf{H}_1(ssid \Vert PRS \Vert CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a \cdot c_{\mathcal{J}}}$		$Y_b = G^{y_b \cdot c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a \cdot c_{\mathcal{J}}}$		$K = Y_a^{y_b \cdot c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \Vert K)$		$sk_1 = \mathsf{H}_2(ssid \Vert K)$

Explicit mutual authentication

$T_a = \mathsf{H}_3(ssid \Vert sk_1)$		$T_a = \mathsf{H}_3(ssid \Vert sk_1)$
$T_b = \mathsf{H}_4(ssid \Vert sk_1)$		$T_b = \mathsf{H}_4(ssid \Vert sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid \Vert sk_1)$		$sk = \mathsf{H}_5(ssid \Vert sk_1)$

AuCPace in a nutshell

1. Password verifiers W

2. Session establishment

AuCPace in a nutshell

The password verifier W is calculated in two steps.

$$\mathrm{salt} \leftarrow_{\$} \{0,1\}^{l}$$

$$w = \mathsf{PBKDF}_{\sigma}(pw,\ \mathrm{username},\ \mathrm{salt})$$

$$\boxed{W} = B^{w\ c_{\mathcal{J}}}$$

AuCPace in a nutshell

The password verifier W is calculated in two steps.

- Memory hard password hash

$$\text{salt} \leftarrow_{\$} \{0,1\}^{l}$$

$$w = \boxed{\mathsf{PBKDF}_{\sigma}}(pw,\ \text{username},\ \text{salt})$$

$$W = B^{w\ c_{\mathcal{J}}}$$

AuCPace in a nutshell

The password verifier W is calculated in two steps as a combination of a

- Memory hard password hash

AuCPace25519:
scrypt, σ = (r = 8, N = 32768, p = 1)

$$\text{salt} \leftarrow_{\$} \{0,1\}^{l}$$

$$w = \boxed{\mathsf{PBKDF}_{\sigma}}(pw,\ \text{username},\ \text{salt})$$

$$W = B^{w\ c_{\mathcal{J}}}$$

Endress+Hauser

AuCPace in a nutshell

The password verifier W is calculated in two steps as a combination of a

- Memory hard password hash
- Fixed-Base-Point Diffie-Hellman group operation

AuCPace25519:
X25519

$$\text{salt} \leftarrow_{\$} \{0,1\}^{l}$$

$$w = \mathsf{PBKDF}_{\sigma}(pw,\ \text{username},\ \text{salt})$$

$$W = \boxed{B^{w\ c_{\mathcal{J}}}}$$

AuCPace in a nutshell

The password verifier W is calculated in two steps as a combination of a

- Memory hard password hash
- Fixed-Base-Point Diffie-Hellman group operation

AuCPace proofs explicitly consider
non-prime-order groups with small co-factors

$$\text{salt} \leftarrow_{\$} \{0,1\}^{l}$$

$$w = \mathsf{PBKDF}_{\sigma}(pw,\ \text{username},\ \text{salt})$$

$$W = B^{w \boxed{c_{\mathcal{J}}}}$$

The modular AuCPace protocol construction

Session key establishment:

Client has access to clear-text password “pw”

Server has access to verifier “W”

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$		
$X = B^{x \cdot c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
W, salt $=$ lookup$W$(user)		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k2}$,		abort if X invalid
else $PRS = W^{x \cdot c_{\mathcal{J}}}$		$PRS = X^{w \cdot c_{\mathcal{J}}}$

CPace substep

$g' = \mathsf{H}_1(ssid \| PRS \| CI)$		$g' = \mathsf{H}_1(ssid \| PRS \| CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a \cdot c_{\mathcal{J}}}$		$Y_b = G^{y_b \cdot c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a \cdot c_{\mathcal{J}}}$		$K = Y_a^{y_b \cdot c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \| K)$		$sk_1 = \mathsf{H}_2(ssid \| K)$

Explicit mutual authentication

$T_a = \mathsf{H}_3(ssid \| sk_1)$		$T_a = \mathsf{H}_3(ssid \| sk_1)$
$T_b = \mathsf{H}_4(ssid \| sk_1)$		$T_b = \mathsf{H}_4(ssid \| sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid \| sk_1)$		$sk = \mathsf{H}_5(ssid \| sk_1)$

AuCPace in a nutshell

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$

$X = B^{x\, c_{\mathcal{J}}}$

$\xleftarrow{\text{username}}$

$W, \text{salt} = \text{lookup}W(\text{user})$

$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$

$w = \mathsf{PBKDF}_\sigma(pw, \text{user}, \text{salt})$

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$,

else $PRS = W^{x\, c_{\mathcal{J}}}$

abort if X invalid

$PRS = X^{w\, c_{\mathcal{J}}}$

AuCPace in a nutshell

AuCPace Augmentation layer

$\boxed{\begin{array}{l} x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\} \\ X = B^{x\, c_{\mathcal{J}}} \end{array}}$

$\xleftarrow{\text{username}}$

$W, \text{salt} = \text{lookupW}(\text{user})$

$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$

$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$, abort if X invalid

else $PRS = W^{x\, c_{\mathcal{J}}}$ $PRS = X^{w\, c_{\mathcal{J}}}$

Server generates DH key pair (x , X)

Ephemeral: “full augmentation” or static: “partial augmentation”

AuCPace in a nutshell

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$

$X = B^{x\ c_{\mathcal{J}}}$

username $\longleftarrow$

W,salt $=$ lookup$W$(user)

$\mathcal{J}, X, \mathrm{salt}, \sigma$ $\longrightarrow$

$w = \mathsf{PBKDF}_{\sigma}(pw, \text{ user, salt})$

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$,

else $PRS = W^{x\ c_{\mathcal{J}}}$

abort if X invalid

$PRS = X^{w\ c_{\mathcal{J}}}$

Username and password hashing information is exchanged

AuCPace in a nutshell

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$

$X = B^{x\, c_{\mathcal{J}}}$

username

$\longleftarrow$

$\boxed{W, \text{salt} = \text{lookup}W(\text{user})}$

$\mathcal{J}, X, \text{salt}, \sigma$

$\longrightarrow$

$\boxed{w = \mathsf{PBKDF}_{\sigma}(pw, \text{ user}, \text{ salt})}$

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$,

else $PRS = W^{x\, c_{\mathcal{J}}}$

abort if X invalid

$PRS = X^{w\, c_{\mathcal{J}}}$

Password verifier lookup // Password hash calculation

AuCPace in a nutshell

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$

$X = B^{x\, c_{\mathcal{J}}}$

Client and server generate a shared DH-style secret PRS (Password-Related String)

username

$\longleftarrow$

$W, \text{salt} = \text{lookup}W(\text{user})$

$\mathcal{J}, X, \text{salt}, \sigma$

$\longrightarrow$

$w = \text{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$

abort if X invalid

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$,

else $\boxed{PRS = W^{x\, c_{\mathcal{J}}}}$

$\boxed{PRS = X^{w\, c_{\mathcal{J}}}}$

AuCPace in a nutshell

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$

$X = B^{x \, c_{\mathcal{J}}}$

PRS is passed as parameter to the balanced CPace protocol substep

username

$\longleftarrow$

$W, \text{salt} = \text{lookup}W(\text{user})$

$\mathcal{J}, X, \text{salt}, \sigma$

$\longrightarrow$

$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$

if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$,

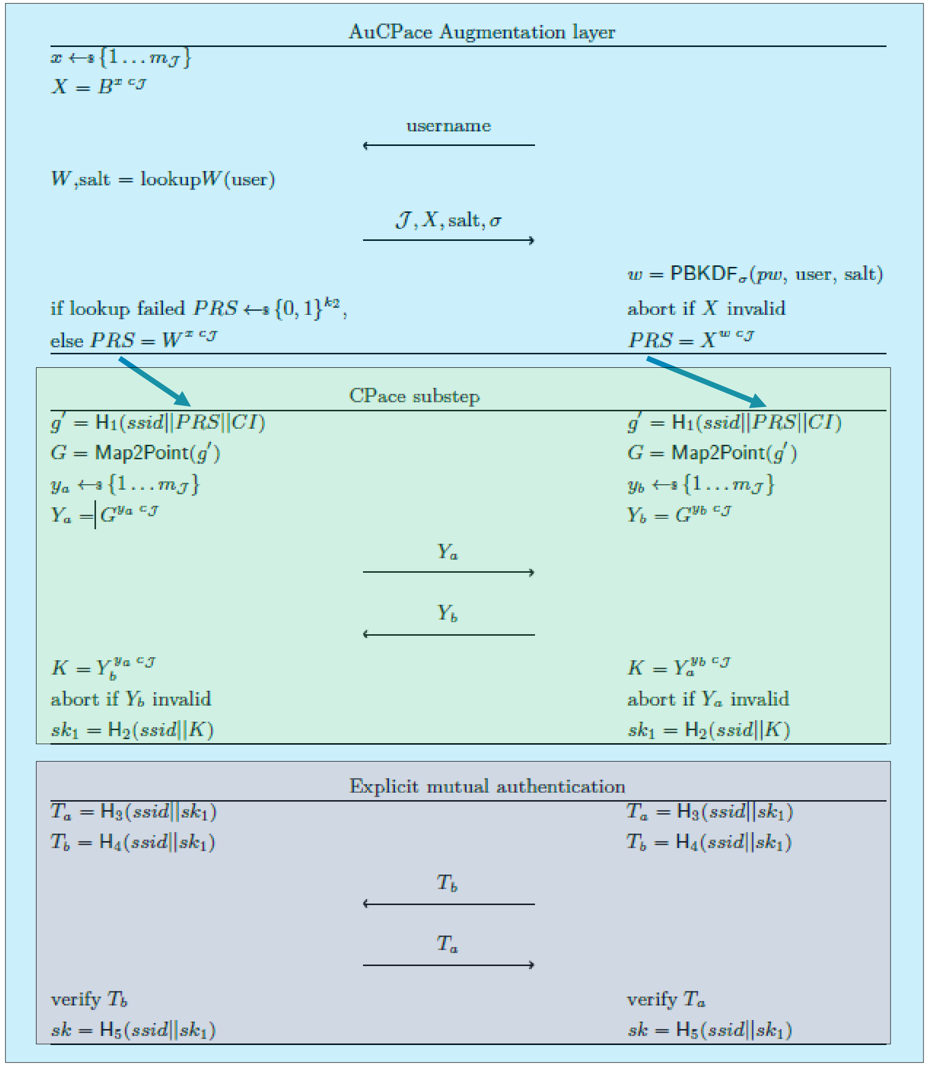
abort if X invalid

else $\boxed{PRS = W^{x \, c_{\mathcal{J}}}}$

$\boxed{PRS = X^{w \, c_{\mathcal{J}}}}$

The modular AuCPace protocol construction

Three subcomponents within AuCPace



Endress+Hauser

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid \| PRS \| CI)$		$g' = \mathsf{H}_1(ssid \| PRS \| CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\, c_{\mathcal{J}}}$		$Y_b = G^{y_b\, c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a\, c_{\mathcal{J}}}$		$K = Y_a^{y_b\, c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \| K)$		$sk_1 = \mathsf{H}_2(ssid \| K)$

