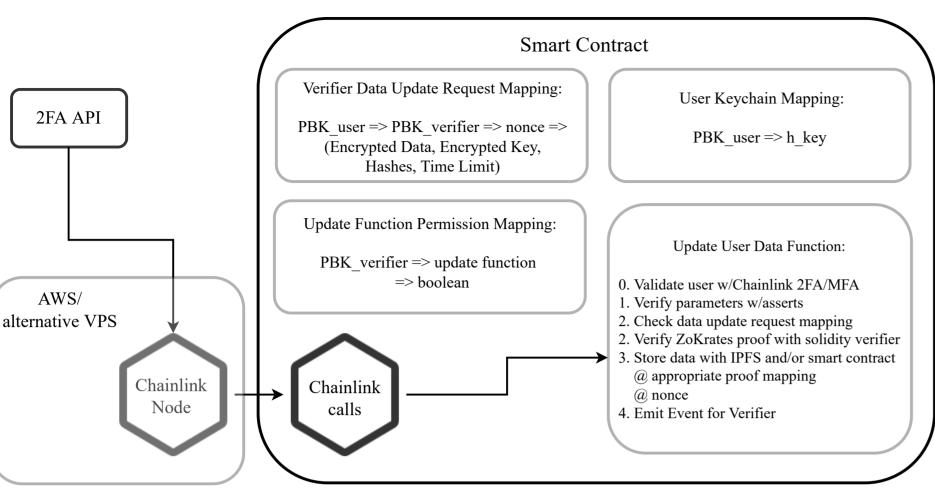
Secure On-boarding of Data To Smart Contract While Maintaining Privacy in Solidity



```
ZoKrates - for users
function(
     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(u)
     2. h ru == hash(d' ++ up() ++ ar ++ v)
              == hash(v ++ u)
               == 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 user
- b. Assert that h_ru == hash submitted in request by public service verifier
- c. Assert that block timestamp $\leq t_0 + t_{\text{limit}}$ [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() ++
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 = \text{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 up() = update operation on variable/field ar = arguments to update function v = variable/field(s) requested to update

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'
- Place (e_ru, k', h_dp, h_ru, t-limit) into Verifier Data Update Request Mapping at transaction Nonce; with 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
                        := verifier's public address
PBK verifier:
u := symmetric key of user
v := variable field requested ("age"), in plaintext
n := transaction nonce (10 byte string)
o := obfuscated field ("age" -> "v2m0t97y")
   := user data stored at some field
   := user data requested to be stored at some field
   := d' encrypted symmetrically with AES
c := IPFS content identifier (URI/URL)
e ru := symmetrically encrypted request to update data
ru := request to update data
         := asymmetrically encrypted request to prove ownership
e_rq
         := request to prove ownership
rq
         := symmetrically encrypted response to prove ownership
e rs
         := response to prove ownership
rs
         := update function requested by verifier on d
up()
         := arguments to update function
ar
         := the time limit for request to update data
t limit
         := one-time key for encryption
         := one-time key encrypted asymmetrically
h ru := hash(d' ++ up() ++ ar ++ v)
h dp
         := hash(d')
h key
         := hash(u)
         := hash(v ++ n)
         := hash(d ++ v ++ n)
         := hash(v ++ u)
         := hash(d' ++ a ++ u)
h da
```

[d for data]

