ZK-SCHNAPS: ENFORCING ARBITRARY PASSWORD POLICIES IN A ZERO-KNOWLEDGE PASSWORD PROTOCOL

MATTHIJS ROELINK

31-10-2022

SUPERVISORS:

DR. M.H. EVERTS
PROF.DR.IR. R.M. VAN RIJSWIJK-DEIJ

UNIVERSITY
OF TWENTE.

OVERVIEW

1. Introduction and problem statement
2. Building blocks
   a. Homomorphic encryption
   b. Zero-knowledge proofs
   c. zk-SNARKs
   d. SAVER
3. zk-SCHNAPS
4. Evaluation
5. Demo
6. Discussion and future work
7. Conclusion
8. Questions

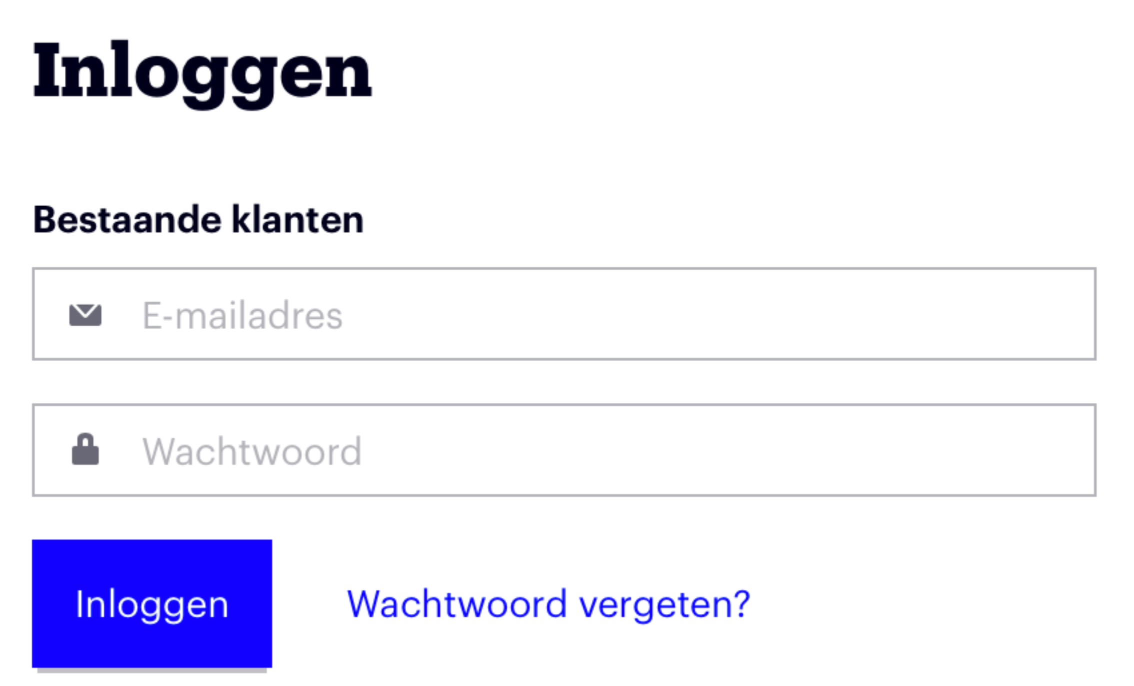
UNIVERSITY
OF TWENTE.

01 INTRODUCTION AND PROBLEM STATEMENT

01 INTRODUCTION AND PROBLEM STATEMENT

INTRODUCTION

- Subject: password authentication
- Registration and login with a username and password



CURRENT SITUATION

Client		Server

Registration

Client: Choose valid username u and password p such that $P(p) = P_1(p) \wedge P_2(p) \wedge \dots \wedge P_n(p)$ evaluates to `true`, where P_i is a single password policy.

Send u and p →

Server: Check that $P(p)$ evaluates to `true`.

Obtain $h = H(p{:}s{:}t)$, where H is a hash function suitable for password hashing, s is a randomly generated n-byte salt, t is a constant m-byte pepper and $:$ represents concatenation.