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \ldots m_\mathcal{J}\}$		$y_b \leftarrow_\$ \{1 \ldots m_\mathcal{J}\}$
$Y_a = G^{y_a\, c_\mathcal{J}}$		$Y_b = G^{y_b\, c_\mathcal{J}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a\, c_\mathcal{J}}$		$K = Y_a^{y_b\, c_\mathcal{J}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$

Both sides calculate an ephemeral generator G for DH (as in PACE [BFK09])

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\, c_{\mathcal{J}}}$		$Y_b = G^{y_b\, c_{\mathcal{J}}}$
	$\xrightarrow{\quad Y_a \quad}$	
	$\xleftarrow{\quad Y_b \quad}$	
$K = Y_b^{y_a\, c_{\mathcal{J}}}$		$K = Y_a^{y_b\, c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$

This also involves all relevant associated data to authenticate ("channel identifier" CI)

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid \| PRS \| CI)$		$g' = \mathsf{H}_1(ssid \| PRS \| CI)$
$G = \boxed{\mathsf{Map2Point}(g')}$		$G = \boxed{\mathsf{Map2Point}(g')}$
$y_a \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\, c_{\mathcal{J}}}$		$Y_b = G^{y_b\, c_{\mathcal{J}}}$
	$\xrightarrow{\quad Y_a \quad}$	
	$\xleftarrow{\quad Y_b \quad}$	
$K = Y_b^{y_a\, c_{\mathcal{J}}}$		$K = Y_a^{y_b\, c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid \| K)$		$sk_1 = \mathsf{H}_2(ssid \| K)$

AuCPace25519 uses Elligator2 and SHA512

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\ c_{\mathcal{J}}}$		$Y_b = G^{y_b\ c_{\mathcal{J}}}$
	$\xrightarrow{\quad Y_a \quad}$	
	$\xleftarrow{\quad Y_b \quad}$	
$K = Y_b^{y_a\ c_{\mathcal{J}}}$		$K = Y_a^{y_b\ c_{\mathcal{J}}}$
abort if Y_b invalid	Diffie-Hellman step allows for x-coordinate-only algorithms	abort if Y_a invalid
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow_\$ \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\, c_{\mathcal{J}}}$		$Y_b = G^{y_b\, c_{\mathcal{J}}}$
	$\xrightarrow{\quad Y_a \quad}$	
	$\xleftarrow{\quad Y_b \quad}$	
$K = Y_b^{y_a\, c_{\mathcal{J}}}$		$K = Y_a^{y_b\, c_{\mathcal{J}}}$
$\boxed{\text{abort if } Y_b \text{ invalid}}$		$\boxed{\text{abort if } Y_a \text{ invalid}}$
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$

Design allows for simplified point verification for groups with a secure quadratic twist.

AuCPace in a nutshell

CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \ldots m_\mathcal{J}\}$		$y_b \leftarrow_\$ \{1 \ldots m_\mathcal{J}\}$
$Y_a = G^{y_a\, c_\mathcal{J}}$		$Y_b = G^{y_b\, c_\mathcal{J}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a\, c_\mathcal{J}}$		$K = Y_a^{y_b\, c_\mathcal{J}}$
abort if Y_b invalid		abort if Y_a invalid
$\boxed{sk_1} = \mathsf{H}_2(ssid\|\|K)$		$\boxed{sk_1} = \mathsf{H}_2(ssid\|\|K)$

Generated session keys match iff both input parameters PRS and associated data “CI” match

Endress+Hauser

AuCPace in a nutshell

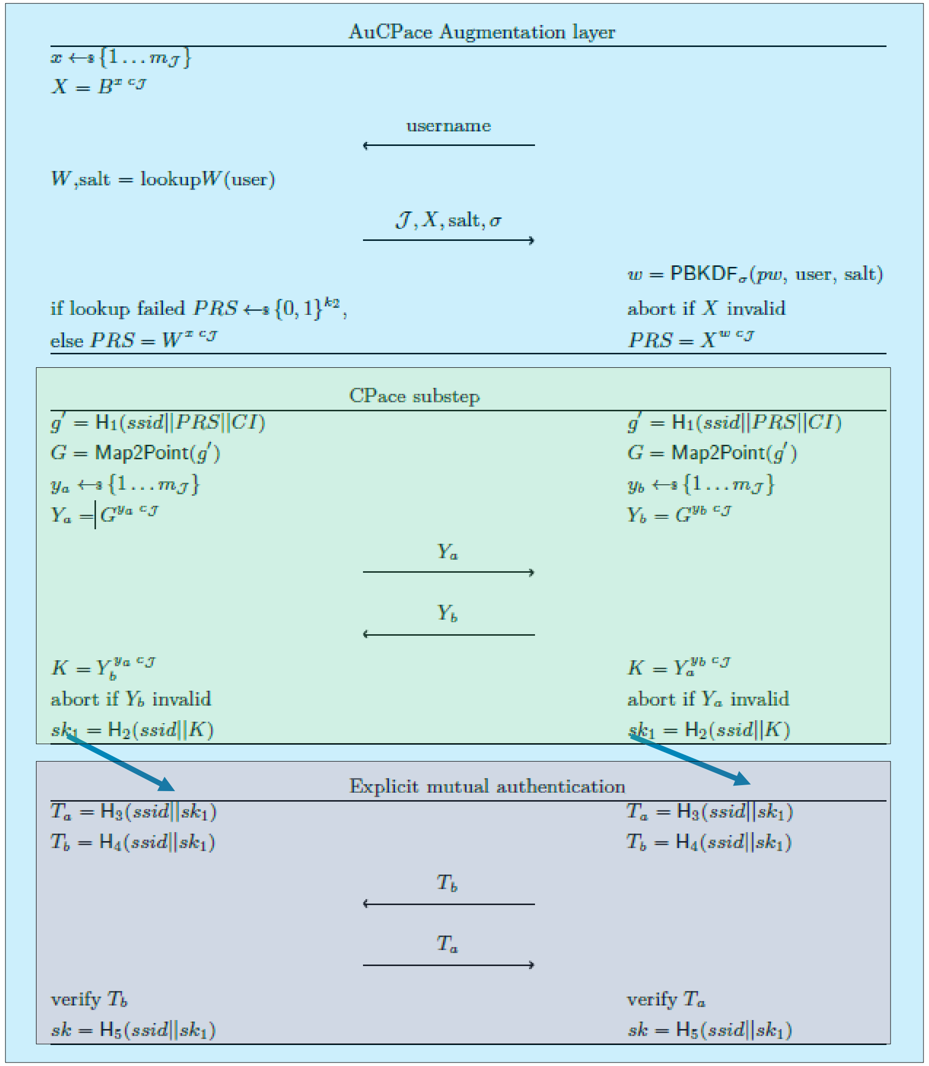
CPace substep

$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathsf{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow_\$ \{1 \dots m_\mathcal{J}\}$		$y_b \leftarrow_\$ \{1 \dots m_\mathcal{J}\}$
$Y_a = G^{y_a \, c_\mathcal{J}}$		$Y_b = G^{y_b \, c_\mathcal{J}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a \, c_\mathcal{J}}$		$K = Y_a^{y_b \, c_\mathcal{J}}$
abort if Y_b invalid		abort if Y_a invalid
$\boxed{sk_1} = \mathsf{H}_2(ssid\|\|K)$		$\boxed{sk_1} = \mathsf{H}_2(ssid\|\|K)$

