Source: ccs_obfuscator docs/CCS_Obfuscator_Compendium_Draft_0.2.md @ 517afb4+ — curated 2026-08-07; edits land in the source first.

CCS Obfuscator Compendium

Draft 0.2 — 2026-08-07

(Draft 0.1, 2026-05-25, covered the project through step8.25. Draft 0.2 adds §6, "Grammar Recognition as a Security Primitive" — the conceptual frame the earlier draft left implicit — and renumbers the appendices to §7. No other section changed.)

A multi-audience reference for the `ccs_obfuscator` project: how to use it, how it works inside, what it does well, what it does badly, what's frankly ugly about it, where the algorithm line should go next, and what the security analyst should do about all of it.

This document covers the project as it stands at step8.25 — that is: algorithm 0 (no-op round-trip oracle, step8.22), algorithm 1 (XTEA on `edpfix_` / `edpdyn_`, step8.24), and the multi-grammar algorithm-1 round-trip oracle on the `cppcc/bench/samples` JSON corpus (step8.25). Items beyond algorithm 1 — additional ciphers, SCR-level renaming, walk-process transformations, the CCT Repository ciphering concept — are forward-looking and live as candidates in the relevant sections.

§6 is the conceptual frame for all of it, and a reader who wants the argument rather than the API should start there: why grammar recognition is a security primitive in its own right, what the complexity results do and do not establish, and why structural hardness and payload encryption are worth stacking rather than choosing between.

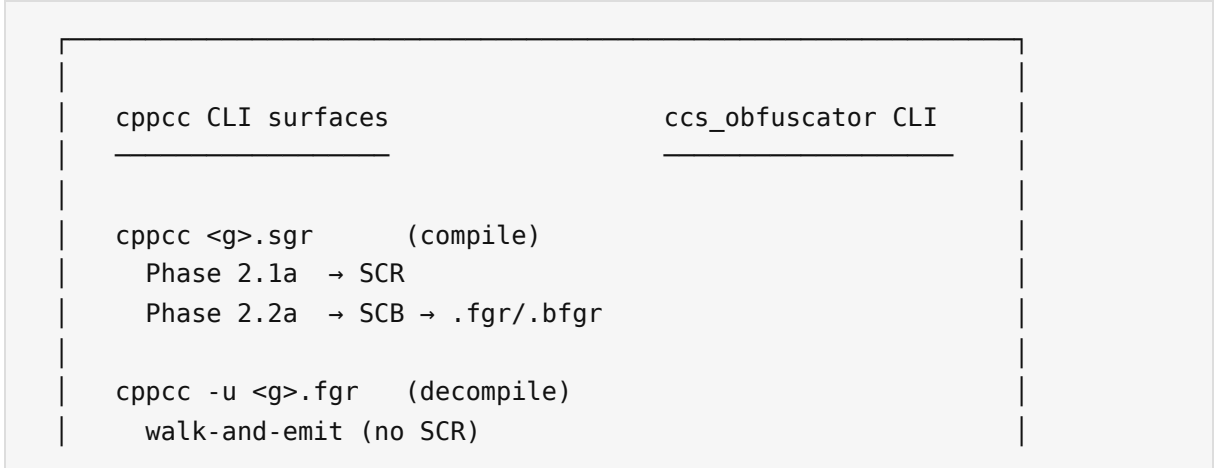
Table of contents

---

1. Programmer's Guide

1.1 What ccs_obfuscator is, where it sits

`ccs_obfuscator` is a standalone executable that exposes **Phase 2.2b** — *binary → runtime* — at the CLI level. Phase 2.2b is the inverse of Phase 2.2a (*runtime → binary*), and is the only one of the four Phase 2.* halves that the canonical `cppcc` binary does not surface on its command line. `cppcc` consumes binaries via `cppcc -u` (walk-and-emit using `BinaryHelper`, no runtime materialised), so the SCR-reconstruction path remained machinery-only inside `SyntaxControlledConverter::generateRuntimeFromBinary` until step8.22 completed its stub and `ccs_obfuscator` wired the CLI surface on top.



```
  │                                                          │
  │                                    obfuscator -a/-d ...  │
  │                                      .fgr/.bfgr          │
  │                                        Phase 2.2b → SCR  │
  │                                          transform       │
  │                                        Phase 2.2a → SCB  │
  │                                      .fgr/.bfgr          │
  │                                                          │
  └──────────────────────────────────────────────────────────┘
```

Once an SCR is in hand, *any* SCR ↔ SCR transformation becomes expressible. That is the deliberate framing — ciphering is the first inhabitant of that surface, but anything that can be written as "walk an SCR and produce another SCR" lives there too. This is why the project name is `ccs_obfuscator` (the near-term application) but its scope is more general (any runtime-level transformation).

1.2 Build and run

```
export CPPCCHOME=$VERSION_HOME/cppcc        # cppcc must be built first
export CCSJSONHOME=$VERSION_HOME/ccs_json   # for the JSON round-trip test
cd $VERSION_HOME/ccs_obfuscator/bin
make                                        # builds bin/obfuscator
```

`bin/Makefile` links against `$CPPCCHOME/lib/libruntime.a` only — no `libgenerate.a`, because the obfuscator never invokes the generator (no parser sources are emitted, no grammar is compiled). The double-link `libruntime.a libgenerate.a libruntime.a libgenerate.a` pattern from `cppcc/bin/makefile` is therefore unnecessary here.

CLI shape:

```
obfuscator -a <algo> [-k <seed>] <input>.fgr|.bfgr     # forward
obfuscator -d <algo> [-k <seed>] <input>.fgr|.bfgr     # inverse
```

Supported algorithms today:

```
  ┌──────┬──────────────────────────────────────────────────┐
  │ Algo │ Behavior                                         │
  ├──────┼──────────────────────────────────────────────────┤
  │  0   │ no-op forward (round-trip oracle; -k ignored)    │
```

```
   │  1   │ XTEA cipher on every edp's edpfix_ + edpdyn_ vectors   │
   └──────┴────────────────────────────────────────────────────────┘
```

Outputs (alongside `<input>`):

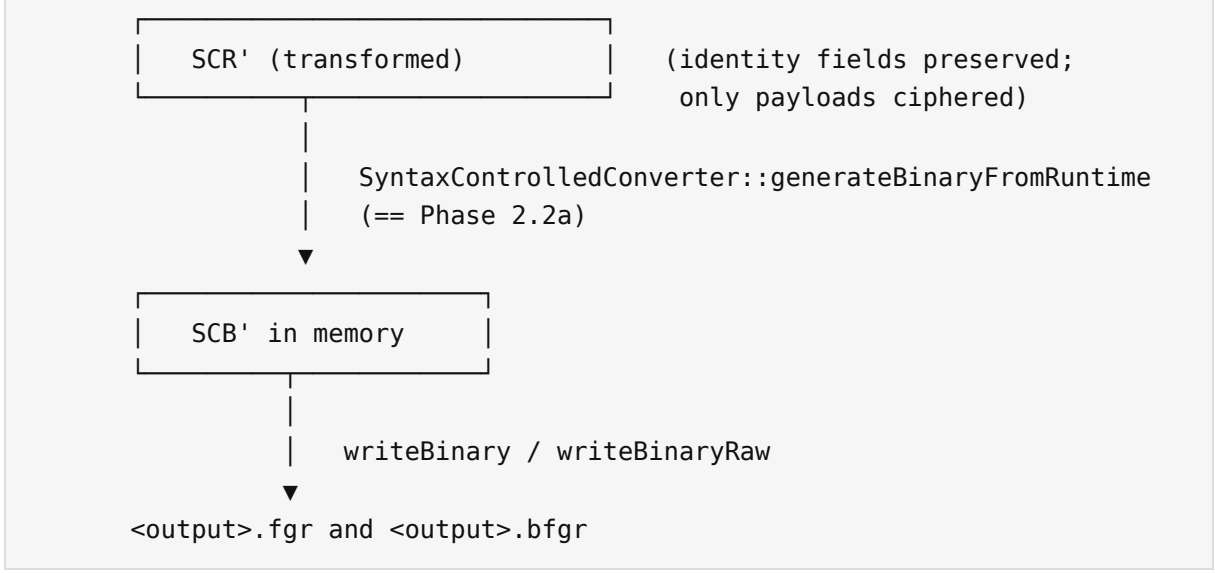
```
   forward (-a <algo>):
     <input-stripped>.obf.<algo>.{scbtxt,scrtxt,fgr,bfgr}
   inverse (-d <algo>):
     <input-stripped>.de.{scbtxt,scrtxt,fgr,bfgr}
```