Store u, h and s.

← $\{$`valid, invalid`$\}$

Login

Client: Enter username u' and password p'.

Send u' and p' →

Server: Look up h and s corresponding to u'.

Compute $h' = H(p'{:}s{:}t)$ and compare h and h'.

← $\{$`valid, invalid`$\}$

PROBLEM

- The server needs to be trusted with:
  - not misusing the password
  - securely storing the password
- Solution: zero-knowledge password protocols
- New problem: server cannot enforce password policies
- Partial solution: Zero-Knowledge Password Policy Checks
  - But only supports very limited password policies
  - Leaks the password length
- We would like a scheme that
  - does not reveal the password to a server
  - but allows enforcing arbitrary password policies

SOLUTION

- zk-SCHNAPS:
  - **z**ero-
  - **k**nowledge
  - -
  - **S**ecure
  - **C**ommitment-based
  - **H**omomorphic
  - **N**on-interactive
  - **A**uthentication with
  - **P**asswords using
  - **S**NARKS
- Uses a zk-SNARK to prove compliance to the password policies

02 BUILDING BLOCKS

HOMOMORPHIC ENCRYPTION

A homomorphic encryption scheme is an encryption scheme with operations $\otimes$ *and* $\oplus$ *such that*

$$E(m_1) \otimes E(m_2) = E(m_1 \oplus m_2)$$

for all plaintexts m_1 *and* m_2*.*

Example - additive homomorphic encryption:
$E(2) \cdot E(5) = E(2 + 5) = E(7)$

ZERO-KNOWLEDGE PROOFS

- Proving knowledge of something without revealing it
- Typical use case: age verification

ZK-SNARKS (1)

- Class of **zero-knowledge proofs**

- Acronym:
  - **z**ero-**k**nowledge: no additional information can be learnt
  - **S**uccinct: small proof size and verification time
  - **N**on-interactive: no interaction required between the prover and verifier
  - **A**rgument of **K**nowledge: the prover can convince the verifier without revealing the secret

- Basic idea: proof of a function F with (private) inputs x and output $y = F(x)$.

ZK-SNARKS (2)

Prover		Verifier
	Setup ⟷	
Create proof π_y of a function F with (secret) inputs x and output $y = F(x)$.		
	Send π_y and y →	
		Verify y using π_y.

SAVER

- Problem: encrypting values in a zk-SNARK
- Traditionally: perform encryption in circuit
- SAVER: **S**NARK-friendly, **A**dditive-homomorphic and **V**erifiable **E**ncryption and decryption with **R**erandomization
- Link encryption to zk-SNARK proof
- Additively homomorphic: $E(m_1) \cdot E(m_2) = E(m_1 + m_2)$

03 ZK-SCHNAPS

MAIN IDEA

- Three phases:
  - Registration
  - Login
  - Change password

- Use a zk-SNARK to prove compliance to the password policies

- Combine the zk-SNARK proof with SAVER to yield an encryption of the password hash

- Compare passwords by combing them using the homomorphic property of SAVER

ENCODING PASSWORDS AS INPUT OF A ZK-SNARK

- zk-SNARKs operate over a field $\mathbb{F}_p$, but a password is a variable-length string
- A password should thus be mapped to an element $e \in \mathbb{F}_p$
- Two steps:
  - Map each character c_i of the password to an element $e_i \in \mathbb{Z}_b$ for a base b
  - Aggregate each e_i into a single element $e \in \mathbb{F}_p$:

$$e = \sum_{i=0}^{k-1} e_i \cdot b^i$$

ENCODING PASSWORD POLICIES IN A ZK-SNARK (1)

- A valid proof can be created if and only if the password complies to the password policies

- Example policies:
    - Minimum password length
    - Minimum number of characters from a subset
    - Password not in blocklist
    - Substring of password not in blocklist

