Attacking the Linux PRNG on Android & Embedded Devices

David Kaplan, Sagi Kedmi, Roee Hay & Avi Dayan
IBM Security Systems

agenda

- Motivation and Introduction
- Linux Random Number Generator

agenda

- Motivation and Introduction
- Linux Random Number Generator
- Our Attack
- 1st Attack Vector – Local Atk.
- Demo
- 2nd Attack Vector – Remote Atk.

agenda

- Motivation and Introduction
- Linux Random Number Generator
- Our Attack
- 1st Attack Vector – Local Atk.
- Demo
- 2nd Attack Vector – Remote Atk.
- Mitigations

MOTIVATION

motivation_keystore_buffer_overflow

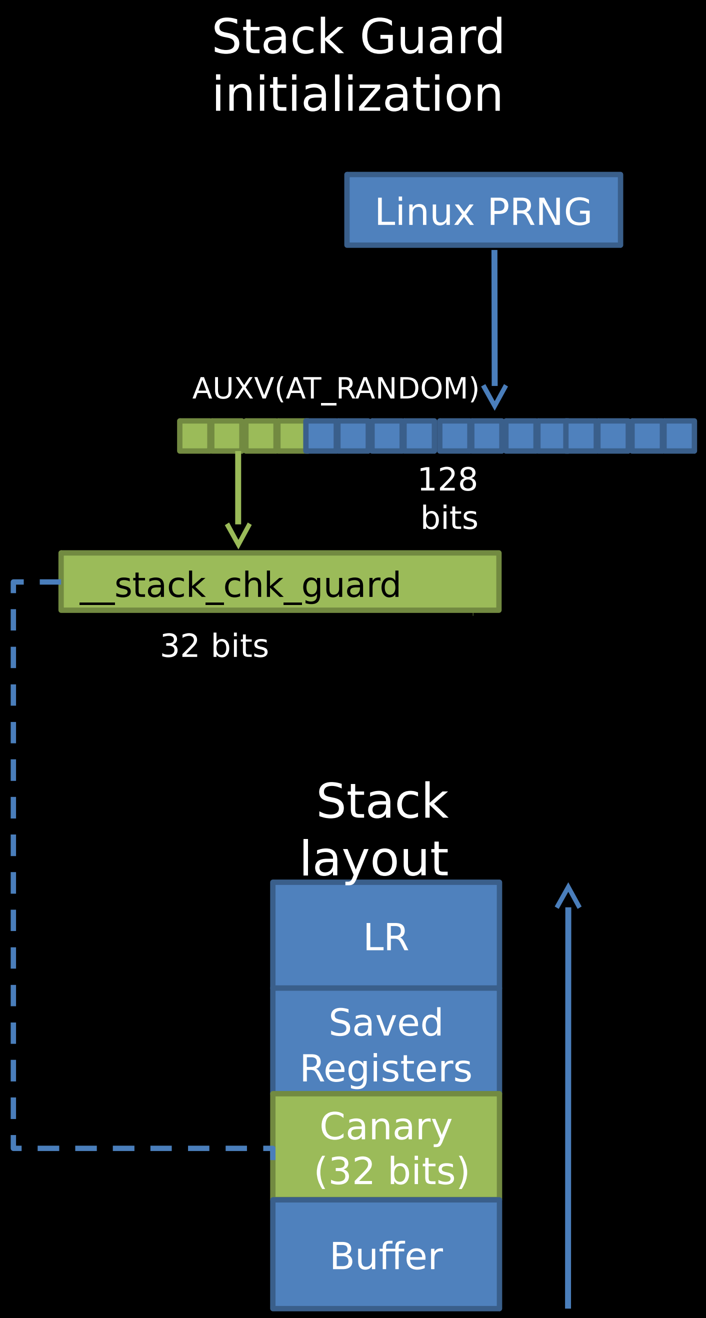
- We discovered CVE-2014-3100, a **stack-based Buffer Overflow** in Keystore
  - Service responsible for securely storing crypto related data
- We had privately reported to Google and they provided a patch available in *KITKAT*. Whitepaper.
- Exploit must overcome various defense mechanisms, including **Stack Canaries.**

```
/* KeyStore is a secured storage for key-value pairs. In this implementation,
 * each file stores one key-value pair. Keys are encoded in file names, and
 * values are encrypted with checksums. The encryption key is protected by a
 * user-defined password. To keep things simple, buffers are always larger than
 * the maximum space we needed, so boundary checks on buffers are omitted. */
```

motivation_keystore_buffer_overflow

Stack canaries and their enforcement

- On libbionic load:
  `__stack_chk_guard = *(uintptr_t *)getauxval(AT_RANDOM));`
- Function Prologue:
  `Place __stack_chk_guard on the stack (before ret).`
- Function Epilogue:
  `Compare saved stack canary with __stack_chk_guard;`
  `→ Crash if mismatch`



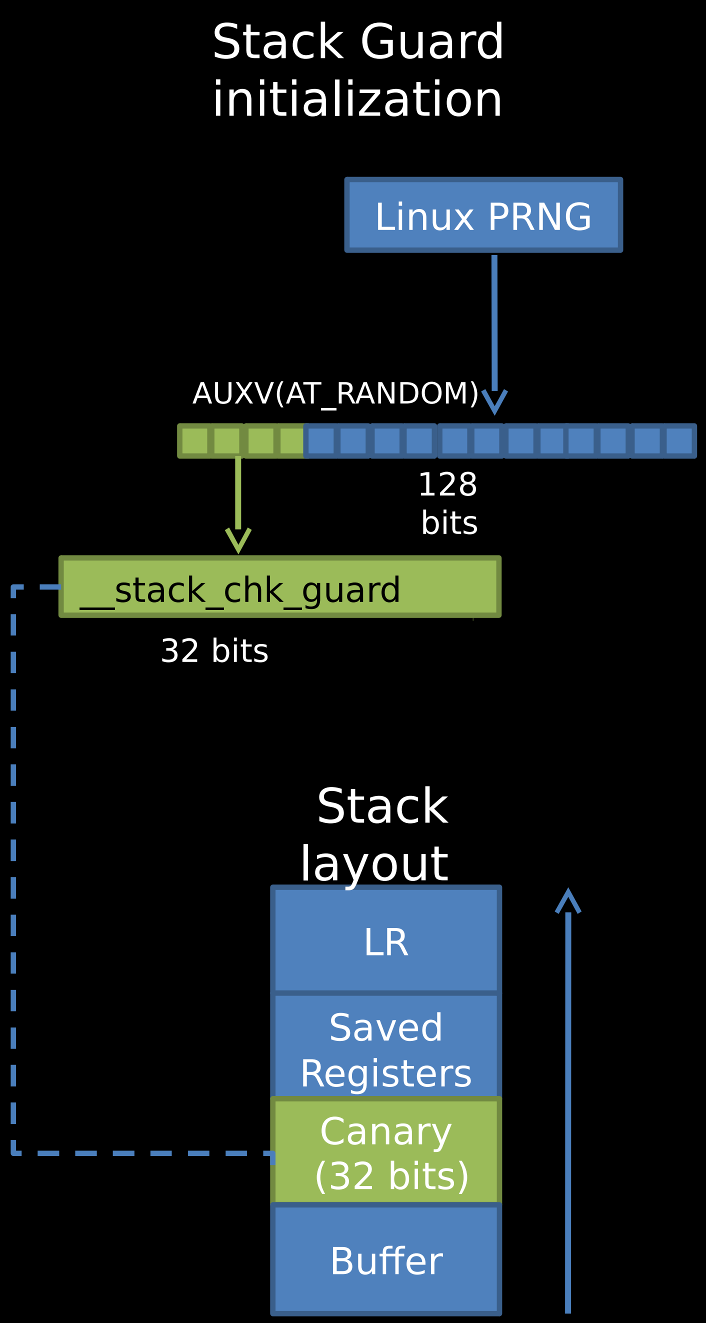
motivation_keystore_buffer_overflow

Stack canaries and their enforcement

- On libbionic load:
  `__stack_chk_guard = *(uintptr_t *)getauxval(AT_RANDOM));`
- Function Prologue:
  `Place __stack_chk_guard on the stack (before ret).`
- Function Epilogue:
  `Compare saved stack canary with __stack_chk_guard;`
  `→ Crash if mismatch`

Canary origins; *nix process creation model

- fork() → execve().
- execve() → Auxiliary vector (AUXV)
- AUXV[AT_RANDOM] = 16 Random bytes from the PRNG
- libbionic assigns canary = first 4 bytes of AT_RANDOM



motivation_keystore_buffer_overflow

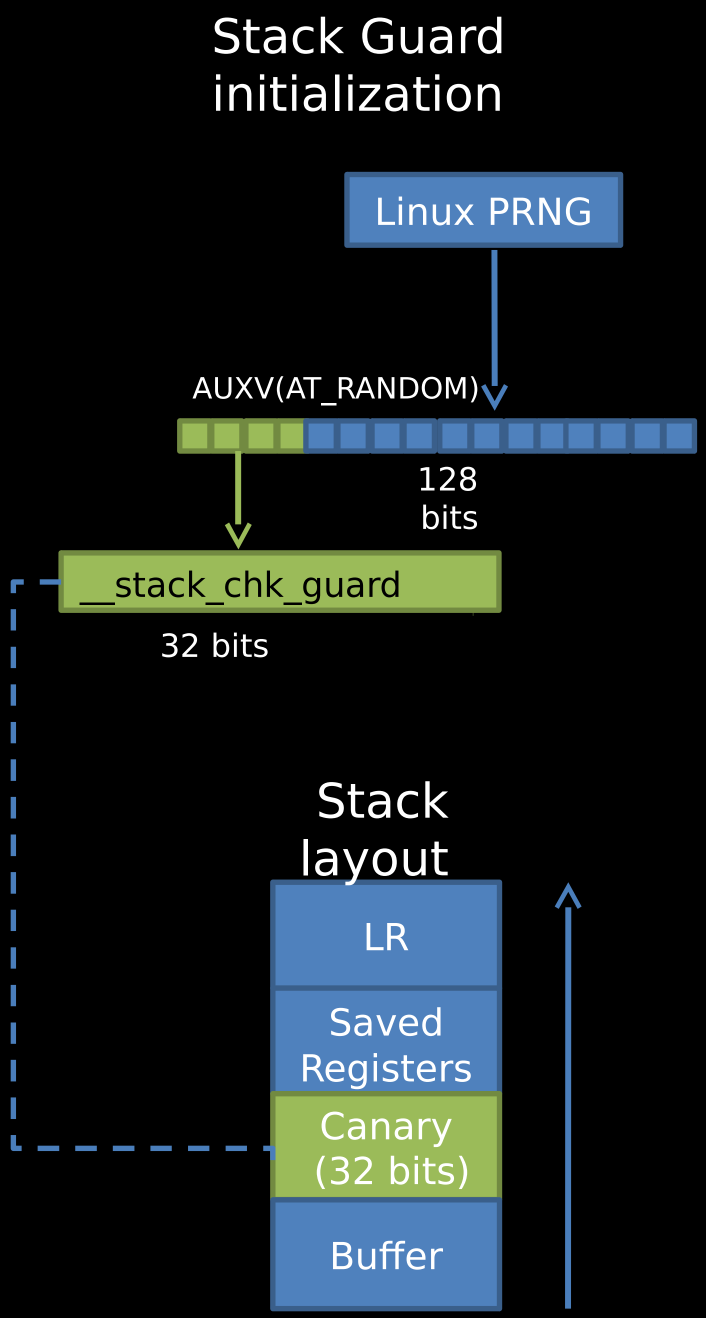
Stack canaries and their enforcement

- On libbionic load:
  `__stack_chk_guard = *(uintptr_t *)getauxval(AT_RANDOM));`
- Function Prologue:
  `Place __stack_chk_guard on the stack (before ret).`
- Function Epilogue:
  `Compare saved stack canary with __stack_chk_guard;`
  `→ Crash if mismatch`

Canary origins; *nix process creation model

- fork() → execve().
- execve() → Auxiliary vector (AUXV)
- AUXV[AT_RANDOM] = 16 Random bytes from the PRNG
- libbionic assigns canary = first 4 bytes of AT_RANDOM

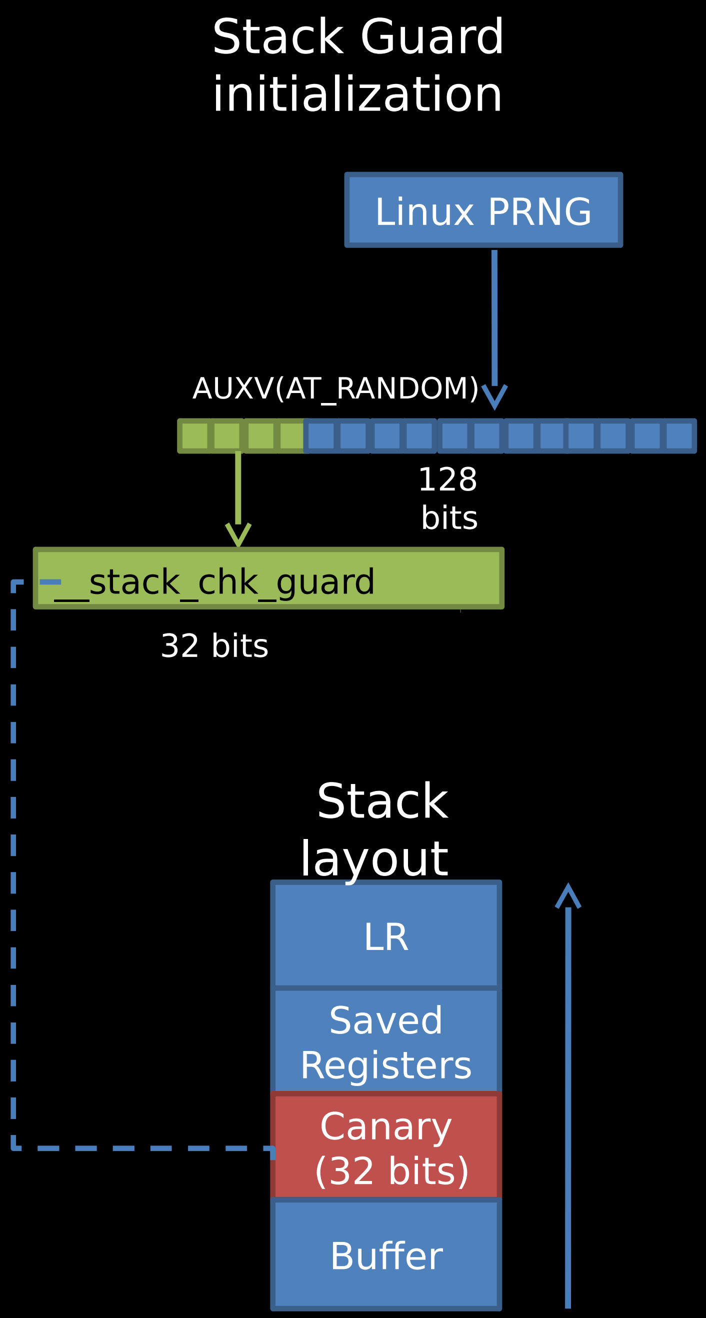
Remember this; We'll get back to it



motivation_keystore_buffer_overflow

Attacks on the Stack-Smashing Protection:

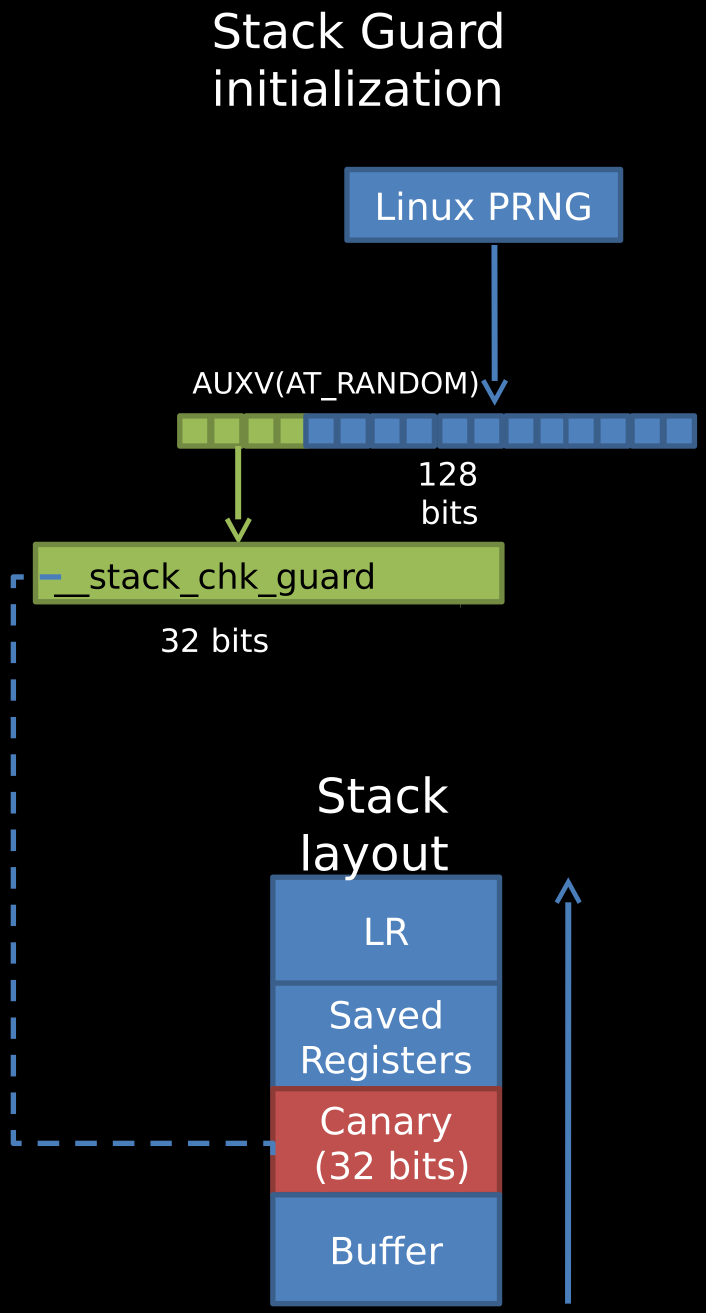
- Naive Online Bruteforce of the *Canary* Value
  - Impractical: 2^32 attempts on average.



motivation_keystore_buffer_overflow

Attacks on the Stack-Smashing Protection:

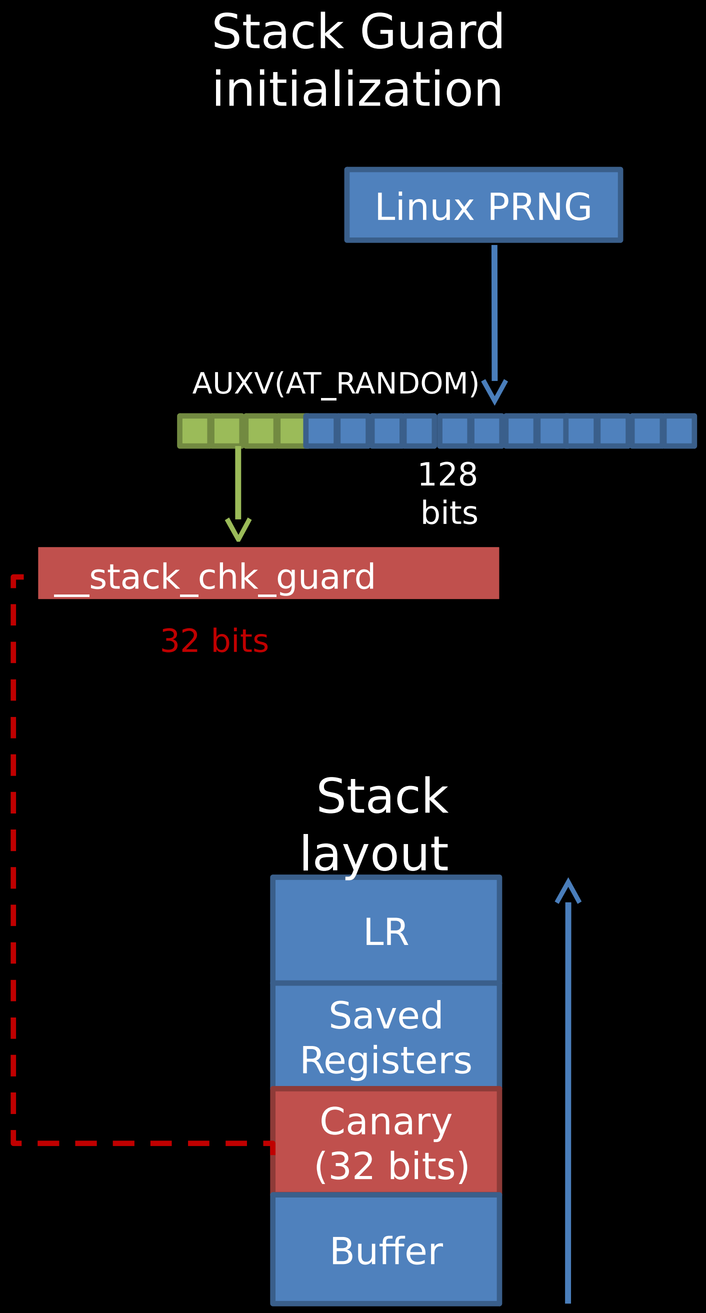
- Naive Online Bruteforce of the *Canary* Value
  - Impractical: 2^32 attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
  - Re-forking server:
    - Very efficient: 514 attempts until success on average



motivation_keystore_buffer_overflow

Attacks on the Stack-Smashing Protection:

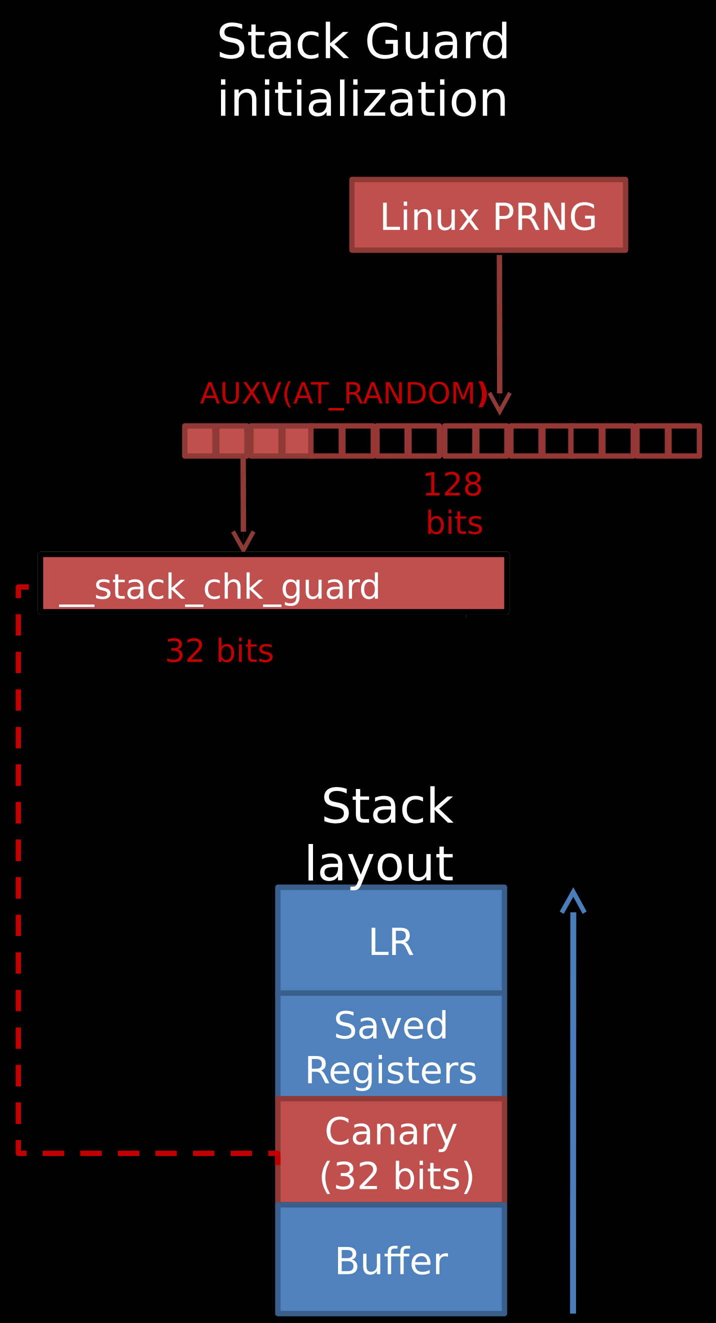
- Naive Online Bruteforce of the *Canary* Value
  - Impractical: 2^32 attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
  - Re-forking server:
    - Very efficient: 514 attempts until success on average
- Overwrite *__stack_chk_guard*
  - By overwriting some pointer



motivation_keystore_buffer_overflow

Attacks on the Stack-Smashing Protection:

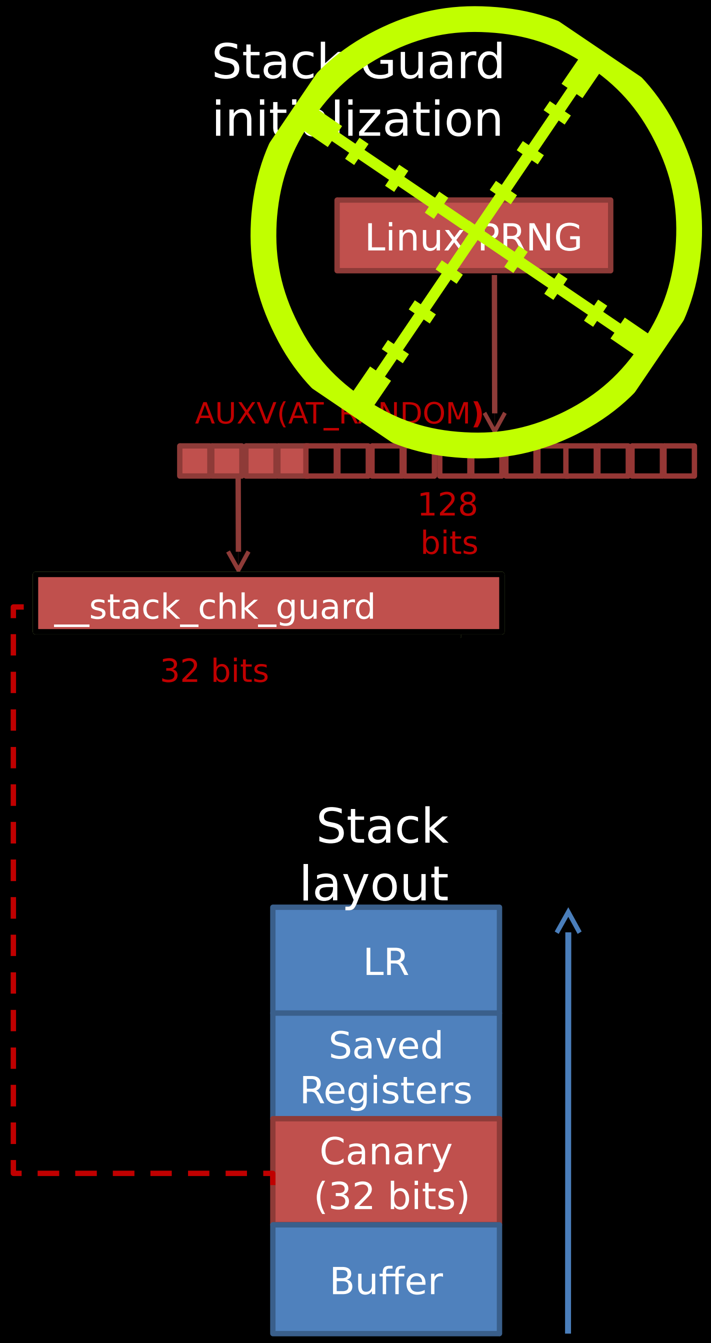
- Naive Online Bruteforce of the *Canary* Value
  - Impractical: 2^32 attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
  - Re-forking server:
    - Very efficient: 514 attempts until success on average
- Overwrite *__stack_chk_guard*
  - By overwriting some pointer
- **Our attack**: *Offline* reconstruction of the PRNG's internal state



motivation_wrap_up

Wrap things up:

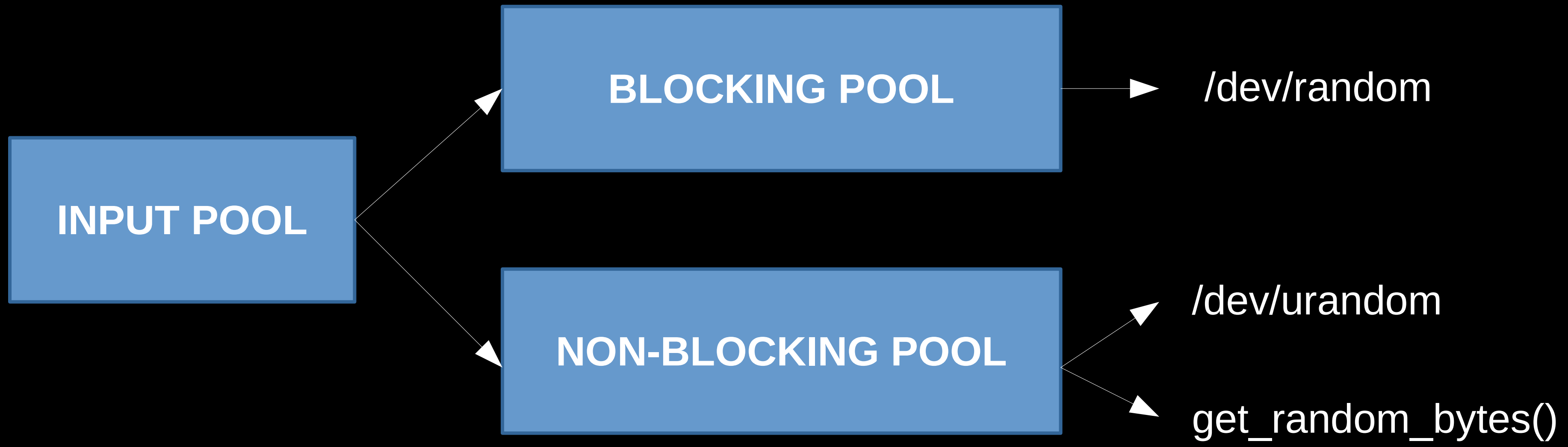
- We found a vulnerability in a critical service in Android 4.3.
- In an effort to exploit it, we had to overcome a stack canary, we couldn't do so using known techniques.
- Canaries are 4 random bytes that are extracted from the Linux PRNG.
- Aimed to find a weakness in the PRNG that will allow us to intelligently guess the canary.
- End up with a full-fledged attack on the Linux PRNG.



