RSAConference2020

San Francisco | February 24 – 28 | Moscone Center

HUMAN ELEMENT

SESSION ID: **CRYP-W02**

Tickets, Please!

Ticket Mediated Password Strengthening

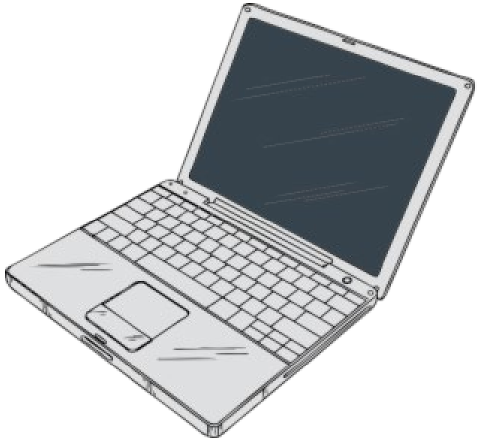
John Kelsey[1,2], Dana Dachman-Soled[3], Meltem Sönmez Turan[1], Shweta Mishra[1,4]

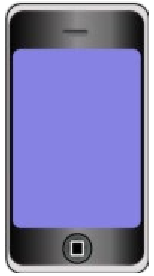
[1]National Institute of Standards and Technology, Gaithersburg MD, USA
[2]Department of Electrical Engineering, ESAT/COSIC, KU Leuven, Belgium
[3]University of Maryland, College Park MD, USA
[4]Department of Computer Science & Engineering Shiv Nadar University, Greater Noida, India

Overview and Background

General Problem: Accessing Local Encrypted Data

- Encrypted data is on my device (laptop, phone, etc.)
  - Probably also extra information: salt, check values, etc.





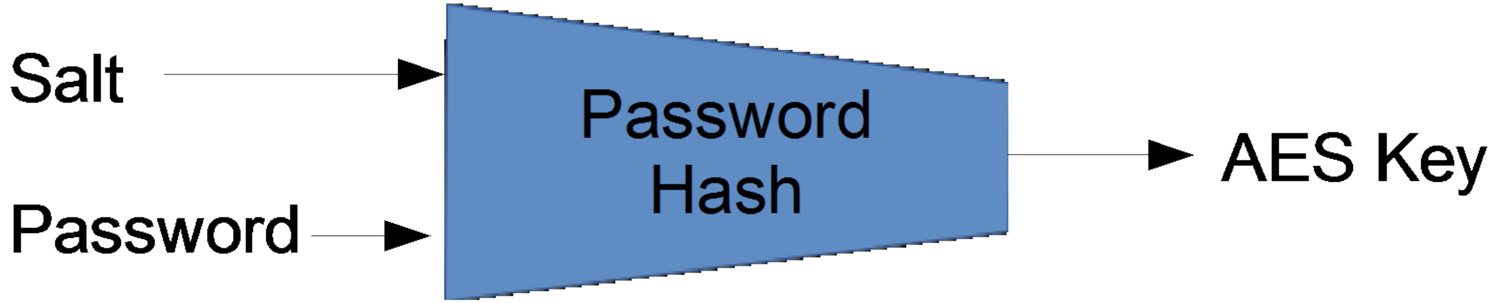
- Only I should be able to unlock it.
  - In practice, this means using a password.

Right password $\Longrightarrow$ unlock the data

Wrong password $\Longrightarrow$ fail

Usual Approach: *Password-Based Key Derivation*

- I have a **password**—need to turn it into an **encryption key**.



- Applications:
  - Disk encryption (laptop)
  - Device encryption (phone, tablet)
  - File encryption (anything)
  - Bitcoin private keys
  - Other cryptographic keys

What Goes Wrong: *Password Guessing Attacks*

Suppose someone steals my device! Can they get my files?

- Online Attack: (User authentication)
  - Each password guess goes through some trusted entity
  - Limit on guesses = how many they will check
  - Easy to rate-limit guesses or lock accounts
- **Offline Attack: (Password-based key derivation)**
  - Attacker moves attack to his own machines
  - Limit on guesses = limit on processors * speed of guessing
  - No way to rate-limit guesses or lock accounts

