

RockyGuard Library - API Reference

Version 1.3.0

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

About this reference:

Describes the RockyGuard v1.3.0 C++ API. Two inline markers are used: "(v1.3+)" for APIs new in v1.3.0 and "(v1.2.1+)" for those new in v1.2.1; anything unmarked has been present since v1.2.0. Everything marked "(v1.3+)" ships in v1.3.0 -- it is not a preview or a forward-copied draft note. 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 v1.3.0 C++ API. Symbols marked "(v1.3+)" do not exist in v1.2.1 headers, and symbols marked "(v1.2.1+)" -- `LicenseVerifier::check_version()`, `SignatureAlgorithm::AutoDetect`, `LicenseStatus::VersionMismatch`, `LicenseStatus::MachineSeatLimitReached`, and the `FloatingServerConfig` fields `max_leases_per_machine_id`, `log_max_bytes`, `log_keep_count` -- do not exist in v1.2.0 headers. Referencing either against an older build produces "undeclared identifier" / "no member named" errors. 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 `Customer_API_Reference.pdf` inside the customer zip you actually received -- that copy was packaged with your binary and matches it. Prior releases' references are also frozen under `docs/archive/`. The website (rockyguard.dev) publishes only the latest release, so a search result can land you on a newer reference than your build.

One change in v1.3.0 is a modification rather than an addition, so it carries no "(v1.3+)" symbol marker: `LicenseVerifier::license()` now returns `License` by value instead of `const License&`. See "Thread safety" in the section 6 introduction.

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>
#include <optional>

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

int main() {
    // CONSOLE-APP SHAPE. load() and check_node_locked() are
    // SYNCHRONOUS and can block for ~9s or more (3 HTTPS time-check
    // hosts x 3s, plus DNS) when the network is offline or behind a
    // captive portal -- see §6.2. Blocking main() is fine here; in a
    // GUI application run this off the UI thread and show the window
    // first, or the app appears frozen on first launch. §6.4 covers
    // the same hazard for periodic re-checks.
    //
    // The constructor is the ONLY verifier call that throws, and it
    // is guarded here even though PUBLIC_KEY is a compiled-in
    // literal. A literal that was truncated or line-wrapped when it
    // was pasted compiles cleanly and throws at construction;
    // uncaught, that is std::terminate() on your end user's machine,
    // with no message and no usable exit code. §6.1 gives the full
    // reasoning. LicenseVerifier is neither copyable nor movable,
    // which is why this is optional + emplace() rather than an
    // assignment.
    std::optional<rockyguard::LicenseVerifier> maybe_verifier;
    try {
        maybe_verifier.emplace(PUBLIC_KEY);
    } catch (const std::runtime_error& e) {
        std::cerr << "License system error: " << e.what() << "\n";
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

    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 advapi32 shell32)
```

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 v1.4: 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. This was scoped for v1.3 and deferred: it changes the layout of the shipped package, which is cleaner to introduce alongside a cycle's other packaging work than as part of a close-out. v1.3 continues to ship the explicit link line above, and v1.4 will keep it working alongside the imported-target form. See [RockyGuard_v1_3_Plan.txt](#) section 3 for the deferral record.

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-manipulation detection is weaker without these). Emitted by load() / load_from_string().
- License creation: same fields empty when generating a license via the license_create CLI tool.
- Date parsing: a license issued_at or expires_at field is not valid ISO-8601. The library treats unparseable dates as Unix epoch (so any unparseable date is treated as expired); the warning makes the malformed input visible.
- Hardware fingerprint: any one of the four components (MAC, CPU, Disk, Motherboard) cannot be read on this machine. The fingerprint is still computed but is weaker.
- Integrity check (DLL builds): library binary path cannot be determined, binary cannot be read, or the integrity .sig file is not next to the DLL.
- 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 stderr-only logging).

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 v1.4 roadmap. It was scoped for v1.3 and deferred -- it adds a new shipped public header and touches every diagnostic site in the library, which was more new surface than a close-out cycle could absorb. 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

(NOTE ON ORDER: this table is grouped by TOPIC for reading, which is NOT the declaration order in types.h. Since this same section warns that deleting a mid-enum value would renumber later values and break anyone persisting numeric codes, do not infer numeric values from the sequence below -- read them from types.h, which is the only authority. NotYetValid in particular is declared between MalformedFile and FeatureNotLicensed in the header, not near the end where it appears here.)

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 -- unreachable by design, not by oversight. No shipped code path returns it in any release to date. A switch that handles every LicenseStatus should include this case (route it the same way as MalformedFile) to silence "unhandled enum" warnings. Why issued_at is recorded but NOT enforced: comparing it against the local clock would reject a legitimate licence on any machine whose clock runs a few minutes slow -- a paying customer locked out on day one, with an error that reads like a vendor mistake. Expiry carries the same skew exposure but fails in the safe direction (slightly too much access rather than none). There is also no adversary: the only party who can mint a future-dated licence is whoever holds the vendor private key, so the check would defend against nothing and fire only on issuance errors or wrong clocks. Enforcing it is really an activation-date FEATURE ("valid from 1 March"), which needs its own design for skew tolerance and interaction with grace periods. The enumerator is held open so that feature can land without renumbering the enum. It will not be removed for tidiness: it sits mid- enum, so deleting it would renumber every later value and break anyone who persists or transmits numeric status codes.

KeyMalformed	(v1.3+) <code>load_from_string()</code> was given a string that begins with the activation-key prefix "RGAK-" (case-insensitive) but whose body is unrecoverable: Base32 chars outside the alphabet, truncated body, or an unsupported format-version byte. The <code>.message</code> field names the precise failure mode. See §6.3.
KeyChecksumMismatch	(v1.3+) An activation key decoded as Base32 but its trailing CRC32 disagreed with the body. Almost always a single-character transcription error -- a flipped char in an email-pasted key. Distinct from <code>SignatureInvalid</code> (which fires on cryptographic failure): on this status the customer needs to recopy the key; on <code>SignatureInvalid</code> the key is tampered or wrong-product. See §6.3.

4.3 enum class SignatureAlgorithm

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

4.4 struct LicenseResult

Member	Type	Description
<code>status</code>	<code>LicenseStatus</code>	Result status (see default below)
<code>message</code>	<code>std::string</code>	Human-readable description
<code>grace_days_remaining</code>	<code>int</code>	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  
}
```