Where `<input-stripped>` is `<input>` with the `.fgr` / `.bfgr` extension removed. The `.scbtxt` is the pre-transform memory dump (the loaded binary as decimal-text-per-word); the `.scrtxt` is the post-transform dump. For algorithm 0 they're byte-identical; for algorithm 1 they differ in `edpfix_` / `edpdyn_` regions.

`-k <seed>` is an arbitrary string passed through `ccs_obf::xtea::make_key` to derive a 128-bit key. If omitted, the hardcoded default seed `"ccs_obfuscator-default-key-step8.24"` is used (see §2.3 "Ugly" for why this default is a footgun).

1.3 Data flow (one diagram)

```
       <input>.fgr or <input>.bfgr
                │
                │   SyntaxControlledBinary::read{Binary,BinaryRaw}
                ▼
       ┌──────────────────────────┐
       │   SCB in memory      │   (binary_.memory_ : vector<tag::Long>)
       └──────────┬───────────────┘
                  │
                  │   SyntaxControlledConverter::generateRuntimeFromBinary
                  │   (== Phase 2.2b, completed in cppcc step8.22)
                  ▼
       ┌──────────────────────────────────┐
       │   SCR in memory                  │
(runtime_.cnms_[…].kwns_[…].edps_[…])
       └──────────┬───────────────────────┘
                  │
                  │   ccs_obf::cipher::encipher_runtime  (algo 1 forward)
                  │   ccs_obf::cipher::decipher_runtime  (algo 1 inverse)
                  │   ── walk every edp, transform its edpfix_/edpdyn_ ──
                  ▼
```



The cipher pass operates **at the SCR level, not at the binary level**. This is deliberate: editing `binary_.memory_` directly would break offset relationships inside the SCB encoding and require special-casing every record type. Operating at the SCR level lets the cipher pass treat the rule-body payloads as opaque `std::vector<tag::Long>` and rely on `generateBinaryFromRuntime` to re-encode them into a valid SCB on the way out.

1.4 `cipher.h` API

`include/cipher.h` lives in the `ccs_obf::cipher` namespace. Two layers of functions:

Per-vector primitives — operate on a single rule-body vector:

```
void encipher_taglong_vector(std::vector<cppcc::scr::tag::Long>& v,
                             const xtea::Key& key);
void decipher_taglong_vector(std::vector<cppcc::scr::tag::Long>& v,
                             const xtea::Key& key);
```

Each element of `v` is treated as one XTEA block (one `std::uint64_t`-equivalent). The signed `tag::Long` ↔ unsigned `std::uint64_t` adaptation goes through `std::memcpy`, which is the only strict-aliasing-safe way to reinterpret bits between two distinct trivial types under aggressive optimizer assumptions.

Per-runtime drivers — walk the SCR and call the primitives on every edp's `edpfix_` and `edpdyn_`:

```
void encipher_runtime(cppcc::scr::SyntaxControlledRuntime& runtime,
                      const xtea::Key& key);
void decipher_runtime(cppcc::scr::SyntaxControlledRuntime& runtime,
                      const xtea::Key& key);
```

Walk shape: `runtime.cnms_.dataByKey(runtime.cnmnumCurrent_)` → for each `kwn` in that context's `kwns_.index2data` → for each `edp` in that kwn's `edps_.index2data` → cipher both `edp.edpfix_` and `edp.edpdyn_` in place. The identity fields (`cnmnum_`, `kwnnum_`, `edpnum_`) are deliberately left untouched so the inverse pass can navigate the same structure.

Structural-validity oracle — used by the algorithm-1 gtest to assert "decompile-quality drops after ciphering":

```
struct EdpRefStats { std::size_t total = 0; std::size_t valid = 0; };
EdpRefStats validate_edp_refs(
    const cppcc::scr::SyntaxControlledRuntime& runtime);
```

For each `tag::Long` in every `edpfix_`/`edpdyn_`, decode it via `setlongedf` into a `(kwn, edpnum)` pair and check whether both indices land inside the runtime's existing kwn/edp ranges. Returns the total and valid counts. The baseline ratio on an un-ciphered runtime is not 100% — `edpfix_[0]` for `STRING_TOKEN` edps holds raw string bytes, not tag pairs, so the oracle counts those as "invalid" too. The useful signal is the *ratio drop* after ciphering: ciphered runtimes have valid counts statistically near zero, which is what the test asserts. This is the same oracle that R4 in the security TODO (§5.2) is designed to reuse for "verify a `.bfgr` was ciphered with key K" via try-decipher-then-check.

A compile-time gate enforces 64-bit `tag::Long`:

```
static_assert(sizeof(cppcc::scr::tag::Long) == sizeof(std::uint64_t),
              "ccs_obf::cipher: requires 64-bit tag::Long");
```

The 32-bit cppcc build (`#define _CPPCC_TAG_32BITS_`) shrinks `tag::Long` to 32 bits, at which point each element is half an XTEA block. Algorithm 1 needs a different block primitive on the 32-bit build — out of scope for step8.24 but a real port target if it ever lands.

1.5 `xtea.h` API

`include/xtea.h` lives in the `ccs_obf::xtea` namespace. It is a from-scratch C++ implementation of David Wheeler and Roger Needham's 1997 XTEA algorithm (public domain; see Appendix B).

```
using Key = std::array<std::uint32_t, 4>;                  // 128-bit

constexpr unsigned int kDefaultRounds = 32;
constexpr std::uint32_t kDelta        = 0x9E3779B9u;       // golden ratio

void encipher_block(std::uint64_t& block, const Key& key,
                    unsigned int rounds = kDefaultRounds);
void decipher_block(std::uint64_t& block, const Key& key,
                    unsigned int rounds = kDefaultRounds);

void encipher_vector(std::vector<std::uint64_t>& v, const Key& key,
                     unsigned int rounds = kDefaultRounds);
void decipher_vector(std::vector<std::uint64_t>& v, const Key& key,
                     unsigned int rounds = kDefaultRounds);

Key  make_key(const std::string& seed);
```

Key facts:

- **Block size**: 64 bits, two `std::uint32_t` halves split via `memcpy` to avoid strict-aliasing UB.
- **Key size**: 128 bits, four `std::uint32_t` words.
- **Rounds**: 32 by default — the Wheeler/Needham recommendation; each iteration does two Feistel half-rounds (one updating v0, one updating v1) for 64 half-rounds total.
- **Mode**: ECB-equivalent at the vector level — each block is ciphered independently. This is *fine for obfuscation* (the threat model is "make casual inspection hard") but *not fine for confidentiality* (repeated plaintext blocks produce repeated ciphertext blocks; see §2.2 "Bad").
- **Round-trip property**: `decipher_X(encipher_X(v, k), k) == v` holds at every level (block, vector, runtime) for any inputs — this is the property the round-trip oracles exercise.
- **`make_key`** is *not* a cryptographic KDF — it XORs seed bytes round-robin into 16 key bytes. Trivial to collide on purpose. Fine for "I want different obfuscation under different seeds"; *not* fine for "this protects a secret" (see §5.1).
- **Endianness**: round-trip is portable (encipher and decipher use the same packing); but ciphertext bytes differ between big-endian and little-endian hosts. Don't exchange

`.obf.1.bfgr` files across host endianness.