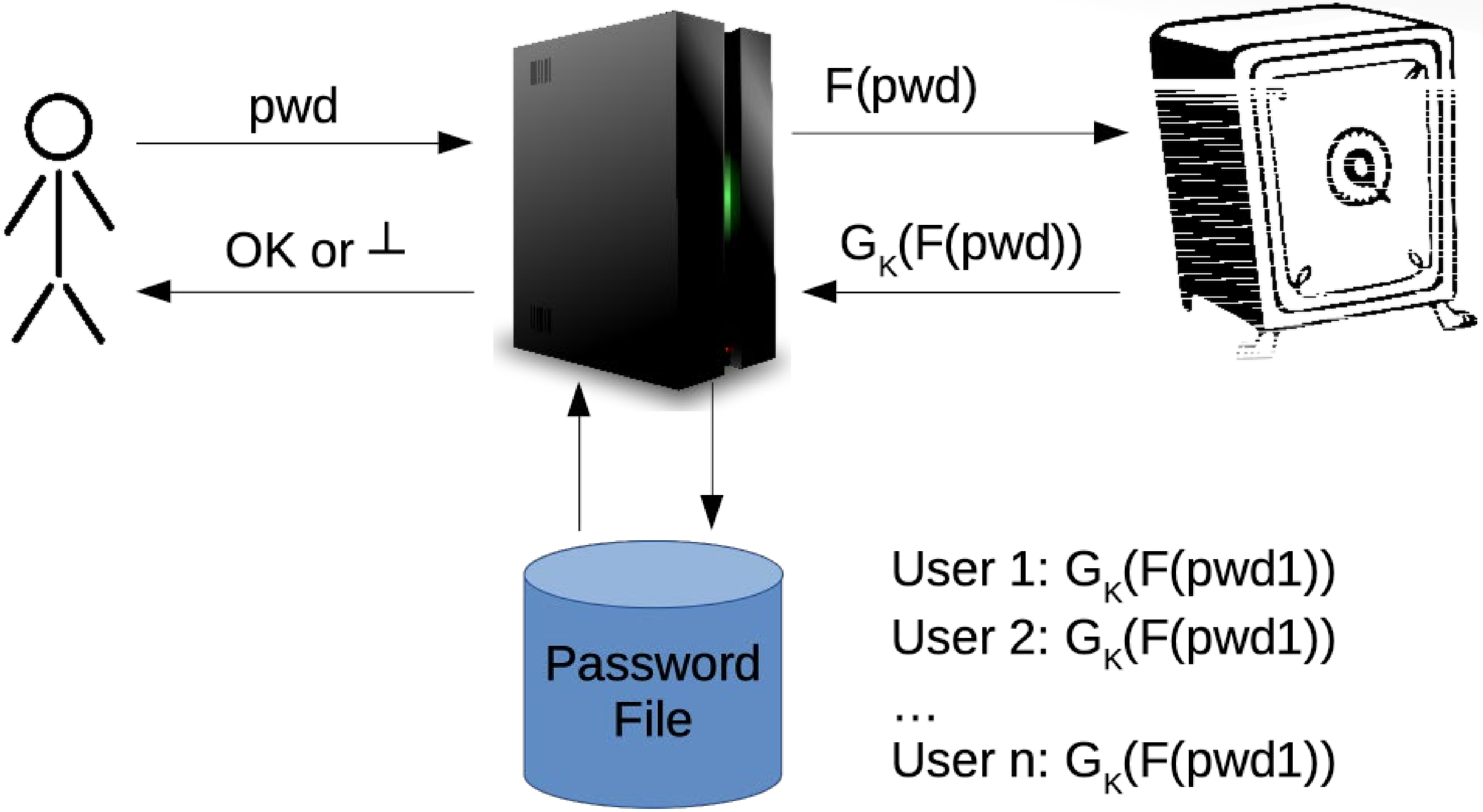
  Same as password guessing after stealing a password file

Potential Solutions

Mostly targeted at logging in, not deriving keys.

- PAKE schemes
  - User and Server establish a shared key from a shared password.
- Password-protected secret sharing
  - User splits secret into shares, gives to many different servers.
  - Password is used along with shares to reconstruct secret.
- **Password strengthening**
  - Use a hardened backend machine to add security

Password Strengthening



- **User:**
  pwd -> Server
- **Server:**
  F(pwd) -> Backend
- **Backend:**
  G(F(pwd)) -> Server
- **Server:**
  Check pwd file
  OK or ⊥ -> User

RSAConference2020

TMPS

Ticket Mediated Password Strengthening

TMPS: Elevator Pitch

- Involve server in password-based key derivation.
  - Prevents offline attack, but requires being online to unlock files.
- Interact with server to get ***tickets***.
- Tickets
  - Entitles me to help from server with **one specific computation.**
  - Server will not accept same ticket twice
  - Result: One ticket = one password guess
- Later: Use tickets to unlock my payload key **K***.
  - Have to interact with server to unlock.
- Steal my laptop with 100 tickets on it
  - You can try 100 guesses for my password
  - After that, no way to unlock my files

The Big Idea

- User device has encrypted data and tickets.
  - Tickets can't be used without help of Server.
  - Each ticket bound to ***specific password and payload key***.
- To decrypt data, user device uses password + ticket
  - Interact with Server to decrypt data
  - Server won’t allow ticket to be reused
  - Server learns nothing about password, key, or who’s using ticket.

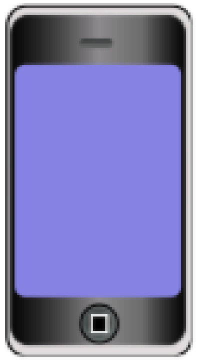
TMPS: The Protocols

In order to make a TMPS scheme work, we need:

- Setup
  - Server establishes its signing and encryption keys.
- **REQUEST**
  - User starts with password **P** and key **K***
  - User ends with t new tickets bound to **(P, K*)**
- **UNLOCK**
  - User starts with password **P’** and a ticket.
  - User interacts with Server.
  - User recovers **K*** **only** if **P’** = **P**, and ticket valid.

REQUEST

- User device must know:
  - **K*** (payload key)
  - **P** (password)
- User forgets **B, C, D** at end.



