Secure On-boarding of Data To Smart Contract While Maintaining Privacy in Solidity

2FA API

AWS/
alternative VPS

Chainlink Node

Smart Contract

Verifier Data Update Request Mapping:

PBK_user => PBK_verifier => nonce => (Encrypted Data, Encrypted Key, Hashes, Time Limit)

Public update functions:

Update functions asserted by name to be called off-chain on data d to transform to d' and verify that up(d, a*) == d'

Update Function Permission Mapping:

PBK_verifier => function name as string => boolean

User Keychain Mapping:

PBK_user => h_key

Chainlink calls

Update User Data Function

Validate User through Chainlink, Check request mapping, check ZoKrates proof, store updated data, emit Event (approved || refused), store (IPFS CID, hash) if IPFS is leveraged

ZoKrates - for users

```
function(

    public field PBK, private field u,
    private field d', private field up,
    private field ar, private field v,
    public field o, public field a,
    public field c, public field h_key,
    public field h_ru, public field h_da,
    public field h_dp, public field h_ipfs_d) {

    prove that:

        1. h_key   == hash(PBK++u)
        2. h_ru    == hash(d' ++ up()
        ++ar++v)
        3. o       == hash(v ++ u)
        4. a       == encrypt_AES(d', u)
        5. h_da    == hash(d' ++ a ++ u)
        6. h_dp    == hash(d')
        7. h_ipfs_d == hash(a ++ o ++ c)

}
```

In the smart contract:

a. Assert that on-chain hash == h_key computed off-chain provided to smart contract by user when onboarding in User Keychain Mapping for PBK
b. Assert that h_ru == hash submitted in request by public service verifier
c. Assert that block timestamp <= t_0 + t_limit [verify times are feasible when requests are onboarded by verifiers through smart contract] [store t_0 as block.timestamp when request is placed into smart contract]
d. Assert that on-chain hash == h_dp computed off-chain provided to smart contract by verifier

In ZoKrates:

1. Verifies that u is valid
2. Verifies that d' and v are valid
3. Verifies o is valid
4. Verifies a is valid
5. Verifies provided hash to contract for (d' ++ a ++ u) is valid (helpful later when proving ownership of data)
6. Three separate ZKP circuits are used for this function to lower computational cost - one for conditions 1/2/3/5, one for 1/4/6, and one for 7. Common parameters are matched between these functions in the smart contract to verify that in combination they are equivalent to the pseudocode outlined. Redundancies are present because only some variables are public, others are hidden.
7. Proves that provided hash h_ipfs_d of content at CID c uploaded on IPFS is valid

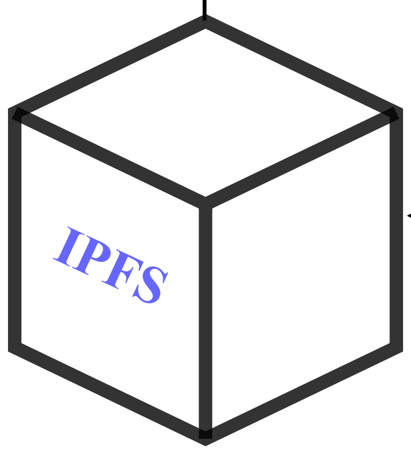
Vera User

Update Approval Function

0. Scan for emitted Events/Request Mapping
1A. Obtain Request data = (e_ru, k', h_dp, h_ru, t-limit) [from SC/IPFS]
1B. decrypt_RSA(k', PVK_RSA_user) = k
1B. decrypt_AES(e_ru, k) = ru = d' ++ up() ++ ar ++ v
2. Validate that up() is permitted operation for verifier through smart contract and that h_ru = hash(ru)
3. Validate up(d, *ar) == d' | where d is data for v w/SC function
4. Compute a = encrypt_AES(d', u)
5. Compute o = hash(v ++ u), h_da = hash(d' ++ a ++ u)
6. Compute h_key = hash(PBK_user ++ u)
7. Send parameters to ZoKrates and smart contract

IPFS:

8. Place (a, o) on IPFS @ CID = c
9. Compute hash(a++o++c) = h_ipfs_d
10. Send parameters to ZoKrates and smart contract with extension that h_ipfs_d = hash of valid params to guarantee that hash of data inside and CID is on blockchain for all readers



