Asiacrypt 2021

Security Analysis of CPace

Michel Abdalla
Dfinity
Zurich, Switzerland

Björn Haase
Endress & Hauser
Gerlingen, Germany

Julia Hesse
IBM Research - Europe
Zurich, Switzerland

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a high-entropy session key
- by use of a low-entropy password

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a high-entropy session key
- by use of a low-entropy password

Many protocols in the literature. Challenges for *efficient* designs:

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a high-entropy session key
- by use of a low-entropy password

Many protocols in the literature. Challenges for *efficient* designs:

- Idealized assumptions: Ideal cipher / Random-Oracle
- Implementation attacks
- Past: Protocol design and actual application hampered by patents

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a higl
- by use of a low

Many protocols in tl

- Idealized assumptions: Ideal cipher / Random-Oracle
- Implementation attacks
- Protocol design and actual application hampered by patents

Example: Encrypted Key Exchange (EKE)
Bellovin and Merrit S&P 1992

Proof requires an ideal cipher for encrypting group elements

This exact required assumption turned out to be the main complexity for any actual instantiation of the scheme.

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a higl
- by use of a low

Many protocols in tl

Example: Dragon Fly
D. Harkins

Non-Constant-Time mapping algorithm: Hunt-And-Peck

Exploit in WPA3 and EAP-pwd:
M. Vanhoef and E. Ronen, “Dragonblood”, IEEE S&P 2020

- Idealized assumptions: Ideal cipher / Random-Oracle
- Implementation attacks
- Protocol design and actual application hampered by patents

Password-Authenticated Key Exchange (PAKE)

Objective:

- Establish a higl
- by use of a low

Many protocols in tl

Example: Secure Remote Password (SRP)
T. Wu (NDSS, 1998)

Additional complexity introduced in SRP for the sake of patent circumvention.

Complexity prevented security-analysis.

- Idealized assumptions: Ideal cipher / Random-Oracle
- Implementation attacks
- Protocol design and actual application hampered by patents

CPace

Design objectives:

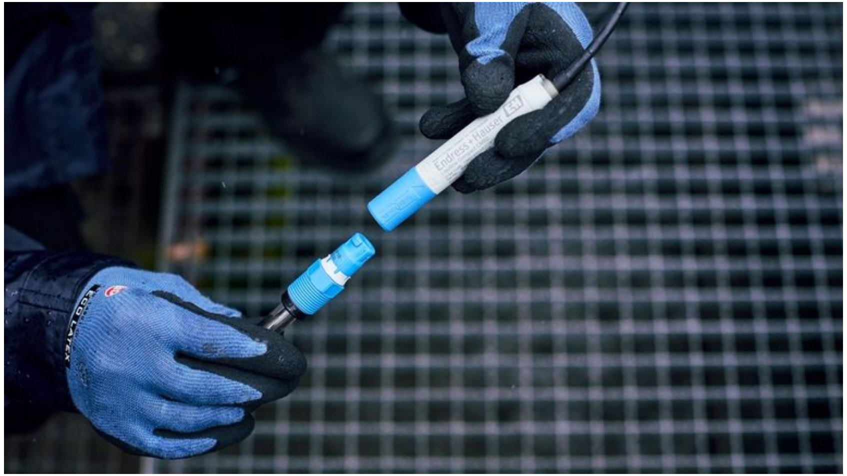
- Consider constrained devices
  - Minimize RAM consumption
  - Minimize computational complexity on microcontrollers
- Consider implementation pitfalls
- Allow for security proofs of the scheme

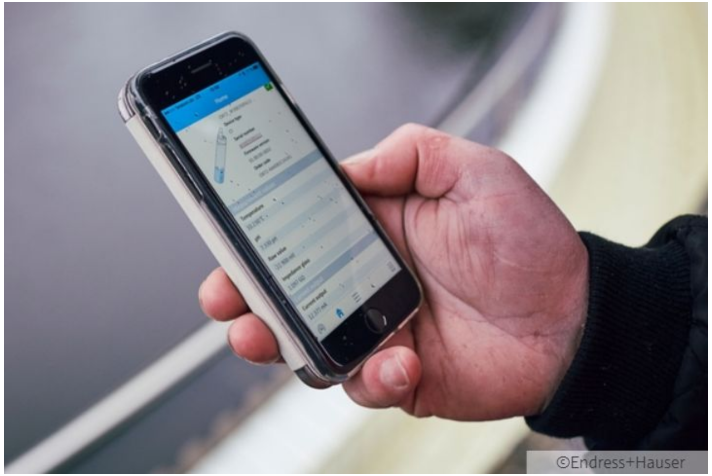
Side-aspect:

- Let’s make it annoying for quantum-computers

CPace is recommended for use in IETF protocols as a result of the CFRG PAKE selection process.