1.6 Test layers

Two complementary gtest harnesses:

```
┌──────────────────────────────────────┬──────────────────────────────────────┐
   │ Suite                                │ What it covers
│
├──────────────────────────────────────┼──────────────────────────────────────┤
   │ tests/test_algo1/                    │ cipher.h API correctness on the
│
   │                                      │   meta.sgr seed grammar. 5
cases:│
   │                                      │   identity, scramble, shape,
│
   │                                      │   wrong-key, structural-validity
│
├──────────────────────────────────────┼──────────────────────────────────────┤
   │ tests/test_algo1_json_round_trip/    │ obfuscator CLI byte-identity
│
   │                                      │   round-trip on the
bench/samples│
   │                                      │   JSON corpus. 6 cases:
│
   │                                      │   3 samples × 2 formats
│
   │                                      │   (twitter/citm/canada × fgr/
│
   │                                      │   bfgr). Each case: cipher then
│
   │                                      │   decipher via subprocess,
│
   │                                      │   diff bytes.
│
└──────────────────────────────────────┴──────────────────────────────────────┘
```

Why keep both: failures localise differently. A `test_algo1` failure points at the API layer (a bug in `cipher.h` or `xtea.h`); a `test_algo1_json_round_trip` failure points at the CLI driver / env-var setup / on-disk-state — *not* at cipher correctness, since `test_algo1` already proves that on the small seed grammar.

Run them via the per-suite scripts:

```
cd $VERSION_HOME/ccs_obfuscator/tests/test_algo1
bash ./test_algo1_script

cd $VERSION_HOME/ccs_obfuscator/tests/test_algo1_json_round_trip
bash ./test_algo1_json_round_trip_script
```

Each script pre-stages inputs, builds the gtest, runs it, and reports per-case timings. Both currently 100% PASS.

1.7 Adding a new algorithm — cookbook

For an algorithm `N` (where `N > 1`) that follows the in-place SCR transform pattern of algorithm 1:

1. Add a new header `include/<algo_name>.h` in namespace `ccs_obf::<algo_name>` exposing: - `transform_runtime(SCR&, const Params&)` — forward - `untransform_runtime(SCR&, const Params&)` — inverse - whatever `Params` shape the algorithm needs (key, mode, permutation table, etc.)
2. `#include` it from `src/obfuscator.cc`.
3. Bump the `algo > 1` guard in `obfuscator.cc::main` to `algo > N`.
4. Add a `case N:` to the `switch (algo)` dispatch that calls `transform_runtime` on forward, `untransform_runtime` on inverse.
5. Extend the algorithm table in `printUsage`.
6. Add a `tests/test_algo<N>/` suite mirroring `tests/test_algo1/` — at minimum the 5 API tests on `meta.sgr`. Add a `tests/test_algo<N>_json_round_trip/` suite if the algorithm preserves byte-identity under round-trip.
7. Extend `.gitignore` with the symmetric `tests/test_algo<N>/` + `tests/test_algo<N>_json_round_trip/` artefact blocks.

For algorithms that *don't* preserve the in-place pattern (e.g. algorithms that produce a *new* SCR rather than mutating the input — see §4.1), the cookbook above mostly applies except step 6's

"test_algo_json_round_trip" expectation changes from byte-identity to whatever weaker property the algorithm actually guarantees.

---

2. The Good, the Bad, the Ugly

A frank assessment of `ccs_obfuscator` as it stands at step8.25. Section 5 ("Security Analyst's TODO Items") converts the Bad and the Ugly into actionable work.

2.1 The Good

Round-trip is mathematically guaranteed. XTEA is a bijection on its 64-bit block domain for any fixed key. The vector / runtime drivers are pointwise applications of that bijection, so they inherit the bijection property. The round-trip oracle is not a heuristic empirical check — it's a structural property that *has to hold* given the algebra; the tests exercise it on real inputs but they couldn't fail on a correctly-implemented XTEA.

Identity fields are preserved. The cipher walk touches only `edpfix_` and `edpdyn_`. The `cnmnum_` / `kwnnum_` / `edpnum_` identity fields, the container shapes (sizes of `cnms_` / `kwns_` / `edps_`), and the kwn-name strings are all untouched. This is what lets the inverse pass *find* the same edps in the same positions — a cipher pass that scrambled identity fields would need to ship a separate "where did everything move to" inverse map.

Substrate-then-validate-at-scale discipline. Algorithm 1 landed in step8.24 with a 5-case API test on the small `meta.sgr` seed grammar. step8.25 immediately followed with a 6-case CLI test on the multi-MB `bench/samples` JSON corpus. The pattern — every substrate addition gets a scale-validation follow-on within one sub-step — makes "this scales" empirically testable rather than asserted.

XTEA is simple, fast, public-domain, easy to audit. The implementation is 30 lines of cipher loop and a memcpy adapter. Anyone with a printout of the Wheeler/Needham 1997 paper can verify it byte-for-byte. The from-scratch implementation lives in `xtea.h` with no external dependencies — no `OpenSSL` linkage, no version-skew exposure. (XTEA's *security* properties are a separate question; see "Bad".)

Test layering catches bugs at the right level. `test_algo1` catches API bugs; `test_algo1_json_round_trip` catches CLI / env / stale-binary bugs. The step8.23 "downstream-binary-stale-against- upstream-substrate" gotcha (a stale `ccsjson` binary against a new

`libruntime`) would have shown up here as a JSON-round-trip-fails / API-still-passes split, pointing straight at "the binary you're invoking is out of date, the cipher itself is fine".

`validate_edp_refs` is a reusable structural oracle. It was written for the algorithm-1 "ciphered SCR has degraded structural validity" test, but it's a general-purpose "does this SCR's rule-body content reference valid (kwn, edp) pairs inside this runtime's structure?" walker. R4 in the security TODO (§5.2.4) reuses it as the cheap-and-correct "key-match oracle" — try decipher under candidate key K, check whether structural validity recovers; if yes, K was the cipher key.

Phase 2.2b is finally surfaced. Until step8.22 this phase existed in the library but had no caller. The substrate fix + the obfuscator CLI together established the surface; any future tool that wants to operate on an SCR materialised from a `.fgr`/`.bfgr` (linters, statistics, transformers, visualisers) inherits the same machinery — `ccs_obfuscator` is the first inhabitant of a more general "CCS-binary transformation tool" surface.

2.2 The Bad

XTEA is theoretically broken. Differential cryptanalysis against reduced-round XTEA is publishable; full-round XTEA is considered weak by modern standards. For *obfuscation* this is fine; the threat model is "make casual inspection and hand-editing harder", not "resist a motivated cryptanalyst with chosen-plaintext access". If the cipher needs to resist a real adversary, replace XTEA with ChaCha20 or AES (see §3.2, §3.3).

