Authentication techniques, protocols, and architectures

Antonio Lioy

< lioy @ polito.it >

Politecnico di Torino

Dip. Automatica e Informatica

Definitions of authentication

- **RFC-4949 (Internet security glossary)**
  - "the process of verifying a claim that a system entity or system resource has a certain attribute value"
- **whatis.com:**
  - "the process of determining whether someone or something is who or what it is declared to be"
- **NIST IR 7298 (Glossary of Key Information Security Terms)**
  - "verifying the identity of a user, process, or device, often as a prerequisite to allowing access to resources in an information system"

Definitions of authentication

- **authentication of an "actor"**
  - human being (interacting via sw running on hw)
  - software component
  - hardware element (interacting via sw)
- **shorthand: authN (or also authC)**
- **different from authorization (authZ) but related**

Authentication factors

- **knowledge**
  - something only the user knows, e.g. static pwd, code, personal identification number

MickeyMouse

- **ownership**
  - something only the user possesses (often called an "authenticator"), e.g. token, smart card, smartphone

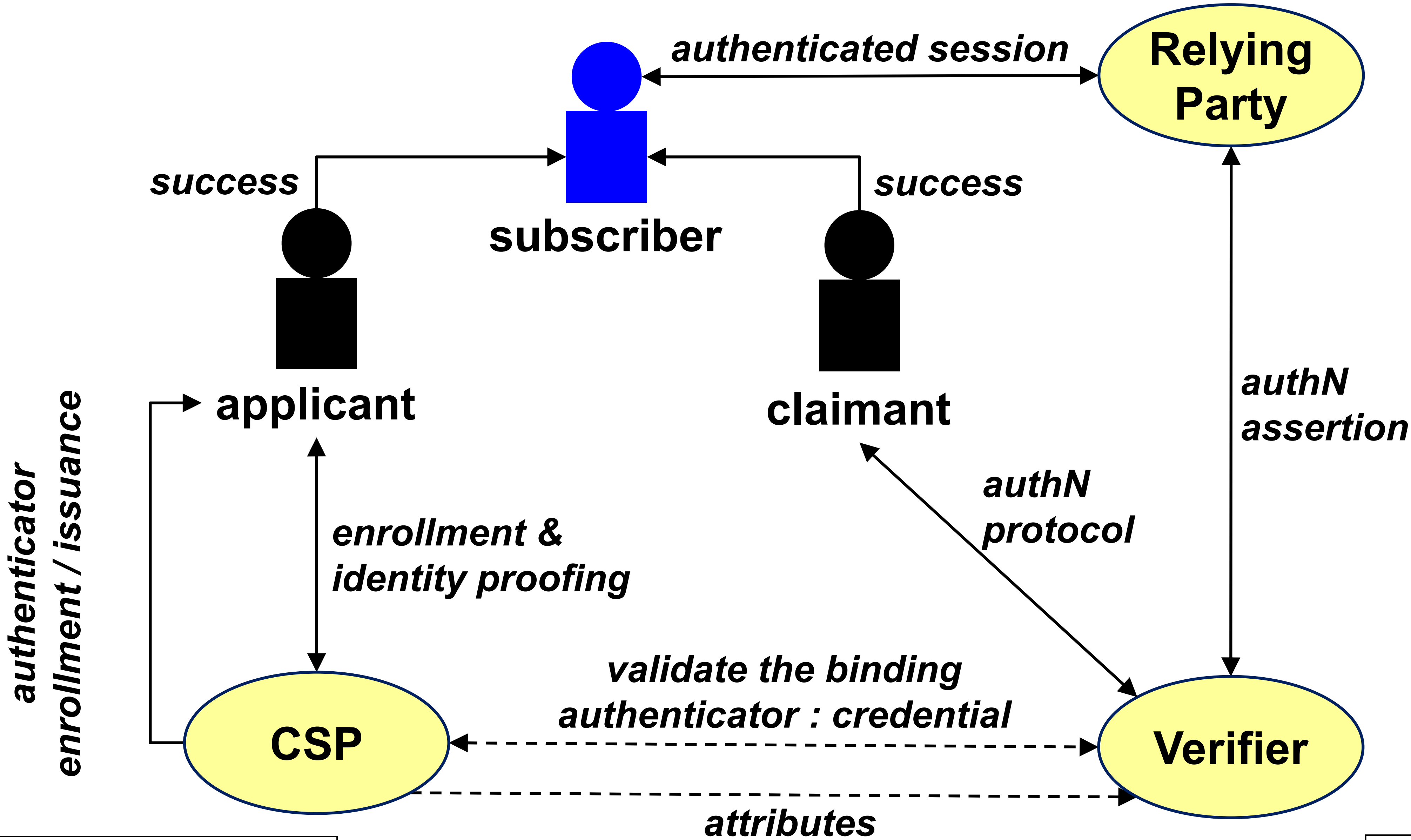
- **inherence**
  - something the user is, e.g. a biometric characteristic (such as a fingerprint)

- **consider application not only to human users but also to processes and devices**

Authentication factors: risks

- **knowledge (e.g. password)**
  - risks = storage and demonstration/transmission
- **ownership (e.g. smartphone)**
  - risks = authenticator itself, theft, cloning, unauthorised usage
- **inherence (e.g. biometrics)**
  - risks = counterfeiting and privacy
  - cannot be replaced when "compromised" (big problem!)
  - use it only for local authentication, as a mechanism to unlock a secret or a device

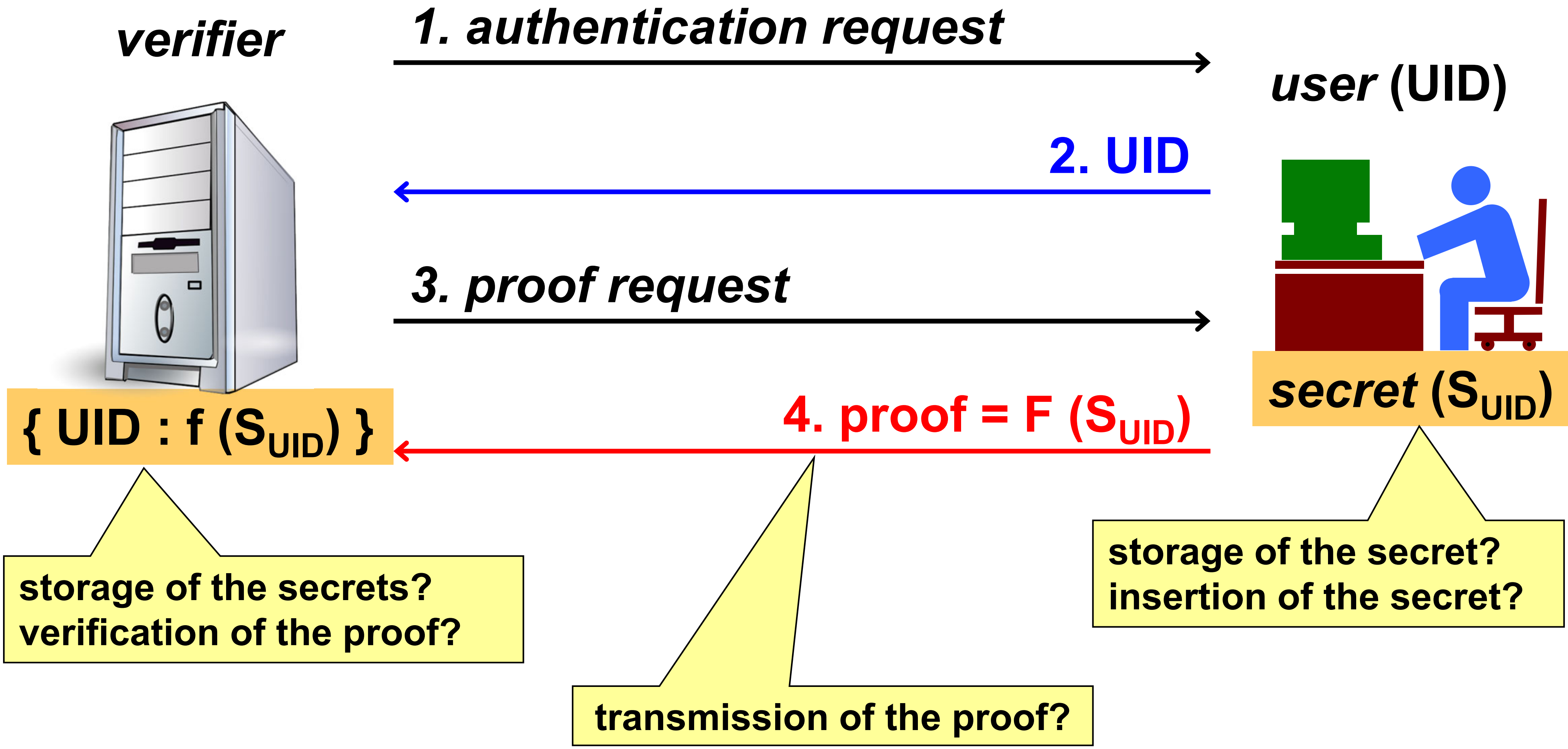
Digital authentication model (NIST SP800.63B)



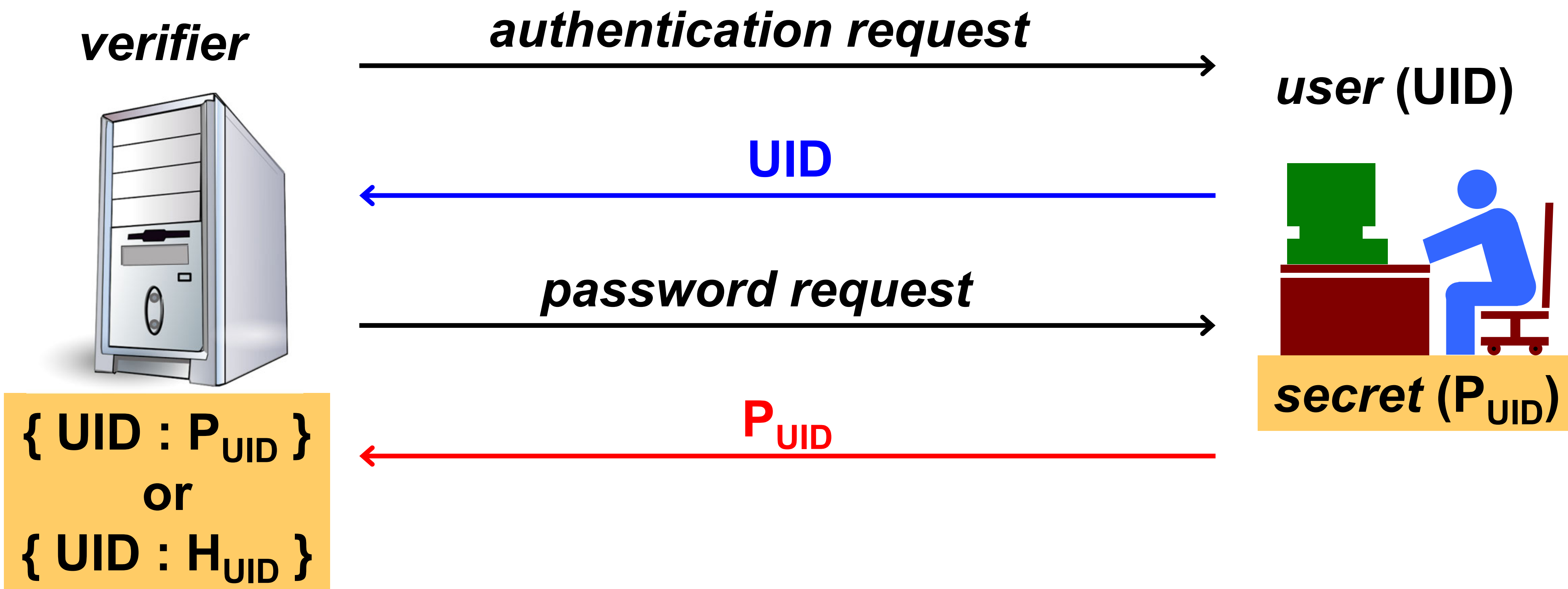
Digital authentication: entities

- **credential binds an authenticator to the subscriber, via an ID**
  - e.g. a X.509 certificate
- **CSP (Credential Service Provider)**
  - will issue or enrol user credential and authenticator
  - verify and store associated attributes
- **Verifier**
  - executes an authN protocol to verify possess of a valid authenticator and credential
- **Relying Party (RP)**
  - will request/receive an authN assertion from the Verifier to assess user identity (and attributes)
- **these roles may be separate or collapsed together**

Generic authentication protocol



Password (reusable)



Password-based authentication

- **secret = the user password**
- **(client) create and transmit proof**
  - F = I (the identity function)
  - i.e. proof = password (cleartext!)
- **(server) verify the proof:**
  - case #1: f = I (the identity function)
    - server knows all passwords in cleartext (!)
    - access control: proof == password ?
  - case #2: f = one-way hash
    - server knows the passwords' digests, H_{UID} (unprotected!)
    - access control: f(proof) == H_{UID} ?

Problems of reusable passwords

- **pwd sniffing**
- **pwd DB attacks (if DB contains plaintext or obfuscated pwd)**
- **pwd guessing (very dangerous if it can be done offline, e.g. against a list of pwd hashes)**
- **pwd enumeration**
  - if pwd limited in length or character type
  - if authN protocol does not block repeated failures
- **pwd duplication (using the pwd for one service against another one, due to user pwd reuse)**
- **cryptography ageing (flexibility on algorithms due to new attacks and more computing power)**
- **pwd capture via server spoofing and phishing**
- **MITM attacks**

Password best practice

- **suggestions to reduce the associated risks:**
  - alphabetic characters (uppercase + lowercase) + digits + special characters
  - long (at least 8 characters)
  - never use dictionary words
  - frequently changed (but not too frequently!)
  - don’t use them ☺
    - but usage of at least one password (or PIN, or access code, or ...) is unavoidable, unless we adopt biometric techniques

Password storage

- **server-side**
  - NEVER in cleartext!
  - encrypted password? then the Verifier must know the encryption key in cleartext …
  - better storing a digest of the password
  - … but beware of the “dictionary” attack
  - … that can be made faster by a “rainbow table”
  - we must insert an unexpected variation, named “salt”
- **client-side**
  - should be only in the user's head … but too many passwords
  - use a post-it ☹ … or an easy pwd (e.g. my son's name) ☹
  - better use an encrypted file (or a "password wallet / manager")

The “dictionary” attack

- **hypothesis:**
  - known hash algorithm
  - known password hash values
- **pre-computation:**
  - for (each Word in Dictionary) do
    store (DB, Word, hash(Word))
- **attack:**
  - let HP be the hash value of a (unknown) password
  - w = lookup (DB, HP)
  - if (success) then write("pwd = ", w)
    else write("pwd not in my dictionary")
- **pre-computation is the key (mounting the attack after discovering HP could take more time than the pwd lifetime)**

Rainbow table (I)

- **a space-time trade-off technique to store (and lookup) an exhaustive hash table (less space, more time)**
  - makes exhaustive attack feasible for certain password sets
- **e.g. table for a 12 digits password**
  - exhaustive = 10^{12} rows { P_i : HP_i }
  - rainbow = 10^9 rows, each representing 1000 pwd
- **uses the reduction function** $r : h \Rightarrow p$ **(which is NOT** h^{-1}**)**
- **pre-computation:**
  - for (10^9 distinct P)
    - for (p=P, n=0; n<1000; n++)
      - k = h(p); p = r(k);
    - store (DB, P, p) // chain head and tail

Rainbow table (II)