- Get password **P** from user
- Gen random **S**,**B**
- **E** = Encrypt($\mathbf{PK_s}$,**B**)

Blinded version of **E** →

- Do blind sig

← Blind signature on **E**

- Unblind signature to get **F**
- **C** = Password hash(**S**,**P**)
- **D** = HMAC(**B**,**C**)
- **Z** = Verifiable Enc(**D**,**K***)

- Ticket = (**S**,**E**,**F**,**Z**)

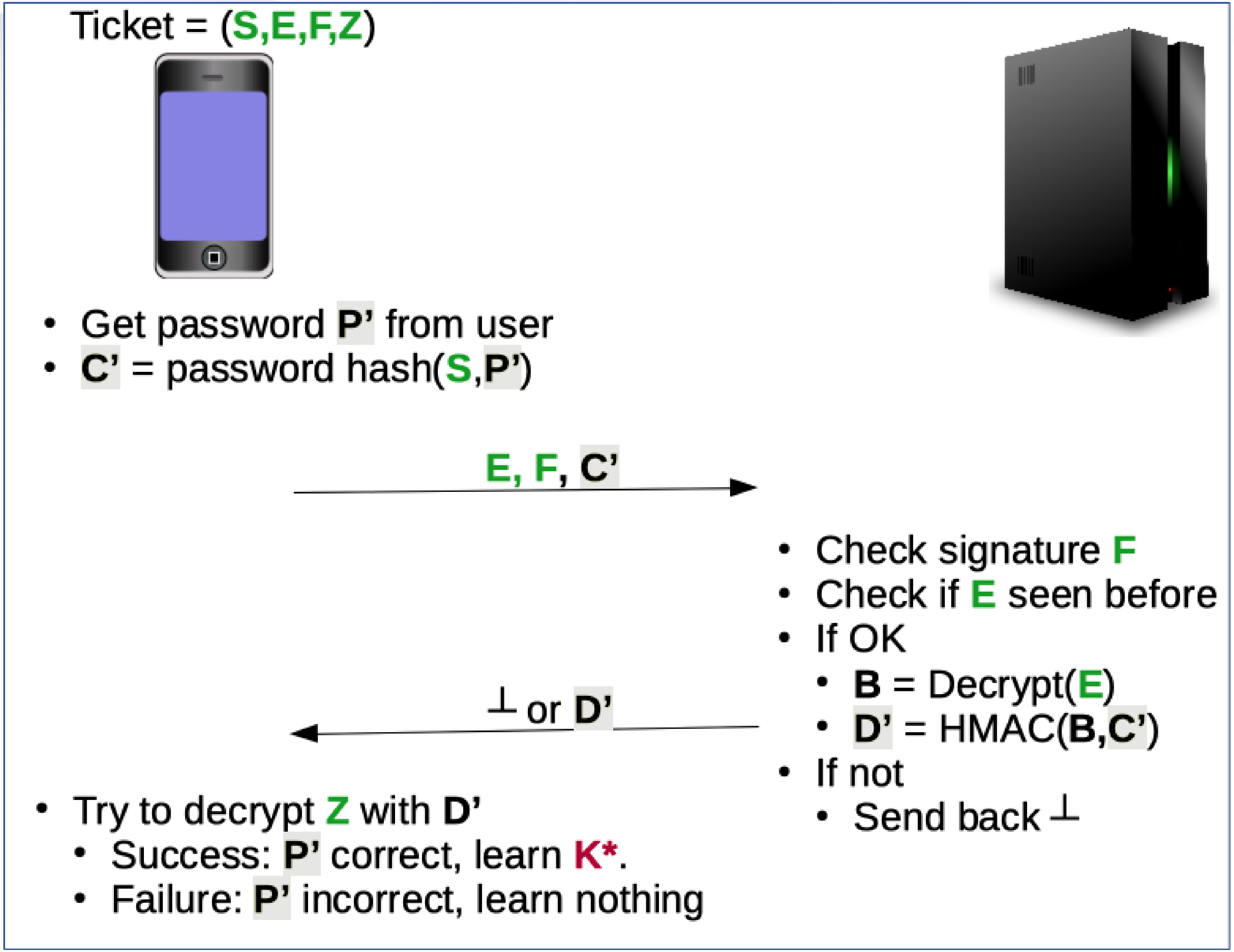
What does a ticket look like?

*Ticket is **S,E,F,Z**.*

- **S** = random salt (different for each ticket)
  - So password hashes sent to server all look different!
- **E** = Secret value **B** encrypted under Server's public key
  - **B** is also different for each ticket
- **F** = blind signature on **E**
  - So Server can't link tickets with users
- **Z** = Verifiable encryption of **K*** under **D**
  - Reminder: **D** is function of salt, password, and **B**
  - Decrypting verifies correctness of password