Optionally, session keys are explicitly authenticated

The modular AuCPace protocol construction

Three subcomponents within AuCPace



Endress+Hauser

AuCPace in a nutshell

Explicit mutual authentication

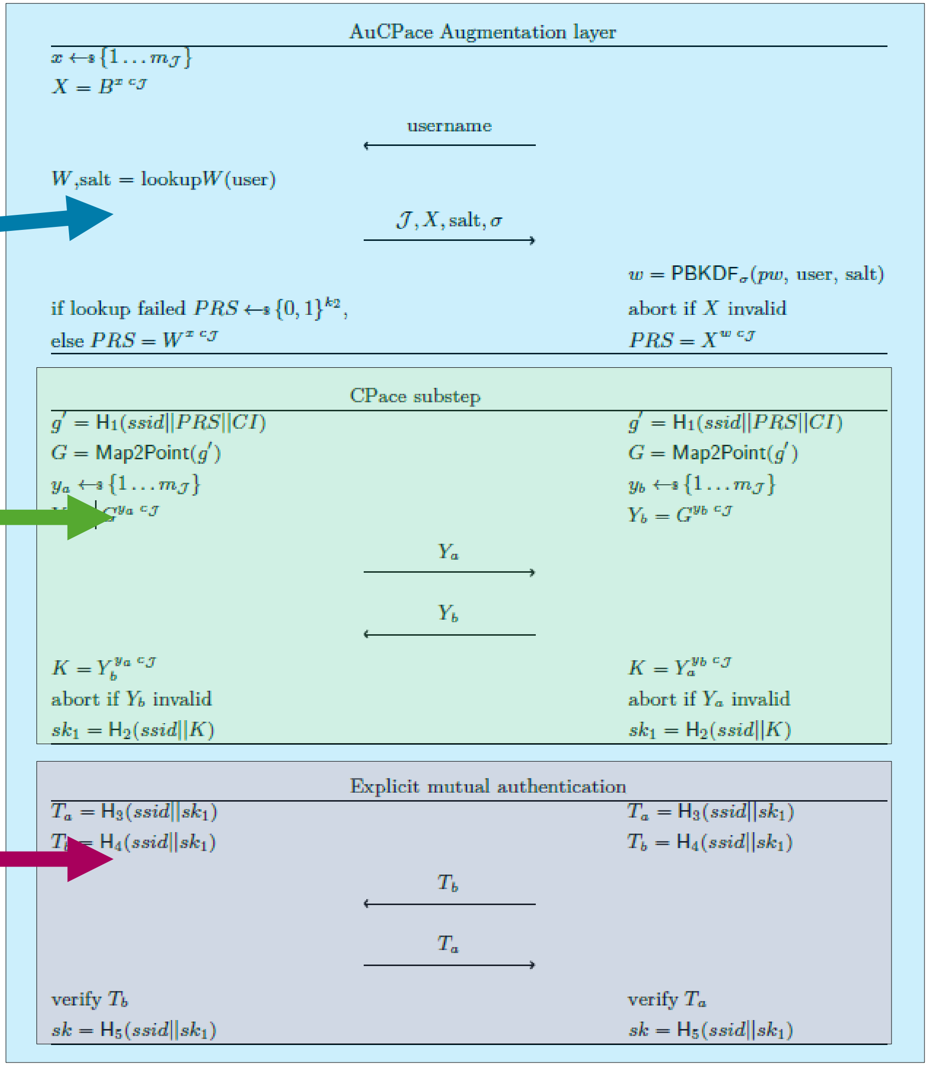
$T_a = \mathsf{H}_3(ssid\|\boxed{sk_1})$		$T_a = \mathsf{H}_3(ssid\|\boxed{sk_1})$
$T_b = \mathsf{H}_4(ssid\|\boxed{sk_1})$		$T_b = \mathsf{H}_4(ssid\|\boxed{sk_1})$
	$\xleftarrow{\quad T_b \quad}$	
	$\xrightarrow{\quad T_a \quad}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid\|\boxed{sk_1})$		$sk = \mathsf{H}_5(ssid\|\boxed{sk_1})$

Note that no communication transcripts were necessary for generating the session keys and authentication messages!

The modular AuCPace protocol construction // Security analysis

Three subcomponents within AuCPace

- AuCPace augmentation layer
- CPace balanced PAKE protocol
- Optional explicit mutual authentication



Endress+Hauser

The modular AuCPace protocol construction

Security analysis – 1 –

Proof that CPace protocol executions are indistinguishable from an ideal functionality

$$\mathcal{F}_{\mathrm{pwKE}}$$

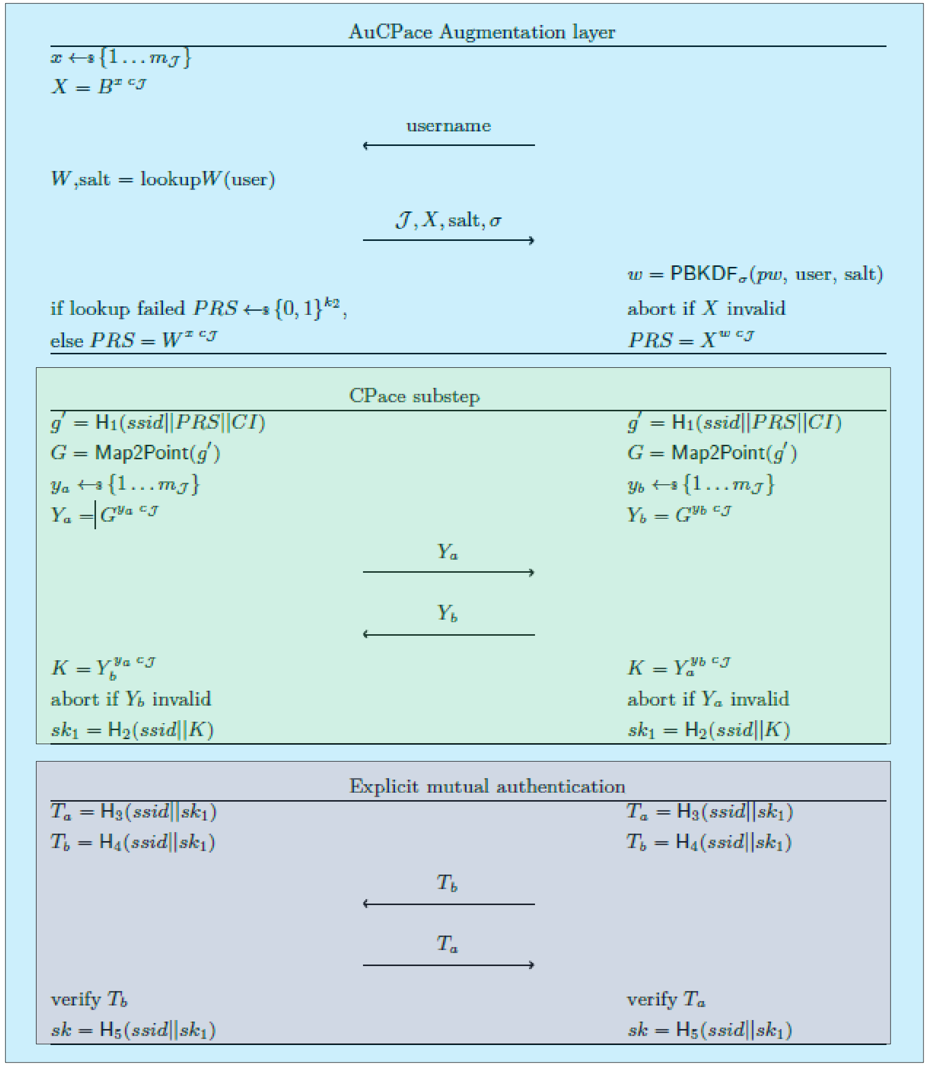
[CHK+05] for an observing environment

$$\mathcal{Z}$$

for all real-world adversaries

$$\mathcal{A}$$

under the specified hardness assumptions



Endress+Hauser

The modular AuCPace protocol construction

Security analysis – 2 –

Replace CPace in AuCPace with $\mathcal{F}_{\text{pwKE}}$

AuCPace Augmentation layer

$x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$ $X = B^{x \cdot c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
$W, \text{salt} = \text{lookup}W(\text{user})$		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k2}$, else $PRS = W^{x \cdot c_{\mathcal{J}}}$		abort if X invalid $PRS = X^{w \cdot c_{\mathcal{J}}}$

$\mathcal{F}_{\text{pwKE}}$

The functionality $\mathcal{F}_{\text{pwKE}}$ is parametrized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties via the following queries:

Upon receiving a query (NewSession, sid, P_i, P_j, pw, role) **from party P_i :**
Send (NewSession, sid, P_i, P_j, role) to $\mathcal{S}$. In addition, if this is the first NewSession query, or if this is the second NewSession query and there is a record (P_j, P_i, pw'), then record (P_i, P_j, pw) and mark this record fresh.

Upon receiving a query (TestPwd, sid, P_i, pw') **from the adversary S :**
If there is a record of the form (P_i, P_j, pw) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to $\mathcal{S}$ with "correct guess". If $pw \neq pw'$, mark the record interrupted and reply with "wrong guess".

Upon receiving a query (NewKey, sid, P_i, sk) **from $\mathcal{S}$ where $|sk| = k$:**
If there is a record of the form (P_i, P_j, pw), and this is the first NewKey query for P_i, then:

- If this record is compromised, or either P_i or P_j is corrupted, then output (sid, sk) to player P_i.
- If this record is fresh, and there is a record (P_j, P_i, pw') with $pw' = pw$, and a key sk' was sent to P_j and (P_j, P_i, pw) was fresh at the time, then output (sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sk') to P_i.

Either way, mark the record (P_i, P_j, pw) as completed.

