Integrating Remote Quantum Random Number Generator as a Shared Resource into GNU/Linux via D-Bus

KRIŠJĀNIS PETRUČEŅA, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
SERGEJS KOZLOVIČS, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
ELĪNA KALNIŅA, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
EDGARS RENCIS, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
JURIS VĪKSNA, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
EDGARS CELMS, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia
LELDE LĀCE, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia

The random number generation capabilities of the GNU/Linux operating system are subject to certain limitations. As of Linux version 5.6, /dev/random operates in a non-blocking manner and, as such, no longer satisfies the criteria for a True Random Number Generator (TRNG). While dedicated quantum random number generator (QRNG) hardware is the preferred source of unpredictable entropy, it is often expensive and difficult to deploy in virtualized/cloud environments and Internet of Things (IoT) devices. Furthermore, hardware RNG integration typically requires cryptographic applications to adhere to vendor-specific APIs.

This article proposes a user-space integration approach for a *shared*, potentially remote QRNG device. We develop a QRNG service on top of D-Bus, a ubiquitous inter-process communication framework. It serves as an interface for applications to retrieve *true* random numbers. Communication with the remote

This work was supported by the European Union Recovery and Resilience Facility Fund under Award No. 2.3.1.1.i.0/1/22/I/CFLA/001.
Authors' Contact Information: Krišjānis Petručeņa, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: krisjanispetrucena@gmail.com; Sergejs Kozlovičs, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: sergejs.kozlovics@lumii.lv; Elīna Kalniņa, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: elina.kalnina@lumii.lv; Edgars Rencis, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: edgars.rencis@lumii.lv; Juris Vīksna, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: juris.viksna@lumii.lv; Edgars Celms, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: edgars.celms@lumii.lv; Lelde Lāce, Laboratory of Systems Modeling and Software Technologies, University of Latvia Institute of Mathematics and Computer Science, Riga, Latvia; e-mail: lelde.lace@lumii.lv.

ACM 2471-2566/2026/04-ART23
https://doi.org/10.1145/3799895

QRNG device occurs over mutually authenticated TLS 1.3 channels, protected by post-quantum cryptography (PQC) algorithms. We show, as a proof-of-concept, how the proposed D-Bus service can be integrated into the OpenSSL 3 cryptographic library, demonstrating the use of TRNG in a wide range of Linux applications.

Our approach is resistant to entropy starvation attacks, supports sharing a QRNG across host and virtualized environments, requires no kernel-level or system-wide modifications, supports mixing multiple sources of entropy, and configuration of post-processing. It provides applications with a TRNG interface suitable for information-theoretically secure (ITS) use cases.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Hardware → Quantum technologies;**

Additional Key Words and Phrases: QRNG, linux, d-bus, post-quantum cryptography

ACM Reference Format:
Krišjānis Petručeņa, Sergejs Kozlovičs, Elīna Kalniņa, Edgars Rencis, Juris Vīksna, Edgars Celms, and Lelde Lāce. 2026. Integrating Remote Quantum Random Number Generator as a Shared Resource into GNU/Linux via D-Bus. *ACM Trans. Priv. Sec.* 29, 2, Article 23 (April 2026), 18 pages. https://doi.org/10.1145/3799895

1 Introduction

Hardware-based **true random number generators** (**TRNGs**) employ well-controlled, non-deterministic, and, ideally, non-manipulable physical processes to produce *fresh* (=non-predictable) entropy. Particularly valuable are Quantum RNGs, which exploit intrinsically random quantum effects and have a well-defined mathematical model [23, 38]. According to the Copenhagen interpretation of quantum mechanics, we can axiomatize **quantum-based RNGs** (**QRNG**) as true sources of non-predictable entropy.

Since the majority of the devices are not supplied with a true hardware-based RNG, the RNG functionality is usually implemented by the operating system by aggregating hardware metrics. Linux collects entropy from various components (e.g., keyboard strokes, mouse movements, system clocks, and CPU jitter). The collected entropy is then used to periodically re-seed the built-in **cryptographically secure pseudorandom number generator** (**CRNG**). Unfortunately, virtualized environments and IoT devices do not have as many resources and physical hardware components in order to collect randomness. Thus, in limited environments, entropy can be insufficient, overestimated, or even predictable, e.g., when running a **virtual machine** (**VM**) from a snapshot [10, 13, 35].

A dedicated hardware-based (preferably, quantum-based) entropy source could address the insufficient entropy problem. However, deploying a **hardware RNG** (**HRNG**) device on every computer incurs additional expenses for both hardware and software components. Even more challenging is to integrate HRNGs into low-energy and physically tiny IoT devices, or into VMs without direct access to the host hardware.

A cost-effective alternative, which does not require extra hardware, could be a *remote* true RNG. In 2022, we proposed a remote QRNG web service backed up with a real IDQ Quantis PCIe device [20]. Our service allows one or more hardware QRNG units to be securely shared among multiple users via the network, using TLSv1.3 with post-quantum algorithms. However, this approach has its own limitations: we need to trust the remote QRNG service provider, and there could be network-related issues (limited bandwidth or unstable connection), which may lead to client starvation. Besides, integrating a remote QRNG into the local OS is also a significant challenge.

In this article, we propose a universal user-space TRNG abstraction in operating systems. The proposed approach is distinguished by the following novel contributions:

— It supports multiple entropy sources (kernel-level, local HRNG, and *remote* RNG). We propose an abstract interface for all such sources.
— It combines entropy from different sources and performs post-processing to devise a stream of secure random bits even if some of the RNG sources are compromised.
— It integrates into the OS via D-Bus, an ubiquitous **inter-process communication** (**IPC**) system with bindings for many programming languages.[1] Our goal is to *fairly* share entropy between multiple OS processes and VMs, while ensuring that entropy remains *isolated* between them.
— It does not require Linux kernel-level changes and can be seamlessly integrated into the OpenSSL 3 cryptographic library, thus establishing a robust TRNG interface for cryptographic applications.

We begin by introducing the Linux RNG subsystem and D-Bus. We then outline the proposed D-Bus entropy distribution service, followed by details on accessing local RNG sources and securing the connection to a remote QRNG service using **post-quantum cryptography** (**PQC**). We continue by describing our test suites and statistical results for real-time monitoring and fault mitigation. Finally, we explain how to integrate our solution into OpenSSL and libssl-based software and provide figures for the amount of bytes necessary for common cryptographic operations.

2 Linux RNG Landscape

The Linux kernel's RNG subsystem, also known as `random.c`, has recently undergone substantial revisions. Earlier, Linux had maintained two separate entropy input pools for `/dev/random` and `/dev/urandom`. The `random` device was designed to block read operations until sufficient entropy was available [2]. Conversely, `/dev/urandom` provided *instant* access to *best-effort* random numbers.

2.1 A Shift Away from True RNG in Linux

Since version 5.6 (March 2020), the Linux kernel has switched to the re-seedable ChaCha20-based CRNG and implements `/dev/random` in a non-blocking way. Now `/dev/random` generates new bytes even with insufficient entropy. On the one hand, this, is no longer considered a *true* RNG, e.g., according to German BSI[3] AIS 20/31 standard [32]. However, the non-blocking approach provides important benefits for time-critical scenarios, such as preventing **denial-of-service** (**DoS**) attacks during TLS connection establishment [26].

Efforts to meet **Federal Information Processing Standards** (**FIPS**) and other security standards have caused the Linux kernel to become fragmented, as different organizations apply their own patches to follow government rules. Stephan Müller, for example, rewrote `/dev/random`, implementing a "non-physical" (according to the classification of German BSI AIS 20/30 [32]) TRNG based on CPU execution time jitter [27] that meets NIST SP 800-90B, C and FIPS 140-3 standards for software-based entropy sources [3, 25, 28, 40]. The patch was not accepted upstream due to the pull request being a wholesale replacement rather than incremental [7].