ECB-mode equivalent leaks repeated-block structure. Each `edpfix_[i]` is ciphered independently of every other position and every other edp. If two edps share a common rule body prefix (e.g. both start with `(KW_OPEN_PAREN, 0)`), the ciphertext at those positions will be identical too. A statistical attacker can use this to recover structure even without recovering plaintext — they can identify "which edps share a prefix" from the ciphertext alone. Not a problem for "casual inspection deterrence"; a problem for "defeats motivated analysis".

`make_key` is not a cryptographic KDF. The seed-byte round-robin XOR collapses any seed longer than 16 bytes into a 16-byte space, with trivially-constructable collisions: `"AAAAAAAAAAAAAAAA"` and `"\x00\x00...\x00"` (with the right XOR-cancelling pattern) produce the same key. PBKDF2 / Argon2 solve this; the current `make_key` is a one-page placeholder that should be replaced before claiming any cryptographic-strength key derivation.

Algorithm 1 leaves structural shape intact. Even after ciphering, an attacker can read: - The number of `kwns_` and the size of each `kwn.edps_`. - The length of each `edp.edpfix_` and

`edp.edpdyn_`. - The kwn-name strings (`kwn.kwnkw_.name_`). - The grammar's axiom, predefined symbol set, and overall shape.

This is what "obfuscation, not encryption" actually means in practice. A determined reverse-engineer can reconstruct the *structure* of the grammar without breaking XTEA at all — they just can't read the rule-body content.

No version negotiation in output filenames. The output naming is `.obf.<algo>.{...}` — there's no algorithm-version suffix. If algorithm 1's internals change (rounds count, key schedule, anything), old `.obf.1.*` files become silently incompatible. A second-version algorithm-1 would need either a new algorithm number or an explicit version subfield in the filename. The current scheme is fine for one-version-per- algorithm but doesn't scale.

32-bit build unsupported. The `static_assert` in `cipher.h` fails the build outright if `tag::Long` is 32 bits. The 32-bit cppcc build is a real configuration (see `_CPPCC_TAG_32BITS_` in `cppcc/include/SyntaxControlledRuntime.h`); for parity, a half-block primitive (or a paired-elements adapter) would be needed. Out of scope until someone runs into it.

2.3 The Ugly

`kDefaultKeySeed` is a footgun. The default key seed `"ccs_obfuscator-default-key-step8.24"` is hardcoded in `src/obfuscator.cc:52`. Anyone who runs `obfuscator -a 1 file.bfgr` without `-k` gets reproducible ciphertext under a globally-known key. This is convenient for testing, *terrible* for the "obfuscation" use case it's named after. The right fix is to require `-k <seed>` for algorithm 1 (or any algorithm > 0) and error out if it's missing. Recommended in §5.1.3.

`.scbtxt` / `.scrtxt` debug artefacts ship on every run. Both files are decimal-text dumps of `binary_.memory_` — useful when developing the obfuscator, mostly noise once an algorithm is past development. Every invocation writes them to disk regardless. They also leak pre-transform state on disk to anyone with filesystem access, which is contrary to the obfuscation goal. Recommended: hide behind a `--debug-dumps` flag, default off.

`validate_edp_refs` takes `const SCR&` then `const_cast`s it. The reason is that `dataByKey` isn't a const method on the underlying container, even though `validate_edp_refs` genuinely doesn't mutate. The smell is real and the type signature is misleading — a future maintainer who sees `const SCR&` and assumes the function is thread-safe-against- concurrent-writes will be surprised.

Fix: either propagate the const correctness through `dataByKey` (probably the right upstream change) or rename the parameter to make the const-stripping explicit.

`edpfix_[0]` for STRING_TOKEN edps holds raw bytes, not tag pairs. This was discovered the hard way in step8.24 when the structural-validity oracle returned `158/377 = 41%` on an un-ciphered runtime instead of the expected `100%`. The test was reframed as a *ratio drop* assertion rather than a boundary, which is correct, but the underlying surprise — that `edpfix_[0]` is overloaded between "tag pair" and "raw string content" depending on the kwn's `kwd_` — is documented only in the step8.24.md commentary. A grep-and-find maintainer debugging a related issue could easily miss it.

No `-h` / `--help`. The CLI only prints usage on error, not on request. `obfuscator -h` is treated as `-a` with a bad algo arg and exits 2 with the usage banner — which technically works, but isn't the expected POSIX behavior. Trivial fix.

Empty `build/`, `incloc/`, `lib/`, `srcloc/` directories. The project's layout was copied from the sibling `ccs_json` / `ccs_config` skeleton, which has SGDL inputs and needs all of these. `ccs_obfuscator` doesn't compile any grammar, so those four directories are permanently empty. They're harmless (`.gitkeep` placeholders only) but tell a future reader "this project compiles a grammar" — which is exactly backwards from what it actually does.

---

3. Candidate Algorithms 2, 3, 4+

Algorithm 1 (XTEA on `edpfix_` / `edpdyn_`) is the only cipher algorithm currently wired. This section sketches plausible algorithm-2-through-7 candidates with the design surface for each. Numbers are suggestive, not committed — the actual allocation depends on which the security analyst (or roadmap) prioritises.

The candidates split into two families:

```
┌──────────────────────┬──────────────────────────────────────────────┐
│ Family               │ What it varies                               │
├──────────────────────┼──────────────────────────────────────────────┤
│ Cipher-strengthening │ Same target (edpfix_/edpdyn_), different     │
│   (Algo 2, 3, 4)     │   primitive: CTR mode, ChaCha20, AES.        │
│                      │   Addresses the §2.2 "ECB leak" and          │
│                      │   "XTEA broken" criticisms.                  │
├──────────────────────┼──────────────────────────────────────────────┤
```

```
| Target-broadening    | Same primitive family, different target: |
|   (Algo 5, 6, 7)     |   identifier renaming, ID permutation,   |
|                      |   composite. Addresses the §2.2          |
|                      |   "structural shape leak" criticism.     |
└──────────────────────┴──────────────────────────────────────────┘
```

3.1 Algorithm 2 — XTEA in CTR mode

Idea. Keep XTEA as the primitive but operate in counter (CTR) mode rather than ECB. For each `edp` (or for each `tag::Long` position globally), derive a unique counter from `(cnmnum_, kwnnum_, edpnum_, position_within_edp)` and cipher *counter* under the key, then XOR the result with the plaintext.

What it fixes. The ECB-mode repeated-block leak (§2.2). Identical plaintext at different positions produces different ciphertext because each position has a unique counter.

Cost. One extra XTEA invocation per block (cipher the counter, then XOR), so ~2× slower than algorithm 1. Still much faster than ChaCha20 (algorithm 3) or AES (algorithm 4).

Risk. Counter reuse across runs with the same key catastrophically breaks the cipher (XORing two ciphertexts cancels the keystream and leaks plaintext-XOR-plaintext). The counter derivation must be *deterministic from the SCR structure*, not random per run, otherwise the inverse pass can't reproduce it.

Round-trip status. Bijective under matching counters → same byte-identity guarantee as algorithm 1.

3.2 Algorithm 3 — ChaCha20 stream cipher

Idea. Replace XTEA entirely with ChaCha20 (Bernstein, RFC 8439). 256-bit key, 96-bit nonce, stream-cipher construction. Apply per-edp with a unique nonce derived from the edp's identity coordinates.

What it fixes. XTEA's weak-cipher status. ChaCha20 is the modern stream cipher of choice for new designs; it has no known attacks at full strength and is fast in pure software.