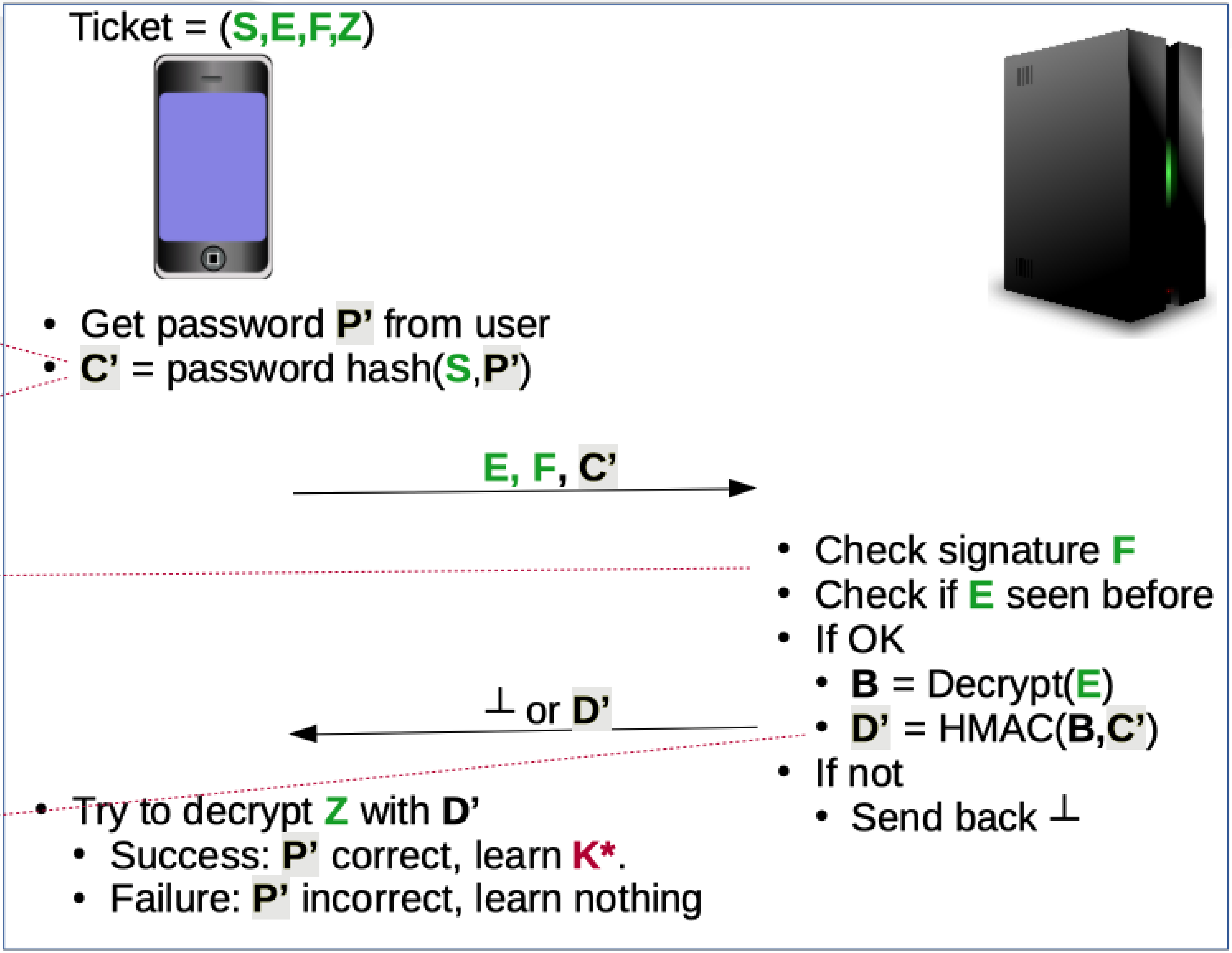
UNLOCK

- Start with ticket and password P’.
- Expend one ticket to test a password guess.



UNLOCK security

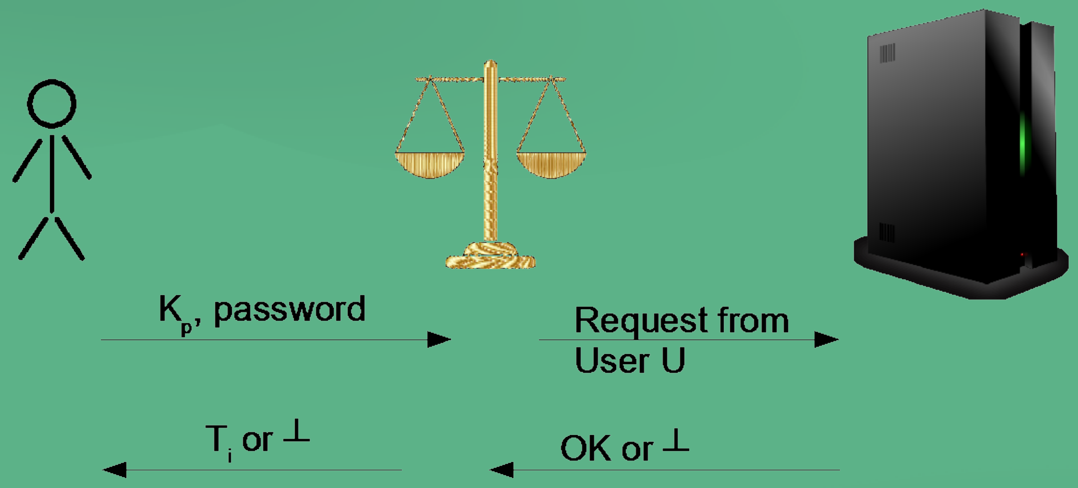
1. Random **S** for each ticket: **C’** different for each ticket.
2. Wrong **P’** means wrong **C’**.
3. Repeated or invalid tickets rejected.
4. Wrong C’ -> wrong D’ -> failed decryption



Getting tickets, limiting guesses

- Can only REQUEST new tickets when you know **P** and **K***
  - At device setup, we know both
  - Later, we use a ticket to UNLOCK **K***
    - Then we can run REQUEST as many times as we like!
- Trick for limiting attacker to 10 guesses
  1. REQUEST lots of tickets (say 1000).
  2. Use K* to derive an encryption key K_T.
  3. Encrypt all but 10 tickets with K_T.
  4. Each time we UNLOCK **K***, derive K_T and decrypt tickets
     - *Till we have 10 left again.*