An unsuccessful attempt to completely unify `random` and `urandom` was made in 2022 (Linux 5.18) [9]. "Linus Jitter Dance", resembling user-space software HAVEGE[4] [37], was deemed to remedy low-entropy conditions during boot. The changes were, however, quickly reverted due to boot test failures in **Quick Emulator** (**QEMU**) VMs where startup would hang indefinitely [8].

[1] We focus on Linux, but D-Bus also has Windows and macOS ports.
[2] `rng-tools` used to be a well-known set of utilities. It shipped with `rngd` daemon that injected hardware-based RNG entropy into the kernel's entropy input pool and manually incremented the entropy estimate. The purpose was, primarily, to reduce blocking periods of `/dev/random` [1].
[3] German Federal Office for Information Security.
[4] Hardware Volatile Entropy Gathering and Expansion.

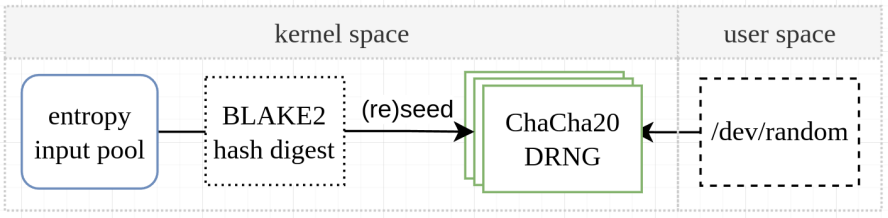


Fig. 1. Linux RNG architecture. Entropy is pushed from input pool through Blake2 hash to periodically seed ChaCha20 cryptographically secure PRNG that serves requests from `/dev/random` and `getrandom` sys call.

2.2 Linux RNG Internal Architecture

The Linux RNG collects entropy from several sources—such as CPU execution time jitter, driver interrupts, human interface devices, and even the timing variations of spinning hard drives. This raw data is then compressed using the Blake2 hashing algorithm to produce a digest, which is subsequently used to seed both the main and secondary[5] ChaCha20 deterministic RNGs.[6] The overall architecture is illustrated in Figure 1.

In order to fully seed the primary ChaCha20 deterministic RNG, a certain wakeup entropy threshold (default is 256 bits) must be reached[7]. Right after that, the system transitions to the *ready* state, and `/dev/random` is unblocked.[8]

2.3 Leaving True RNG up to the User-Space

Although the Linux RNG is part of the kernel, some maintainers have suggested also using user space[9] for implementing true RNG functionality [6].

According to [6], Lutomirski proposed that true randomness should be offered via a separate kernel interface or handled in user space by supplying raw event samples for dedicated entropy pools, thereby safeguarding primary RNG systems.

Similarly, Theodore Ts'o has noted the kernel's inability to reliably gather and estimate entropy across diverse hardware, and suggests that cryptographers collect entropy in user space. He advises against general applications like GPG or OpenSSL directly requesting "TrueRandom" outputs from the kernel, recommending instead that the kernel provide structured, authenticated access to raw noise sources so only authorized entities can securely utilize high-quality entropy [6].

3 Introduction to D-Bus

Desktop Bus (**D-Bus**) is an IPC and **Remote Procedure Calling** (**RPC**) system. Today, D-Bus has become an ubiquitous IPC solution on Linux.[10] It can handle complex inter-process procedure call marshalling and process authentication. Other features include lifecycle tracking, service activation, and security policies. Besides, it has bindings for many programming languages. Typically, D-Bus sets up two buses:

[5]Interestingly, in multi-core CPU and **Non-Uniform Memory Access** (**NUMA**) systems each core has its own "secondary" deterministic RNG. That reduces contention, since each processor accesses its nearby local memory faster.
[6]Such RNG reseed frequency (default: 60 seconds) is specified in `/proc/sys/kernel/random/urandom_min_reseed_secs`.
[7]Wakeup entropy threshold can be read from `/proc/sys/kernel/random/write_wakeup_threshold`.
[8]This is a departure from the historical behavior of `/dev/random`, which decremented the entropy counter when utilized and would thereafter block until sufficient entropy was available.
[9]an OS concept—the unprivileged execution environment where applications operate with restricted hardware access.
[10]Windows ports `windbus` and `dbus4win` are merged into the freedesktop dbus dev branch and are mostly finished [24]. A macOS port can be installed, e.g., via the `brew` package manager.

— The **system bus** enables system-wide communication, e.g., with Bluetooth devices, printers, laptop battery settings, and other hardware devices.
— The **session bus** is created when a user logs in, allowing applications to communicate within that shell session of the user. Prominent examples that expose APIs through session bus are *KDE wallet* and *GNOME keyring* password storage systems.

With D-Bus, services register on the bus, and other applications can discover them. A process offers its services by exposing objects. These objects have methods that can be invoked and signals that the object can emit to notify listeners. The APIs for objects exported over D-Bus are defined by the interfaces they support. The bus name identifies which service you want to communicate with. Usually, applications are identified by a "well-known" name, which is a dot-separated string starting with a reversed domain name, such as `org.freedesktop.NetworkManager` or `com.example.WordProcessor` [22]. For our TRNG service, we use `lv.lumii.TRNG` as a well-known name.

D-Bus includes a standard introspection mechanism for run-time querying of object interfaces. Many D-Bus bindings for dynamically-typed languages use this feature so that developers do not have to explicitly declare the remote interfaces that are being using.

Since many Linux distributions rely on *systemd* for launching background services or daemons (in essence, single-instance applications), there exists *dbus-daemon*, which allows automatic starting of *systemd* services when the first request arrives at the message bus [34].[11]

For local IPC D-Bus uses Unix domain sockets, which are bound to special files in the filesystem [33]. Interestingly, although D-Bus can also function over TCP/IP via IPv4 sockets (which are bound to an IPv4 address + port), remote TCP usage for D-Bus on Unix has been deprecated since 2018 due to its lack of built-in encryption and integrity checks. Besides, IPv4 sockets incur an additional TCP/IP overhead.[12]

Comparison with Other IPC Mechanisms

D-Bus is generally thought to be unsuitable for "high-performance" IPC due to multiple context switches, request validation, and response marshalling. In both the sense of latency, and throughput, D-Bus is likely to perform worse than other IPC primitives such as named pipes, `UNIX` sockets, and definitely well-implemented shared-memory. As stated in D-Bus FAQ, one-to-one communication is $\approx$ 2.5x slower than simply pushing the data over a socket [31].

To show that D-Bus is sufficiently performant for entropy distribution, we evaluate it against (i) raw `UNIX` sockets and (ii) gRPC. We do not aim to exhaustively benchmark IPC frameworks and skip many IPC primitives such as shared-memory ring buffers. Meaningful shared-memory evaluation would require implementing robust synchronization and failure handling for comparability.

We implemented a benchmark in which we make consecutive `unary` calls to a mock server via D-Bus, gRPC, and raw `UNIX` socket and measure total "wall time" for the client process. The request always contains one number—the number of random bytes to receive. The requests are made in a loop from the same client process. Both the server and the client are implemented in `C++` except for D-Bus, for which we re-use the service written in `Rust` (described in Section 4). The server reactively uses `getrandom` system call to fetch the requested number of random bytes.

Overall, the benchmark shows a significant latency gap between all three technologies. Surprisingly, gRPC achieved only 334 calls/s when limited to single in-flight request, which is $\approx$ 10x slower than D-Bus. See Figure 2.

[11] This is done by placing a `.service` file in a certain folder, e.g., `~/.local/share/dbus-1/services`.
[12] In POSIX headers these appear as `AF_UNIX`/`PF_UNIX` and `AF_INET`/`PF_INET`; `AF` refers to address families, `PF` to protocol families. On Linux these are defined to the same constant. We do not attempt to distinguish them.