Default value of `status`. As of v1.3.1 `status` is default-initialised to a NON-PASSING value (currently `LicenseStatus::MalformedFile`). This is worth knowing because `LicenseStatus::Valid` is the first enumerator and therefore has the value 0: with no default, a value-initialised `LicenseResult` reported `Valid` and converted to `true` through operator `bool`, i.e. it failed `OPEN`. Every library entry point assigns a real status before returning, so the default is only ever observable in your own code -- typically a result declared up front and assigned on some branches but not all. Treat the specific default status as unspecified; rely only on the guarantee that it is not a passing one.

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.

5.8 validate_for_issue() (v1.3+)

```
bool validate_for_issue(std::string* error = nullptr) const
```

```
static constexpr int kMaxGraceDays = 36500;
```

Returns true if the license is fit to be signed and issued. Rejects an empty `license_id`, `product`, or `licensee`; a `grace_period_days` outside `[0, kMaxGraceDays]`; or an `expires_at` that is neither "permanent"/empty nor a parseable ISO 8601 date. On failure, `*error` (if non-null) receives a human-readable reason for the first failing check; on success it is cleared. `LicenseGenerator::generate_string()` and the `license_create` CLI call this and refuse to mint when it fails (v1.3 made issuance enforcing rather than warning-only). It does NOT run during verification -- already-signed licenses, including older ones with empty ids, still verify.

5.9 is_evaluation_mint() (v1.3.2+)

```
bool is_evaluation_mint() const

static constexpr int kEvaluationMintDays = 7;
```

Returns true if this license expires within kEvaluationMintDays of the current time -- which deliberately INCLUDES an expiry already in the past, since a pre-expired license is the normal way to exercise the expiry path. Returns false for "permanent", for an empty expires_at, and for an expires_at that does not parse.

This exists for the generation side and is unlikely to be interesting to a verifying application, but it sits on the public License struct so it is documented here. LicenseGenerator meters licenses that answer true against a separate counter, so the short-lived licenses an integrator mints while wiring RockyGuard up do not consume the vendor license's max_end_user_licenses budget. Verification never consults it: a short-dated license verifies and expires exactly like any other, and nothing about a license's runtime behaviour changes because it was classified this way at issuance.

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.

Note that as of v1.3 license() returns a License by value rather than a const reference. See "Thread safety" in the section 6 introduction for the change and its rationale; existing code using License l = verifier.license(); or const License& l = verifier.license(); is unaffected.

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.	On the hardware-pass path only: writes time-anchor and generation-counter state to disk/registry AND may block on a synchronous outbound HTTPS time check (both inherited from check_expiry). Takes the WRITER lock, so it serialises against all other verifier calls. The failing path short-circuits and is cheap -- the success path is the expensive one. See 6.4 and "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. license() returns a by-value snapshot (v1.3+).	None. Safe to call concurrently from multiple threads after load(), including concurrently with a 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 (v1.3+):

LicenseVerifier owns an internal `std::shared_mutex` that serializes writers against readers automatically. v1.2.1 documented this contract but left enforcement to the caller; v1.3 implements it inside the library. The classification used by the internal lock:

```
unique_lock (writers):
    load()
    load_from_string()
    check_expiry()          -- has time-anchor + integrity-check
                           side effects; concurrent invocations
                           would race the time-anchor file writes
                           if both ran simultaneously.
    check_node_locked()     -- writer because it transitively calls
                           check_expiry() on the success path
                           (§6.4). The alternative -- treating it
                           as a reader and letting it nest a
                           writer -- would deadlock std::shared_mutex,
                           which is not recursive.

shared_lock (readers):
    check_feature()
    check_version()
    license()
    is_loaded()
```

Practical consequences for caller-side code:

- Concurrent readers are now first-class. Multiple threads calling `check_feature()` / `check_version()` / `license()` / `is_loaded()` on the same verifier instance run concurrently with no caller-side coordination. Throughput is bounded only by the work done inside each method (a few hash lookups and string comparisons -- microseconds per call).
- A writer call blocks readers and vice versa, the standard `shared_mutex` contract. While `load()` (or `check_expiry()` / `check_node_locked()`) is running, concurrent reader calls block until it returns. Reader calls block subsequent writer calls only if the readers are still in flight when the writer arrives. No starvation in practice given how brief both sides are.
- Construction is still your responsibility. The mutex protects the verifier AFTER construction; the constructor itself runs once on one thread and the standard "publish after the constructor returns" pattern still applies. Constructing on thread A and immediately handing the verifier to thread B without a release/acquire pair is undefined behavior independent of the mutex.
- `license()` returns a `License` by value (changed in v1.3). Up to v1.2.1 it returned a `const License&` pointing into the verifier's internal state, and this section told you to copy the result before storing it across a reload. That guidance is obsolete: the copy is now made for you, while the internal reader lock is held, so what you get back is a consistent snapshot that no later `load()` can disturb. No caller-side discipline is required.

The v1.2.1 hazard, for anyone auditing older integration code: `License` holds seven strings, a vector and a ...

Source compatibility: both of the common forms still compile and are now both safe.

```
rockyguard::License snap = verifier.license();           // fine
const rockyguard::License& r = verifier.license();       // fine: the temporary's lifetime is extended
```

Only a non-const reference binding stops compiling, which is deliberate -- a compile error is a better signal than a silent behavior change:

```
auto& bad = verifier.license();                          // no longer compiles
```

Cost is one License copy per call. `license()` is a metadata accessor rather than a hot path, so this is not ...

- Two distinct `LicenseVerifier` instances are completely independent. No synchronization between them is needed.

Migration note for v1.2.x callers: code that previously wrapped `check_*`() in its own mutex still works correctly (the locks are independent). You can remove the caller-side mutex when you're sure all your call sites are running against v1.3+. If you must support a mixed v1.2 / v1.3 binary deployment for a transition period, keep the caller-side mutex -- it's redundant but harmless.

Summary: in v1.3+, load once, publish, then read freely from many threads. The library guarantees write-vs-read serialization.

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 -- the public key string you embedded into your binary at build time. "You control it" does not mean "it parses", though: see the guidance below on why the embedded-literal case still deserves a try/catch.

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;  
}
```

We recommend the try/catch for the embedded-literal case too. Earlier revisions of this section said the throw "cannot fire" when the PEM is a const char* literal compiled into your binary. That is too strong: what the compiler guarantees is that the string is present and unmodified at runtime, not that its contents parse as a key. A literal that was truncated, line-wrapped incorrectly, or pasted with a missing BEGIN/END armor line compiles cleanly and throws at construction -- and pasting public.pem into a source file is the most common way the standard pattern goes wrong, particularly when an AI assistant performs the paste (see `AI_INTEGRATION_GUIDE.md` section 4).

The cost of guarding is a few lines; the cost of not guarding is `std::terminate()` on the end user's machine, which surfaces as a hard crash with no actionable message. Note that `LicenseVerifier` is neither copyable nor movable, so if the verifier must outlive the try block, hold it in a `std::optional` and construct with `emplace()` rather than assigning:

```
#include <optional>  
  
std::optional<rockyguard::LicenseVerifier> maybe_verifier;  
try {  
    maybe_verifier.emplace(ROCKYGUARD_PUBLIC_KEY);  
} catch (const std::runtime_error& e) {  
    std::cerr << "License system error: " << e.what() << "\n";  
    return 1;    // see the host-type list below before copying this line  
}  
rockyguard::LicenseVerifier& verifier = *maybe_verifier;
```

This also avoids re-indenting an existing `main()` body into a try block.

How to refuse on failure depends on what you are integrating into. The rule is "do not proceed into licensed functionality", which is not the same as "call `exit()`":

- Console app / CLI: print to `stderr`, return non-zero from `main()`.
- GUI app: show a dialog carrying `result.message`, then quit the event loop. Exiting before the main window appears looks like a crash to the end user.
- Service / daemon: log and either stay up in a disabled state or exit deliberately. A bare `exit()` under a supervisor usually produces a restart loop that obscures the cause.
- Plugin / shared library: return a failure code to the host and disable your features. Never call `exit()` or `abort()` -- you have no `main()` to return from, and terminating takes down the host application and any unsaved user work with it.

A future release will remove the throw from this path entirely by adding a non-throwing factory that reports key-loading failure through `LicenseResult` (`LicenseStatus::KeyMalformed`) like every other verifier entry point. The throwing constructor will remain supported for source compatibility. Until then, the try/catch above is the supported approach.

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).

BOTH LICENSE FORMS ARE ACCEPTED (v1.3+). `load()` reads the file and hands the contents to the same code path as `load_from_string()` (§6.3), so it auto-detects which form it received. You do NOT need to read a .key file into a string yourself:

```
verifier.load("license.lic"); // JSON envelope - works
verifier.load("license.key"); // RGAK- activation key - also works
```

The file extension is irrelevant; detection is by content. Anything whose first non-whitespace characters are "RGAK" is decoded as an activation key, and anything else is parsed as the JSON envelope. Leading whitespace, the dash separators inside the key, and a trailing newline are all tolerated, so a key file written by `license_create --output-key --` or one a user saved from an email -- loads as-is. A key that is detected but does not decode returns `KeyMalformed` or `KeyChecksumMismatch` rather than `MalformedFile`, which is what distinguishes "corrupted activation key" from "not a license at all".

The same applies in reverse: `load_from_string()` accepts a JSON envelope just as readily as a key string. The two entry points differ only in where the bytes come from.

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 detect a rolled-back clock, so it consults the time-anchor host pool).
- 10% of the time on subsequent runs (random; routine verification, low traffic on the host pool).

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 v1.4: 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. Scoped for v1.3 and deferred, for two reasons worth knowing if you are working around it today: returning a `std::future` across a DLL boundary is sensitive to the consumer's C-runtime matching the library's, and `load()` holds an internal writer lock for the whole online check -- so a naive async wrapper relocates the blocking rather than removing it, and every concurrent reader would still stall. Wrapping `load()` in `std::async`

on your own thread has the same effect and the same caveat; keep any status polling off the critical path. See RockyGuard_v1_3_Plan.txt section 3.

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

6.3 load_from_string()

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

Loads a license from a string instead of a file path. Accepts either of two formats; the format is auto-detected by prefix.

(a) RAW JSON ENVELOPE. The same {"payload": "...", "signature": "..."} shape your library wrote to disk in v1.2.x. This path is unchanged from v1.2.0 -- code that already passes JSON envelopes continues to work without modification.

(b) ACTIVATION KEY (v1.3+). The same signed envelope encoded as a Base32 string with the literal "RGAK-" prefix and dash-separated 5-character body groups, e.g. RGAK-A4DGB-2P67Q-RXKMN-...-Z9V77. The decoder is tolerant of mixed case, embedded whitespace / line breaks, and stray dashes (whatever an email client may have added during transport).

Detection is by prefix: a case-insensitive "RGAK" at the start of the (leading-whitespace-stripped) input picks path (b); anything else falls through to path (a). The cryptographic guarantees are identical for both paths -- the activation-key form decodes back to the same JSON envelope that load() reads from a file, and the same Ed25519/RSA signature is the only thing that authenticates the license. The decoder adds a CRC32 trailer (4 bytes inside the decoded body) so that single-character transcription errors produce a distinct status code (KeyChecksumMismatch) BEFORE the signature verification step runs -- this is a customer-support aid, NOT a security primitive.

Possible LicenseStatus values specific to the activation-key path (in addition to all the statuses load() can return):

- KeyMalformed (v1.3+): the input starts with the RGAK prefix but the body contains characters outside the Base32 alphabet, is too short to decode, or uses an unsupported format version. The .message field names the precise failure.
- KeyChecksumMismatch (v1.3+): the body decoded successfully but its trailing CRC32 disagrees with the rest -- almost always a single-character transcription error. Distinguishable from SignatureInvalid, which fires when the cryptographic signature itself fails to verify.

Generating an activation key from your end is the vendor side: license_create --output-key <file> writes the key to disk; --output (the .lic file) and --output-key may be passed in the same run. The portal exposes both formats side-by-side after a license is minted.

The thread-safety contract on load_from_string() is identical to load(): both acquire the unique writer lock on the internal shared_mutex (see §6 "Thread safety"). load_from_string() incurs no file I/O, but on the activation-key path it performs Base32 decoding + CRC32 verification before delegating to the same internal pipeline load() uses; expect roughly the same latency in practice.

6.4 check_node_locked()

```
LicenseResult check_node_locked()
```

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

NOT A PASSIVE CHECK -- READ THIS BEFORE PROFILING OR CALLING IT ON A HOT PATH. The name suggests a read-only hardware hash comparison, and that is only the first half of what happens. When the hardware matches, this call goes on to do all of the following:

- **WRITES TO DISK** (and on Windows, the registry): the time-anchor and generation-counter state files are updated. `check_node_locked()` is a mutating call, not a query.
- **MAY PERFORM SYNCHRONOUS NETWORK I/O**: an outbound HTTPS time check can fire, blocking the calling thread for as long as that request takes. The conditions are listed under §6.2 "IMPORTANT - synchronous network I/O"; they apply here identically.
- **TAKES THE WRITER LOCK**, not the reader lock. It therefore serialises against every other verifier call, including the ones that look like pure reads. Two threads calling `check_node_locked()` concurrently do not run concurrently.

None of this is reachable when the hardware check fails: that path short-circuits and is genuinely cheap. So the expensive case is the **SUCCESS** case, which is the opposite of the usual intuition and makes a naive benchmark of the failing path badly misleading.

If you want only the hardware comparison -- no disk writes, no network, no writer lock -- there is currently no API for it; `check_node_locked()` is the whole pipeline or nothing. Tell us if you need one. The practical pattern today is to call it once at startup and cache the `LicenseResult`, rather than re-checking per operation.

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`.

DOES NOT RE-CHECK EXPIRY, THE CLOCK, OR BINARY INTEGRITY. This is a pure lookup in the features array of the License cached at `load()` time. It answers "was this feature licensed?", never "is the licence still valid right now?". The distinction does not matter in a tool that starts, checks, and exits. It matters a great deal in anything long-running.

The failure mode, concretely: a daemon or desktop application loads a licence on Monday and thereafter gates features

on `check_feature()` alone. The licence expires on Wednesday. `check_feature()` keeps returning Valid on Thursday, Friday, and for as long as the process stays up, because nothing has re-evaluated the expiry date. The same is true of a clock rollback or a failed integrity check -- those are detected by the anti-tampering pipeline, and `check_feature()` does not run it.

Re-validate on a schedule and gate on THAT result:

```
// At startup, and then periodically -- hourly is a reasonable
// default; align it with your tolerance for a stale licence.
auto status = verifier.check_expiry();    // or check_node_locked()
if (!status) {
    enter_unlicensed_mode(status.message);
    return;
}

// Only meaningful once the check above has passed recently.
if (verifier.check_feature("export_pdf")) {
    menu.enable_export();
}
```

`check_expiry()` (§6.6) is the cheaper of the two re-validation calls; `check_node_locked()` (§6.4) additionally re-confirms the hardware. Both write state and may perform network I/O, which is precisely why they are not folded into `check_feature()` -- doing so would turn every feature-gate lookup into a disk write and a potential blocking HTTPS request.

There is currently no built-in staleness policy: the library will not force a re-check for you, and it does not track how long ago the last one happened. Scheduling is the integrator's responsibility. Tell us if you would use an opt-in maximum-age setting.

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 anchors are missing; otherwise 10% random during normal operation).
- 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: < <= > >= == !=

Like check_feature() (§6.5), check_version() DOES NOT re-check expiry, the clock, or binary integrity -- it is a pure comparison against the version_range of the License cached at load() time. In a long-running process it will keep returning Valid indefinitely after the licence has expired. See §6.5 for the failure mode and the re-validation pattern; the same scheduling applies here.

Pre-release suffixes ("1.2-rc1", "1.2+build5") are NOT supported 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()`

```
License license() const
```

Access the parsed license after successful `load()`. Returns a snapshot BY VALUE as of v1.3 -- it returned `const License&` through v1.2.1; see "Thread safety" in the section 6 introduction for why that changed and what it means for existing code.

```
const License 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 to PARSE the license -- `MalformedFile` or `SignatureInvalid`. It returns TRUE after a `load()` that parsed a structurally valid, correctly signed license but returned a policy status such as `Expired` or `ClockManipulated`: the verifier does hold a usable `License` object in those cases, which is what lets you read `license()` and report the expiry date to the user. `is_loaded()` answers "is there a `License` object to inspect", NOT "is the license currently valid" -- the `check_*` methods answer the second question, and you must still call them. 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`.

7.6 VmInfo struct + detect_vm() (v1.3+)

```
struct VmInfo {  
    bool        detected = false;  
    std::string hypervisor;  
};  
  
VmInfo detect_vm();
```

Read-only inspection of CPUID and platform sysctls -- no network, no file I/O outside Linux's `/sys`, no privileged operations. Returns immediately (<1 ms in typical measurements).

VmInfo.detected is true when any of these signals is present:

- x86 CPUID leaf 1 ECX bit 31 (hypervisor-present) is set (Windows, Linux, x86 macOS).
- `/sys/hypervisor/type` is populated (Linux).
- `/sys/class/dmi/id/sys_vendor` matches a known hypervisor vendor string (Linux fallback, also catches AWS / GCP / Azure Hyper-V).
- `sysctl kern.hv_vmm_present` is non-zero (macOS, Apple Silicon).

`VmInfo.hypervisor` is a friendly name when known ("VMware", "Hyper-V", "KVM", "Xen", "VirtualBox", "Parallels", "QEMU", "bhyve", "ACRN", "AWS", "GCP", "Apple Hypervisor"). For unknown hypervisors that still expose the CPUID signature, the raw 12-byte signature string is returned so a customer can identify it from their support log.

The library does NOT use this value to gate verification. Licenses are valid in VMs by default. Customers who want anti-VM policy implement it themselves -- e.g., refuse to start the application when `detect_vm().detected` is true, or downgrade to a trial mode, or just log the event for compliance audit.

Known caveat on modern Windows: VBS (Virtualization-Based Security), HVCI (Hypervisor-Protected Code Integrity), Credential Guard, and the WSL 2 platform all run the Windows kernel inside a Hyper-V root partition on otherwise bare-metal hosts. CPUID cannot distinguish "guest of a hypervisor" from "host with VBS enabled" -- both report `detected=true` with `hypervisor="Hyper-V"`. Customers writing anti-VM policy on Windows should treat the Hyper-V signal as advisory and combine it with additional probes (e.g., the WMI `Win32_ComputerSystem.HypervisorPresent` + `Manufacturer` fields, or the `rg_fingerprint` Motherboard ID -- a real bare-metal motherboard reports a vendor like "Dell Inc." or "ASUSTeK" while a true Hyper-V guest reports "Microsoft Corporation") if they need to differentiate.