Explicit mutual authentication

$T_a = \mathsf{H}_3(ssid \Vert sk_1)$ $T_b = \mathsf{H}_4(ssid \Vert sk_1)$		$T_a = \mathsf{H}_3(ssid \Vert sk_1)$ $T_b = \mathsf{H}_4(ssid \Vert sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b $sk = \mathsf{H}_5(ssid \Vert sk_1)$		verify T_a $sk = \mathsf{H}_5(ssid \Vert sk_1)$

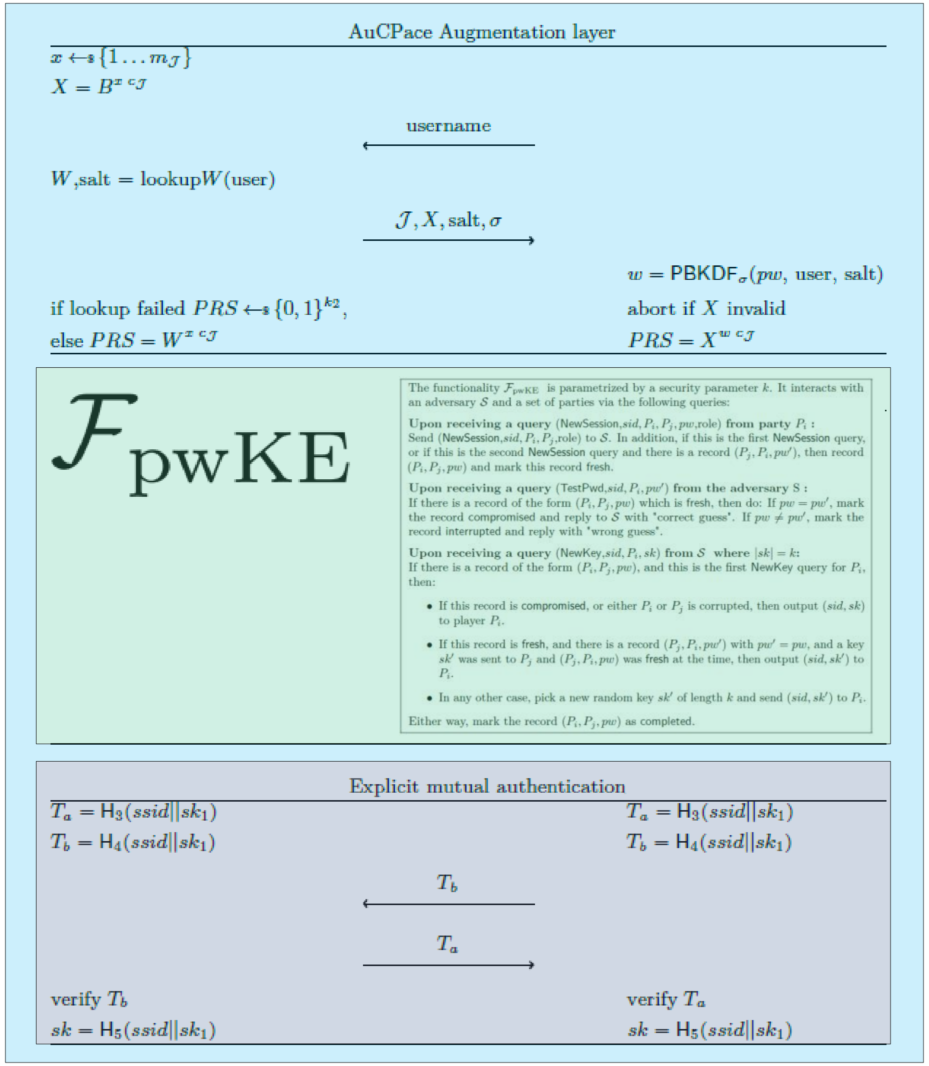
The modular AuCPace protocol construction

Security analysis – 3 –

Proof that execution of AuCPace protocol runs that use $\mathcal{F}_{\mathrm{pwKE}}$ are indistinguishable from executions using the ideal functionality

$$\mathcal{F}_{\mathrm{apwKE}}$$

[GMR06]



Endress+Hauser

The modular AuCPace protocol construction

Security analysis – 4 –

Conclusion: AuCPace is a secure verifier-based PAKE protocol

$\mathcal{F}_{\text{apwKE}}$

The functionality $\mathcal{F}_{\text{apwKE}}$ is parametrized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties via the following queries:

Password storage and authentication sessions

Upon receiving a query (StorePWfile, sid, P_i, P_j, pw) **from party** P_i **:**
If this is the first StorePWfile query, store password data record (file, P_i, P_j, pw) and mark it uncompromised.

Upon receiving a query (CltSession, $sid, ssid, P_i, pw$) **from party** P_i **:**
Send (CltSession, $sid, ssid, P_j, P_j$) to $\mathcal{S}$, and if this is the first CltSession query for $ssid$, store session record ($ssid, P_i, P_j, pw$) and mark it fresh.

Upon receiving a query (SvrSession, $sid, ssid$) **from party** P_j **:**
If there is a password data record (file, P_i, P_j, pw) then send (SvrSession, $sid, ssid, P_i, P_j$) to $\mathcal{S}$, and if this is the first SvrSession query for $ssid$, store session record ($ssid, P_j, P_i, pw'$) and mark it fresh.

Stealing password data

Upon receiving a query (StealPWfile, sid) **from adversary** $\mathcal{S}$**:**
If there is no password data record, reply to $\mathcal{S}$ with 'no password file'. Otherwise do the following. If the password data record (file, P_i, P_j, pw) is marked uncompromised, mark it as compromised. if there is a tuple (offline, pw') stored with $pw = pw'$, send pw to $\mathcal{S}$, otherwise reply to $\mathcal{S}$ with 'password file stolen'.

Upon receiving a query (OfflineTestPwd, sid, pw') **from adversary** $\mathcal{S}$**:**
If there is no password data record, or if there is a password record (file, P_i, P_j, pw) that is marked uncompromised, then store (offline, pw'). Otherwise, do: If $pw = pw'$, reply to $\mathcal{S}$ with 'correct guess'. If $pw \neq pw'$, reply with 'wrong guess'.

Active session attacks

Upon receiving a query (TestPwd, $sid, ssid, P, pw'$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to $\mathcal{S}$ with 'correct guess'. Otherwise, mark the session record interrupted and reply with 'wrong guess'.

Upon receiving a query (SvrImpersonate, $sid, ssid$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P_i, P_j, pw$) which is fresh, then do: If there is a password data record (file, P_i, P_j, pw) that is marked compromised, mark the session record compromised and reply to $\mathcal{S}$ with 'correct guess', else mark the the session record interrupted and reply with 'wrong guess'.

Key Generation and Authentication

Upon receiving a query (NewKey, $sid, ssid, P, key$) **from adversary** $\mathcal{S}$**, where** $|key| = k$**:**
If there is a record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is compromised, or either P or P' is corrupted, then output ($sid, ssid, key$) to P.
- If this record is fresh, there is a session record ($ssid, P', P, pw'$), $pw' = pw$, a key key' was sent to P', and ($ssid, P', P, pw$) was fresh at the time, then let $key'' = key'$, else pick a random key key'' of length k. Output ($sid, ssid, key''$) to P.
- In any other case, pick a random key key'' of length k and output ($sid, ssid, key''$) to P.

Finally, mark the record ($ssid, P, P', pw$) as completed.

Upon receiving a query (TestAbort, $sid, ssid, P$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is fresh, there is a record ($ssid, P', P, pw'$),and $pw' = pw$, let $b' =$ succ.
- In any other case let $b' =$ fail

Send b' to $\mathcal{S}$. If $b' =$ fail, send (abort, $sid, ssid$) to P, and mark record ($ssid, P, P', pw$) completed.

The modular AuCPace protocol construction

Security analysis – 4 –

Conclusion: AuCPace is a secure verifier-based PAKE protocol *optionally allowing for explicit mutual authentication of session keys*

$\mathcal{F}_{\text{apwKE}}$

The functionality $\mathcal{F}_{\text{apwKE}}$ is parametrized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties via the following queries:

Password storage and authentication sessions

Upon receiving a query (StorePWfile, sid, P_i, P_j, pw) **from party** P_i **:**
If this is the first StorePWfile query, store password data record (file, P_i, P_j, pw) and mark it uncompromised.

Upon receiving a query (CltSession, $sid, ssid, P_i, pw$) **from party** P_i **:**
Send (CltSession, $sid, ssid, P_j, P_j$) to $\mathcal{S}$, and if this is the first CltSession query for $ssid$, store session record ($ssid, P_i, P_j, pw$) and mark it fresh.

Upon receiving a query (SvrSession, $sid, ssid$) **from party** P_j **:**
If there is a password data record (file, P_i, P_j, pw) then send (SvrSession, $sid, ssid, P_i, P_j$) to $\mathcal{S}$, and if this is the first SvrSession query for $ssid$, store session record ($ssid, P_j, P_i, pw'$) and mark it fresh.

Stealing password data

Upon receiving a query (StealPWfile, sid) **from adversary** $\mathcal{S}$**:**
If there is no password data record, reply to $\mathcal{S}$ with "no password file". Otherwise do the following. If the password data record (file, P_i, P_j, pw) is marked uncompromised, mark it as compromised. if there is a tuple (offline , pw') stored with $pw = pw'$, send pw to $\mathcal{S}$, otherwise reply to $\mathcal{S}$ with "password file stolen".

Upon receiving a query (OfflineTestPwd, sid, pw') **from adversary** $\mathcal{S}$**:**
If there is no password data record, or if there is a password record (file, P_i, P_j, pw) that is marked uncompromised, then store (offline , pw'). Otherwise, do: If $pw = pw'$, reply to $\mathcal{S}$ with "correct guess". If $pw \neq pw'$, reply with "wrong guess".

Active session attacks

Upon receiving a query (TestPwd, $sid, ssid, P, pw'$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to $\mathcal{S}$ with "correct guess". Otherwise, mark the session record interrupted and reply with "wrong guess".

Upon receiving a query (SvrImpersonate, $sid, ssid$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P_i, P_j, pw$) which is fresh, then do: If there is a password data record (file, P_i, P_j, pw) that is marked compromised, mark the session record compromised and reply to $\mathcal{S}$ with "correct guess", else mark the the session record interrupted and reply with "wrong guess".

Key Generation and Authentication

Upon receiving a query (NewKey, $sid, ssid, P, key$) **from adversary** $\mathcal{S}$**, where** $|key| = k$**:**
If there is a record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is compromised, or either P or P' is corrupted, then output ($sid, ssid, key$) to P.
- If this record is fresh, there is a session record ($ssid, P', P, pw'$), $pw' = pw$, a key key' was sent to P', and ($ssid, P', P, pw$) was fresh at the time, then let $key'' = key'$, else pick a random key key'' of length k. Output ($sid, ssid, key''$) to P.
- In any other case, pick a random key key'' of length k and output ($sid, ssid, key''$) to P.

