

RockyGuard Library API Reference

Version 1.2.1

RockyGuard - C++ License Checking Library

This document provides comprehensive documentation for the RockyGuard library,
a C++17 library for node-locked and floating license management.

RockyGuard Library - API Reference

Version 1.2.1-dev

Copyright (c) 2025-2026 Rocky Software Inc. All rights reserved.

About this reference:

Describes the RockyGuard v1.2.1 C++ API. The "-dev" suffix above marks an in-flight build during the v1.2.1 development cycle; the suffix is dropped at the release cut. APIs new since v1.2.0 are marked inline with "(v1.2.1+)". Everything else has been present since v1.2.0 and is described as the current behaviour of the library you are linking against. Companion document: docs/Customer_Documentation.txt -- the operator-facing manual; this file is the engineer-facing API reference.

Version applicability (READ THIS IF YOU ARE NOT SURE WHICH RELEASE YOU HAVE):

This reference documents the v1.2.1 C++ API. If your installed library is v1.2.0, the symbols and methods marked "(v1.2.1+)" -- including `LicenseVerifier::check_version()`, `SignatureAlgorithm::AutoDetect`, `LicenseStatus::VersionMismatch`, `LicenseStatus::MachineSeatLimitReached`, and the new `FloatingServerConfig` fields `max_leases_per_machine_id`, `log_max_bytes`, `log_keep_count` -- DO NOT EXIST in your headers. Any code that references them will fail to compile against a v1.2.0 build of the library. To check the version you have linked, read `ROCKYGUARD_VERSION_STRING` from `<rockyguard/version.h>`. If your installed library is older than this reference, consult the docs/Customer_API_Reference.pdf inside the customer zip you actually received -- that copy matches your binary. The website (rockyguard.dev) only publishes the latest release; mixing v1.2.1 reference material with a v1.2.0 binary will produce "undeclared identifier" / "no member named" errors.

v1.2.1+ also renames the floating-license CLI binaries from `floating_server_example` / `floating_client_example` to `rg_floating_server` / `rg_floating_client`; references throughout this reference use the new names.

TABLE OF CONTENTS

1. Quick Start
2. Namespace and Headers
3. Diagnostic Output
4. Enums and Types
5. License Struct
6. LicenseVerifier Class
7. HardwareFingerprint Class
8. FloatingLicenseClient Class (Premium)
9. CLI Tools Reference
10. Integration Examples

1. QUICK START

```
#include <rockyguard/rockyguard.h>

static constexpr char PUBLIC_KEY[] = R("-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEA...your key...
-----END PUBLIC KEY-----");

int main() {
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);
    auto result = verifier.load("license.json");
    if (!result) { std::cerr << result.message << "\n"; return 1; }
    result = verifier.check_node_locked();
    if (!result) { std::cerr << result.message << "\n"; return 1; }
    // Licensed and running
}
```

Use the provided CLI tools to manage licenses. See §9.

If you are integrating via an AI agent (Claude Code, Cursor, GitHub Copilot, ChatGPT, Gemini, Cody, or similar), point it at `AI_INTEGRATION_GUIDE.md` at the package root or at https://rockyguard.dev/AI_INTEGRATION_GUIDE.md. The guide is written in AI-imperative voice and walks the agent through the full integration deterministically (CMake wiring, public-key embedding, startup verification, feature gating). See `Customer_Documentation §2.6` for the human-side description of that path.

2. NAMESPACE AND HEADERS

All API is in the rockyguard namespace.

```
#include <rockyguard/rockyguard.h>           // All-in-one include
#include <rockyguard/types.h>                 // Enums, LicenseResult
#include <rockyguard/license.h>               // License data struct
#include <rockyguard/license_verifier.h>      // License verification
#include <rockyguard/hardware_fingerprint.h>  // Machine fingerprinting
#include <rockyguard/floating_client.h>       // Floating client (Premium)
#include <rockyguard/export.h>                // ROCKYGUARD_API export macro
#include <rockyguard/version.h>               // ROCKYGUARD_VERSION_* macros
```

The `export.h` header carries the `ROCKYGUARD_API` macro that decorates every public class and function. You normally do not include it directly -- the other headers pull it in -- but it is listed here for completeness so you know what is in `include/rockyguard/`. `version.h`, generated at library build time from `CMakeLists.txt`, provides `ROCKYGUARD_VERSION_MAJOR` / `MINOR` / `PATCH` integer macros and a `VERSION_STRING` constant for runtime display.

Linking with CMake (static, Windows; full link line with per-config CRT variant):

```
add_executable(your_app main.cpp)
target_include_directories(your_app PRIVATE
    ${ROCKYGUARD}/include
    ${ROCKYGUARD}/deps/include)
target_link_libraries(your_app PRIVATE
    $<IF:$<CONFIG:Debug>,
        ${ROCKYGUARD}/lib/static/rockyguard_mdd.lib,
        ${ROCKYGUARD}/lib/static/rockyguard.lib>
    ${ROCKYGUARD}/deps/lib/libssl.lib
    ${ROCKYGUARD}/deps/lib/libcrypto.lib
    ws2_32 crypt32 iphlapi ole32 oleaut32 wbemuuid)
```

Where `${ROCKYGUARD}` is the path to the extracted package directory. See [Customer_Documentation §3.1](#) for the Linux equivalent (Linux ships a single static lib; the CRT-variant split is Windows-only).

CRT VARIANTS on Windows (v1.2.1+): the static-lib bundle ships TWO variants of the RockyGuard library so a consumer can build in either Release or Debug without LNK2038 / LNK1319 errors on `_ITERATOR_DEBUG_LEVEL` or `RuntimeLibrary`:

```
rockyguard.lib           /MD-built -- link from a Release consumer.
rockyguard_mdd.lib       /MDD-built -- link from a Debug consumer.
```

The bundled OpenSSL import libs (`libssl.lib`, `libcrypto.lib`) are CRT-agnostic: `vcpkg's` Release and Debug import stubs are byte-identical, and OpenSSL's API does not pass C-runtime objects across the DLL boundary, so a single pair serves both consumer variants. The customer-side `CMakeLists` template under `examples/` uses `IMPORTED_LOCATION_RELEASE` / `IMPORTED_LOCATION_DEBUG` so a consumer running `cmake --build build --config Debug` automatically picks the right variant -- the explicit `$<IF:$<CONFIG:Debug>,...>` form above is only

needed when wiring in a hand-rolled CMakeLists.