Example -- log VM context at startup:

```
rockyguard::VmInfo vm = rockyguard::detect_vm();  
if (vm.detected) {
```

```
std::cerr << "[my-app] starting in " << vm.hypervisor << " environment\n";  
}
```

Example -- anti-VM policy (vendor's choice):

```
if (rockyguard::detect_vm().detected) {  
    std::cerr << "[my-app] this product is licensed for physical machines only\n";  
    return 1;  
}
```

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

    // v1.3+
    int          max_heartbeat_failures = 10;
    FloatingHeartbeatCallbacks callbacks;
    bool         recheckout_on_lease_lost = false;
    bool         allow_insecure_tls      = false;
    bool         allow_insecure_http     = false;
};
```

server_public_key_pem, use_tls and tls_ca_cert_path combine into layered defenses. They protect different things and are independent of each other. allow_insecure_tls and allow_insecure_http are the two opt-outs that unlock the insecure combinations; both default false and both are enforced at construction (see §8.2).

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. As of v1.3 the default (false) does not stand on its own: leaving it false requires allow_insecure_http = true, or the constructor throws.

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.

allow_insecure_tls (v1.3+, default false). As of v1.3, use_tls = true with an empty tls_ca_cert_path is a hard error: the constructor throws rather than silently disable certificate verification. Set allow_insecure_tls = true to deliberately accept any certificate (TLS encrypts but does not verify the peer; an active MITM with any cert can intercept). Doing so prints a one-shot MITM warning to stderr. Use only in controlled test environments; set tls_ca_cert_path for anything else.

allow_insecure_http (v1.3+, default false). As of v1.3, use_tls = false is a hard error: the constructor throws rather than silently put license traffic on the wire in cleartext. Set allow_insecure_http = true to accept plaintext deliberately. Pre-v1.3 clients simply used HTTP with no diagnostic at all.

THESE TWO FLAGS ARE DIFFERENT AXES AND ARE EASY TO CONFUSE. allow_insecure_tls relaxes certificate VERIFICATION once TLS is already on; allow_insecure_http governs whether there is any encryption at all. Setting allow_insecure_tls does not permit cleartext, and setting allow_insecure_http has no effect once use_tls is true. The fail-closed rationale is the same for both: the checkout response carries session_secret, the HMAC key for every later request, so one sniffed or intercepted exchange gives an attacker full request-forgery capability.

Recommended configurations:

Configuration	Settings	When to use
Production	use_tls = true, tls_ca_cert_path set, server_public_key_pem set	All three layers active; no single defense failure exposes the deployment.
Internal / pinned	use_tls = false, allow_insecure_http = true, server_public_key_pem 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. The explicit opt-in is mandatory in v1.3+ -- without it the constructor throws.
Development only	use_tls = false, allow_insecure_http = true, 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().

THROWS `std::runtime_error` (v1.3+) on either insecure-transport configuration, before any network activity:

- `use_tls = false` and `allow_insecure_http = false`. The message names both the fix (`use_tls = true` with `tls_ca_cert_path`) and the opt-out, and warns that `allow_insecure_tls` is a different setting that does not permit cleartext.
- `use_tls = true`, `tls_ca_cert_path` empty, and `allow_insecure_tls = false`.

Both were silent downgrades before v1.3 (plain HTTP, and `SSL_VERIFY_NONE`, respectively). Failing at construction means a misconfiguration surfaces at integration time rather than as a quiet production exposure, but it also means code that compiled and ran against v1.2.x can throw on first construction after upgrading. See §8.1 for the two flags and the `Customer_Documentation` migration notes.

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 this `FloatingLicenseClient` instance. The server uses it to recognize repeated requests from the same in-process client (e.g., heartbeat refresh of an existing lease).

machine_id The host's hardware fingerprint, computed once at construction and cached for the session (no per-request platform syscalls). Same primitive used by the node-locked path: a `"|"`-joined string of four SHA-256 hashes (MAC address, CPU ID, disk serial, motherboard ID). The server only ever sees the hashes, never the raw hardware values. v1.2.1+ servers use `machine_id` to enforce `max_leases_per_machine_id`; v1.2.0 servers logged the field but did not act on it. Multiple concurrent `FloatingLicenseClient` instances on the same host produce different `client_id` values but the same `machine_id`, which is what lets the server bound per-host concurrency without collapsing legitimate concurrent sessions into one lease.

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.

8.7 Heartbeat lifecycle callbacks (v1.3+)

In v1.2.x the host application could observe lease state only by polling `is_checked_out()`. This collapsed three operationally distinct events into a single observable transition: transient connectivity loss, permanent server-side revocation, and long-term server outage all manifested only as `is_checked_out()` flipping to false. v1.3 adds an event-driven model via four `std::function` callbacks on `FloatingHeartbeatCallbacks` (a new struct that lives on `FloatingClientConfig.callbacks`):

```
struct FloatingHeartbeatCallbacks {  
    std::function<void()>          on_connection_lost;  
    std::function<void()>          on_connection_restored;  
};
```