Finally, mark the record ($ssid, P, P', pw$) as completed.

Upon receiving a query (TestAbort, $sid, ssid, P$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is fresh, there is a record ($ssid, P', P, pw'$),and $pw' = pw$, let $b' =$ succ.
- In any other case let $b' =$ fail

Send b' to $\mathcal{S}$. If $b' =$ fail, send (abort, $sid, ssid$) to P, and mark record ($ssid, P, P', pw$) completed.

Endress+Hauser

The modular AuCPace protocol construction

AuCPace security assumptions:

- Computational Diffie-Hellman problem (CDH)
- Discrete log of S' = Map2Point(s) unknown.
- Programmable random oracle $\mathcal{F}_{\mathrm{RO}}$
- Upon availability of an inverse map Map2Point^{-1} security also maintained with respect to adaptive adversaries.

$\mathcal{F}_{\mathrm{apwKE}}$

The functionality $\mathcal{F}_{\mathrm{apwKE}}$ is parametrized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties via the following queries:

Password storage and authentication sessions

Upon receiving a query (StorePWfile,sid, P_i, P_j, pw) **from party** P_i **:**
If this is the first StorePWfile query, store password data record (file, P_i, P_j, pw) and mark it uncompromised.

Upon receiving a query (CltSession,$sid, ssid, P_i, pw$) **from party** P_i **:**
Send (CltSession, $sid, ssid, P_j, P_j$) to $\mathcal{S}$, and if this is the first CltSession query for $ssid$, store session record ($ssid, P_i, P_j, pw$) and mark it fresh.

Upon receiving a query (SvrSession,$sid, ssid$) **from party** P_j **:**
If there is a password data record (file,P_i, P_j, pw) then send (SvrSession,$sid, ssid, P_i, P_j$) to $\mathcal{S}$, and if this is the first SvrSession query for $ssid$, store session record ($ssid, P_j, P_i, pw'$) and mark it fresh.

Stealing password data

Upon receiving a query (StealPWfile,sid) **from adversary** $\mathcal{S}$**:**
If there is no password data record, reply to $\mathcal{S}$ with 'no password file'. Otherwise do the following. If the password data record (file,P_i, P_j, pw) is marked uncompromised, mark it as compromised. if there is a tuple (offline ,pw') stored with $pw = pw'$, send pw to $\mathcal{S}$, otherwise reply to $\mathcal{S}$ with 'password file stolen'.

Upon receiving a query (OfflineTestPwd,sid, pw') **from adversary** $\mathcal{S}$**:**
If there is no password data record, or if there is a password record (file,P_i, P_j, pw) that is marked uncompromised, then store (offline ,pw'). Otherwise, do: If $pw = pw'$, reply to $\mathcal{S}$ with 'correct guess'. If $pw \neq pw'$, reply with 'wrong guess'.

Active session attacks

Upon receiving a query (TestPwd,$sid, ssid, P, pw'$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to $\mathcal{S}$ with 'correct guess'. Otherwise, mark the session record interrupted and reply with 'wrong guess'.

Upon receiving a query (SvrImpersonate,$sid, ssid$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P_i, P_j, pw$) which is fresh, then do: If there is a password data record (file, P_i, P_j, pw) that is marked compromised, mark the session record compromised and reply to $\mathcal{S}$ with 'correct guess', else mark the the session record interrupted and reply with 'wrong guess'.

Key Generation and Authentication

Upon receiving a query (NewKey,$sid, ssid, P, key$) **from adversary** $\mathcal{S}$**, where** $|key| = k$**:**
If there is a record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is compromised, or either P or P' is corrupted, then output ($sid, ssid, key$) to P.
- If this record is fresh, there is a session record ($ssid, P', P, pw'$),$pw' = pw$, a key key' was sent to P', and ($ssid, P', P, pw$) was fresh at the time, then let $key'' = key'$, else pick a random key key'' of length k. Output ($sid, ssid, key''$) to P.
- In any other case, pick a random key key'' of length k and output ($sid, ssid, key''$) to P.

Finally, mark the record ($ssid, P, P', pw$) as completed.

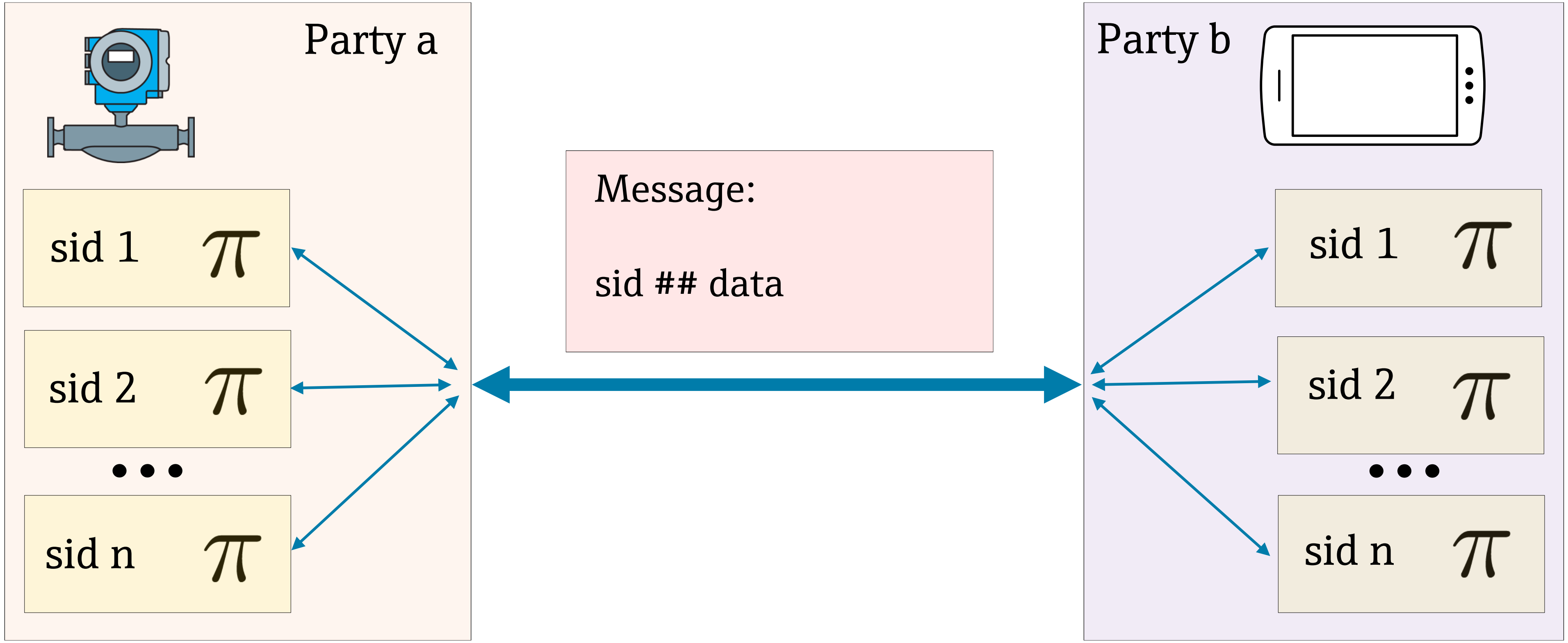
Upon receiving a query (TestAbort,$sid, ssid, P$) **from adversary** $\mathcal{S}$**:**
If there is a session record of the form ($ssid, P, P', pw$) that is not marked completed, then:

- If this record is fresh, there is a record ($ssid, P', P, pw'$),and $pw' = pw$, let $b' =$ succ.
- In any other case let $b' =$ fail

Send b' to $\mathcal{S}$. If $b' =$ fail, send (abort,$sid, ssid$) to P, and mark record ($ssid, P, P', pw$) completed.

Concurrent sessions within the UC framework

- UC[Can01] allows for an unlimited number of concurrently executed protocol instances π distinguished by a session ID (sid) (sid,ssid pair in JUC [CR03])



Concurrent sessions within the UC framework

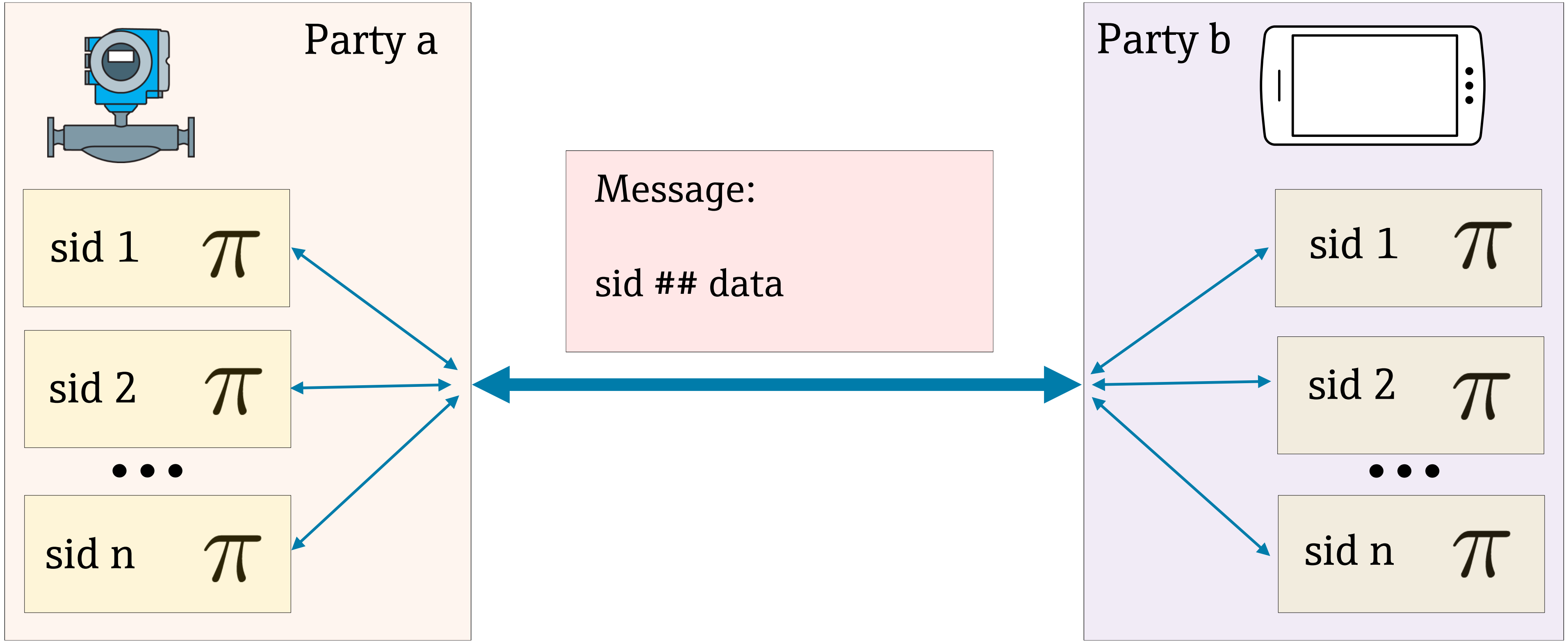
- Straight-forward approach for establishing *sid* in the real world: nonce-round prior to the protocol.