Random Public Service Verifier

Nonce Generator

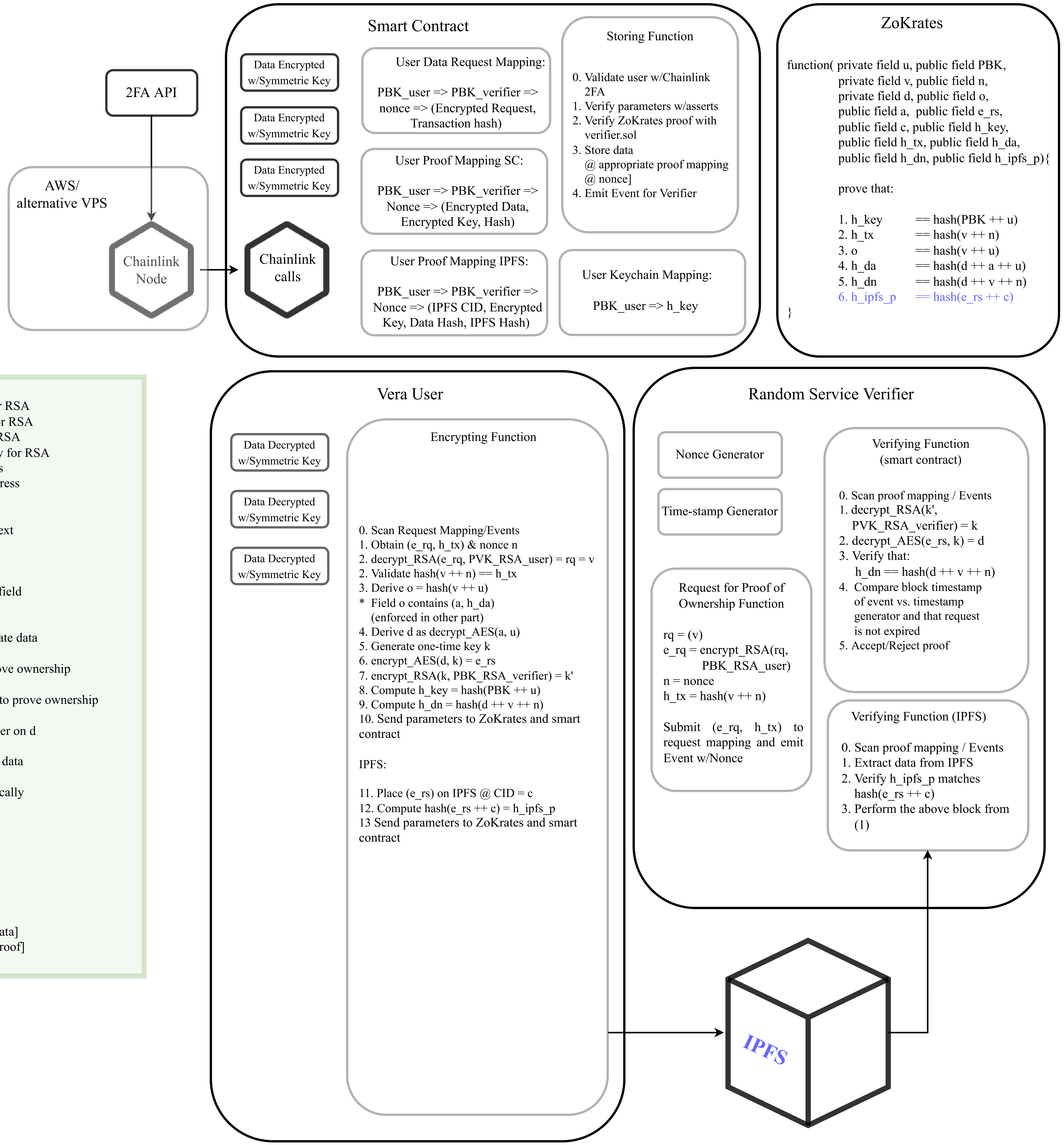
Data On-boarding Function

d' = new data padded to 64 bytes
up() = update operation on variable/field
ar = arguments to update function
v = variable/field(s) requested to update
t-limit = time limit for verification
ru = d' ++ up() ++ ar ++ v
h_ru = hash(ru)
h_dp = hash(d')
k = one-time key

1. encrypt_AES(ru, k) = e_ru
2. encrypt_RSA(k, PBK_RSA_user) = k'
3. Place (e_ru, k', h_dp, h_ru, t-limit) into Verifier Data Update Request Mapping at transaction Nonce; store t_0 as block.timestamp
4. Emit Event For User