- **attack:**
  - let HP be the hash of a password
  - for (k=HP; n=0; n<1000; n++)
    - p = r(k)
    - if lookup(DB, x, p) then exit ("chain found!")
    - k = h(p)
  - exit ("HP is not in any chain of mine")
- **to avoid "fusion" of chains r_0( ) … r_n( ) are used for the different reduction steps**
- **on sale pre-computed rainbow tables for various hash functions and password sets (e.g. SHA1 for alphanumeric)**
- **this technique is used by various attack programs**

Using the salt in storing passwords

- **for each user UID:**
  - create / ask the pwd
  - generate a salt (different for each user)
    - random (unpredictable) and long (increased dictionary complexity)
    - should contain rarely used or control characters
  - compute HP = hash (pwd || salt)
- **store the triples { UID, HP_{UID}, $salt_{UID}$ }**
- **additional benefit: we have different HP for users having the same pwd**
- **makes the dictionary attacks nearly impossible**
  - included those based on rainbow tables (a space-time trade-off technique to enable exhaustive search for a character set)

Example: passwords in Linux

- **originally stored in /etc/passwd, hashed with a DES-based hash function named crypt()**
- **since /etc/passwd needs to be world-readable (contains usernames, UID, GID, home, shell, …) passwords have been moved to /etc/shadow readable only by system processes**
- **passwords are stored in the following form – see crypt(5):**
  - idsalt$hashedpwd
  - different hash functions used depending on ID, for example:
    - 1 = MD5, …, 5 = SHA-256, 6 = SHA-512, …
  - if idsalt is absent then the old DES-based hash is used (with 12-bits salt, pwd truncated to 8 characters) – danger!
  - some algorithms have adjustable complexity (to counter brute force attacks)

The Linkedin attack

- **June 2012, copied 6.5 M password from Linkedin**
  - … unsalted, plain SHA-1 hash!!!
- **crowdsourcing used for cooperative password cracking**
  - at least 236,578 passwords found (before ban of the site publishing the password hashes)
- **note: nearly simultaneous problem with the discovery that the Linkedin app for iPad/iPhone was sending in clear sensible data (not relevant to Linkedin!)**

Example: passwords in MySQL

- **username and password stored in the "user" table**
- **MySQL (from v 4.1) uses a double hash (but no salt!) to store the password**
  - sha1(sha1(*password*))
- **the hex encoding of the result is stored, preceded by * (to distinguish this case from MySQL < 4.1)**
- **example (for the password "Superman!!!"):**
  - field user.password = *868E8E4F0E782EA610A67B01E63EF04817F60005
  - verification

```
$ echo -n 'Superman!!!'| openssl sha1 -binary | openssl sha1 -hex
(stdin)= 868e8e4f0e782ea610a67b01e63ef04817f60005
```

Strong (peer) authN

- "strong authN" often requested in specifications
- … but never formally defined (or defined in too many different ways, which is useless)

Strong authN: ECB definition

- strong customer authN is a procedure based on the use of two or more of knowledge, ownership, and inherence
- the elements selected must be mutually independent, i.e. the breach of one does not compromise the other(s)
- at least one element should be non-reusable and non-replicable (except for inherence), and not capable of being surreptitiously stolen via the Internet
- the strong authentication procedure should be designed in such a way as to protect the confidentiality of the authentication data

Strong authN: PCI-DSS definition

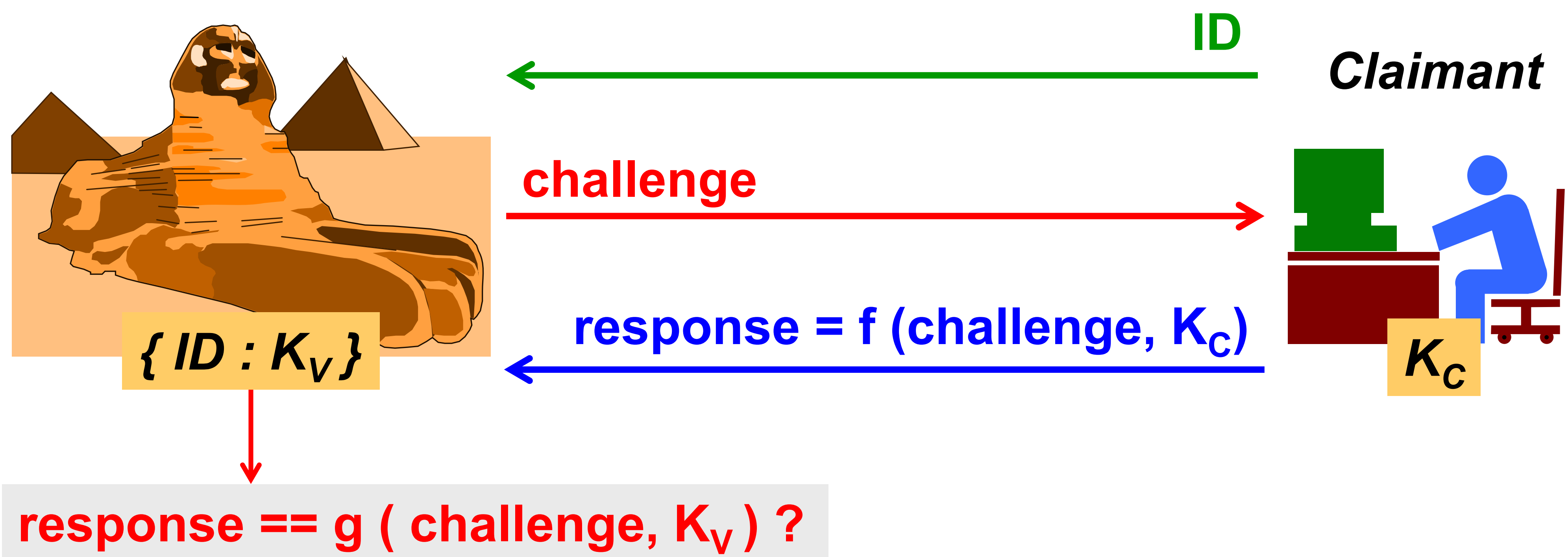
- **v3.2 requires multi-factor authentication (MFA) for access into the cardholder data environment (CDE)**
  - from trusted or untrusted network
  - by administrators
  - exception: direct console access (physical security)
- **… and for remote access**
  - from untrusted network
  - by users and third-parties (e.g. maintenance)
- **best practice until 2018/01, compulsory afterwards**
- **MFA is *not* twice the same factor (e.g. two passwords)**

Strong authN: other definitions

- **Handbook of Applied Cryptography**
  - a cryptographic challenge-response identification protocol
- **more in general**
  - technique resisting to a well-defined set of attacks
- **conclusion:**
  - an authN technique can be regarded as strong or weak depending on the attack model
    - e.g. users of Internet banking > ECB definition
    - e.g. employees of PSP > PCI-DSS definition
- **watch out for your specific application field = risks**

Challenge-response authentication (CRA)

- a challenge is sent to the Claimant...
- ... who replies with the solution computed using some secret knowledge and the challenge
- the Verifier compares the response with a solution computed via a secret associated to the Claimant



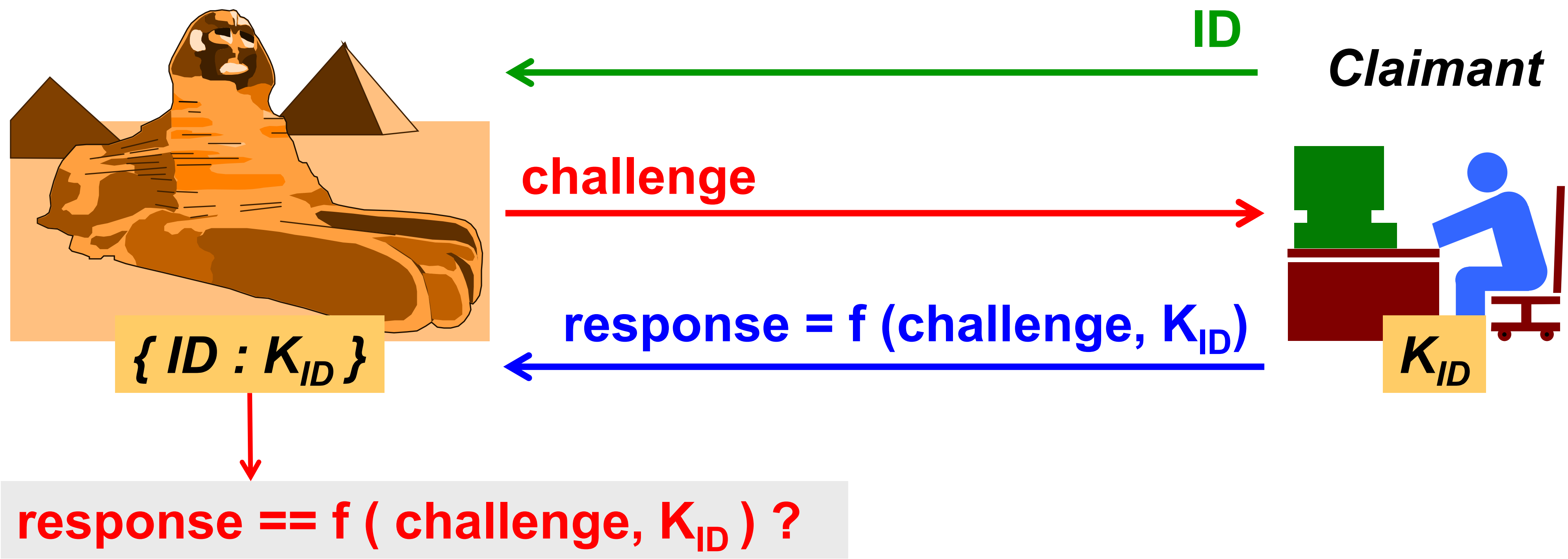
CRA: general issues

- **the challenge must be non-repeatable to avoid replay attacks**
  - usually the challenge is a (random) nonce
- **the function f must be non-invertible**
  - otherwise, a listener can record the traffic and easily find the shared secret

$$K_C = f^{-1} \text{ (response, challenge)}$$

Symmetric CRA

- Claimant and Verifier share a secret (e.g. a pwd or a key)
- a challenge is sent to the Claimant ...
- ... who replies with the solution after a computation R involving the shared secret and the challenge

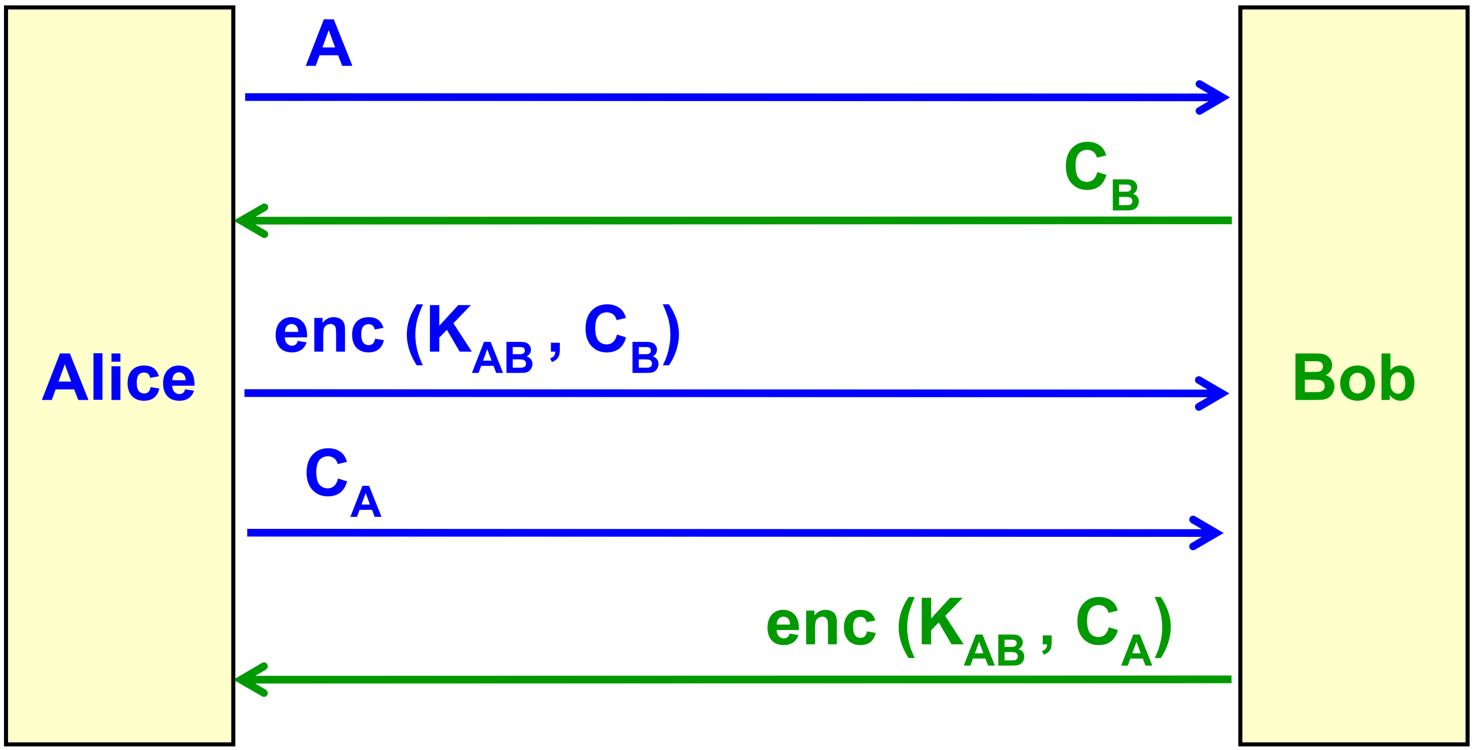


Symmetric CRA: general issues

- **the easiest implementation uses a hash function (faster than encryption)**
  - sha1 (deprecated), sha2 (recommended), sha3 (future)
- **Kc must be known in cleartext to the Verifier**
  - attacks against the { ID:K } table at the Verifier
- **SCRAM (Salted CRA Mechanism) solves this problem by using hashed passwords at the Verifier**
  - offers also channel binding
  - offers also mutual authentication

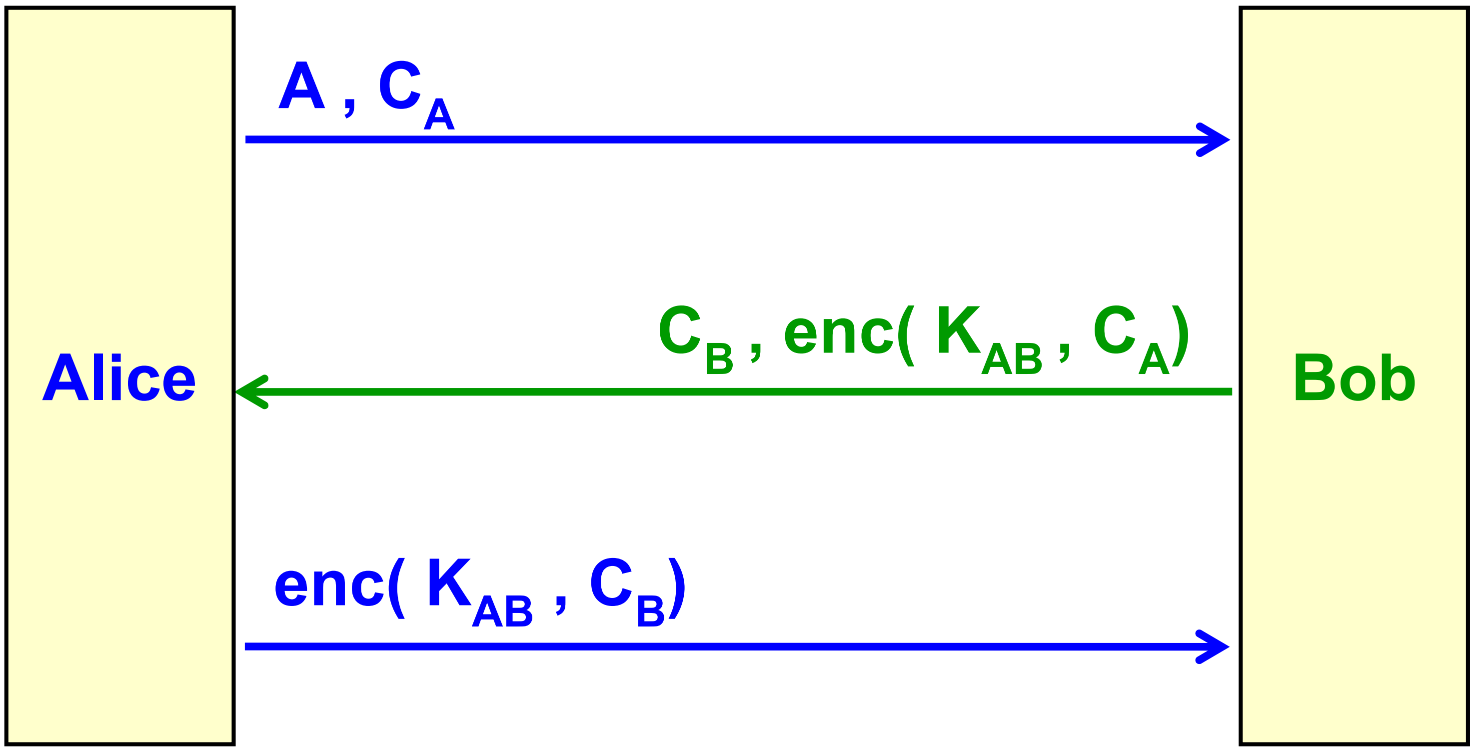
Mutual symmetric CRA (v1)