```
std::function<void(std::string reason)> on_license_revoked;  
std::function<void()> on_give_up;  
};
```

Each callback is optional; unset (default-constructed empty `std::function`) callbacks are silently skipped. Wire only the events your host cares about.

`on_connection_lost`. Fires once when the first consecutive heartbeat fails with a transient error (network unreachable, 5xx, request timeout, connection refused). Does NOT fire again on subsequent consecutive failures within the same outage. Suggested host reaction: show a "connection issues, retrying..." indicator; pause non-essential network work; trigger an autosave.

`on_connection_restored`. Fires after a heartbeat succeeds following one or more transient failures. Symmetric counterpart to `on_connection_lost`: when this fires, the host can clear the connection-issues indicator. v1.3 collapses LM-X-style separate "retry" and "success" events into this single hook because for an HTTP-based heartbeat both occur in the same successful POST.

`on_license_revoked(reason)`. Fires when the server returns a permanent error (HTTP 403 / 404) OR when a heartbeat response fails signature verification (only when `server_public_key_pem` is set). The lease is forfeit; the client will need a fresh `checkout()` to recover. The reason string carries the server's response body when available, or a short diagnostic for signature failures (e.g. "Heartbeat response signature verification failed"). Suggested host reaction: initiate graceful shutdown of licensed features; surface the reason to the user.

`on_give_up`. Fires when consecutive transient failures reach the configured `max_heartbeat_failures` budget. The lease is forfeit (`checked_out` flips to false) and the heartbeat thread exits. Suggested host reaction: same as `on_license_revoked` plus offer the user a "retry" UI affordance that calls `checkout()` afresh.

8.7.1 max_heartbeat_failures

```
int max_heartbeat_failures = 10;    // default
```

Retry budget for the heartbeat loop. After this many CONSECUTIVE transient failures the client gives up, marks the lease lost, and fires `callbacks.on_give_up`. The default of 10 with the default `heartbeat_interval_sec` = 60 gives the server roughly 10 minutes of grace before the client treats it as gone.

A value of 0 means "retry forever" (the v1.2.x behavior, kept for backward compatibility). A finite value is strongly recommended in production -- without it a client whose server has been permanently shut down keeps opening TCP connections to the now-closed port indefinitely.

8.7.2 Threading model

Callbacks are invoked from the `FloatingLicenseClient`'s internal heartbeat thread, NOT from the thread that called `checkout()`. Two consequences:

- 1) Re-entrancy: callbacks MUST NOT invoke `checkout()` or `checkin()` on the same client instance. Both of those methods join the heartbeat thread (the thread the callback is running on), producing a deadlock where the heartbeat thread waits for itself to exit. `is_checked_out()` is safe -- it reads an atomic and does not touch the mutex. Use a flag / queue and call `checkout()` / `checkin()` from a different thread if state changes are needed in response to a callback.
- 2) Synchronization: the host is responsible for synchronizing access to any captured state from inside the callback. The heartbeat thread runs concurrently with the host's other threads. Callbacks should not block -- they delay the next heartbeat by however long they take. For slow work (logging, UI updates), capture state and dispatch to a dedicated worker thread / queue.

8.7.3 Mapping to LM-X callback functions

For readers familiar with X-Formation's LM-X library, the RockyGuard hooks correspond as follows:

LM-X callback	RockyGuard hook
HEARTBEAT_CONNECTION_LOST_FUNCTION	on_connection_lost
HEARTBEAT_RETRY_FEATURE_FUNCTION	(collapsed into restored)
HEARTBEAT_CHECKOUT_SUCCESS_FUNCTION	on_connection_restored
HEARTBEAT_CHECKOUT_FAILURE_FUNCTION	on_license_revoked
HEARTBEAT_EXIT_FUNCTION	on_give_up

LM-X's separate `RETRY_FEATURE` and `CHECKOUT_SUCCESS` events are collapsed in RockyGuard because in our HTTP-based protocol both happen in the same successful POST.

8.7.4 Server-restart auto-recovery (`recheckout_on_lease_lost`)

```
bool recheckout_on_lease_lost = false;
```

When a floating server restarts (planned reboot, container redeploy, crash + restart) its in-memory leases vector is wiped. Every active client's next heartbeat returns 404 ("lease not found"), which is indistinguishable on the wire from an admin revocation. The default v1.3 behavior treats this as a permanent error -- fires `on_license_revoked` and exits the heartbeat loop. The host application has to surface a "license revoked" message even though nothing was actually revoked; the end user reconnects manually.

When `recheckout_on_lease_lost` is true, a heartbeat 404 / 403 triggers an automatic `/checkout` against the same server BEFORE the `on_license_revoked` path runs:

- Re-checkout succeeds (server is back up and has free seats): the client transparently picks up a fresh lease with a new session secret and continues the heartbeat loop. The host application never observes the lease loss. If the host had previously seen `on_connection_lost` (e.g. the server was down for a while before restarting), `on_connection_restored` fires now to symmetrize. If the lease loss happened mid-flight with no preceding transient failures, no callback fires at all -- a silent recovery.
- Re-checkout fails (pool full because someone else grabbed the seat, server still unreachable, signature mismatch, ...): the original `on_license_revoked` path fires with a compound reason string of the form "Lease lost (<original>); auto-re-checkout failed: <recheckout failure>" so the host's log captures both why the original heartbeat failed and why the recovery attempt didn't succeed.

Defaults to false because some operators want a server restart to forcibly drop all leases (for compliance, audit, or pool-resizing reasons); default-on would defeat that. Customers running high-availability deployments where end users should not see a license blip on server reboot should set it true.

Caveats:

- The re-checkout uses the same `client_id` and `machine_id` as the original checkout. The new lease's `session_secret` is fresh (server-issued). The host doesn't see the secret change because it's internal to the client.
- A 403 caused by a per-machine seat limit (`max_leases_per_machine_id`) on the server hitting its cap will turn into "Lease lost ... auto-re-checkout failed: machine seat limit reached" -- the host can string-match the inner failure if it cares to distinguish.
- This does not retry across multiple servers. It only re-attempts the same `server_host` / `server_port`. Multi-server failover is a roadmap item for v1.4.

- Signature verification failures on heartbeat responses go straight to `on_license_revoked` without attempting re-checkout (the trust chain itself is broken; a fresh checkout would also fail the same way).

8.7.5 Example

