Fast and just as random

getrandom() in userspace

1. Linux API history to obtain entropy
2. userland/glibc API to get entropy
3. Attempts to provide a CSRNG in glibc/userland
4. The vDSO getrandom implementation

First on how kernel improved

Linux APIs

linaro Connect

- Entropy is used in a lot of applications
  - BY the kernel itself on multiple places
  - On Cryptography for key generation and other places
  - For algorithm hardening, like Hash Tables initialization or heap hardening

- Having a **good** and **fast** CSRNG (Cryptographic Secure Random Number Generation) allows multiple application to leverage it without reinventing the wheel
  - And also to avoid multiple pitfalls by reimplement it

- Provide a good entropy source is **not easy**

Current Linux APIs

linaro Connect

- getrandom() - Linux 3.17
  - Added to overcome two main issues with random devices: file-descriptor exhaustion and lack of procps access (i.e chroot and/or container).
  - Designed to use /dev/random entropy pool

- /dev/urandom
  - Dot not block
  - ‘Good enough’ entropy at the time of call

- Linux 5.5 change /dev/random to behave like getrandom()
  - It would only block untils it is initialized
  - Added **GRND_INSECURE** to avoid blocking, by provided ‘good enough’ entropy

Recent Linux developments

linaro Connect

- Linux 5.17 added a mechanism for VM forks to reinitialize the CSRNG state
  - The idea is to avoid VM snapshots or duplications to have the same entropy pool state.
- Linux 5.17 as added a per-cpu CSRNG state with **Fast Key Erasure**
  - Improved multicore performance due lockless fast-path entropy pool access
- Linux 5.18 improved /dev/urandom initialization with ‘opportunistically’ heuristics
  - If architecture has fast cycle counters (rtdsc, cntfrq_el0, etc.) it will try try to make /dev/urandom as good as /dev/random
- Linux 6.11 added the **vDSO** getrandom implementation
  - Initially for **x86**, but later for **aarch64**, **loongarch**, **powerpc**, and **s390** on **6.12**

glibc/userspace API to obtain entropy

- AT_RANDOM
  - Used internally to initialize stack guard (for -fstack-protector) and to pointer mangling
  - Very limited entropy (32 or 64 bit)

- getrandom() added on **glibc 2.25**
  - Cancellable syscall wrapper

- getentropy() added on **glibc 2.25**
  - From BSD system
  - Implemented using getrandom(), but non-cancellable
  - Limited entropy (maximum of 256 bytes per call)

- arc4random() on **glibc 2.36**
  - Also from BSD
  - Should not fail, and abort process if entropy can not be obtained
  - Current implemented on top of **getentropy()** with fallback to **/dev/urandom**

The arc4random stirred the discussion on how properly provide a CSRNG in userland

Attempts to provide a CSRNG in glibc/userland

Some history

linaro Connect

- Florian Weimer proposed a AES-128 based arc4random for **glibc 2.28**
  - Per-thread CRNG state
  - Aimed to use crypto instructions like AES-NI and RDRAND
  - Complex fork detection to support multiple kernel version (with and without **MADV_WIPEONFORK**).
  - **Lockless** and **async-signal-safe**.
  - Patch stalled without review

- I proposed a **ChaCha20** based one for **glibc 2.36**
  - ChaCha20 is performance and lightweight **stream cipher.**
  - Simpler and limited fork-detection: either **MADV_WIPEONFORK** or an atfork handler
  - Based on OpenBSD implementation
  - Arch-specific block Chacha20 optimizations (aarch64, x86, powerpc, s390x)
  - After some iteration it was committed

More history

- Just after inclusion Jason Donerfeld raised multiple concerns about the implementation
  - How to properly reseed the crng state where only kernel has all the required information (on VM fork, on system resume, from hibernation)?
  - The userland CRNG state might leak dependending on how VMs are configured.