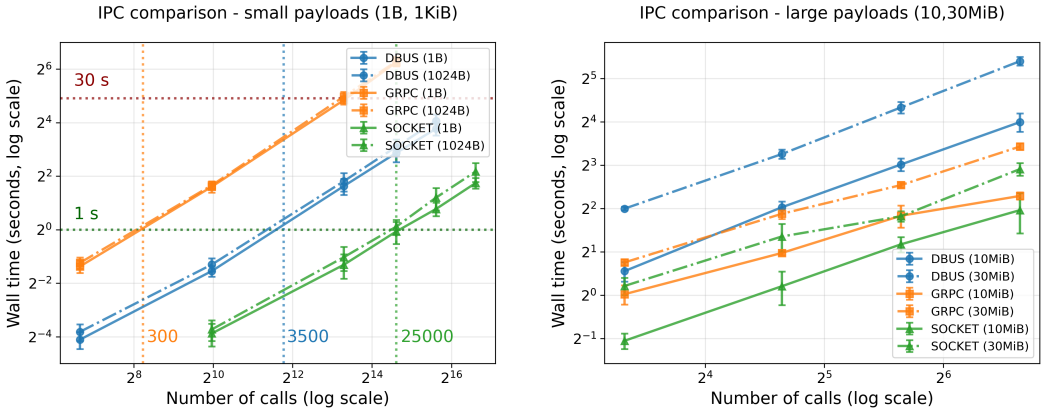


Fig. 2. D-Bus vs. raw Unix socket vs. gRPC. Total wall time for consecutive calls.

The laptop used for the benchmark has the following specifications:

— CPU: Intel 12th Gen i5-1235U (10c,12 t)
— Storage: KIOXIA BG5 NVMe SSD
— RAM: 32 GB DDR4 @ 3200 MHz
— OS: Ubuntu 25.04 LTS, Linux 6.14

For small requests (32 bytes per call), executing 10,000 consecutive calls in a single client loop, D-Bus achieved ~3574 calls/s (2.89 ± 0.48 s total), gRPC achieved ~334 calls/s (30.36 ± 3.73 s), while a raw Unix socket reached ~26,963 calls/s (0.39 ± 0.09 s). For large payloads of 10 MiB per call, measured over 100 consecutive calls, D-Bus sustained ~6.4 calls/s (15.96 ± 2.33 s), gRPC ~20.5 calls/s (4.90 ± 0.28 s), and raw Unix sockets ~28.2 calls/s (3.90 ± 1.21 s).

We conclude that D-Bus is well-suited for our use case at a latency of ≈ 0.3 ms and throughput of ≈ 60 MiB/s which is above 40 Mbps (≈ 4.8 MiB/s) advertised by the IDQ Quantis QRNG PCIe card. Our in-lab non-exhaustive tests on the test-system also indicate that for small requests, calls per seconds for D-Bus scales with the number of concurrent requests and peaks when in-flight requests do not exceed CPU core count achieving ≈ 4 times more calls/s.

In general D-Bus is not designed for transmitting large messages. It is considered a “control plane” in networking terminology and explicitly provides a way to transmit file descriptors between peers to establish the “data plane” [45]. On the test system, D-Bus has `max_message_size` set to 32 MiB by default in `/usr/share/dbus-1/system.conf`.

Compared to D-Bus, re-creating a “control plane” using Unix sockets would require additional engineering to handle several challenges: (i) retry mechanisms when the message receive-queue of a peer is full and cannot be awaited for availability, (ii) security policies as there is no access control in the abstract namespace, (iii) peer discovery and broadcast capabilities, and (iv) notifications when peers disconnect. While conceptually similar to D-Bus—both use Unix sockets for local communication—implementing these features would essentially recreate much of D-Bus functionality. Refer to the excellent blog post by David Herrmann [14] for more details on the topic.

4 QRNG D-Bus Service Prototype

We put forward the following requirements for the QRNG D-Bus service:

— It has to be shared among multiple applications; the random stream must be split securely and fairly.
— It must be able to combine multiple entropy sources (thus, it would remain secure in case one or a few of the entropy sources are compromised).

— It has to operate within a specified timeout. This matches POSIX[13]-way of implementing, e.g., `poll` system call. In case of timeout, output is not "expanded" to the requested length. For use-cases, where starvation is unacceptable, `/dev/urandom` (or `/dev/random` in modern kernels) can be used as a backup.
— Entropy sources and applied post-processing should be configurable (e.g., in the user's `.config` directory).
— To reduce latency and starvation, entropy should be buffered locally up to a specified amount.

We have developed a proof-of-concept prototype of the QRNG D-Bus service, conforming to the requirements above, in Rust. We chose Rust since its emphasis on safety, robust memory model, and low memory footprint make it an excellent choice for a background service for cryptographic needs. In addition, the *zbus* library provides easy-to-use high-level bindings for D-Bus (in both calling directions). Furthermore, Rust is now supported by the Linux kernel. That would allow us to use the same code base to develop Linux kernel modules in the future, should such a necessity occur.

The prototype daemon asynchronously waits for data from the remote QRNG device in a buffered manner. Buffer is implemented as a ring buffer. The data is then, by default, XORed[14] with that of the Linux RNG subsystem retrieved using the `getrandom` system call. By applying XOR, we not only mitigate risks of potential *total failure* of the HRNG device, but also reduce the *trust* required in the external service provider[15] when utilizing ***Entropy-as-a-Service*s (EaaSs)**[16].

In practice, the raw output of a HRNG may not conform to the uniform distribution out of the box. In this case, the result needs to be conditioned prior to use. If this conditioning is performed outside an entropy source, as opposed to the one built into the device by the manufacturer, the output is said to be *externally conditioned*. NIST SP 800-90C specifies SHA3-512 as one of the cryptographic primitives approved for conditioning [3]. Furthermore, according to the same set of recommendations, if we have a physical device that produces a string σ with $2t$ bits of entropy which is necessarily at least $2t$ bits long, and we can pick an *approved* conditioning function $C\colon \{0,1\}^* \to \{0,1\}^t$, then the string $C(\sigma)$ is t bits long and has *nearly* t (or, precisely, $(1-\varepsilon)t$) bits of entropy. For all NIST-approved C (e.g., SHA3-512), it is guaranteed that $\varepsilon < 2^{-64}$.

For the IDQ QRNG chips that we use, vendor guarantees at least 1.90 bits of min-entropy per 2-bit sample under IID assumptions, and more than 1.75 bits per 2-bit sample under the conservative non-IID track [16]. Conservatively, this corresponds to ≥ 0.875 bits of min-entropy per raw output bit. We can hash the raw output of length $2/0.875 \lesssim 2.5$ bits to obtain a string of length t with *nearly* t bits of entropy [30]. Under the independence assumption between entropy sources - compressed via hf QRNG output and `/dev/random`, the XORed output inherits the stronger lower bound, and degrades gracefully. We apply the SHA3-512 hash function by default to configured entropy sources on $512 \cdot 3 = 1536$ bit blocks. This can be disabled on a per-entropy source basis or replaced/supplemented with alternative methods such as Von Neumann entropy extractor [42] which removes bias from IID bits.

The overall scheme of the D-Bus service is sketched in Figure 3. As we see, the client process (end user application) does not access QRNG entropy directly—it receives only the conditioned and combined bitstring. Raw QRNG data are exposed only to the monitoring pipeline to avoid masking device degradations by post-processing; the monitoring service runs ENT3000 on the unconditioned raw stream and exports metrics to Prometheus (see Section 6).