- this is the base exchange
- only the initiator provides explicitly its (claimed) identity
- BEWARE! old & bad protocol

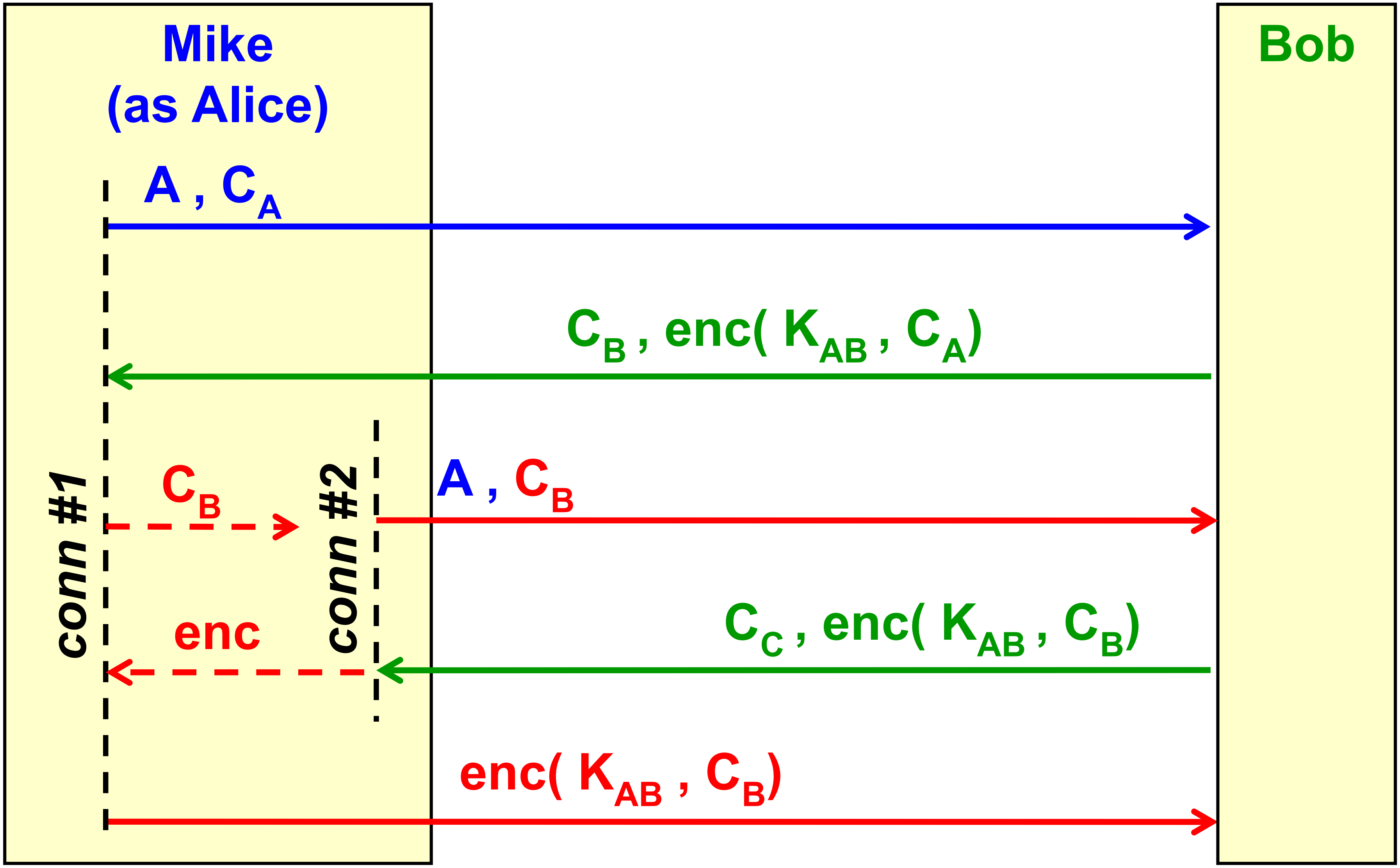


Mutual symmetric CRA (v2)

- reduction in the number of messages (better performance but no impact on security)
- used by the IBM SNA



Attack to the mutual symmetric CRA



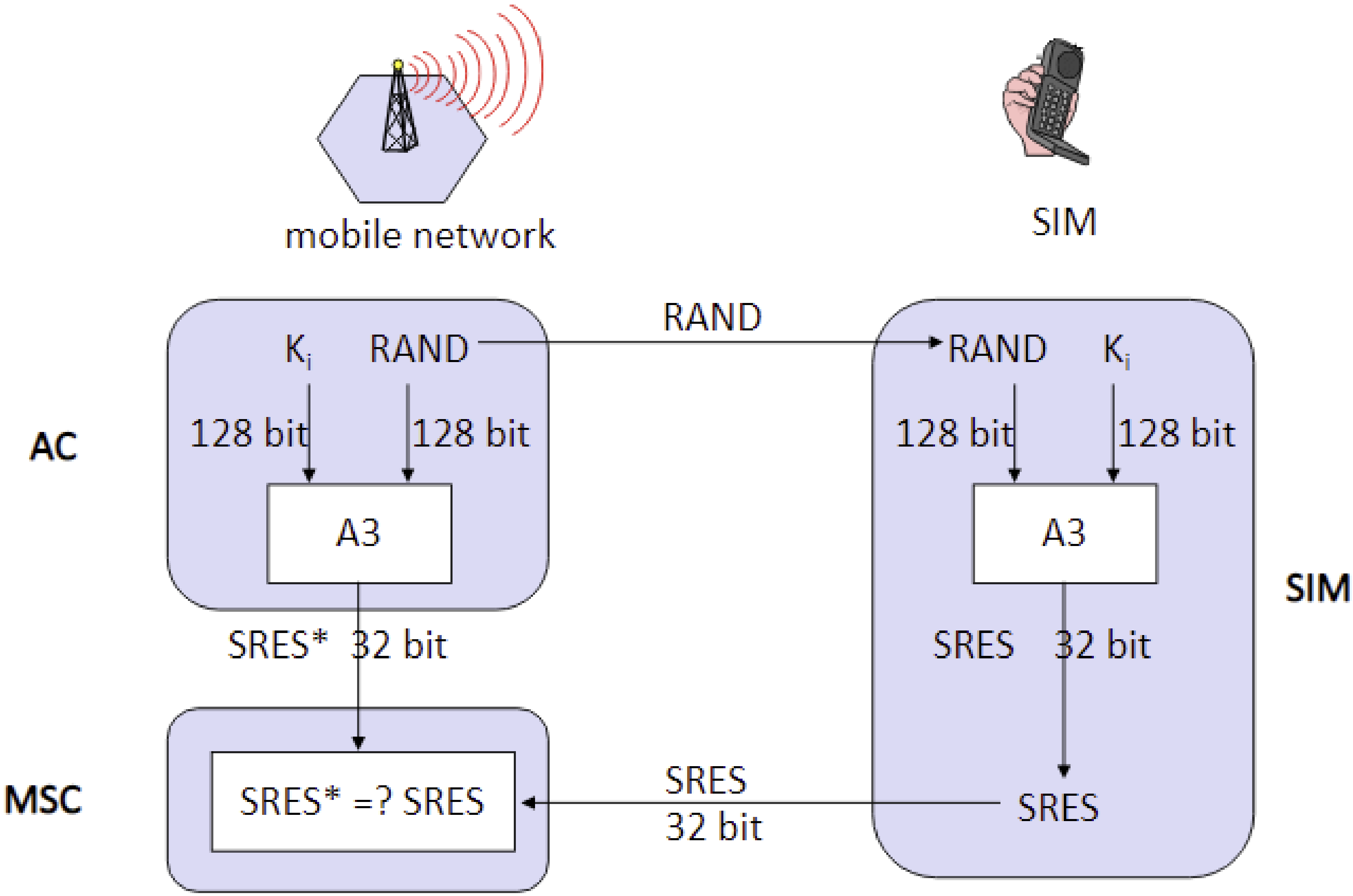
GSM (in)security

- **GSM uses three secret algorithms:**
  - A8 for symmetric key generation (in the SIM)
  - A3 for authentication (in the SIM)
  - A5 (stream cipher) for encryption (in the mobile device)
    - LFSR-based: A5/1 most used, A5/2 weak (some countries)
    - A5/3 based on the Kasumi block cipher
- **this is security-through-obscurity … always a bad idea ☹**
- **A8, A3, A5 are left to the choice of the MNO**
  - A8 and A3 usually built upon the COMP128 (secret) function
  - Z = COMP128(X, Y) … with X, Y, Z 128 bits each
  - A8: Kc = lsb(54, Z) [connection key]
  - A3: SRES = msb(32, Z) [Signed RESponse]

GSM authentication

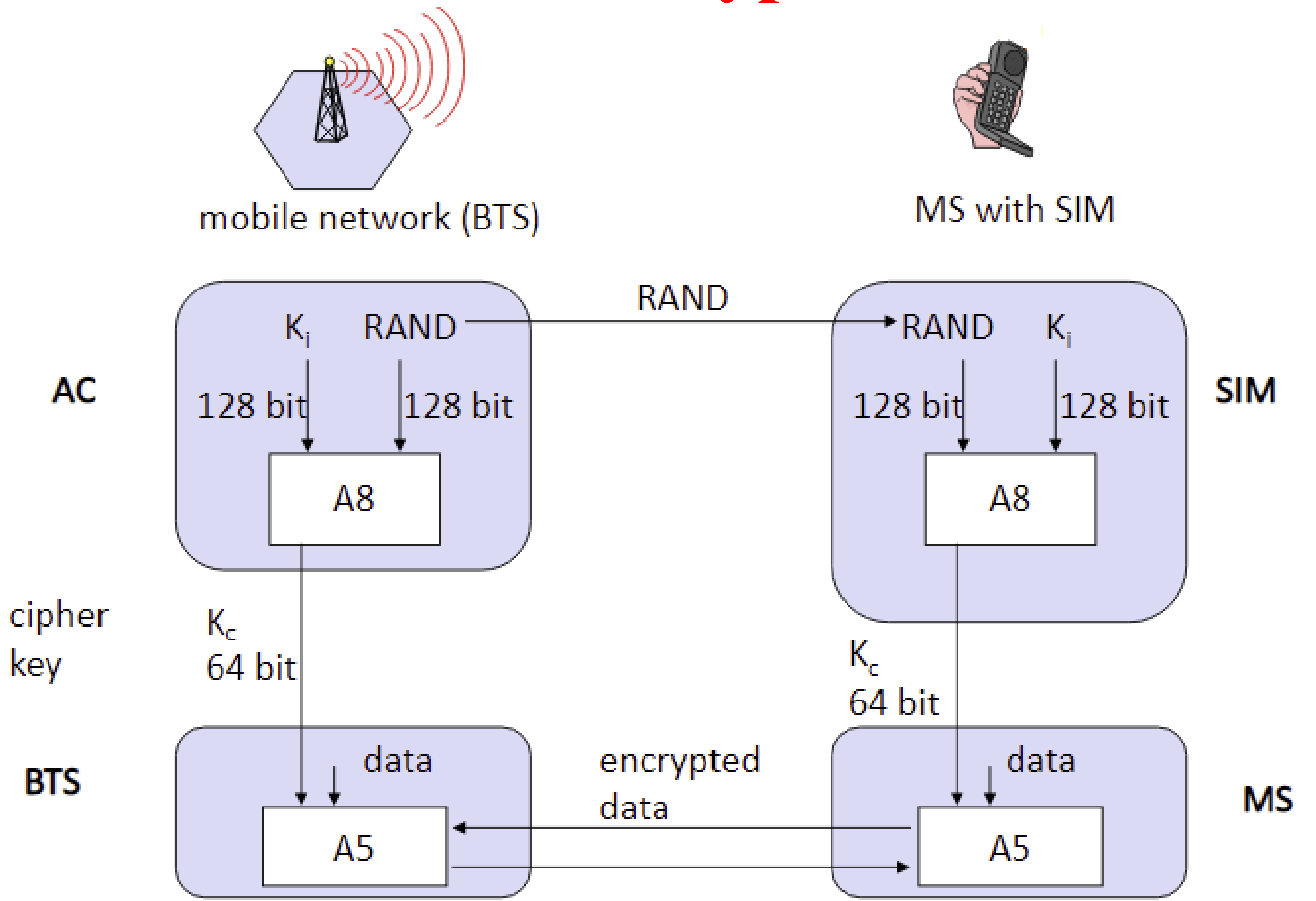
- **a symmetric CRA is used to authenticate the Mobile Station (MS) via its SIM to the Base Station (BS)**
  - SIM contains Ki (individual subscriber authN key)
  - Ki is a 128-bit secret shared with the AC (AuthN Centre)
- **the BS sends to the SIM a random challenge C of 128 bit**
- **the SIM returns SRES = A3(C, Ki) of 32 bit**
- **but …**
  - COMP128-1 is weak … with chosen-challenge (and differential cryptoanalysis) 150,000 challenges are sufficient to compute Ki
  - now we can
    - clone the SIM (i.e. same Ki)
    - decrypt the traffic by computing Kc for that Ki and C sent by the BS

GSM authentication



(source: https://www.ques10.com/p/48395/gsm-authentication-procedure/)