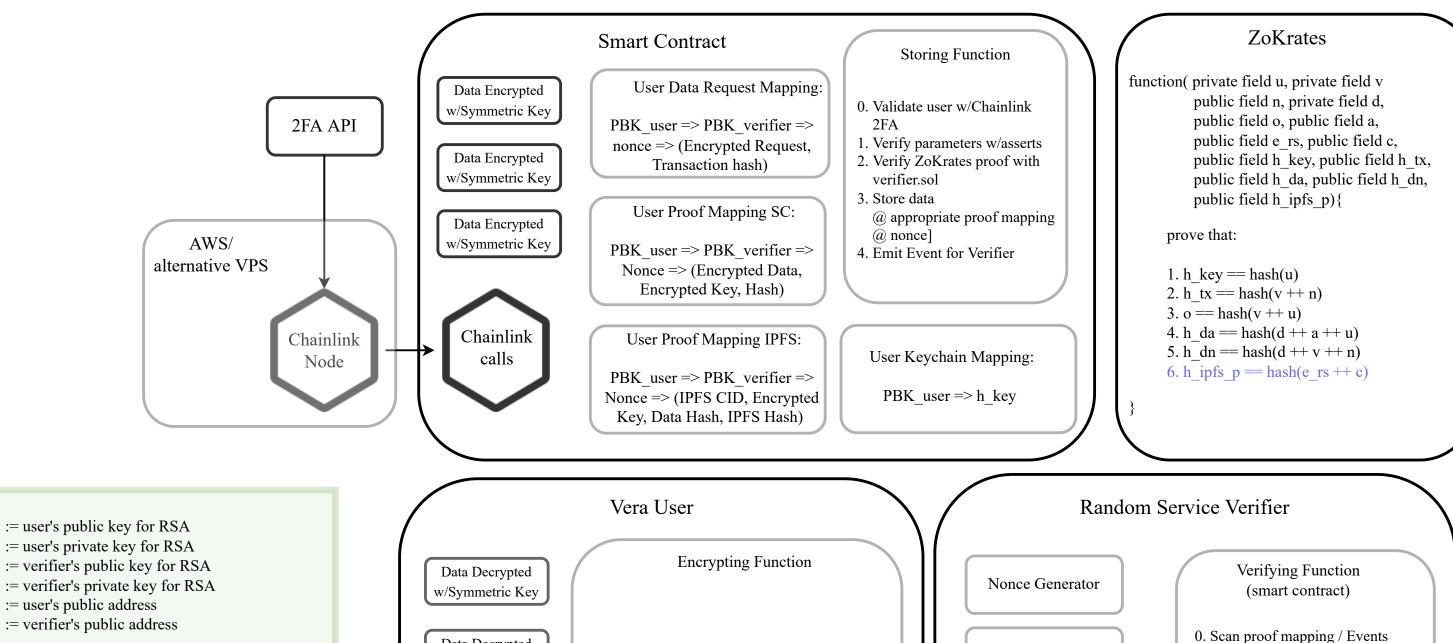
[p for proof]

h ipfs d := hash(a ++ o ++ c)

h ipfs p := hash(e rs ++ c)

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

IPFS



PVK RSA verifier := verifier's private key for RSA PBK user: := user's public address := verifier's public address PBK verifier: 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 := asymmetrically encrypted request to prove ownership := request to prove ownership := symmetrically encrypted response to prove ownership := response to prove ownership := update function requested by verifier on d up() := arguments to update function ar := the time limit for request to update data := one-time key for encryption := one-time key encrypted asymmetrically h ru := hash(d' ++ up() ++ ar ++ v)= hash(d')h dp h key := hash(u)= hash(v ++ n):= hash(d ++ v ++ n):= hash(v ++ u):= hash(d' ++ a ++ u)h ipfs d := hash(a ++ o ++ c)[d for data]

[p for proof]

PBK RSA user

PVK RSA user

PBK RSA verifier

h ipfs p := hash(e rs ++ c)

Data Decrypted 1. decrypt_RSA(k', Time-stamp Generator w/Symmetric Key PVK RSA verifier) = k 0. Scan Request Mapping/Events 2. decrypt AES(e rs, k) = d1. Obtain (e rq, h tx) & nonce n 3. Verify that: 2. decrypt RSA(e rq, PVK RSA user) = rq = vData Decrypted h dn == hash(d ++ v ++ n)2. Validate hash(v ++ n) == h txw/Symmetric Key 4. Compare block timestamp 3. Derive o = hash(v ++ u)Request for Proof of * Field o contains (a, h da) of event vs. timestamp Ownership Function generator and that request (enforced in other part) is not expired 4. Derive d as decrypt AES(a, u) rq = (v)5. Accept/Reject proof 5. Generate one-time key k e_rq = encrypt_RSA(rq, 6. encrypt_AES $(d, k) = e_rs$ PBK RSA user) 7. encrypt RSA(k, PBK RSA verifier) = k' n = nonce8. Compute h key = hash(PBK ++ u)h tx = hash(v ++ n)9. Compute h dn = hash(d ++ v ++ n)Verifying Function (IPFS) 10. Send parameters to ZoKrates and smart Submit (e rq, h tx) to contract request mapping and emit 0. Scan proof mapping / Events Event w/Nonce 1. Extract data from IPFS IPFS: 2. Verify h ipfs p matches 11. Place (e rs) on IPFS (a) CID = c hash(e rs ++ c)3. Perform the above block from 12. Compute hash(e rs ++ c) = h ipfs p 13 Send parameters to ZoKrates and smart (1)