LINUX PRNG

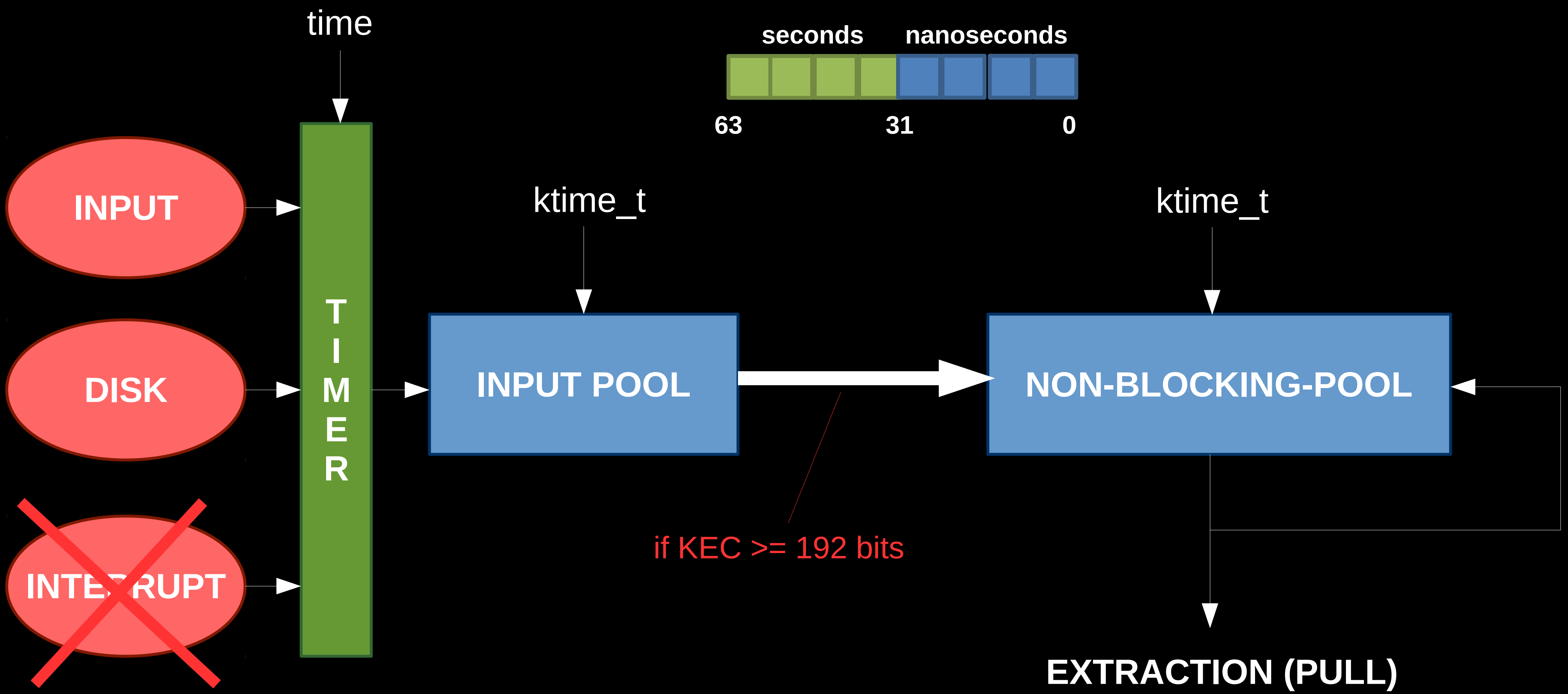
lprng_overview

Bird's eye view



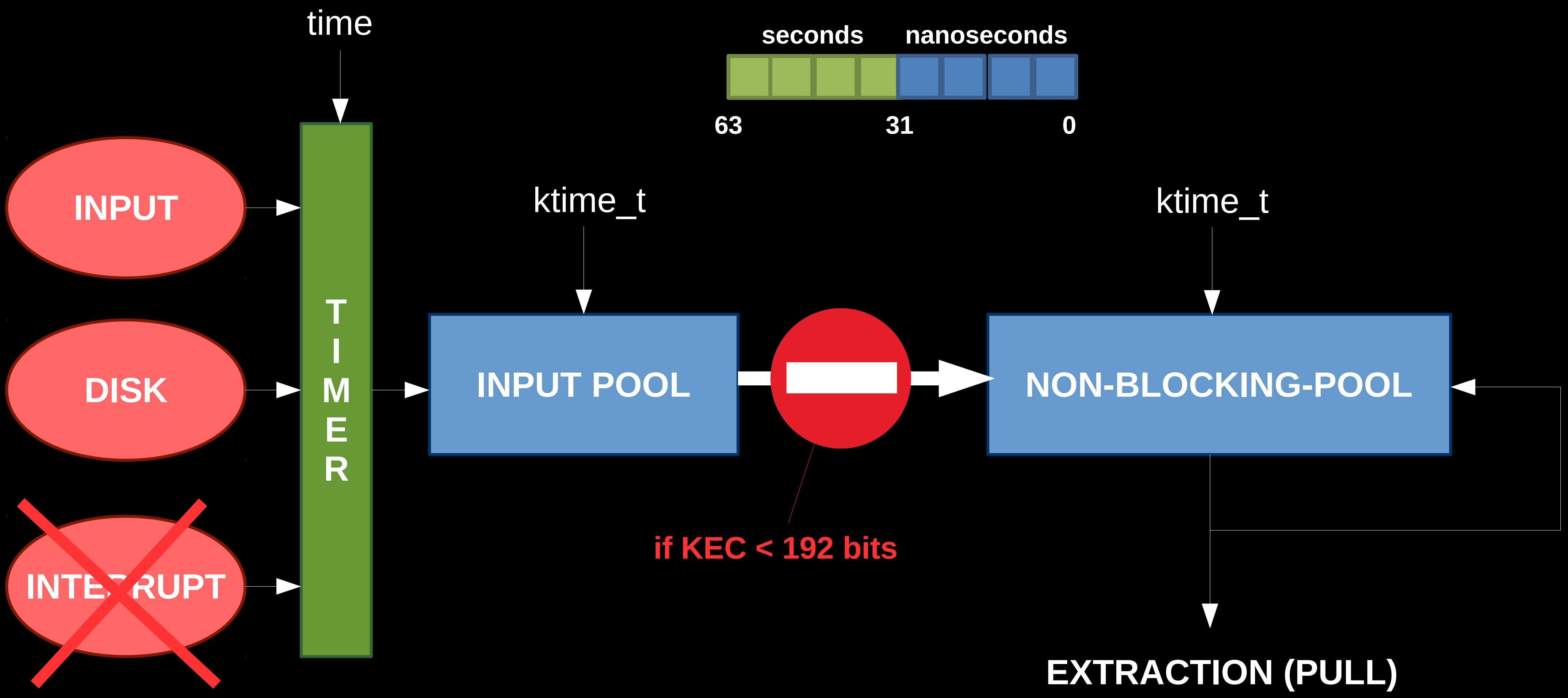
- Output is hashed twice using SHA1
- Extracts in blocks of 10 bytes and truncates if necessary.

entropy_sources



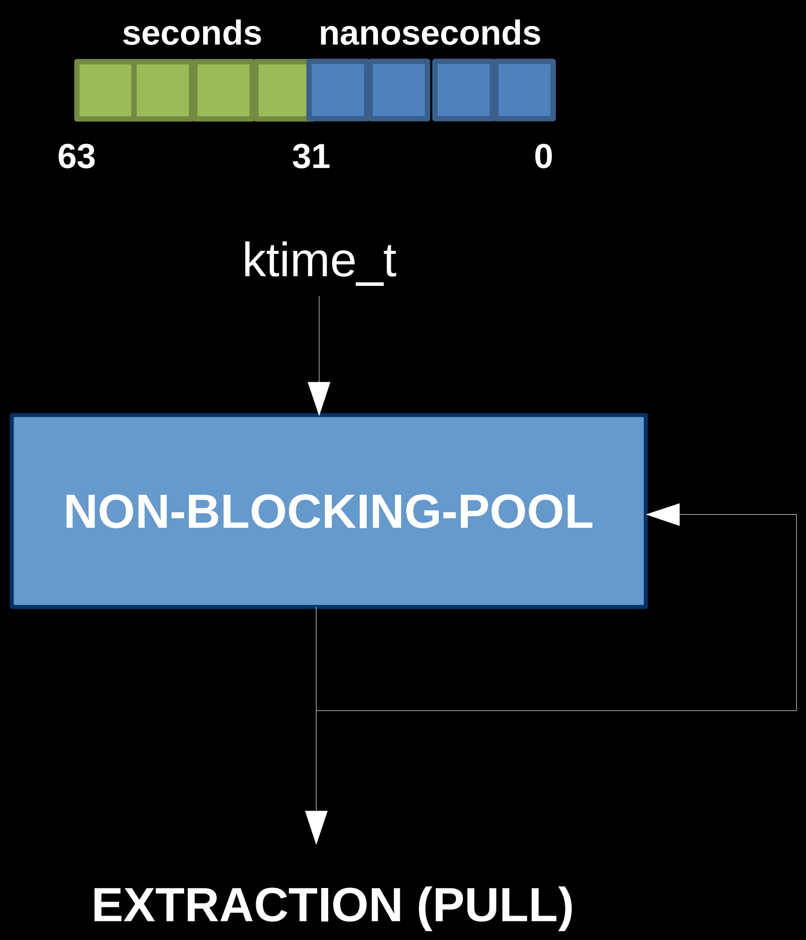
*KEC = Kernel Entropy Count

boot_time_vulnerability

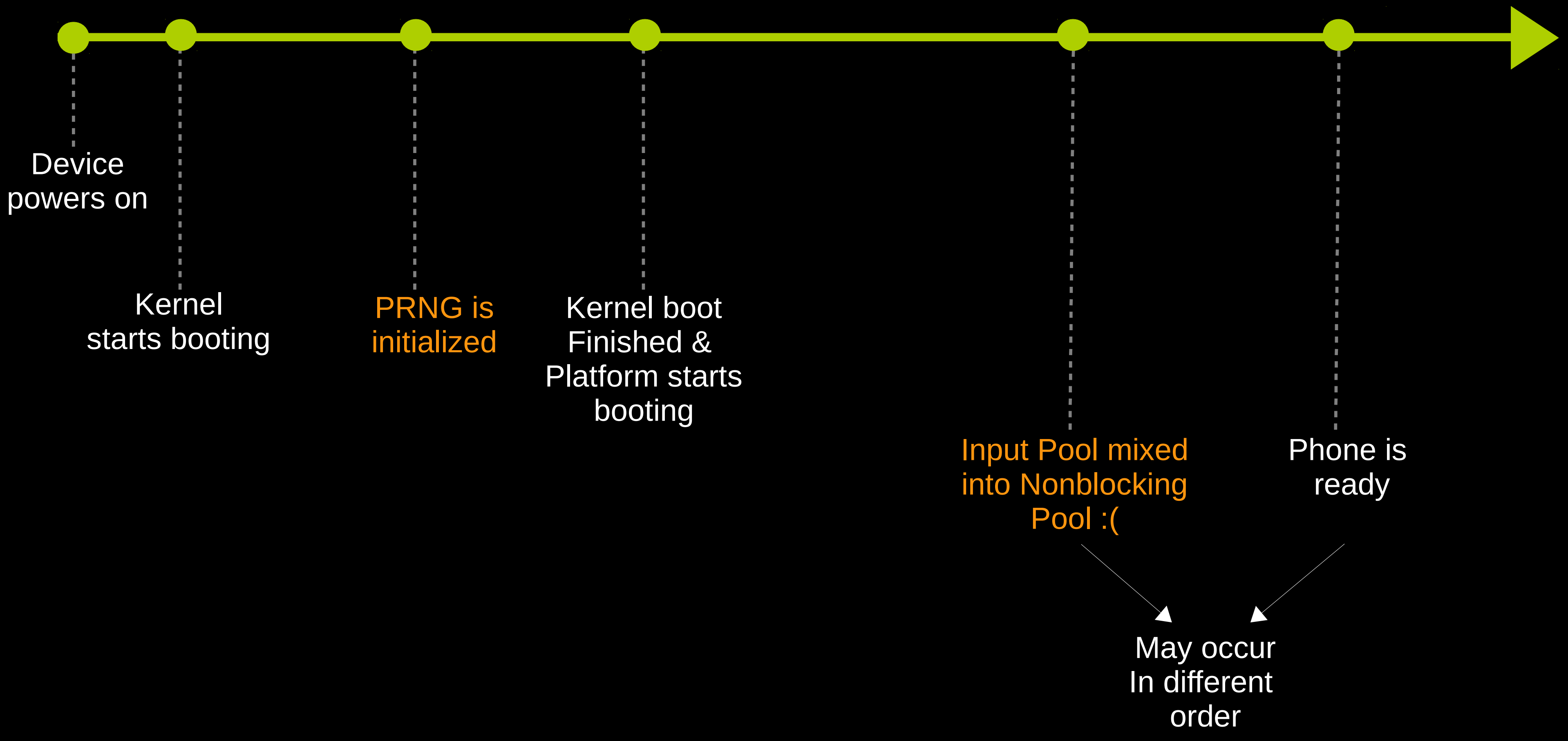


*KEC = Kernel Entropy Count

boot_time_vulnerability



boot_timeline



OUR WORK

contribution

Prior art on weakness in early boot *

Present practical run-time attack

Formalize attack

Demonstrate PoC against current mobile platforms

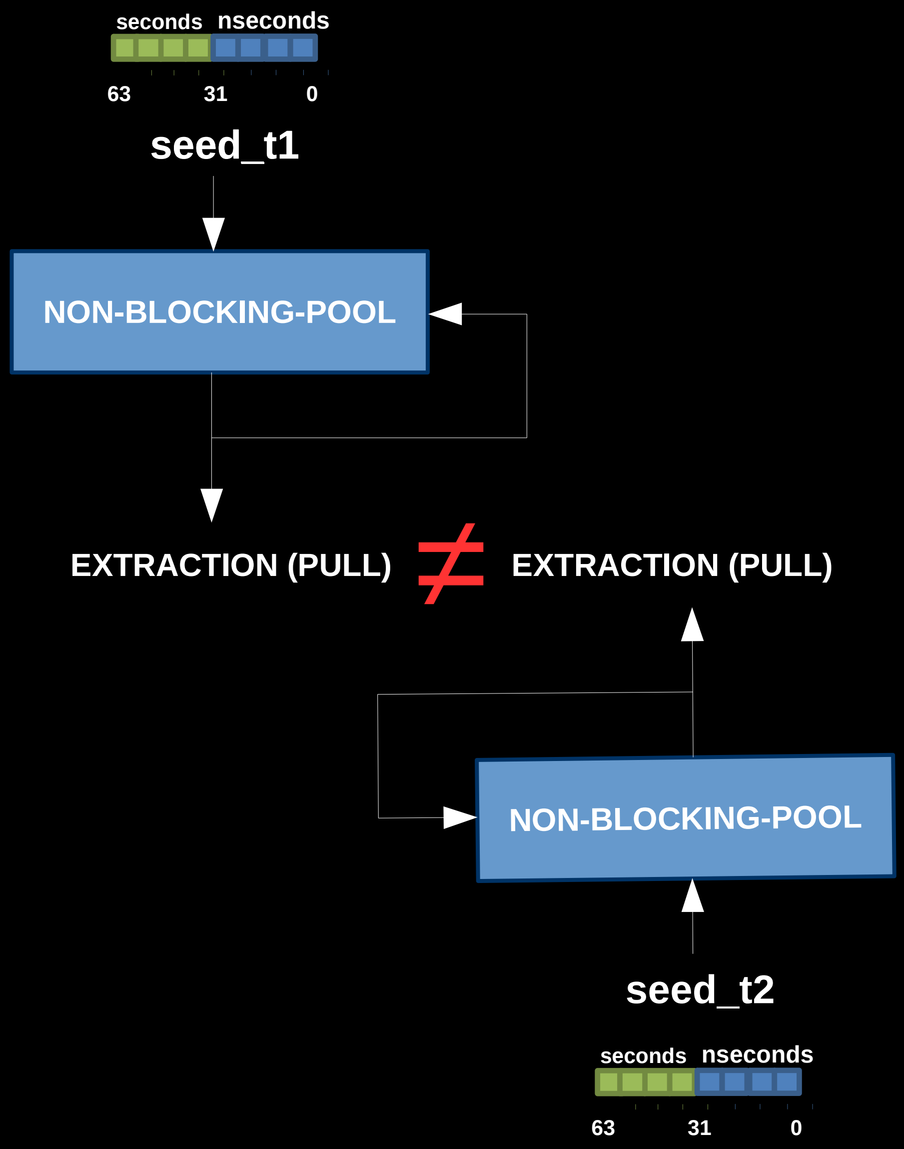
** Heninger et al. 2012, Becherer et al. 2009, Ding et al. 2014*

attack_outcome

Given a **LEAK** of a value extracted from the non-blocking pool and **LOW ENTROPY AT BOOT**, the **STATE** of the PRNG can be determined until external entropy is too high attack_leak

Using the PRNG against itself

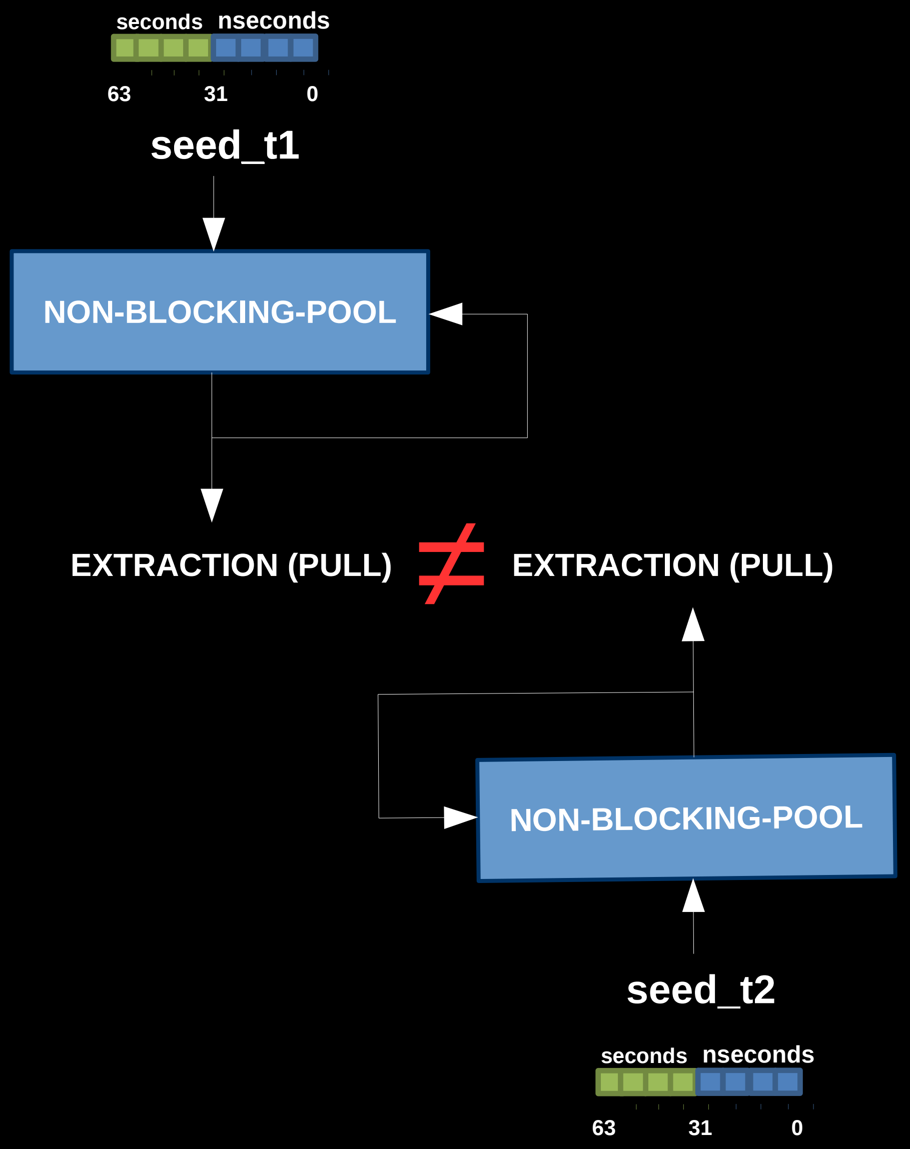
- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.



attack_leak

Using the PRNG against itself

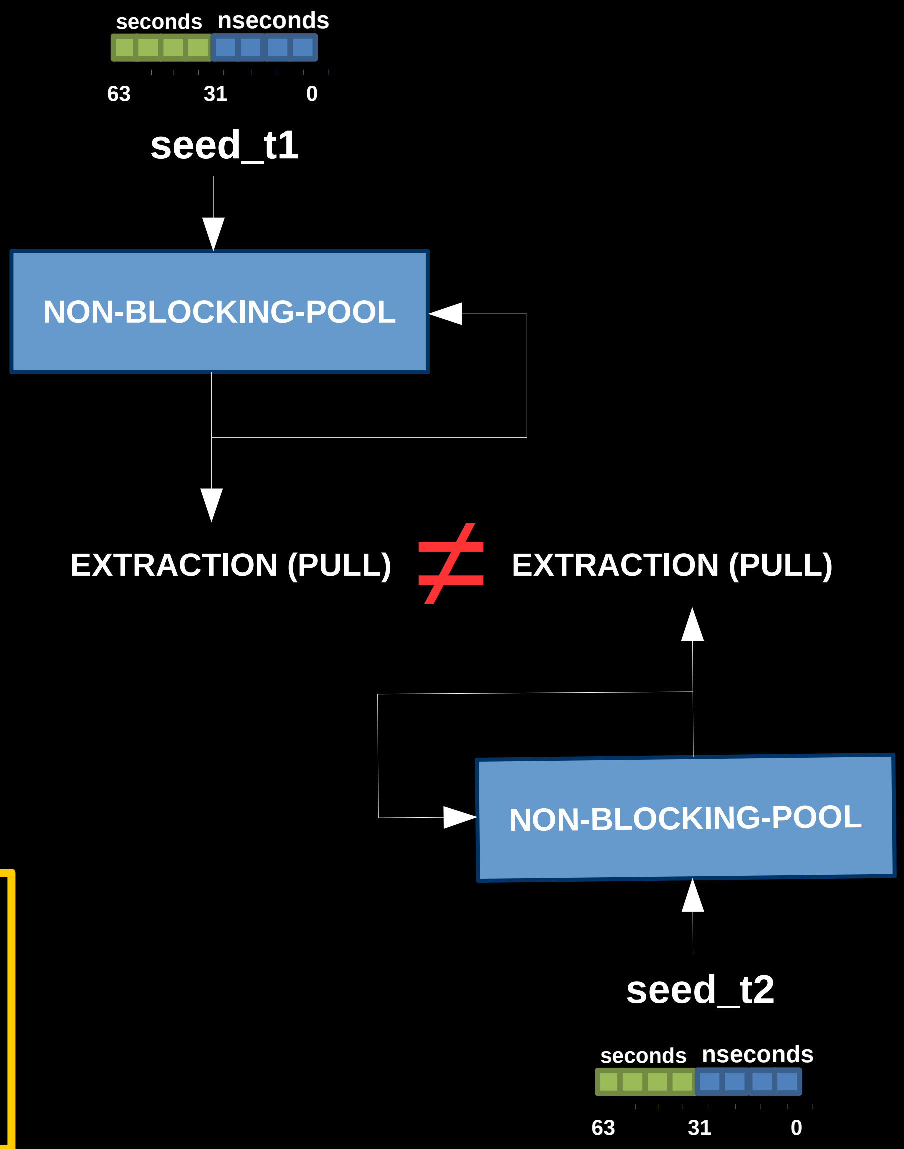
- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.
- It is highly unlikely that PRNGs that are seeded with different seeds will result in the same output. Regardless of the order of extractions.



attack_leak

Using the PRNG against itself

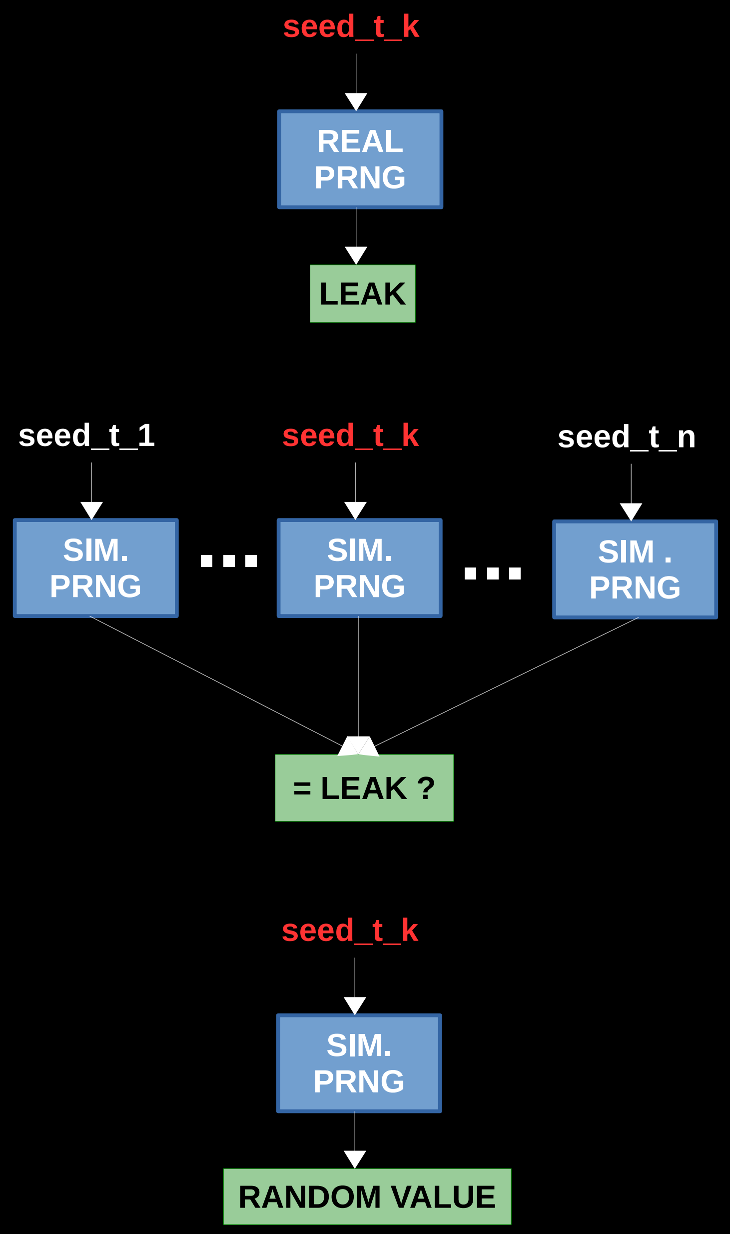
- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.
- It is highly unlikely that PRNGs that are seeded with different seeds will result in the same output. Regardless of the order of extractions.
- Result: Every leak(sequence of random bytes) from the non blocking pool is almost certainly the offspring of **one** specific seed.



attack_overview

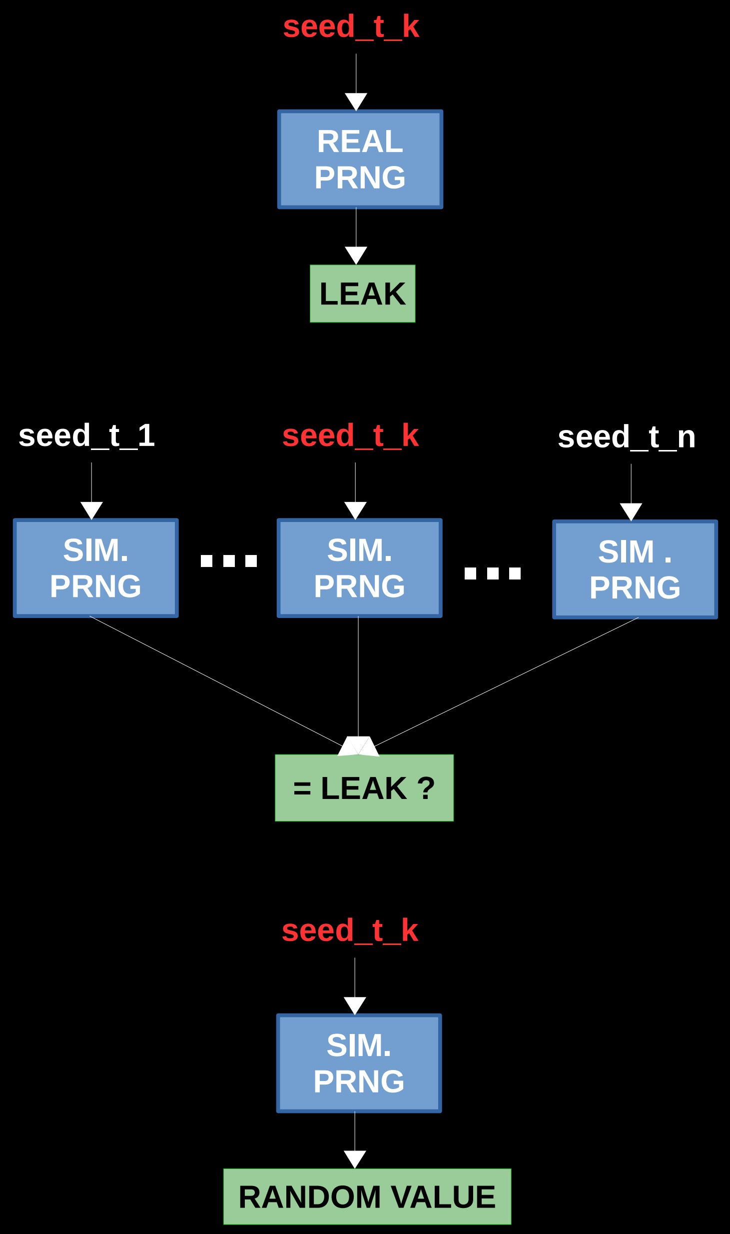
Using the PRNG against itself

- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.



Using the PRNG against itself

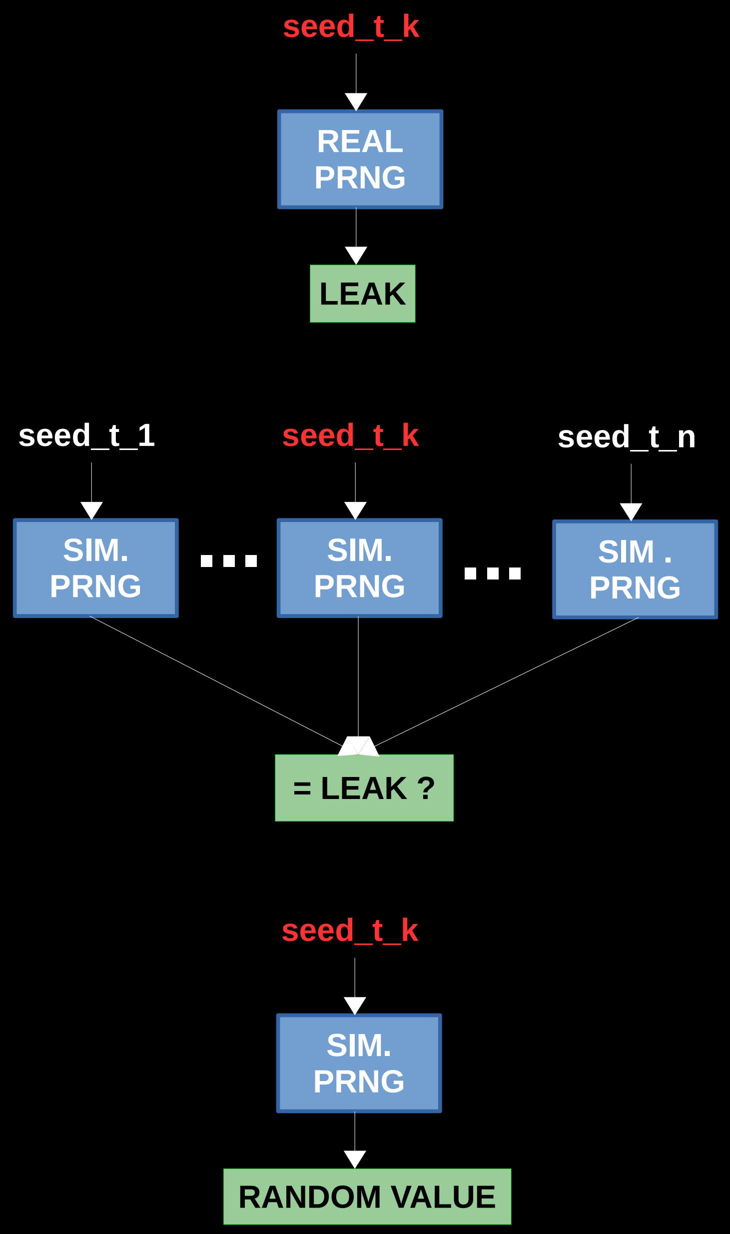
- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.
- Due to SHA1's collision resistance, if one of the simulated PRNGs produces a sequence of random bytes that is the same as the leak value – we almost certainly found the seed.



attack_overview

Using the PRNG against itself

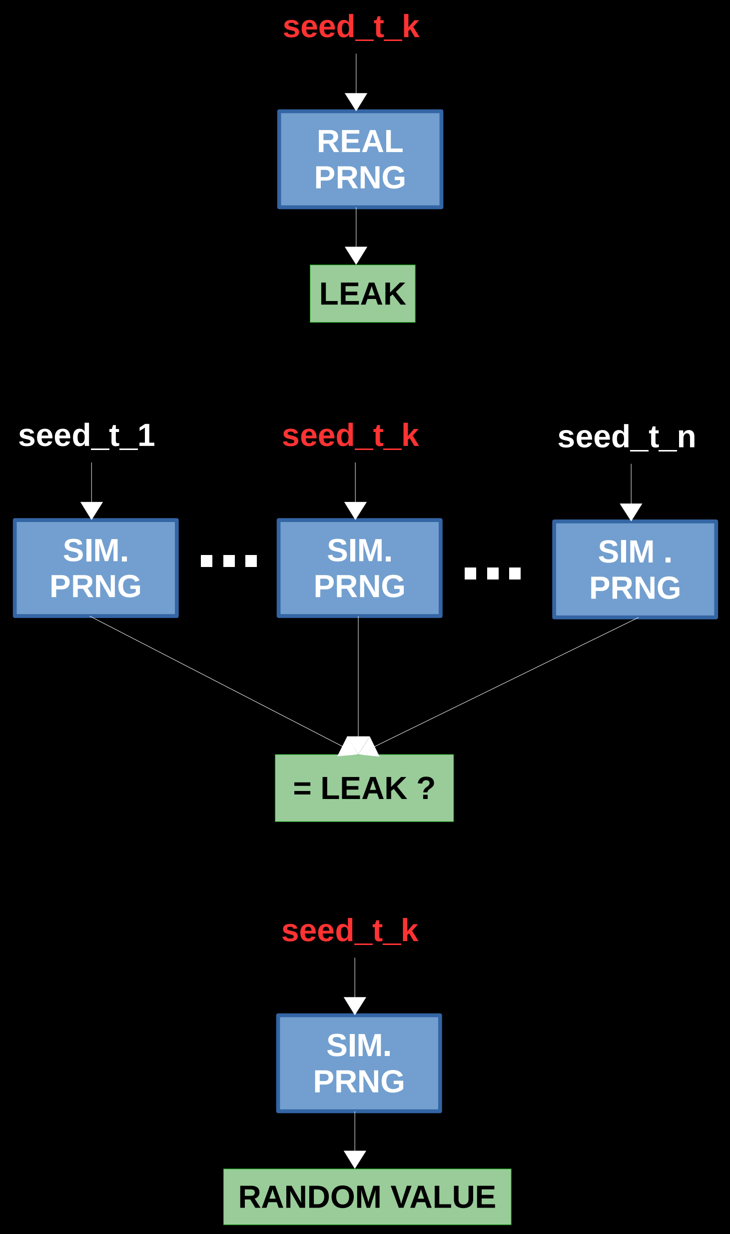
- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.
- Due to SHA1's collision resistance, if one of the simulated PRNGs produces a sequence of random bytes that is the same as the leak value – we almost certainly found the seed.
- Once we have the seed we can produce the same outputs of the “Real” PRNG until noise from the Input pool is mixed to the Nonblocking pool



attack_overview

Even After the mixing, the PRNG is vulnerable

- **Note:** in the whitepaper we demonstrated a more intricate attack flow

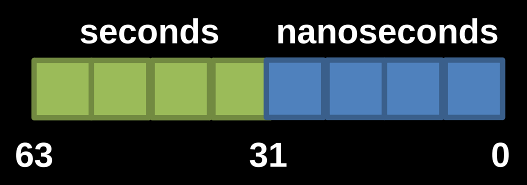


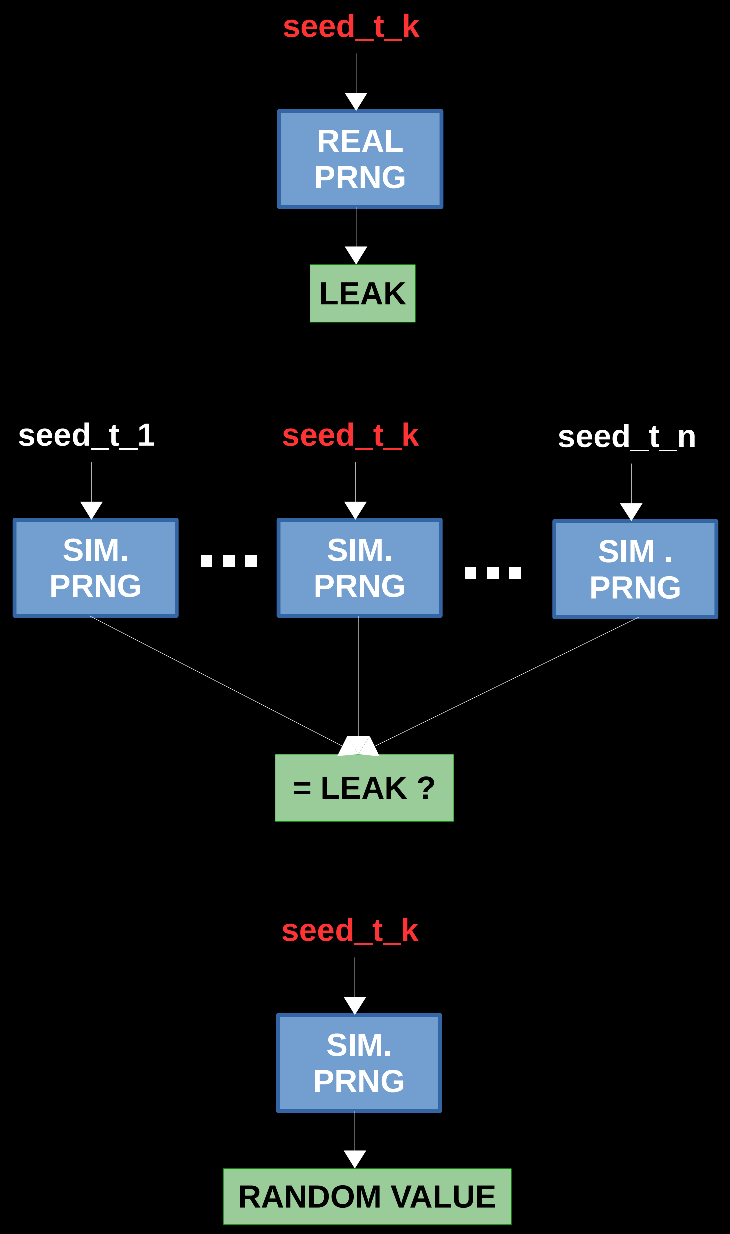
attack_overview

Problems we faced:

- The Nonblocking pool seed is 8 bytes long, Say we consider only the nanoseconds and assuming uniform distribution

$$10^9 = 2^{\log_2(10^9)} \simeq 2^{30}$$





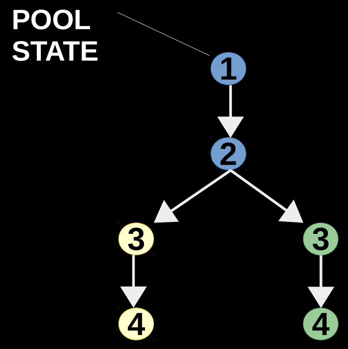
attack_overview

Problems we faced:

- The Nonblocking pool seed is 8 bytes long, Say we consider only the nanoseconds and assuming uniform distribution

$$10^9 = 2^{\log_2\left(10^9\right)} \simeq 2^{30}$$

- Hidden entropy source – Concurrency

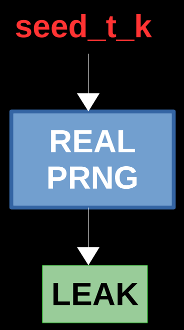


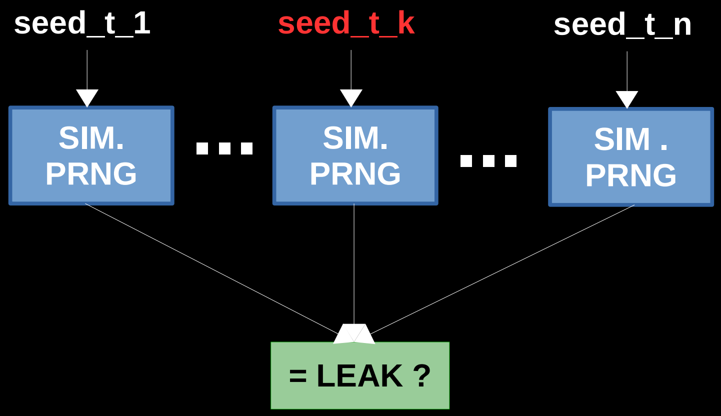
Yellow Path

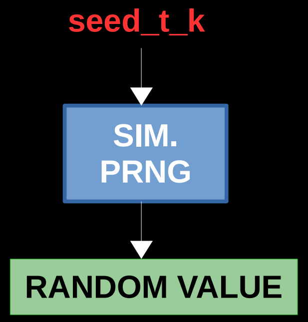
- **Process A:** extract from pool
- **Process A:** mix into pool
- **Process B:** extract from pool
- **Process B:** mix into pool

Green Path

- **Process A:** extract from pool
- **Process B:** extract from pool
- **Process A:** mix into pool
- **Process B:** mix into pool







attack_overview

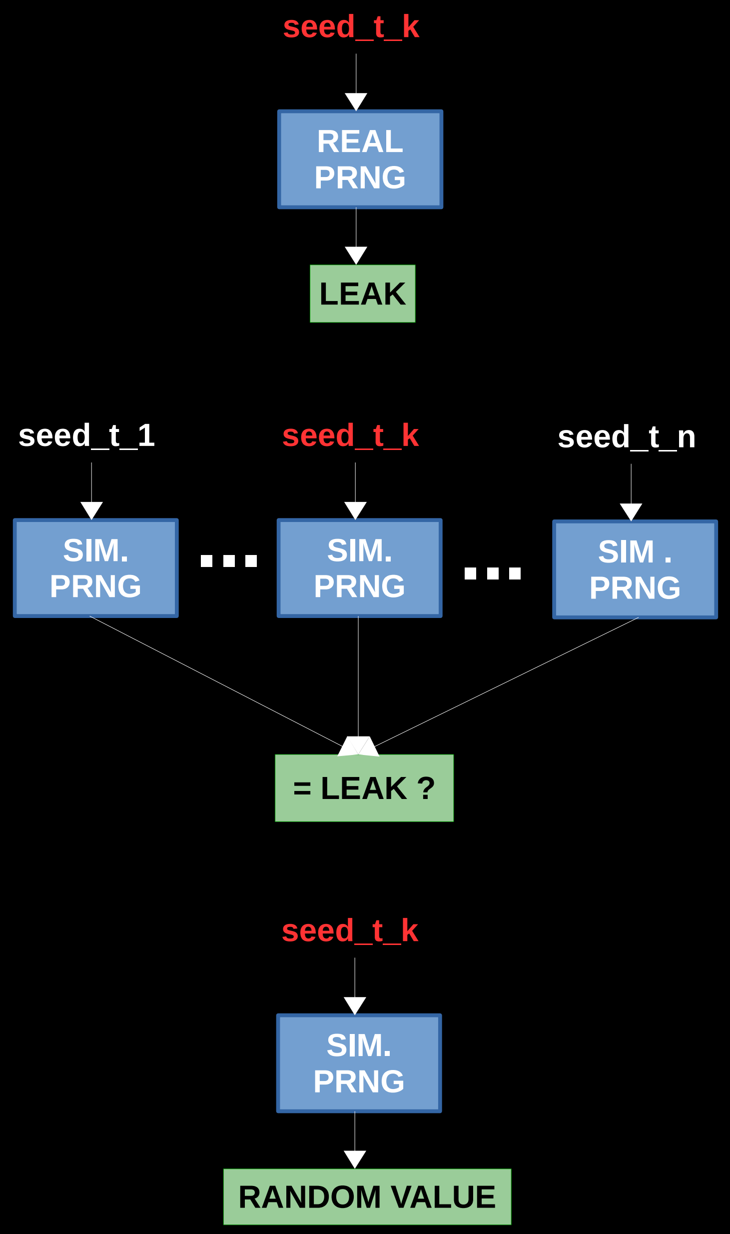
Problems we faced:

- The Nonblocking pool seed is 8 bytes long, Say we consider only the nanoseconds and assuming uniform distribution

$$10^9 = 2^{\log_2\left(10^9\right)} \simeq 2^{30}$$

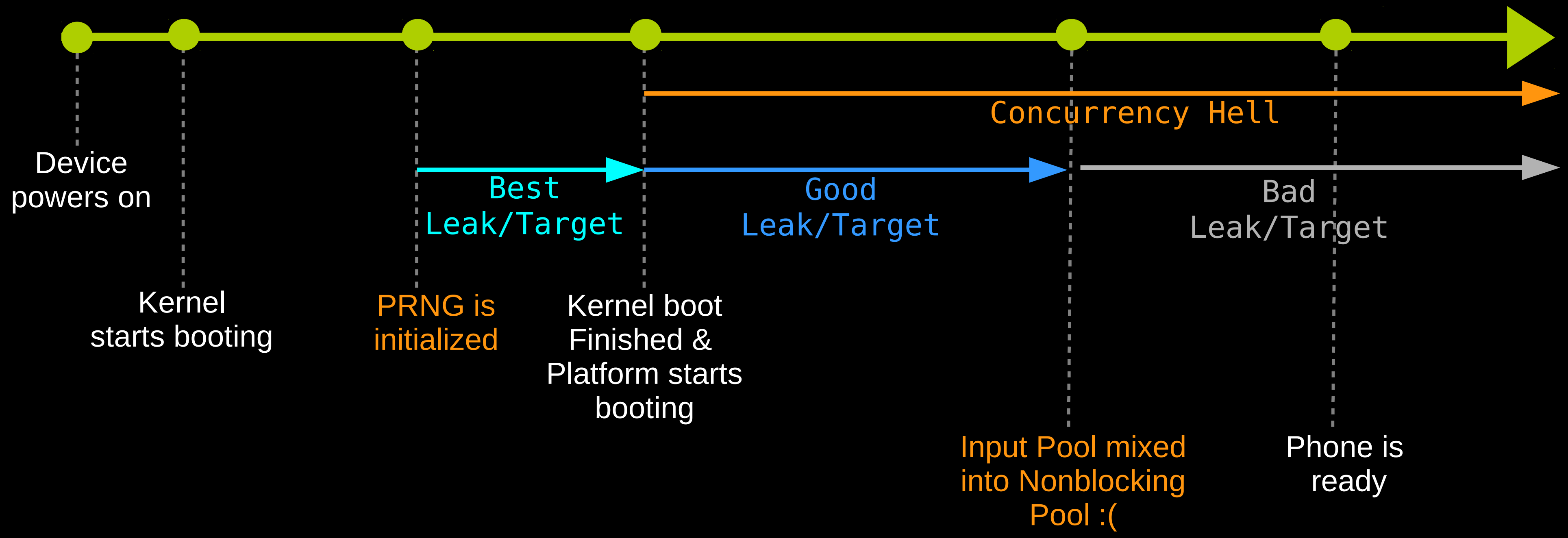
- Hidden entropy source – Concurrency

- What can be attacked?
- Where can we get the leak value?



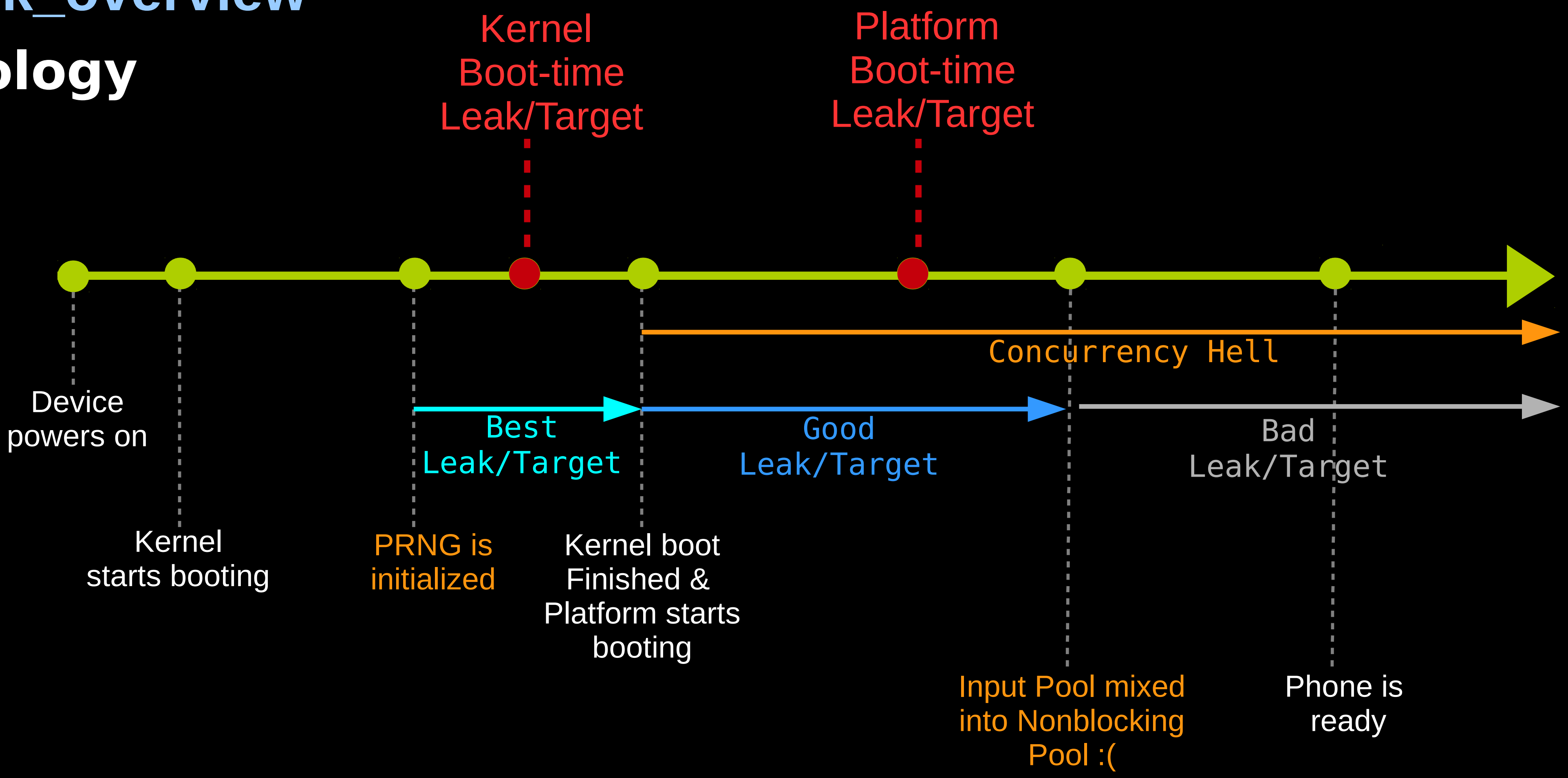
attack_overview

Where can we find leaks and attack targets ?



attack_overview

Terminology

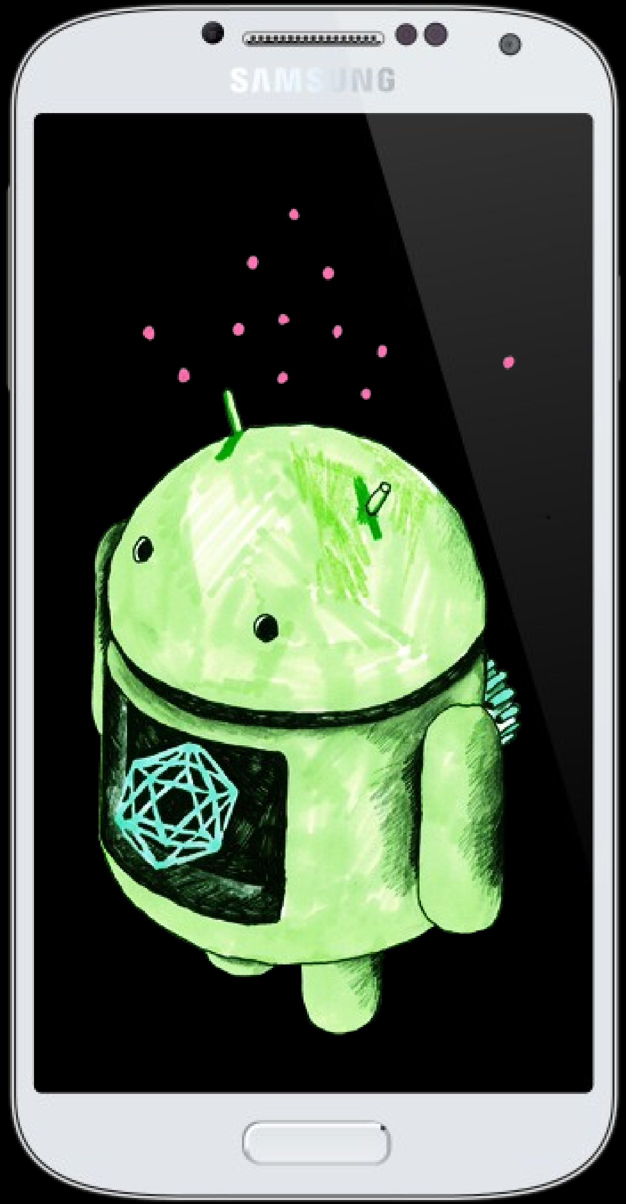


1st Attack Vector
Malware → PRNG Seed → Keystore's Canary

s4_offline_study

Instrumenting a device

- Samsung Galaxy S4, Android 4.3



s4_offline_study

Instrumenting a device

- Samsung Galaxy S4, Android 4.3

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets

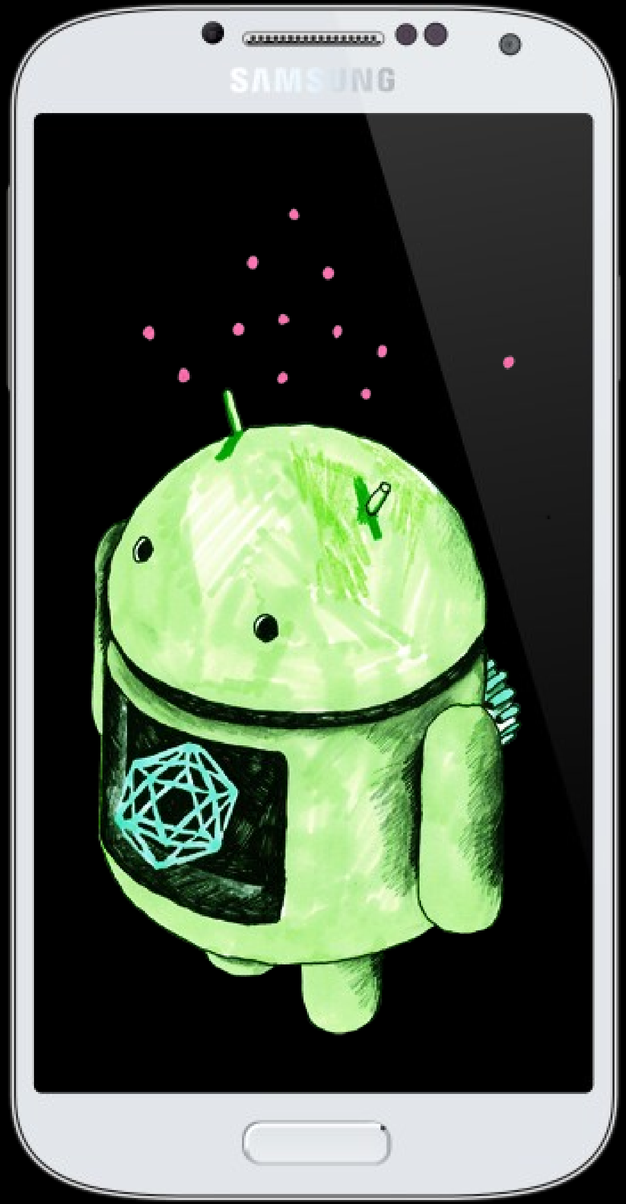
s4_offline_study

Instrumenting a device

- Samsung Galaxy S4, Android 4.3

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets

- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.



s4_offline_study

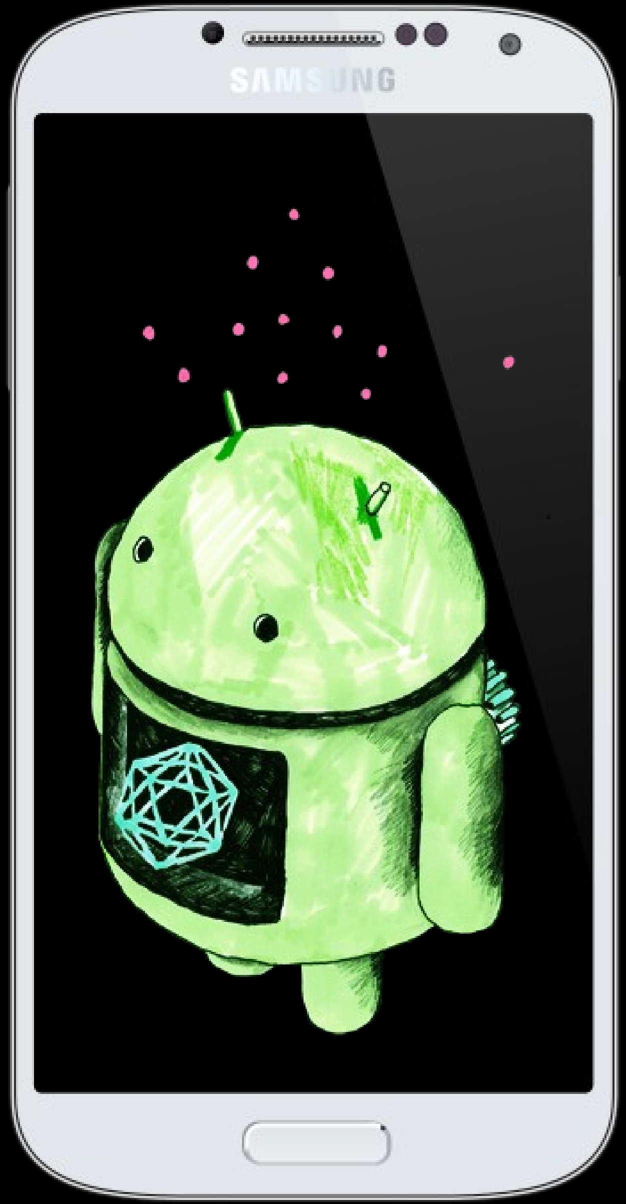
Instrumenting a device

- Samsung Galaxy S4, Android 4.3

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets

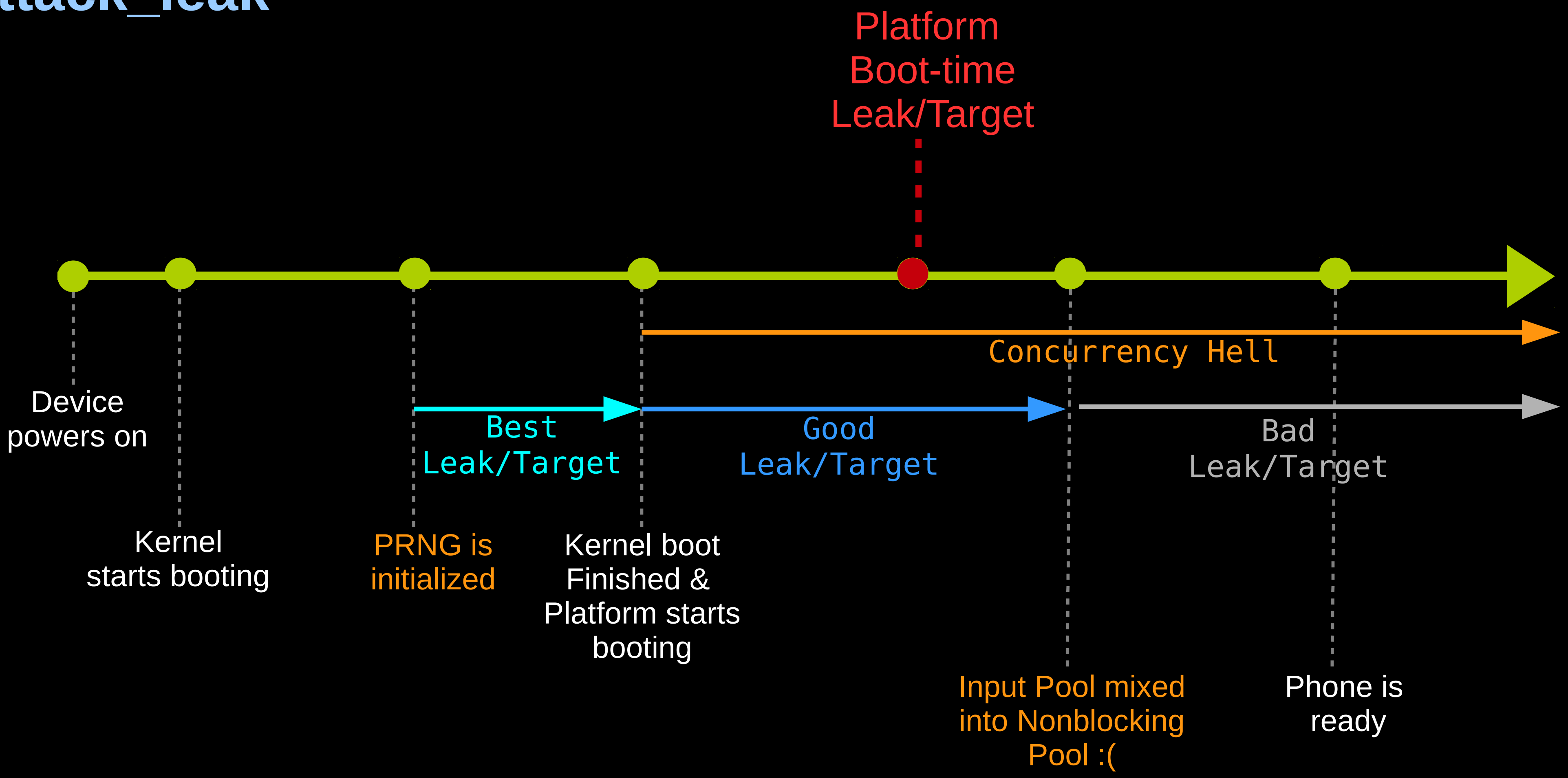
- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.

- In total, we rebooted(script) the device more than 2000 times, each time we dumped the kernel ring buffer to a file.



s4_attack_leak

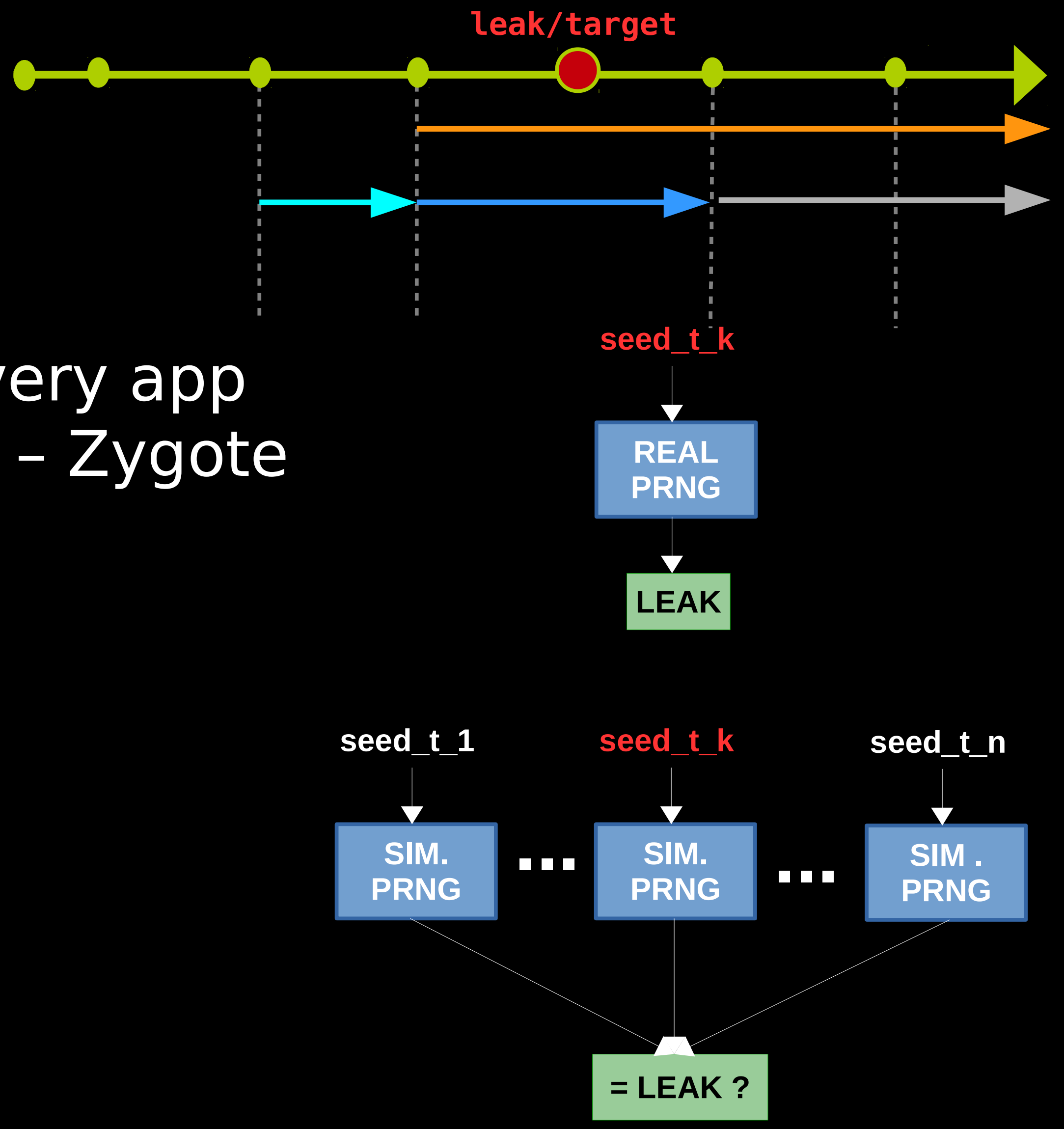
Details



s4_attack_leak

Details

- Android designers chose to spawn every app process by **fork**ing a master process – Zygote

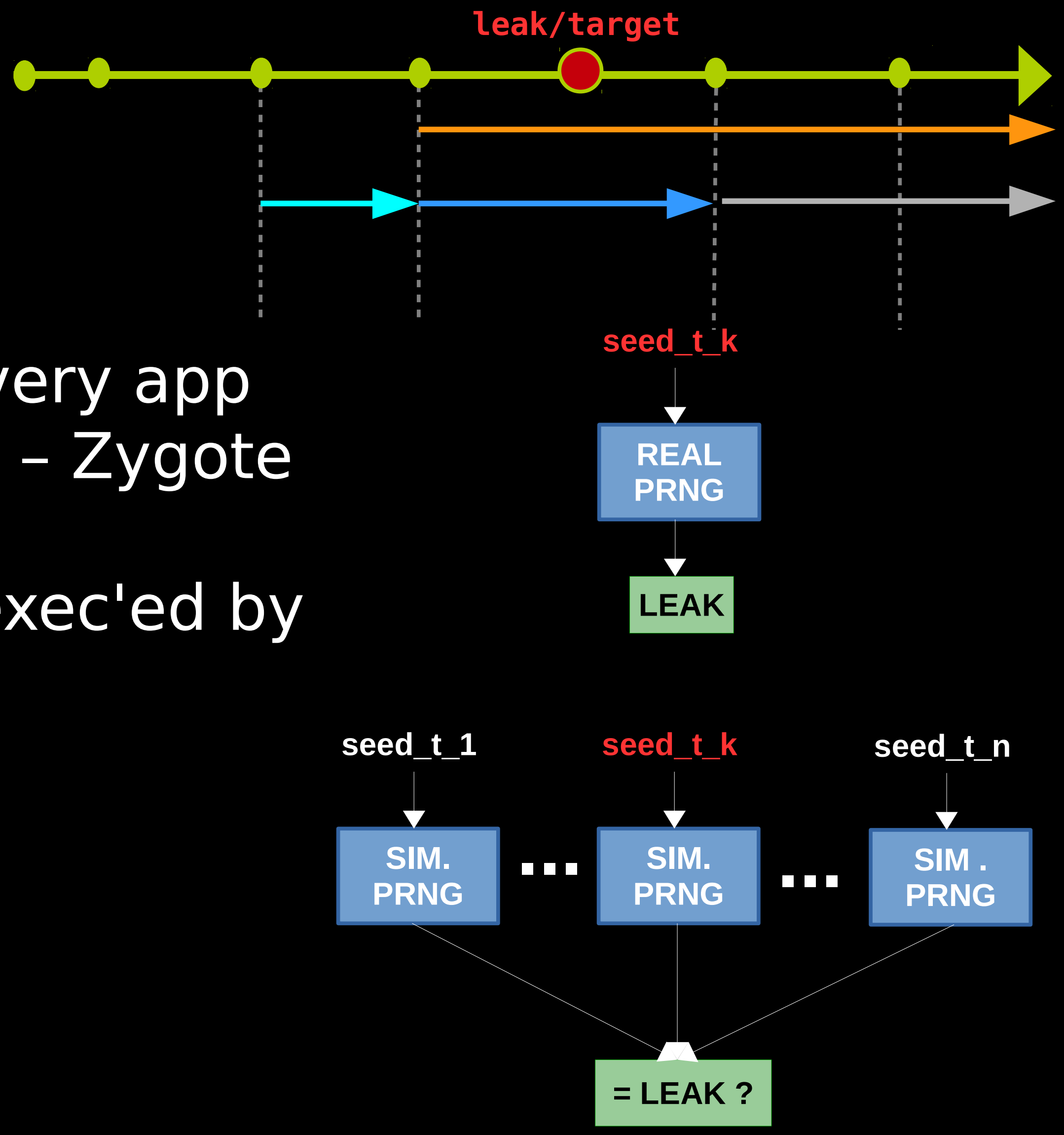


s4_attack_leak

Details

- Android designers chose to spawn every app process by **fork**ing a master process – Zygote

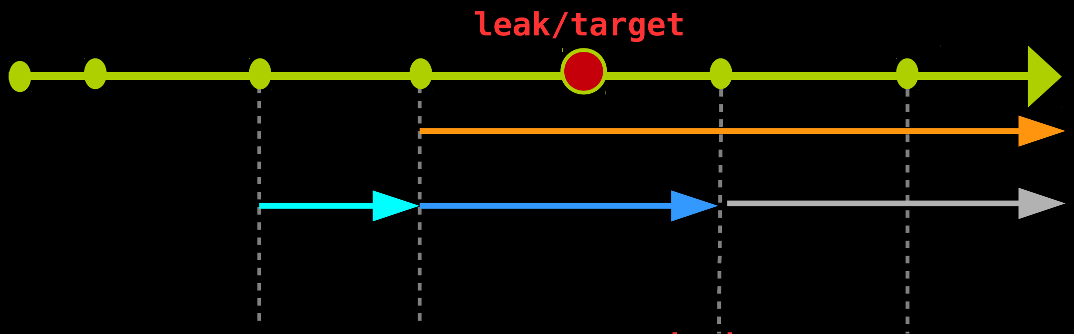
- Zygote(app_process) is fork'ed and exec'ed by init at platform boot-time

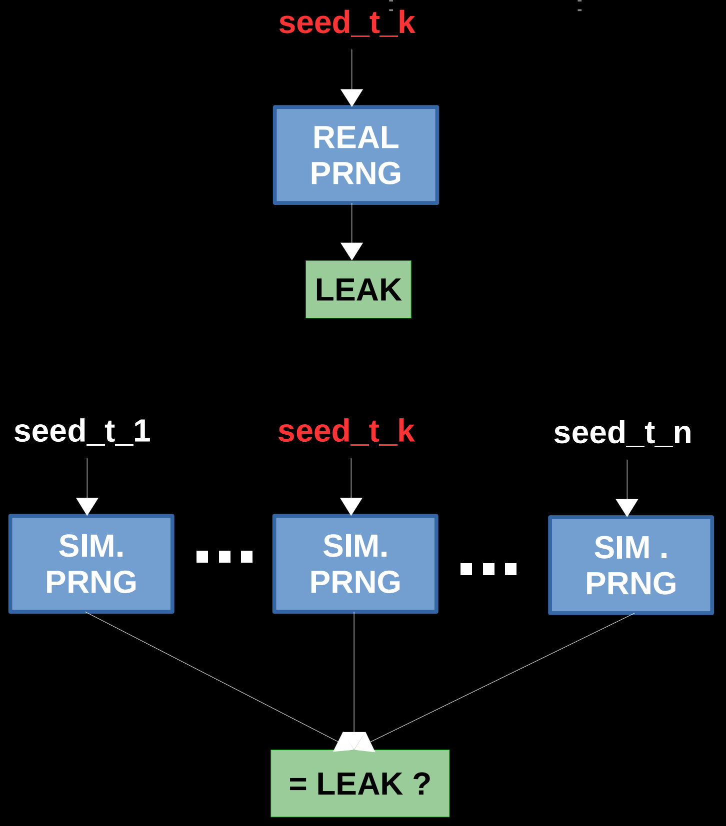


s4_attack_leak

Details

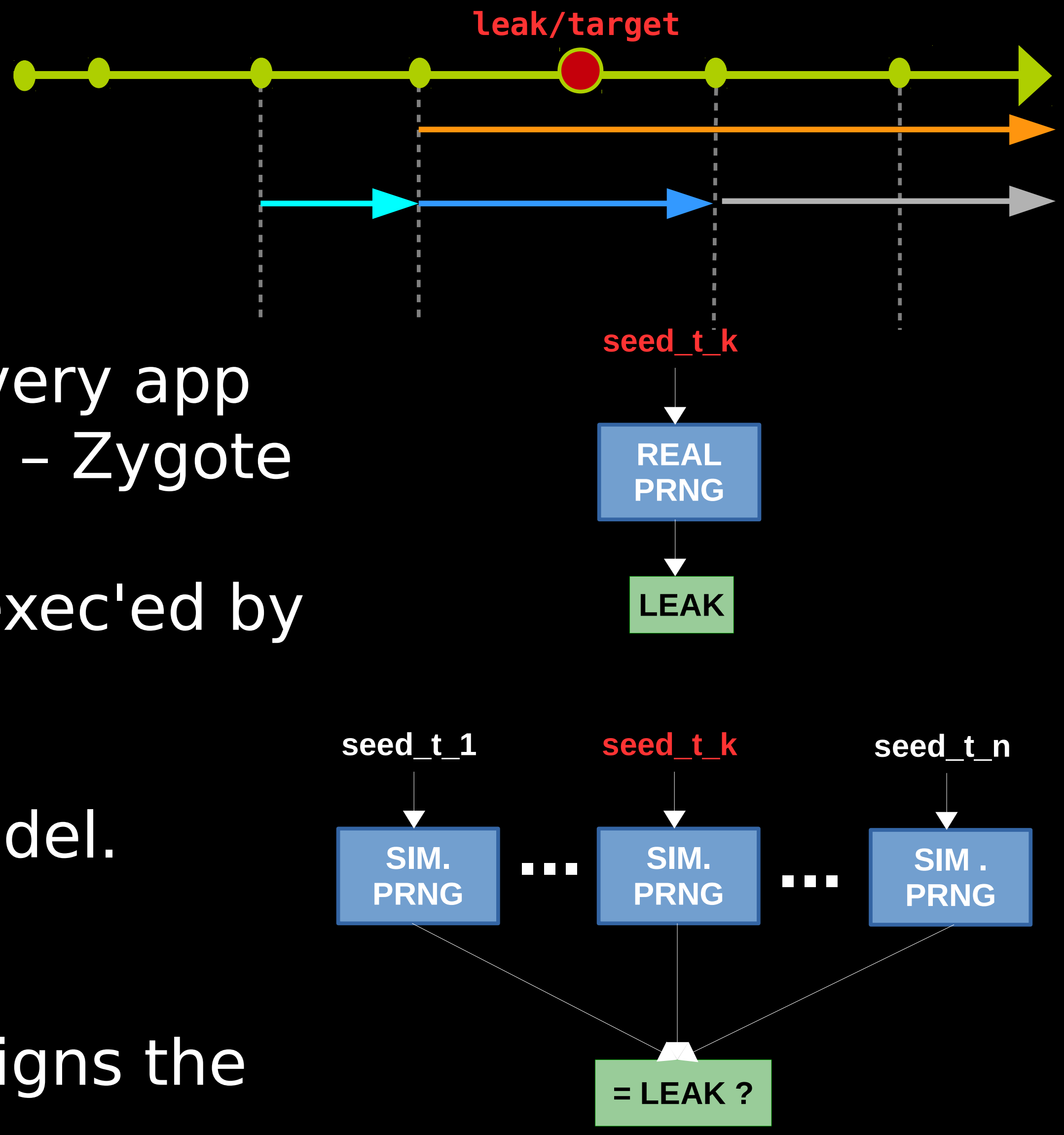
- Android designers chose to spawn every app process by **fork**ing a master process – Zygote
- Zygote(app_process) is fork'ed and exec'ed by init at platform boot-time
- *nix-like vs. App process creation model. Exec() ?