Cost. Larger key + nonce state per edp; ~3× cipher cost of XTEA per block for unoptimised C++ implementations, comparable or faster with AVX2-aware impls. Adds a dependency choice: vendor a ChaCha20 implementation (a few hundred lines of clean C) or link `libsodium`.

Risk. Nonce-reuse failure mode same as algorithm 2 but slightly more catastrophic (ChaCha20's keystream is 64 GiB before exhaustion vs XTEA-CTR's much smaller domain). Vendor implementation needs to be audited (RFC 8439's test vectors are mandatory).

Round-trip status. Bijective under matching nonces → byte-identity preserved.

3.3 Algorithm 4 — AES-CTR

Idea. Replace XTEA with AES (Rijndael) in CTR mode. 128-bit key, 128-bit block, well-studied since 2000.

What it fixes. Same as Algorithm 3 — XTEA's weak-cipher status.

Cost. AES is hardware-accelerated on x86_64 (AES-NI) and ARMv8 (AES instructions); software AES is slower than ChaCha20 by a factor of ~3-5. CCS deployment target is unclear; if the obfuscator runs server-side on AES-NI-capable hardware, AES-CTR is the speed winner.

Risk. Reliable AES implementations are non-trivial (timing-attack mitigations, constant-time round functions); vendoring requires care. If AES-NI is *not* available, software fallback paths need their own audit.

Round-trip status. Bijective under matching counters.

3.4 Algorithm 5 — SCR-level identifier renaming

Idea. Rename every non-predefined kwn from its readable name (`kwn.kwnkw_.name_`) to an opaque tag (`"kwn_<N>"` for a permuted `N`). Preserve the permutation in a sidecar map file (`<output>.obf.5.namemap`) so the inverse pass can restore originals.

What it fixes. The §2.2 "kwn-name strings are still readable" leak. After algorithm 5 a `.urt` decompile shows opaque `kwn_42 : kwn_7 | kwn_99` instead of meaningful rule names.

Cost. Read-cheap (string replacement during walk). Inverse pass needs the sidecar map; without it, recovery is impossible. The map is plaintext (otherwise you need a key-managed cipher on the map too) — so the cipher strength is the map's *storage* security, not any cryptographic primitive.

Risk. Round-trip *does not* preserve binary byte-identity — the binary contains the renamed kwn-name strings, so forward+inverse only recovers the original if the inverse explicitly restores names from the map. Round-trip oracle needs to be reframed as "after inverse-with-correct-map,

binary is byte-identical to original" rather than the algorithm-1 "raw byte-identity after forward+inverse".

Round-trip status. Conditionally bijective (depends on sidecar map availability).

3.5 Algorithm 6 — Symbol-ID permutation

Idea. Apply a key-derived permutation to the `(cnmnum_, kwnnum_, edpnum_)` identity space — kwn 7 becomes kwn 42, kwn 42 becomes kwn 13, etc. Apply the same permutation transitively to every reference (the `(kwn, edp)` pairs encoded in `edpfix_` / `edpdyn_`) so internal references stay consistent under the renaming.

What it fixes. The §2.2 "structural shape leak": attackers can still see the *number* of kwns but no longer which is which. Combined with algorithm 5 (name renaming), the structural shape is fully opaque.

Cost. One walk to build the permutation, one walk to apply it (rewrite identity fields + rewrite references). Bijective under any permutation, so round-trip-clean.

Risk. The permutation must be a true bijection on the existing kwn-number space, not a stream-cipher output that could collide. Easiest construction: Fisher-Yates shuffle seeded by `make_key(seed)`. The inverse pass needs to reconstruct the *inverse permutation* — either derive it from the same seed (deterministic Fisher-Yates), or store the forward permutation in a sidecar (like algorithm 5).

Round-trip status. Bijective under matching seed.

3.6 Algorithm 7 — Composite (payload cipher + ID permutation)

Idea. Compose algorithm 1 (or 2 / 3 / 4) with algorithm 6. First permute identity fields, then cipher payloads. Decipher in reverse order.

What it fixes. The combined leak surface: payloads are ciphered (no readable rule-body content), identity fields are permuted (no readable kwn/edp numbering), and optionally — if composed with algorithm 5 too — names are renamed (no readable rule names).

Cost. Sum of component costs. The walk-per-pass overhead dominates; the cipher math is cheap.

Risk. Composition order matters for the inverse pass. The inverse of `compose(A, B)` is `compose(B^-1, A^-1)` — easy to get wrong if both passes happen in the same loop. Algorithm 7

should explicitly serialize "cipher pass first, permute pass second" or vice versa, and document the inverse ordering in the source.

Round-trip status. Bijective composition of bijections.

3.7 Selection guidance

Algo	When to pick	When NOT to pick
2	Want to keep XTEA (no new dependency) but eliminate the ECB leak	Want cryptographic strength — XTEA is still the underlying primitive
3	Want modern-strength cipher in pure software, no HW accel assumption	Cannot accept any new vendored code
4	Deployment targets x86_64 server with AES-NI	Software-only deployments on older / embedded HW
5	Want to hide *names*, not payloads. Pairs with 1-4.	Need binary byte-identity round-trip on its own
6	Want to hide *structure*, not payloads. Pairs with 1-4.	Have a downstream consumer that depends on stable kwn-number identity
7	Want maximum surface coverage. Production use.	Need to debug — composition makes failure diagnosis noticeably harder

---

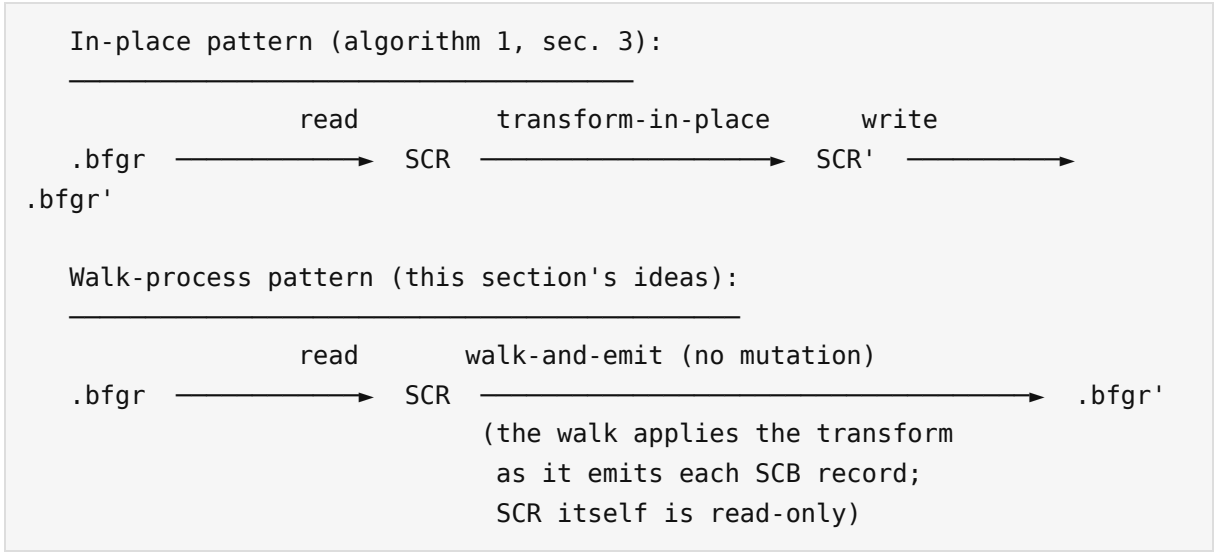
4. SCR → SCB Walk-Process Algorithmic Ideas

Algorithm 1 follows what we'll call the **in-place transform pattern**: load SCR, mutate it in place, serialize. Section 3's algorithms 2-6 follow the same pattern. But there's a separate family — the **walk-process pattern** — where the transformation happens *during* the SCR → SCB serialization

walk rather than as an in-place SCR edit before serialization. This section explores ideas in that family.