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_user
- 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 $(PBK_user) \Rightarrow o \Rightarrow (a, h_da)$

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

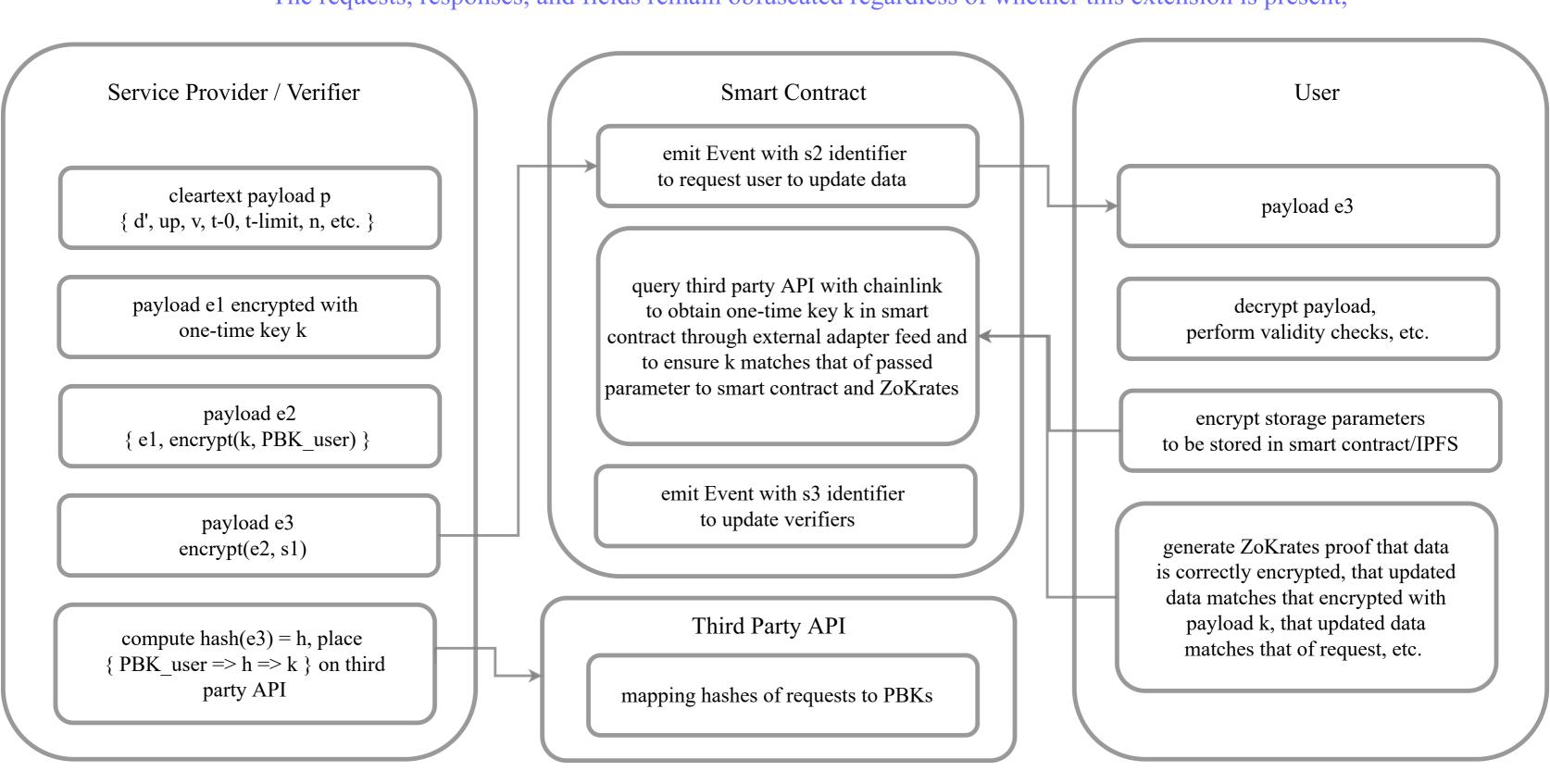
In the smart contract:

- 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
- 0. verify hash(c_1s ++ c) == h_lpis_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