ENCODING PASSWORD POLICIES IN A ZK-SNARK (2) - PASSWORD NOT IN BLOCKLIST

- Naive solution: embed blocklist in zk-SNARK and iterate through it
- Problem: large password blocklist results in a large circuit size
- Solution: store passwords in an AMQ-Filter (xor filter)
- Filter is encoded for space-efficiency

PROTOCOL - SETUP

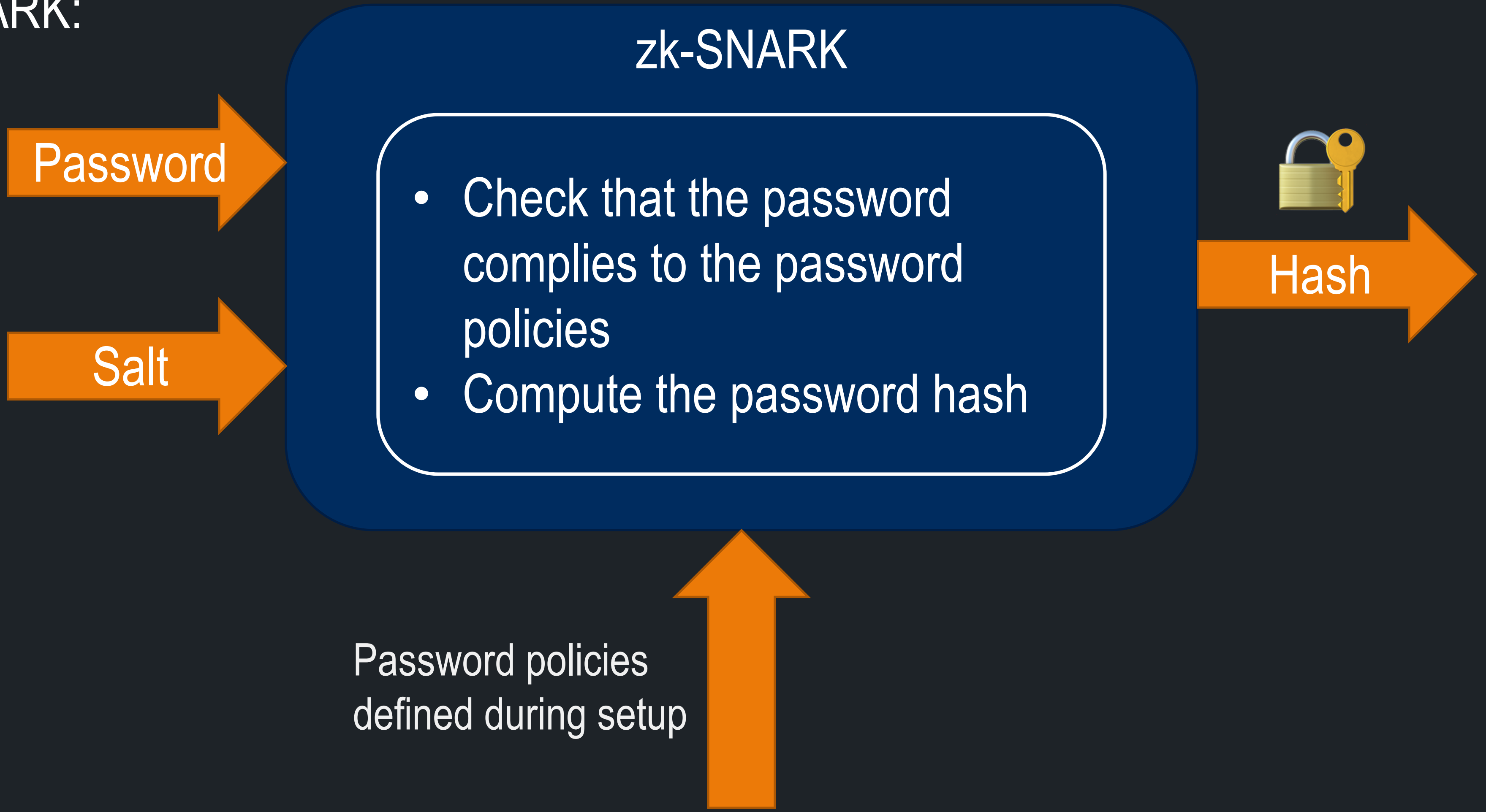
- Performed by server
- Two setups:
  - SAVER setup
  - SAVER key generation