- There were more discussion if arc4random should be designed as a **CSRNG** or not
  - Although documentation is not clear, user assume **it is** from BSD design

- The Chacha20 implementation was reverted in favor of one based on getrandom()

- And as so it was released, users complained that it was **too slow** (BZ 29437)

The vDSO getrandom() support

Requisites for a userland CRNG

- User expects a **CSRNG**, otherwise they roll their own (and most likely not handling all the corner cases)
- The userspace implementation should provide **the same security guarantee** as the kernel
- It requires low-latency and good throughput
  - Otherwise the syscall is good enough

The vDSO symbol provides all the guarantees

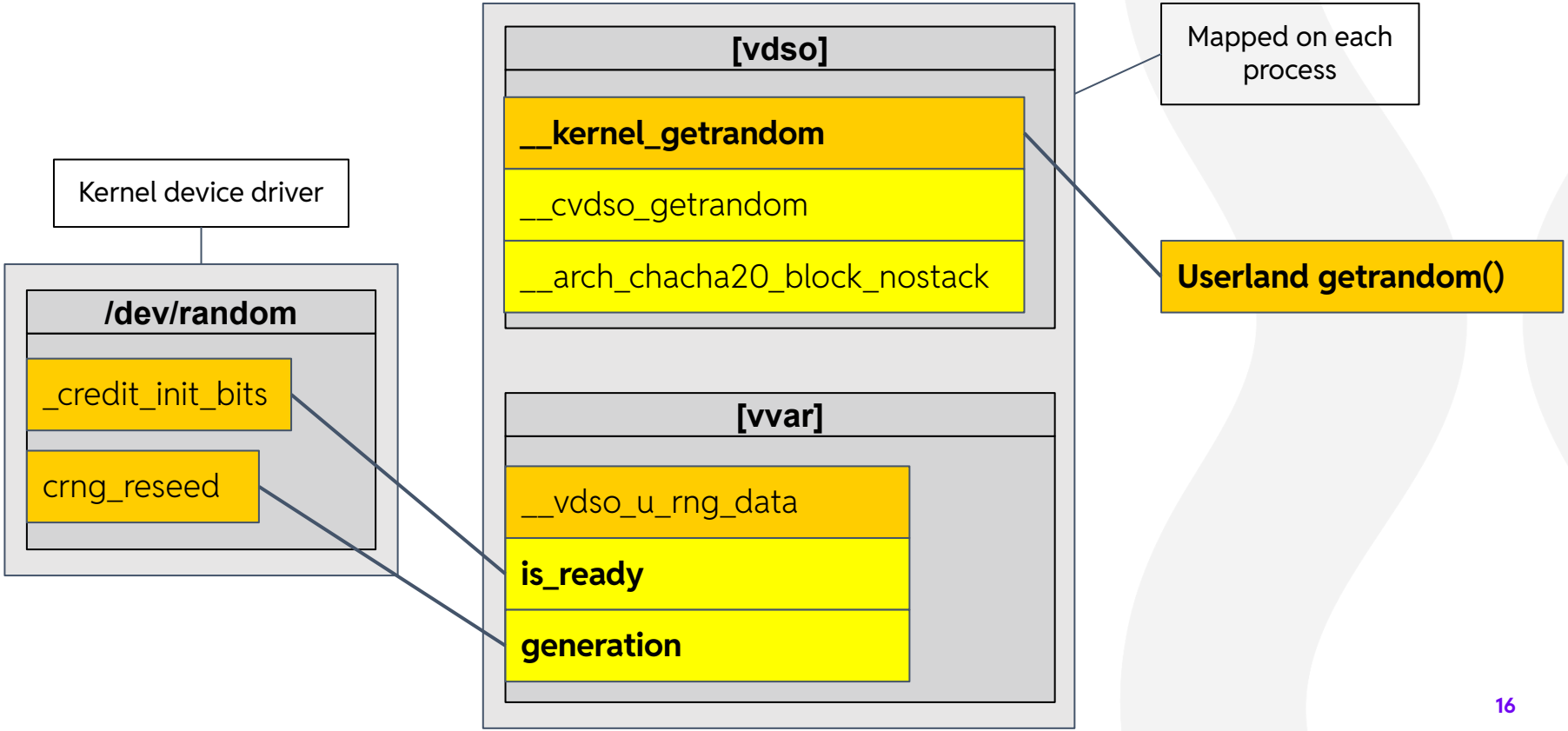
linaro Connect

- The kernel provides the algorithm and can signal the userland when to reseed
  - The userland code should not leak information
  - **So it requires a arch-specific implementation**

- The CSRNG state kept in userland requires some extra semantics
  - It should never be backed by swap
  - It can zero out anytime under memory pressure (the state can be recreated any time)
  - It should not be counted as mlocked
  - **A new mmap flag (MAP_DROPPABLE)**