[13]POSIX (IEEE 1003.1) standardizes Unix APIs—signals, threads, files, system calls in C—for cross-platform portability.
[14]We assume the two data sources are independent; under independence, XOR preserves a lower bound on min-entropy: $H_\infty(X \oplus Y) \geq \max\{H_\infty(X), H_\infty(Y)\}$.
[15]This QRNG XOR `/dev/random` approach is not a novel idea. It was mentioned in e.g., [4]. More details in Section 7.
[16]Such as the one provided by US NIST. See https://csrc.nist.gov/projects/entropy-as-a-service

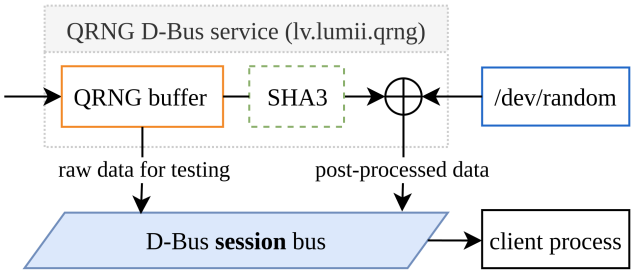


Fig. 3. QRNG D-Bus service architecture. Data flows from raw QRNG buffer through SHA3-512 conditioning, XOR with `/dev/random`, to client applications. Additional raw data access enables degradation monitoring.

For security reasons, our QRNG service deliberately uses the **session bus** rather than the system bus. This choice ensures that only processes that belong to the same user session can access the service. D-Bus enforces a security model where processes must share the same **user ID** (**UID**) as the session daemon to communicate, preventing other users from eavesdropping on or tampering with the entropy distribution. We delegate process authentication to this mechanism.

Starting session bus services for headless systems (and also when using users as an authentication scoping mechanism for shared entropy sources) is non-trivial but possible [18]. As an alternative solution, we could use the **system** bus. We would specify in the security policy that only *root* can own our service's bus name, as well as the whitelist of users that can access the service.

As our threat model, we assume a trustworthy kernel and exclude a fully privileged local adversary (root); our goal is per-UID isolation on the host and confidentiality against non-privileged local or remote attackers. Once an attacker has root, sniffing the session D-Bus is trivial; they can also replace the RNG (e.g., via a kernel module with a fixed-seed PRNG) or read process memory. The mitigation is operational: use least privilege, separate tenants/users (e.g., distinct UIDs/containers/VMs), and correct filesystem permissions

Our D-Bus service connects to entropy sources via the `EntropySource` interface. On the other hand, our D-Bus service implements the `AggregatedEntropy` interface for entropy consumers.

API for Entropy Sources

The `EntropySource` interface defines the `read_bytes(number_of_bytes, timeout_ms)` method, which returns a vector containing the requested number of random bytes or less, if it was impossible to aggregate the desired amount of entropy within the given timeout.[17] All possible errors (e.g., connection issues) are logged into a `systemd` journal.

Currently, we support three `EntropySource` types (we have implemented all of them):

- A file, such as device files `/dev/random` or `/dev/qrandom0` (the latter is created by the IDQ driver, which represents a Quantis QRNG device attached directly to the system). A pre-generated large random file may also serve as an entropy source; upon reaching the end of the file, the process continues by looping back to the beginning.
- The Linux `getrandom` system call. This represents the most direct method for obtaining entropy from the kernel, thereby minimizing latency and mitigating the risk of malicious overrides of `/dev/random`.
- Our remote PQC-secured QRNG service, which uses a websocket-based simple protocol with almost no overhead: the client sends one integer n—the number of bytes to retrieve, and the server replies with a stream of n bytes (in 1 KiB blocks). This is a long-lived websocket

[17] The interface also defines additional technical methods, which may have empty implementations.

connection, which remains active until the client disconnects. In Section 5, we provide more detail

All entropy sources and their configurations are specified in a TOML file located in the user's `$HOME/.config` directory, which is read on startup.

API for Entropy Consumers

Consumers (Linux applications) access the aggregated entropy via the D-Bus service `lv.lumii.TRNG`, which is launched automatically on the first call. Each call is an invocation of the `ReadBytes` method of the `lv.lumii.TRNG.AggregatedEntropy` interface. Clients usually rely on some D-Bus binding for their programming language to communicate with D-Bus.

The `ReadBytes` method accepts two 64-bit unsigned integers as input parameters: the number of random bytes to generate and the timeout in milliseconds. Upon success, the call returns a message containing the requested amount of random data (or less, if the desired amount of entropy was not available within the timeout).

5 Quantum-safe Remote QRNG Data Acquisition

There are several public QRNG services such as RandomNumbers.info[18], the Ruder Bošković Institute service[19], the Humboldt-Universität zu Berlin service[20] , and the Australian National University service.[21] However, some of these services operate over insecure HTTP, and none can establish quantum-safe TLS connections. If the QRNG accessed via D-Bus is located outside the host machine on a LAN or WAN, secure transmission is essential to protect the data. Our goal is to access a remote QRNG service in a quantum-safe way.

In 2022, we presented a technique for remotely acquiring QRNG data, enabling distribution to multiple remote clients through a mutually authenticated TLSv1.3 channel secured with PQC [20]. The service ensures a fair distribution of entropy to authenticated users, which means that if separate users authenticate as different clients to the web server, entropy starvation is avoided by misuse on a per-user basis. Developed software is open-source and documented at https://qrng.lumii.lv. External access to the service is provided upon request.

Our setup utilizes *QRNG web server* integrated with a physical **ID Quantique** (**IDQ**) Quantis QRNG PCIe device. The *QRNG web server* is placed behind a reverse proxy (*HAProxy*) compiled with PQC support from `liboqs` of the **Open Quantum Safe** (**OQS**) project. Currently, communication is secured with FrodoKEM key encapsulation mechanism and SPHINCS+ signature scheme, but they can be easily switched to other PQC algorithms [11, 29].

On the client side, the *receiver*—a dynamic link library on Windows (`.dll`) and a shared object on Linux (`.so`) were developed to establish a quantum-safe TLSv1.3 websocket connection with the *QRNG web server*. The shared library uses a forked Bouncy Castle[22] library with a modified handshake process to establish a heterogeneous quantum-safe communication channel between the OQS and Bouncy Castle implementation, see Figure 4. That allows us to test different TLS and PQC implementations and their interoperability, although it is possible to use `liboqs` at the client side as well. The *receiver* is implemented in Java and compiled using the GraalVM JDK as a native binary. By choosing WebSockets and maintaining persistent connections, the client needs only one PQC TLS handshake to access the remote QRNG web service (unless the client disconnects).

[18] www.randomnumbers.info (Quantis QRNG PCIe Legacy).
[19] http://random.irb.hr/index.php (QRBG121).
[20] formerly at qrng.physik.hu-berlin.de (PicoQuant PQRNG 150), currently inactive. See press release: www.picoquant.com/images/uploads/page/files/7287/2012_11_pqrng.pdf
[21] https://qrng.anu.edu.au (quantum fluctuations).
[22] Bouncy Castle provides FIPS-certified open-source cryptographic APIs for Java and C#. www.bouncycastle.org

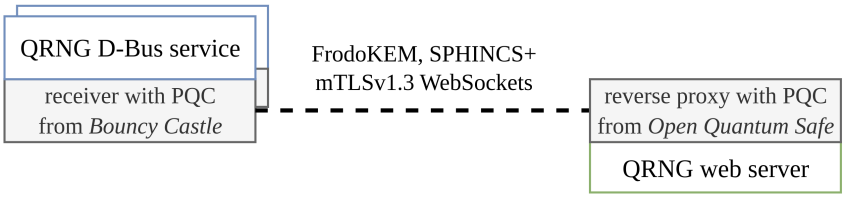


Fig. 4. Accessing remote QRNG device. The *receiver* and the reverse proxy implement a mutually-authenticated quantum-safe connection. Connection's remote nature is emphasized using a dashed line.

To connect to the *QRNG web server*, the following authentication credentials are required for **mutual TLS** (**mTLS**) authentication:

- `ca.truststore`: the root **certification authority** (**CA**) certificate used to sign the public keys of the server and all of the clients;
- `token.keystore`: a pair of private key and certificate (signature chain of its public key);
- `qrng.properties`: a configuration file with server hostname/IP, port, buffer sizes, and so on.

