Entrusting Your Secrets to an Oblivious PRF

Hugo Krawczyk
Algorand Foundation

Future of PI – EuroS&P 2021

Collaborators: Stas Jarecki, Aggelos Kiayas, Jason Resch, Nitesh Saxena, Maliheh Shirvanian, Jiayu Xu

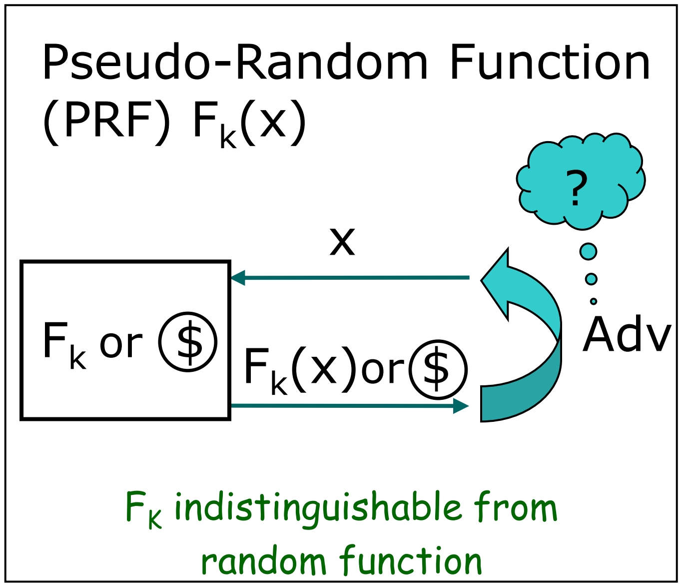
Imagine a world where

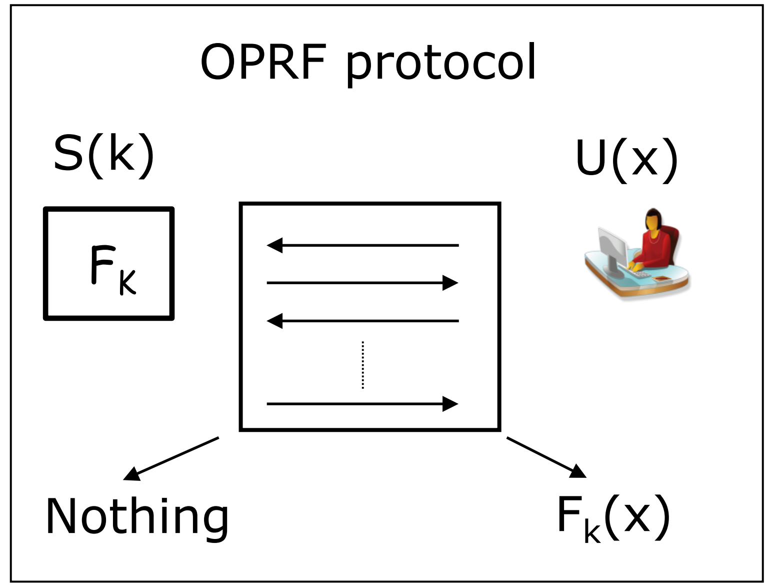
- *Each human being can remember a 256-bit entropy a secret*
- You could encrypt other secrets with it. Such as
  - A private key you could use to prove yourself to others
  - Private credentials, medical and financial information, love letters
  - Crypto wallets where you spent all your savings
  - Encrypt your cloud data without anyone being able to get to it
  - …
- Not a panacea, but what a great foundation a strong secret would be to our digital identity and privacy…

Can we trade a memorizable password for a strong secret, securely?

- Till the day we all carry a chip in our brains…
- … the closest to secrets we can remember are passwords
- So, what are the best ways to trade passwords for strong secrets?
- Enter Oblivious Pseudo-Random Functions (OPRF)

Oblivious PRF (OPRF) [..., FIPR'05,...]