- To allow a per-thread lockless implementation the userland requires some extra management
  - **A new glibc getrandom() implementation if vDSO is provided**

The kernel side



The kernel side

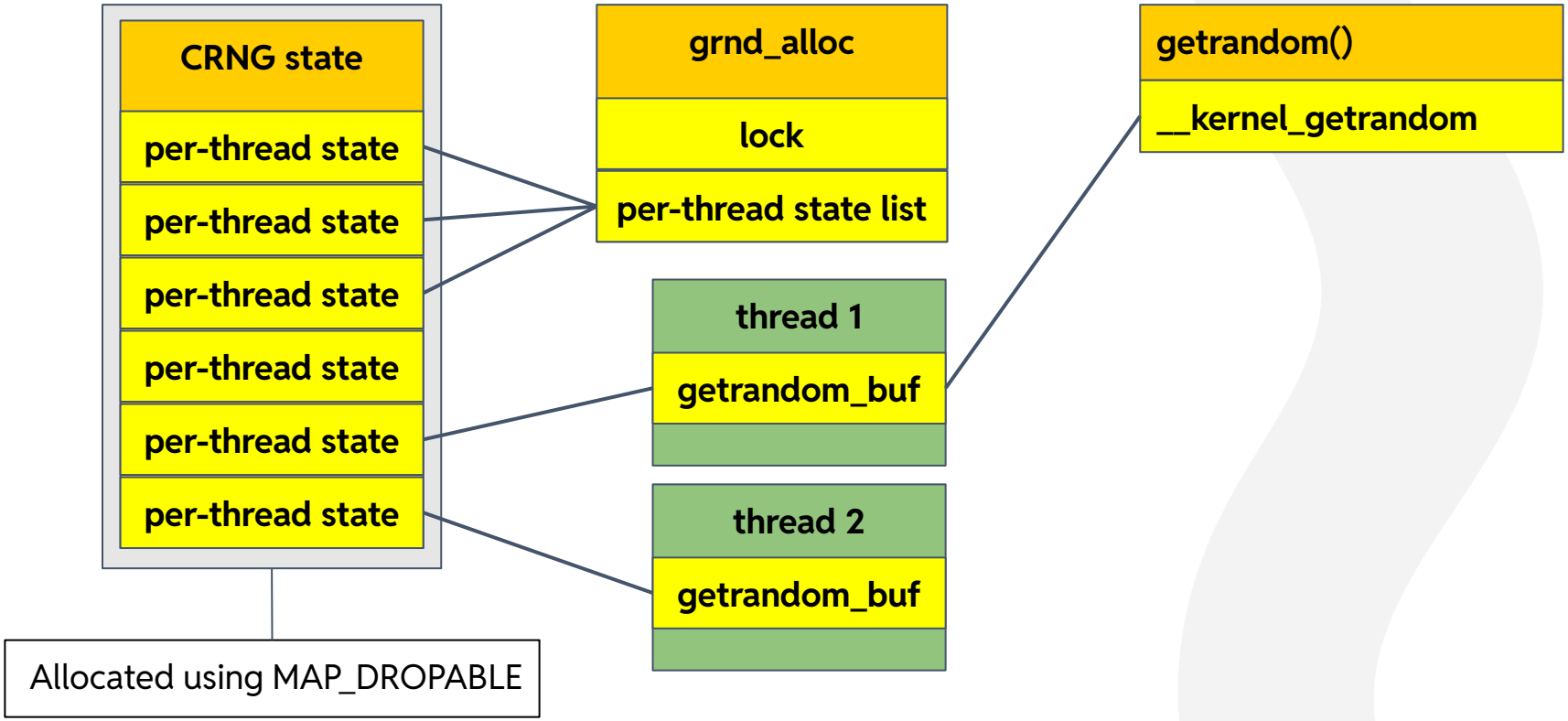
linaro Connect

- The crypto driver is responsible to advertise when the crypto subsystem is ready (**is_ready)** and when to reseed (**generation**) through the **vvar** (datapage)
  - It is mapped on each process and shared among all processes

- The vDSO generic implementation (**__cvdso_getrandom**) check if reseed is required and call the **getrandom** syscall to if so

- If everything is ok, it call the arch-specific **Chacha20** implementation (**__arch_chacha20_blocks_nostack**) which generates entropy on **userland state**
  - It is a simplified Chacha20 implementation that does not leak any state on stack and work only on multiple of cypher blocks.