CPace: Example of target system





1.5 mW power budget for Wireless operation + Security

RAM Budget:, 2 kByte for Application + Security
PAKE: 660 bytes RAM (stack+static), 11,252 Bytes Flash

CPace

- CPace is derived from SPEKE (Jablon, 1997)
- Bases on Diffie-Hellman Key-Exchange

Recall: Diffie Hellmann Key Exchange in CPace notation

g		g
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g , y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Recall: Diffie Hellmann Key Exchange in CPace notation

g

$y_a \leftarrow \mathsf{ScSam}()$

$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$

g

$y_b \leftarrow \mathsf{ScSam}()$

$Y_b \leftarrow \mathsf{ScMul}(g, y_b)$

$$\xleftarrow{\quad Y_b \quad}$$

$$\xrightarrow{\quad Y_a \quad}$$

$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$

$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Agree on a common generator g of a group

Recall: Diffie Hellmann Key Exchange in CPace notation

g		g
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g, y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Sample secret scalar as ephemeral private key

Recall: Diffie Hellmann Key Exchange in CPace notation

g		g
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g, y_b)$
$\xleftarrow{Y_b}$
$\xrightarrow{Y_a}$
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Calculate public keys from private keys

Recall: Diffie Hellmann Key Exchange in CPace notation

g		g
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g, y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Exchange public keys

Recall: Diffie Hellmann Key Exchange in CPace notation

g		g
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g\ , y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$

Verify inputs and derive shared secrets

CPace

$g \leftarrow \mathsf{Gen}(\mathsf{pw}, \mathcal{P}', \mathcal{P})$		$g' \leftarrow \mathsf{Gen}(\mathsf{pw}', \mathcal{P}, \mathcal{P}')$
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g', y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$
Abort if $K = I_{\mathcal{G}}$		Abort if $K' = I_{\mathcal{G}}$
$ISK \leftarrow \mathsf{H}_2(K \| \mathsf{oc}(Y_a, Y_b))$		$ISK' \leftarrow \mathsf{H}_2(K' \| \mathsf{oc}(Y_a, Y_b))$
Output ISK		Output ISK'

CPace

Calculate generator from password and party identifiers

Party $\mathcal{P}$		Party $\mathcal{P}'$
$g \leftarrow \mathsf{Gen}(\mathsf{pw}, \mathcal{P}', \mathcal{P})$		$g' \leftarrow \mathsf{Gen}(\mathsf{pw}', \mathcal{P}, \mathcal{P}')$
$y_a \leftarrow \mathsf{ScSam}()$		$y_b \leftarrow \mathsf{ScSam}()$
$Y_a \leftarrow \mathsf{ScMul}(g, y_a)$		$Y_b \leftarrow \mathsf{ScMul}(g', y_b)$
	$\xleftarrow{\quad Y_b \quad}$	
	$\xrightarrow{\quad Y_a \quad}$	
$K \leftarrow \mathsf{ScMulVf}(Y_b, y_a)$		$K' \leftarrow \mathsf{ScMulVf}(Y_a, y_b)$
Abort if $K = I_{\mathcal{G}}$		Abort if $K' = I_{\mathcal{G}}$
$ISK \leftarrow \mathsf{H}_2(K \Vert \mathsf{oc}(Y_a, Y_b))$		$ISK' \leftarrow \mathsf{H}_2(K' \Vert \mathsf{oc}(Y_a, Y_b))$
Output ISK		Output ISK'

Deriving the generator for CPace

- Calculate generator by use of a random oracle?
- Hash password and ordered concatenation of party identifiers directly to a group element

$$g \leftarrow \mathsf{H}_{\mathcal{G}}(\mathsf{pw} || \mathsf{oc}(\mathcal{P}, \mathcal{P}'))$$

- Unclear how to construct a random oracle directly hashing to the group!