RSAConference2020



Security and Performance

We took a mixed approach to proving security

- Started with a lot of informal analysis
  - Trying to break it.
- Defined ideal functionality
  - UC Model proof: ideal functionality indistinguishable from our protocols.
- Game-based proofs built on top of UC model proofs
  - Show that ideal functionality actually guarantees our security goals.
  - Give an intuitive definition of what security we achieve.
- Example:

Given t tickets and N possible passwords, attacker unlocks **K*** with prob

$$t/N + \epsilon$$

See paper for details

Memory Requirements

Assuming: RSA with 3072-bit keys, 10 tickets per user per day

- User Device
  - Each ticket takes <1 KiB
  - One year’s supply about 4 MiB
  - This will fit easily on a phone
- Server
  - Need to store/check list of used tickets.
  - Each used ticket needs 16 bytes of storage.
  - 1000 users, one year's worth of tickets: 64 MiB.
  - This will fit in a hash table in RAM.

Computing Requirements: Experimental Results

**We did a minimal Python implementation with no optimizations.*

REQUEST:

Password hash, RSA encryption, blind/unblind signature

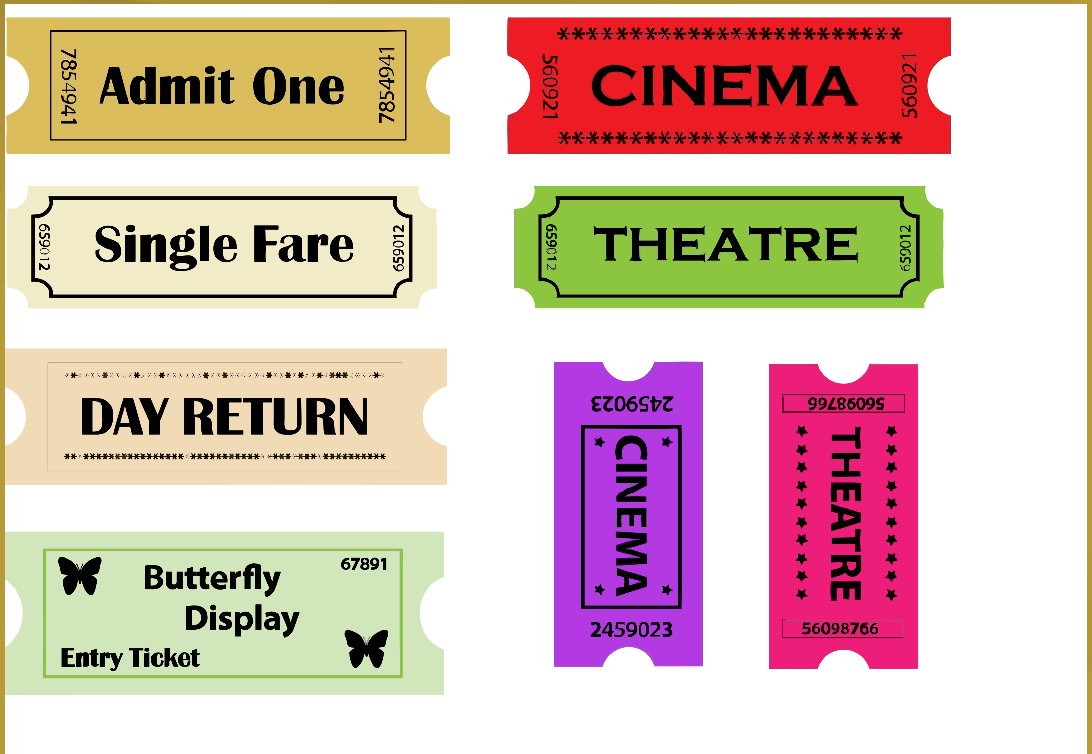
- Our implementation: REQUEST 100 tickets:
  - User: 0.7 seconds
  - Server: 7.6 seconds*

UNLOCK:

Password hash, RSA decryption, verify signature

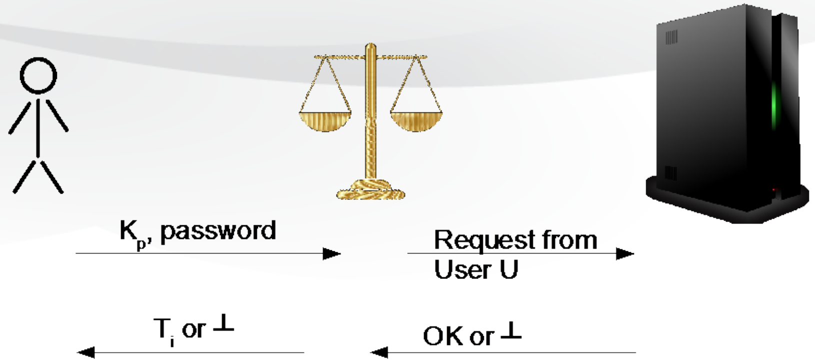
- Our implementation:UNLOCK 1 ticket:
  - User: : 0.0049 seconds
  - Server: 0.002 seconds

RSAConference2020



Wrapup

Applying (1)