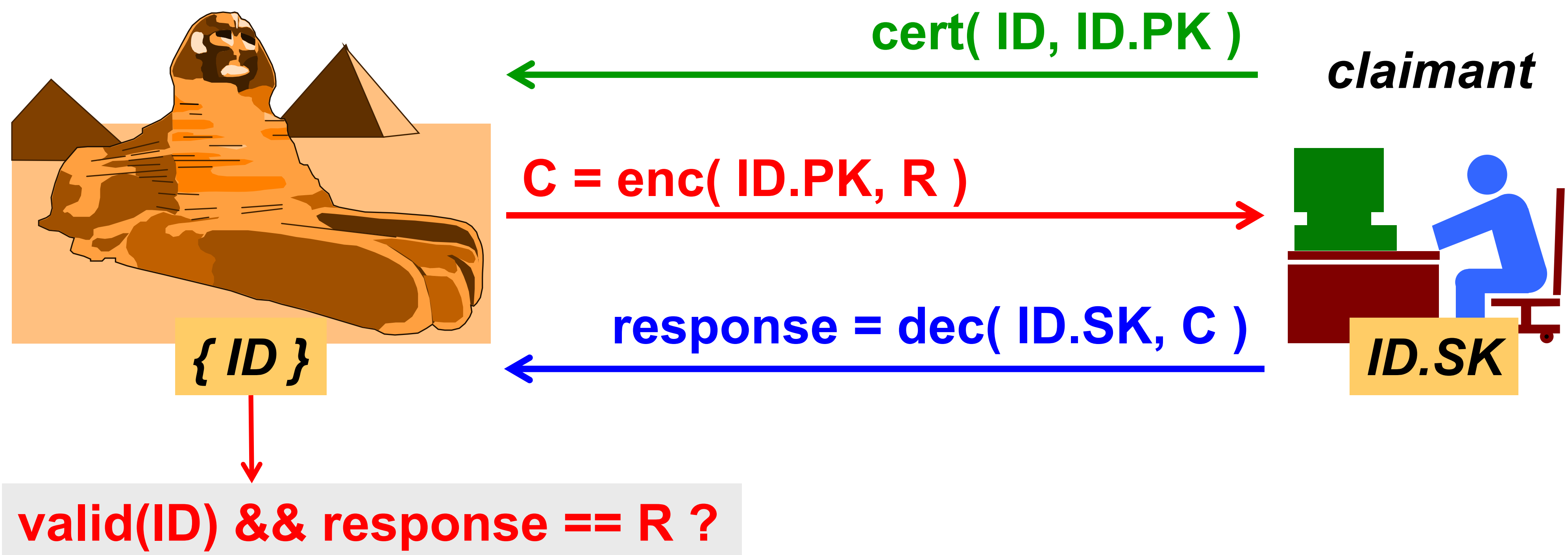
GSM encryption



(source: https://www.ques10.com/p/48395/gsm-authentication-procedure/)

Asymmetric CRA

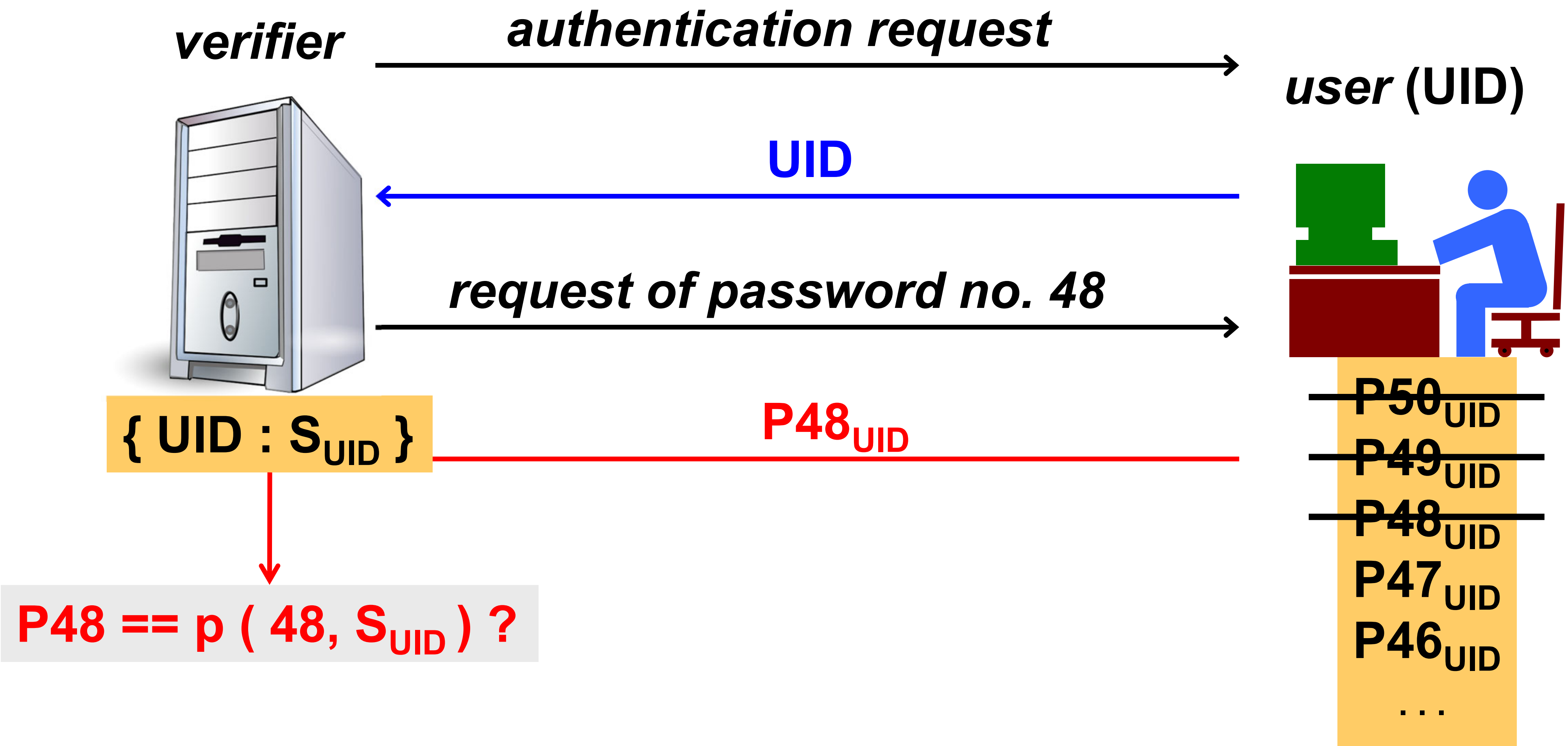
- a random nonce R is encrypted with the user's public key ...
- ... and the users replies by sending R in clear, thanks to its knowledge of the private key



Asymmetric CRA: analysis

- **the strongest mechanism**
- **does not require secret storage at the Verifier**
- **implemented for peer authentication (client and server) in IPsec, SSH, and TLS**
- **cornerstone for user authentication in FIDO**
- **problems**
  - slow
  - if designed inaccurately may lead to an involuntary signature by the Claimant
  - PKI issues (trusted root, name constraint, revocation)
    - avoidable if the Verifier stores ID.PK
    - … but this moves equivalent PKI effort to the Verifier

One-time password (OTP)



One-Time Password (OTP)

- **password valid only for one run of the authentication protocol**
  - next run requires another password
- **immune to sniffing**
- **subject to MITM (needs Verifier authentication)**
- **difficult provisioning to the subscribers**
  - lot of passwords
  - password exhaustion
- **difficult password insertion**
  - typically contains random characters to avoid guessing

OTP provisioning to the users

- **on “stupid” or insecure/untrusted workstation:**
  - paper sheet of pre-computed passwords
    - “password cards”
  - hardware authenticator (crypto token)
- **on intelligent and secure/trusted workstation :**
  - automatically computed by an ad-hoc application
  - typical for smartphone, tablet, laptop, …

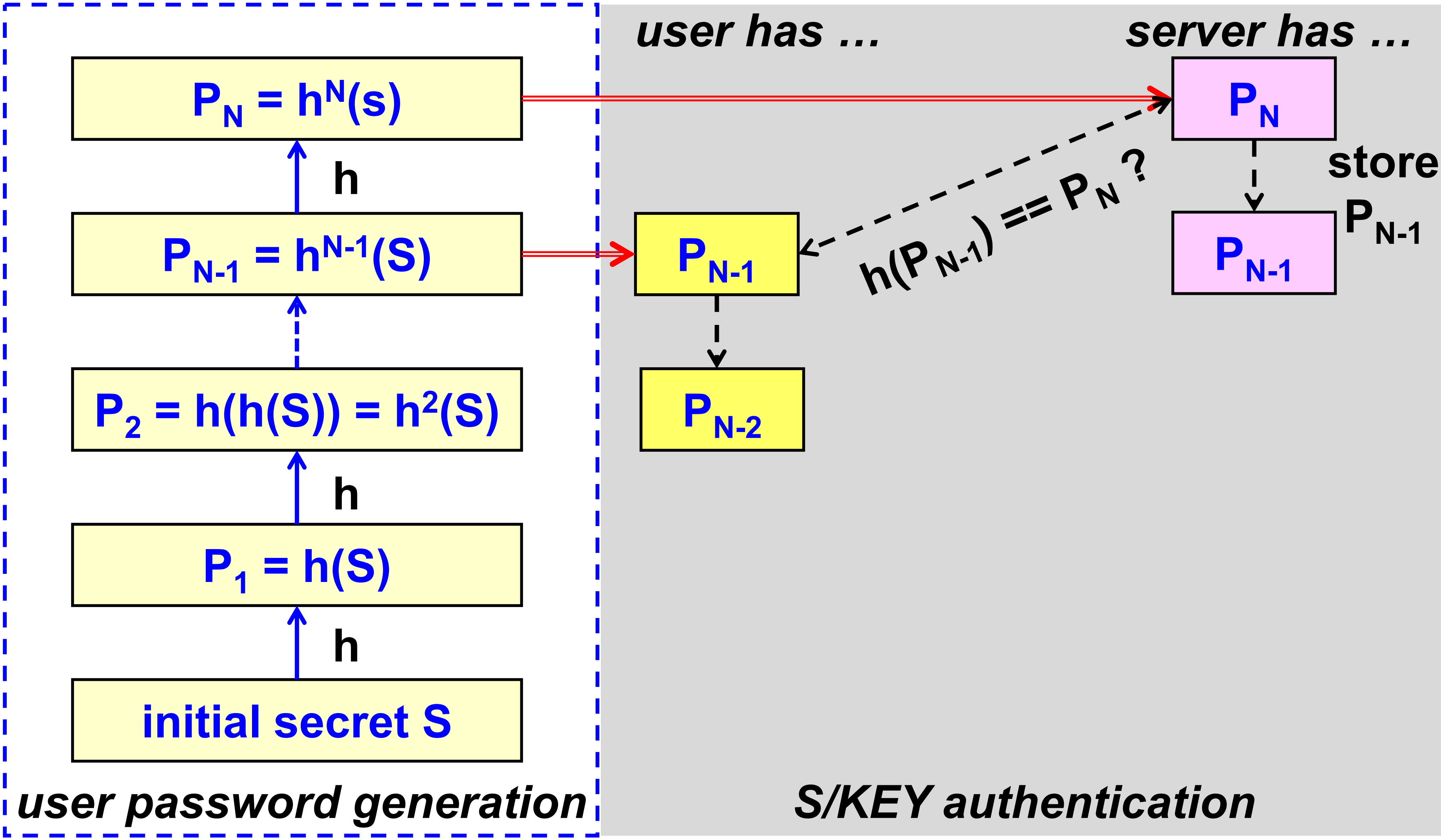
The S/KEY system (I)

- **first OTP definition and implementation by Bell Labs (1981)**
- **the user generates a secret S_{ID}**
- **the user computes N one-time passwords:**
  - $P_1=h(S_{ID})$, $P_2=h(P_1)$, …, $P_N=h(P_{N-1})$
- **the Verifier stores the last one P_N**
  - this password will never be used directly for authentication, but only indirectly
- **Verifier asks for P_{N-1} and gets X**
  - i.e. asks for pwd in inverse order
  - if (P_N != h(X)) then FAIL else {OK; store X as P_{N-1}}

The S/KEY system (II)

- **in this way:**
  - the Verifier has no need to know the user's secret
  - only the user knows all passwords
- **RFC-1760**
  - uses MD4 (other choices are possible)
- **S/KEY is an example of Off-line / Pre-computed OTP**

The S/KEY system (III)



S/KEY – generation of the password list

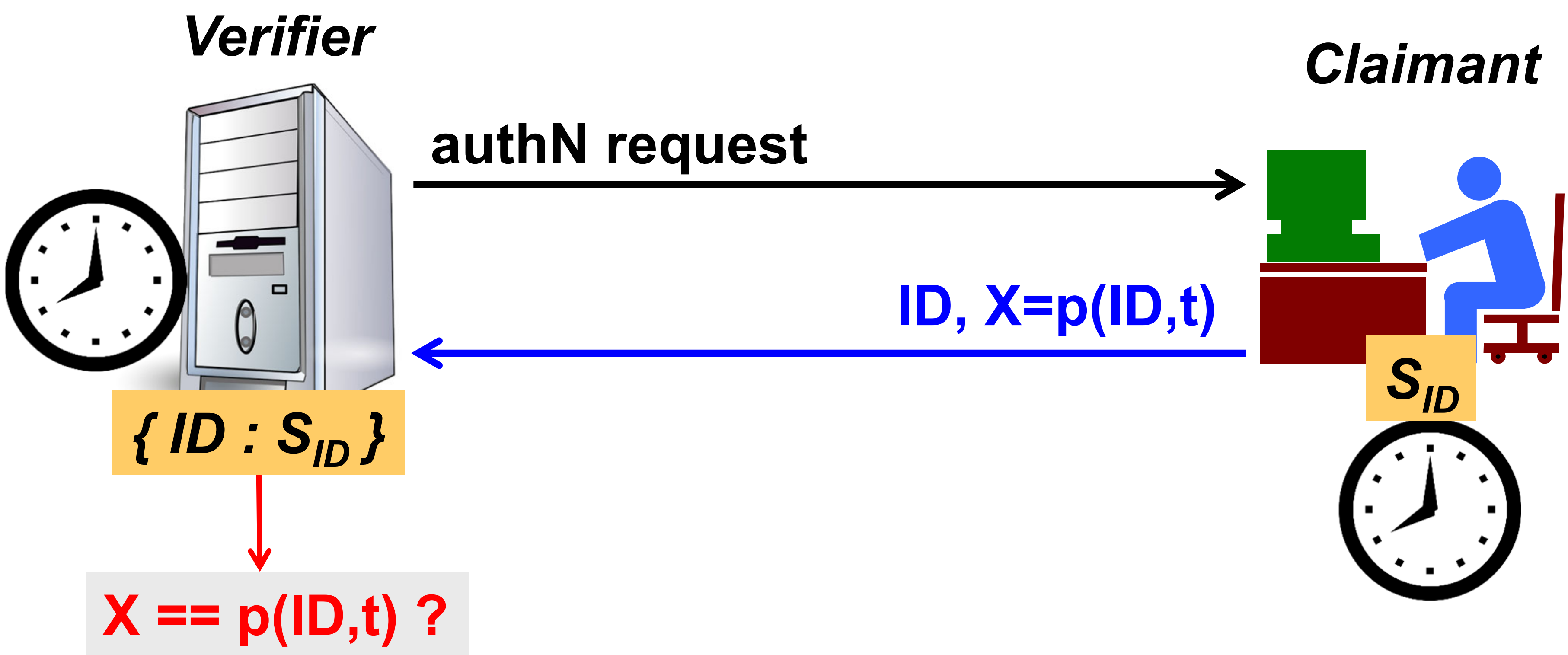
- **the user inserts a pass phrase (PP):**
  - minimum 8 char long
  - secret! (if disclosed then the security of S/KEY is compromised)
- **PP is concatenated with a server-provided seed**
  - the seed is not secret (sent in cleartext from S to C)
  - allows to use the same PP for multiple servers (using different seeds) and to safely reuse the same PP by changing the seed
- **a 64-bit quantity is extracted from the MD4 hash (by XORing the first / third 32-bit groups and the second / fourth groups)**

S/KEY – passwords

- **64-bit passwords are a compromise**
- **neither too long (complex) nor too short (insecure)**
- **entered as a sequence of six short English words chosen from a dictionary of 2048 (e.g. 0="A", 1="ABE", 2="ACE", 3="ACT", 4="AD", 5="ADA")**
- **client and server must share the same dictionary**
- **example (using the dictionary in RFC-1760):**
  - password (text) YOU SING A NICE OLD SONG
  - password (numeric)
    - 1D6E5001884BD711 (hex)
    - 2,120,720,442,049,943,313 (decimal)