The number of issued authentication credentials must be carefully controlled. If too many credentials are distributed, clients could experience entropy starvation due to resource contention. To mitigate this, the system implements a **fair queuing** mechanism that guarantees a minimum entropy rate R_{min} (bits/s) per credential (i.e., per identity), where $R_{\text{min}} = R_{\text{total}}/N_{\text{max}}$, where R_{total} is the total entropy generation rate of the QRNG device and N_{max} is the maximum number of concurrently active clients. In our current setup, one IDQ Quantis device can serve up to 1,000 users with a guaranteed 1 KiB/s rate (which is actually much higher, because of server-side buffering). Furthermore, our QRNG web service is able to collect (and buffer) entropy from multiple QRNG chips. Thus, we can easily scale horizontally.

To ensure a fair distribution of entropy, the server implements a **longest-wait-first** (**LWF**) scheduling policy, taking into a consideration that a single client can post multiple requests, thus enforcing per-credential fairness without violating the single-outstanding-request rule.[23]

Although our QRNG web service is secured with PQC, it is worth to note that for compliance with NIST SP 800-90C, the QRNG device shall reside within the "same computing platform" as the cryptographic module, meaning that the entropy data shall not traverse an L3 network. However, if the remote QRNG is located in an isolated LAN network, many risks are averted.

6 QRNG Statistical Testing and D-Bus Service Monitoring

The remote nature of the shared QRNG device exposes it to *DoS* attacks and banal connection issues. Even if installed locally, a failure due to malicious intent or malfunction can affect the availability and quality of sourced entropy. To address these issues, a monitoring and alerting system is necessary for timely detection and potential response to such incidents.

We suggest implementing a dedicated pull-based monitoring service. It would periodically retrieve data and forward it to an external **statistical test suite** (**STS**). By maintaining separation between monitoring and core QRNG services, the QRNG D-Bus service codebase remains lean

[23]Formally, let C be the set of credentials (identities), and let Q be the global request queue. Each credential $c_j \in C$ has a *virtual queue* V_j that aggregates requests from all client processes that authenticate as c_j. At most one request from V_j can be present in Q at any time. When a client using credential c_j submits a request r with timestamp t, the server enqueues r into V_j. If V_j is not currently represented in Q, the server promotes the head of V_j to Q with eligibility time t. The scheduler serves Q in order of increasing eligibility time (oldest first). Upon service, random bytes are allocated in 1 KiB blocks to the active request. After completion, if V_j is non-empty, its next request becomes eligible and may be promoted to Q; otherwise, V_j remains idle.

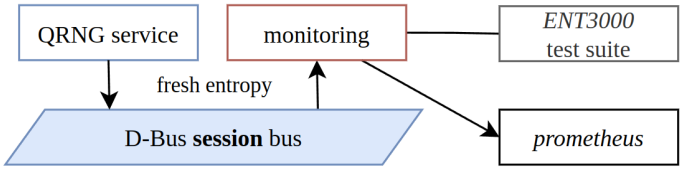


Fig. 5. QRNG D-Bus dedicated monitoring and alerting system. Arrows show monitoring data flow: raw entropy from D-Bus service to external statistical tests (ENT3000), then results to Prometheus for alerting.

and minimizes dependencies. This continuous operation aligns with the *online test* concept[24] as defined in BSI AIS 31 PTG.3.7 [32]. By integrating the monitoring service with Prometheus[25], the system can inspect the exposed data, collect the *min-entropy* estimate, and alert on failure via email, messenger, and so on, see Figure 5.

Each entropy source in the QRNG service is assigned a unique identifier during its configuration. The monitoring is configured separately with identifiers to monitor and at what fixed rate. We default this rate to 10 kilobytes per second to balance testing coverage with resource consumption.

To support this functionality, we extend the QRNG service with an additional D-Bus method - to retrieve raw data from a specific source by identifier. This raw data bypasses all post-processing and combination with other sources. Samples used for statistical testing are never reused for client requests or subsequent testing cycles.

The following subsection surveys available STSs and justifies our selection of ENT3000 for the implementation.

Statistical Test Suites

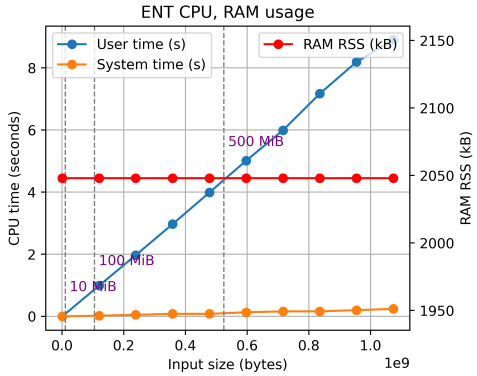
STSs give confidence in RNG's performance and absence of generic exploitable patterns. Many ready-made STSs are available for use. The most well-known among those are:

- *NIST STS*: developed by the National Institute of Standards and Technology, the NIST STS comprises 15 tests designed to evaluate various aspects of randomness. These tests require a minimum input size of 12.5 megabytes to achieve statistical significance [17]. The suite is widely recognized and utilized for the certification within the U.S. NIST **Cryptographic Module Validation Program** (**CMVP**).
- *Dieharder*: an extension of the original *Diehard* tests that includes additional assessments to provide a more comprehensive evaluation of randomness [5]. For reliable results, Dieharder recommends processing at least **1 gigabyte** of input data [5, 12].
- *ENT*: a simple tool for computing Shannon entropy (information density), Chi square tests, and a few other metrics [43].
- *ENT3000* (not to be confused with *ENT*): the ENT3000 suite is tailored for scenarios with limited data availability, enabling reliable randomness testing for data sizes below **1 megabyte** [41]. Unlike other suites, ENT3000 does not assume that the data are **independent and identically distributed** (**IID**), and it employs machine learning techniques to deliver accurate p-values even with constrained data volumes [41].

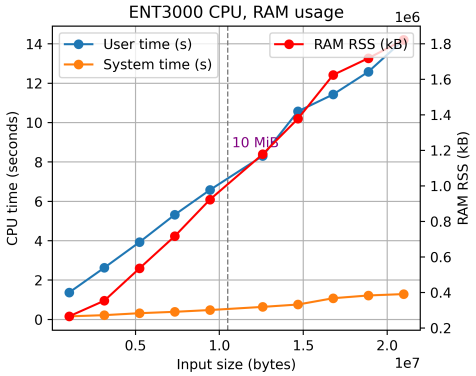
On our system, processing 100 MB of data with ENT took just 1 second, scaling linearly to 10 seconds for 1 GB (see Figures 6(a) and (b)). This performance makes ENT a viable option for real-time monitoring. However, note that by default, ENT reports Shannon entropy, rather than

[24]Unlike he *startup test*, which is conducted only once, the online test operates concurrently with the normal functioning of the service.

[25]Prometheus is an open-source monitoring and alerting toolkit. See https://prometheus.io/



(a) ENT resource usage for [1MiB;1GiB]. Execution time is linear. Memory usage, however, is constant. 1073741824,8.9,0.24,0.0,2048



(b) ENT3000 resource usage for [1MiB;20MiB]. Both execution time and memory usage grow linearly. 14.2 seconds 20971520 bytes

Fig. 6. ENT vs. ENT3000 comparison.

the more conservative and cryptographically relevant *min-entropy*. A key limitation of ENT is its assumption that the data are IID. If the data contain correlated bits, ENT will significantly overestimate the true entropy.

The lesser-known *ENT3000* is of particular interest for real-time monitoring where data throughput may be limited. It does not presuppose IID data but comes at the cost of slower processing as it takes about 14 seconds to process 20MiB of data, see Figure 6b. Our QRNG produces random data at a rate around 5 MiB/s when built-in post-processing is disabled and so, in context, makes ENT3000 viable.

As shown in Figures 6(a) and 6(b), ENT exhibits constant memory usage (resident set size, RSS) regardless of input size, indicating that it cannot detect periodic patterns. This behavior confirms that ENT assumes the input data are IID.