Coming in v1.3: a shipped rockyguard-config.cmake imported target so consumers can simply

```
find_package(rockyguard CONFIG REQUIRED)
target_link_libraries(your_app PRIVATE rockyguard::rockyguard)
```

with all transitive dependencies (OpenSSL, Windows system libraries, the right CRT variant) handled by the imported target. v1.2.1 ships the explicit link line above; v1.3 will keep it working alongside the imported-target form.

For shared (DLL) linking, see [Customer_Documentation §3.2](#).

3. DIAGNOSTIC OUTPUT

The library writes human-readable diagnostic messages to stderr in a small set of well-defined situations. Every line is prefixed with "[RockyGuard] WARNING: " so it is easy to identify, filter, or redirect.

Conditions that produce a stderr warning:

- License loading: license_id or product field is empty in the payload (per-license clock-man...
- License creation: same fields empty when generating a license via the license_create CLI to...
- Date parsing: a license issued_at or expires_at field is not valid ISO-8601. The library tr...
- Hardware fingerprint: any one of the four components (MAC, CPU, Disk, Motherboard) cannot b...
- Integrity check (DLL builds): library binary path cannot be determined, binary cannot be re...
- Vendor license loading: a date field in the vendor license is not parseable.
- Floating server (Premium): logger cannot open the configured log file (server falls back to...

These messages are diagnostic: they do not affect return values. The library still fails closed via LicenseStatus where the condition is security-relevant (e.g., an empty license file path returns MalformedFile regardless of whether a warning was emitted). A customer can therefore safely treat the absence of a warning as "all expected fields were valid" and can use return values for control flow.

How to suppress these warnings:

Process-wide redirect at startup (recommended for GUI hosts):

```
// C++
std::freopen("nul", "w", stderr); // Windows
std::freopen("/dev/null", "w", stderr); // Linux
```

Note: this also suppresses your application's own stderr output. If you want to keep your own diagnostics, redirect stderr to a file instead and let the [RockyGuard] prefix filter the file at read time:

```
std::freopen("rockyguard.log", "a", stderr);
```

Shell-level redirect when launching the host:

```
your_app 2> /dev/null          # Linux
your_app.exe 2> NUL            # Windows cmd.exe
your_app.exe 2> $null          # PowerShell
```

A configurable logging callback (set_log_callback) that lets the host application route library diagnostics through its own log sink, with selectable severity levels, is on the v1.3 roadmap. The warnings remain on stderr until then.

4. ENUMS AND TYPES

4.1 enum class LicenseType

NodeLocked	License bound to specific hardware
Floating	License managed by a pool server (Premium)

4.2 enum class LicenseStatus

Valid	License is valid
Expired	License has expired
InGracePeriod	Expired but within grace period
HardwareMismatch	Machine doesn't match the license
SignatureInvalid	License file has been tampered with
MalformedFile	License file cannot be parsed
FeatureNotLicensed	Requested feature not in this license
NoLicensesAvailable	Floating: all licenses in use (pool exhausted)
MachineSeatLimitReached	Floating: this machine reached its per-machine cap
VersionMismatch	Application version does not satisfy version_range
ServerUnreachable	Floating: cannot reach server
LibraryNotInitialized	Vendor license not loaded (generation tools only)
TierNotAuthorized	Feature requires Premium tier
GenerationLimitReached	License generation limit exceeded
MachineNotAuthorized	Machine not authorized for generation
ClockManipulated	System clock rolled back detected
IntegrityCheckFailed	Library binary has been modified
NotYetValid	Reserved. Present in the public header so a future release can add a "license not yet active" code path without an ABI break; no shipped code path in v1.2.1 returns this value. A `switch` that handles every LicenseStatus should include this case (route it the same way as MalformedFile) to silence "unhandled enum" warnings.

4.3 enum class SignatureAlgorithm

Ed25519	Ed25519 (recommended)
RSA_SHA256	RSA with SHA-256
AutoDetect	(v1.2.1+; default for the LicenseVerifier constructor) Inspect the loaded key and pick the matching algorithm. Eliminates the customer-tracked "must remember which algo I used" state. Pass Ed25519 or RSA_SHA256 explicitly to override.

4.4 struct LicenseResult

Member	Type	Description
--------	------	-------------

```
-----
status          LicenseStatus  Result status
message         std::string   Human-readable description
grace_days_remaining  int           Grace days left (0 if N/A)
```

operator bool() returns true if status is Valid or InGracePeriod:

```
if (auto result = verifier.load("license.json")) {
    // License is valid
}
```

5. LICENSE STRUCT

Defined in: <rockyguard/license.h>

5.1 Fields

Member	Type	Default
license_id	std::string	" " (REQUIRED *)
licensee	std::string	" "
product	std::string	" " (REQUIRED *)
version_range	std::string	" "
type	LicenseType	NodeLocked
hardware_fingerprint	std::string	" "
fingerprint_match_threshold	int	2
issued_at	std::string	" "
expires_at	std::string	" "
grace_period_days	int	0
max_concurrent_users	int	0
features	std::vector<std::string>	{ }
metadata	std::map<std::string, std::string>	{ }

(*) license_id and product are REQUIRED by the license generator (license_create CLI returns an error and refuses to write the license if either is empty). They are STRONGLY RECOMMENDED on the verifier side: a license that somehow reaches load() with either field empty will load successfully but the per-license clock-manipulation defense becomes weaker, and the library emits a stderr warning at load time (see §3 "Diagnostic Output"). The asymmetry is deliberate: forward compatibility with future schema versions that might omit these fields takes precedence over hard-rejecting an otherwise-valid signed license.