Server		**Client**
	Agree on $ssid$	
$s \leftarrow_\$ \{0,1\}^{k_1}$		$t \leftarrow_\$ \{0,1\}^{k_1}$
	$\xrightarrow{s}$	
	$\xleftarrow{t}$	
$ssid = \mathsf{H}_0(s\|\|t)$		$ssid = \mathsf{H}_0(s\|\|t)$

- In the literature this complexity coming with *any* UC security proof is not always considered to the same extend [JKX18,GMR06].

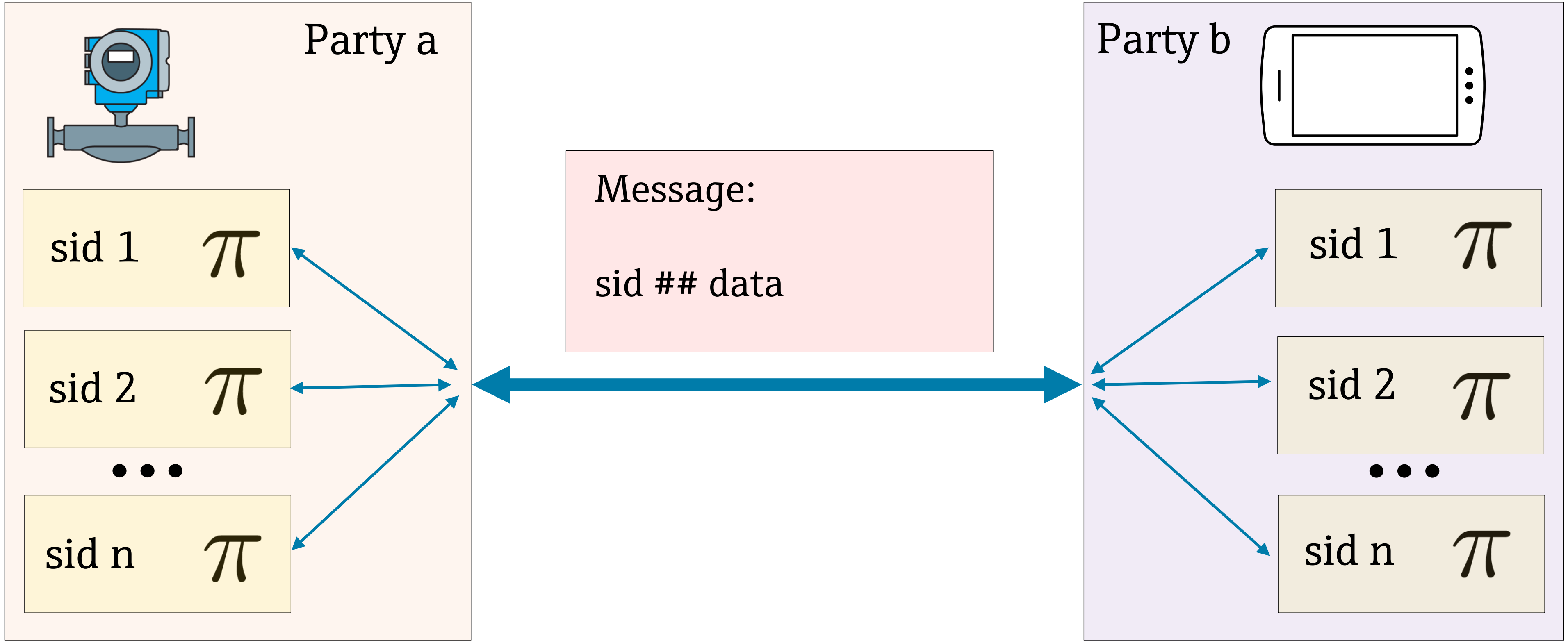
Concurrent sessions within the UC framework

- Proof technicality: *sid* needed for addressing purposes in the simulation environment (the UC Turing machines don't have something such as "concurrent TCP channels")



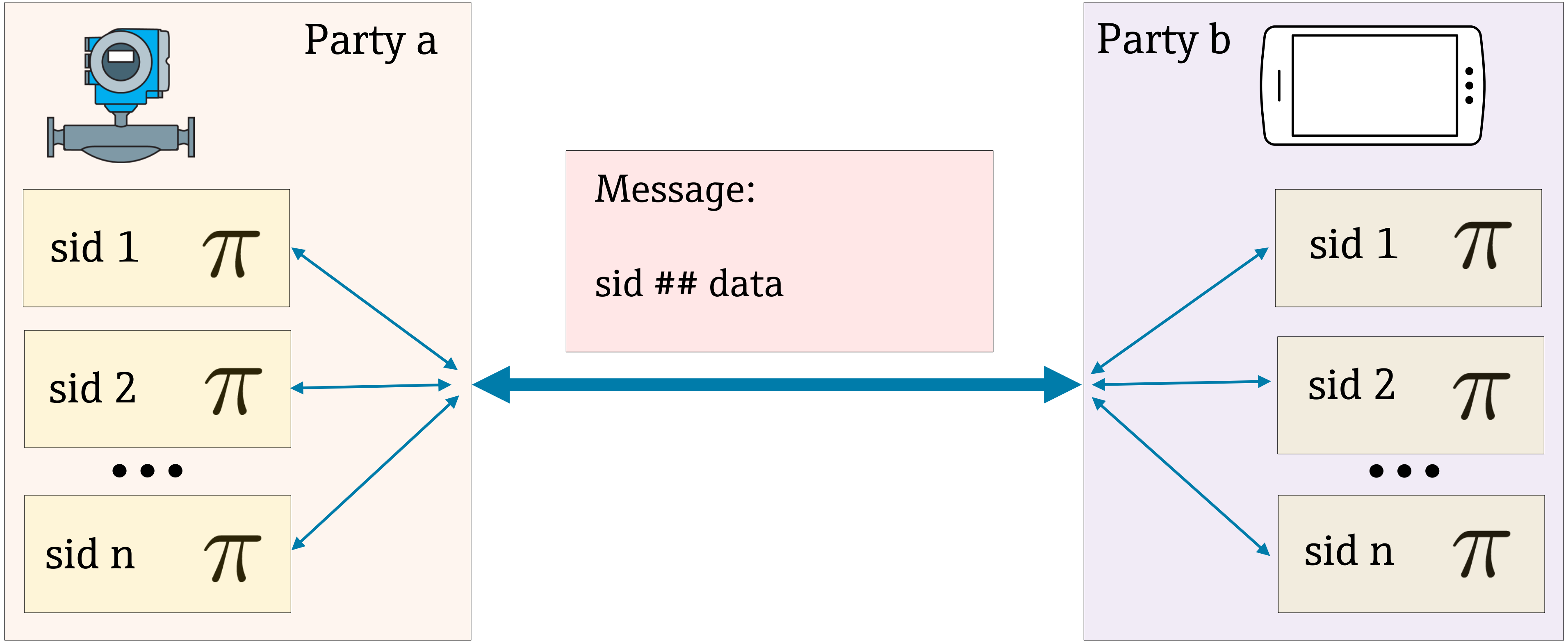
Concurrent sessions within the UC framework

- Proof technicality: *sid* needed for addressing purposes in the simulation environment
  (Need for addressing => Technical need for establishment **prior** to the protocol run)



Concurrent sessions within the UC framework

- Session IDs are sometimes also used for a session specific nonce value
  (Here: No technical need for nonce agreement *prior* to entering the protocol)



Use of the UC session ID as ephemeral nonce value in the AuCPace protocol

- AuCPace uses *sid* as nonce
- *sid* prepended to hash inputs
  => outputs become ephemeral
  => different *sid* never share queries to $\mathcal{F}_{\mathrm{RO}}$

Küsters, Tüngerthal and Rausch [KTR13]: doing so is important for composability guarantees when combining joint state with global random oracles (IITM model).

CPace su

$g' = \mathsf{H}_1(\boxed{ssid}||PRS||CI)$

$G = \mathsf{Map2Point}(g')$

$y_a \leftarrow_{\$} \{1 \ldots m_{\mathcal{J}}\}$

$Y_a = G^{y_a\ c_{\mathcal{J}}}$

$\longleftarrow$

$K = Y_b^{y_a\ c_{\mathcal{J}}}$

abort if Y_b invalid

$sk_1 = \mathsf{H}_2(\boxed{ssid}||K)$

Explicit ı

$T_a = \mathsf{H}_3(\boxed{ssid}||sk_1)$

$T_b = \mathsf{H}_4(\boxed{ssid}||sk_1)$

Comparison of different PAKE protocols

Following slides:

Comparison of AuCPace with the other augmented PAKE protocols that come with proven forward security.