Interestingly, *ENT3000* reported a difference in entropy quality between the *RNG* and *SAMPLE* modes of the IDQ Quantis QRNG—an effect that was not observed with *ENT* (see Figure 9 in Appendix 9). In the default RNG mode, post-processing is carried out using a NIST 800-90 A/B/C compliant **Deterministic Random Bit Generator** (**DRBG**), whereas the SAMPLE mode provides the unprocessed raw quantum entropy data. Importantly, applications never receive the raw *SAMPLE* stream from our D-Bus service by default; clients only obtain the conditioned and combined output (see Section 4), while raw samples are reserved for the monitoring pipeline to detect device drift.

7 Solution Integration into OpenSSL

While it is possible to redirect reads from `/dev/random` file to our QRNG D-Bus service[26], that would affect only *some* applications, as others use the `getrandom` Linux system call directly. Seamlessly switching `/dev/random` consumers to our QRNG D-Bus service is not further explored here.

Given that our service runs in the user space (as opposed to the kernel space), we are looking for user-space integration of the QRNG D-Bus service into existing applications. Shared cryptographic libraries emerge as prime candidates, since it should be possible to re-implement their getrandom-like functions by substituting their output with ours. OpenSSL, being the leading cryptographic library, naturally stands out. In this section, we demonstrate how to seamlessly replace the OpenSSL built-in RNG mechanism with the invocation of QRNG D-Bus service.

[26]via symlink or FUSE mount.

OpenSSL is a widely used toolkit for general-purpose cryptography and secure communication. Starting with OpenSSL 3, a new *provider*-based architecture was introduced, deprecating the older *engine* framework. *Providers* can be built-in or dynamically loadable modules, each supplying implementations of cryptographic *operations*.

Because OpenSSL plays a central role in establishing TLS connections, integrating HRNGs into OpenSSL is desirable for various reasons. Notable examples include:

— the **Trusted Platform Module** (**TPM**) 2.0, used by the `tpm2-openssl` provider [39];
— a QRNG-based engine developed by Huang et al. for a cloud-based QRNG platform [15];
— FPGA-based implementations used for non-deterministic RNG (performance benefits) [19].

In a recent closely related work [4], Blanco-Romero et al. evaluate two ways to bring QRNG output into OpenSSL's TLS implementation: (i) an OpenSSL provider that reads directly from `/dev/qrandom0` (optionally XOR-ing with `/dev/random`); and (ii) feeding quantum entropy into the kernel pool via `rng-tools`, letting OpenSSL consume it through its default RNG stack. [4] They benchmark an nginx server with PQC-enabled TLS and conclude that the provider path can increase connection and time-to-first-byte relative to both *no-QRNG* and `rng-tools` modes, whereas `rng-tools` performs similarly to *no-QRNG* in their setup. [4] To highlight the differences between our approach and the one in [4], our OpenSSL integration uses a provider as in [4], but *sources* bytes over a D-Bus API from a user-space service that (a) can fairly share a QRNG across processes/VMs with per-credential scheduling, (b) can both XOR with `/dev/random` and apply configured post-processing, and (c) can acquire and *buffer* remote entropy over a quantum-safe channel. (Sections. 4, 5). D-Bus service centralizes implementation of aforementioned functions.

By examining a "toy" provider implementation of an expanded Vigenère cipher [21] and the TPM2 module integration provider [39], we developed our own OpenSSL provider, `librqrng`, which substitutes the default RNG operation. In an experiment to test the provider, we configured the QRNG D-Bus service to return the same pre-recorded value after each restart. Repeatedly generating RSA private keys with our custom RNG provider produced the same key, confirming that cryptographic operations do indeed depend on the substituted RNG. To connect to the D-Bus service, we use the `<systemd/sd-bus.h>` header.

Furthermore, we instrumented a D-Bus QRNG service to measure how many bytes OpenSSL requests for representative operations. Five RSA-2048 key generations consumed between 7.7 to 28.9 KiB per key (*mean* 16.6KiB); and 500 TLS 1.3 full handshakes (ECDHE + RSA-PSS server cert) consumed 208 369 bytes in total, i.e., ≈417 B per handshake. Interpreting these as upper bounds on entropy demand and assuming the device can supply up to 40 Mbps, the RNG budget alone would permit roughly ∼300 RSA-2048 keygens/s on average (range ∼170 to 636 across our five trials), and ∼12 000 TLS 1.3 full handshakes/s. For bottleneck that may be induced by the D-Bus IPC mechanism itself, see Section 3.

OpenSSL offers several ways to specify which *provider* should be used for a given *operation*: setting the `OPENSSL_MODULES` environment variable, using the `-provider` command-line option, configuring it programmatically, or modifying the `/etc/ssl/openssl.cnf` file. The environment variable method is particularly useful when one wants to restrict the scope of a custom or modified OpenSSL library to a specific environment. Specified providers have priorities. If a provider does not implement an operation, the next provider (if any) is attempted.

8 Full Architecture Outline

The overall solution decomposition is illustrated in Figure 7. The QRNG device is exposed through a web server and ensures fair distribution and authentication. Communication with the web server is secured with PQC. The entropy is then distributed to processes from a *QRNG service*.

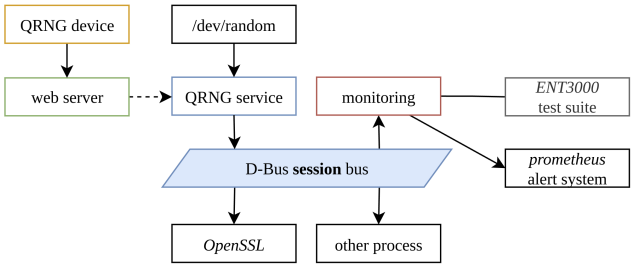


Fig. 7. Shared QRNG integration via D-Bus decomposition.

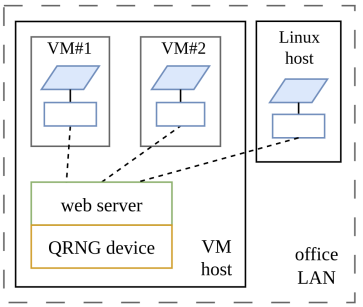


Fig. 8. Remote use-case topologies.

IPC is implemented via D-Bus, which takes care on RPC and authentication. Process authentication is important to isolate entropy starvation when the QRNG can't keep up with demand. Likewise, a flexible API allows behavior configuration on a per application basis when an QRNG entropy is not available.

The QRNG web server can be located either on the same machine, on the host of the guest VM, or in the **local area network** (**LAN**), see Figure 8. It is also possible to launch the QRNG web server in the wide area network such as in the case of *Entropy as a Service* offered by NIST and other companies. Alternatively, our QRNG web service qrng.lumii.lv can be used, albeit with a limited bitrate.

9 Conclusion

Linux's `/dev/random` is not a TRNG. CPU jitter-based or more generally RNGs that gather entropy from system data or user's interactions such as `/dev/random` do not allow precise stochastic modeling and rely on conservative entropy estimates and a large data compression rate [36].

We have presented a user-space solution for integrating a **quantum random number generator** (**QRNG**) into GNU/Linux. The QRNG hardware can be attached directly or accessed remotely via a quantum-safe link. Our solution provides software with a *true* RNG interface, which can be shared in LANs or in the VM host-guest scenario. This is necessary for **information-theoretical secure** (**ITS**) use-cases and welcome in many other scenarios.

Our Linux/D-Bus prototype demonstrates a portable pattern grounded in separation of concerns: decoupling cryptographic applications from entropy sourcing improves deployment flexibility and reduces code duplication. The pattern includes: (i) a user-space TRNG service with a minimal RPC surface (generate/timeout), authorization policy, and failure handling; (ii) XOR mixing under a source-independence assumption that preserves min-entropy, plus online health tests; (iii) remote entropy over mutually authenticated, quantum-safe TLS with fair queueing; and (iv) a plugin boundary (an OpenSSL provider) that other crypto stacks can mirror. Process-level separation enables language-agnostic integration and an independent daemon lifecycle (which allows for buffering). A user-space implementation keeps the OS kernel simpler while leveraging standard development tools and cross-platform IPC availability (D-Bus exists on Linux, macOS via Homebrew, and Windows ports). These abstractions are OS-agnostic and IPC-agnostic; D-Bus is an example, not a requirement.