Time-based OTP

- **the password depends upon time and the user's secret:**
  - $p(ID,t) = h(t, S_{ID})$



Time-based OTP: analysis

- **requires local computation at the subscriber**
- **requires clock synchronization (or keeping track of time-shift for each subscriber)**
- **requires time-slot and authentication window**
  - X == p(ID,t) || X == p(ID,t-1) || X == p(ID,t+1)
- **only one authentication run per time-slot**
  - typically 30s or 60s (not good for some services)
- **time attacks against subscriber and Verifier**
  - fake NTP server or mobile network femtocell
- **sensitive database at the verifier**
  - see the attack against RSA SecurID

A TOTP example: RSA SecurID

- **the Claimant sends to the Verifier in clear**
  ***user* , PIN , *token-code* (seed, time)**
  or (if an authenticator with pinpad is used)
  ***user* , *token-code** (seed, time, PIN)**
- **based on *user* and PIN the Verifier checks against three possible token-codes:**
  $\mathbf{TC_{-1}, TC_0, TC_{+1}}$
- ***duress code*: PIN to generate an alarm (to be used under attack)**
- **ACE (Access Control Engine) components**
  - ACE client (installed at the Relying Party)
  - ACE server (implements the Verifier)

RSA SecurID: recent products

SID700

SD600

SID800

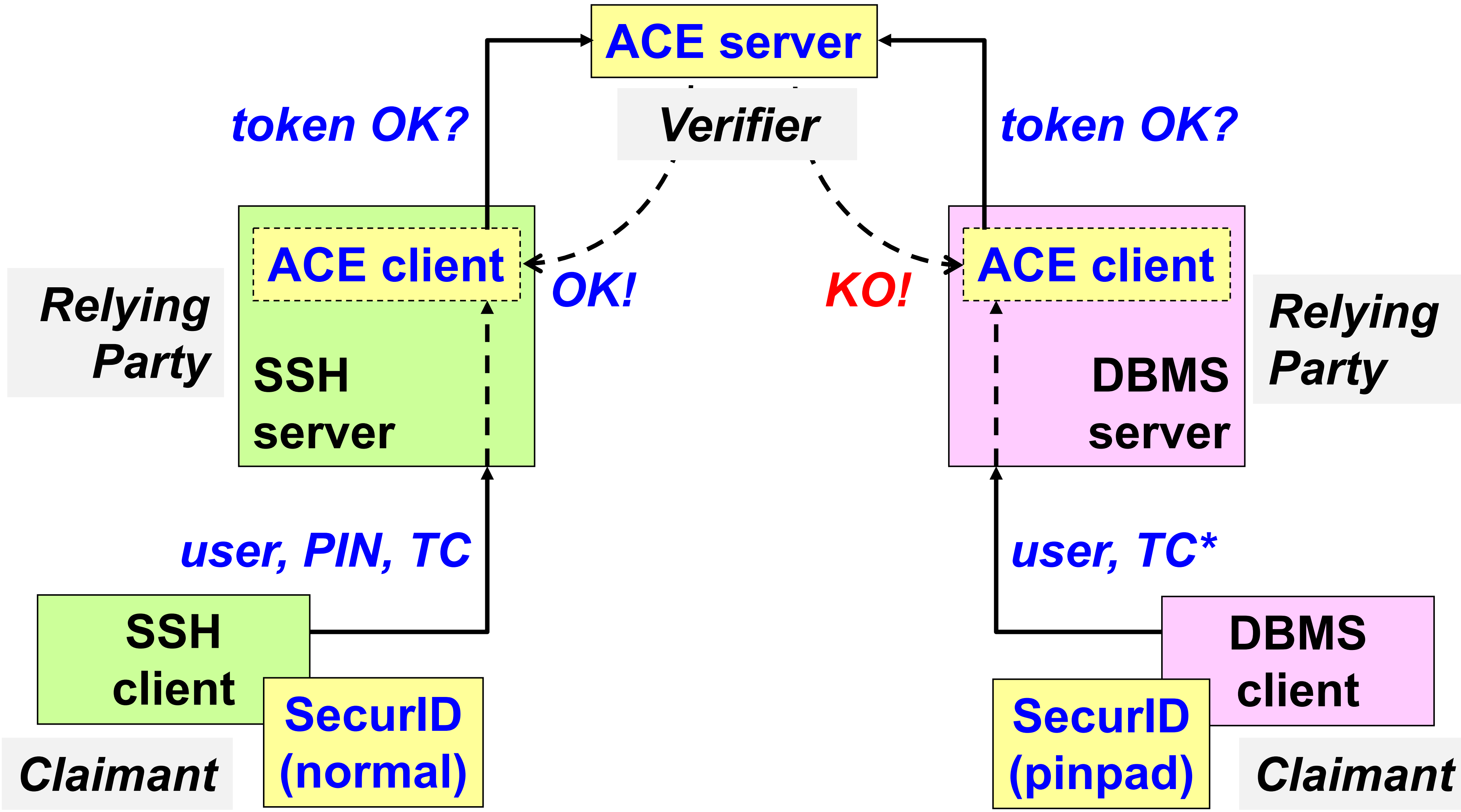
SD200

SD520

SoftID Token

PINPAD

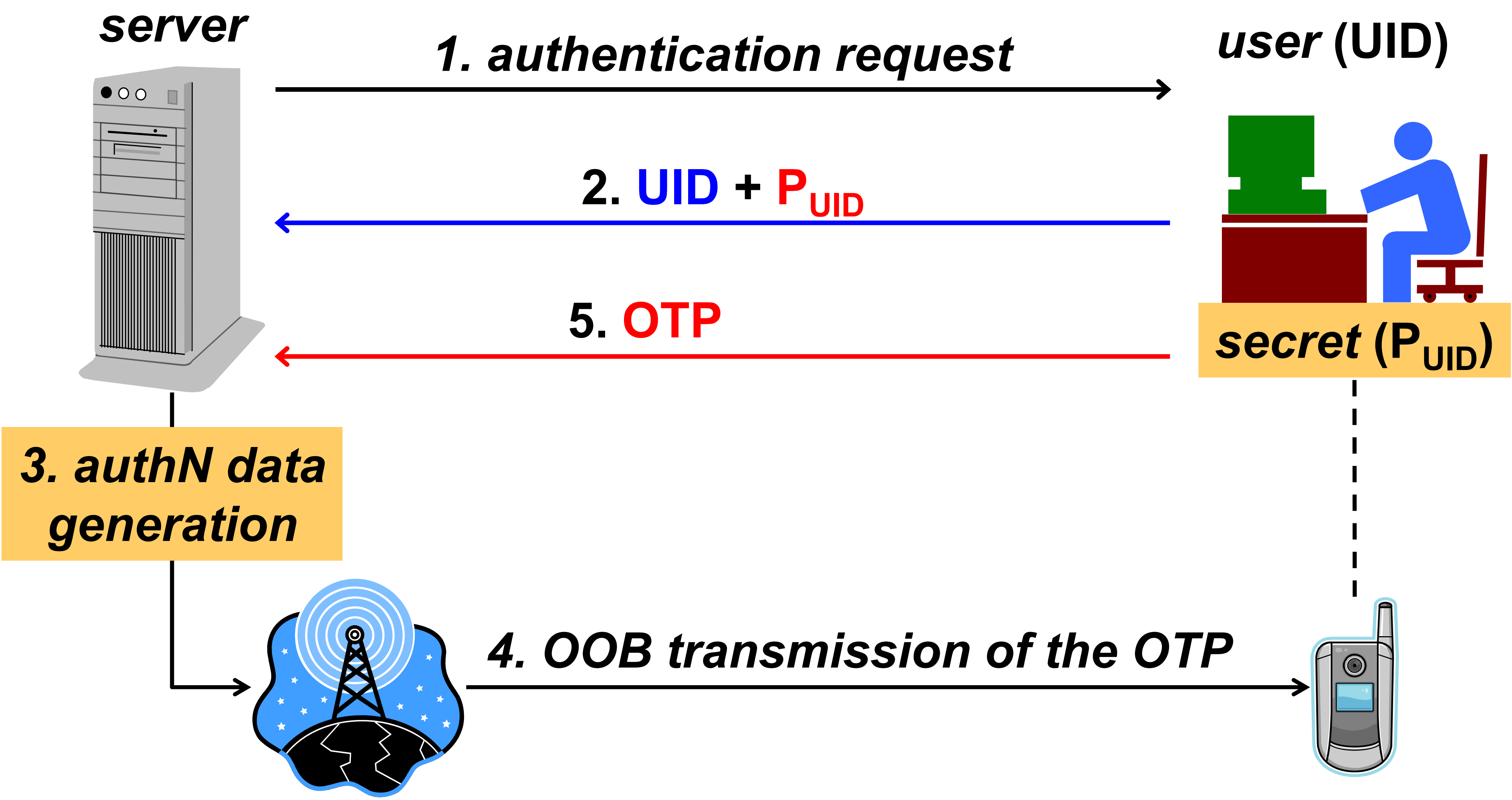
SecurID: architecture



Event-based OTP

- **uses monotonic integer counter C as input besides the seed**
  - $p(ID,C) = h(C, S_{ID})$
- **requires local computation at the subscriber**
- **counter incremented at the subscriber (e.g. button)**
- **frequent authentication runs are possible**
- **OTP pre-computation is possible**
  - useful to travel w/o authenticator (avoid risk of loss/stealing)
  - … but can also be done by an adversary with temporary access to the authenticator ☹
- **Verifier must accommodate desynchronization**
  - the subscriber pushed button unwillingly
  - X==p(ID,C) || X==p(ID,C+1) || X==p(ID,C+2) || … ?

Out-of-band OTP



Out-of-Band OTP

- **at step 5 secure channel w/ server authentication needed to avoid MITM attacks**
- **OOB channel frequently is text/SMS message**
  - can be attacked due to problems of VoIP, mobile user identification, and SS7 protocol
- **NIST SP800-63.B**
  - use of PSTN (SMS or voice) as OOB channel is deprecated
  - suggest using Push mechanism over TLS channel to registered subscriber device

Two-/Multi-Factors AuthN (2FA/MFA)

- **use more than one factor**
  - to increase authN strength
  - to protect authenticator
- **PIN used for authenticator protection**
  - PIN transmitted along with OTP
  - PIN entered to compute the OTP itself
  - PIN (or inherence factor) used to unlock the authenticator, very risky if:
    - lock mechanism weak
    - no protection from multiple unlock attempts
    - unlocking valid for a time window

Importance of MFA: the Iphone ransomware

- **May 2014**
- **iCloud accounts (with 1-factor authN) violated**
- **then "remote lock" used with "find my device"**
- **also a message is sent to the device (iphone, ipad):**
  - "Device hacked by Oleg Pliss!"
  - to regain control send 100 USD/EUR via Paypal to lock404(at)hotmail.com
- **don't wanna pay? then use "recovery mode" (but all the device data and app are lost…)**
- **paying doesn't help either! (fake Paypal account)**

http://thehackernews.com/2014/05/apple-devices-hacked-by-oleg-pliss-held.html

Authentication of human beings

- **how can we be sure of interacting with a human being rather than with a program (e.g. sensing a password stored in a file)?**
- **two solutions:**
  - CAPTCHA techniques (Completely Automated Public Turing test to tell Computers and Humans Apart)
    - e.g. picture with images of distorted characters
  - biometric techniques
    - e.g. fingerprint

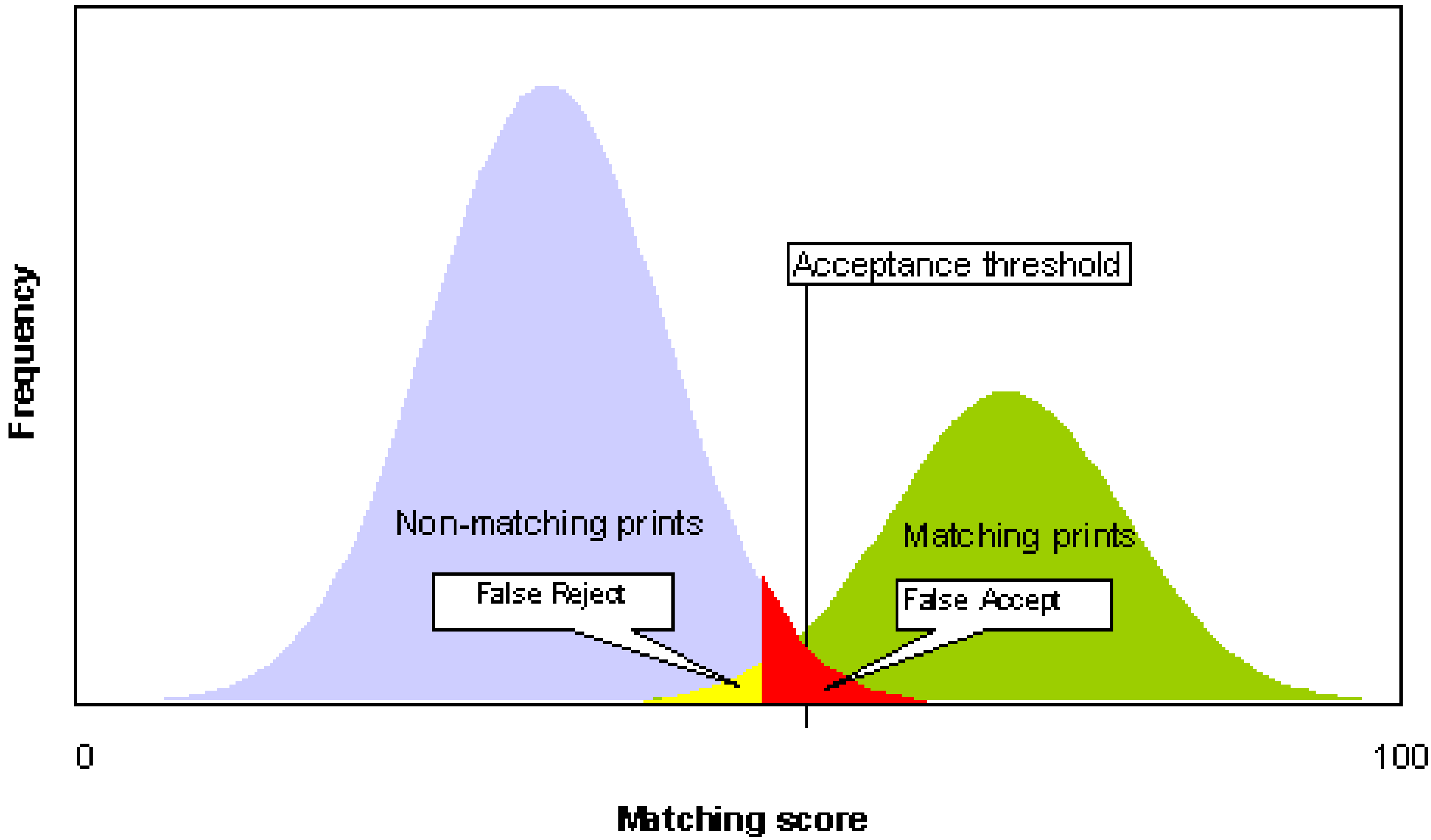
Biometric systems

- **measure of one biologic characteristics of the user**
- **main characteristics being used:**
  - fingerprint
  - voice
  - retinal scan
  - iris scan
  - hands' blood vein pattern
  - heart rate
  - hand geometry
- **each technique can potentially be circumvented**
- **additionally … NOT REPLACEABLE (!!!)**

Problems of biometric systems

- **FAR = False Acceptance Rate**
- **FRR = False Rejection Rate**
- **FAR and FRR may be partly tuned but they heavily depend on the cost of the device**
- **variable biological characteristics:**
  - finger wound
  - voice altered due to emotion
  - retinal blood pattern altered due to alcohol or drug