The metadata field:

A customer-defined string -> string map (std::map<std::string, std::string>) that the library round-trips through the signed license file without inspecting it. The full discussion -- semantics, why metadata instead of a sidecar config, recommended use cases (audit trails, anti-leak forensics, display strings, numeric limits beyond binary feature flags), copy-pasteable issuance and read-side examples, and size guidance -- is in Customer_Documentation §8.2 "Payload Fields". On the API side: set values from the license_create CLI with --metadata key=value (Customer_Documentation §7.3 has the parse-rule details); read them at verification time via verifier.license().metadata.

Date format for `issued_at` and `expires_at`:

The canonical format is ISO 8601 with explicit time and UTC timezone -- "2027-12-31T23:59:59Z" -- and that is the form `license_create` writes when called with `--expires`. The parser also accepts a date-only form ("2027-12-31"), which is interpreted as 23:59:59 UTC on that date (i.e. the license expires at end-of-day UTC on the named day). Mixing the two within one license set is fine; the verifier compares parsed UTC instants regardless of which form produced them. Any other shape -- local-timezone offsets without "Z", date-only with US-style separators, etc. -- is treated as unparseable, which the library treats fail-closed as already expired, with a warning emitted to `stderr` (see §3). For new licenses, prefer the full ISO 8601 form: it is unambiguous and matches what `license_verify` and the customer-facing CLI tools display.

5.2 `from_json()`

```
static License from_json(const std::string& json_str)
```

Parses a JSON payload string into a `License`. Throws `std::exception` on malformed JSON or wrong field types. Most customers never call this directly: `LicenseVerifier::load()` uses it internally and reports parse failures via `LicenseStatus::MalformedFile` rather than exceptions.

5.3 `to_json()`

```
std::string to_json() const
```

Serializes this `License` back to a JSON string. The result is the unsigned PAYLOAD form (no envelope, no signature). Used by the license generator on the vendor side; on the verifier side this is mainly useful for diagnostics ("what license is currently loaded?") via the `License` object returned by `LicenseVerifier::license()`.

5.4 `is_expired()`

```
bool is_expired() const
```

Returns true if `expires_at` is set and the current system time is past it. The verifier uses this internally; you can call it directly if you want the raw boolean without the grace-period semantics that `LicenseStatus::InGracePeriod` provides.

5.5 `is_in_grace_period()`

```
bool is_in_grace_period() const
```

Returns true if the license has expired but is still within the `grace_period_days` window. Equivalent to `(is_expired() && grace_days_remaining() > 0)`.

5.6 `grace_days_remaining()`

```
int grace_days_remaining() const
```

Returns the number of grace days remaining after expiry, clamped at zero. Returns 0 if the license has not expired or if

the grace period has already elapsed.

5.7 has_feature()

```
bool has_feature(const std::string& feature) const
```

Returns true if the named feature is in the features vector (exact match, case-sensitive). The verifier's `check_feature()` wraps this and returns it as a `LicenseResult` so it composes with the rest of the verification API.

6. LICENSEVERIFIER CLASS

Defined in: <rockyguard/license_verifier.h>

Verifies end-user licenses.

Order of operations:

The expected lifecycle is: construct, load(), then any number of check_*() calls and license() accesses. Calling check_node_locked(), check_feature(), check_version(), or check_expiry() before a successful load() is defined behavior, not a crash: each returns {LicenseStatus::MalformedFile, "No license loaded"}. license() called before load() returns a default-constructed License (all fields empty / zero / NodeLocked); is_loaded() (§6.9) is the recommended way to test whether load() has succeeded. The constructor itself never reads a license -- it only loads the public key.

What each check method re-evaluates:

The check methods are not all equivalent. Some re-run the full expiry + anti-tampering + clock-manipulation pipeline; others are pure data lookups against the license that load() already parsed. Choose based on whether you need a fresh time check or just a "what does this license say" answer:

Method	What it re-evaluates	Side effects and threading
load() / load_from_string()	Full pipeline: signature, integrity self-check, clock-manipulation check via time anchors, expiry. Implicitly calls check_expiry() before returning.	Writes time-anchor files (§9.2). Must not race with anything else on the same verifier instance; see "Thread safety" below.
check_expiry()	Full pipeline: integrity self-check, clock-manipulation check, expiry / grace-period evaluation.	Writes time-anchor files (§9.2). Single-writer; see "Thread safety" below.
check_node_locked()	Hardware-fingerprint match against the license's hardware_fingerprint. On a hardware pass, tail-calls check_expiry() so the full integrity + clock + expiry pipeline runs too. A license that passed load() but expires between load() and check_node_locked() will surface here.	Writes time-anchor files on the hardware-pass path (inherited from check_expiry). Single-writer; see "Thread safety" below.
check_feature(name)	Pure features-array lookup. Does NOT re-check expiry, clock, or integrity.	None. If you need a fresh time check before calling check_feature(), call check_expiry() or check_node_locked() first and gate on its result. Safe to call concurrently from multiple threads after load().
check_version(current)	Pure version-range match against the license's version_range string. Does NOT re-check expiry, clock, or integrity.	None. Same pre-gate guidance as check_feature(). Safe to call concurrently from multiple threads after load().
is_loaded(), license()	Pure accessors; no checks of any kind.	None. Safe to call concurrently from multiple threads after load().

Practical pattern for a long-running application: call load() at startup, then re-call check_expiry() or check_node_locked() periodically (say, hourly) to catch clock manipulation or expiry that happened mid-session; in between, check_feature() and check_version() are cheap read-only paths.

Thread safety:

LicenseVerifier holds no internal mutex. The threading contract is:

- Construction and the first successful `load()` (or `load_from_string()`) call must complete before the verifier is observed by any other thread. The standard C++ publish-after-construction pattern -- e.g., construct + `load()` in `main()`, then hand the verifier to worker threads via a const reference; or store in a `std::shared_ptr` only after `load()` returns -- is sufficient.
- After `load()` returns, the truly read-only methods can be called concurrently from any number of threads on the same verifier instance:

```
is_loaded(), license(), check_feature(), check_version()
```

These methods only read impl-internal state that `load()` finalized; they do not mutate the ver...