The D-Bus QRNG service also offers a springboard for implementing standard-compliant post-processing and interfaces, e.g., "RBG3(XOR)" construction, defined in NIST 800-90A/B/C, within the Linux user space. This addresses the needs of applications requiring adherence to government standards, leveraging the simpler development environment of the user space over.

We envisage the following further research directions:

— Exploring alternative two-source entropy extractors (other than XOR) to combine output from `/dev/random` and the QRNG when a higher throughput is required.
— Conducting experiments with D-Bus ports for Windows and macOS.
— Implementing wiretaps for Linux (`getrandom` call), Windows (functions `CryptGenRandom`, `BCryptGenRandom`, and `RtlGenRandom`), and macOS (`getentropy` and `SecRandomCopyBytes`) for substituting default RNG implementations with our D-Bus-based solution.
— Testing and benchmarking `kdbus`, a kernel-level D-Bus-compatible IPC mechanism reported to significantly speed up messaging [14]. The `kdbus` implementation is available at boot time, and allows zero-copy messages between processes [2].
— Investigating secure enclaves and hardware-based trust mechanisms for D-Bus communication to further strengthen security guarantees [44].

Our work, presented in this article, can be used as a foundation for expanding these directions. Our solution is open-source and can be ported to Windows and macOS. Besides, we provide necessary abstractions for extending our code base.

References

[1] ArchWiki contributors. 2022. Rng-tools. Retrieved July 29, 2025 from https://wiki.archlinux.org/title/Rng-tools

[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX has become outdated. *;Login: The USENIX Magazine* 41, 3 (Fall 2016), 5–5. Retrieved September 12, 2025 from https://www.usenix.org/system/files/login/articles/login_fall16_02_atlidakis.pdf

[3] Elaine Barker, John Kelsey, Kerry McKay, Allen Roginsky, and Meltem Sönmez Turan. 2024. *Recommendation for Random Bit Generator (RBG) Constructions.* Technical Report NIST SP 800-90C 4pd. National Institute of Standards and Technology. DOI: https://doi.org/10.6028/NIST.SP.800-90C.4pd

[4] Javier Blanco-Romero, Vicente Lorenzo, Florina Almenares, Daniel Díaz-Sánchez, Carlos García Rubio, Celeste Campo, and Andrés Marín. 2024. Evaluating integration methods of a quantum random number generator in OpenSSL for TLS. *Computer Networks* 255 (2024), 110877. DOI: https://doi.org/10.1016/j.comnet.2024.110877

[5] Robert G. Brown. 2006. *Dieharder: A GNU Public License Random Number Tester.* Duke University, Physics Department, Durham, NC, USA. Retrieved April 30, 2025 from https://rurban.github.io/dieharder/manual/dieharder.pdf

[6] Jake Edge. 2020. Removing the /dev/random blocking pool. Retrieved April 30, 2025 from https://lwn.net/Articles/808575/

[7] Jake Edge. 2021. FIPS-compliant random numbers for the kernel. *LWN.net* (7 December 2021). Retrieved from https://lwn.net/Articles/877607/

[8] Jake Edge. 2022. Problems emerge for a unified `/dev/*random`. Retrieved April 30, 2025 from https://lwn.net/Articles/889452/. LWN.net.

[9] Jake Edge. 2022. Uniting the Linux random-number devices. Retrieved April 30, 2025 from https://lwn.net/Articles/884875/

[10] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. 2014. Not-So-Random numbers in virtualized linux and the whirlwind RNG. In *2014 IEEE Symposium on Security and Privacy* (San Jose, CA). IEEE, Los Alamitos, CA, USA, 559–574. DOI: https://doi.org/10.1109/SP.2014.42

[11] Federal Office for Information Security (BSI). 2024. *Cryptographic Mechanisms: Recommendations and Key Lengths.* Technical Guideline TR-02102-1. Federal Office for Information Security (BSI), Bonn, Germany. Retrieved May 4, 2025 from https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf

[12] Cameron Foreman, Richie Yeung, and Florian J. Curchod. 2024. Statistical testing of random number generators and their improvement using randomness extraction. *Entropy* 26, 12 (Dec. 2024), 1053. arXiv:2403.18716 DOI: https://doi.org/10.3390/e26121053

[13] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 35.

[14] David Herrmann. 2015. *From AF_UNIX to kdbus.* Retrieved September 10, 2025 from https://dvdhrm.wordpress.com/2015/06/20/from-af_unix-to-kdbus/. Blog post.

[15] Leilei Huang, Hongyi Zhou, Kai Feng, and Chongjin Xie. 2021. Quantum random number cloud platform. DOI: https://doi.org/10.1038/s41534-021-00442-x Cited by: 37; All Open Access, Gold Open Access.

[16] ID Quantique SA. 2023. *SP800-90B Non-Proprietary Public Use Document: Quantis QRNG IID Chips IDQ250C2, IDQ250C3, IDQ6MC1, IDQ20MC1, IDQ20MC1-S1, IDQ20MC1-S3*. Public Use Document E63. National Institute of Standards and Technology, Cryptographic Module Validation Program. Retrieved September 12, 2025 from https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/entropy/E63_PublicUse.pdf. NIST CMVP Entropy Validation Document.

[17] Lawrence E. Bassham III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, et al. 2010. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22r1a. National Institute of Standards and Technology, Gaithersburg, MD, USA. 1–131 pages. DOI: https://doi.org/10.6028/NIST.SP.800-22r1a. Revised April 2010.

[18] Keithel. 2018. Starting systemd services sharing a session D-Bus on headless system. Retrieved September 12, 2025 from https://serverfault.com/questions/892465/starting-systemd-services-sharing-a-session-d-bus-on-headless-system. ServerFault answer.

[19] Mohamed Khalil-Hani, Vishnu P. Nambiar, and Muhammad Nadzir Marsono. 2010. Hardware Acceleration of OpenSSL Cryptographic Functions for High-Performance Internet Security. 374-379 pages. Retrieved from https://api.semanticscholar.org/CorpusID:16126362

[20] Sergejs Kozlovičs and Juris Vīksna. 2022. POSTER: A transparent remote quantum random number generator over a quantum-safe link. In *Applied Cryptography and Network Security Workshops*. Lecture Notes in Computer Science, Vol. 13285. Springer International Publishing, Rome, Italy, 595–599. DOI: https://doi.org/10.1007/978-3-031-16815-4_32

[21] Richard Levitte. 2023. `vigenere` – an OpenSSL 3 provider that implements an expanded Vigenère cipher. Retrieved January 23, 2025 from https://github.com/provider-corner/vigenere

[22] Linus Lewandowski. 2016. Pydbus documentation. Retrieved April 13, 2025 from https://pydbus.readthedocs.io/en/latest/legacydocs/tutorial.html#d-bus-objects

[23] Vaisakh Mannalatha, Sandeep Mishra, and Anirban Pathak. 2023. A comprehensive review of quantum random number generators: concepts, classification and the origin of randomness. *Quantum Information Processing* 22, 12, Article 439 (2023), 1–45. Retrieved from https://api.semanticscholar.org/CorpusID:247187462

[24] Simon McVittie, Ralf Habacker, and Joe Rayhawk. 2022. *D-Bus - A Message-Bus System (Project Home)*. freedesktop.org. Retrieved May 4, 2025 from https://www.freedesktop.org/wiki/Software/dbus/

[25] Stephan Müller. 2019. /dev/random - a new approach with full SP800-90B compliance. Retrieved April 30, 2025 from https://lore.kernel.org/lkml/20191111181721.23209-1-smueller@chronox.de/. PATCH v24 00/12, Linux Kernel Mailing List.