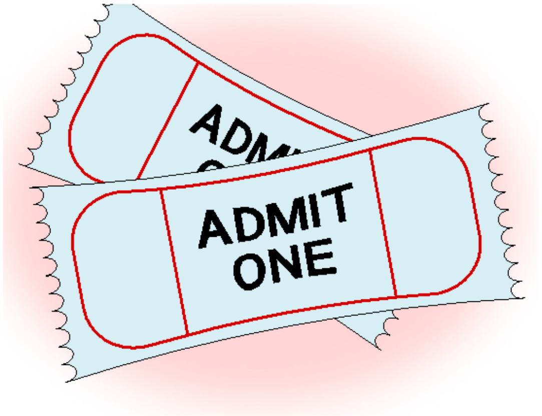
- We introduced TMPS protocol

  Server-assisted local key derivation
  - Someone steals your device = not such a big problem
  - …but you can only unlock your device if you're online

- Many variants described in the paper
  - Offline access (with a security cost)
  - Group signatures instead of blind signatures
  - Proof of work instead of blind signatures

Applying (2)

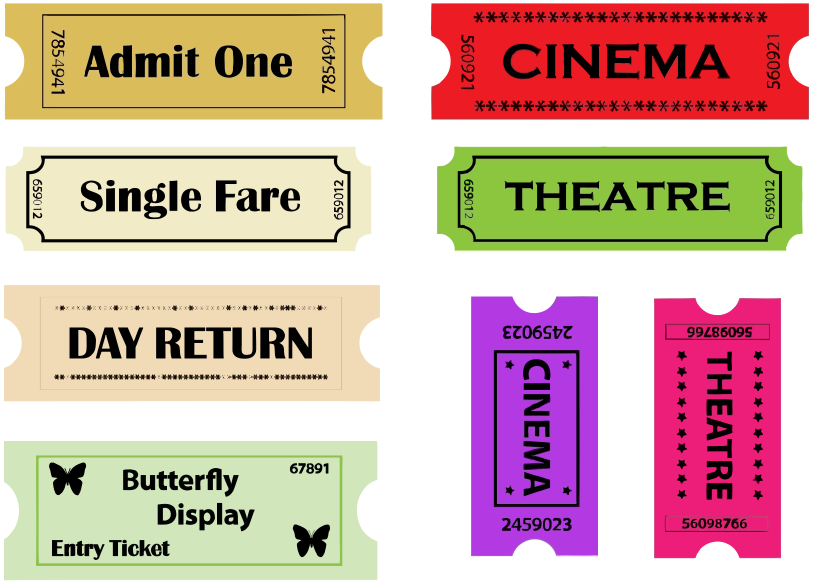
- We have introduced the idea of *tickets*
  - Allow a limited number of cryptographic operations
  - Preserve user privacy
  - Limit access to authorized users



- Tickets seem like a generally useful tool

  Where else could we use them?
  - Enforcing limits on DB queries with differential privacy?
  - Preventing reuse of hash-based signatures?
  - Other stuff?

Questions?


Admit One
7854941
CINEMA
560921
Single Fare
659012
THEATRE
659012
DAY RETURN
2459023
CINEMA
2459023
56098766
THEATRE
56098766
Butterfly Display
67891
Entry Ticket

RSA Conference2020

Extra Slides

TMPS: Security Goals

- User needs Server and tickets to do anything:
  - Test password guess
  - Learn K*.
  - So when attacker steals my laptop, he can’t do offline attack.
- Server will only help with a valid ticket.
  - Server won't allow reuse of tickets.
  - Can't generate new tickets to help with unknown password/ticket.
- Server learns nothing about:
  - Password **P**
  - Whether **P** right or wrong
  - **K***
  - Which user unlocking key

Main TMPS Protocol: Algorithms and Requirements

- Algorithms we need:
  - Public key encryption
  - Blind signatures*
  - Password hash
  - HMAC