- `check_expiry()` and `check_node_locked()` are NOT safe to call concurrently from multiple threads on the same verifier. `check_expiry()` performs filesystem / Windows-registry writes for the clock-manipulation time anchors (see Customer_Documentation §9.2 for the storage locations) and runs the binary integrity self-check; `check_node_locked()` tail-calls `check_expiry()` on the hardware-pass path (see §6.4), so it inherits the same side effects. Concurrent invocations of either method can race those side effects and produce false positives. Call them from a single thread, OR serialize calls through your own mutex, OR cache the first call's result if your application semantics allow it. (`load()` implicitly calls `check_expiry()` once internally; no extra call is needed at startup.)
- `load()` and `load_from_string()` must NOT race with anything else on the same verifier instance. They mutate `impl_->license`, `impl_->loaded`, and `impl_->license_path`. Calling `load()` while another thread is in any `check_*`() call or holds the result of `license()` is a data race and the result is undefined. If you need to swap the active license at runtime (e.g., the user pasted in a fresh one), gate it behind your own writer-side lock.
- Two distinct `LicenseVerifier` instances are independent and may be used concurrently from different threads with no synchronization between them. If you genuinely want concurrent `check_expiry()` from multiple threads, the simplest pattern is one verifier instance per thread.

Summary: load once, publish, then read freely from many threads -- but treat `check_expiry()` as single-writer because its side effects are persistent.

6.1 Constructor

```
explicit LicenseVerifier(const std::string& public_key_pem,  
                        SignatureAlgorithm algo = SignatureAlgorithm::AutoDetect)
```

Pass your public key as a PEM string (not a file path).

The default algo is `SignatureAlgorithm::AutoDetect` (v1.2.1+): the library inspects the loaded public key and picks the matching signature algorithm automatically -- Ed25519 keys are verified with Ed25519, RSA keys with RSA-SHA256. The customer no longer has to track which algorithm they used at keypair creation. Existing callers that pass an explicit Ed25519 / RSA_SHA256 value continue to work; the explicit value overrides auto-detection. (v1.2.0 callers passing the

previous default Ed25519 keep working unchanged; nothing breaks for them.)

If you want your binary to ACCEPT only one algorithm (e.g., a security-policy decision to reject RSA licenses signed by a stale RSA key after migrating to Ed25519), pass that algorithm explicitly. AutoDetect intentionally trusts the key; an explicit value asserts your expectation.

If the loaded key is neither Ed25519 nor RSA, the constructor throws `std::runtime_error` (the customer hit a key type the library does not support; this is the same surface as a malformed PEM and the existing recommendation to wrap construction in try/catch when loading from untrusted sources applies).

Exception behavior: the constructor throws `std::runtime_error` if `public_key_pem` cannot be parsed as a valid PEM-encoded public key. Once construction succeeds the verifier never throws -- every subsequent failure mode (bad license file, signature mismatch, hardware mismatch, etc.) is reported via `LicenseStatus` from `load()` / `check_node_locked()` / `check_expiry()`. The constructor is the only point in the verifier path that throws, and it does so only on malformed input you control (you embedded the public key string into your binary at build time; if you got it from a trusted build, it parses).

If your public key string is sourced from outside your build (e.g., loaded from a file or fetched at runtime), wrap construction in try/catch:

```
try {
    rockyguard::LicenseVerifier verifier(public_key_pem);
    auto result = verifier.load("license.json");
    // ... LicenseStatus-based handling continues from here
} catch (const std::runtime_error& e) {
    std::cerr << "Bad public key: " << e.what() << "\n";
    return 1;
}
```

For the standard pattern (PEM string is a `const char*` literal embedded in your application source), the throw cannot fire and the try/catch is unnecessary; we recommend it only for the loaded-from-untrusted-source case.

6.2 load()

```
LicenseResult load(const std::string& license_file_path)
```

Load and verify a license file. Checks signature, parses payload, verifies expiry, runs anti-tampering checks (clock + integrity).

Safe against malformed input: if the license file contains invalid JSON, wrong value types, or missing fields, returns `MalformedFile` without crashing. Never throws unhandled exceptions.

Prints a warning to `stderr` if `license_id` or `product` is empty in the license payload. These fields are required for clock manipulation detection to work correctly per-license.

IMPORTANT - synchronous network I/O:

`load()` invokes the same anti-tampering pipeline as `check_expiry()` (§6.6), which means it CAN perform a synchronous online time check via HTTPS. The online check fires:

- ALWAYS on first run (when no time anchors exist yet, the library has no local-only way to d...
- 10% of the time on subsequent runs (random; routine verification, low traffic on the host p...

Each online check tries up to 3 hosts from a 12-host TLS pool with a 3-second per-host socket timeout. Worst-case wall-clock latency is bounded by $3 * (\text{system DNS timeout} + 3 \text{ seconds})$. On a healthy network the typical latency is sub-second; on a flaky network or behind a captive portal, the call can block for tens of seconds. When all online attempts fail (offline), `load()` proceeds as if the online check were skipped (does not return an error for "offline"; see §6.6 for the full state matrix).

Recommendation for GUI / desktop applications: call `load()` from a background thread (`std::thread`, `std::async`, a worker pool, or your application's existing IO/runtime executor) and `join / .get()` before unblocking the UI on the result. Calling `load()` on the UI thread can freeze the application's first-run startup for tens of seconds when the time-anchor pool is unreachable.

```
// Recommended for any UI application:
auto fut = std::async(std::launch::async, [&]() {
    return verifier.load("license.json");
});
// ... show splash / continue UI work ...
LicenseResult r = fut.get();
```

Recommendation for server / daemon / CLI applications: calling `load()` on the main thread is fine; tens-of-seconds startup is acceptable for these hosts and avoids the complexity of threading. No special handling required.

Coming in v1.3: a non-blocking `load_async()` returning `std::future<LicenseResult>` with the same semantics, so GUI hosts do not have to roll their own async wrapper.

Returns: Valid, InGracePeriod, Expired, SignatureInvalid, MalformedFile, ClockManipulated, IntegrityCheckFailed.