```
rockyguard::FloatingClientConfig cfg;  
cfg.server_host = "license.example.com";  
cfg.server_port = 8443;  
cfg.use_tls      = true;  
cfg.tls_ca_cert_path = "server.crt";  
cfg.server_public_key_pem = server_pubkey;  
  
cfg.max_heartbeat_failures = 5;    // give up after 5 minutes  
cfg.recheckout_on_lease_lost = true; // survive server restarts  
  
std::atomic<bool> degraded{false};  
  
};  
  
};  
  
};  
  
cfg.callbacks.on_connection_lost = [&]() {  
    degraded = true;  
    // post a UI message via your event loop -- do NOT touch the client here  
cfg.callbacks.on_connection_restored = [&]() {  
    degraded = false;  
cfg.callbacks.on_license_revoked = [&](std::string reason) {  
    // log + initiate graceful shutdown of licensed features  
    log_error("License revoked: " + reason);  
cfg.callbacks.on_give_up = [&]() {  
    // permanent server outage: tell the user to check connectivity / call support  
  
};  
  
rockyguard::FloatingLicenseClient client(cfg);  
auto r = client.checkout();  
// ... use license ...  
client.checkin();
```

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>
#include <optional>

static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----END PUBLIC KEY-----)";

int main() {
    // Guarded construction, per §6.1. This listing is labelled the
    // PRODUCTION pattern, so it carries the guard that §6.1 and
    // Customer_Documentation §12.2 both require -- an embedded
    // literal is not exempt. Without it, a mis-pasted key is
    // std::terminate() on your end user's machine: no stdout, no
    // stderr, exit 0xC0000409 on Windows.
    std::optional<rockyguard::LicenseVerifier> maybe_verifier;
    try {
        maybe_verifier.emplace(PUBLIC_KEY);
    } catch (const std::runtime_error& e) {
        std::cerr << "License system error: " << e.what() << "\n";
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

    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

    // v1.3+: transport security is mandatory-by-default. Exactly one
    // of these two blocks must be present or the constructor throws.
    config.use_tls = true; // Encrypt the wire
    config.tls_ca_cert_path = "server.crt"; // Verify server cert

    // Local testing against a plaintext server instead? Drop the two
    // lines above and opt in explicitly. Never ship this:
    // config.allow_insecure_http = true;

    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)
client_timeout: 5	Drop slow clients after N seconds (default: 5)
private_key: private.pem	Sign responses (prevents server spoofing)
tls_cert: server.crt	TLS certificate (v1.3: see below)
tls_key: server.key	TLS private key (v1.3: see below)
allow_insecure_http: false	Accept cleartext deliberately (v1.3+; default false). AS OF v1.3 THE SERVER REFUSES TO START unless either <code>tls_cert</code> + <code>tls_key</code> are both set or this is true. It is the one key whose absence stops a cert-less server from starting at all, so a config copied from an older example will not run. Note the YAML keys are <code>tls_cert</code> and <code>tls_key</code> ; the <code>FloatingServerConfig</code> members are <code>tls_cert_path</code> and <code>tls_key_path</code> .
require_tls: false	Refuse to start without <code>tls_cert</code> / <code>tls_key</code> (v1.3+; default false). Belt-and-braces on top of the above: it closes the <code>allow_insecure_http</code> escape hatch.
rate_limit_rps: 50	Per-IP token-bucket refill rate, req/sec (v1.3+; default 50; 0 disables).
rate_limit_burst: 250	Per-IP burst capacity (v1.3+; default 250). Over-limit -> HTTP 429 (or drop under TLS). See Customer_Documentation §5.2 .
max_leases_per_machine_id: 0	Per-machine seat cap (v1.2.1+; 0 = uncapped). Defends against ghost-checkout exhaustion. See Customer_Documentation §5.2 .
log_max_bytes: 104857600	Rotate active log file at N bytes (v1.2.1+; default 100 MiB; <= 0 disables rotation).
log_keep_count: 5	Archives kept (server.log.1..N; v1.2.1+; default 5; capped at 100).

End of API Reference