Userland side



glibc implementation

linaro Connect

- Kernel API
  - The kernel vDSO provides the required **mmap** flags and the **opaque state** size used.
  - Some extra care to align the **opaque state** to L1 data cache line size to avoid **false-sharing.**
  - The **opaque state** is used a per-thread **CRNG** state

- At glibc **initialization**
  - Query the vDSO for the mmap flags and opaque state params

- On **getrandom()**
  - Try to **reserve** a per-thread opaque state from the per thread list and update the TLS pointer.
  - Reentrancy handling, fallback to syscall.
  - Call the **vDSO** with similar arguments as the syscall plus the opaque state
  - **Release** the per-thread opaque state

glibc challengers

linaro Connect

- The data structures
  - The **per thread list** is simple block allocator organized as FIFO.
  - Neither the **CRNG state** nor the **per thread list** are deallocated or shrink during execution
  - The **per thread list** requires to be async-signal-safe, so it is based on mmap

- **fork()** handling: the per thread list needs to be in a consistent state on any case, even though there is memory leak during fork
    - Extra care with memory fences on its internal update

Performance numbers

Latency

- The kernel has a benchmark to evaluate the vDSO improvements
  - tools/testing/selftests/vDSO/vdso_test_getrandom
  - It focus on small buffers (32-bits), which seems the most usual case

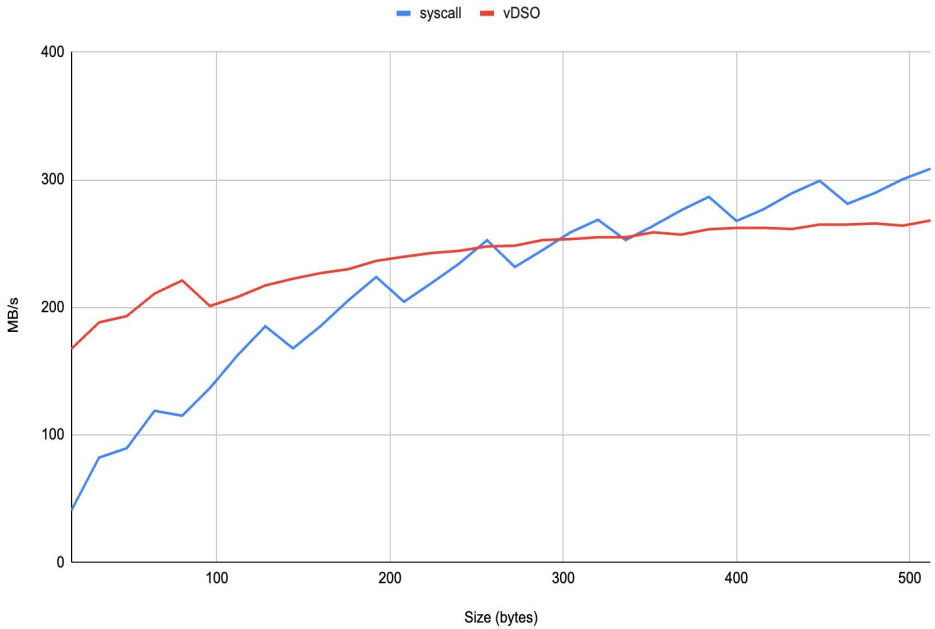
```
$ ./vdso_test_getrandom bench-single
   vdso: 25000000 times in 0.770766986 seconds
   libc: 25000000 times in 0.821580789 seconds
syscall: 25000000 times in 9.093588456 seconds
```

```
$ ./vdso_test_getrandom bench-single
   vdso: 25000000 x 256 times in 1.633428630 seconds
   libc: 25000000 x 256 times in 1.850602995 seconds
   syscall: 25000000 x 256 times in 20.310248913 seconds
```

* Running on a Neoverse1, Linux 6.15, gcc 14.2.1, and glibc master

Throughput

- Larger buffers amortize the syscall overhead
- And leverages the Chacha20 block Linux crypto optimization
  - chacha_4block_xor_neon
- This can be a room for improvement
  - Check if the block Chacha20 NEON optimizations can be used on vDSO



* Running on a Neoverse1, Linux 6.15, gcc 14.2.1, and glibc master

How to try it

- Linux 6.11 for x86 or 6.12 for aarch64 and other architectures
- glibc 2.41
- Hardware with NEON support.

Thank You!