6.3 load_from_string()

```
LicenseResult load_from_string(const std::string& license_json)
```

Same as `load()` but from a JSON string.

6.4 check_node_locked()

```
LicenseResult check_node_locked()
```

Verify the license matches this machine's hardware. Call after `load()`.

Internally this is a two-stage check: first the hardware fingerprint is matched against the license's `hardware_fingerprint` field; if (and only if) the hardware passes, the call tail-invokes `check_expiry()` to re-run the full anti-tampering pipeline (clock-manipulation detection, binary integrity self-check, expiry / grace-period evaluation -- see §6.6). The single

LicenseResult returned is whichever stage decided the outcome: a hardware failure short-circuits before check_expiry() runs, so the hardware-related statuses are returned directly; otherwise the result is whatever check_expiry() produced.

Returns the union of the two stages:

- Hardware stage: HardwareMismatch (also returned with the specific message "Node-locked license has no hardware fingerprint" when the license carries an empty hardware_fingerprint AND fingerprint_match_threshold is non-zero -- a node-locked license with no fingerprint is rejected by default as a safety against issuance bugs. The opt-in to bypass this safety is fingerprint_match_threshold == 0, which is the documented "intentionally not hardware-locked" path; see Customer_Documentation §4.2).
- Expiry / anti-tampering stage (only reached when the hardware passes): Valid, InGracePeriod, Expired, ClockManipulated, IntegrityCheckFailed.
- Pre-load guard: MalformedFile (with message "No license loaded") if check_node_locked() is called before a successful load(); see the "Order of operations" notes at the top of §6.

Because check_node_locked() reaches check_expiry() on the success path, it inherits check_expiry()'s side effects: it WRITES time-anchor files (§9.2) and is therefore subject to the same single-writer threading constraint as check_expiry() itself (see "Thread safety" earlier in §6). Treat check_node_locked() as a writing call when reasoning about concurrency, not a pure read.

6.5 check_feature()

```
LicenseResult check_feature(const std::string& feature_name)
```

Check if a feature is in the license. Call after load().

Returns: Valid, FeatureNotLicensed.

Example:

```
if (verifier.check_feature("export_pdf")) {  
    menu.enable_export();  
}
```

6.6 check_expiry()

```
LicenseResult check_expiry()
```

Explicitly check expiry. Also runs anti-tampering checks:

- Multi-location time anchor verification.
- Online time verification via HTTPS with TLS certificate timestamps (mandatory when stored a...
- Binary integrity self-check (DLL builds).

Online-check behavior, by combination of state:

State	Behavior
Anchors PRESENT, online	Drift > 1h => ClockManipulated; otherwise proceed to evaluate stored anchors (a roll-back beyond tolerance there is also ClockManipulated).
Anchors PRESENT, offline	Online check skipped silently; stored anchors still authoritative. Roll-back beyond tolerance => ClockManipulated.
Anchors MISSING, online	Online time fetched and validated; drift > 1h => ClockManipulated; else fresh anchors written and check returns Valid.
Anchors MISSING, offline	Fail-open by design: fresh anchors are written using the current system clock and check returns Valid (see note below).

The (Anchors MISSING, offline) fail-open is a deliberate trade-off so that a legitimate first run on an air-gapped or briefly-disconnected machine is not blocked. The narrow attack this exposes (rolled-back clock + all anchors deleted + permanent air-gap) requires three simultaneous adversarial conditions; any one of internet connectivity, anchor presence, or normal clock advancement closes it.

Returns: Valid, InGracePeriod, Expired, ClockManipulated, IntegrityCheckFailed. There is no ServerUnreachable status: a missing online check is treated as "no signal", never as a verification failure, so customers behind captive portals or transient network outages are not falsely rejected.

6.7 check_version() (v1.2.1+)

```
LicenseResult check_version(const std::string& current_version) const
```

Checks whether the running application's version satisfies the license's version_range field. Pass YOUR application's version as a dotted string ("3.1.5"); the comparison is component-wise and zero-pads shorter versions, so "3.0" and "3.0.0" compare equal.

Two range syntaxes are supported, and the matcher chooses based on the first non-whitespace character of the license's version_range:

Glob form (the form shown in the license_create --version examples):

```
" "      empty -- matches any version (default; preserves v1.2.0 behavior)
"*"      matches any version
"3.*"    matches versions whose first component is 3 (3.0, 3.1.5, 3.99.99...)
"3.1.*"  matches versions whose first two components are 3.1
"3.1.5"  exact match (component-wise, zero-padded)
```

Comparator form (when version_range starts with <, >, =, or !):

```
">=3.0"    matches versions >= 3.0
"<4.0"     matches versions strictly less than 4.0
">=3.0,<4.0" comma-separated AND -- both clauses must hold
"!3.5.0"   excludes an exact version
"=3.1.5", "=3.1.5" exact match (alias for the bare form)
Operators: < <= > >= = == !=
```

Pre-release suffixes ("1.2-rc1", "1.2+build5") are NOT supported in v1.2.1 and surface as MalformedFile. If you need them, drop the suffix at issuance time.

Empty current_version: if the license's version_range is empty or "*" (the open range), check_version() returns Valid without inspecting current_version. For any other range, an empty current_version is unparseable and returns

MalformedFile with message "current_version is not parseable: """. Guard against this in your build wiring -- if you derive APP_VERSION from a build-time substitution, assert it is non-empty before calling check_version() so a misconfigured build fails loud at startup rather than silently producing a MalformedFile under a non-empty range.

Returns:

```
Valid           current_version satisfies the range
VersionMismatch current_version does NOT satisfy the range
MalformedFile   current_version or the license's version_range is unparseable
```

Example:

```
auto result = verifier.check_version("3.2.1");
if (!result) {
    std::cerr << result.message << "\n";
    return 1;
}
```

A note on history: License::version_range existed in v1.2.0 but was informational only -- no shipped library code consulted it. v1.2.1 makes it actionable through this API. Licenses issued under v1.2.0 with informational version strings still load correctly under v1.2.1; if you do not call check_version(), behavior is unchanged.