FAR / FRR



Frequency
Acceptance threshold
Non-matching prints
Matching prints
False Reject
False Accept
0
100
Matching score

Problems of biometric systems

- **psychological acceptance:**
  - “Big Brother” syndrome (=personal data collection)
  - some technologies are intrusive and could harm
- **privacy**
  - it's an identification
- **cannot be changed if copied**
  - hence only useful to *locally* replace a PIN or a password
- **lack of a standard API / SPI:**
  - high development costs
  - heavy dependence on single/few vendors

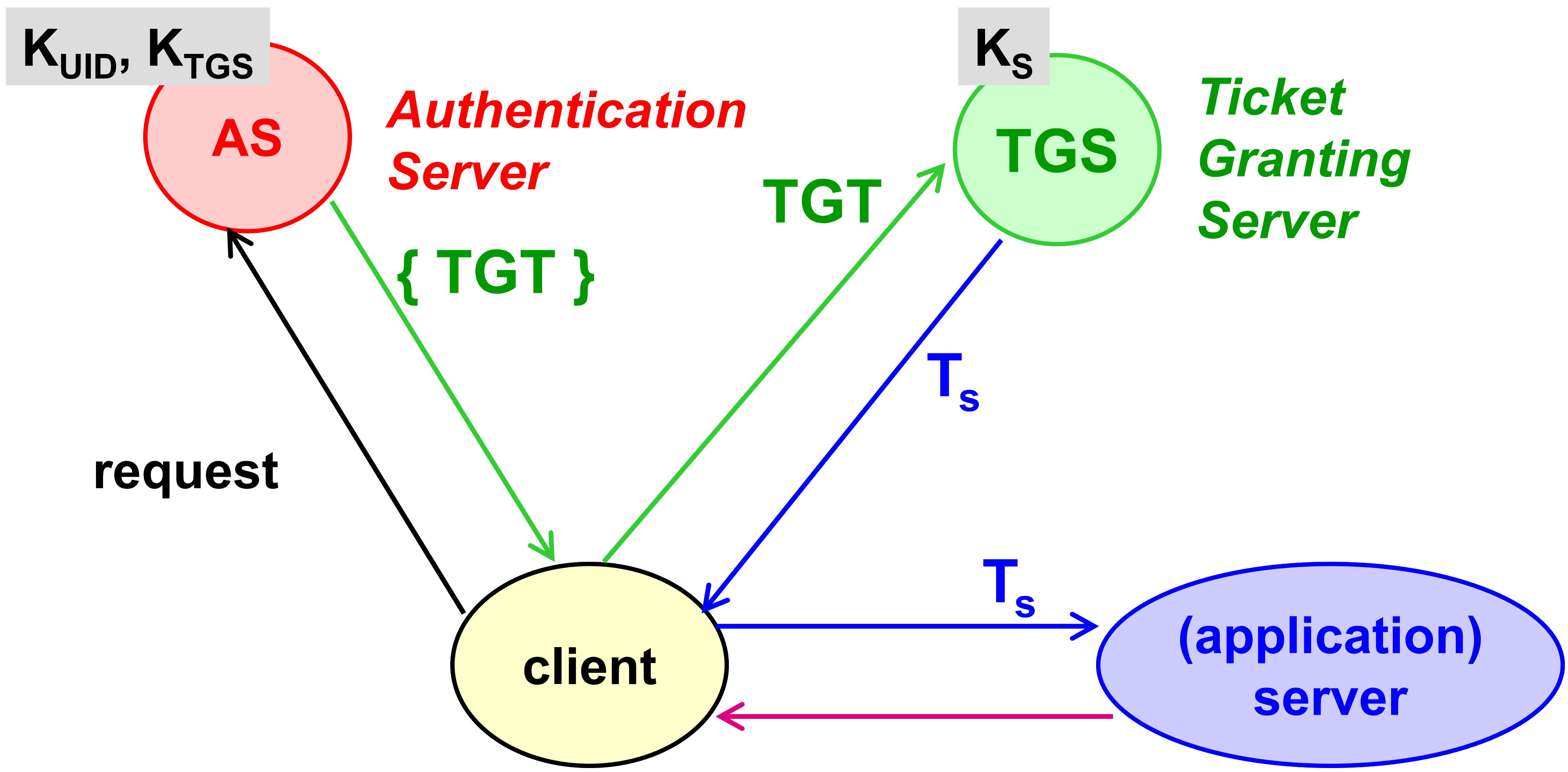
Kerberos

- **authentication system based on a TTP (Trusted Third Party)**
  - important for non-HTTP services
- **invented as part of the MIT project Athena**
- **user password never transmitted but only used <u>locally</u> as (symmetric) cryptographic key**
- ***realm* = Kerberos domain, that is the set of systems that use Kerberos as authentication system**
- ***credential* = user.instance@realm**

Kerberos

- **ticket**
  - data structure to authenticate a client to a server
  - variable lifetime
    (V4: max 21 hours = 5’ x 255)
    (V5: unlimited)
  - encrypted with the symmetric key of the target server
  - bound to the IP address of the client
  - bound to just one credential
- **client authentication compulsory**
  - server authentication optional

Kerberos high-level view



Kerberos: data formats (v4)

TICKET

server-id
client-id
client-address
timestamp
life
$K_{S,C}$

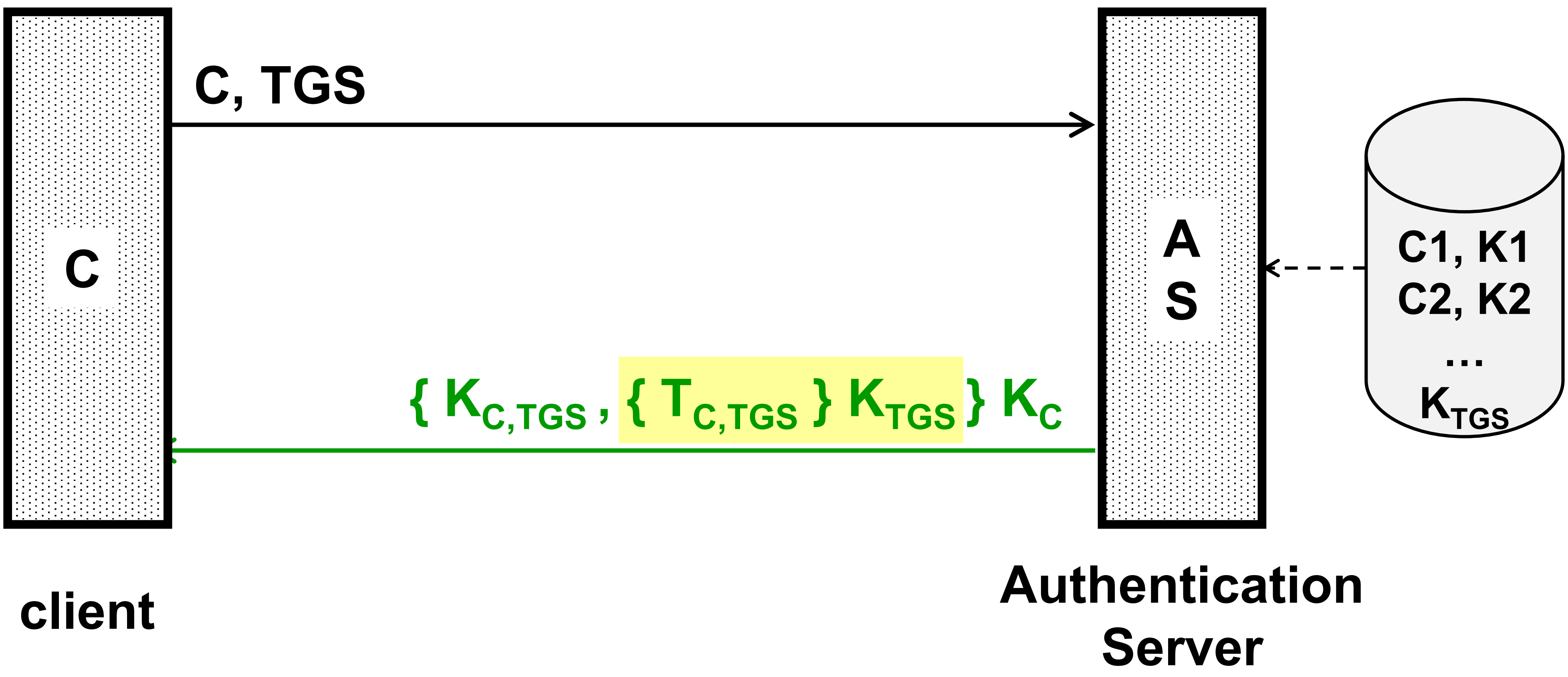
K_S

AUTHENTICATOR

client-id
client-address
timestamp

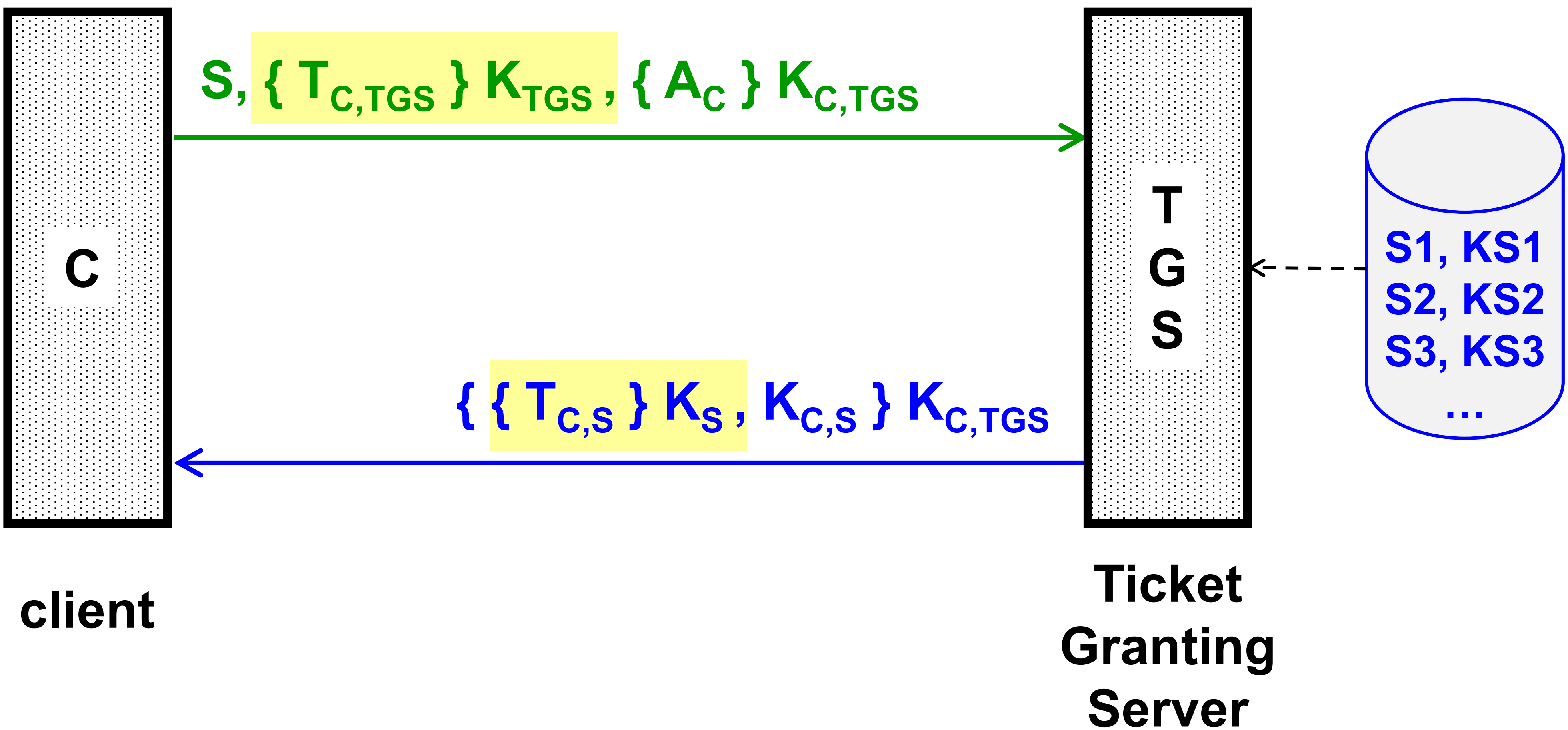
$K_{S,C}$

TGT request

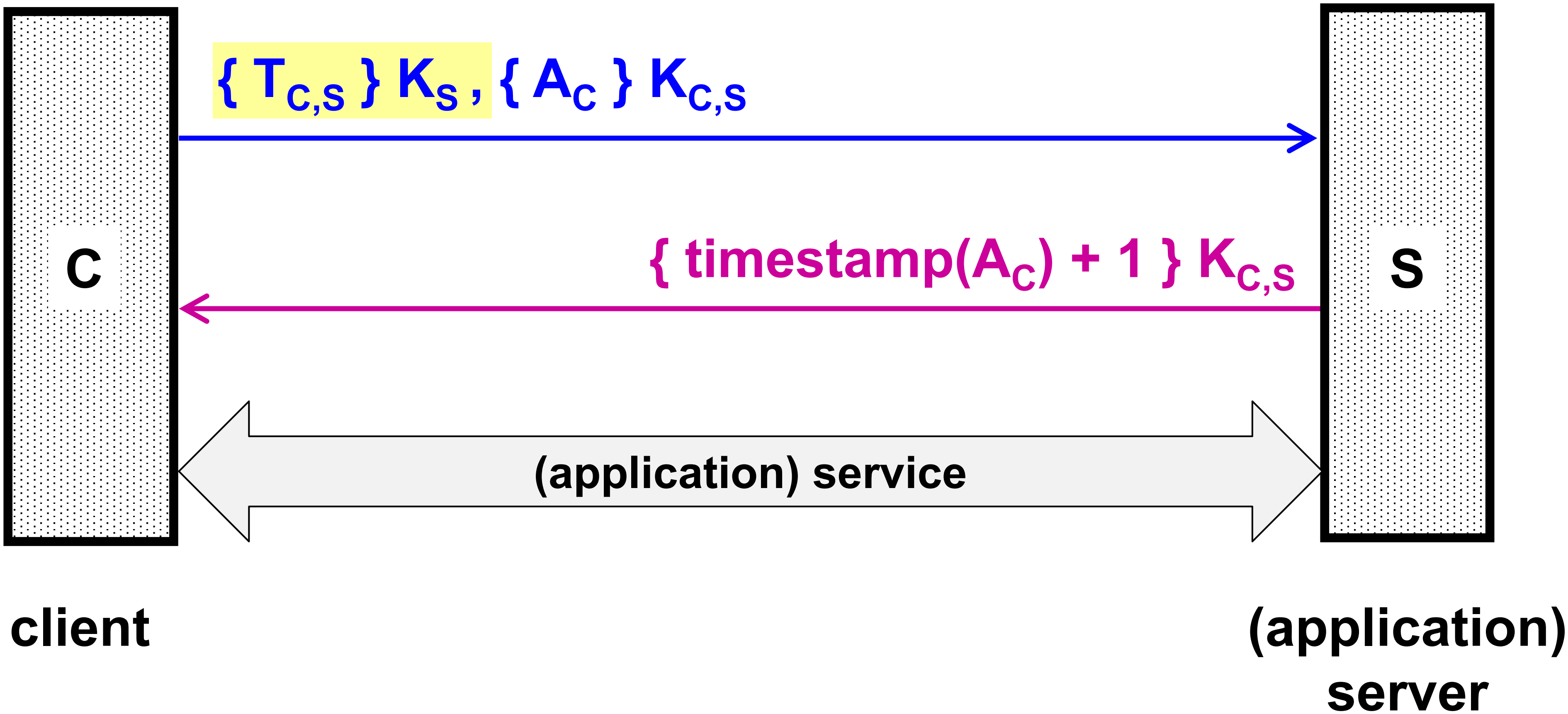


C, TGS
C
A S
C1, K1
C2, K2
...
K TGS
{ K C,TGS , { T C,TGS } K TGS } K C
client
Authentication Server