PBK_RSA_user	:= user's public key for RSA
PVK_RSA_user	:= user's private key for RSA
PBK_RSA_verifier	:= verifier's public key for RSA
PVK_RSA_verifier	:= verifier's private key for RSA
PBK_user:	:= user's public address
PBK_verifier:	:= verifier's public address
u	:= symmetric key of user
v	:= variable field requested ("age"), in plaintext
n	:= transaction nonce (10 byte string)
o	:= obfuscated field ("age" -> "v2m0t97y")
d	:= user data stored at some field
d'	:= user data requested to be stored at some field
a	:= d' encrypted symmetrically with AES
c	:= IPFS content identifier (URI/URL)
e_ru	:= symmetrically encrypted request to update data
ru	:= request to update data
e_rq	:= asymmetrically encrypted request to prove ownership
rq	:= request to prove ownership
e_rs	:= symmetrically encrypted response to prove ownership
rs	:= response to prove ownership
up()	:= update function requested by verifier on d
ar	:= arguments to update function
t_limit	:= the time limit for request to update data
k	:= one-time key for encryption
k'	:= one-time key encrypted asymmetrically
h_ru	:= hash(d' ++ up() ++ ar ++ v ++ t_limit)
h_dp	:= hash(d')
h_key	:= hash(PBK_user ++ u)
h_tx	:= hash(v ++ n)
h_dn	:= hash(d ++ v ++ n)
o	:= hash(v ++ u)
h_da	:= hash(d' ++ a ++ u)
h_ipfs_d	:= hash(a ++ o ++ c) [d for data]
h_ipfs_p	:= hash(e_rs ++ c) [p for proof]

Proving Ownership of Encrypted Data In Smart Contract While Maintaining Privacy in Solidity



PBK_RSA_user	:= user's public key for RSA
PVK_RSA_user	:= user's private key for RSA
PBK_RSA_verifier	:= verifier's public key for RSA
PVK_RSA_verifier	:= verifier's private key for RSA
PBK_user:	:= user's public address
PBK_verifier:	:= verifier's public address
u	:= symmetric key of user
v	:= variable field requested ("age"), in plaintext
n	:= transaction nonce (10 byte string)
o	:= obfuscated field ("age" -> "v2m0t97y")
d	:= user data stored at some field
d'	:= user data requested to be stored at some field
a	:= d' encrypted symmetrically with AES
c	:= IPFS content identifier (URI/URL)
e_ru	:= symmetrically encrypted request to update data
ru	:= request to update data
e_rq	:= asymmetrically encrypted request to prove ownership
rq	:= request to prove ownership
e_rs	:= symmetrically encrypted response to prove ownership
rs	:= response to prove ownership
up()	:= update function requested by verifier on d
ar	:= arguments to update function
t_limit	:= the time limit for request to update data
k	:= one-time key for encryption
k'	:= one-time key encrypted asymmetrically
h_ru	:= hash(d' ++ up() ++ ar ++ v ++ t_limit)
h_dp	:= hash(d')
h_key	:= hash(PBK_user ++ u)
h_tx	:= hash(v ++ n)
h_dn	:= hash(d ++ v ++ n)
o	:= hash(v ++ u)
h_da	:= hash(d' ++ a ++ u)
h_ipfs_d	:= hash(a ++ o ++ c) [d for data]
h_ipfs_p	:= hash(e_rs ++ c) [p for proof]

In the smart contract:

a. Assert that on-chain hash == h_key computed off-chain provided to smart contract by user when onboarding in User Keychain Mapping for PBK
b. Assert that on-chain hash == h_tx computed off-chain provided to smart contract by public service verifier
c. Assert that User Data Mapping at (PBK_user) => o => (a, h_da)

In ZoKrates:

1. Therefore the symmetric key u provided is valid
2. Therefore v (and n) is valid and request is decrypted correctly [coupled with assertion (b)]
3. Therefore o is valid, the obfuscated field from which we are retrieving data
4. Therefore (d, a) is valid [one MUST onboard with valid hash h_da]
5. Therefore h_dn is a valid hash of d with v and n that can be used to verify that the encrypted response e_rs is valid upon decryption to rs
6. Proves that provided hash h_ipfs_p of content at CID c uploaded on IPFS is valid

In the smart contract:

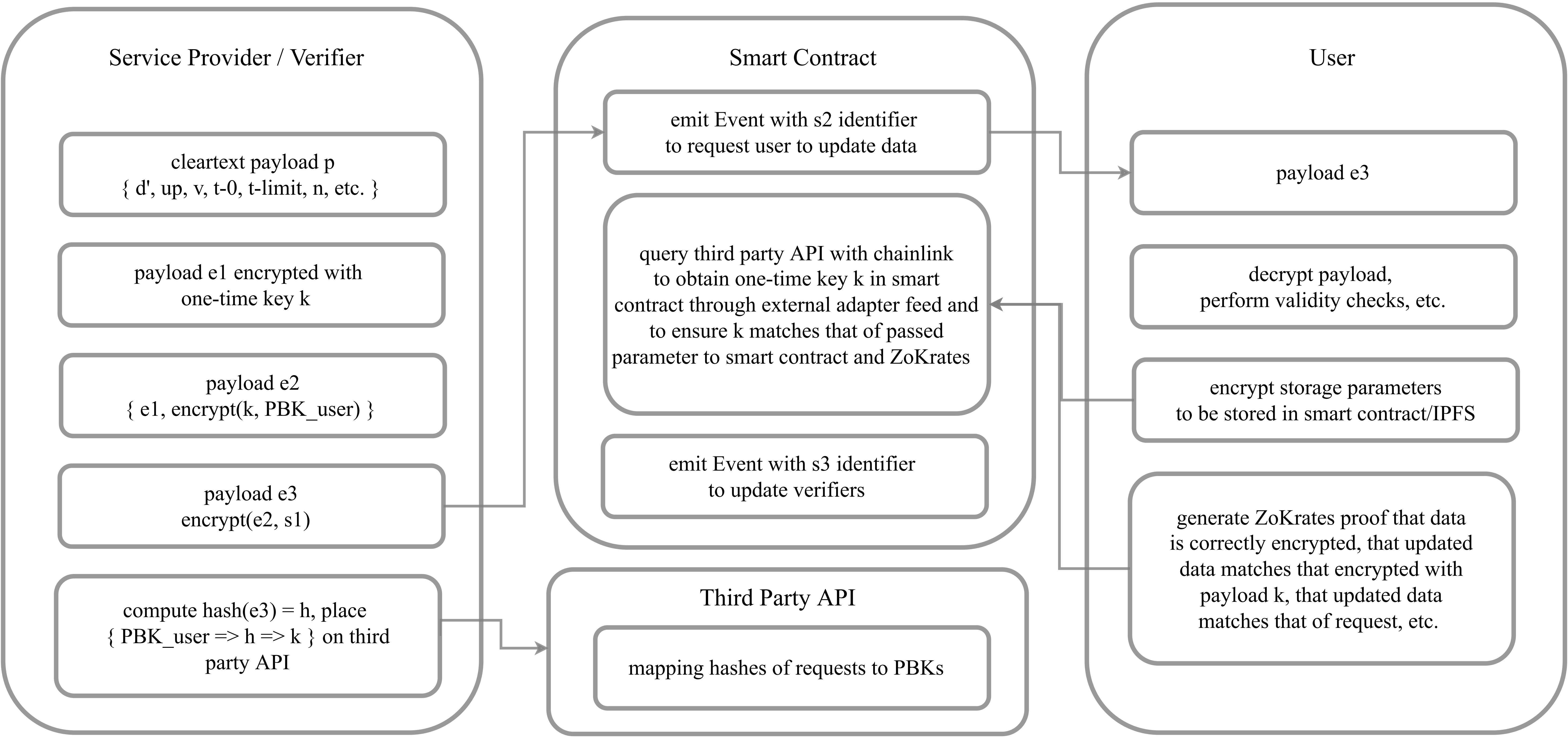
1. Place (e_rs, k', h_dn) into smart contract w/storing function
2. Place (IPFS hash [c], k', h_dn, h_ipfs_p) into smart contract w/storing function

Off-chain:

1. a. Extract (e_rs, k', h_dn, h_ipfs_p) from smart contract/IPFS CID
   b. Verify hash(e_rs ++ c) == h_ipfs_p
2. decrypt_RSA(k', PVK_RSA_verifier) = k
3. a. decrypt_AES(e_rs, k) = d
   b. Verify that h_dn == hash(d ++ v ++ n)
   c. Compare timestamps of proof
4. Decide to accept or reject proof

Accounting For Side-Channel Inference: An Implementable Extension With Chainlink and/or Asymmetric Encryption Verification Components To Hide Vera Request and Response Receivers For Proof of Valid Storage When On-boarding

*The requests, responses, and fields remain obfuscated regardless of whether this extension is present,



Assume a number of negotiated shared secrets { s1, s2, s3, ... } between user and verifier are available negotiated through DH key exchange, secondary secure channels, etc.

Accounting For Side-Channel Inference: An Implementable Extension With Diffie-Hellman Exchange And/Or Secondary Secure Channel To Hide Vera Request and Response Receivers For Proof of Ownership

*The requests, responses, and fields remain obfuscated regardless of whether this extension is present,