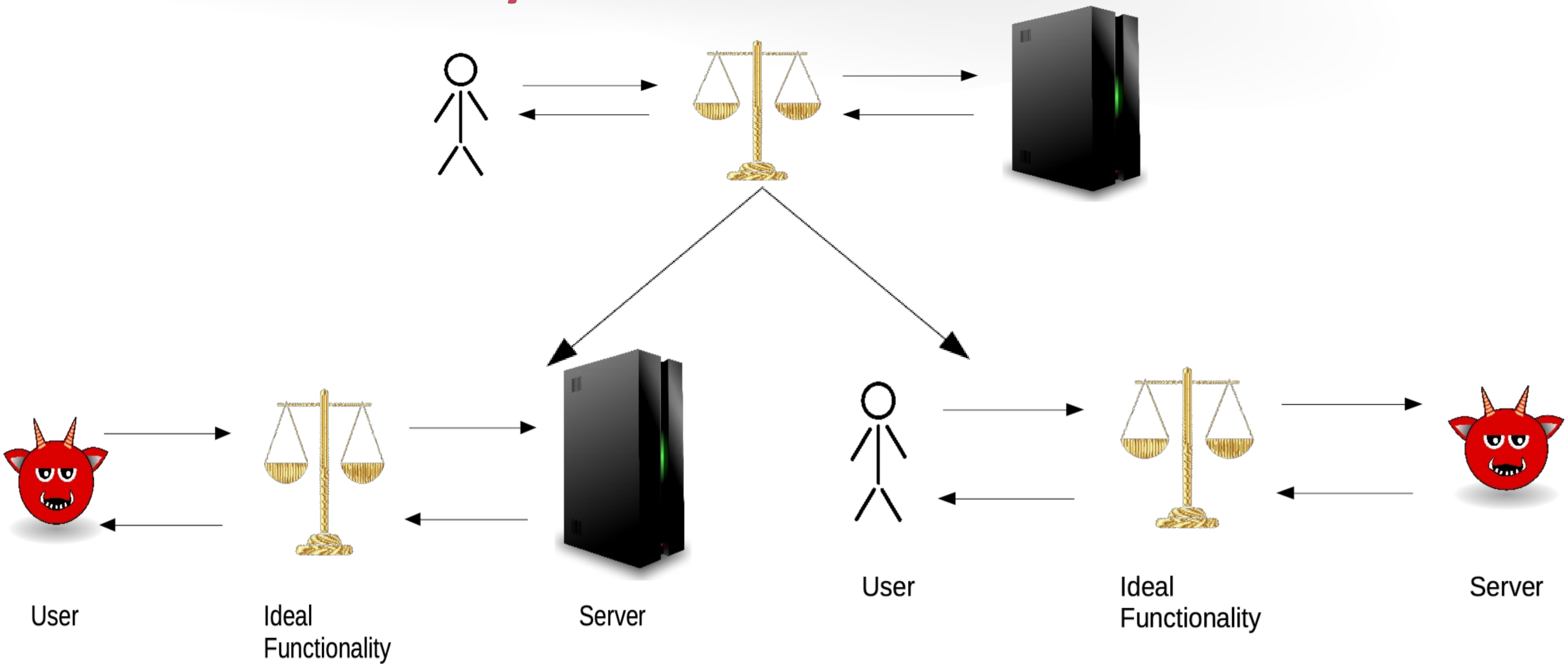
- REQUEST and UNLOCK require interaction with server
- Server stores hashes of all previously-used tickets

Unlock: Why does this work?

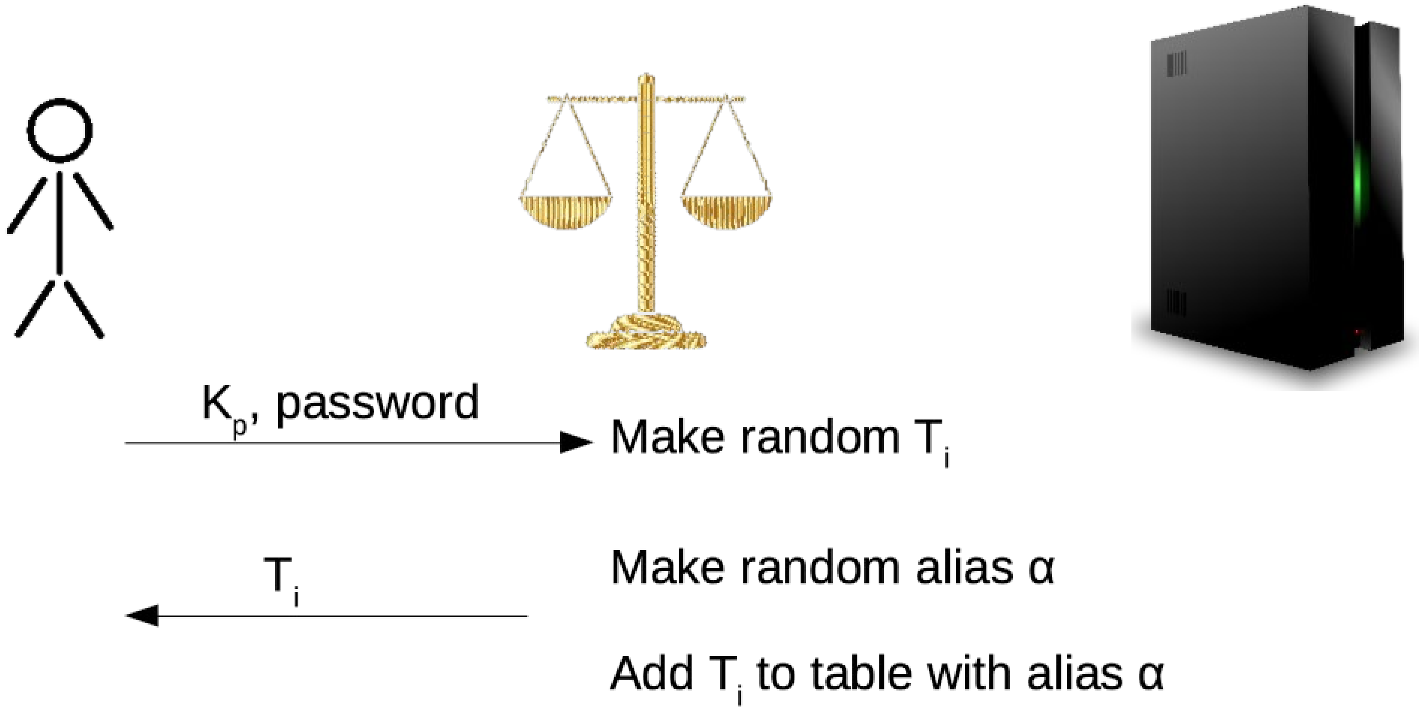
Note: P' is user-entered password, K is payload key,* ticket is **S,E,F,Z**

- User:
  - Hash Password: **C'** = password hash(**S,P'**) ← If **P'** is wrong, **C'** is wrong
  - Send **E,F,C'** to Server.
- Server:
  - Make sure **E** hasn't been used before. ← Previously used tickets get caught here.
  - Check signature in **F.** ← Made-up / unauthorized tickets stop here.
  - Decrypt **E** to get **B**
  - Compute **D'** = HMAC(**B,C'**) ← If **P'** wrong, we get the wrong value for **D'**
  - Send **D'** back to User
- User:
  - Try to decrypt **Z** with **D'** ← If P' wrong, D' wrong, so this fails.