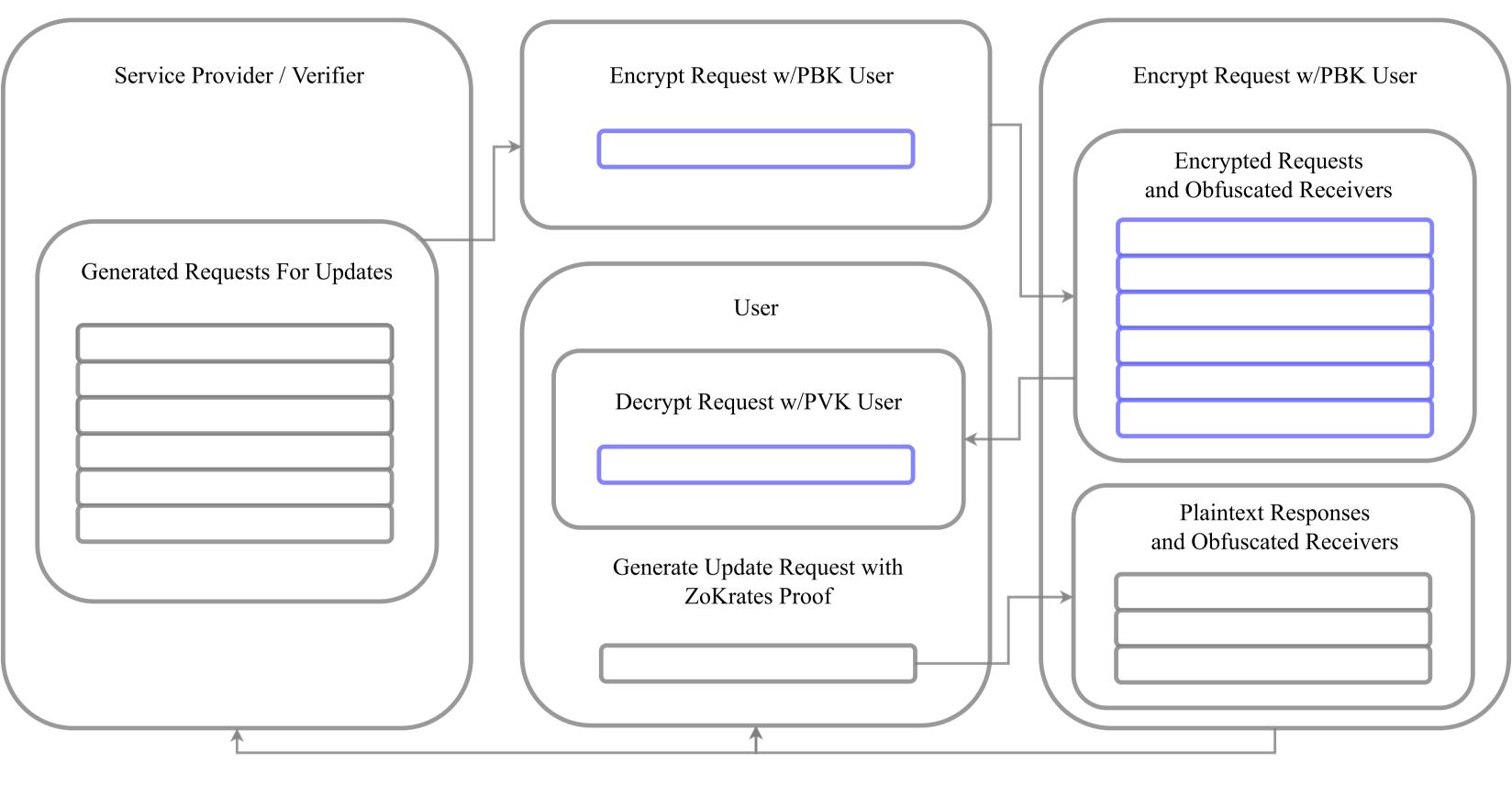
Accounting For Side-Channel Inference: An Implementable Extension With Chainlink and/or Asymmetric Encryption Verification Components To Hide Vera Request and Response Receivers For Proof of Valid Storage When On-boarding *The requests, responses, and fields remain obfuscated regardless of whether this 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,



Alert Each Other Of Available Request
Through Secure Channel Or Through
Emitting Events/Information With Shared
Secret In Smart Contract Such as With
SmartDHX Diffie-Hellman Key Exchange
(Using A Separately Negotiated DH Key for
Each Direction), etc.

Alternatively, The User/Verifier
Periodically Monitors The Blockchain
To Identify Matching Requests