[26] Stephan Müller, Sebastian Mayer, Caroline Holz auf der Heide, and Andreas Hohenegger. 2022. *Documentation and Analysis of the Linux Random Number Generator*. BSI Study, Version 5.0 Study No. 449. Federal Office for Information Security (BSI), Germany, Bonn, Germany. Prepared for BSI by atsec information security GmbH, Accessed: 2025-04-30.

[27] Stephan Müller. 2022. *CPU Jitter Based Non-Physical True Random Number Generator*. Technical Report. chronox. Retrieved July 30, 2025 from https://www.chronox.de/jent/CPU-Jitter-NPTRNG-v2.2.0.pdf

[28] National Institute of Standards and Technology. 2019. *FIPS PUB 140-3: Security Requirements for Cryptographic Modules*. Technical Report FIPS PUB 140-3. National Institute of Standards and Technology, Gaithersburg, MD. DOI: https://doi.org/10.6028/NIST.FIPS.140-3

[29] National Institute of Standards and Technology. 2024. NIST Releases First 3 Finalized Post-Quantum Encryption Standards. Retrieved April 30, 2025 from https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

[30] Squeamish Ossifrage. 2018. What is the relationship between entropy conditioning and final output bias in a TRNG? Cryptography Stack Exchange. Retrieved September 17, 2025 from https://crypto.stackexchange.com/questions/58023/what-is-the-relationship-between-entropy-conditioning-and-final-output-bias-in-a/58168#58168

[31] Havoc Pennington and David A. Wheeler. 2005. D-Bus FAQ. Retrieved September 9, 2025 from https://dbus.freedesktop.org/doc/dbus-faq.html

[32] Matthias Peter and Werner Schindler. 2024. *A Proposal for Functionality Classes for Random Number Generators*. Technical Report AIS 31, Version 3.0. Federal Office for Information Security (BSI), Bonn, Germany.

[33] Lennart Poettering. 2015. The new sd-bus API of systemd. Retrieved April 30, 2025 from https://0pointer.net/blog/the-new-sd-bus-api-of-systemd.html

[34] Lennart Poettering and Zbigniew Jędrzejewski-Szmek. 2024. System and Service Manager. Retrieved April 16, 2025 from https://systemd.io/

[35] Thomas Ristenpart and Scott Yilek. 2010. When good randomness goes bad : Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*. The Internet Society, San Diego, California, USA, 121–128. Retrieved from https://www.ndss-symposium.org/ndss2010/when-good-randomness-goes-bad-virtual-machine-reset-vulnerabilities-and-hedging-deployed. Symposium dates: 28 February–3 March 2010.

[36] Werner Schindler. 2023. Overview of AIS 20/31. NIST Presentation. Retrieved September 19, 2025 from https://csrc.nist.gov/csrc/media/Presentations/2023/overview-of-ais-2031/images-media/session-2-schindler-overview-of-ais-20-31.pdf

[37] André Seznec and Nicolas Sendrier. 2003. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation* 13, 4 (Oct. 2003), 334–346. DOI: https://doi.org/10.1145/945511.945516

[38] Mario Stipčević and Çetin Kaya Koç. 2014. True random number generators. In *Open Problems in Mathematics and Computational Science*, Çetin Kaya Koç (Ed.). Springer International Publishing, Cham, 275–315. DOI: https://doi.org/10.1007/978-3-319-10683-0_12

[39] TPM2-software community and Petr Gotthard. 2025. `tpm2-openssl` - OpenSSL Provider for TPM 2.0 integration. Retrieved January 23, 2025 from https://github.com/tpm2-software/tpm2-openssl

[40] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. 2018. *NIST SP 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation.* Technical Report NIST SP 800-90B. National Institute of Standards and Technology. DOI: https://doi.org/10.6028/NIST.SP.800-90B

[41] Paul Uszak. 2024. ENT3000: Rationale & FAQ. Retrieved from http://www.reallyreallyrandom.com/ent3000/rational/index.html

[42] John von Neumann. 1951. Various techniques used in connection with random digits. *Applied Math Series* 12 (1951), 36–38.

[43] John Walker. 2008. ENT — Fourmilab Random Sequence Tester. GitHub repository, Retrieved April 30, 2025 from https://github.com/Fourmilab/ent_random_sequence_tester

[44] Newton C. Will, Tiago Heinrich, Amanda B. Viescinski, and Carlos A. Maziero. 2020. A trusted message bus built on top of D-Bus. In *Proceedings of the 20th Brazilian Symposium on Information and Computational Systems Security (SBSeg 2020).* Sociedade Brasileira de Computação, Sociedade Brasileira de Computação (SBC), Petrópolis, Rio de Janeiro, Brazil, 175–187. DOI: https://doi.org/10.5753/sbseg.2020.19236

[45] Philip Withnall. 2017. Comment on “Passing a large data structure over D-Bus”. Stack Overflow comment; Retrieved September 10, 2025 from https://stackoverflow.com/questions/6220704/passing-a-large-data-structure-over-dbus#comment76868948_12622541

Appendix
IDQ Quantis PCIe-40M statistical testing

ENT3000 test results for RNG mode

```
Testing quantis-40m-10MiB-rng.bin
Testing 10,485,760 bytes.
-----------------------------------
Monobit,          p = 0.9106,  PASS.
ChiBit,           p = 0.5639,  PASS.
ChiByte,          p = 0.0986,  PASS.
MeanByte,         p = 0.4846,  PASS.
KS,               p = 0.8081,  PASS.
Pi,               p = 0.4554,  PASS.
Shells,           p = 0.6825,  PASS.
Gaps,             p = 0.3337,  PASS.
Avalanche,        p = 0.9998,  PASS.
Runs,             p = 0.5522,  PASS.
RunUps,           p = 0.7147,  PASS.
Prediction,       p = 0.8741,  PASS.
UnCorrelation,    p = 0.9562,  PASS.
-----------------------------------
13/13 tests passed.
Finished.
```

ENT3000 test results for SAMPLE mode

```
Testing quantis-40m-10MiB-sample.bin
Testing 10,485,760 bytes.
-----------------------------------
Monobit,          p = 0.0000,  FAIL.
ChiBit,           p = 0.0000,  FAIL.
ChiByte,          p = 0.0218,  PASS.
MeanByte,         p = 0.0001,  FAIL.
KS,               p = 0.0516,  PASS.
Pi,               p = 0.0219,  PASS.
Shells,           p = 0.0915,  PASS.
Gaps,             p = 0.5493,  PASS.
Avalanche,        p = 0.9981,  PASS.
Runs,             p = 0.6244,  PASS.
RunUps,           p = 0.3484,  PASS.
Prediction,       p = 0.3494,  PASS.
UnCorrelation,    p = 0.5525,  PASS.
-----------------------------------
10/13 tests passed.
Finished.
```

ENT test results for RNG mode

```
Entropy = 7.999998 bits per byte.

Optimum compression would reduce the
size of this 104857600 byte file by 0
percent.

Chi square distribution for 104857600
samples is 238.29, and randomly would
exceed this value 76.64 percent of the
times.

Arithmetic mean value of data bytes is
127.4831 (127.5 = random). Monte Carlo
value for Pi is 3.141762205 (error 0.01
percent). Serial correlation coefficient
is -0.000234 (totally uncorrelated =
0.0).
```

ENT test results for SAMPLE mode

```
Entropy = 7.999996 bits per byte.

Optimum compression would reduce the
size of this 104857600 byte file by 0
percent.

Chi square distribution for 104857600
samples is 576.99, and randomly would
exceed this value less than 0.01 percent
of the times.

Arithmetic mean value of data bytes is
127.4146 (127.5 = random). Monte Carlo
value for Pi is 3.143071867 (error 0.05
percent). Serial correlation coefficient
is -0.000048 (totally uncorrelated =
0.0).
```

Fig. 9. Side-by-side comparison of test results for IDQ Quantis PCIe-40M.

Received 15 May 2025; revised 24 September 2025; accepted 5 November 2025