s4_attack_leak

Details

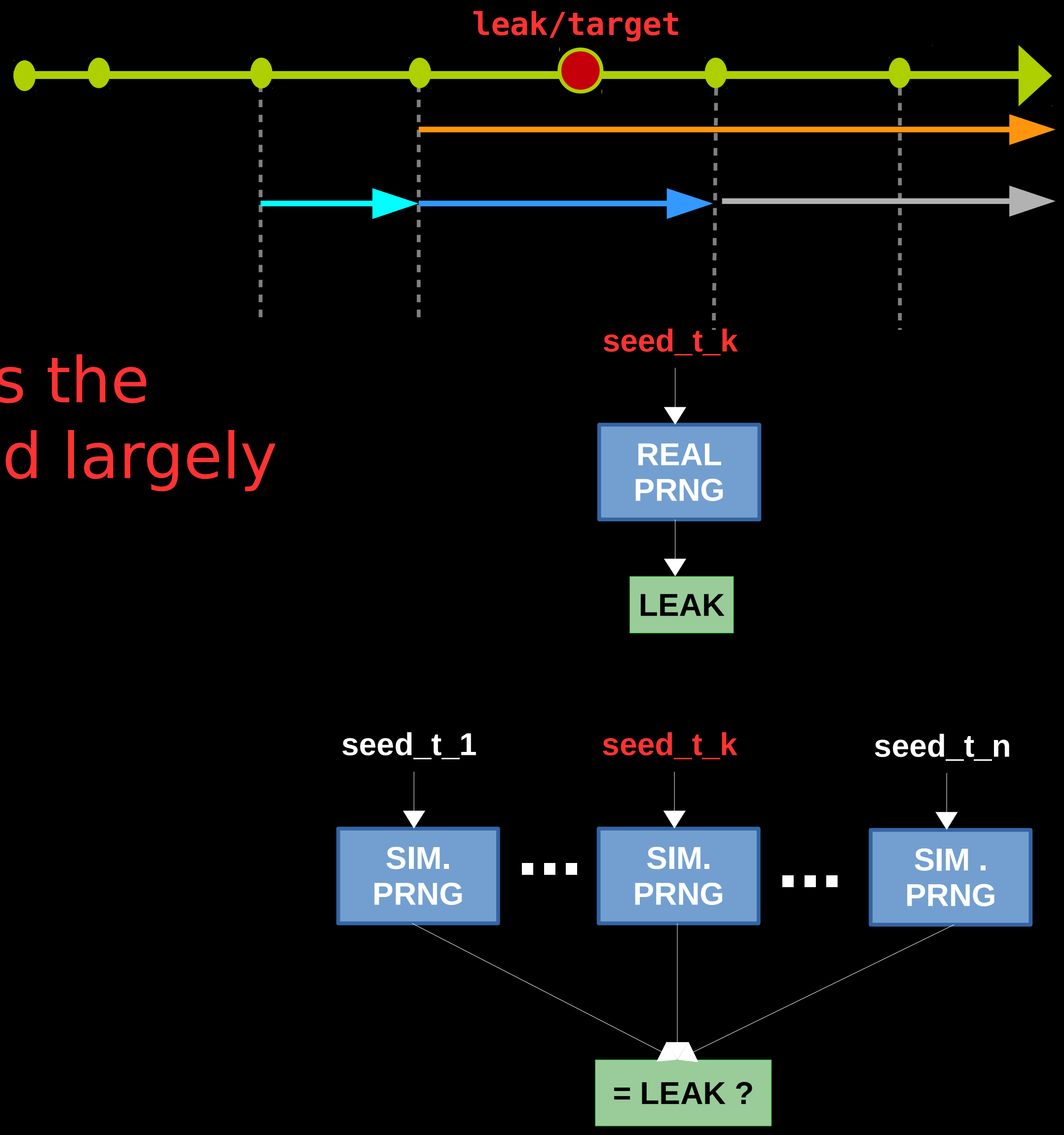


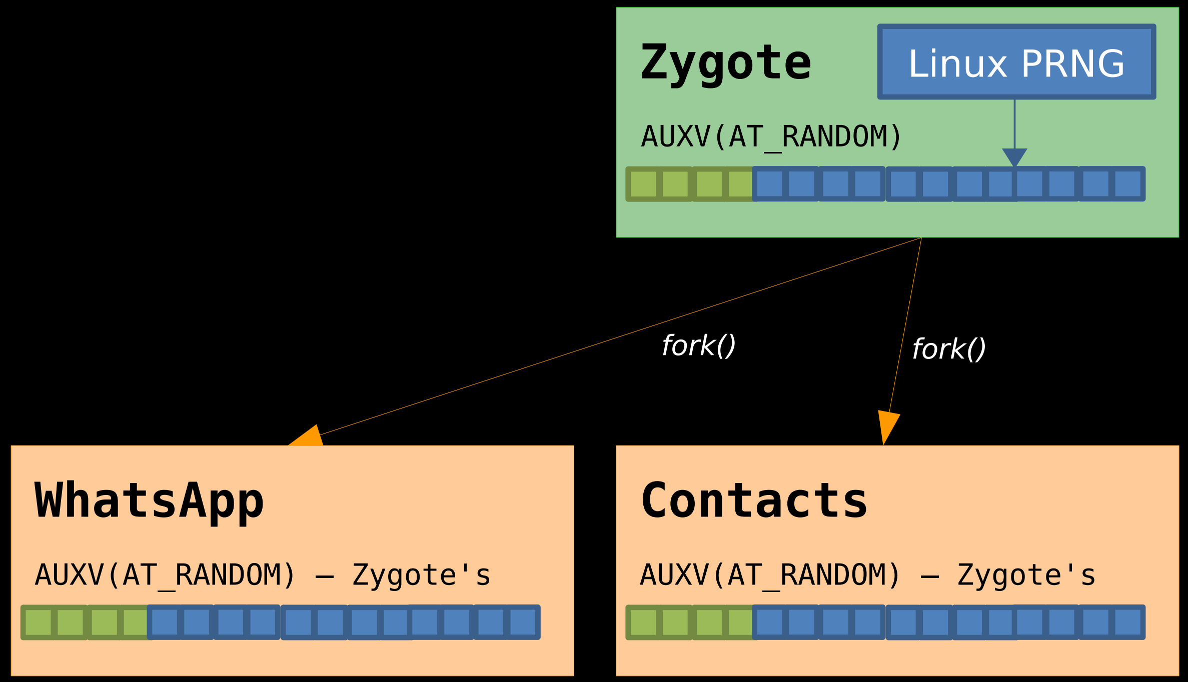
- Android designers chose to spawn every app process by **fork**ing a master process – Zygote
- Zygote(app_process) is fork'ed and exec'ed by init at platform boot-time
- *nix-like vs. App process creation model. Exec() ?
- Recall: exec() enforces ASLR and assigns the AT_RANDOM

s4_attack_leak

Details

- Result: All Applications in Android has the same Canary value (AT_RANDOM) and largely the same address space layout

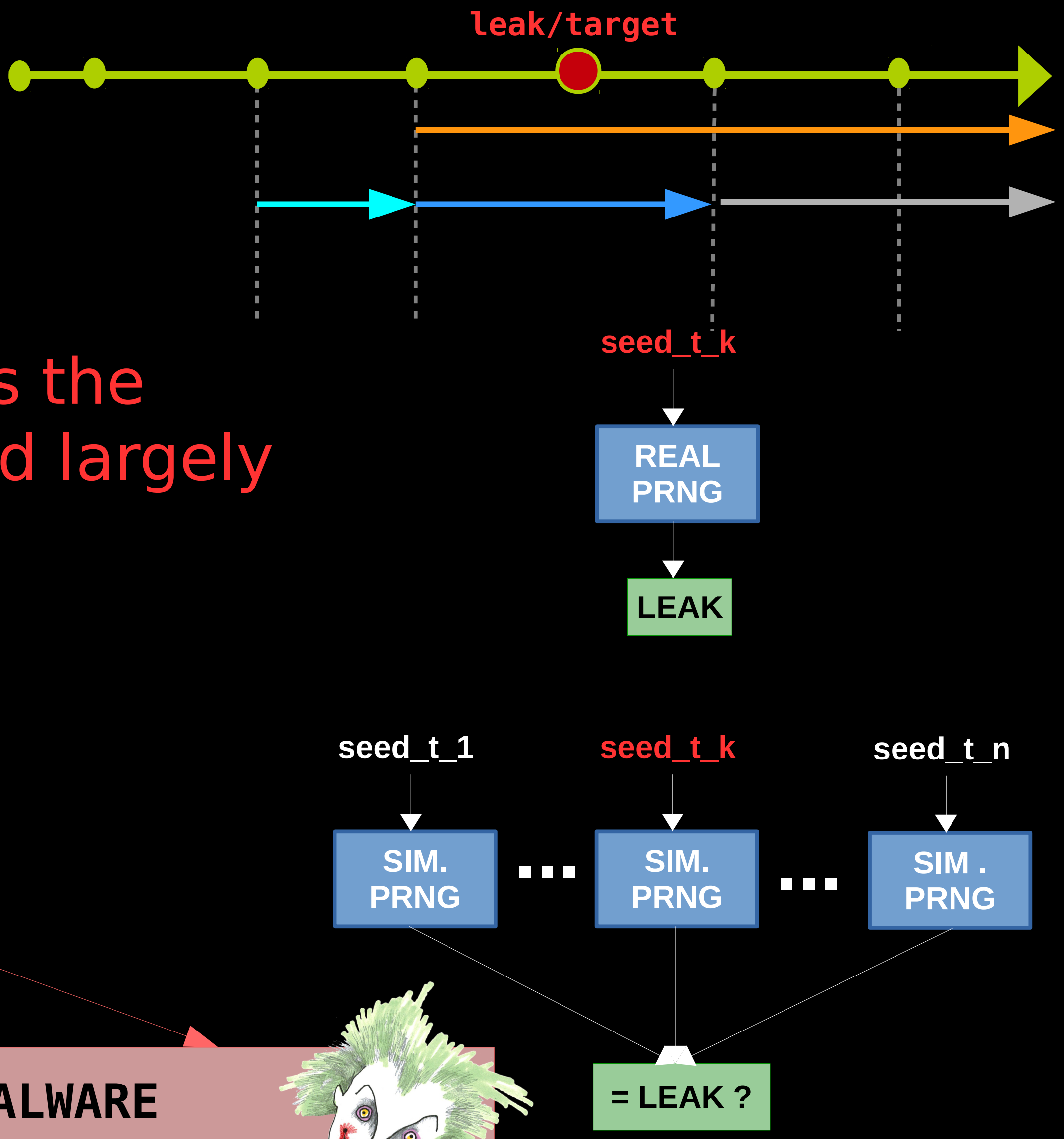


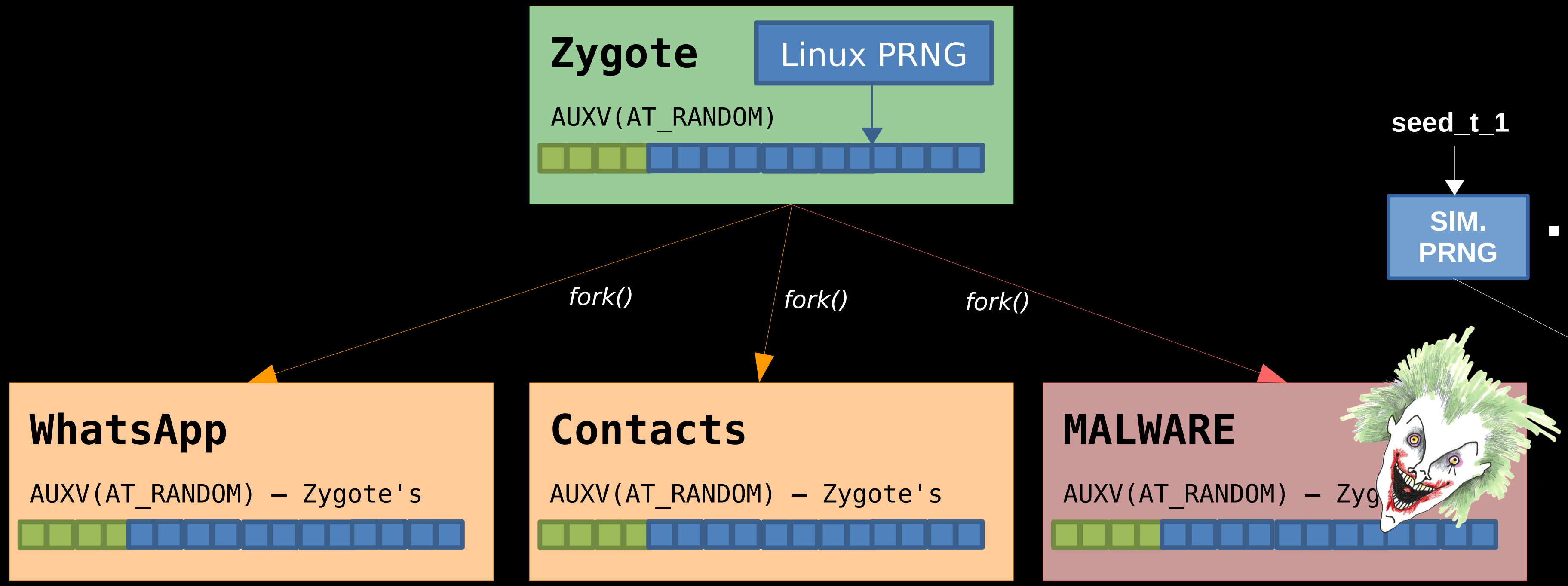


s4_attack_leak

Details

- Result: All Applications in Android has the same Canary value (AT_RANDOM) and largely the same address space layout

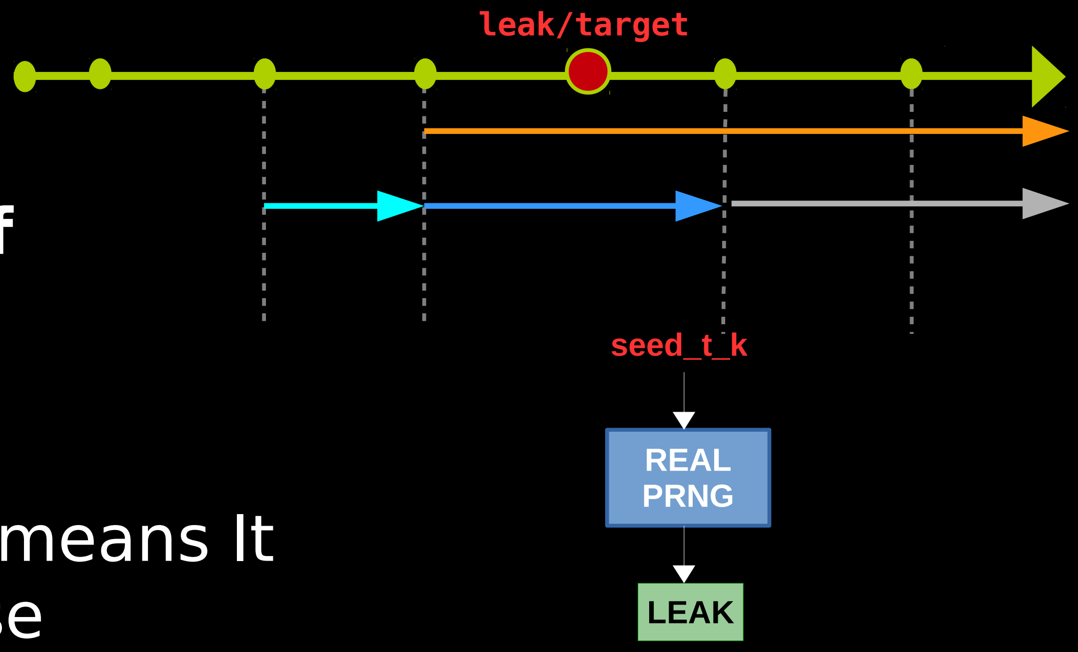


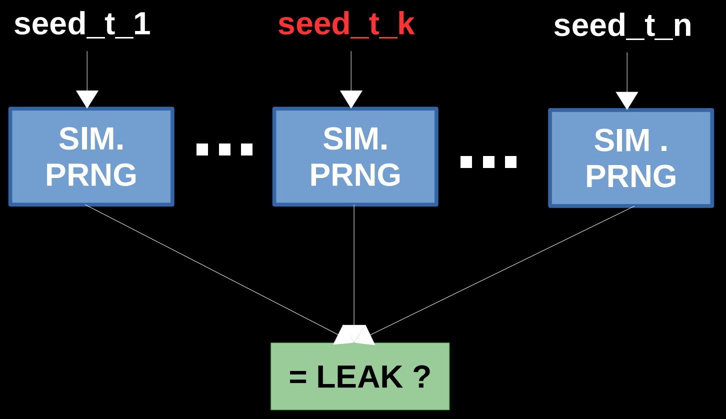


s4_attack_leak_concurrency

Given a leak, what's the probability of finding the original seed ?

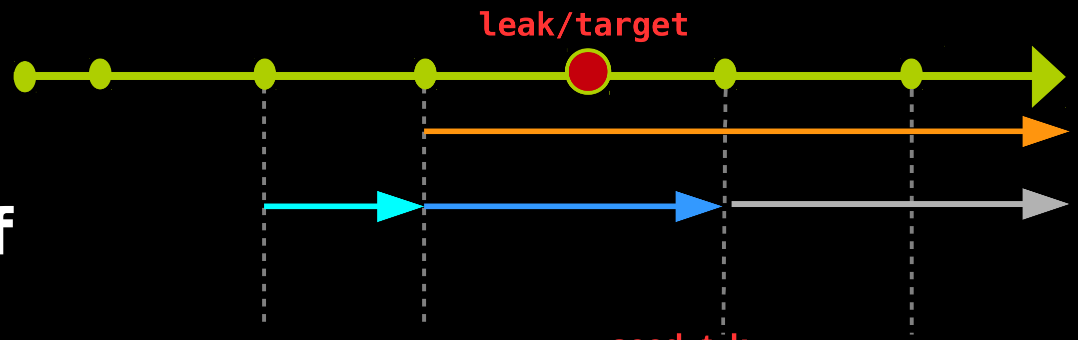
- Zygote's AT_RANDOM is our leak
  It's a platform boot-time leak, which means It occurs in the 'Concurrency Hell' phase



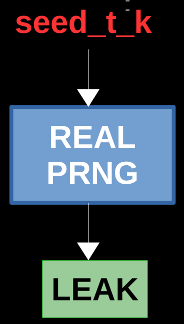


s4_attack_leak_concurrency

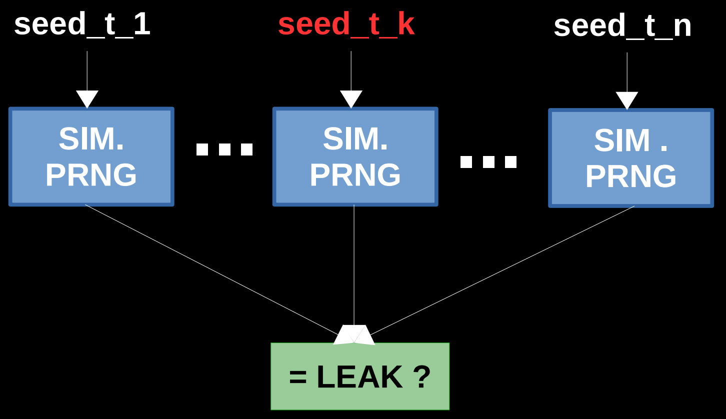
Given a leak, what's the probability of finding the original seed ?



- Zygote's AT_RANDOM is our leak
  It's a platform boot-time leak, which means It occurs in the 'Concurrency Hell' phase

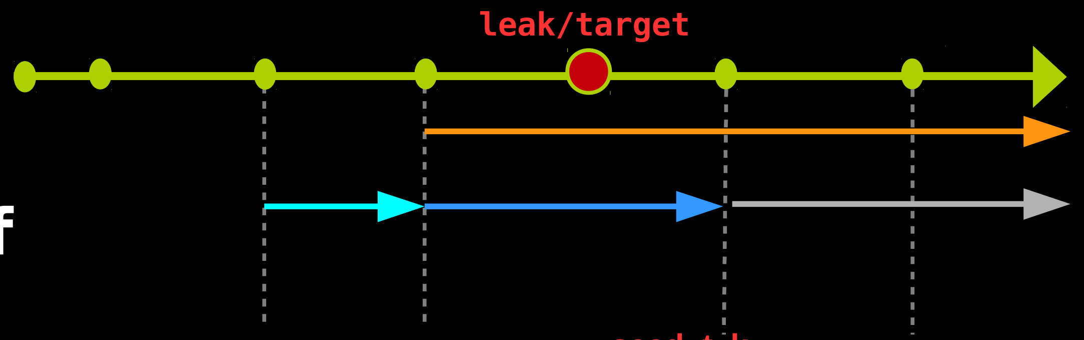


- An offline study of the samples revealed bias towards a specific extraction path from the nonblocking pool



s4_attack_leak_concurrency

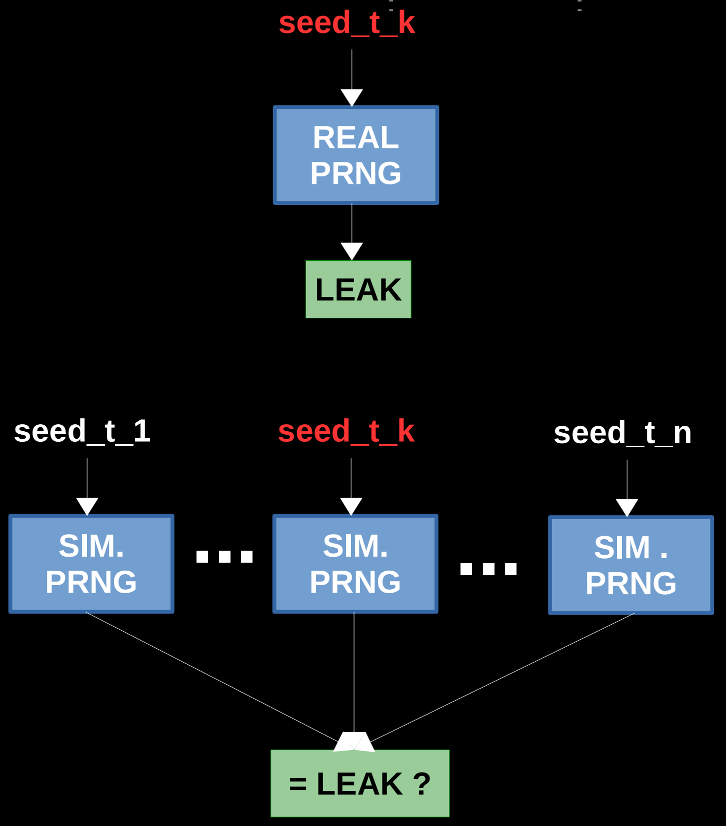
Given a leak, what's the probability of finding the original seed ?



- Zygote's AT_RANDOM is our leak
  It's a platform boot-time leak, which means It occurs in the 'Concurrency Hell' phase

- An offline study of the samples revealed bias towards a specific extraction path from the nonblocking pool

- 20% of the samples had Zygote's AT_RANDOM bytes somewhere in the extraction path

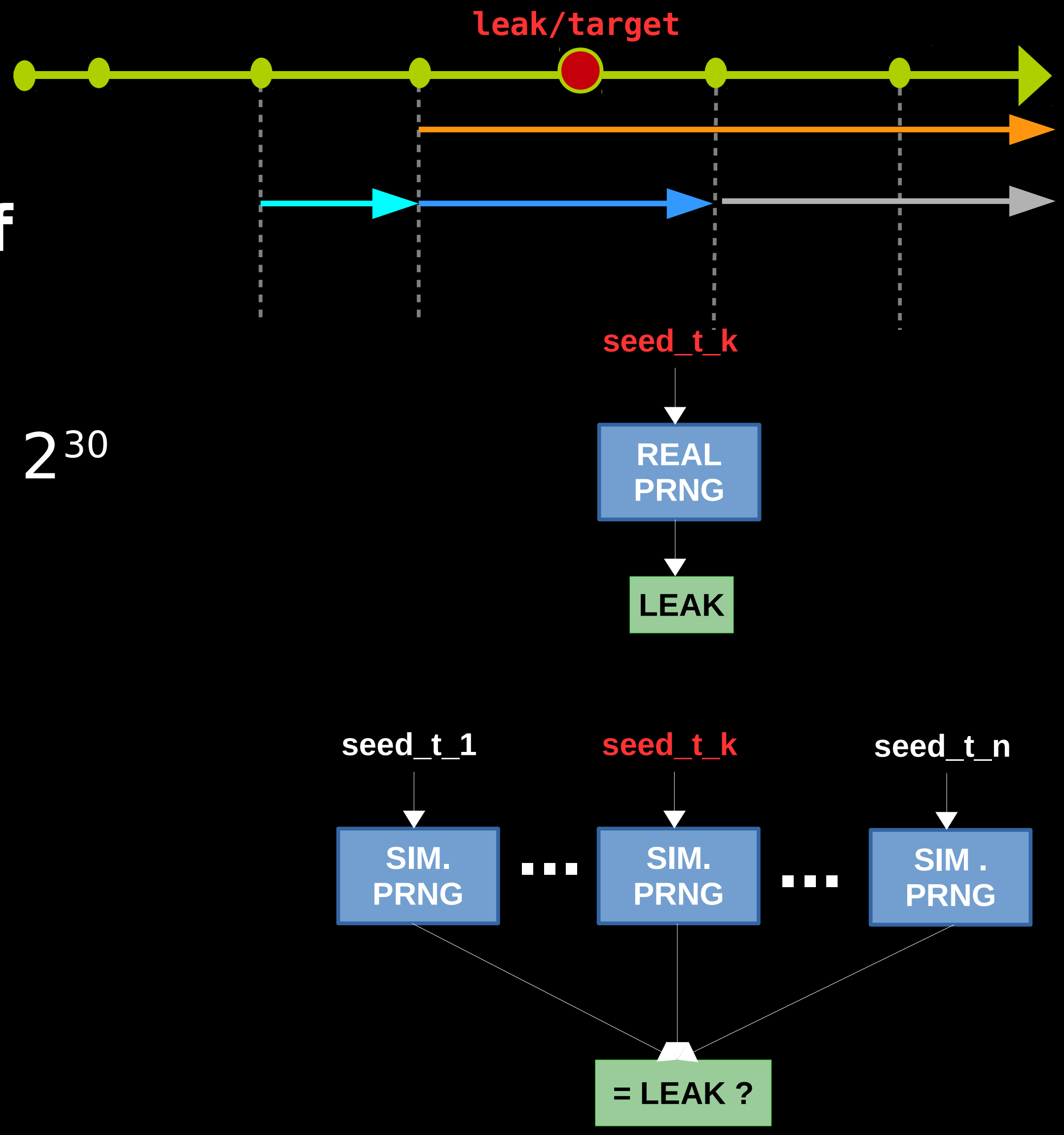


s4_attack_leak_concurrency

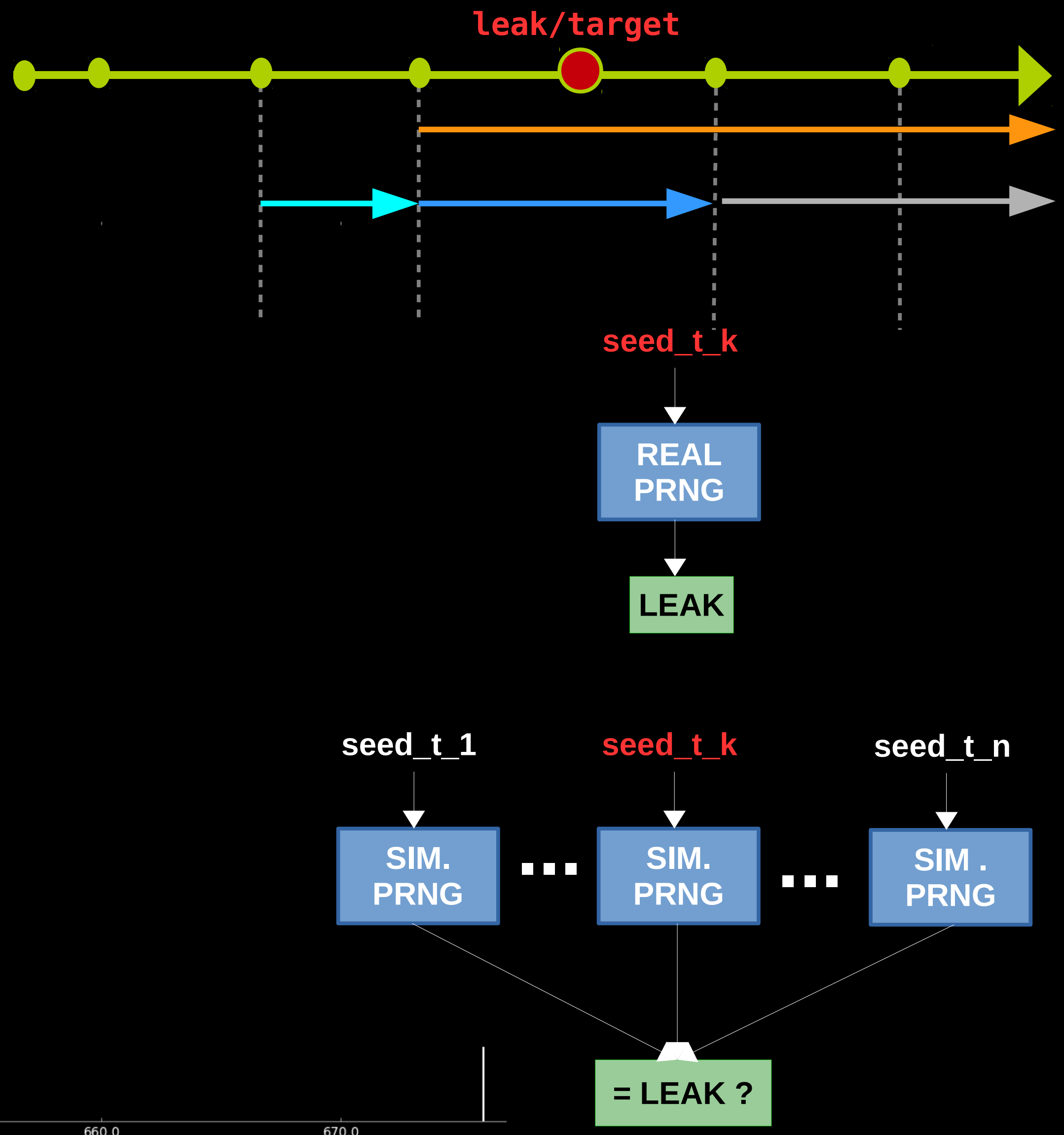
Given a leak, what's the probability of finding the original seed ?