- OPRF: Protocol b/w a user with input x and server with key k; user learns $F_k(x)$ and *nothing else* and server learns *nothing* (neither the input or output of the computation)

Implementation: DH-OPRF

- **PRF**: $F_K(x) = H(x)^k$; H = RO into group of prime order q; key k in Z_q

$$F_k(x) = H'(\quad H(x)^k)$$

Implementation: DH-OPRF

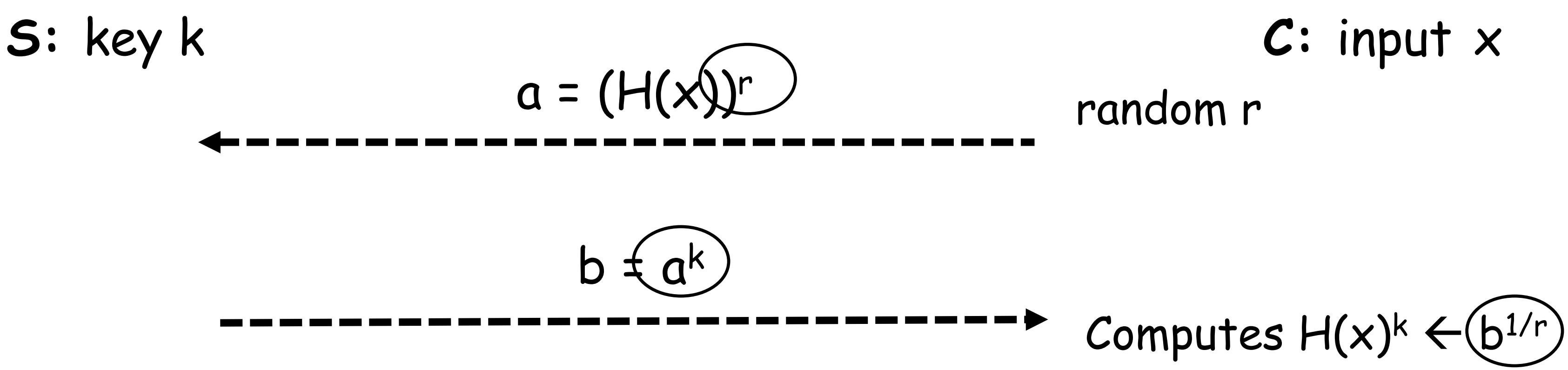
- **PRF**: $F_K(x) = H(x)^k$; H = RO into group of prime order q; key k in Z_q
- Oblivious computation via Blind DH Computation (S has k, C has x)

S: key k		C: input x
	$a = (H(x))^r$ ←------------------------	random r
	$b = a^k$ ------------------------→	Computes $H(x)^k \leftarrow b^{1/r}$

- $b^{1/r} = (a^k)^{1/r} = (((H(x)^r)^k)^{1/r} = (((H(x)^k)^r)^{1/r} = (H(x))^k$
- The blinding factor r works as a one-time encryption key: ***hides*** H(x), x ***and*** $F_K(x)$ ***perfectly from S*** (and from any observer)

DH-OPRF Efficiency

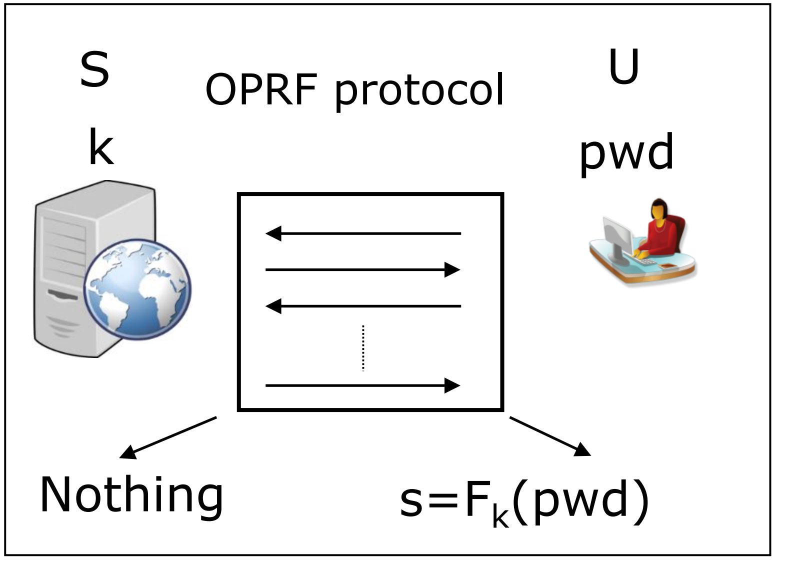
- **PRF**: $F_K(x) = H(x)^k$; input x, key k from 0…q-1
- Oblivious computation via Blind DH Computation (S has k, C has x)