- VTBPEKE: Pointcheval and Wang [PW17]
- OPAQUE: Jarecki, Krawczyk and Xu [JKX18]

Other related V-PAKE protocols:

- BSPAKE, SPAKE2+: (no security proof provided)

Comparison of different PAKE protocols

Following slides:

Comparison of AuCPace with the other augmented PAKE protocols that come with proven forward security.

- VTBPEKE: Pointcheval and Wang [PW17]
- OPAQUE: Jarecki, Krawczyk and Xu [JKX18]

Other related V-PAKE protocols:

- BSPAKE, SPAKE2+: (no security proof provided)

Protocols nominated in the currently ongoing PAKE selection process at CFRG

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
precomp. res.	optional	optional	no	yes
proof	UC	UC	BPR(ROR)	UC
comp. complexity server	2v	3v+1f	3v+1f+1i	3v+1f
comp. complexity client	3v	3v	3v+1f	4v+1f
x-coordinate only	possible	possible	-	-
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	≈ 96B	≈ 64B	≈ 64B	≈ 280B
total message size estimate	≈ 160B	≈ 160B	≈ 160B	≈ 280B
Map2Point necessary	yes	yes	no	yes

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
precomp. res.	optional	optional	no	yes
proof	UC	UC	BPR(ROR)	UC
comp. complexity server	2v	3v+1f	3v+1f+1i	3v+1f

AuCPace and OPAQUE provide stronger security guarantees than VTBPEKE by offering pre-computation attack resistance and universal composability.

In comparison to OPAQUE, AuCPace considers a more powerful adaptive adversary model.

Pre-computation attack resistance option of AuCPace

- Pre-computation attack resistance as introduced by Jarecki, Krawczyk and Xu [JKX18]
- The salt value for password hashing is kept secret from the adversary.
- Offline attacks become possible only after stealing the password database.

- See Appendix C of the updated eprint paper version as prepared for CFRG PAKE selection process (https://eprint.iacr.org/2018/286.pdf)

- Cost of this additional security feature for AuCPace:
  +1 scalar multiplication for server, +2 scalar multiplications + 1 inversion for client.

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
comp. complexity client	3v	3v	3v+1f	4v+1f
x-coordinate only	possible	possible	-	-
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	≈ 96B	≈ 64B	≈ 64B	≈ 280B
total message size estimate	≈ 160B	≈ 160B	≈ 160B	≈ 280B
Map2Point necessary	yes	yes	no	yes

OPAQUE and VTBPEKE are monolithic constructions and merge authentication and session key generation.
Require one message less than AuCPace.

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
comp. complexity server	2v	3v+1f	3v+1f+1i	3v+1f
comp. complexity client	3v	3v	3v+1f	4v+1f
x-coordinate only	possible	possible	-	-
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	≈ 96B	≈ 64B	≈ 64B	≈ 280B
total message size estimate	≈ 160B	≈ 160B	≈ 160B	≈ 280B
Map2Point necessary	yes	yes	no	yes

For OPAQUE the parallelism comes at the cost of significantly larger password verifiers, even when considering point compression.

Endress+Hauser

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c		
precomp. res.	optional	optior		
proof	UC	UC		
comp. complexity server	2v	3v+		
comp. complexity client	3v	3v		
x-coordinate only	possible	possit		
simplified point ver.	possible	possit		
pw-verifier size estimate	≈ 96B	≈ 64		
total message size estimate	≈ 160B	≈ 160B	≈ 160B	≈ 280B
Map2Point necessary	yes	yes	no	yes

AuCPace needs particularly little computational resources on constrained servers in the partially augmented configuration.

Main design target for our specific setting. [HL17]

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	≈ 96B	≈ 64B	≈ 64B	≈ 280B
total message size estimate	≈ 160B	≈ 160B	≈ 160B	≈ 280B
Map2Point necessary	yes	yes	no	yes

Unlike VTBPEKE both, AuCPace and OPAQUE don't mandatorily require explicit mutual authentication.

In case that explicit mutual authentication is not required by the application, one communication round could be avoided.

Efficiency comparison of different PAKE protocols

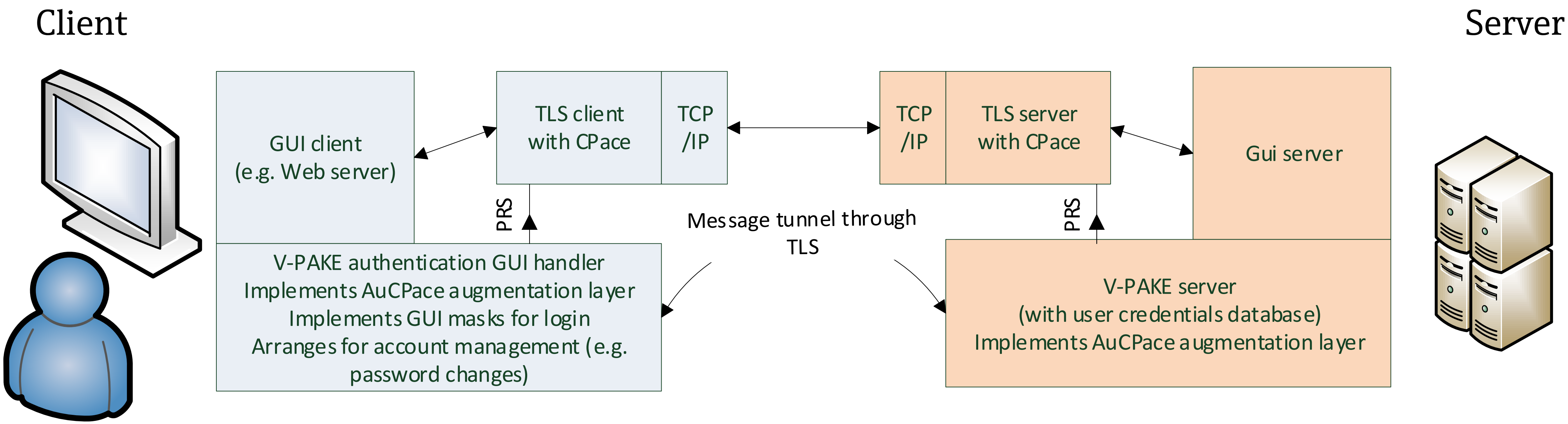
	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
total message size estimate	≈ [illegible]	≈ [illegible]	≈ [illegible]	≈ [illegible]
Map2Point necessary	yes	yes	no	yes

AuCPace: modular construction
Separation into an augmentation layer and balanced CPace.

Possible advantage for V-PAKE integration into transport layer

User account complexity of augmented PAKE could be better kept away from transport layer software components.

CFRG PAKE selection process: Suggestion for augmented PAKE (V-PAKE)

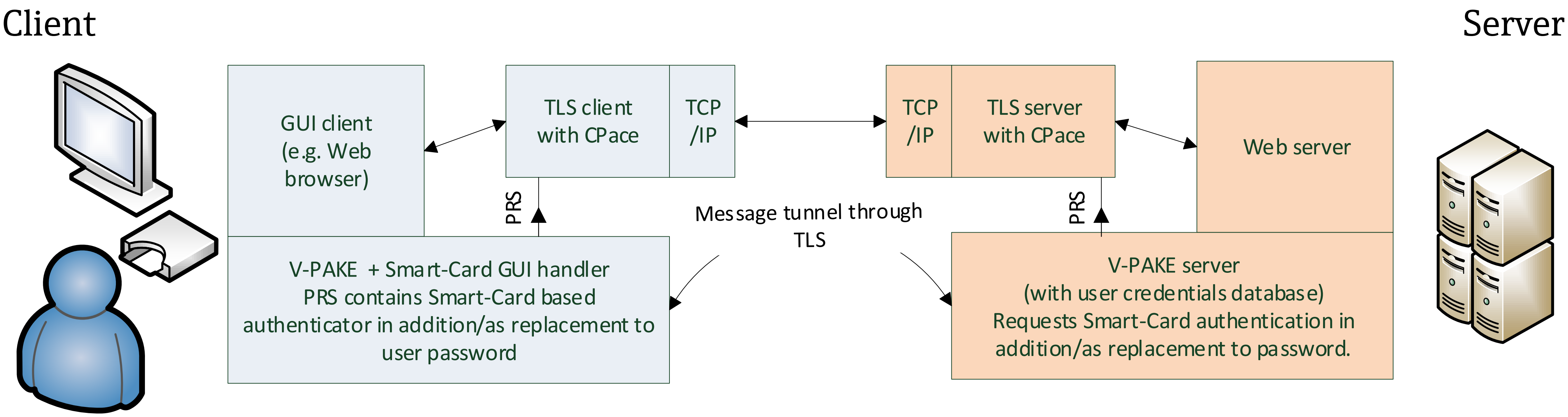


TLS implements a tunneling mechanism for authentication message exchange

TLS implements UC-secure balanced PAKE CPace

UC-Secure “augmentation layer” establishes ephemeral PRS on both sides using tunneled information messages in the TLS handshake and post-handshake phases.