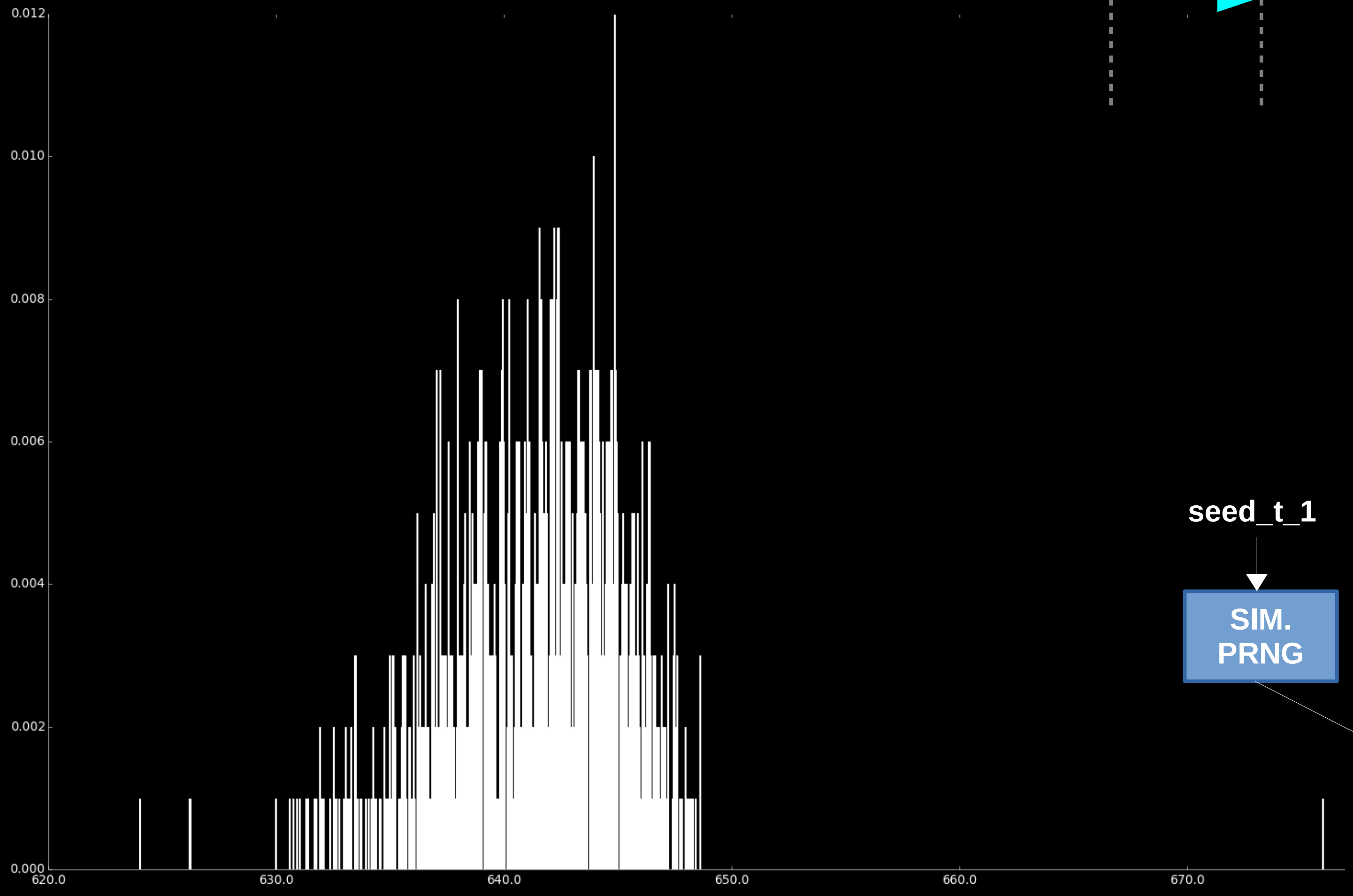
- Given a leak and assuming we try all 2^{30} possible seeds the chance is

$$\frac{1}{5}$$



s4_non-blocking_seed

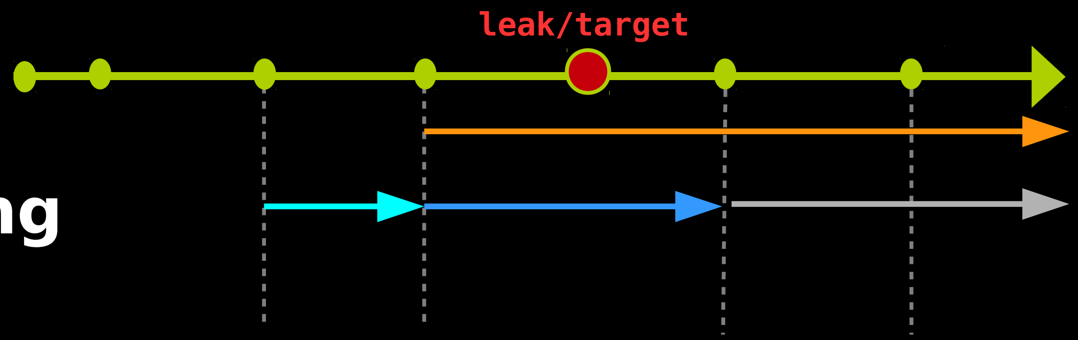


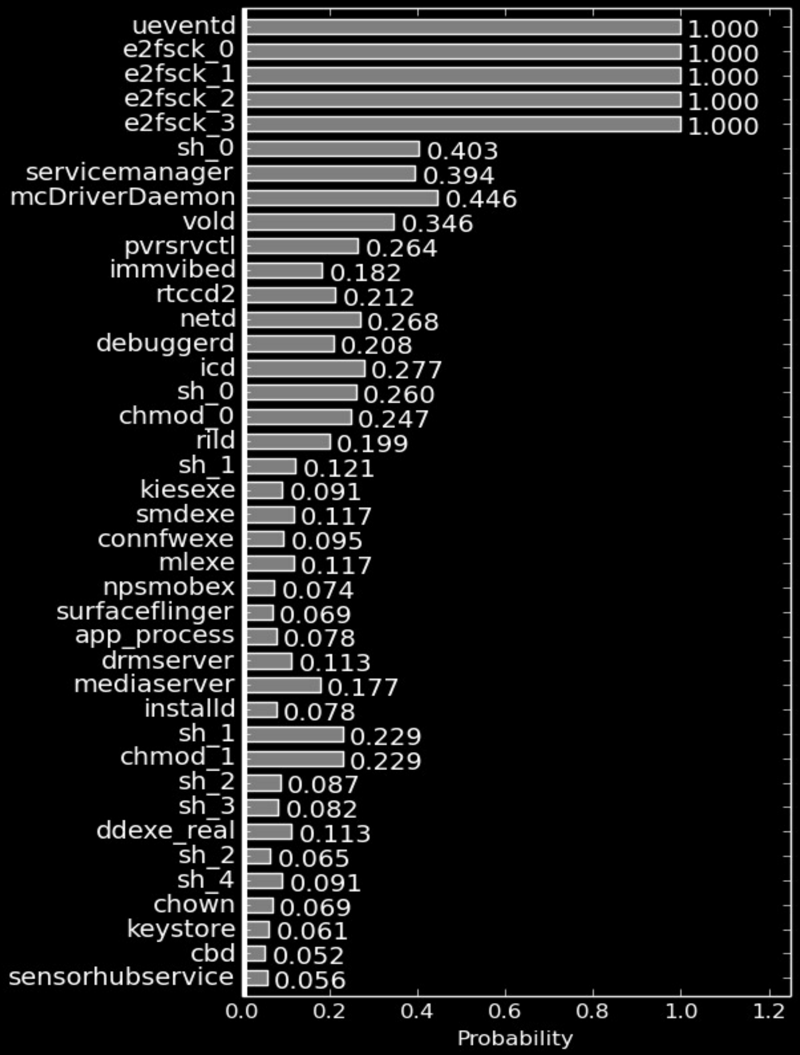


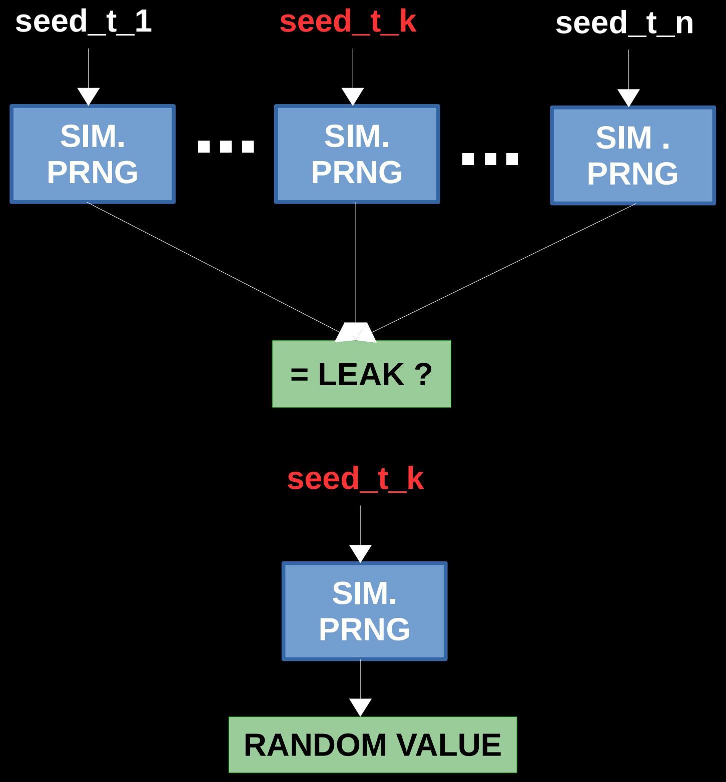
$$H(s_{nb}) = 23.5\,bits$$

s4_attack_targets

Given a seed, Probabilities of finding the canary of early boot services

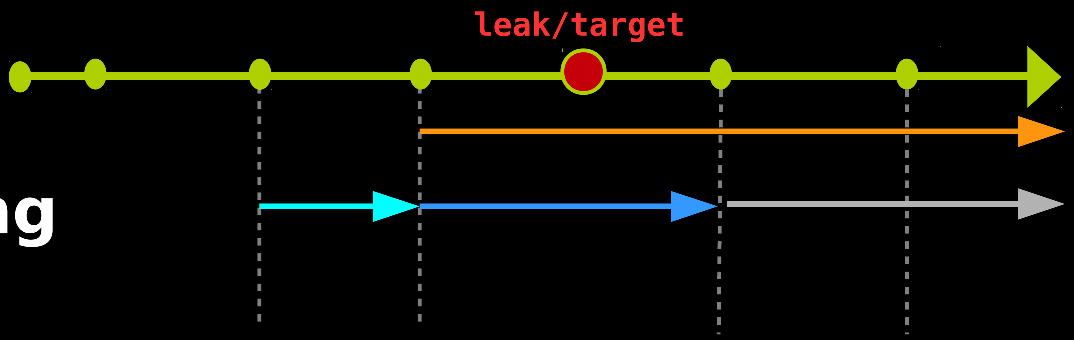


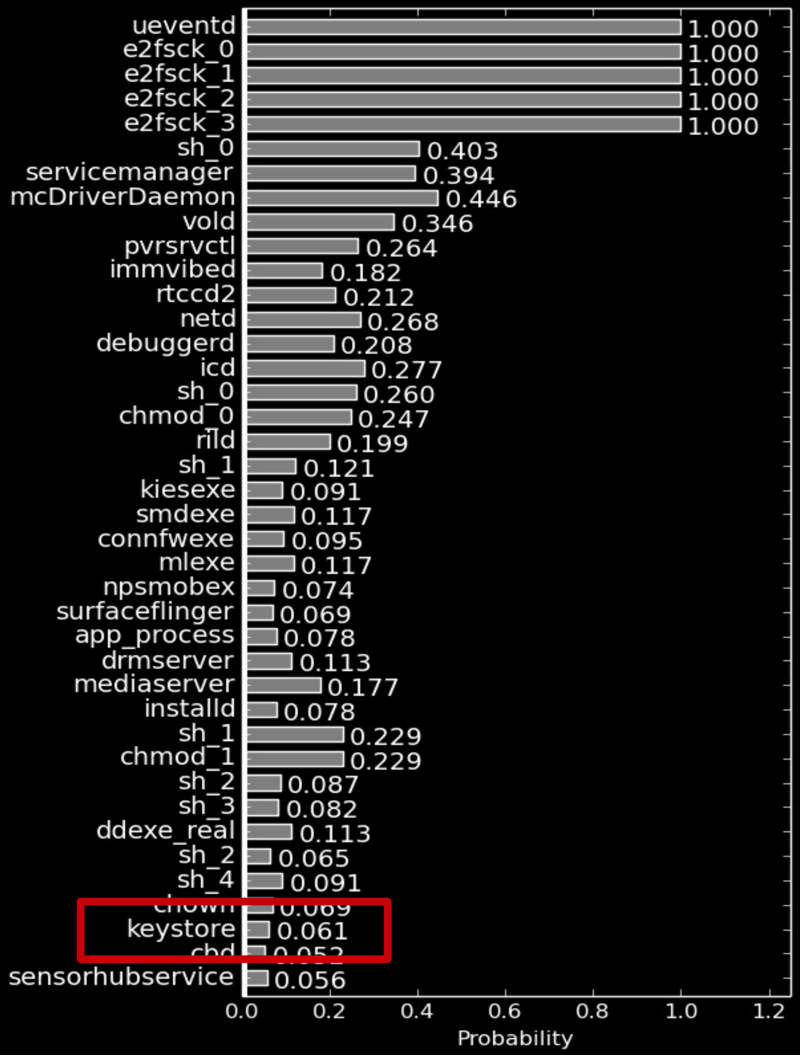




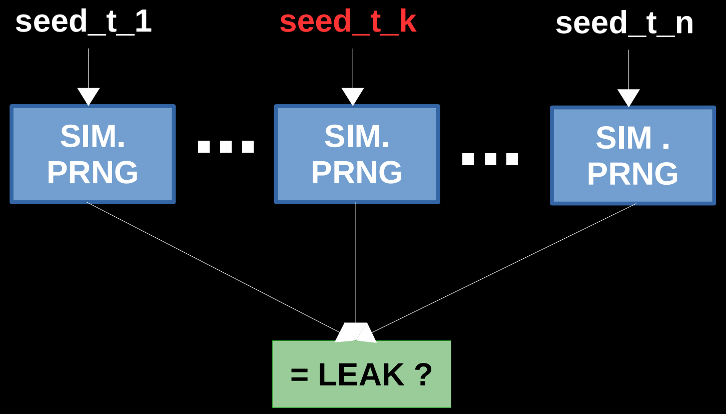
s4_attack_targets

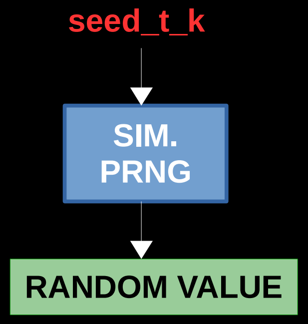
Given a seed, Probabilities of finding the canary of early boot services

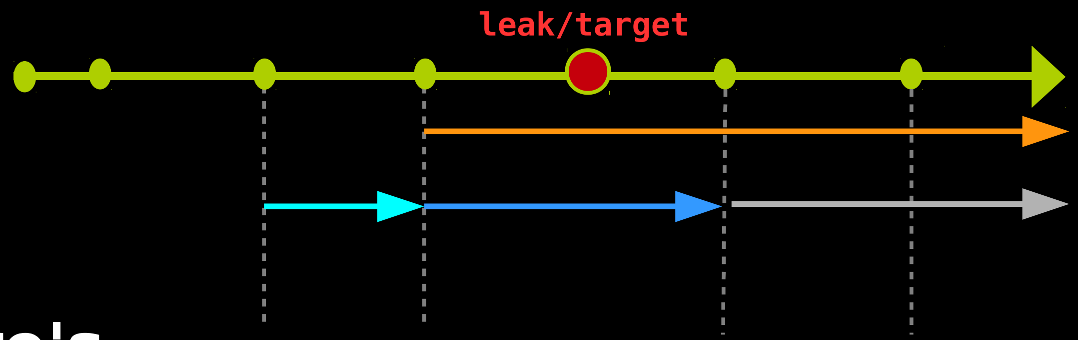




$$\frac{6}{100}$$



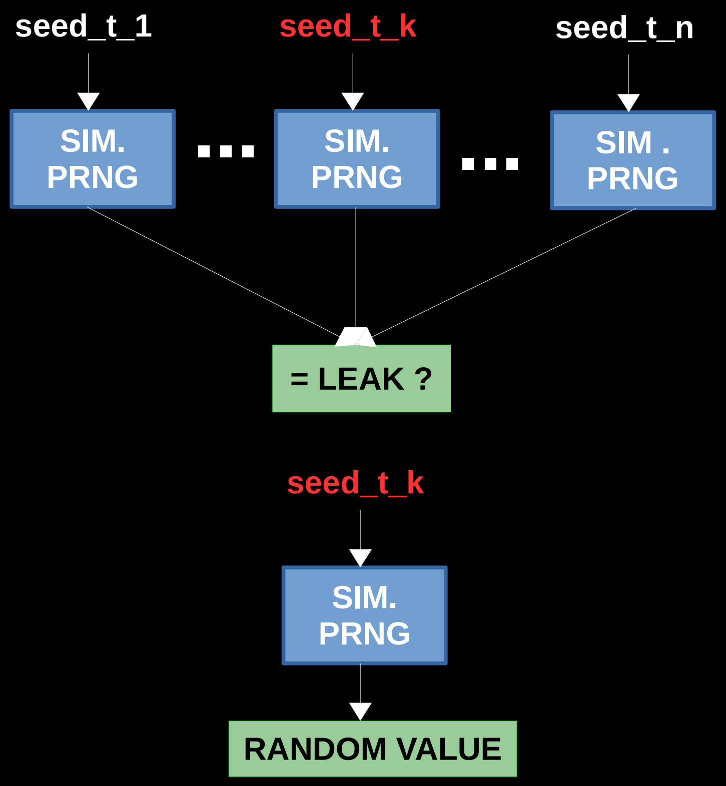




s4_attack_targets

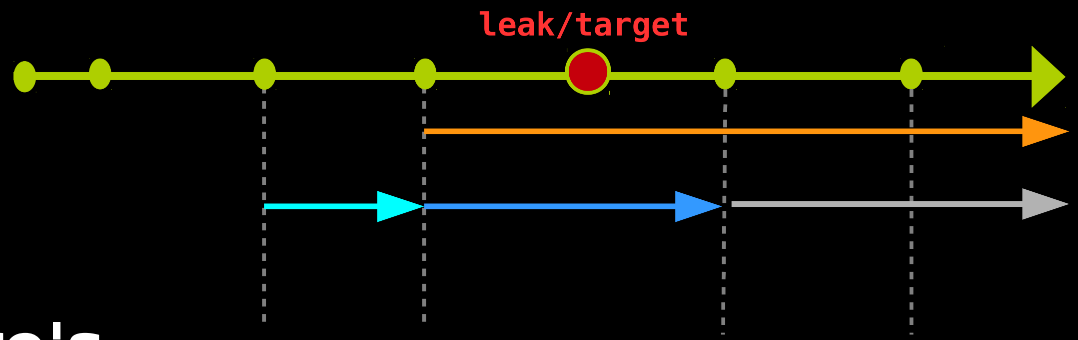
Given Zygote's AT_RANDOM, the probability of guessing the Keystore's canary value is:

$$\frac{1}{5}\cdot\frac{6}{100}\simeq 0.01 \rightarrow 1\%$$



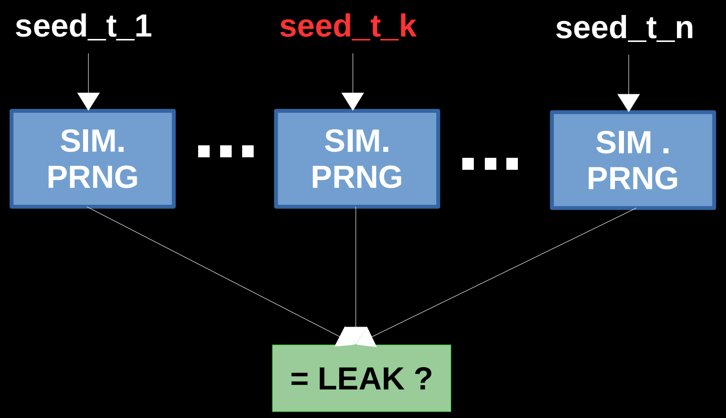
Remember where we came from...
we needed to guess 32 random bits

s4_attack_targets

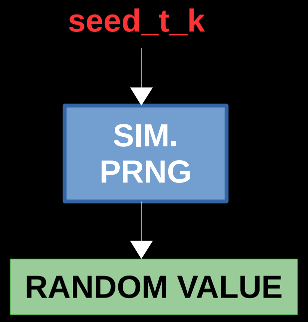


Given Zygote's AT_RANDOM, the probability of guessing the Keystore's canary value is:

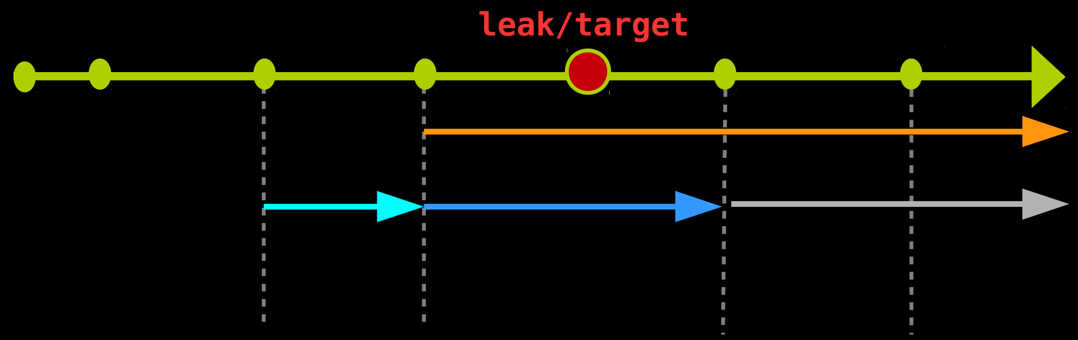
$$\frac{1}{5} \cdot \frac{6}{100} \simeq 0.01 \rightarrow 1\%$$



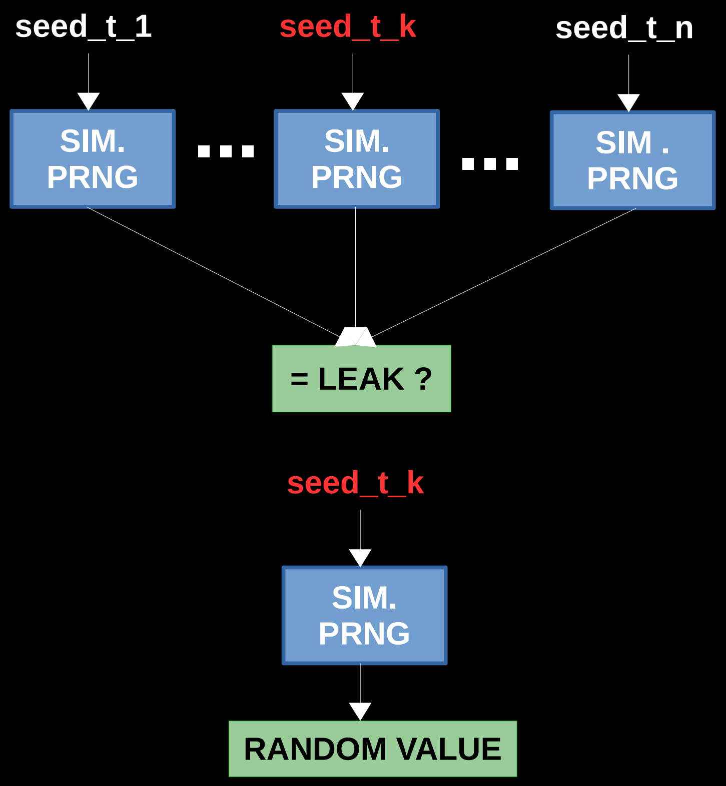
$$\frac{1}{2^{32}} \simeq 0.0000000023 \rightarrow 0.00000023\,\%$$



s4_demo


leak/target

DEMO

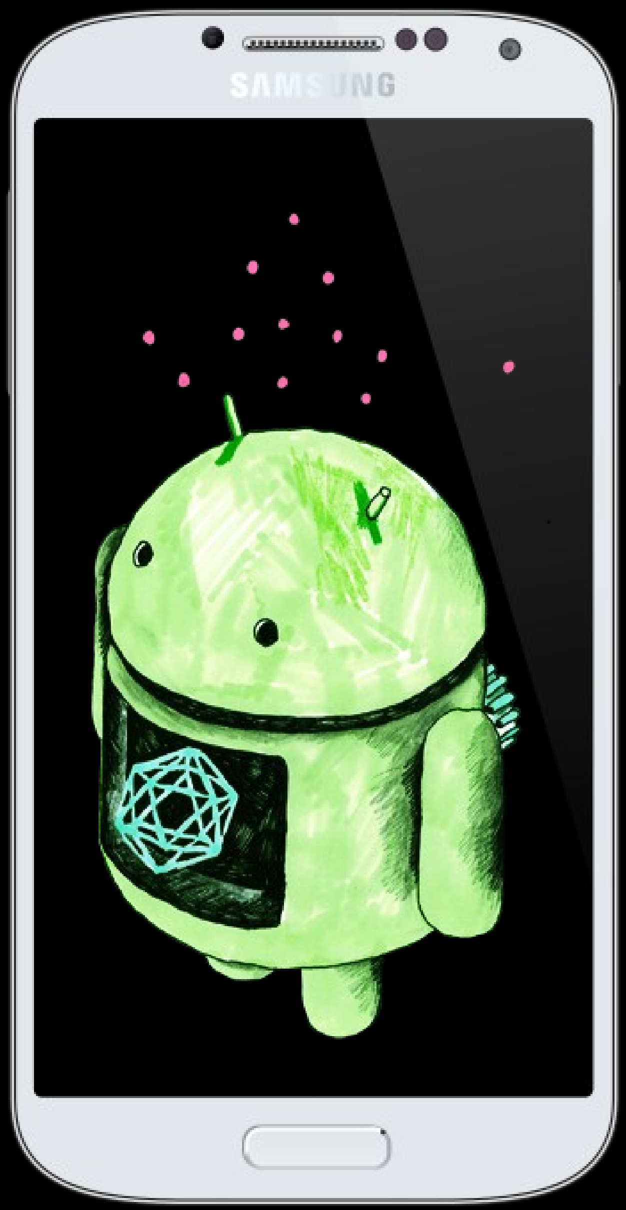

seed_t_1
seed_t_k
seed_t_n
SIM. PRNG
...
SIM. PRNG
...
SIM . PRNG
= LEAK ?
seed_t_k
SIM. PRNG
RANDOM VALUE

2nd Attack Vector
Ping6 → PRNG Seed → IPv6 Fragment Injection & Getting Keystore's Canary

s2_offline_study

Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2

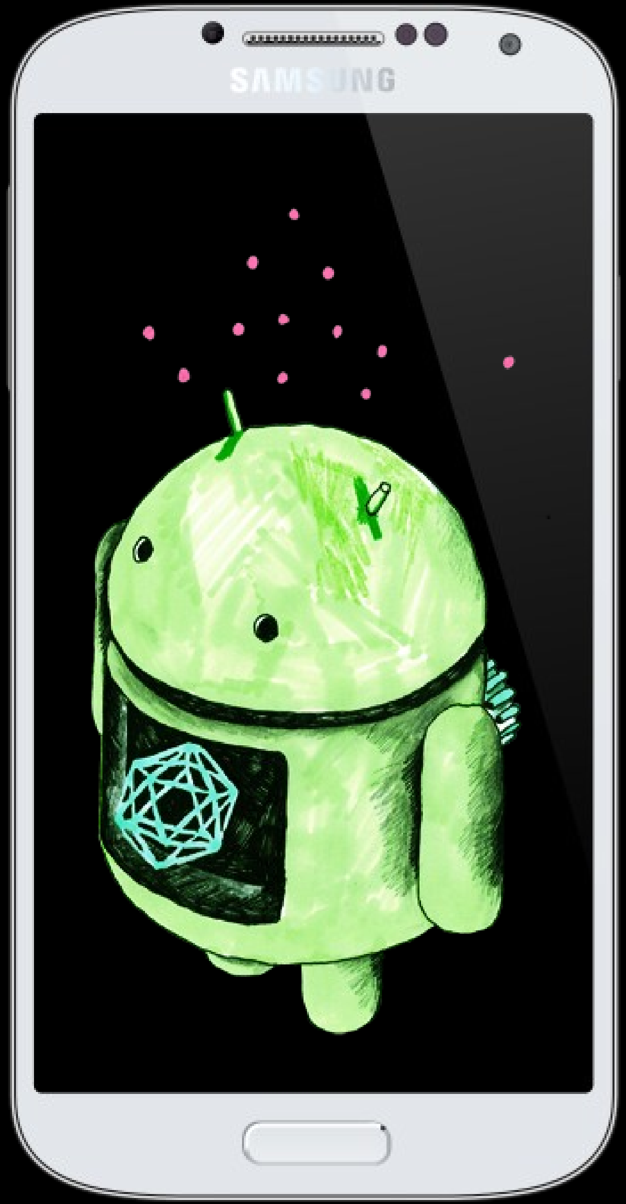


s2_offline_study

Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets



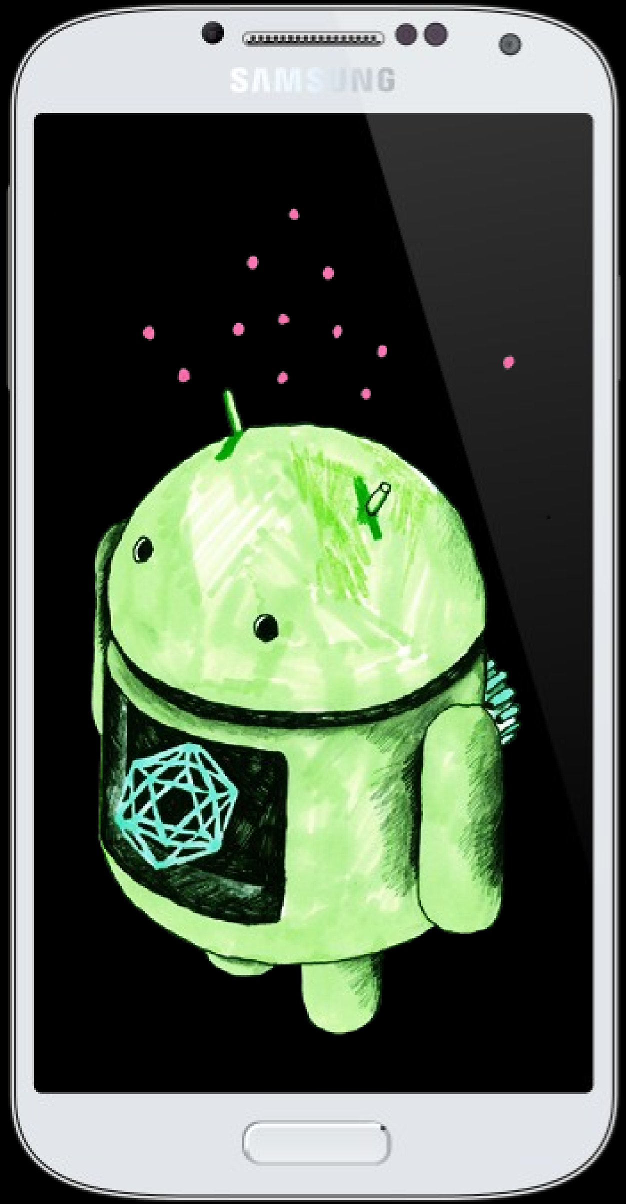
s2_offline_study

Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets

- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.



s2_offline_study

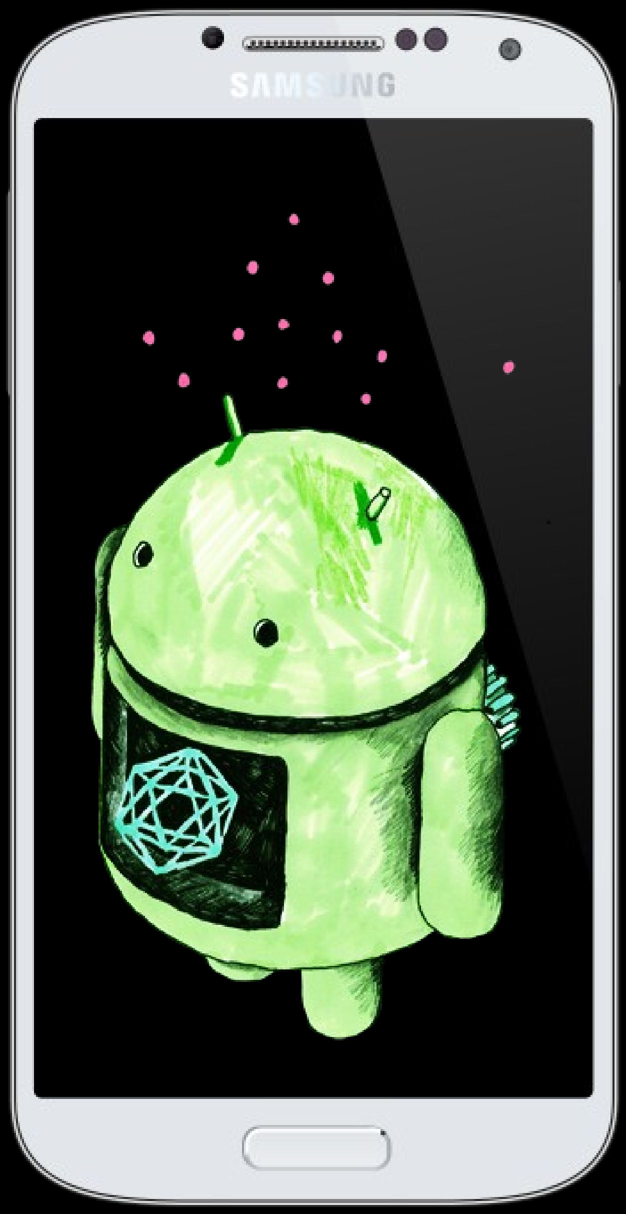
Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2

- **printk**() input and nonblocking pool seeds - find a bias in the seed value
- **printk**() get_random_bytes() callers and amount of random bytes requested – find leak and attack targets

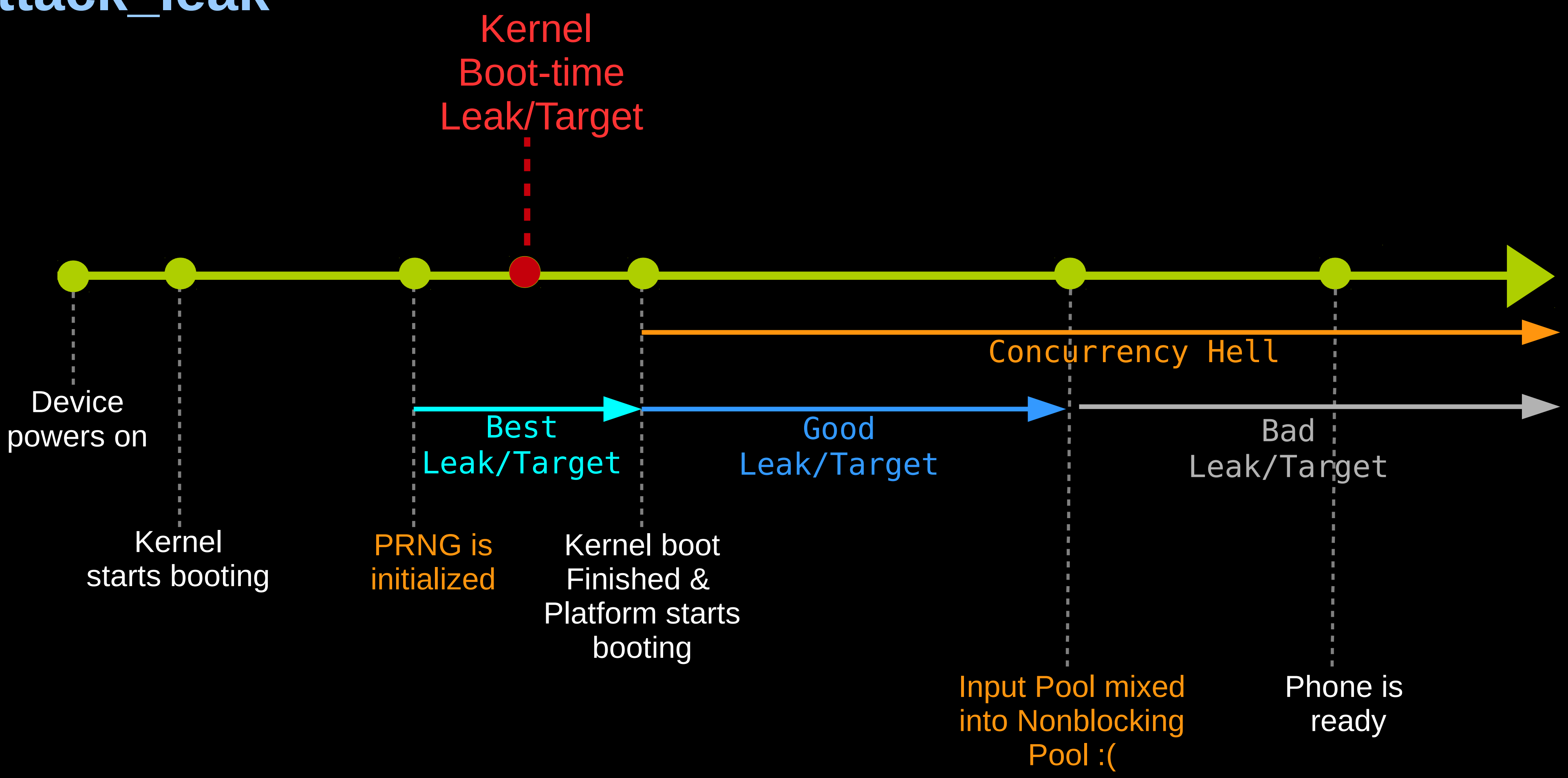
- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.