PROTOCOL - REGISTRATION (1)

Client		Server
Choose valid username $\hat{u}$ and password $\hat{p}$.		
$\hat{s} \xleftarrow{\$} \mathbb{Z}_p$		
$(\pi, \mathcal{CT}) \leftarrow \texttt{Enc}_{\texttt{registration}}(CRS, \hat{p}, \hat{s})$		
	Send $\hat{u}, \pi, \mathcal{CT}, \hat{s}$ $\longrightarrow$	
		$\texttt{Verify\_Enc}_{\texttt{registration}}(CRS, PK, \pi, \mathcal{CT}, \hat{s})$
		Store $\hat{u}$, $\hat{s}$ and $\mathcal{CT}$.
	$\longleftarrow$ $\{\texttt{valid, invalid}\}$	

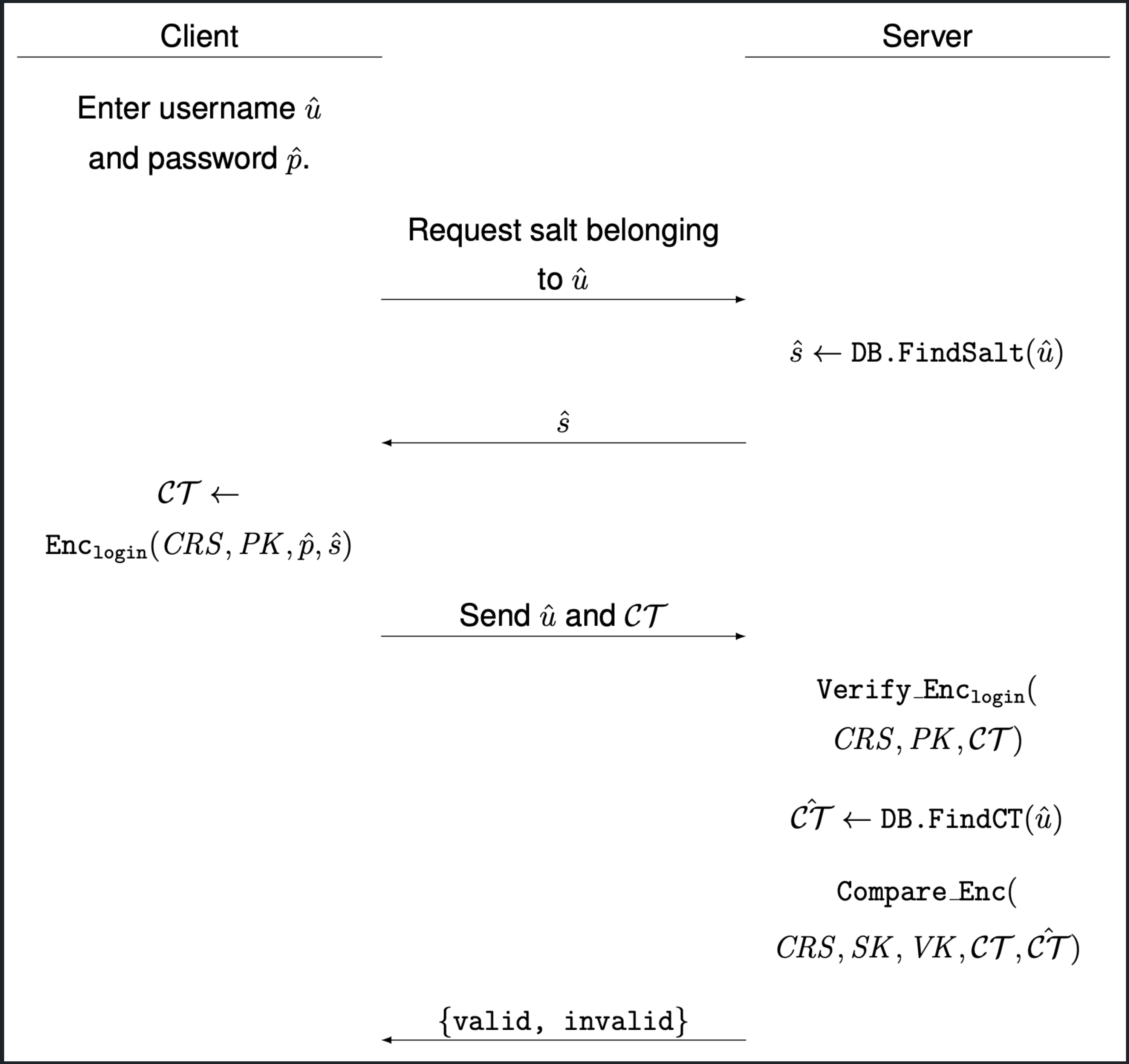
PROTOCOL - REGISTRATION (2)