Suggestion

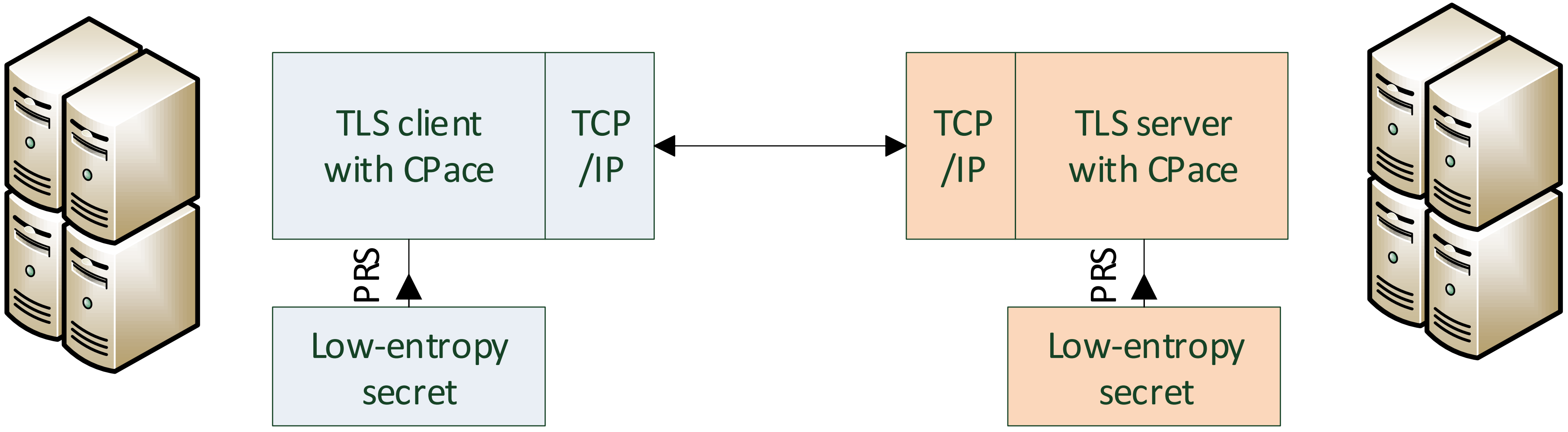


Future extensions (e.g. “UC-Secure smart-card-based authentication”, “UC-Secure fingerprint-based” authentication, RADIUS-server based authentication) could use the same TLS-CPace APIs for future extensions without need of modification of the TLS stack core.

Different ways of calculating the PRS input to CPace will be possible.

TLS-CPace just manages session confidentiality, integrity, forward secrecy and authenticates PRS.

Machine-Machine balanced Use-Case



- Machine/Machine interfaces could use CPace without an augmentation layer based on a pre-shared secret “PRS” which may be of low entropy.

Efficiency comparison of different PAKE protocols

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4		
message count pw-Registr.	1c	1c		
precomp. res.	optional	optional		
proof	UC	UC	B	
comp. complexity server	2v	3v+1f	3v+1f+1i	3v+1f
comp. complexity client	3v	3v	3v+1f	4v+1f
x-coordinate only	possible	possible	-	-
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	$\approx$ 96B	$\approx$ 64B	$\approx$ 64B	$\approx$ 280B
total message size estimate	$\approx$ 160B	$\approx$ 160B	$\approx$ 160B	$\approx$ 280B
Map2Point necessary	yes	yes	no	yes

AuCPace specifically designed for avoiding implementation pitfalls and for ease-of-implementation

Improvements regarding Elligator2 in comparison to [HL17]

- Standard (naive) implementation of Elligator2 [BHKL13] requires two separate field exponentiations (one for the inverse and one for the Legendre symbol).
- Using the inverse square root algorithm of [BDL+11]: one single exponentiation.
- Improvement accounts for about 4% of speed/power improvement regarding the balanced CPace protocol on a Cortex M0
- (Recall Riad Wahby‘s talk yesterday)

Fe25519 field operations on ARM Cortex M4

- Schoolbook multiplication strategy
- Sequence of partial word products optimized for keeping input operands and partial results in registers
- Important difference in comparison to previous speed record Hayato Fujii and Diego Aranha [FA17]: Merging integer arithmetic with reduction
- A+B, A-B, A + 121666 B as inline assembly

$A_i \times B_j$	A0	A1	A2	A3	A4	A5	A6	A7
B0	1	5	10	15	20	25	28	48
B1	0	6	11	16	21	26	29	31
B2	2	7	12	17	22	27	30	32
B3	3	8	13	18	23	49	50	51
B4	4	9	14	19	24	52	53	54
B5	33	36	39	42	45	55	56	57
B6	34	37	40	43	46	58	59	60
B7	35	38	41	44	47	61	62	63

$A_i \times A_j$	A0	A1	A2	A3	A4	A5	A6	A7
A0	1	2						
A1	0	3						
A2	5	6	15					
A3	4	11	12	19				
A4	8	9	16	23	32			
A5	7	13	20	24	27	34		
A6	10	17	21	25	28	30	35	
A7	14	18	22	26	29	31	33	36

Fe25519 field operations on ARM Cortex M4

- Schoolbook multiplication strategy
- Sequence of partial word products optimized for keeping input operands and partial results in registers
- Important difference
  record [FA17]:
  Merging integer arithmetic with reduction
- A+B, A-B, A + 121666 B as inline assembly

Assembly code created by use of automatic code generator handling register allocation. (correctness issue!)

$A_i \times B_j$	A0	A1	A2	A3	A4	A5	A6	A7
B0	1	5	10	15	20	25	28	48
B1	0	6	11	16	21	26	29	31
B2	2	7	12	17	22	27	30	32
B3	3	8	13	18	23	49	50	51
B4	4	9	14	19	24	52	53	54
B5	33	36	39	42	45	55	56	57
B6	34	37	40	43	46	58	59	60
B7	35	38	41	44	47	61	62	63

A_j	A0	A1	A2	A3	A4	A5	A6	A7
A0	1	2						
A1	0	3						
A2	5	6	15					
A3	4	11	12	19				
A4	8	9	16	23	32			
A5	7	13	20	24	27	34		
A6	10	17	21	25	28	30	35	
A7	14	18	22	26	29	31	33	36

Experimental results for fe25519 field operations

- Significant cycle-count improvement in comparison to previous speed record

Target	f	$x+y$	$x-y$	$*A_0$	$+*A_0$	x^2	$x*y$	
nRF51822	16	120	147	193	-	998	1478	$\mathbb{F}_{(2^{255}-19)}$, this work
STM32F411	?	73	77	129	-	563	631	$\mathbb{F}_{(2^{255}-19)}$, [DSS16]
MK20DX	72	86	86	76	-	252	276	$\mathbb{F}_{(2^{255}-19)}$, [FA17]
STM32F411	16	55	72	-	58	153	222	$\mathbb{F}_{(2^{255}-19)}$, this work
STM32L476	16	52	65	-	55	153	220	$\mathbb{F}_{(2^{255}-19)}$, this work
STM32L476	80	95	124	-	95	168	237	$\mathbb{F}_{(2^{255}-19)}$, this work
nRF52832	64	62	70	-	65	162	229	$\mathbb{F}_{(2^{255}-19)}$, this work
STM32F407	84	56	74		56	155	223	$\mathbb{F}_{(2^{255}-19)}$, this work
STM32F407	84	86	-	-	-	215	358	$\mathbb{F}_{(2^{127}-1)^2}$ [LLP+17]

Speed results for X25519 on Cortex M0 and Cortex M4

- Speed of X25519 competitive even in comparison with solutions using endomorphisms.

Target	f / MHz	X25519	
nRF51822	16	3,474,201	this work
STM32F411	?	1,816,351	[dG15]
STM32F411	?	1,563,852	[DSS16]
MK20DX	72	907,240	[FA17]
STM32L476	16; $80^{(p)}$; 80	609,779; 857,002; 971,272	this work
nRF52832	64	634,567	this work
STM32F411	16; $100^{(p)}$; 100	625,347; 625,449; 734,554	this work
STM32F407	16; 84(p); $168^{(p)}$; 168	625,358; 626,719; 655,891; 847,048	this work
?	?	548,873	[Len18]
STM32F407	$84^{(p)}$	542,900 (FourQ)	[LLP+17]

Speed results for X25519 and AuCPace

- Speed of our X25519 competitive even in comparison with solutions using endomorphisms.

Update August 2019: New X25519 speed record by Emil Lenngren [LEN18]

Full X25519 in assembly using non-standard ABI function calls passing full fe25519 operands in registers.

=> even fewer operand load/store operations

STM32F411	16; $100^{(p)}$; 100	625,347; 625,449; 734,554	this work
STM32F407	16; 84(p); $168^{(p)}$; 168	625,358; 626,719; 655,891; 847,048	this work
?	?	548,873	[Len18]
STM32F407	$84^{(p)}$	542,900 (FourQ)	[LLP+17]

Endress+Hauser

RAM/ROM requirements for AuCPace

Target Target	RAM ACE	ROM ACE	RAM X25519	ROM X25519	
Cortex-M0	264 (396)	11252	0 (572)	6108	this work
Cortex-M4	264 (268)	8896	0 (444)	3324	this work
Cortex-M4				4152	[FA17]
Cortex-M4				3786	[DSS16]

Table 7: Memory consumption in bytes for asynchronized implementation of AuCPace (ACE) and X25519 for Cortex M0 and M4 microcontrollers. Results were obtained with arm-none-eabi-gcc -O2 (gcc version 4.9.3). RAM consumption is separated in static memory (stack memory) respectively.

Endress+Hauser

Summary

- If you cannot avoid using password for remote access authentication, we recommend: V-PAKE + memory hard password hashing
- Result of our *system-level optimization strategy* for constrained servers: AuCPace and CPace
- AuCPace / CPace analysis in the UC framework
- AuCPace25519 and X25519 very efficient on ARM Cortex-M0 and M4, competitive even with the fastest known approaches benefiting from endomorphisms.

We thank all reviewers/referees from CHES and CFRG for their care with the manuscript and the constructive and helpful feedback.

Thank you very much for your attention

Updates from summer 2019 included in eprint version of the TCHES paper
https://eprint.iacr.org/2018/286.pdf (pre-computation attack resistance option)