- In total, we rebooted(script) the device more than 2000 times, each time we dumped the kernel ring buffer to a file.



s2_attack_leak

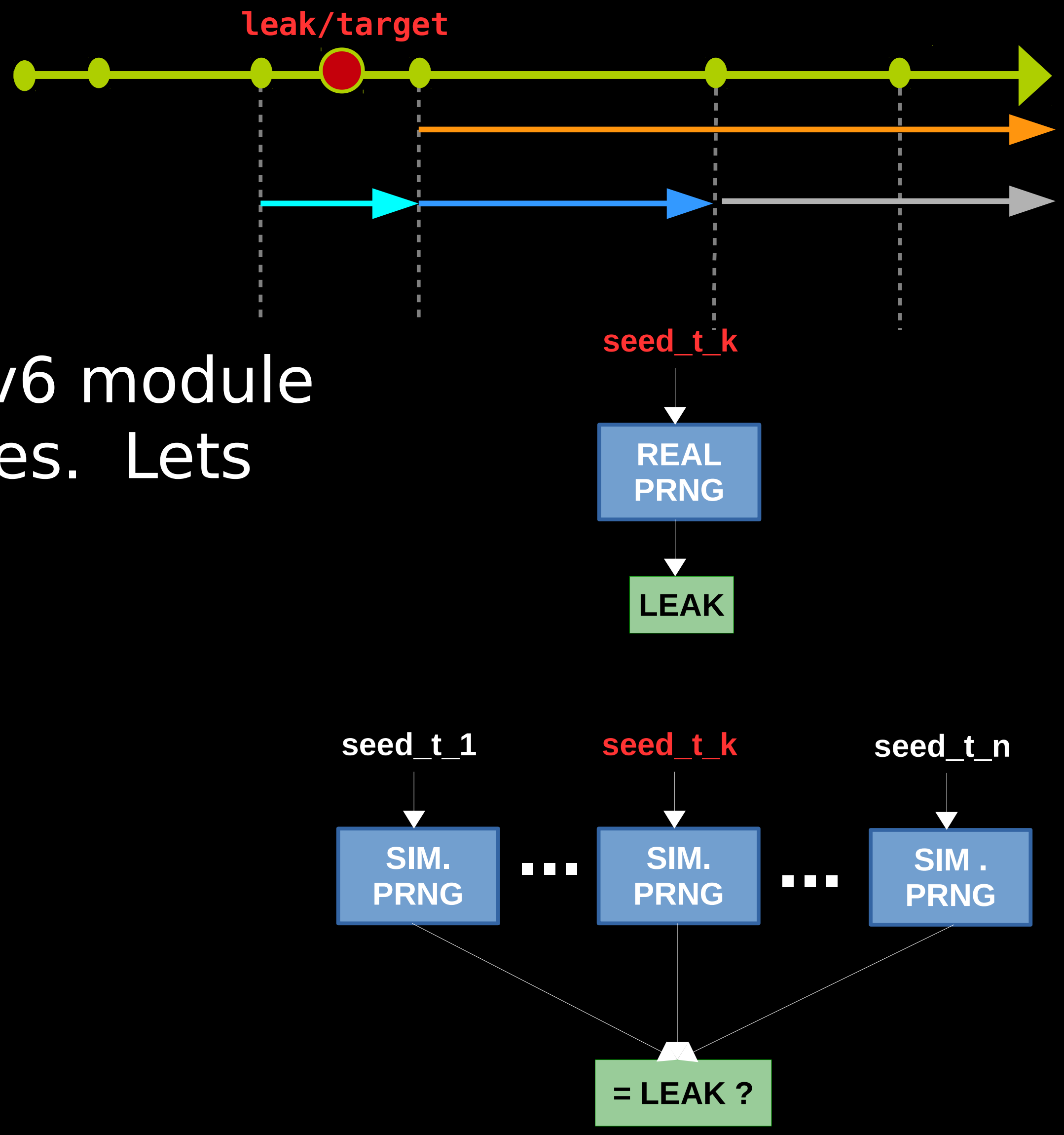
Details

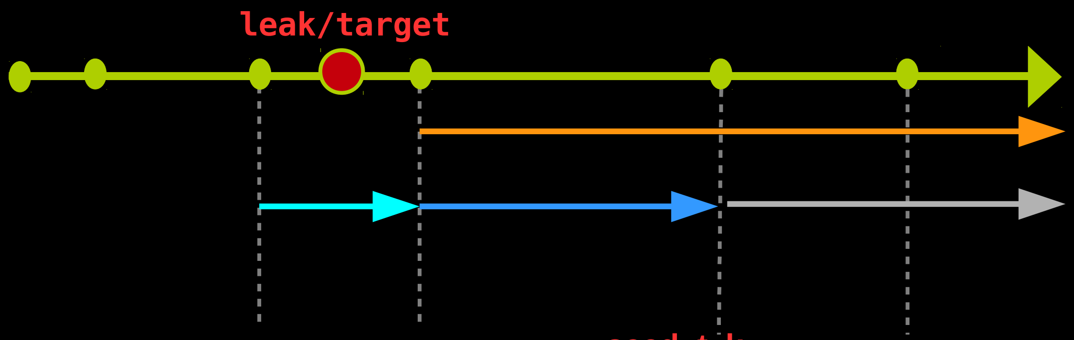


s2_attack_leak

Details

- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Lets call them **rand**.



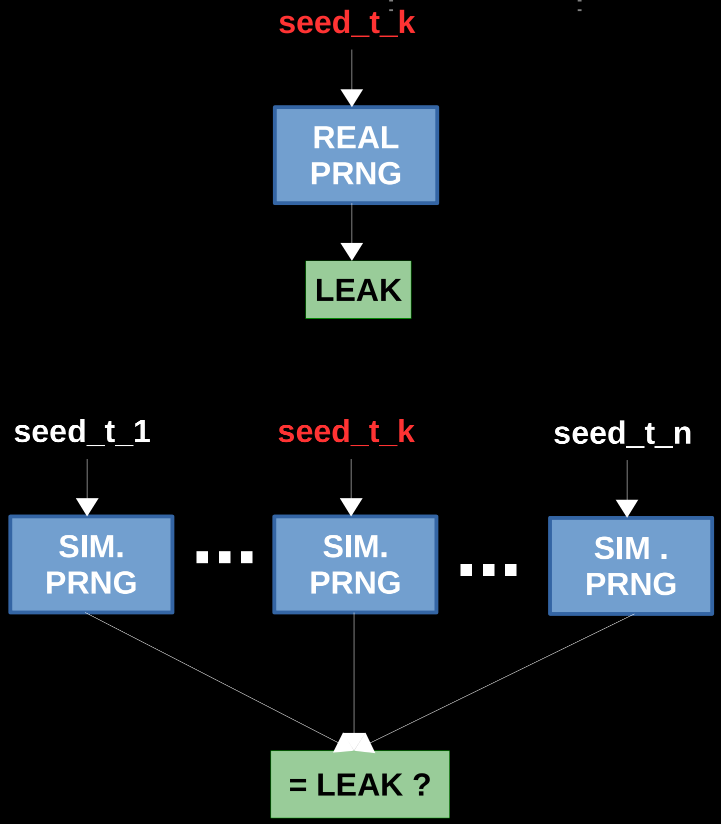


s2_attack_leak

Details

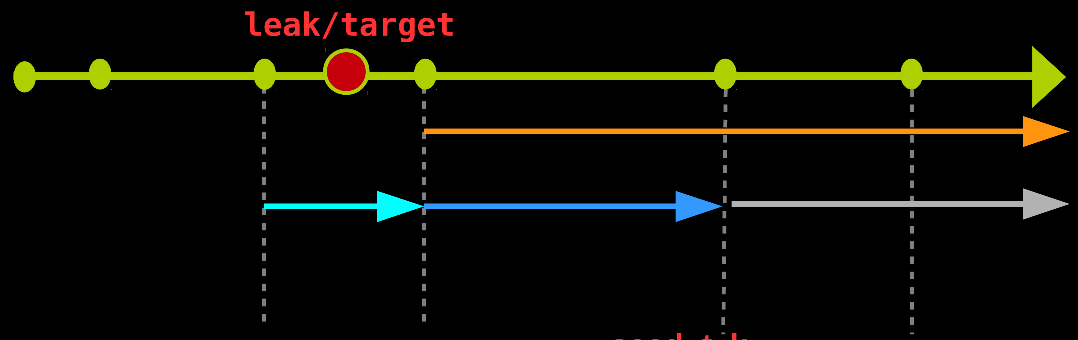
- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Lets call them **rand**.
- IPv6 packet fragment identifier is computed by a deterministic function.

$$f(\texttt{rand, ipv6\_dst\_addr}) = \texttt{ipv6\_frag\_id}$$



s2_attack_leak

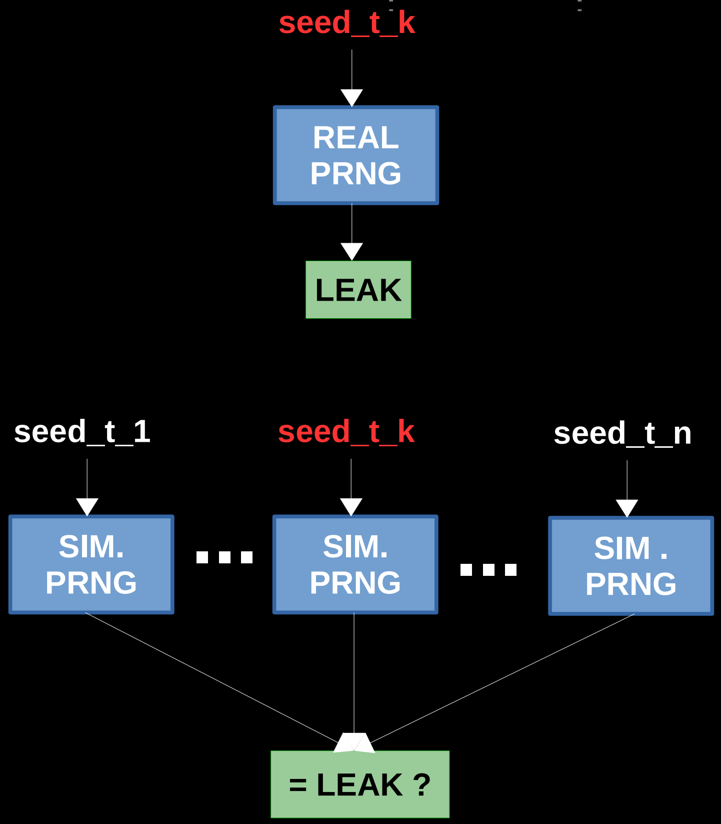
Details



- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Lets call them **rand**.
- IPv6 packet fragment identifier is computed by a deterministic function.

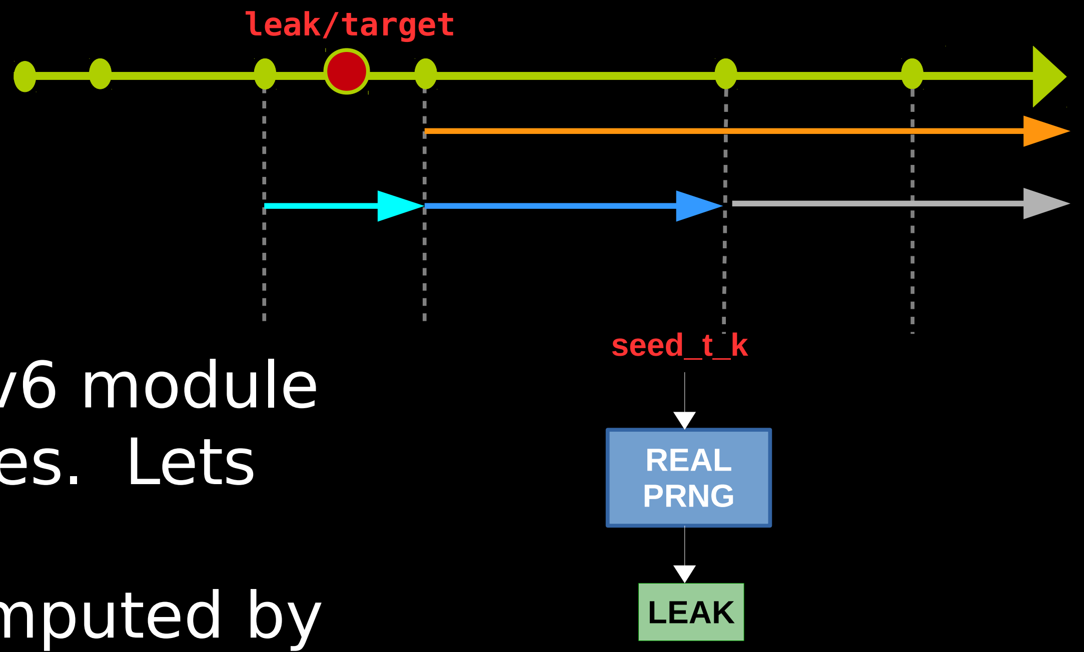
$$f(\texttt{rand},\ \texttt{ipv6\_dst\_addr}) = \texttt{ipv6\_frag\_id}$$

- The pair (**ipv6_dst_addr**,**ipv6_frag_id**) is our leak. Why?



s2_attack_leak

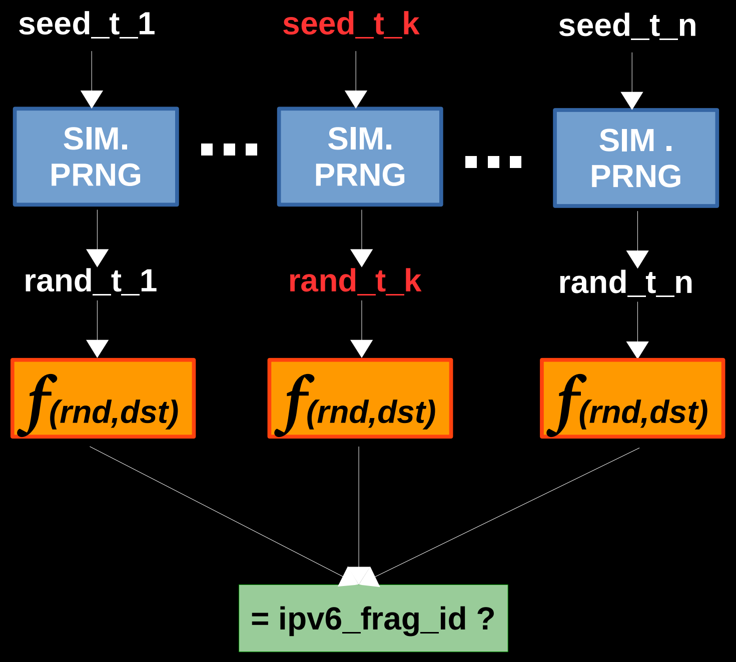
Details



- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Lets call them **rand**.
- IPv6 packet fragment identifier is computed by a deterministic function.

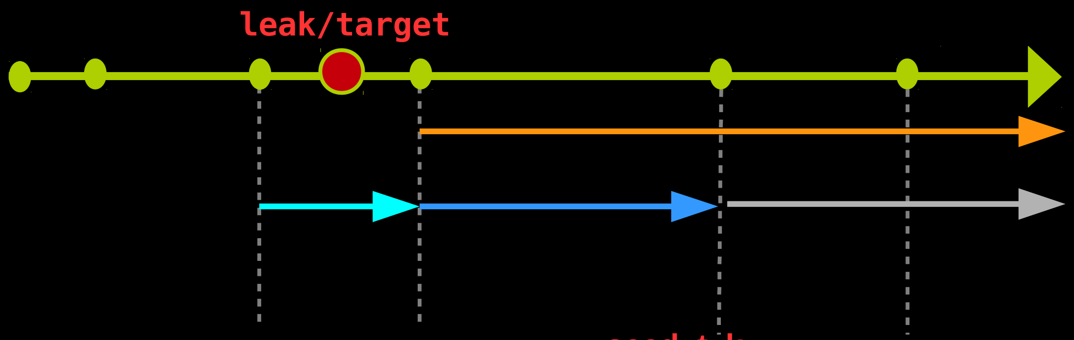
$$f(\texttt{rand, ipv6\_dst\_addr}) = \texttt{ipv6\_frag\_id}$$

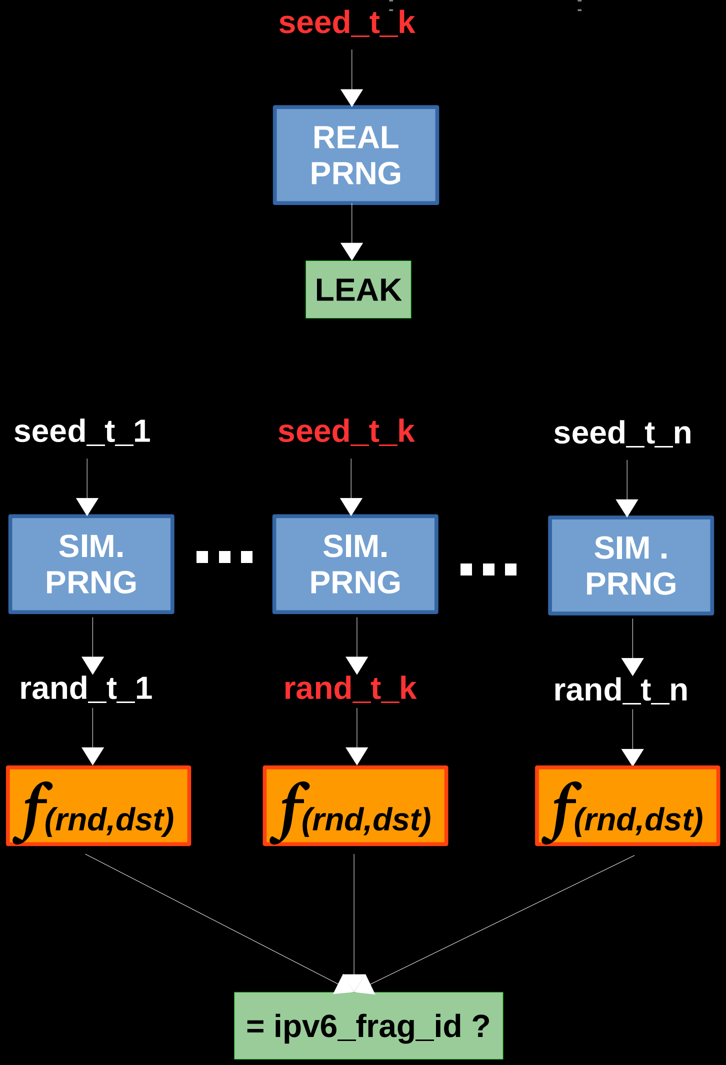
- The pair `(ipv6_dst_addr,ipv6_frag_id)` is our leak. Why?
- We simulate PRNGs up to **rand**, and feed it to the deterministic function f



s2_attack_leak

Details

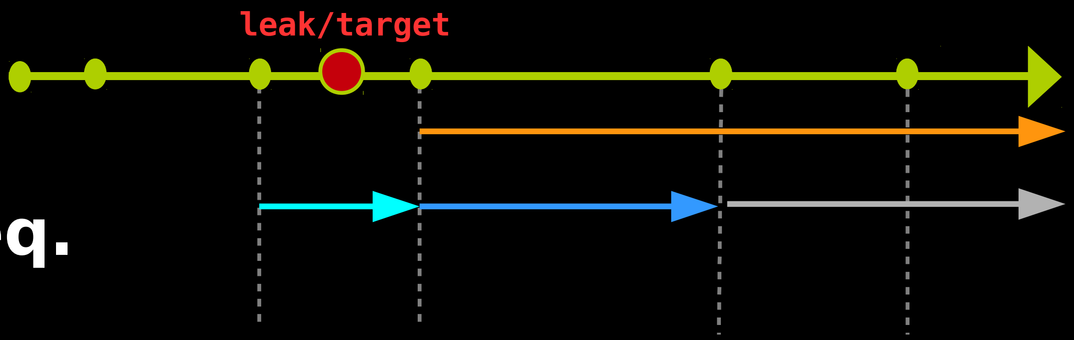




- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Lets call them **rand**.
- IPv6 packet fragment identifier is computed by a deterministic function.

$$f(\texttt{rand},\ \texttt{ipv6\_dst\_addr}) = \texttt{ipv6\_frag\_id}$$

- The pair (**ipv6_dst_addr,ipv6_frag_id**) is our leak. Why?
- We simulate PRNGs up to **rand**, and feed it to the deterministic function f
- OK, fine, but how did you get **ipv6_dst_addr?**



s2_attack_leak

IPv6 fragmentation & ICMPv6 Echo Req.

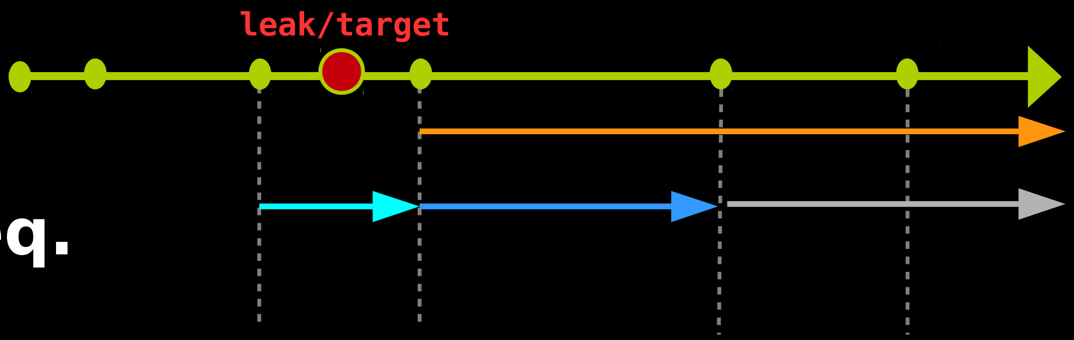
- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.

0	8	16	24	31
Vers	Flow Label			
Payload Length		Next Header	Hop Limit	
Source Address (16 bytes)				
Destination Address (16 bytes)				

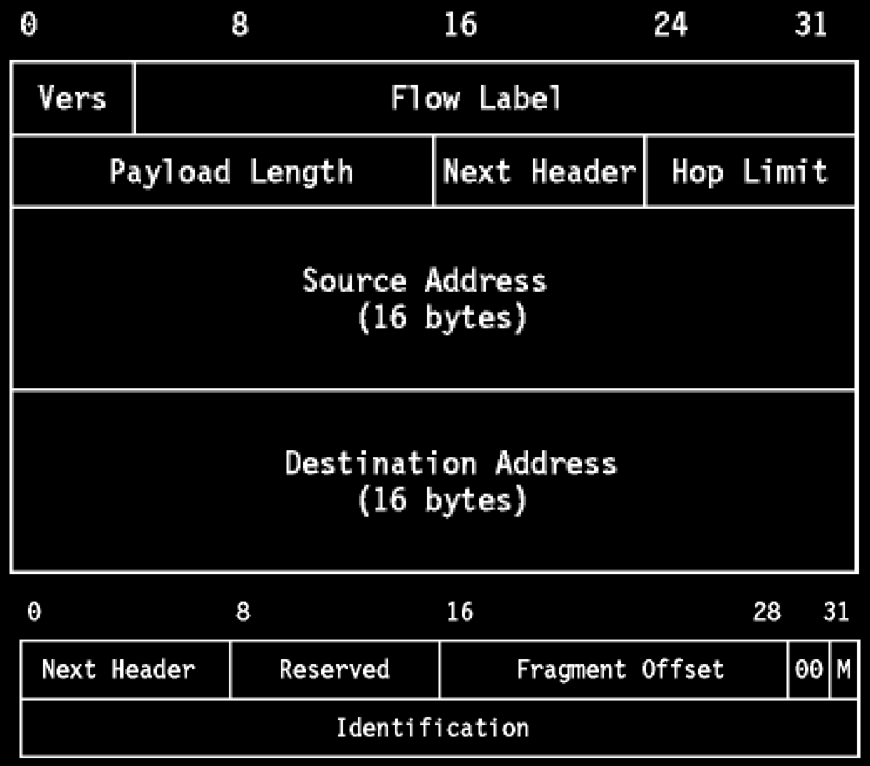
0	8	16	28	31
Next Header	Reserved	Fragment Offset	00	M
Identification				

s2_attack_leak

IPv6 fragmentation & ICMPv6 Echo Req.

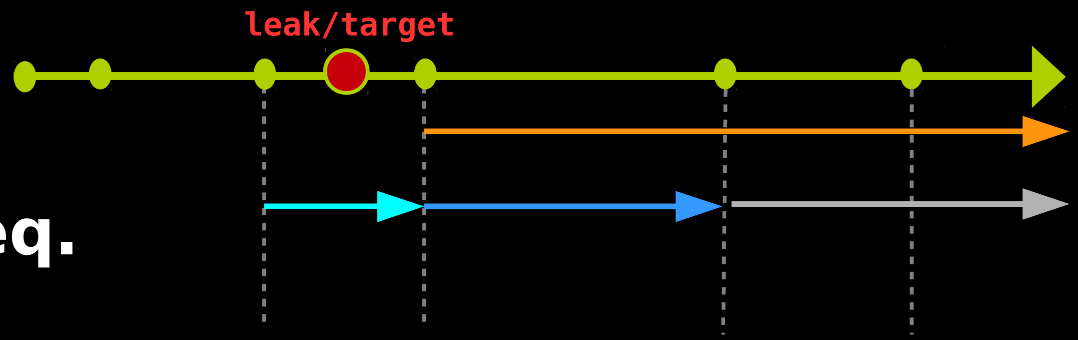


- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.
- Each fragment of the packet contains the same fragment id. Which is used by the receiver to identify fragments of a packet.

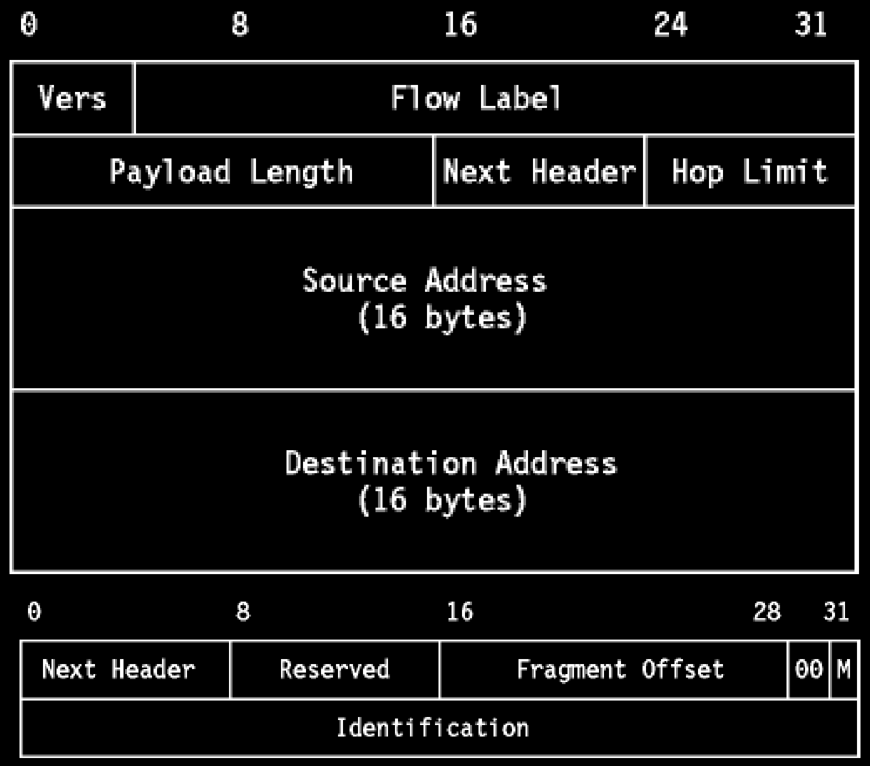


s2_attack_leak

IPv6 fragmentation & ICMPv6 Echo Req.

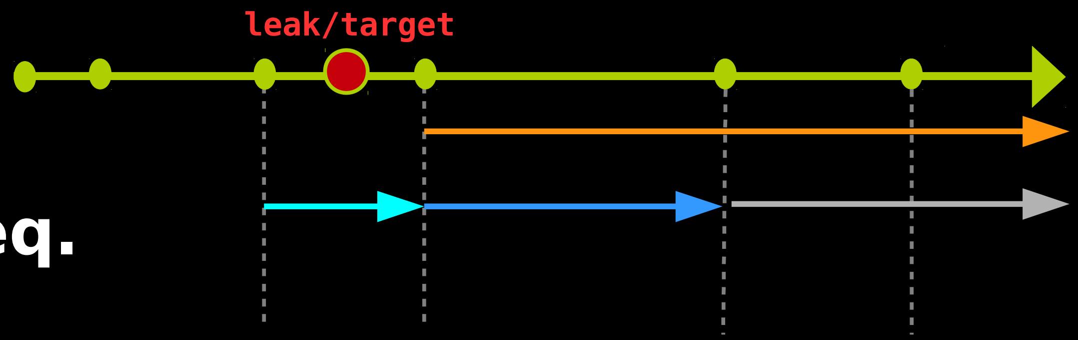


- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.
- Each fragment of the packet contains the same fragment id. Which is used by the receiver to identify fragments of a packet.
- IPv6 fragmentation doesn't happen very often. How do we make it happen ?



s2_attack_leak

IPv6 fragmentation & ICMPv6 Echo Req.



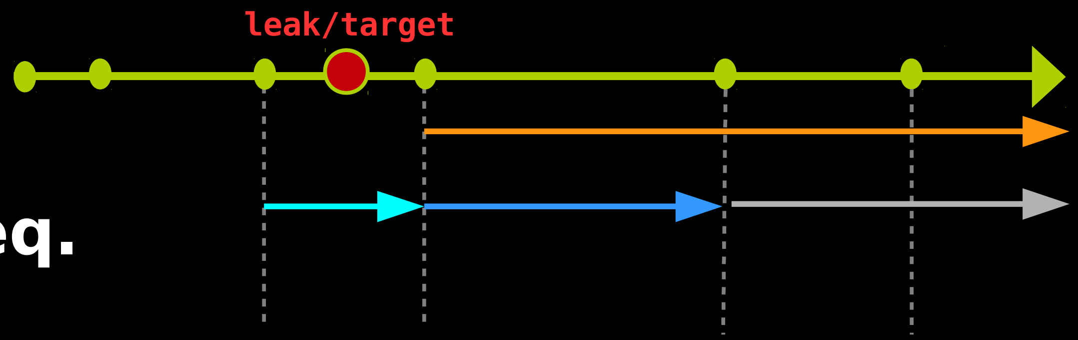
- Ping6 – a utility for sending ICMPv6 Echo Requests which requires the target to send an ICMPv6 Echo Replay with the exactly the same data.

0	8	16	24	31
Vers	Flow Label			
Payload Length		Next Header	Hop Limit	
Source Address (16 bytes)				
Destination Address (16 bytes)				

0	8	16	28	31
Next Header	Reserved	Fragment Offset	00	M
Identification				

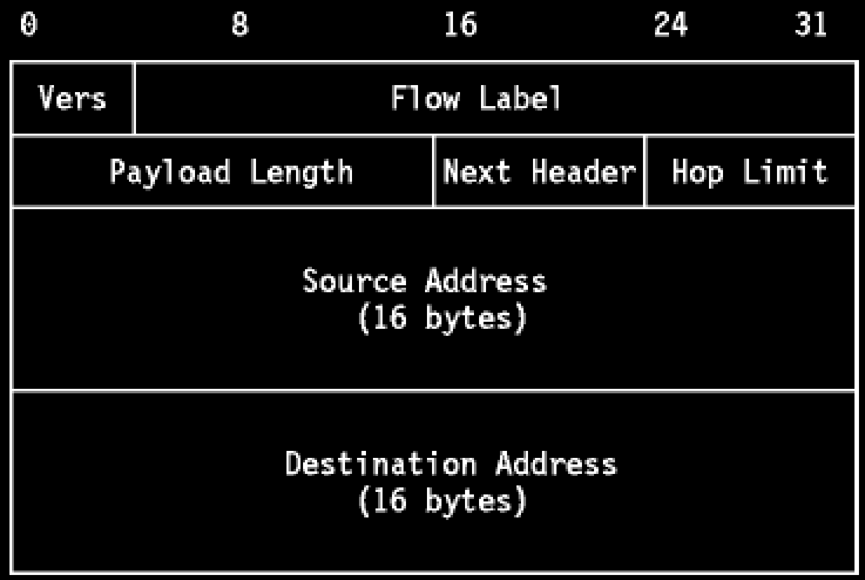
s2_attack_leak

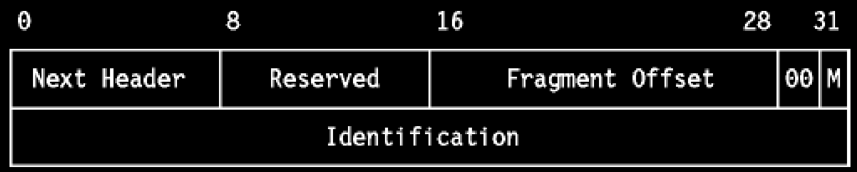
IPv6 fragmentation & ICMPv6 Echo Req.



- Ping6 – a utility for sending ICMPv6 Echo Requests which requires the target to send an ICMPv6 Echo Replay with the exactly the same data.