- Cost: 2 messages, 2 exponentiations for C, 1 for S
  - Commodity laptop: > 10,000 exponentiations/second
  - Variant: fixed base exponentiation for C (even faster)

Trading a Password for a Strong Secret

- Server S has an OPRF key k; user U enters its password pwd and gets a strong secret s = $OPRF_K(pwd)$



- No one (including the server) learns *anyhing* about pwd or s
  → a *strong crypto key* for anyone that does not know pwd

Simple, Intuitive and … *Insecure*

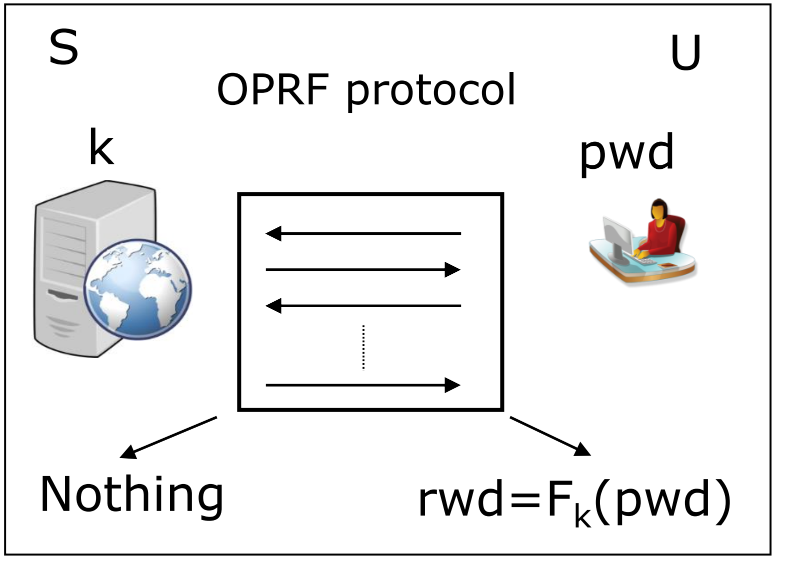
- Assume a setting where given a value s’, one can check that s=s’
  - E.g., use s’ to decrypt a ciphertext encrypted with the user’s key s
- Attacker impersonates S with fake OPRF key k*, creates dictionary D={F_{K*}(pwd): pwd ∈ PwdDict} and for each s’ in D, it checks if s=s’
- Countermeasures (depending on application):
  - “Verifiable OPRF“ : can verify the output from the server via a PK
  - Server stores user-related state (e.g., OPAQUE, PPSS);
  - The application does not allow to verify values for s (e.g., SPHINX)
  - Use a threshold OPRF (more later)

OPRF application to aPAKE

- aPAKE: Asymmetric Password Authenticated Key Exchange
- Password protocol between a client and server; generates a key to protect communication between C and S
  - Common implementation: Password-over-TLS
- aPAKE is a stronger notion: no PKI (except for registration), no exposure of password outside the client machine (incl. to S)
- Only allowed attacks are *unavoidable* ones
  - Online password guessing and offline dict attack upon server compromise
- Note: “Strong aPAKE” (offline attack cannot be pre-computed)

OPRF Application to aPAKE

- User U logs to server S with password pwd: Runs OPRF with S to "exchange" pwd for the OPRF output rwd = $OPRF_K$(pwd)



[FK'00, Boyen'09, JKKX'16, JKX'18]

- rwd is a *strong key* for anyone that does not know pwd (incl. S)
- U uses rwd as a private key in a key exchange (KE) protocol with S

OPAQUE [JKX'18]