- zk-SNARK:



PROTOCOL - LOGIN (1)

- Password hash is locally computed



PROTOCOL - LOGIN (2)

- Password comparison can be achieved using the homomorphic property of SAVER
- Two ciphertexts $CT = Enc(\hat{h})$ and $CT' = Enc(\hat{h}')$
- $CT'' = \frac{CT}{CT'} = \frac{Enc(\hat{h})}{Enc(\hat{h}')} = Enc(\hat{h} - \hat{h}')$
- SAVER's decryption yields g^x for an encryption $Enc(x)$ and some base g
- If $\hat{h} = \hat{h}'$, then $\hat{h} - \hat{h}' = 0$ and decryption will result in $\mathrm{g}^0 = 1$

PROTOCOL - LOGIN (3)

- Problem: adversary can use the stored password hash encryption to log in
- Solution: add a zero-knowledge proof φ proving knowledge of r and $\hat{h}$ in $\mathrm{X}_1^r \mathrm{G}_1^{\hat{h}}$
- Sigma protocol made non-interactive using the Fiat-Shamir heuristic:

$$\varphi = (\varphi_{Co}, \varphi_{Ch}, \varphi_{Re})$$

PROTOCOL - CHANGE PASSWORD

- Combination of registration and login phase

PROTECTING AGAINST REPLAY-ATTACKS

- Problem: if an adversary gets hold of a login encryption, it can reuse it
- Solution: store commitment of φ

04 EVALUATION

IMPLEMENTATION

- Extended *snarkjs* library to support subset of SAVER's functions
- Created *schnapsjs*, which implements the zk-SCHAPS protocol functions
- Created Rust program to create and encode password blocklists
- Created demo application, showcasing real-world use of *schnapsjs*

PERFORMANCE

- Most zk-SCHNAPS functions under 1 second
- Creating the registration proof is practical, but time depends on the implemented password policies
- Creating and using large password blocklists is practical as well

Function	Scenario	Time (s)
REGISTER.CREATEPROOF	A	1.987
	B	2.150
	C	2.918
	D	4.345
	E	2.481
	F	4.823

05 DEMO

06 DISCUSSION AND FUTURE WORK

DISCUSSION AND FUTURE WORK

- Password hash function
  - Ideally: slow and memory-hard
  - Not possible yet in a zk-SNARK
  - Future work:
    - SNARK-friendly hash function suitable for passwords
    - Modifying SAVER to prevent decryption
- Proving speed
- Fetching salts
  - Exposes which usernames are taken
  - Solution: return HMAC of the username if the username is unknown

07 CONCLUSION

CONCLUSION

- zk-SCHNAPS: zero-knowledge Secure Commitment-based Homomorphic Non-interactive Authentication with Passwords using SNARKs
- Supports arbitrary password policies
- Uses a zk-SNARK to enforce password policies, combined with SAVER
- Practical performance

08 **QUESTIONS**