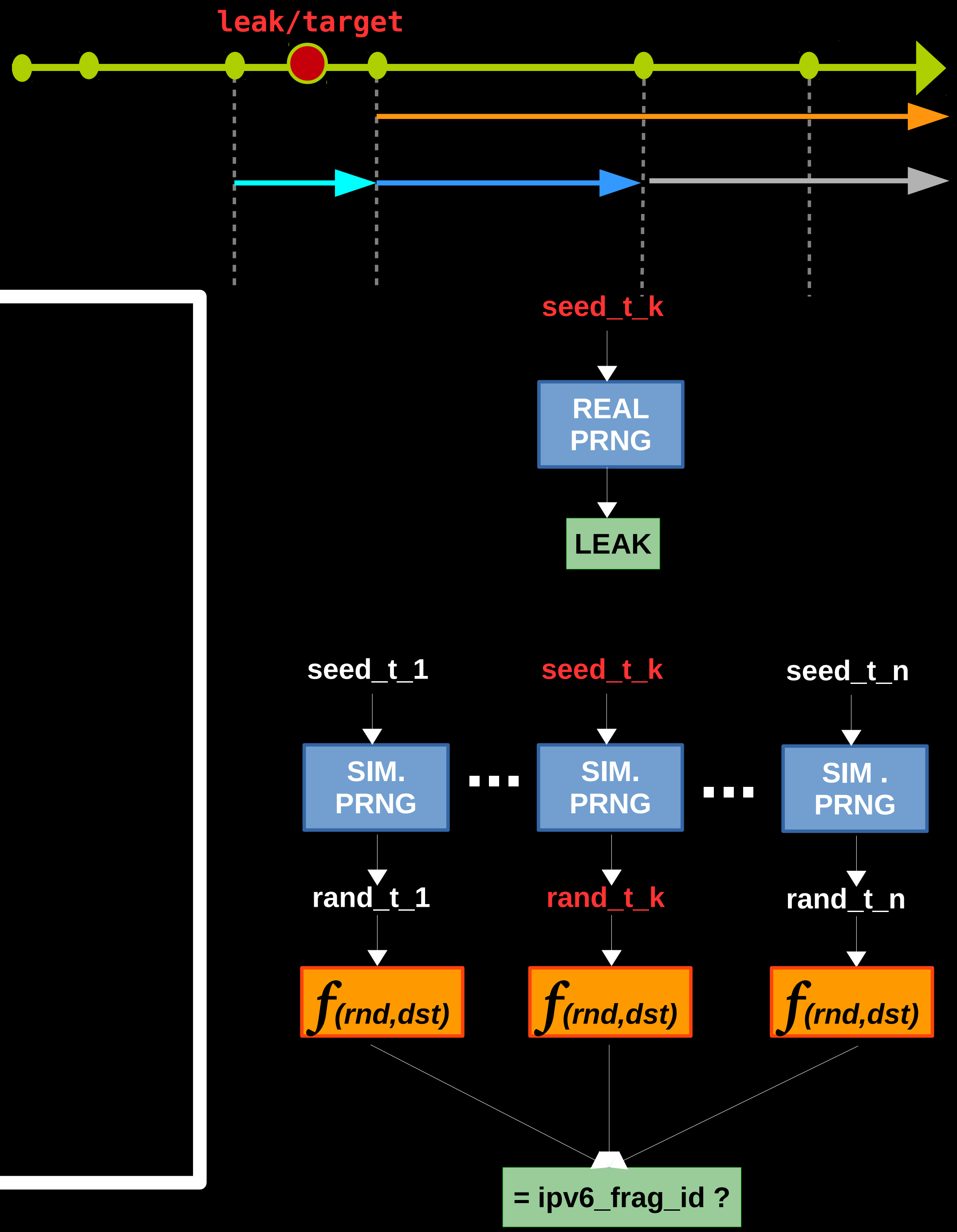
- Result: Sending ICMPv6 Echo Request with data > MTU will make the receiver send a fragmented reply





s2_attack_get_leak

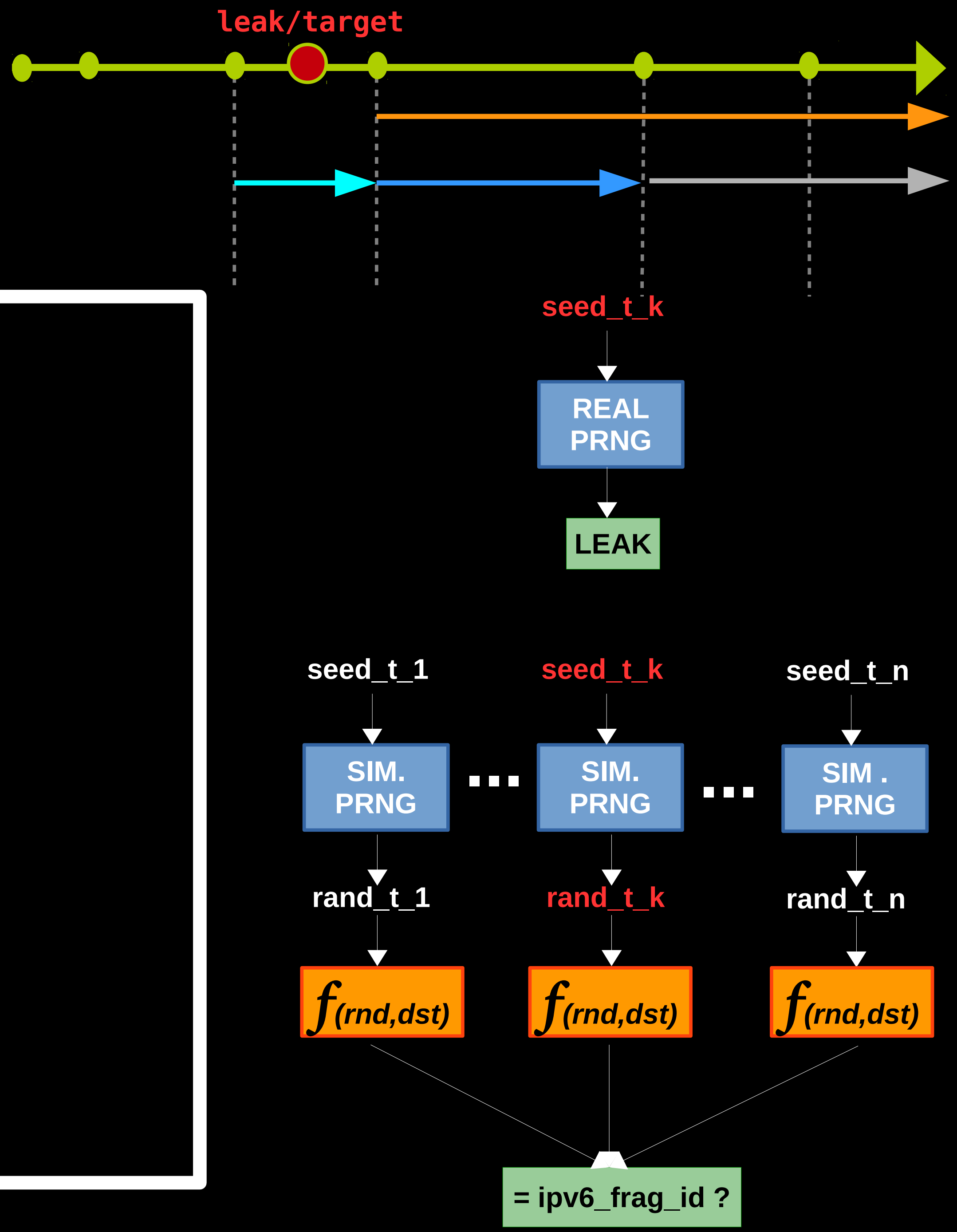
Amsterdam
Schiphol
Airport



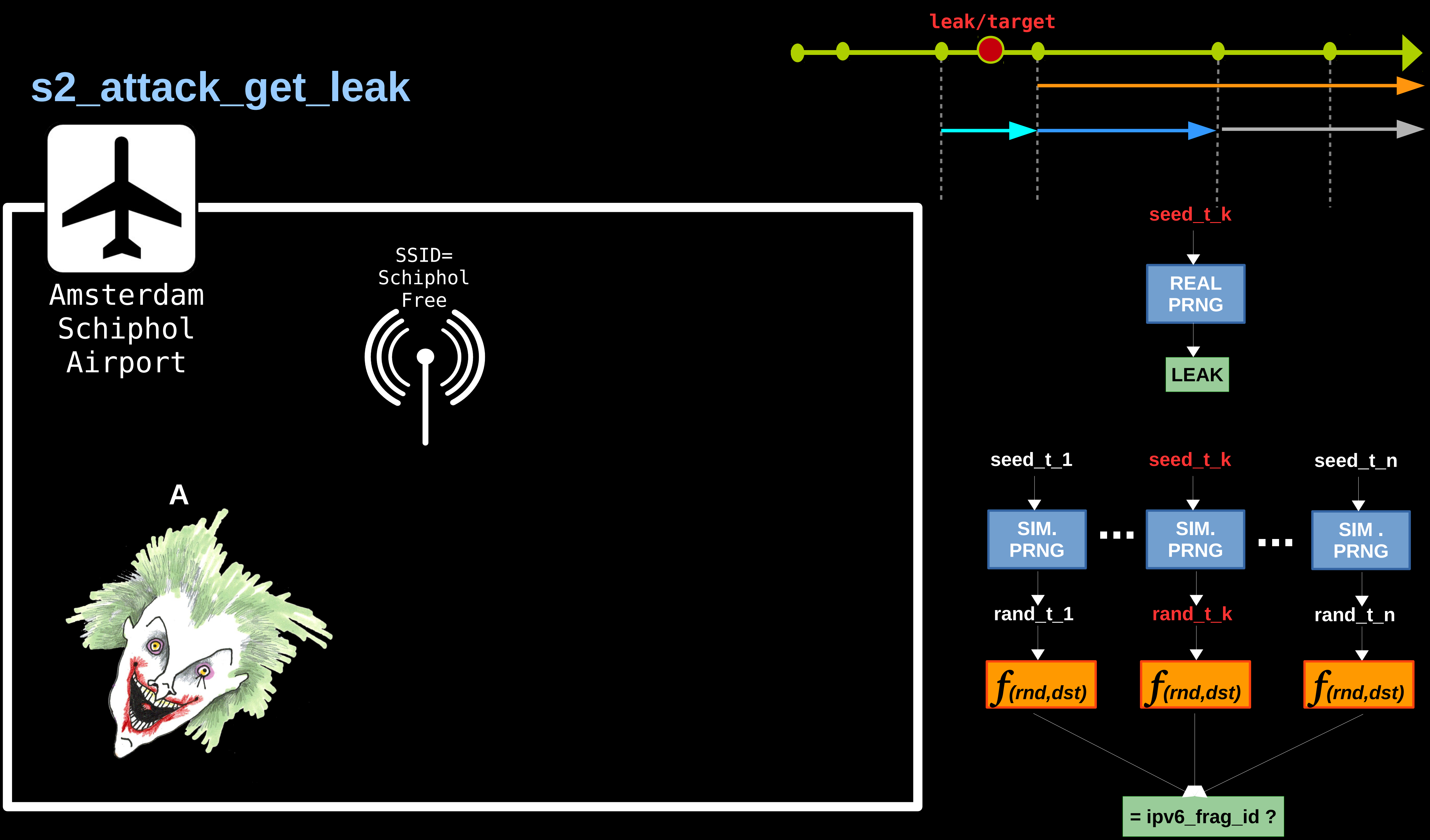
s2_attack_get_leak

Amsterdam
Schiphol
Airport

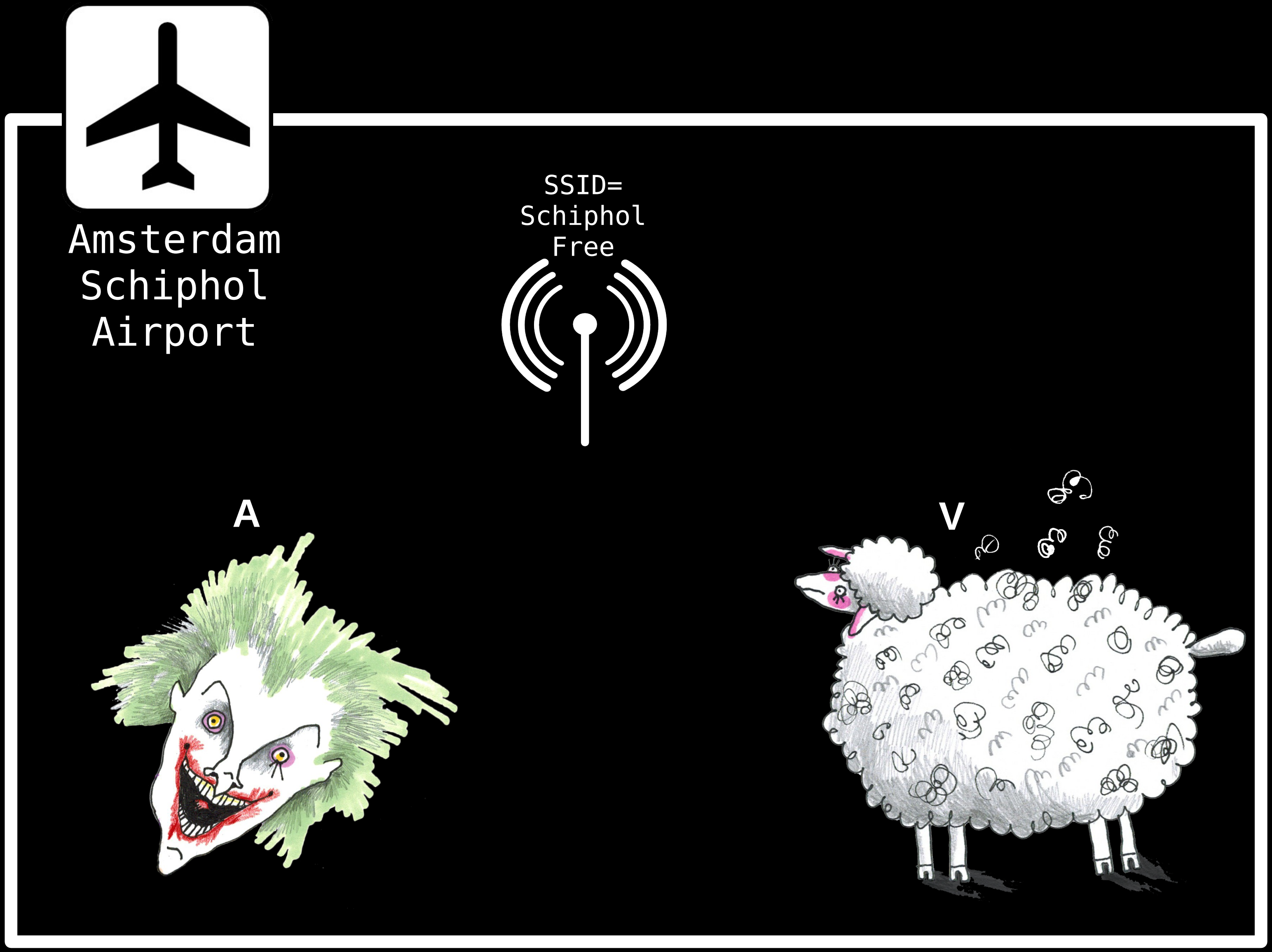
A

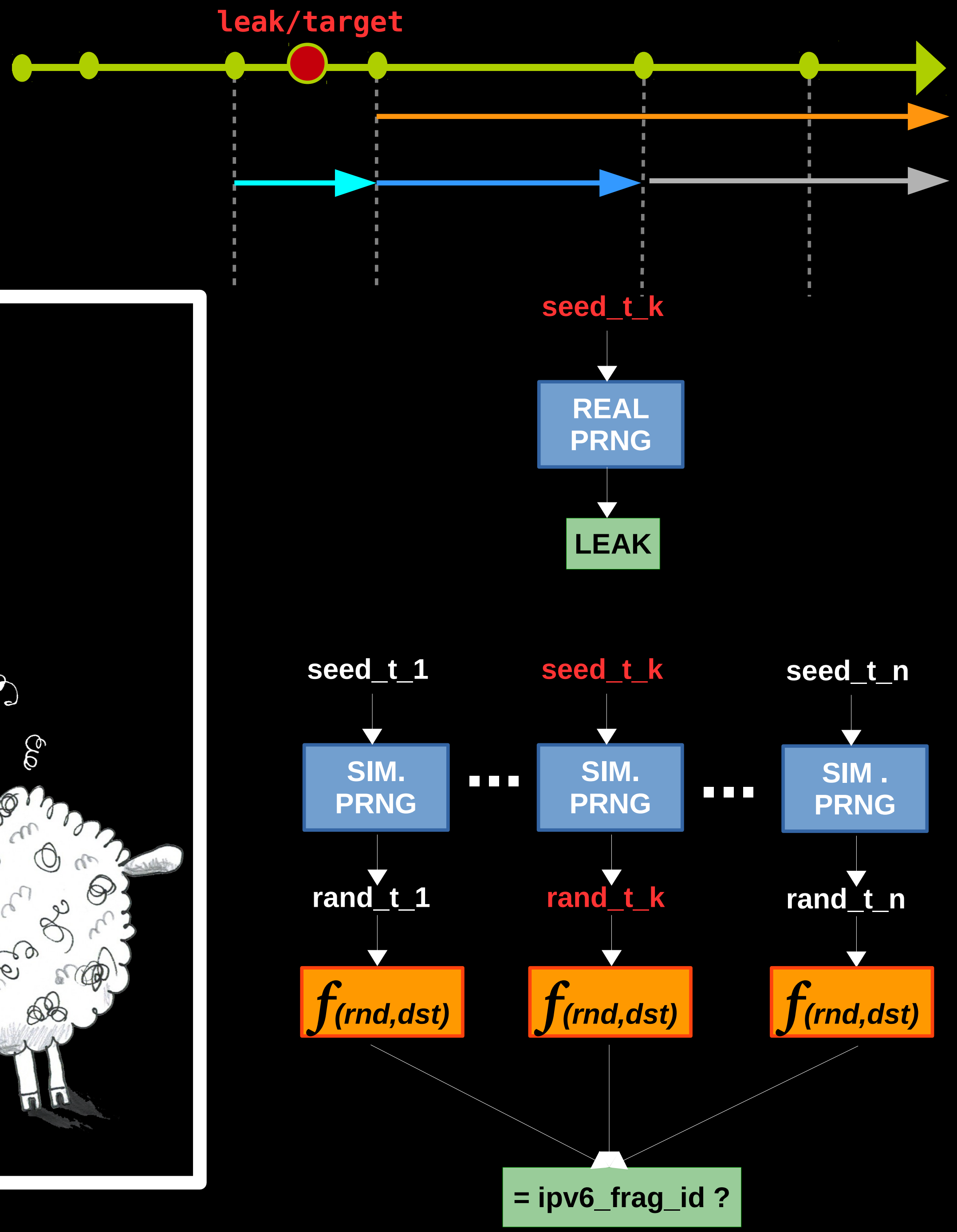


s2_attack_get_leak

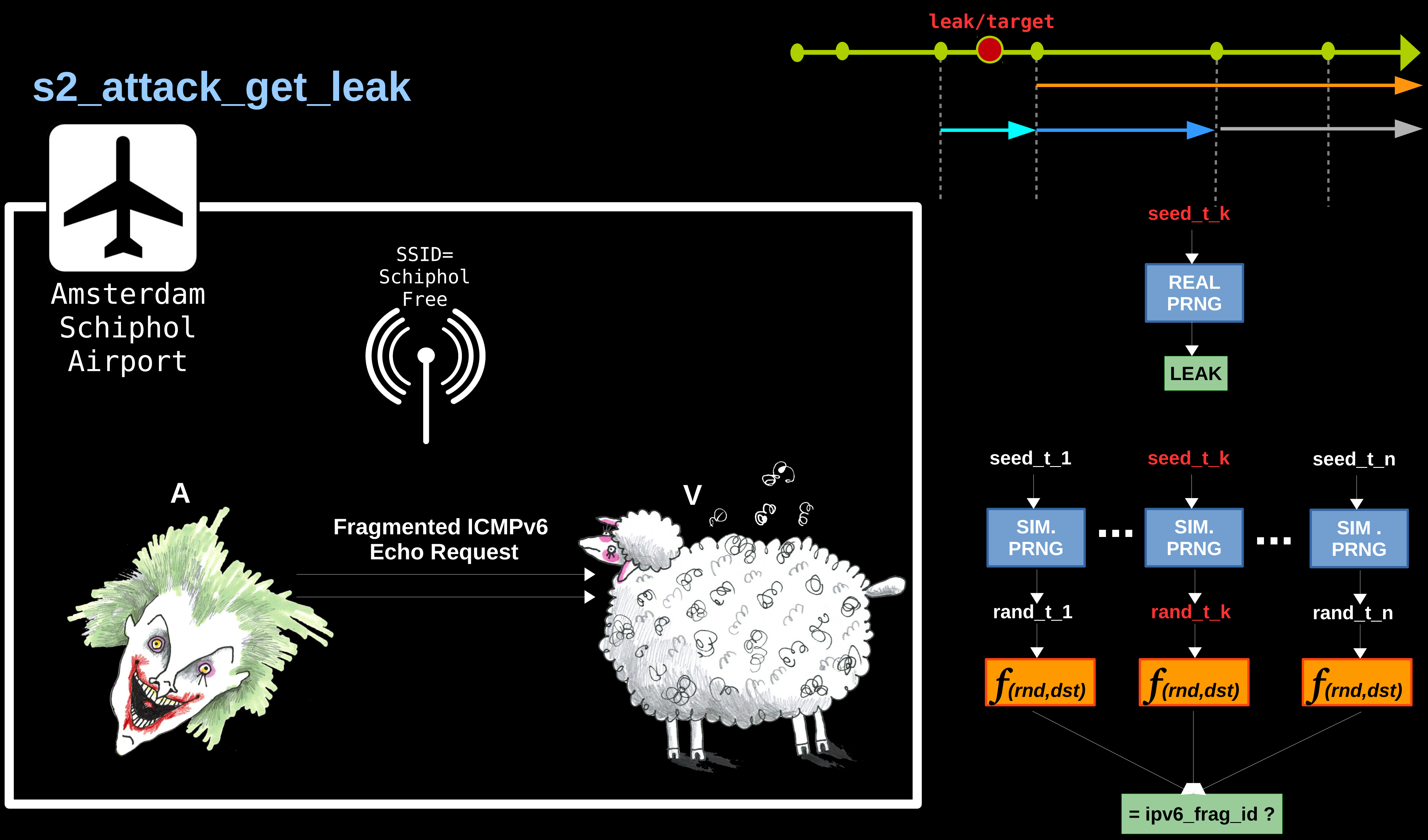


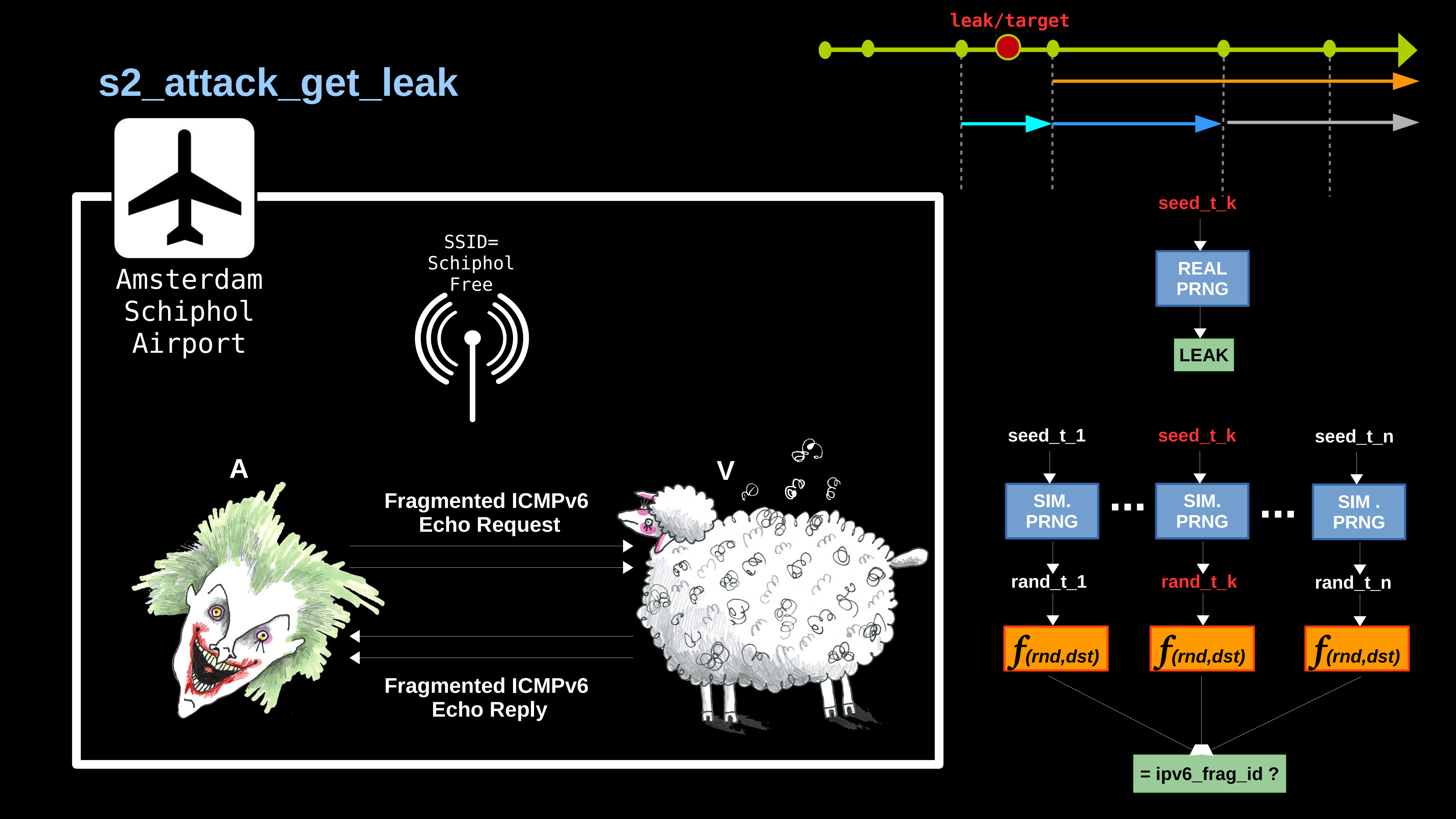
s2_attack_get_leak





s2_attack_get_leak





leak/target
s2_attack_get_leak
Amsterdam Schiphol Airport
SSID= Schiphol Free
A
V
Fragmented ICMPv6 Echo Request
Fragmented ICMPv6 Echo Reply
seed_t_k
REAL PRNG
LEAK
seed_t_1
seed_t_k
seed_t_n
SIM. PRNG
...
SIM. PRNG
...
SIM . PRNG
rand_t_1
rand_t_k
rand_t_n
f(rnd,dst)
f(rnd,dst)
f(rnd,dst)
= ipv6_frag_id ?

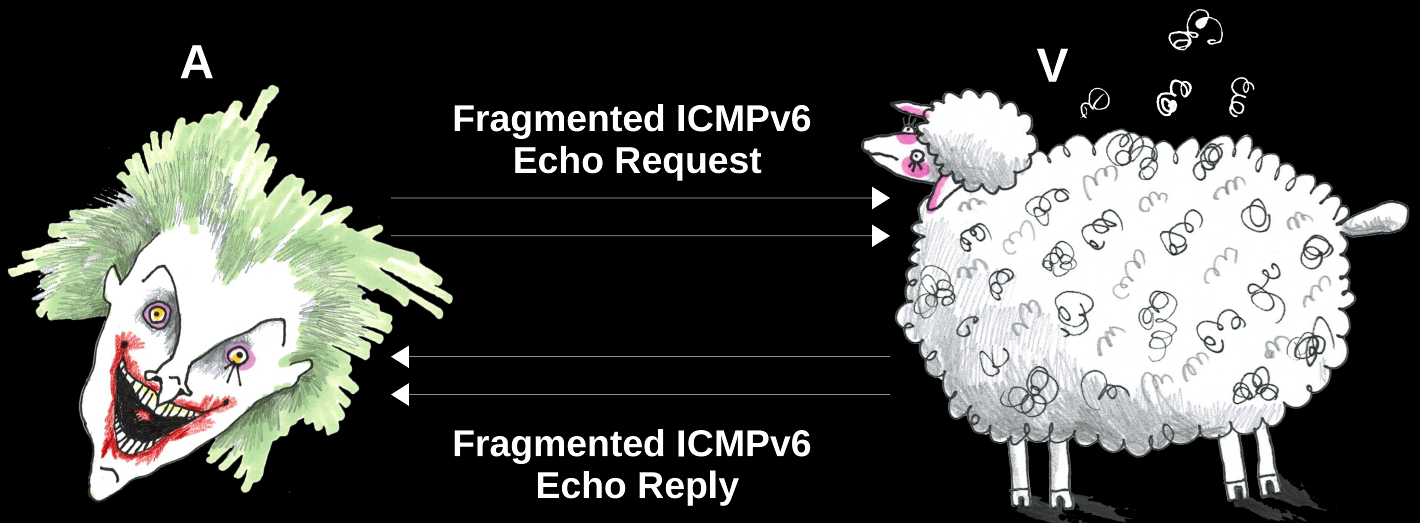
s2_attack_get_leak

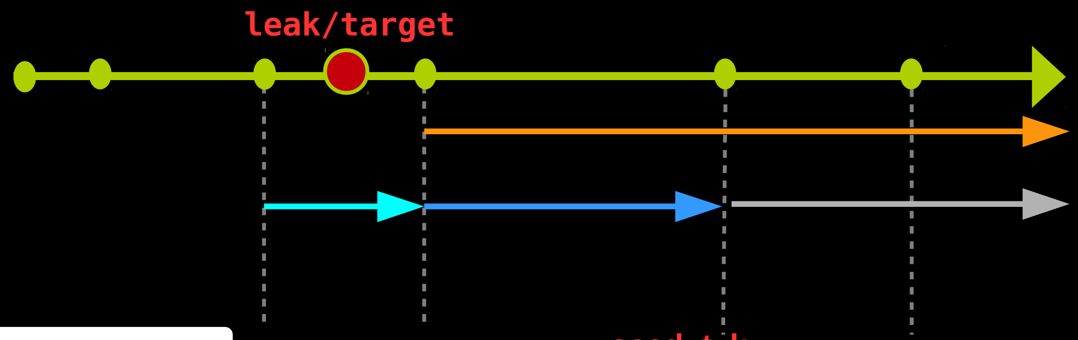
Amsterdam Schiphol Airport

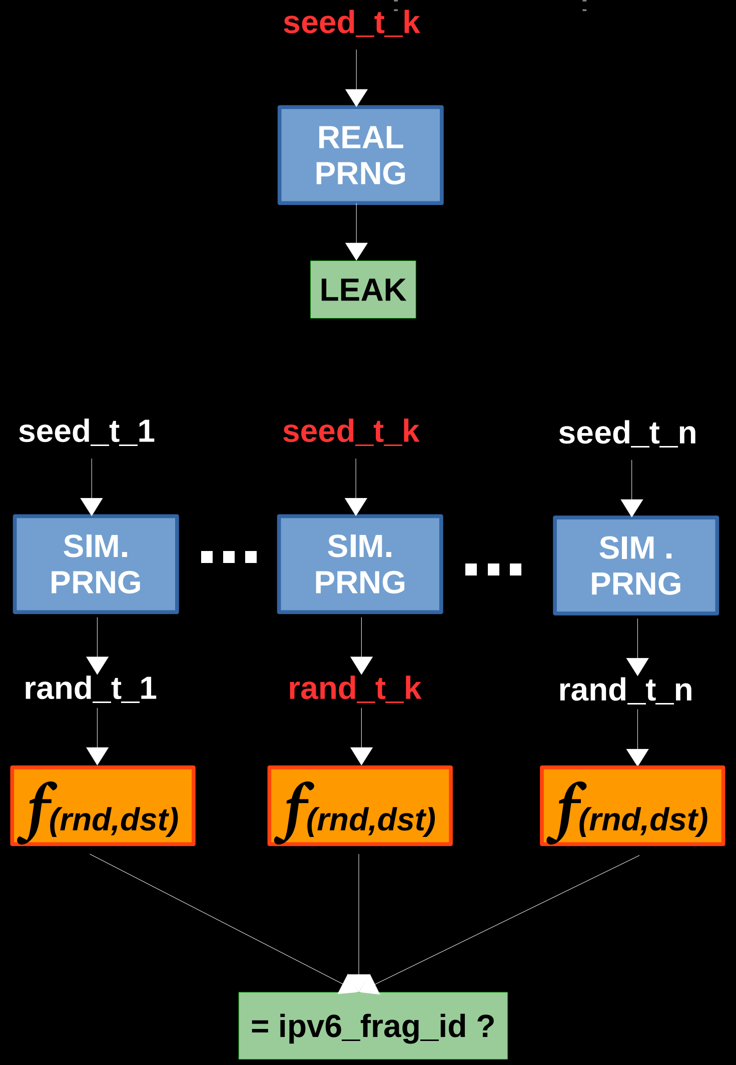
SSID= Schiphol Free

Attacker got the leak:

- V computed ipv6_frag_id with A's ipv6_src_addr
- A knows ipv6_frag_id and ipv6_dst_addr.