The walk-process pattern is interesting because it can encode transformations that don't have a natural in-place SCR representation — adding or removing rules, restructuring the edp graph, inserting decoys, and so on.

4.1 The Walk-process pattern vs the in-place pattern



Implementation-wise, the walk-process pattern would extend `SyntaxControlledConverter::generateBinaryFromRuntime` with a visitor / hook interface — each visited element of the SCR runs through a registered transformer before being emitted to the SCB output. The current implementation has no such hook; adding it is a substrate task that this section's ideas collectively motivate.

Compared with the in-place pattern:

Property	In-place	Walk-process
Round-trip byte-identity	Inherent (cipher is a bijection)	Per-algo: only if walk is too
Can add / remove rules	Awkward — would mutate kwn vector	Natural
Can restructure edp graph	Hard	Natural

```
├──────────────────────────────┼──────────────────────┼──────────────────────┤
│ Memory cost                  │ One SCR              │ One SCR + visit      │
│                              │                      │   state              │
├──────────────────────────────┼──────────────────────┼──────────────────────┤
│ Inverse complexity           │ Cipher inverse       │ Often requires       │
│                              │   primitive          │   sidecar trace      │
├──────────────────────────────┼──────────────────────┼──────────────────────┤
│ Substrate dep                │ None (works today)   │ Visitor hooks        │
│                              │                      │   in                 │
│                              │                      │   generateBinary-    │
│                              │                      │   FromRuntime        │
└──────────────────────────────┴──────────────────────┴──────────────────────┘
```

4.2 Idea W.A — Walk-time kwn renaming

Same effect as algorithm 5 (§3.4) — rename kwn-name strings — but apply the renaming *during* the walk rather than mutating the SCR first. The kwn-name visitor receives `(orig_name, visit_index)` and returns the rewritten name; the SCR is unchanged after the walk.

Why prefer over algorithm 5? It composes more cleanly with other walk-process ideas (W.B, W.C, etc.) — multiple visitors chained at the same emission point, single walk, no intermediate SCR mutation needed.

4.3 Idea W.B — Walk-time dead-rule elimination

Visit each kwn from the axiom; track reachability. Kwns unreachable from the axiom are silently dropped from the emitted SCB. After the walk, the output binary has only the reachable rules.

What it's for. Compaction (smaller `.bfgr` files, faster parser builds), and as an obfuscation step it removes "unused" rules an attacker might use to fingerprint the grammar's design lineage.

Round-trip status. *Not* round-tripping — dropped rules are gone. This is the first non-bijective transform in this document, and it deliberately is so.

Inverse. Doesn't exist; the dropped rules are unrecoverable. Forward-only.

4.4 Idea W.C — Walk-time alphabet permutation

For each emitted symbol reference (every `(kwn, edpnum)` tag in `edpfix_` / `edpdyn_`), apply a key-derived permutation to the *target kwn number*. The SCR stays unchanged; the emitted SCB's

edpfix_/edpdyn_ records contain permuted references.

Why is this different from algorithm 6 (§3.5)? Algorithm 6 mutates the SCR's identity fields *and* the references. Idea W.C only permutes the references during emission — the SCR itself, and a decompile from the original SCR, looks unchanged. The permuted SCB only emerges at write-time.

Round-trip status. Bijective under matching permutation seed. Inverse pass applies the inverse permutation during its own walk.

Why prefer over Algorithm 6? Same composition benefit as W.A — chains cleanly with other walk visitors. Also leaves the in-memory SCR usable for other tools (linters, stats, visualisers) that want to see the *original* runtime, not the permuted one — useful in a multi-tool pipeline.

4.5 Idea W.D — Walk-time epsilon insertion

For each emitted edp, optionally insert decoy `KW_EPSILON` markers at key-derived positions. The parser treats `KW_EPSILON` as match-nothing-consume-zero, so the generated parser still behaves correctly — but a decompile shows a noisier rule shape than the original (`A B C` might emit as `A ε B ε C ε ε`).

What it's for. Defeating "I can read the rule shape even through ciphered payloads" structural inspection — the inserted ε markers change the *visible* rule shape without changing the *parsing* behaviour.

Round-trip status. Forward-only. The inverse can't tell which ε markers were original (if the grammar legitimately contained any) vs decoys.

Risk. The parser-time cost of ε markers is non-zero (one match-and-skip per marker); inserting many noticeably slows parsing of large inputs. The walk should rate-limit ε insertion to a small percentage of total edp positions.

4.6 Idea W.E — Walk-time edpfix/edpdyn shuffle

For each emitted edp, apply a key-derived structure-preserving permutation to the order of *commutative* rule-body terms. Most context-free productions are *not* commutative (order matters), but some constructions — alternation rule bodies in particular — are: `A | B | C` parses equivalently to `C | A | B`. Walk-time can detect alternation edps and shuffle their order.

What it's for. Defeats positional fingerprinting — even if an attacker recovers the grammar shape, the alternation order no longer matches the original.

Round-trip status. Bijective under matching permutation seed for the alternation edps; identity for non-alternation edps.

Risk. False-positive shuffling on edps that *look* commutative but aren't (e.g., semantic-action-dependent order) would silently break the generated parser. The detection logic needs to be conservative — when in doubt, don't shuffle.

4.7 Pattern summary table

Idea	What it does	Round-trip	Pairs with
W.A	Rename kwn names during emit	Yes (with sidecar)	Algo 1-4 in §3
W.B	Drop unreachable kwns during emit	No (lossy)	Standalone (compaction)
W.C	Permute kwn references during emit	Yes	Algo 1-4 in §3
W.D	Insert decoy ε markers during emit	No (lossy)	Standalone (decoy noise)
W.E	Shuffle commutative edp bodies during emit	Yes (with seed)	W.C, Algo 1-4

Common substrate prerequisite. All five W. *ideas need visitor hooks in* `generateBinaryFromRuntime`*. The current implementation is monolithic — adding the hook interface is a one-time substrate addition (analogous to the Phase 2.2b substrate addition that step8.22 made for the in-place family). Once that substrate exists, the W.* ideas become roughly equal-cost to implement individually.

---

5. Security Analyst's TODO Items

This section converts §2's "Bad" and "Ugly" findings, plus the user-proposed CCT Repository ciphering concept, into a prioritised work list. Items here cross-reference TODO.md entries (cct_boot/TODO.md is the master inventory; this section duplicates the R1-R6 entries for context).

5.1 Algorithm-1 hardening

These items address weaknesses *internal to algorithm 1*. They do not require new algorithm numbers — they tighten what's already there. Each can land independently; none are prerequisites for §5.2.

5.1.1 Replace ECB-mode with CTR mode (or jump to algorithm 2/3/4)

Problem. §2.2 "ECB-mode equivalent leaks repeated-block structure". Identical plaintext blocks produce identical ciphertext blocks; a statistical attacker can identify shared prefixes across edps without breaking XTEA.

Fix. Either (a) extend `cipher.h` with a CTR-mode wrapper around XTEA (the per-edp counter derivation is the design surface) and switch algorithm 1 to use it, or (b) leave algorithm 1 as-is and introduce algorithm 2 (XTEA-CTR) as its successor. Option (b) is non-breaking for existing `.obf.1.*` files; option (a) breaks them.