- Let KE be a PK-based Authenticated Key Exchange protocol
- Registration (of user U at server S):
  - S chooses fresh OPRF key k and a pair ($priv_S$, pub_S) for protocol KE;
  - U runs OPRF with S on input pwd to learn rwd=F_k(pwd);
    it generates KE keys ($priv_U$, pub_U) and sets Env_U = $AuthEnc_{rwd}$($priv_U$, pub_S)
  - S stores ($priv_S$, pub_U), OPRF key k and Env_U
- Login:
  - U runs F_K(pwd) with S to learn rwd; receives and decrypts Env_U ;
  - U and S run KE with keys ($priv_U$, pub_U, $priv_S$, pub_S)

OPAQUE compiler

OPRF + AuthEnc + AKE → Strong aPAKE

- Idea is simple but subtleties abound
  - KE must satisfy perfect forward secrecy and security against “reverse impersonation” (KCI security)
  - AuthEnc must have a “key committing” property
  - OPRF needs to be collision resistant and secure against adversarially-chosen keys
- Proven UC (strong) aPAKE in RO: non-trivial (minefield!)

OPAQUE with DH-OPRF

- C has pwd; S has OPRF key k and private key $priv_S$;
  S stores $Env_U = AuthEnc_{rwd}(priv_U, pub_S)$ where $rwd = H'(pwd, H(pwd)^k)$
  - C sends $a = (H(pwd))^r$, g^x (random r, x)
  - S replies with $b = a^k$, Env_U , g^y, $AuthKE_S(g^y)$ (random y)
  - C sets $rwd = H'(pwd, b^{1/r})$, decrypts-verifies Env_U , sends $AuthKE_C(g^x)$
- $AuthKE_S(g^y)$, $AuthKE_C(g^x)$ and a session key are computed by C and S according to protocol KE and their corresponding private/public keys
- Example: With KE=HMQV, the session key computation for both C and S is little more than one exponentiation (1.17)
- OPAQUE well suited for TLS 1.3 (with KE = SIGMA)

Summary: OPAQUE Protocol

- Modular/flexible: Can compose with any AKE with KCI and PFS
- Efficient instantiations (e.g., HMQV, 3DH, SIGMA, TLS 1.3)
- Standardization: CFRG and TLS working groups
- Security: Strong aPAKE in the UC model (under ROM/Gap-OneMoreDH)
  - Only unavoidable attacks: *online guessing and exhaustive offline dictionary attacks upon server compromise (no PKI! , pwd never exposed outside client)*
- Extensions:
  - Credential/secret retrieval
  - Multi-server implementation via threshold OPRF (no change on client side)

PPSS: Password Protected Secret Sharing

(password-protected distributed storage)

How to store a secret (and not lose it)

- We want to protect <u>*secrecy*</u> and <u>*availability*</u> of information while remembering a *single* password
  - Single server = Single point of compromise for secrecy (offline dict attacks)
  - Single server = Single point of failure for availability (server gone, secret gone)
  - ➔ Multi-server solution a must.
- Crypto solution: keep the secret encrypted in multiple locations; *secret share the encryption key* in multiple servers
  - Share among n servers, retrieve from t+1 servers (e.g. n=5, t=2)
- Protects availability and secrecy: *available* as long as t+1 available, *secret* as long as no more than t corrupted

Wait, but how do you authenticate to each server for share retrieval?

- Server needs to authenticate the user before delivering a share
- All we have is a user and a password
  - A strong independent password with each server? Not realistic
  - Same (or slight-variant) password for each server? Not good

➔ *Each server as a single point of compromise!*

  - *From one point of compromise to n*. We haven't achieved much, have we?

How to protect a secret with a password

- What we want: *"(n,t)-Password-Protected Secret Sharing (PPSS)"*
  - n servers, t+1 reconstruct the secret, breaking into t servers is useless (even if all t servers' memory leaks: shares, long-term keys, password file, etc.)
  - Only adversary option: Guess the password, try it in an <u>online attack</u>.
- We show a solution based on *(n,t)-Threshold OPRF* : Instead of one server storing an OPRF key K,
  1. n servers, each stores a share k_i of K so that any t+1 can compute the OPRF
  2. but no collusion of t malicious servers can learn anything about K or the input/output of the function on any value
  3. K is *never* reconstructed, shares used to compute OPRF, not to compute K

PPSS Solution = Threshold OPRF

- n servers share a T-OPRF $F_k(x)$
- U's secret defined as $s=F_k(pwd)$
- To retrieve s, U runs T-OPRF with any t+1 servers
- More precisely (crucial detail):
  - U's secret defined as $s'=H_1(s)$
  - In addition to k_i , servers store c= $H_2(s)$, which they send to U together with OPRF response; if not t+1 servers send c=H(s,2), U aborts
- Security bonus: Even if t+1 servers compromised, a full exhaustive offline attack still needed to find password!