Ticket request



Ticket use



$\{ T_{C,S} \} K_S , \{ A_C \} K_{C,S}$
C
S
$\{ timestamp(A_C) + 1 \} K_{C,S}$
(application) service
client
(application) server

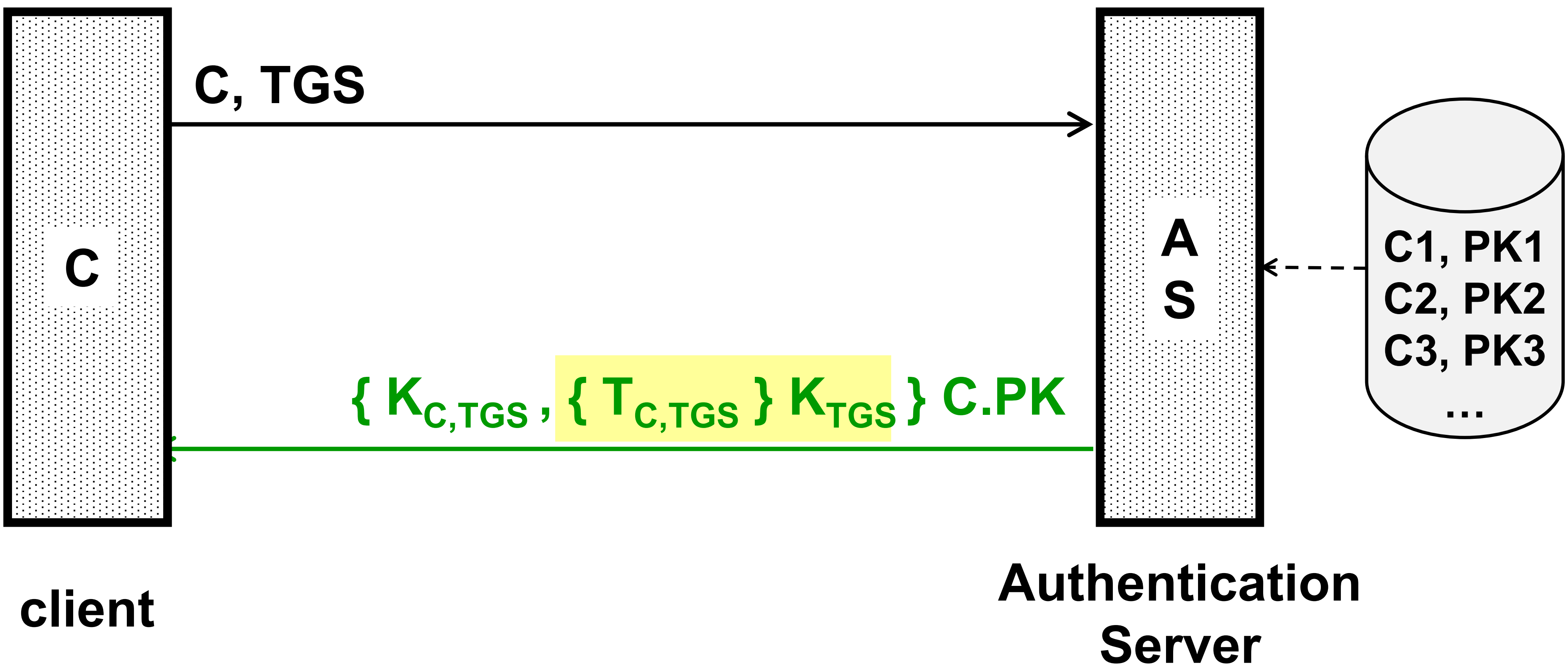
Kerberos versions and usage

- **MIT V4**
  - the original one
- **MIT V5 (also RFC-1510)**
  - not only DES
  - extended ticket lifetime (begin-end)
  - inter-realm authentication
  - forwardable ticket
  - extendable ticket
- **single login to all Kerberized services**
  - K-POP, K-NFS, K-LPD, K-telnet, K-ftp, K-dbms
  - services in a Windows domain (MS has adopted Kerberos* since Windows-2000)

Kerberos v5

- **RFC-4120 (obsoletes RFC-1510)**
- **algorithm flexibility**
  - client and servers may support different algorithms
  - originally it was DES-CRC32
  - then 3DES, RC4, AES, Camellia and MD4, MD5
- **pre-authentication**
  - to prevent pwd enumeration or dictionary attacks on the TGT
  - e.g in Windows, the AS_REQ must contain enc(K_C, T)
- **support for asymmetric crypto (in AS_REQ only)**

TGT request with PKINIT

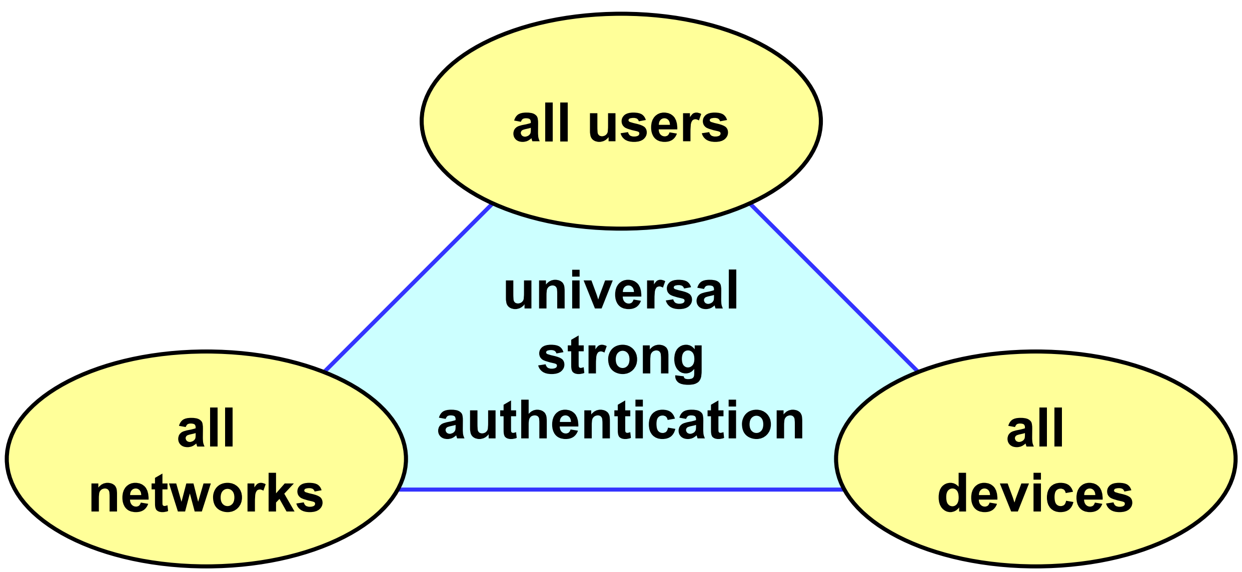


SSO (Single Sign-On)

- **the user has a single "credential" to authenticate himself and access any service in the system**
- **fictitious SSO:**
  - client for automatic password synchronization / management (alias "password wallet")
  - specific for some applications only
- **integral SSO:**
  - multiapplication authentication techniques (e.g. asymmetric CRA, Kerberos)
    - likely requires a change in the applications
  - multi-domain SSO (e.g. with SAML tokens, that generalize Kerberos tickets)

Authentication interoperability

- OATH (www.openauthentication.org)
- interoperability of authentication systems based on OTP, symmetric or asymmetric challenge
- development of standards for the client-server protocol and the data format on the client



OATH specifications

- **http://www.openauthentication.org/specifications**
- **HOTP (HMAC OTP, RFC-4226)**
- **TOTP (Time-based OTP, RFC-6238)**
- **OATH challenge-response protocol (OCRA, RFC-6287)**
- **Portable Symmetric Key Container (PSKC, RFC-6030)**
  - XML-based key container for transporting symmetric keys and key-related meta-data
- **Dynamic Symmetric Key Provisioning Protocol (DSKPP, RFC-6063)**
  - client-server protocol for provisioning symmetric keys to a crypto-engine by a key-provisioning server

HOTP

- K : shared secret key
- C : counter (monotonic positive integer number)
- h : cryptographic hash function (default: SHA1)
- sel : function to select 4 bytes out of a byte string
- HOTP(K,C) = sel(HMAC-h(K,C)) & 0x7FFFFFFF
- note: the mask 0x7FFFFFFF is used to set MSB=0 (to avoid problems if the result is interpreted as a signed integer)
- to generate a N digits (6-8) access code:

  HOTP-code = HOTP(K,C) mod 10^N

TOTP

- **as HOTP but the counter C is the number of intervals TS elapsed since a fixed origin T0**

  $C = (T - T0) / TS$

- **default (RFC-6238):**
  - T0 = Unix epoch (1/1/1970)
  - T = unixtime(now)
    seconds elapsed since the Unix epoch
  - TS = 30 seconds
  - … equivalent to C = floor (unixtime(now) / 30)
  - h = SHA1 (buy MAY use SHA-256 or SHA-512)
  - N = 6

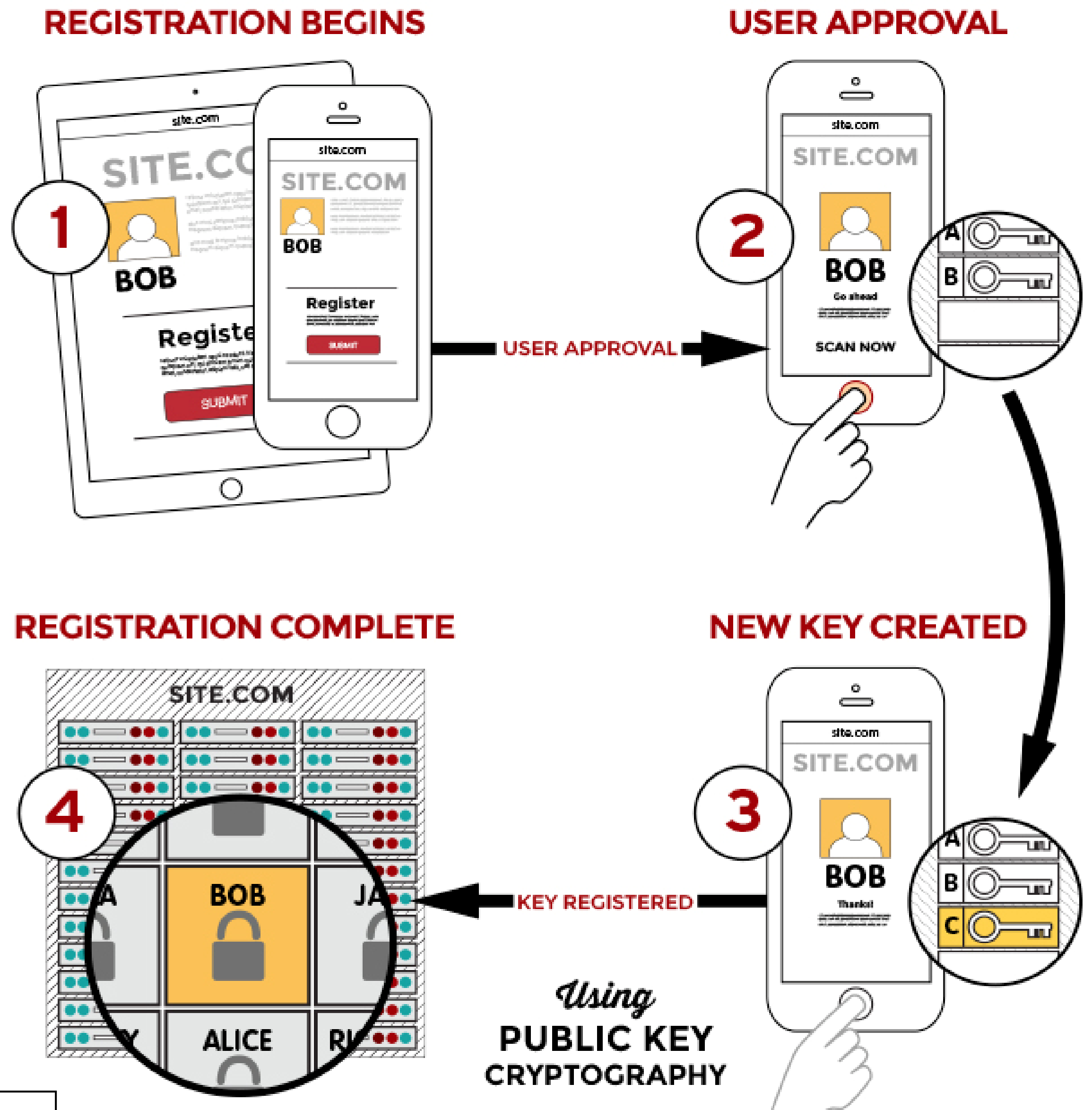
Google authenticator

- **supports HOTP and TOTP with the following assumptions:**
  - K is provided base-32 encoded
  - C is provided as uint_64
  - sel(X)
    - offset = 4 least-significant-bits of X
    - return X[offset … offset+3]
  - TS = 30 seconds
  - N = 6
  - if the generated code contains less than 6 digits then it's left padded with zeroes (e.g. 123 > 000123)

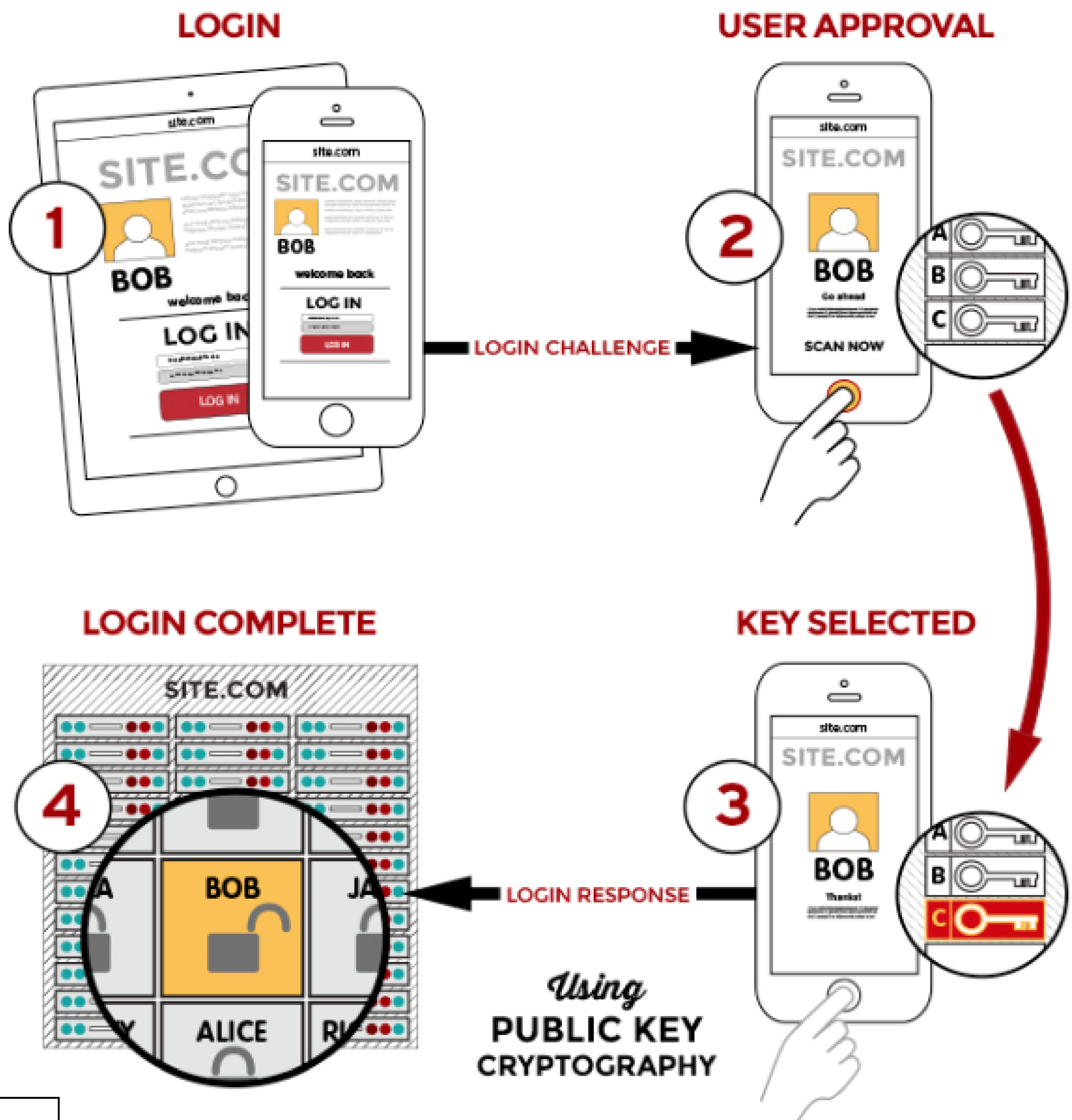
FIDO

- **Fast IDentity Online**
- **industry standard of the FIDO Alliance for:**
  - biometric authN = passwordless user experience
  - 2-factor authN = 2nd factor user experience
- **based on personal devices capable of asymmetric crypto**
  - for responding to an asymmetric challenge
  - for digital signature of texts
- **UAF = Universal Authentication Framework**
- **U2F = Universal 2nd Factor**
- **ASM = Authenticator-Specific Module**
- **available for major services (Google, Dropbox, GitHub, Twitter, …) and also for the cloud (GCP, AWS, Azure, …)**