Deriving the generator for CPace

- CPace is designed for elliptic-curve Diffie-Hellman
  - Public key: coordinates of a point on the curve group $\mathcal{G}$
  - Each coordinate is encoded as a field element from $\mathbb{F}_q$

- Mapping algorithms exist

$$Map2Pt : \mathbb{F}_q \rightarrow \mathcal{G}$$

Deriving the generator for CPace

Calculating the generators for CPace in the real-world:

- First hash inputs to field elements

$$h \leftarrow \mathsf{H}_1(\mathsf{pw} || \mathsf{oc}(P_i, P_j))$$

- then Map field elements to group element

$$g \leftarrow \mathsf{Map2Pt}(h)$$

CPace in the real world

- Use elliptic curves of non-prime order for efficiency
- Use-Single-Coordinate Diffie-Hellman Protocols
- Drop checks for invalid curve attacks
- Drop checks for group membership
- Rely on twist-security of curves
- Allow for non-uniform sampling of scalars

- Various Mapping primitives used for deriving group elements:
  - Elligator2 for curves with one point of order 2
  - SWU, Icart’s map, SSWU for most curves in Short-Weierstrass form
  - SvdW for the general case
- Map once and Map twice constructions

This work

Are all these variants of CPace secure?

Previous results

M. Abdalla and M. Barbosa, https://eprint.iacr.org/2019/1194)

- Requires a RO hashing directly to the group
- Requires a modification of the protocol (password in the final hash)
- Mandates prime-order groups
- Proof does not cover use of single-coordinate public-keys

Previous results

M. Abdalla and M. Barbosa, https://eprint.iacr.org/2019/1194)

- Requires a RO hashing directly to the group
- Requires a modification of the protocol (password in the final hash)
- Mandates prime-order groups
- Proof does not cover use of single-coordinate public-keys

- Required properties for the map remained unclear
- This has been used as an argument for promoting PAKE constructions that are much more complex than CPace

Result 1: No random oracle for hashing to the group!

Requirement for $Map2Pt : \mathbb{F}_q \rightarrow \mathcal{G}$ *probabilistic invertability:*

- Invertability: Algorithm for calculating all pre-images in $\mathbb{F}_q$ of a point g
- Known maximum number of pre-images n_g of a point g.

$$\forall g \in \mathcal{G},\ n_{\max} \geq n_g \geq 0$$

With these properties an inverse mapping algorithm can be defined and used by the simulator in the UC framework.

Properties fulfilled by all of Elligator2, Icart's map, SWU, SvdW.
Proof works for both, map once and map twice constructions.

Result 2: Adaptive security

- Typical issue for Diffie-Hellman-Type protocols: Committment problem
  - Need to simulate public keys of honest parties first without access to secrets
  - After the adaptive corruption event: learn the secrets.
    Need to provide consistent picture.

- Approach used in CPace:
  - Use ephemeral generators in each session
  - Use *probabilistic invertability* properties of Map2Pt for setting up a trapdoor for secret exponent of generator
  - Use trapdoor exponents for adjusting secret scalars

Result 3: Security of implementation variants

- Analyzed the impact of groups of non-prime order
  - Does not impact security of CPace if all secret exponents are chosen to be multiples of the cofactor

- CPace secure on *groups* and on *groups modulo negation*
  - Single-Coordinate Scalar-Multiplication can be used securely

- Formalized Twist-Security for Elliptic-Curve groups: TCDH problem
  - Point verification can be dropped when implementing CPace using single-coordinate scalar multiplication on twist-secure curves (e.g. X25519 and X448)

Btw: Our simulators use assumption libraries!

- Conventional approach: simulation and reduction *separated*
  - First *simulator algorithm* with a set of bad events where the algorithm aborts
  - Second *reduction algorithm* that embeds challenges for an hard problem and solves the problem in case of the bad events

- Our proof approach: simulation and reduction *unified*
  - Embed the assumptions as part of the simulator code by using *assumption libraries*

Assumption definition by using executable code

Assumption fully specified by its corresponding experiment algorithm:

Experiment:

- Generate a random challenge
- Provide all oracles that are made available for the adversary
- Check the adversary’s output for the correct solution

An efficient experiment algorithm is available for any *falsifiable* assumption. The assumption can be uniquely defined by its experiment.