Threshold DH-OPRF (n-out-of-n)

- Single server solution: $F_k(x) = (H(x))^k$ (H' omitted for simplicity)
- Multi-server solution: Server S_i has share k_i, $k = k_1 + k_2 + \cdots + k_n$
  - $F_k(x) = (H(x))^{k_1} \cdot (H(x))^{k_2} \cdot \cdots \cdot (H(x))^{k_n} = (H(x))^{\sum k_i}$
- To compute $F_k(x)$ obliviously, U sends *same* $a = (H(x))^r$ to each server; S_i returns $b_i = \mathrm{a}^{k_i}$; U sets $F_k(x) = (\prod b_i)^{1/r}$
- Efficiency: 2 exp's for client (indep of n), 1 per server, 1 round
- Key k is never reconstructed: "function sharing" vs "secret sharing"

Threshold DH-OPRF (t-out-of-n)

- *t-out-of-n* threshold DH-OPRF: Each server S_i has share k_i
- $F_k(x)$ computed from any set of *t* servers $S_{i1},\ldots, S_{it}$
  - $F_k(x) = (H(x))^{\lambda_{i1}k_{i1}} \cdot (H(x))^{\lambda_{i2}k_{i2}} \cdot \cdots \cdot (H(x))^{\lambda_{it}k_{it}}$
  - λ_{ij} is a Lagrange interpolation coefficient ("Shamir in the exponent")
- As before: key k is never reconstructed
  - Not even during generation/sharing: Distributed key generation
- Note: share recovery and proactive share refreshing

PPSS Efficiency (same as Threshold OPRF)

- Computation:
  - Single exponentiation for each server
  - Only two exponentiations *in total* for the client (*independent* of t and n)
- Communication: Single parallel message from user to t+1 servers, one msg back from each server. No inter-server communication.
- *No assumed PKI or secure channels* (other than for initialization)
- Any t, n ($t \leq n$)
- Robustness/Verifiability: V-OPRF (e.g via NIZK or interactive verif.)