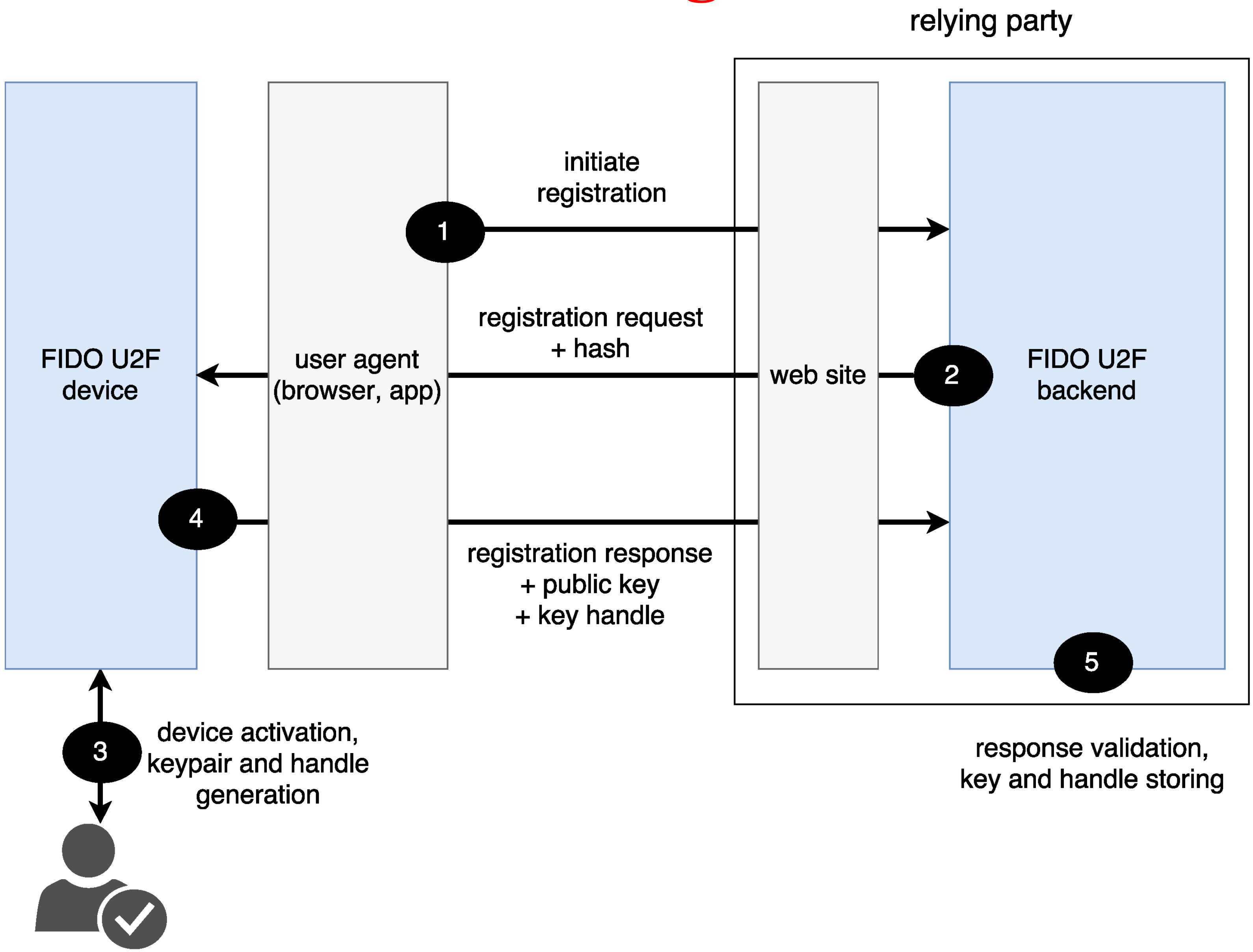
FIDO registration



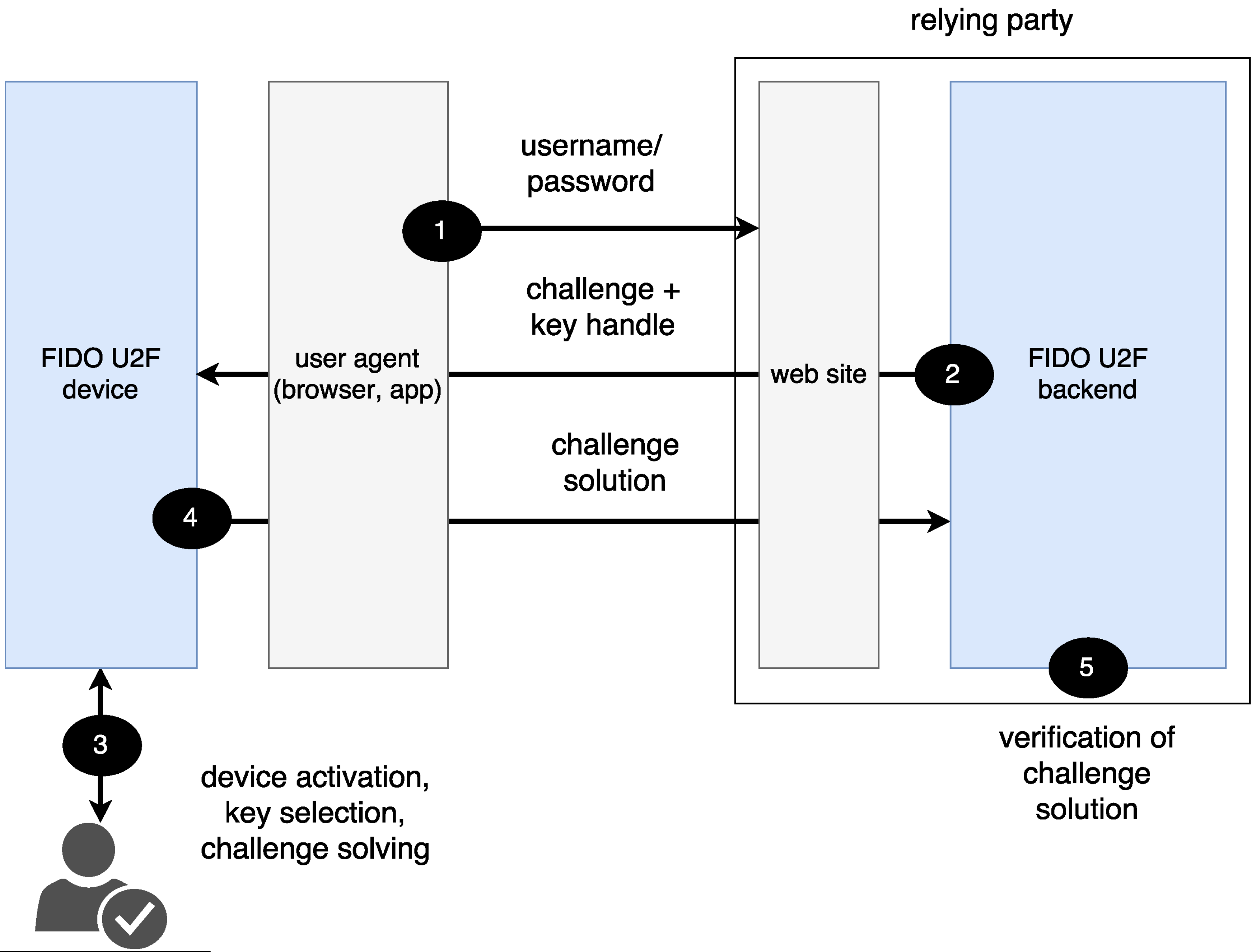
FIDO Login


LOGIN
USER APPROVAL
1
BOB
welcome back
LOG IN
2
BOB
SCAN NOW
A
B
C
LOGIN CHALLENGE
LOGIN COMPLETE
KEY SELECTED
SITE.COM
4
BOB
ALICE
3
BOB
LOGIN RESPONSE
Using
PUBLIC KEY
CRYPTOGRAPHY

FIDO U2F registration



FIDO U2F authentication



FIDO: other characteristics

- **biometric techniques**
  - local authentication method to enable the FIDO keys stored on the user device
- **secure transactions**
  - digital signature of a transaction text (in addition to the response to the challenge)
- **FIDO backend (or server)**
  - to enable the use of FIDO on an application server
- **FIDO client**
  - to create and manage credentials FIDO on a user device

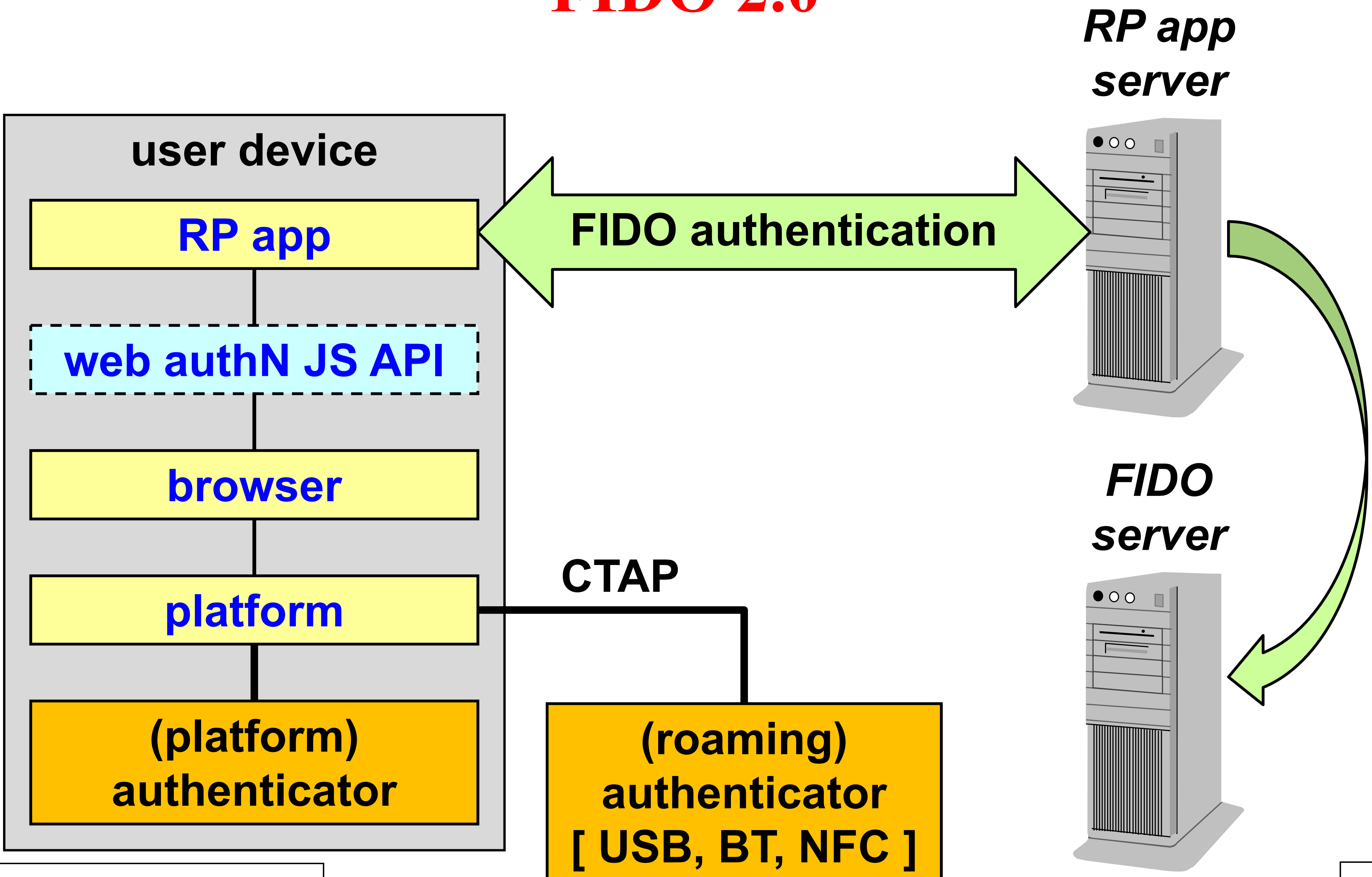
FIDO: security and privacy

- **strong authentication (asymmetric cryptography)**
- **no 3rd party in the protocol**
- **no secrets on the server side**
- **biometric data (if used) never leave user device**
- **no phishing because authN response can't be reused:**
  - it's a signature over various data, including the RP identity
- **since one new key-pair is generated at every registration, we obtain no link-ability among:**
  - different services used by the same user
  - different accounts owned by the same user
- **there is no limit because private keys are not stored in the authenticator but recomputed as needed based on an internal secret and RP identity**

Fido: evolution

- Feb.2013: FIDO alliance launched
- Dec.2014: FIDO v1.0
- Jun.2015: Bluetooth and NFC as transport for U2F
- Nov.2015: submission to W3C of the Web API for accessing FIDO credentials
- Feb.2016: W3C creates the Web Authentication WG to define a client-side API that provides strong authentication functionality to Web Applications, based on the FIDO Web API
- Nov.2017: FIDO v2.0

FIDO 2.0



RP app server
user device
RP app
web authN JS API
browser
platform
(platform) authenticator
FIDO authentication
CTAP
(roaming) authenticator [USB, BT, NFC]
FIDO server

FIDO 2.0: some details

- **CTAP = Client To Authenticator Protocol**
- **the platform (bound, internal) authenticators are cryptographic elements (more or less secure) able to store (and use) asymmetric keys**
  - packed attestation = authenticator with limited resources (e.g. Secure Element)
  - TPM attestation = TPM as cryptographic element
  - Android Key attestation = authenticator of Android Nougat
  - Android SafetyNet attestation = authenticator of Android via SafetyNet API
  - FIDO U2F attestation = authenticator FIDO U2F using the FIDO-U2F Message Format
- **being extended for authN of IoT devices**