6.8 license()

```
const License& license() const
```

Access the parsed license after successful load():

```
const auto& lic = verifier.license();
std::cout << "Licensed to: " << lic.licensee << "\n";
```

If called before load() has succeeded, returns a default-constructed License (every field empty / zero / NodeLocked). is_loaded() (§6.9) is the recommended pre-check if you are not sure whether load() has run yet.

6.9 is_loaded()

```
bool is_loaded() const
```

Returns true if a license has been successfully parsed via load() or load_from_string() and the verifier holds a usable License object. Returns false before the first load() call, or after a load() call that failed (returned MalformedFile or SignatureInvalid). Use this when the verifier is held by long-lived state and you need to confirm it is ready before calling check_node_locked(), check_feature(), check_version(), or check_expiry().

7. HARDWAREFINGERPRINT CLASS

Defined in: <rockyguard/hardware_fingerprint.h>

All methods are static.

7.1 HardwareComponents Struct

```
struct HardwareComponents {  
    std::string mac_address;  
    std::string cpu_id;  
    std::string disk_serial;  
    std::string motherboard_id;  
};
```

The four hardware fields the library hashes into a fingerprint. Empty strings indicate the value could not be read on this machine; the library handles that case gracefully (see `match_count()` below).

7.2 collect()

```
static HardwareComponents collect()
```

Collect hardware info from this machine. Reads MAC, CPU id, disk serial, and motherboard serial via OS-specific APIs (WMI on Windows, `/sys + /proc` on Linux). The library also contains an IOKit-based macOS implementation, available upon request as a separate build (see [Customer_Documentation §1, Supported platforms](#)). Returns a struct with one `std::string` per slot; any slot the OS could not provide is left empty.

7.3 fingerprint()

```
static std::string fingerprint()
```

Get the full fingerprint string for this machine: SHA-256 of each component, pipe-separated in fixed order (MAC | CPU | Disk | Motherboard). Equivalent to `compute_fingerprint(collect())`. This is the canonical form passed to `license_create's --fingerprint-value` flag.

7.4 compute_fingerprint()

```
static std::string compute_fingerprint(const HardwareComponents& hw)
```

Compute the fingerprint string from given components. Same output format as `fingerprint()`; use this when you have already called `collect()` and want to inspect the components or compute the fingerprint without re-querying the OS.

7.5 match_count()

```
static int match_count(const std::string& fp_a, const std::string& fp_b)
```

Compare two fingerprints, return matching component count (0-4). Unavailable components (empty hash) are skipped on either side - they don't count as matches or mismatches.

Edge case - all four components unavailable: if every slot on one side is empty (e.g., a sandbox with no MAC, no CPU

id, no disk serial, no motherboard serial), `match_count` returns 0. The verifier compares this against the license's `fingerprint_match_threshold` (default 2): with the default, `LicenseStatus::HardwareMismatch` is returned and the license is rejected (fail-closed). A vendor who deliberately issues a threshold-0 license has chosen "not hardware-locked" semantics, in which case an all-empty fingerprint is accepted by design.

Diagnostics: `compute_fingerprint()` emits one "[RockyGuard] WARNING: Hardware component '<name>' is unavailable. Fingerprint will be weaker." line per missing component on `stderr` at every collect, so an operator deploying on a stripped-down host sees the weakness at runtime. Important: this warning comes from the `LIBRARY` function, so it surfaces only when the host application links against `rockyguard.lib` / `librockyguard.a` (e.g., `license_create`, your own application, or any tool that calls `HardwareFingerprint::compute_fingerprint`). It does NOT surface from `rg_fingerprint`, which is a deliberately library-free standalone tool with its own platform fingerprint code (see [Customer_Documentation §7.1](#)) and therefore does not exercise the library's warning path. To diagnose hardware-component availability when `rg_fingerprint` output looks suspicious, run `license_create` (or any rockyguard-linked tool) on the same host and read its `stderr`.

8. FLOATINGLICENSECLIENT CLASS (PREMIUM)

Defined in: <rockyguard/floating_client.h>

Checks out floating licenses from a server.

Thread safety:

FloatingLicenseClient is internally synchronized. The class owns a background heartbeat thread that runs from a successful checkout() to the matching checkin() (or to destruction); checkout() and checkin() are serialized by an internal mutex. The concrete contract is:

- Construction must complete before any other thread observes the client. The constructor itself is not thread-safe (no reasonable C++ object's constructor is). Use the standard publish-after-construction pattern: construct in one thread, then hand the client to other threads via reference, pointer, or shared_ptr.
- After construction, checkout() and checkin() may be called from any thread. Both methods take an internal mutex for their full duration, so concurrent calls serialize rather than race. Two threads racing to checkout() will not cause a double checkout: the second caller observes the first call's result via the internal checked-out flag and returns {Valid, "Already checked out"} without contacting the server. Two threads racing to checkin() likewise: the second returns {Valid, "Not checked out"}. Note that because the mutex is held across the network round-trip (potentially several seconds on a slow link), a thread blocked on this mutex waits for the in-flight call to complete; do not assume checkout() and checkin() return promptly when called concurrently from contending threads.
- is_checked_out() reads an internal atomic flag directly and does NOT take the mutex. It is safe to call from any thread concurrent with anything else, including a checkout() or checkin() in flight. The returned value is advisory: between the call returning true and the caller acting on it, a different thread can complete a checkin() and flip the value. If your application needs a stable "license held for the duration of this scope" guarantee, hold that fact at your own application layer (a member flag protected by your code's own mutex, or a token returned from your own wrapper) -- the FloatingLicenseClient API exposes the lease state but does not lock it for caller-defined critical sections.
- The internal heartbeat thread reads only fields that are immutable after construction (config, client_id, machine_id, the TLS context) plus atomic flags (checked-out, heartbeat-running, client sequence). It does not race with caller-thread access to those fields, and you cannot interact with it directly -- it is started by checkout() and stopped by checkin() or the destructor.
- The destructor is NOT safe to run concurrently with any in-flight checkout(), checkin(), or is_checked_out() from another thread. This is the standard C++ object-lifetime rule: the owner must guarantee no other thread is using the client when it goes out of scope or is deleted. The destructor itself joins the heartbeat thread and (if still checked out) sends a final checkin() to release the seat.
- Two distinct FloatingLicenseClient instances are independent and may be used concurrently from any threads

with no synchronization between them. They obtain DIFFERENT `client_id` values but the SAME `machine_id` (see §8.2), so the server treats them as two independent leases originating from the same host. The operator's per-machine cap (Customer_Documentation §5.2; `max_leases_per_machine_id`) bounds how many concurrent leases one host may hold.