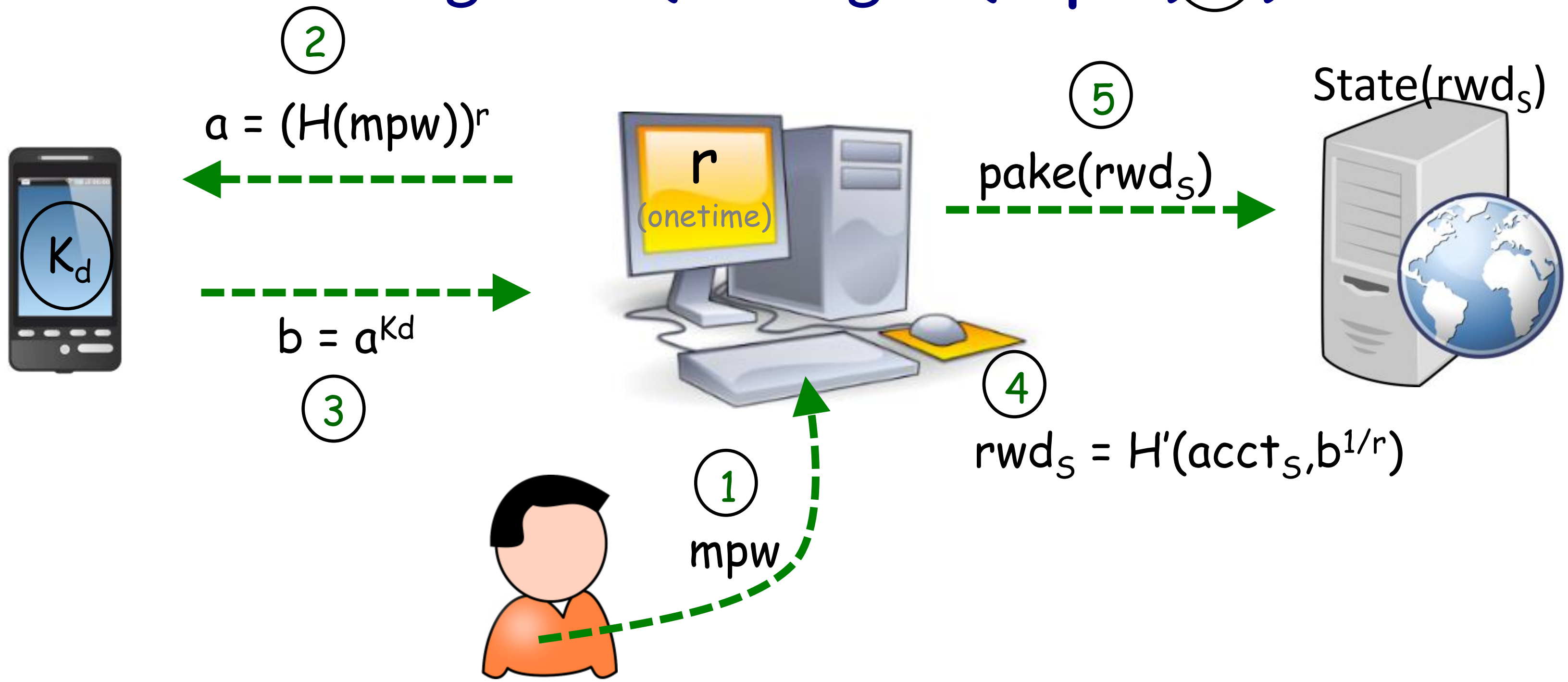
SPHINX: Magic Password Manager

- What if we could remember really strong independent passwords, **securely**?
- We would have solved online and offline dictionary attacks!
- But what about breaking the password manager itself?

Password Manager

- OPAQUE and PPSS do not solve the "entropy problem"
- Password manager: Stores passwords for user (browsers, lastpass, etc)
  - If it uses random passwords, online and offline attacks are resolved
  - A list of all user's passwords encrypted under a master password
    → Can run offline dictionary attack given the user's encrypted list
- Wanted: Attacker with full control of the manager and storage:
  - Learns *nothing* about stored passwords (even during password registration)
  - Even with full control of the manager, a dictionary attack on master password requires an *online attack per guess*

$$rwd_S = H'(acct_S, H(mpw)^{K_d})$$



- rwd is ~random hence secure against online guessing and offline attacks on server
- mpw and rwd_S independent of device storage (K_d): secure upon device compromise (even in this case, an mpw dictionary attack needs online verification with server)
- master mpw is perfectly hidden on the wire and from device: secure against network attackers and fully malicious device

SPHINX Security

- Network attacks: *Unconditional* security device-client communication
- Online dictionary attacks: *Infeasible* (random and independent rwd's)
- Offline dictionary attacks: *Infeasible* (random rwd)
  - Offline against master pwd ONLY if *server AND device compromised*
- Device compromise: *Unconditional* secure pwd/rwd (online-only attack)
- Password leakage: Partial defense (rwd useless in another server, master pwd useless w/o device, url hashing prevents phishing)

SPHINX: A password Store that Perfectly Hides from Itself, No Xaggeration

Final Remarks

Passwords, can't live with them, can't live without them

- If you are one of those that believe passwords are about to disappear, this work is not for you
  - I was in that camp 25 years ago... life taught me I was wrong
  - Deployment, convenience, portability , familiarity, inertia, …
- Hardware-based solutions need to be pursued, but password security cannot be disregarded, it still fuels most of our security
- Goal: Password visibility at client machine as the only vulnerability
  - OPAQUE with T-OPRF addresses offline attacks upon server compromise
  - OPAQUE (or any aPAKE) with SPHINX manager eliminates online and offline attacks too (leaves client machine as only attack target!)

Thanks!

- OPAQUE ia.cr/2018/163
- OPAQUE Internet draft
  https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque
- PPSS ia.cr/2017/363
- SPHINX ia.cr/2018/695
- 2-factor authentication eprint 2018/033 (not presented)