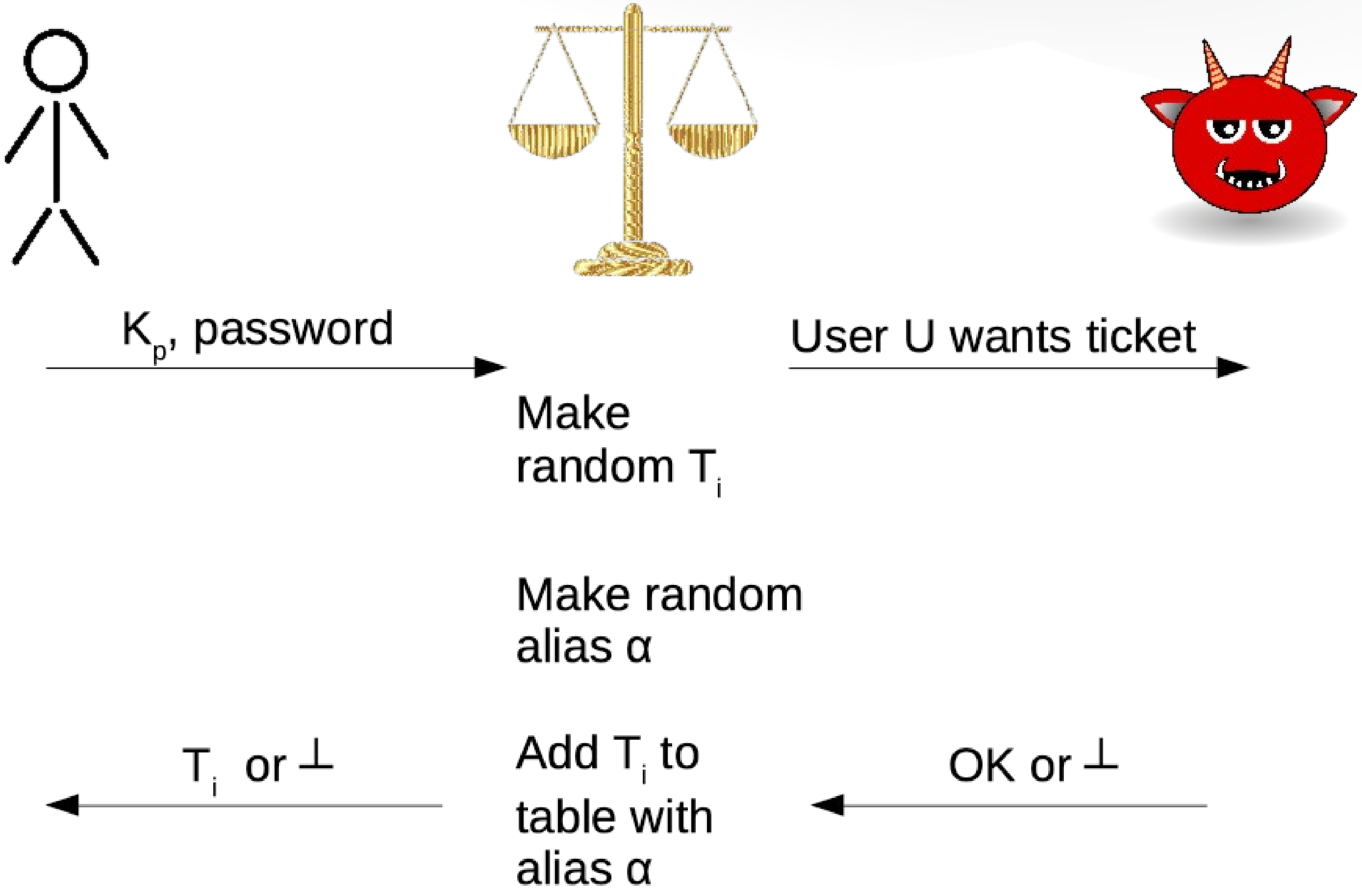
Ideal Functionality



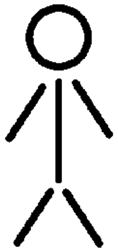
Ideal Functionality: REQUEST with honest server



Ideal Functionality: REQUEST with dishonest server



Ideal Functionality: UNLOCK with honest server



T_i, password →

IF

T_i in not inTable

OR

Password wrong for T_i

THEN

Return $\perp$

← K_p or $\perp$

ELSE

Return K_p

Delete T_i from Table

Ideal Functionality: UNLOCK with dishonest user

T_i, password $\longrightarrow$

IF

T_i in not inTable

OR

Password wrong for T_i

THEN

Return $\perp$

$\longleftarrow$ K_p or $\perp$

ELSE

Return K_p

Delete T_i from Table