Practical pattern: a single `FloatingLicenseClient` instance per logical caller is the simplest model. If you genuinely need concurrent in-progress checkouts from many threads on one host (rare), the cleanest approach is one client instance per thread or per worker, letting each instance hold its own lease, and relying on the operator-side per-machine cap to bound aggregate usage. Sharing one client across threads is supported via the internal mutex, but the cost is that one thread's blocking `checkout()` serializes another thread's `checkin()` and vice versa.

8.1 FloatingClientConfig

```
struct FloatingClientConfig {
    std::string server_host      = "127.0.0.1";
    uint16_t    server_port     = 8080;
    int         heartbeat_interval_sec = 60;

    std::string server_public_key_pem; // Verify server responses
    bool        use_tls = false;       // Enable TLS encryption
    std::string tls_ca_cert_path;      // Server cert for TLS verification
};
```

The last three fields combine into layered defenses. They protect different things and are independent of each other.

`server_public_key_pem` (payload signing). Set to your server's public key PEM string. The client verifies every server response against this key, so even a successful transport-layer MITM cannot forge a checkout, heartbeat, or checkin response: an attacker without the server's private key cannot produce a valid signature and the client returns `SignatureInvalid`. **STRONGLY RECOMMENDED** for any production deployment.

`use_tls`. Enables TLS encryption of the wire traffic. TLS protects the session secret issued by the server at checkout (used for HMAC on subsequent requests) from passive network observers. Without TLS, that secret is transmitted in cleartext and a sniffer can capture it and forge later HMAC-authenticated heartbeats / checkins.

`tls_ca_cert_path`. Path to the server certificate (or its issuing CA) that the client uses to verify the TLS handshake. With this set, the client refuses to talk to anyone presenting a different cert. Empty path = TLS encrypts but does not verify the peer; an active MITM with any cert can intercept and decrypt.

Recommended configurations:

Configuration	Settings	When to use
Production	<code>use_tls = true</code> , <code>tls_ca_cert_path</code> set, <code>server_public_key_pem</code> set	All three layers active; no single defense failure exposes the deployment.
Internal / pinned	<code>use_tls = false</code> , <code>server_public_key_pem</code> set	Acceptable on a trusted network where payload authenticity is the primary concern and transport encryption is not required; the captured-session-secret risk above remains.

Configuration	Settings	When to use
Development only	use_tls = false, server_public_key_pem empty	No transport encryption AND no payload signature verification. An active MITM can serve fake checkouts. Do not ship this configuration to end users.

Note: these flags control the CLIENT side. The server-side counterparts (signing_private_key_pem, tls_cert_path, tls_key_path in FloatingServerConfig) must be configured to match.

8.2 Constructor

```
explicit FloatingLicenseClient(const FloatingClientConfig& config)
```

Constructs a floating-license client with the given configuration. Does NOT contact the server; the first network call happens at checkout().

At construction time, the client populates two identifiers used on every subsequent request to the server:

```
client_id    A fresh random UUID (cryptographically random via OpenSSL RAND_bytes), unique to ...  
  
machine_id   The host's hardware fingerprint, computed once at construction and cached for the...
```

8.3 checkout()

```
LicenseResult checkout()
```

Acquire a license. Starts background heartbeat thread. If server_public_key_pem is set, verifies the server's response signature (returns SignatureInvalid if fake server detected). Always receives a session secret from the server and uses it to HMAC-authenticate subsequent checkin / heartbeat requests; this is required regardless of whether TLS is enabled, because TLS secures the transport but does not authenticate the application-level client identity (a peer who learns another client's client_id could otherwise forge eviction or heartbeat requests over the same TLS channel). Without TLS, the session secret is sent in cleartext during the checkout response and a passive sniffer could capture it; enable TLS (and pin the server certificate via tls_ca_cert_path) to close that window.

Wire data sent on every checkout / heartbeat / checkin: client_id, machine_id (the hashed hardware fingerprint described in §8.2), a fresh random nonce, a monotonic sequence number for replay protection, and the session-secret-keyed HMAC. No raw hardware identifiers, license file content, or end-user identifiers leave the host.

Returns: Valid, NoLicensesAvailable, MachineSeatLimitReached, ServerUnreachable, SignatureInvalid, MalformedFile.

NoLicensesAvailable means the global pool is exhausted across all machines. MachineSeatLimitReached means this specific machine has hit the per-machine cap configured by the operator (max_leases_per_machine_id) even though the global pool may still have seats free for other machines; the user should close another session on the same machine before retrying.

8.4 checkin()

```
LicenseResult checkin()
```

Release the license back to the server's pool. Stops the background heartbeat thread. Safe to call multiple times; subsequent calls are no-ops.

Returns: Valid, ServerUnreachable.

8.5 `is_checked_out()`

```
bool is_checked_out() const
```

Returns true if a license is currently held (between successful checkout() and checkin() or destruction). Useful for UI gating: show "Licensed" only when this returns true.

8.6 `~FloatingLicenseClient()`

```
~FloatingLicenseClient()
```

Destructor. If a license is still checked out, automatically calls checkin() on the server to release the seat. Stops the heartbeat thread cleanly. No exceptions are propagated out of the destructor.

9. CLI TOOLS REFERENCE

The following binary tools are provided for license management.

9.1 rg_fingerprint

Print machine hardware fingerprint.

```
rg_fingerprint [-v] [-o file.txt]
```

End users run this on their own machine and send the output hash to you so you can issue a node-locked license bound to their hardware.