Assumption library example: Strong CDH problem (sCDH)

```
class sCDH:
   # Implementation of the sCDH problem with restricted DDH oracle access

   produce_sCDH_problem_challenge(self):
      self.g = sample_random_generator()
      self.ya = sample_scalar(); self.Ya = self.g^ya
      self.yb = sample_scalar(); self.Yb = self.g^yb
      return (g, self.Ya, self.Yb)

   DDH(self,g,Yab,X,K): # Yab required to be self.Ya or self.Yb for sCDH
      if (g,Yab,X) in [(self.g, self.Ya, self.Yb),
                       (self.g, self.Yb, self.Ya)]:
         if (K == self.Yb^self.ya): abort ("sCDH solution provided")
      elif (g == self.g) and Yab == self.Ya: return K == X^self.ya
      elif (g == self.g) and Yab == self.Yb: return K == X^self.yb
      else return "Not a valid sCDH query but one the for full GAP CDH!";
```

Assumption library example: Strong CDH problem (sCDH)

Generate a fresh CDH problem challenge by sampling 3 generators (g, Ya, Yb)

```
class sCDH:
   # Implementation of the sCDH pro

   produce_sCDH_problem_challenge(self):
      self.g = sample_random_generator()
      self.ya = sample_scalar(); self.Ya = self.g^ya
      self.yb = sample_scalar(); self.Yb = self.g^yb
      return (g, self.Ya, self.Yb)

   DDH(self,g,Yab,X,K): # Yab required to be self.Ya or self.Yb for sCDH
      if (g,Yab,X) in [(self.g, self.Ya, self.Yb),
                       (self.g, self.Yb, self.Ya)]:
         if (K == self.Yb^self.ya): abort ("sCDH solution provided")
      elif (g == self.g) and Yab == self.Ya: return K == X^self.ya
      elif (g == self.g) and Yab == self.Yb: return K == X^self.yb
      else return "Not a valid sCDH query but one the for full GAP CDH!";
```

Assumption library example: Strong CDH problem (sCDH)

```
class sCDH:
```

```
                                                            ...cle access
```

Difference between CDH problem and strong CDH problem sCDH:

Give the adversary access to a restricted DDH oracle where the first two inputs are fixed.

```
     self.yb = sample_scalar(); self.Yb = self.g^yb
     return (g, self.Ya, self.Yb)
```

```
DDH(self,g,Yab,X,K): # Yab required to be self.Ya or self.Yb for sCDH
   if (g,Yab,X) in [(self.g, self.Ya, self.Yb),
                    (self.g, self.Yb, self.Ya)]:
      if (K == self.Yb^self.ya): abort ("sCDH solution provided")
   elif (g == self.g) and Yab == self.Ya: return K == X^self.ya
   elif (g == self.g) and Yab == self.Yb: return K == X^self.yb
   else return "Not a valid sCDH query but one the for full GAP CDH!";
```

Assumption library example: Strong CDH problem (sCDH)

Simulation-based proof strategy using assumption libraries:

- Write a simulator that embeds assumption library objects.
- Embed the challenges produced by the assumption library objects in the simulated protocol execution
- Write the main simulator's code such that it never aborts itself.
- Only permissible abort conditions are aborts in the assumption libraries.
- => "Bad events" coincide with events where correct solution for the challenged problem is provided

```
cl
                                                                        ss

                                                                        H
    if (g,Yab,X) in [(self.g, self.Ya, self.Yb),
                     (self.g, self.Yb, self.Ya)]:
       if (K == self.Yb^self.ya): abort ("sCDH solution provided")
    elif (g == self.g) and Yab == self.Ya: return K == X^self.ya
    elif (g == self.g) and Yab == self.Yb: return K == X^self.yb
    else return "Not a valid sCDH query but one the for full GAP CDH!";
```

Approach for our simulator

- Embed the assumption’s code in the main code body of the simulator
- Don’t let the main code body of the simulator itself abort at any place
- Only allow for abort events within the assumption libraries (which occurs iff a challenge was solved)

Advantage:

- We can re-use the exact same simulator code body and only replace the assumption libraries for studying protocol variants.
- E.g. replace sCDH assumption library with a library for the strong twist-secure TCDH assumption.
- Reduction strategy clearly visible in executable code

Conclusion

- CPace is a fast & secure PAKE
  - Enjoys composability under strong adaptive adversary models
  - Various variants and tweaks for resource-constrained devices don’t impair security

- We formalized reduction arguments by embedding assumption libraries within the simulator’s code
- Assumption library technique works whenever assumptions are *falsifiable*.

- Internet draft: https://datatracker.ietf.org/doc/draft-irtf-cfrg-cpace/
- Full paper: https://eprint.iacr.org/2021/114