Acceptance. Recomputed leak surface: ciphertext for identical plaintext blocks at different positions must be *different*. Test by constructing an SCR with two known identical edpfix_ vectors, ciphering, asserting the ciphertexts differ.

5.1.2 Replace `make_key` with a real KDF

Problem. §2.2 "`make_key` is not a cryptographic KDF". The round-robin XOR collapses 17+-byte seeds into a 16-byte space with trivially-constructable collisions.

Fix. Replace `make_key` with PBKDF2-HMAC-SHA256 or Argon2id. PBKDF2 has the lower implementation cost (a few hundred lines, SHA-256 + an HMAC wrapper); Argon2id is the memory-hard modern choice but adds significantly more code.

Acceptance. Verify against PBKDF2 RFC 8018 test vectors; verify two seeds differing by one bit produce keys differing in expectation by ~64 bits (avalanche).

5.1.3 Remove the `kDefaultKeySeed` footgun

Problem. §2.3 "`kDefaultKeySeed` is a footgun". Anyone who runs `obfuscator -a 1 file.bfgr` without `-k` gets reproducible ciphertext under a globally-known key.

Fix. Require `-k <seed>` for any algorithm > 0. Error out if missing. Algorithm 0 (the no-op round-trip oracle) keeps the no-`-k` shape since it doesn't cipher anything.

Acceptance. `obfuscator -a 1 file.bfgr` (no `-k`) exits non-zero with a clear error message; `obfuscator -a 1 -k secret file.bfgr` works as before. Update `test_algo1_json_round_trip_script` to pass `-k` explicitly.

5.1.4 Document the structural-shape leak surface

Problem. §2.2 "Algorithm 1 leaves structural shape intact". An attacker who reads a `.obf.1.bfgr` can still infer kwn/edp counts, edpfix/edpdyn lengths, kwn-name strings, and grammar shape.

Fix. Document this explicitly in `cipher.h` and in this compendium (§2.2 covers it; consider promoting to a prominent banner in the header file too). Reference §3 for which algorithms close which leak surfaces (algorithm 5 for names, algorithm 6 for structure, algorithm 7 for the composite).

Acceptance. A doc-only change; no code impact. Future maintainers reading `cipher.h` should see the leak surface called out.

5.1.5 Hide debug-dump files behind a flag

Problem. §2.3 "`.scbtxt` / `.scrtxt` ship on every run". Both files leak pre-transform memory state to disk.

Fix. Add a `--debug-dumps` flag; default off. When off, skip the `writeMemoryDump` calls. When on, behave as today.

Acceptance. A run without `--debug-dumps` produces only `.obf.<algo>.fgr` and `.obf.<algo>.bfgr`; a run with it produces all four.

5.2 CCT Repository ciphering / deciphering (R1-R6)

Tracked in `cct_boot/TODO.md` under section R. Six work items addressing "cipher every binary in a CCT Repository in place, keyed off a managed key collection". Origin: user proposal

recorded 2026-05-25. Reproduced here for the security-analyst audience.

Idea	Concern	TODO
–	Threat model (foundation)	R1
1	Key collection storage	R2
2	Next-key selection mechanism	R3
3	Verify-binary-was-ciphered-with-K	R4
4	Cipher entire repo in place + decipher	R5
5	`repocli cipher -k` / `decipher -k`	R6

5.2.1 R1 — Threat-model doc

Foundational item. Before anything in R2-R6 can be designed sensibly, the threat model needs to be written down: who is the attacker, what do they have access to, what do they *not* have access to, and what does ciphering the CCT Repository protect against?

Two specific gaps must be answered explicitly:

- **Plaintext-seed problem.** If `.sgr` is left on disk and only `.bfgr` is ciphered, the cipher is moot — `ccsjson <s>.sgr` regenerates the plaintext binary. Either cipher `.sgr` too, or write down explicitly that this protects only the binary form against an attacker who has the binary but not the source.
- **Self-hosting bootstrap protection.** `cppcc`'s own build needs `srcloc/metaFirstParser.cc` + `incloc/` in cleartext to compile. Cipher scope must exclude these; the exclusion list is itself a known-plaintext attack surface that needs explicit documentation.

Acceptance. One-page document, separately reviewable, that anchors all later R-section choices.

5.2.2 R2 — Key collection storage

A collection of 128-bit keys (today produced by `ccs_obf::cipher::make_key(string_view)`). Open design surface:

- **Location.** On-disk file (then: protected how?), env-var-driven (then: rotation friction), or external KMS shim (then: deployment friction).

- **Self-protection.** Anchored by R1's threat model. Unprotected keystore on the same host as the ciphered binaries means the cipher is theater.
- **Lifecycle.** Rotation, revocation, retirement, versioning.

Dependency. Blocked on R1.

5.2.3 R3 — Key selection mechanism

"Next key" needs an answer to *next per what* (per-file? per-repo? per-time-window?). The decipher side needs to know which key to use *without* trying all of them. Two canonical shapes:

- **Embed key-ID in ciphertext header.** Header is plaintext, reveals key-ID but hides payload. Constrains the binary format.
- **Key-derivation hierarchy.** One root key + path-derived per-file key. Single CLI key still unlocks everything. No format change.

The choice constrains R6's CLI: `-k stringToken` works trivially for KDF, awkwardly for ID-in-header.

Dependency. Blocked on R1 + R2.

5.2.4 R4 — Key-match oracle

Verify whether a given `.bfgr` / `.fgr` was ciphered using a specific key. Three candidate approaches:

```
┌──────────────────────────────┬──────────┬──────────────────┬──────────────────┐
   │ Approach                     │  Speed   │ Format impact    │ Leak
│
├──────────────────────────────┼──────────┼──────────────────┼──────────────────┤
   │ Magic-canary in header       │   O(1)   │ Modifies .bfgr   │ "is XTEA,
key=?" │
   │ Try-decipher + struct chk    │   O(N)   │ None             │ None
│
   │ Sidecar .bfgr.kt             │   O(1)   │ Extra file       │ Visible file
│
```

```
└──────────────────────────────┴──────────┴──────────────────┴───────────────────┘
```

The **try-decipher + structural-validity** path reuses `ccs_obf::cipher::validate_edp_refs` (already in `cipher.h`, exercised as the structural-validity oracle in test_algo1 case 5). Slow but correct, no leak — the natural first implementation.

Dependency. Depends on R3 (key-selection decision drives which approach is appropriate); independent of R5.

5.2.5 R5 — Bulk in-place cipher/decipher

Cipher / decipher every binary in a CCT Repository in place. Operationally the riskiest of the six. Four concrete gaps:

1. **Per-file atomicity.** Cipher to <f>.bfgr.tmp, fsync, atomic rename. Standard POSIX pattern.
2. **Across-the-repo atomicity.** Needs a journal — list of files to process, mark each completed after rename, recovery reads the journal on restart. Without one, an interrupted run leaves a half-ciphered repo with no way to identify the boundary; recovery degenerates to "ask R4 on every file" — the slow path at repo scale.
3. **Scope definition.** "All CCT Repository instances" — `.bfgr/.fgr` only? `.sgr` too? Generated `*.cc/*.h/*.py/*.mjs`? Each tier has different operational consequences (and the `.sgr` decision is the plaintext-seed problem from R1).
4. **Self-hosting bootstrap exclusion list** (per R1). The exclusion list is the load-bearing safety mechanism; it also leaks "these are the boot-critical files" to an attacker who can read the script.

Dependency. Depends on R1 (scope decision) + R2/R3 (which keys orchestrator uses for which files); coordinates with R6.