9.2 license_keygen

Generate Ed25519 or RSA keypair.

```
license_keygen --private private.pem --public public.pem
```

Run once, at the start of your project. Keep private.pem secret on a secure machine; ship the public.pem string embedded in your end-user application as PUBLIC_KEY.

9.3 license_create

Create signed end-user licenses (requires vendor license). Typical usage (vendor issues a node-locked license bound to a specific customer's hardware; customer runs rg_fingerprint on their machine and sends the hash to you):

```
license_create --vendor-license vendor_license.json \  
  --key private.pem --id "LIC-001" --product "App" \  
  --licensee "User" \  
  --fingerprint-value "311867...|ff9f0f...|877f34...|f3b059..." \  
  --expires "2027-12-31T23:59:59Z"
```

Fingerprint flag variants:

Flag	Behavior
--fingerprint-value <hash>	Bind to the explicit hash supplied by the customer (the typical vendor workflow shown above).
--fingerprint	Auto-detect THIS machine's fingerprint (used for self-licensing or testing; not what you want when issuing to a remote customer).
(omit both)	Issue an unbound / floating license (use --type floating).

9.4 license_verify

Verify and inspect a license file.

```
license_verify --key public.pem --license license.json
```

Useful for debugging: prints the parsed license payload and verification status without requiring an end-user application.

9.5 rg_floating_server

Run a floating license server (Premium, requires vendor license).

```
rg_floating_server server_config.yaml
```

The companion executable to FloatingLicenseClient on the customer side. See Customer_Documentation §5 for full server configuration.

9.6 rg_floating_client

Test floating license checkout/checkin against a running server.

```
rg_floating_client [host] [port] [public_key.pem] [--tls] [--ca-cert server.crt]  
rg_floating_client --help
```

A minimal CLI client useful for smoke-testing the floating server before integrating FloatingLicenseClient into your real application. Run with --help (or -h) to print the full argument list. On a connection failure (server unreachable) or invalid port, the client prints the usage block to stderr alongside the error message so the right invocation is one paste away.

See the main documentation for full CLI options and examples.

10. INTEGRATION EXAMPLES

Complete working examples are provided in the `examples/` folder with a `CMakeLists.txt` for building. See [Customer_Documentation §2.5 "Building and Running the Examples"](#) for the `cmake` commands; [Customer_Documentation §2.4](#) has the Quick Start code walkthrough.

10.1 Node-Locked End-User Application (`examples/node_locked_example.cpp`)

Note: the shipped `examples/node_locked_example.cpp` reads the public key from a file path passed on the command line ("`node_locked_example <public_key.pem> <license.json>`") so the same binary can be re-run against different keys during testing. The listing below shows the PRODUCTION pattern -- embed the key as a string constant per §3.3.B of [Customer_Documentation](#). Do not ship a binary that reads the public key from a runtime file; an attacker who can replace that file can issue their own licenses ([Customer_Documentation §12.1](#)).

```
#include <rockyguard/rockyguard.h>
#include <iostream>

static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----
MCoWBQYDK2VwAyEAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----END PUBLIC KEY-----)";

int main() {
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);

    auto result = verifier.load("license.json");
    if (!result) {
        std::cerr << "License error: " << result.message << "\n";
        return 1;
    }

    result = verifier.check_node_locked();
    if (result.status == rockyguard::LicenseStatus::InGracePeriod) {
        std::cerr << "WARNING: " << result.grace_days_remaining
            << " grace days remaining\n";
    } else if (!result) {
        std::cerr << result.message << "\n";
        return 1;
    }

    // OPTIONAL version gate (v1.2.1+). Skip the block below if
    // you do not want runtime version enforcement; version_range
    // is otherwise informational. See §6.7. Pass YOUR application's
    // version; empty version_range matches anything.
    static constexpr char APP_VERSION[] = "3.1.0";
    result = verifier.check_version(APP_VERSION);
    if (!result) {
        std::cerr << "Version not allowed: " << result.message << "\n";
        return 1;
    }
}
```

```
    if (verifier.check_feature("pro")) {
        std::cout << "Pro features enabled\n";
    }

    return 0;
}
```

10.2 Floating Client Application (examples/rg_floating_client.cpp)

See examples/rg_floating_client.cpp for a complete working example.

```
#include <rockyguard/rockyguard.h>
#include <iostream>

// PUBLIC_KEY is the server's signing public key, embedded as a
// string constant at file scope exactly as shown in §10.1. The
// floating server signs lease responses; the client uses this key
// to verify them. See Customer_Documentation §3.3.B for the
// embed-as-constant rationale.

int main() {
    rockyguard::FloatingClientConfig config;
    config.server_host = "license-server.internal";
    config.server_port = 8080;
    config.server_public_key_pem = PUBLIC_KEY; // Verify server
    // config.use_tls = true; // If server uses TLS
    // config.tls_ca_cert_path = "server.crt"; // Verify server cert

    rockyguard::FloatingLicenseClient client(config);
    auto result = client.checkout();
    if (!result) {
        std::cerr << result.message << "\n";
        return 1;
    }

    // ... application logic ...

    client.checkin();
    return 0;
}
```

10.3 Floating License Server

Use the provided server binary with a YAML config file:

```
rg_floating_server server_config.yaml
```

See tools/floating_server_config.yaml for a sample configuration.

The server uses a thread pool for connection handling. Key config fields:

```
thread_pool_size: 4           Worker threads (default: 4)
```

<code>client_timeout: 5</code>	Drop slow clients after N seconds (default: 5)
<code>private_key: private.pem</code>	Sign responses (prevents server spoofing)
<code># tls_cert: server.crt</code>	Optional: enable TLS encryption
<code># tls_key: server.key</code>	Optional: enable TLS encryption
<code>max_leases_per_machine_id: 0</code>	Per-machine seat cap (v1.2.1+; 0 = uncapped). Defends against ghost-checkout exhaustion. See <code>Customer_Documentation §5.2</code> .
<code>log_max_bytes: 104857600</code>	Rotate active log file at N bytes (v1.2.1+; default 100 MiB; <= 0 disables rotation).
<code>log_keep_count: 5</code>	Archives kept (server.log.1..N; v1.2.1+; default 5; capped at 100).