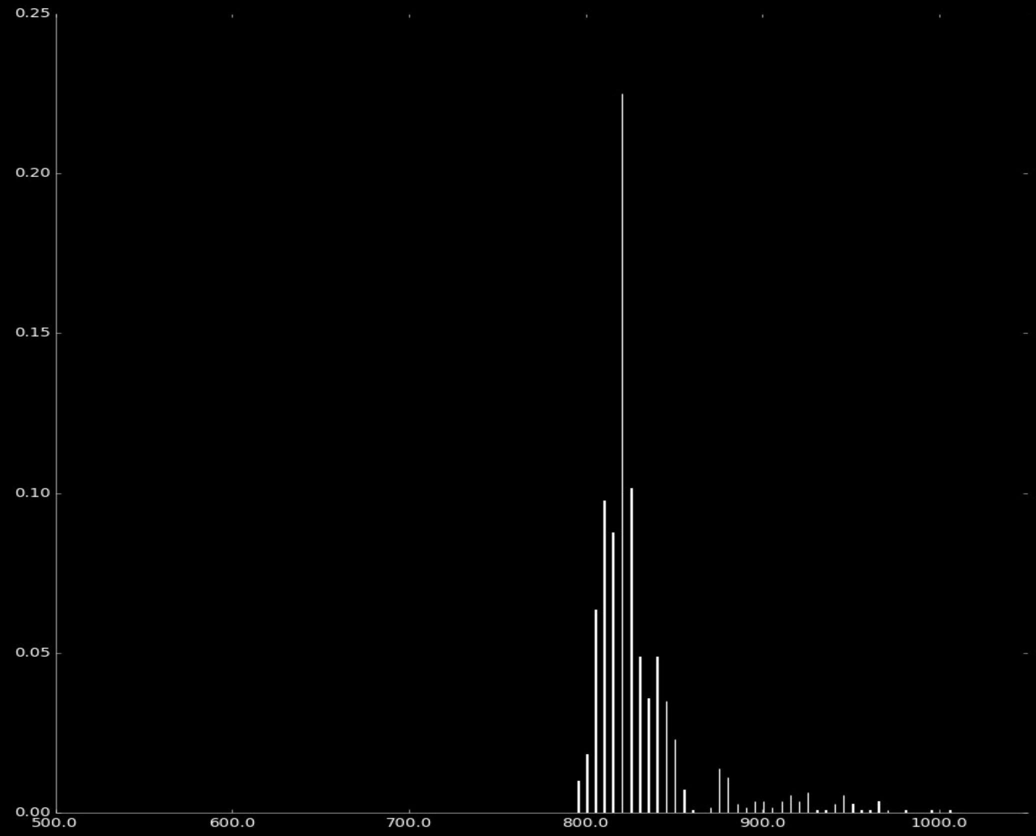




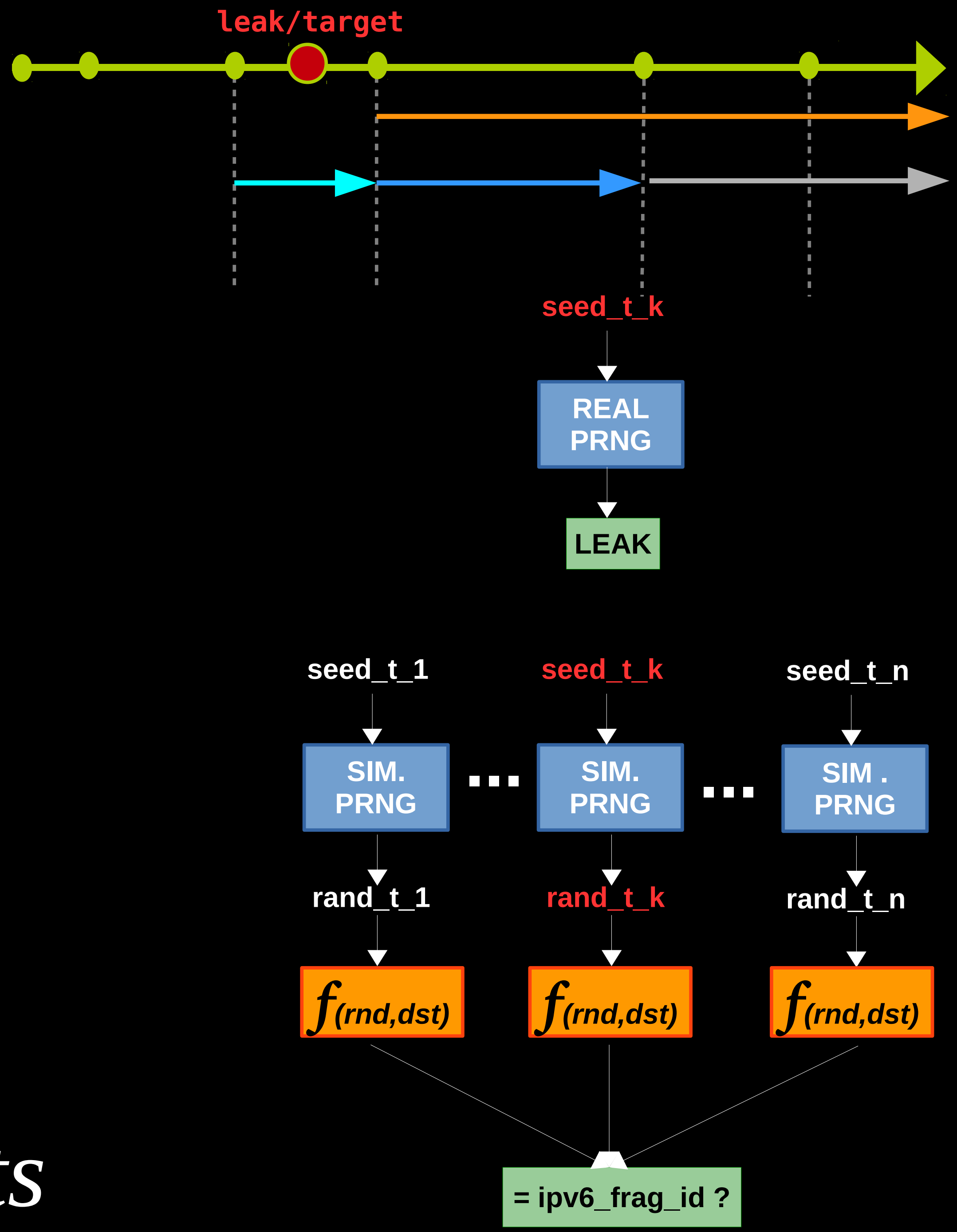


s2_attack_finding_seed

Given the leak we find the seed



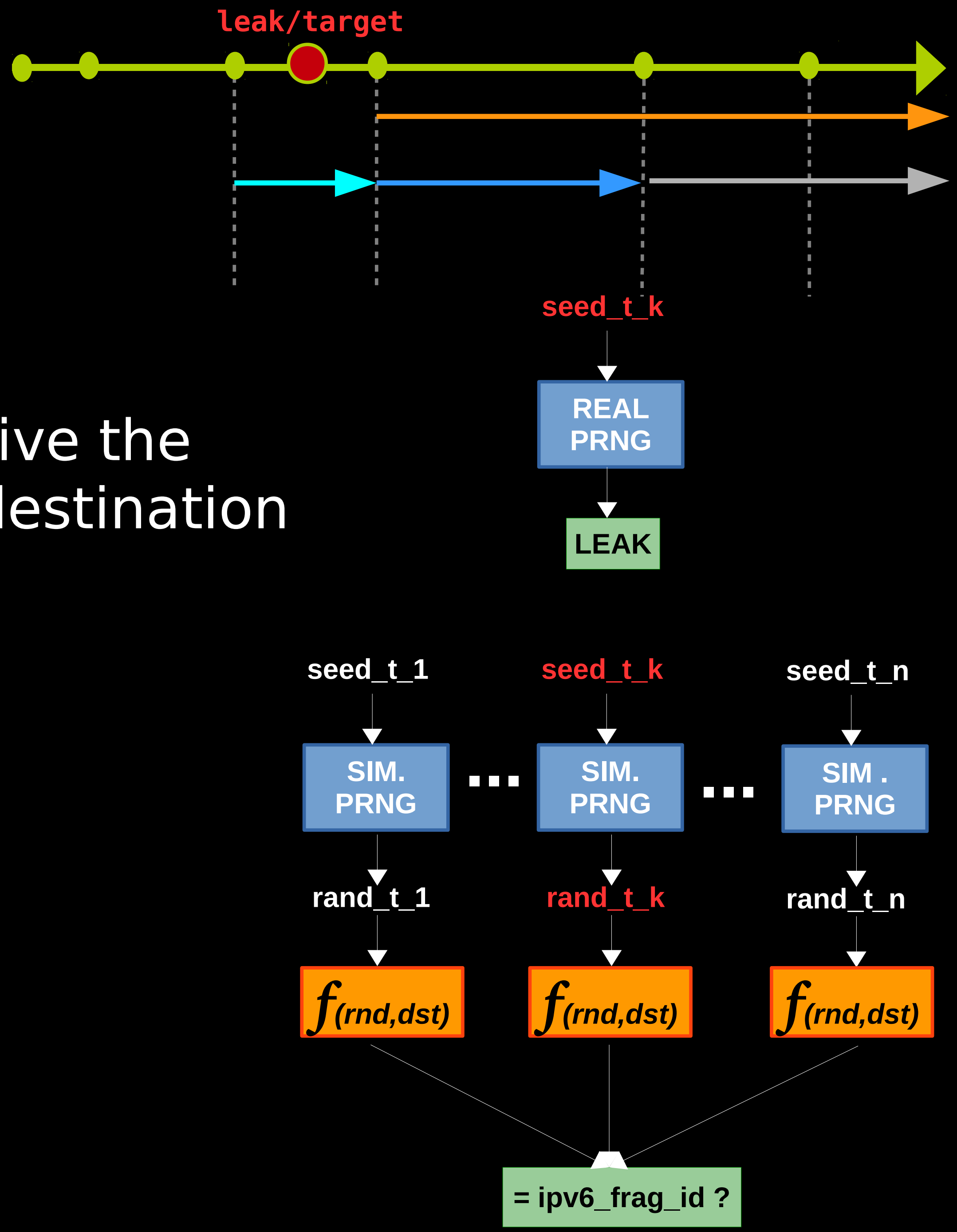
$$H(s_{nb}) = 18.4\, bits$$



s2_attack_targets

Given the seed what can we attack ?

- IPv6 Fragment injection – We can derive the exact fragment id V will use for any destination address.

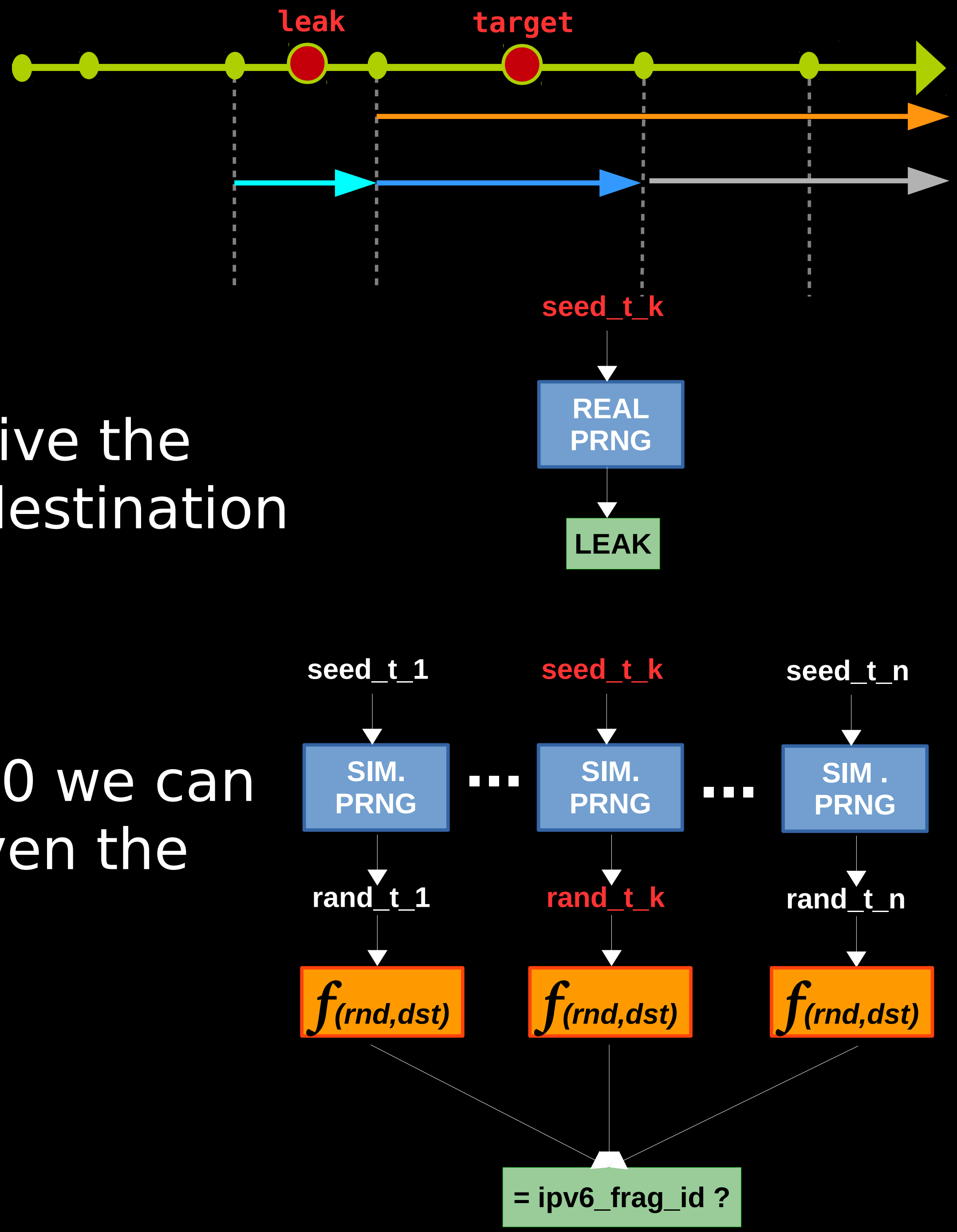


s2_attack_targets

Given the seed what can we attack ?

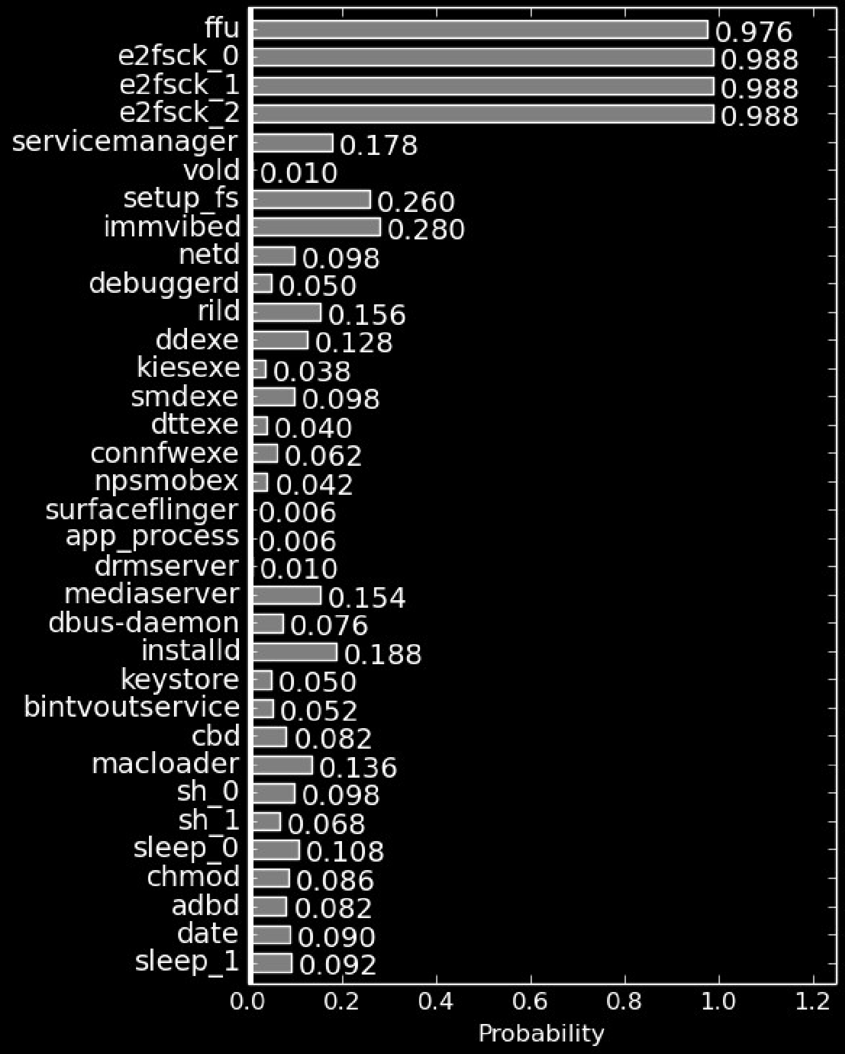
- IPv6 Fragment injection – We can derive the exact fragment id V will use for any destination address.

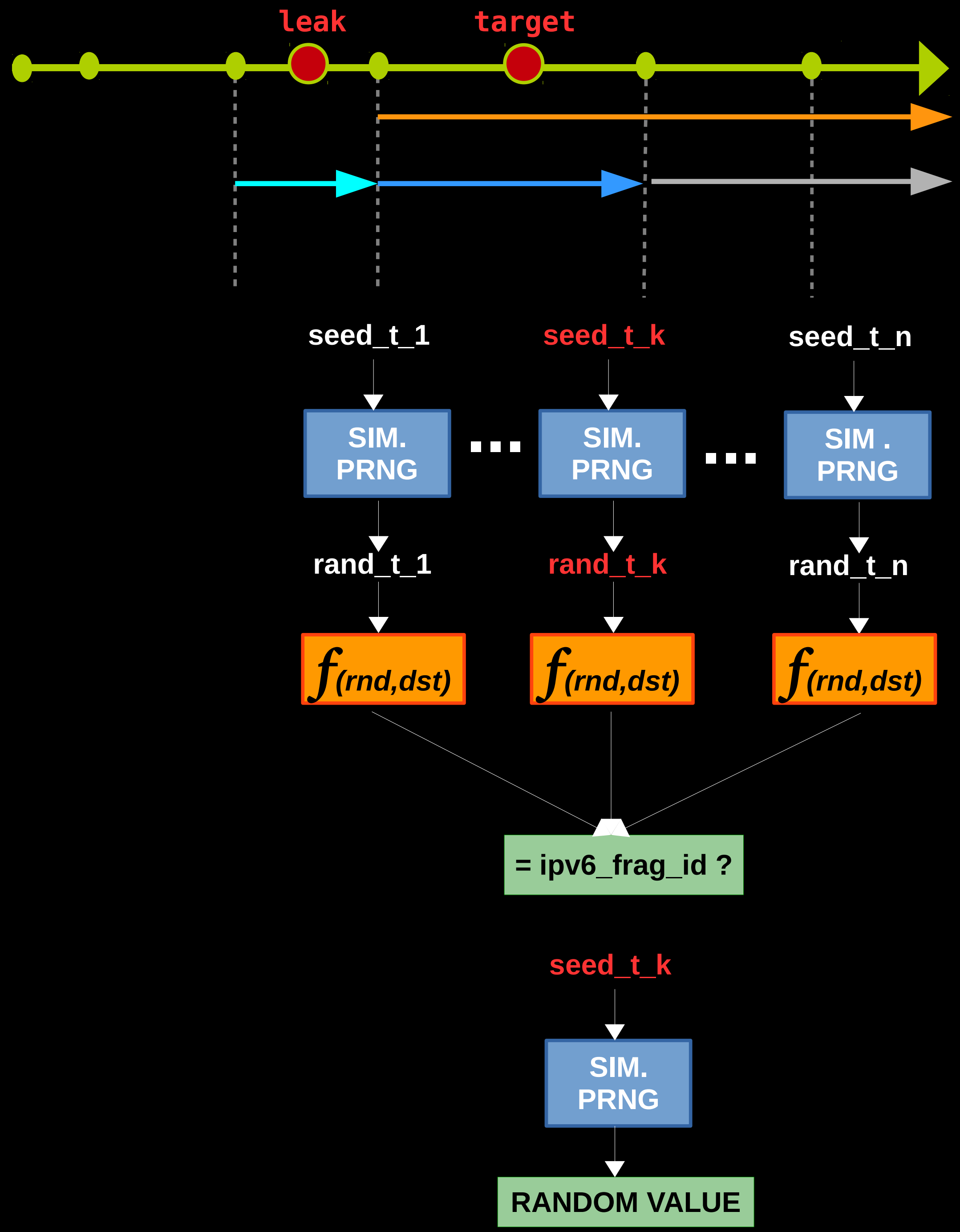
- Canary value of early boot services.
  For instance, with a probability of 1/20 we can compute Keystore's canary value, given the seed.



s2_attack_targets

Probabilities of finding the canary of early boot services



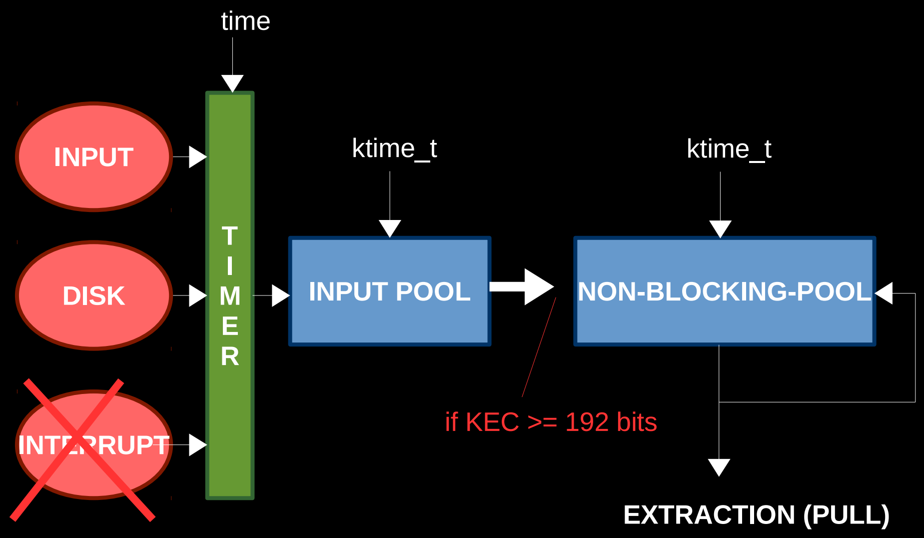


Mitigations

mitigations

Current mitigations

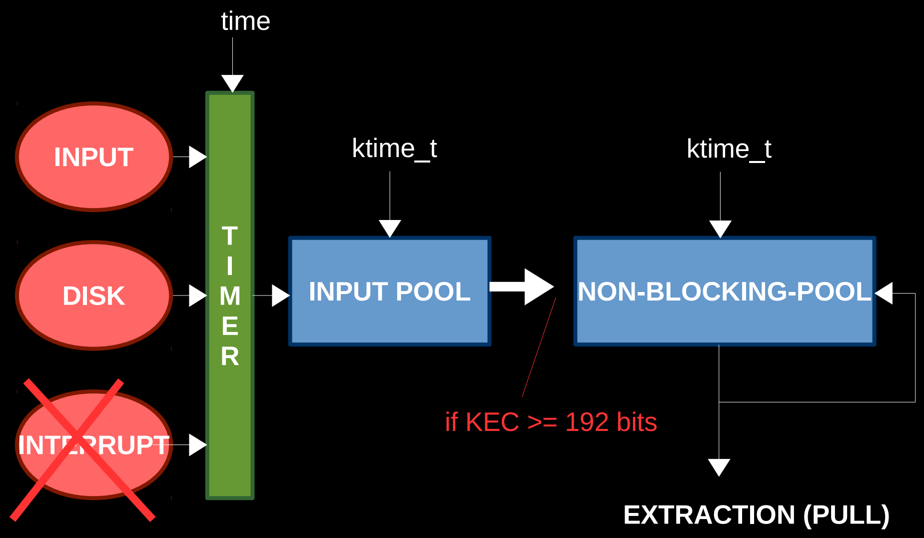
- Save entropy across boots



mitigations

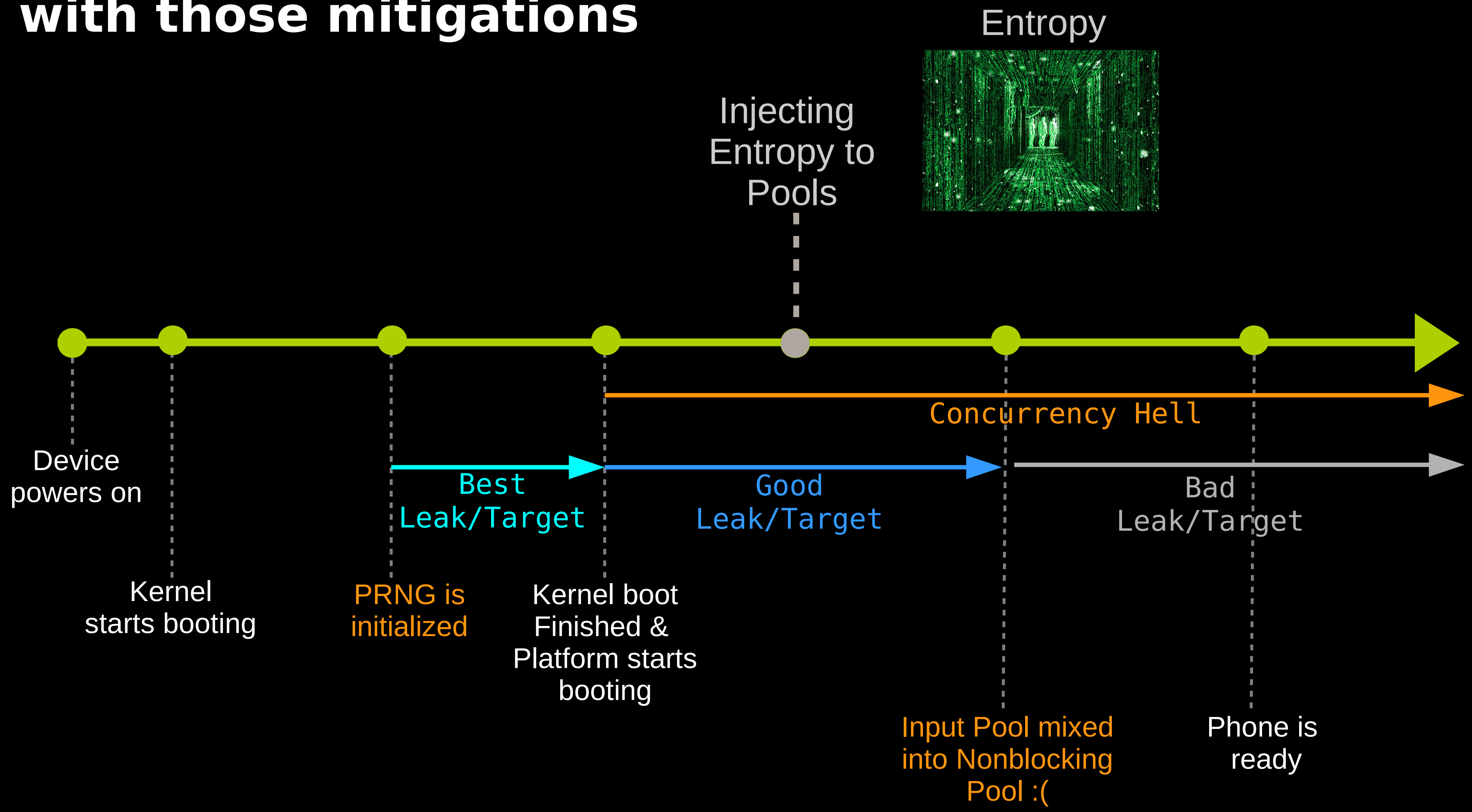
Current mitigations

- Save entropy across boots
- Trusted external entropy injection – web service / HWRNG



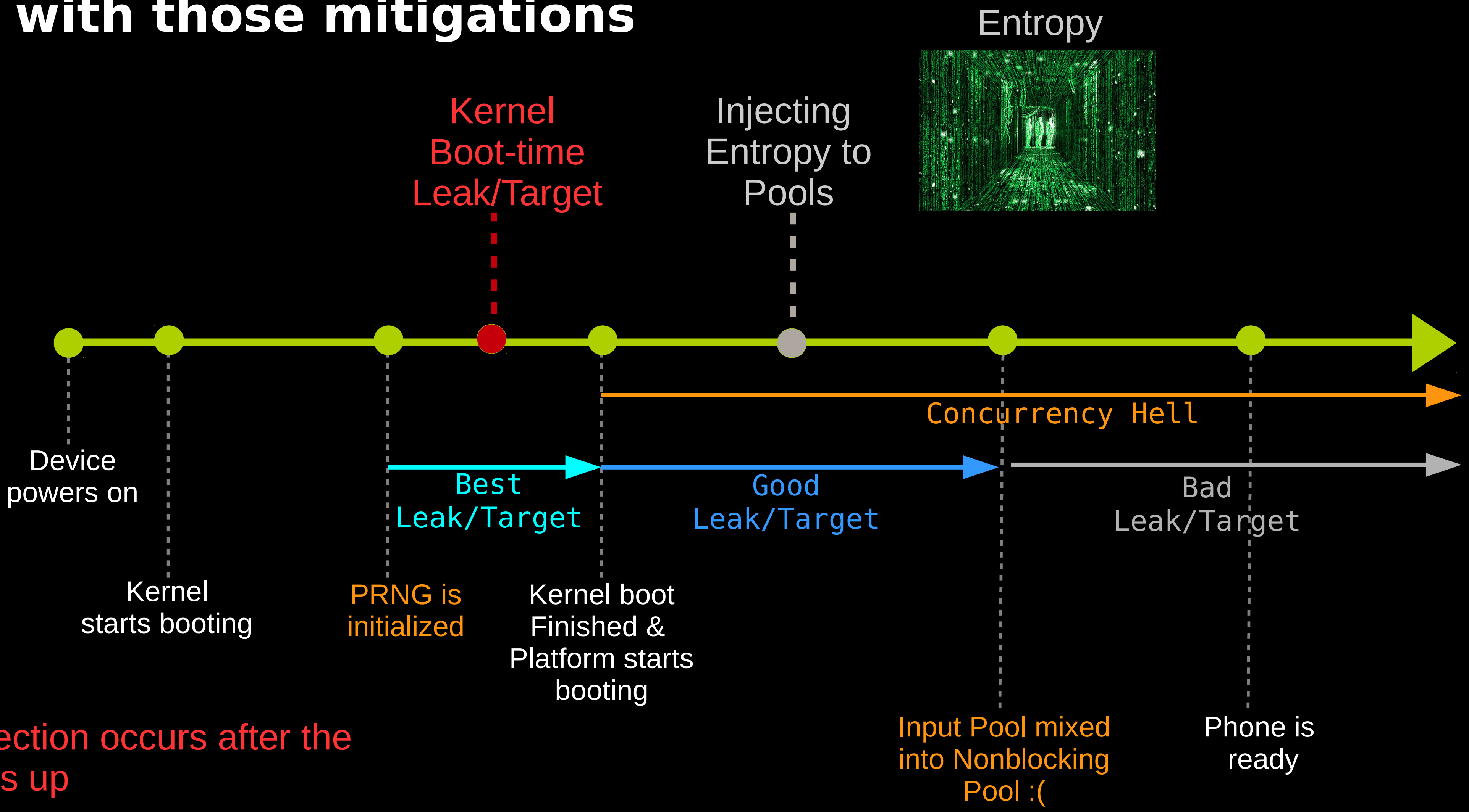
mitigations

Problem with those mitigations



mitigations

Problem with those mitigations

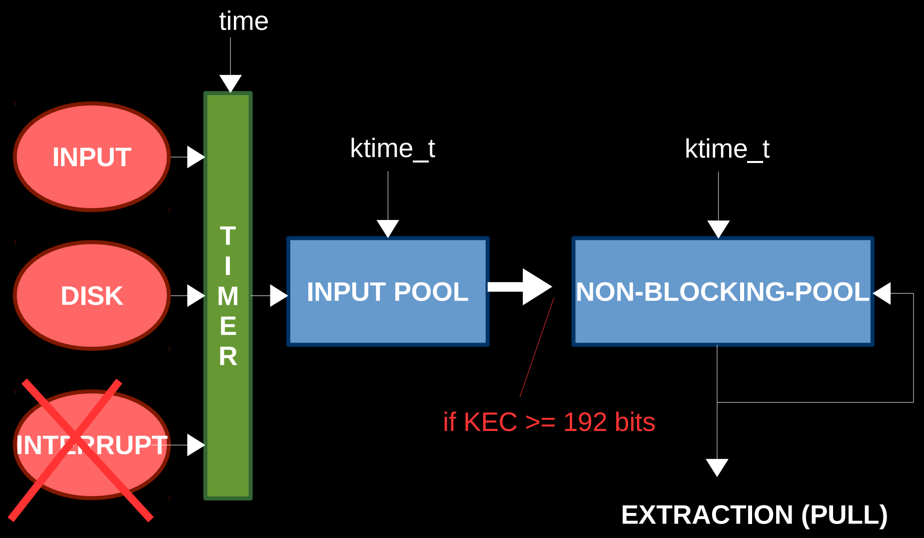


Entropy injection occurs after the kernel boots up

mitigations

Current mitigations

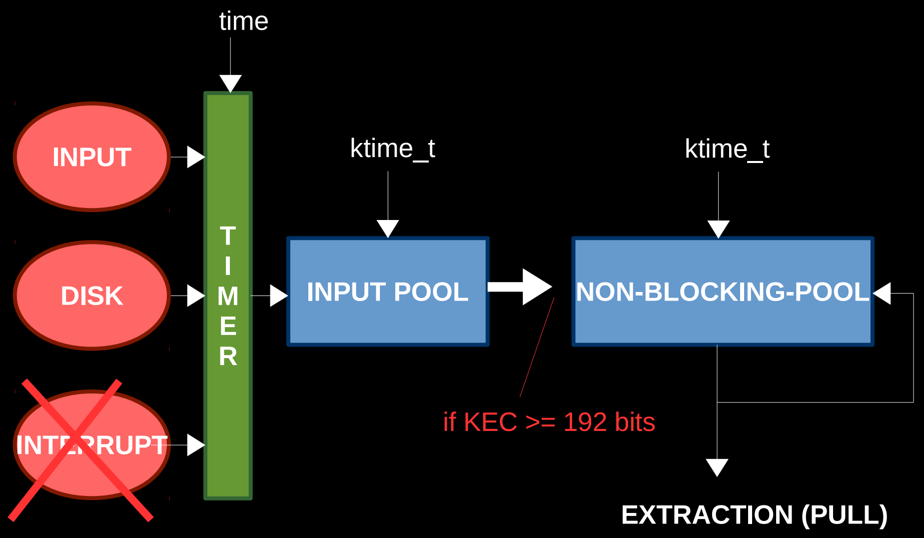
- Initialize the seeds using a hardware RNG
  - RDRAND,RDSEED Intel's ISA
  - Early random, Qualcomm



mitigations

Current mitigations

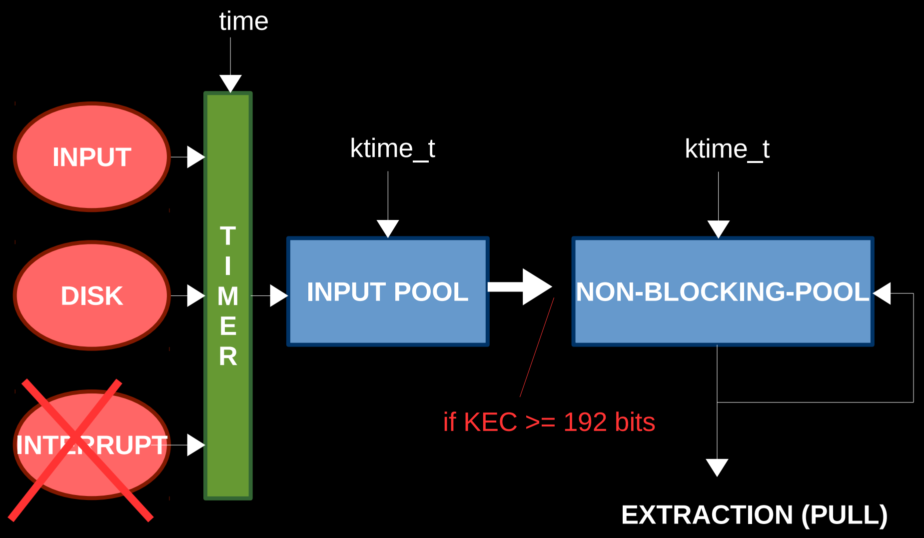
- Initialize the seeds using a hardware RNG
  - RDRAND,RDSEED Intel's ISA
  - Early random, Qualcomm
- Mix device-specific data to nonblocking and blocking pools



mitigations

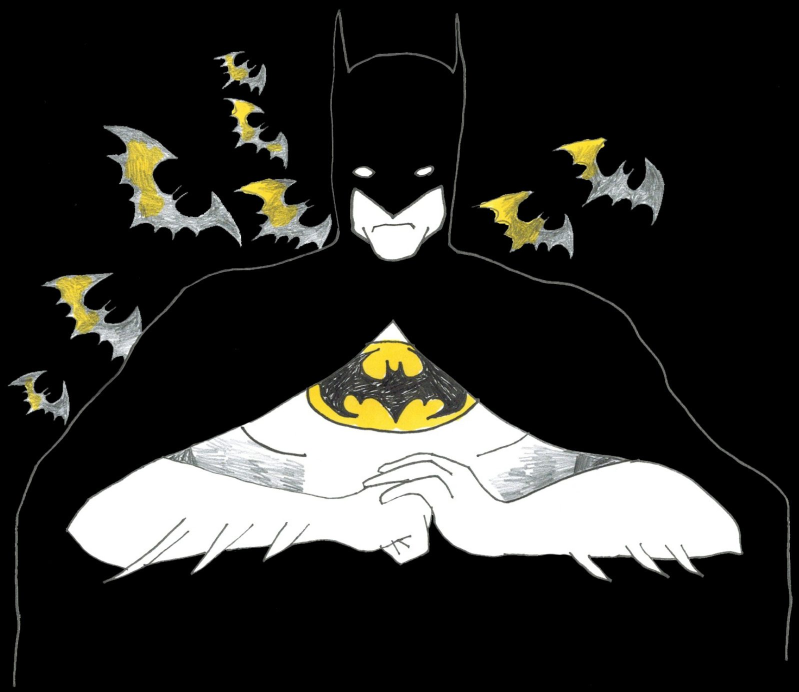
Current mitigations

- Initialize the seeds using a hardware RNG
    - RDRAND,RDSEED Intel's ISA
    - Early random, Qualcomm
- Mix device-specific data to nonblocking and blocking pools
- Changes to newer kernels allow for more boot time entropy



talk_wrap_up

- Linux-based devices with low boot time entropy may allow a practical, low-cost attack on the PRNG
- The attack requires an offline study of a device and an online leak
- Allows the attacker to predict a random number which is generated by the victim's PRNG
- Two manifestations - Local/Remote Atk.
- Mitigations



?

Thank you

Thanks Nadja Kahan for the illustrations !
http://www.nadjakahan.com