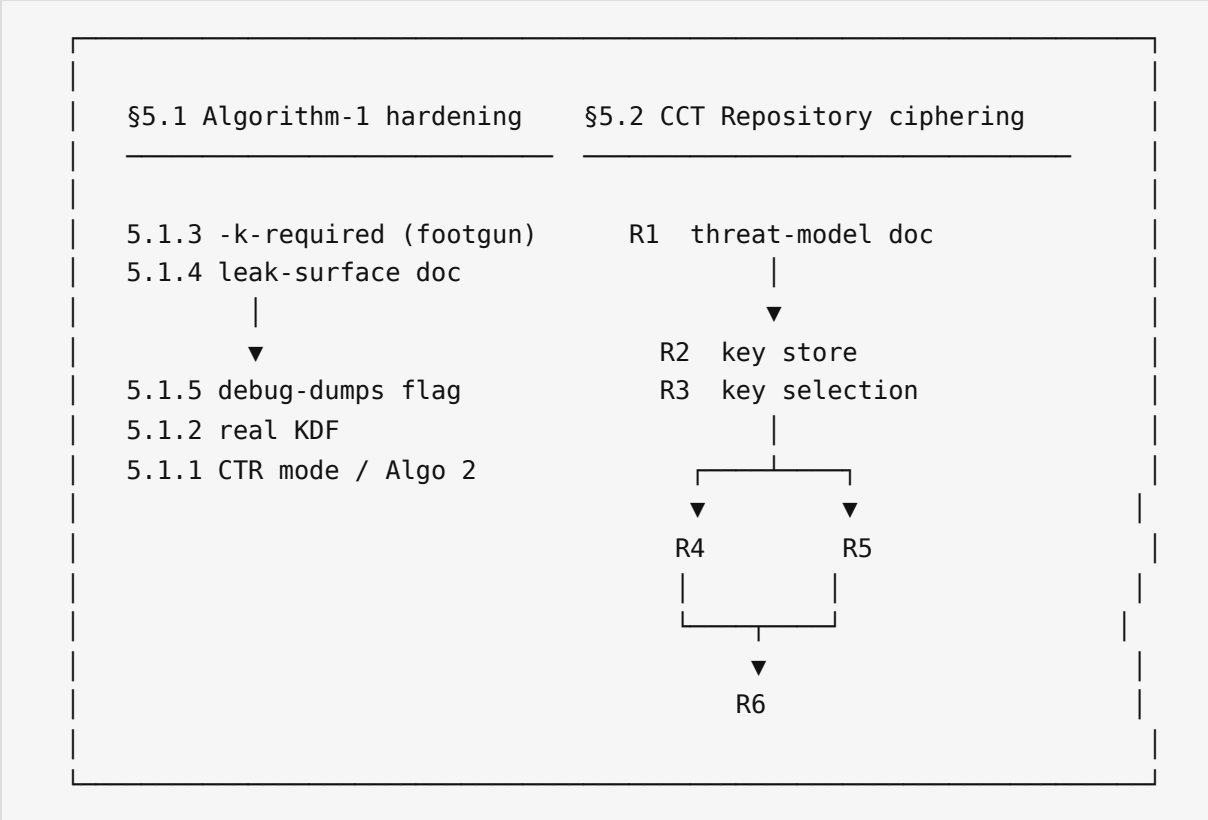
5.2.6 R6 — `repocli cipher -k` / `decipher -k` CLI

Smallest surface of the six; orchestrates R2-R5. The `-k stringToken` form matches the existing `cipher::make_key(string_view)` convention from step8.24, so the CLI doesn't add new key-derivation semantics on its own — it defers to R3.

One open routing question: is `repocli` (the cct_repo CLI) the right host for cipher/decipher, or should it sit in `ccs_obfuscator` and be invoked by `repocli` as a subprocess? The answer probably depends on R2 — if the keystore is a CCS-wide concern, `repocli` is the right home (consumer of `ccs_config`-style data + the repo-scope orchestrator); if it's a cipher-tool-internal concern, `ccs_obfuscator` extends its own CLI.

Dependency. Smallest piece but waits on R1-R5 to land first; implementing it before the others would just relocate the unanswered questions.

5.3 Recommended sequencing



Reasoning. 5.1 items are local to the obfuscator and can land in any order without coordination — pick the easy ones first (5.1.3, 5.1.4 are doc-only or trivial; 5.1.5 is small). The §5.2 chain has hard dependencies: R1 unblocks R2/R3; R2+R3 unblock R4 and R5; R6 sits at the top.

Single highest-priority item: R1 (threat-model doc). Every other §5.2 decision is unanchored without it. Single highest-priority §5.1 item: 5.1.3 (remove the `kDefaultKeySeed` footgun) — minimal code change, disproportionate security improvement.

---

6. Grammar Recognition as a Security Primitive

Everything up to this point has treated `ccs_obfuscator` as a cipher tool: algorithm 1 encrypts payload bytes, §5 lists the ways that encryption is currently weak. That framing is accurate but it is not the interesting one, and reading only that framing leads to the wrong conclusion — that this project is a mediocre implementation of a solved problem.

This section states the other framing. It is the reason the project exists, and it is what an algorithm author on this platform should understand before writing anything.

6.1 Two different things a secret can be

Classical cryptography rests on Kerckhoffs's principle: assume the adversary knows the algorithm; security must depend only on the key. It is a discipline of *withheld values*. The system is public, one number is not, and the entire defence is the cost of searching for that number.

This has a structural consequence that is easy to overlook because it is so familiar: **a key is a thing that exists.** It sits in memory, in a config file, in an environment variable, in a backup, in a developer's shell history. It can be copied without being removed. It can be disclosed by a person. Every serious key-management practice — rotation, escrow, HSMs, split knowledge — is engineering around the fact that the secret is an object in the world.

A CCS SCB instance suggests a second kind of secret. A `.bfgr` file is not text and not a general-purpose serialisation; it is a structure whose interpretation is defined by the grammar that produced it. Symbol identities, production numbering, and the `edpfix_`/`edpdyn_` layout are all *relative to that grammar*. Given the grammar, reading the instance is mechanical — that is precisely what the generated frontends do. Without it, the reader is not decrypting; the reader is trying to infer a formal language from examples of its output.

That is a categorically different problem, and it is a hard one.

6.2 What is actually known about the hardness

Grammar induction — recovering a formal grammar from example strings — is one of the older studied problems in learning theory, and the results are consistently negative for the learner.

Identification in the limit is impossible from positive examples alone. Gold (1967) showed that no algorithm can identify an arbitrary language from any *superfinite* class (one containing all finite languages plus at least one infinite language) given only positive examples, no matter how

many. The class of regular languages is superfinite. So is the class of context-free languages. An adversary holding a corpus of valid instances — and nothing that tells them what is *invalid* — is in this setting.

Finding the smallest consistent automaton is NP-complete. Gold (1978) and Angluin (1978) established that, given finite sets of accepted and rejected strings, deciding the existence of a DFA with at most *k* states consistent with them is NP-complete. Exact minimal inference is not merely inconvenient; it is intractable in the general case.

Even approximating it is hard. Pitt and Warmuth (1993) strengthened this substantially: the minimum consistent DFA problem cannot be approximated within any polynomial factor unless P = NP. This closes the usual escape hatch, since for most NP-hard problems a good-enough approximation is the practical answer. Here there isn't one.

For context-free grammars — which is what SGDL describes — the picture is worse still, since even *deciding equivalence* of two CFGs is undecidable.

The practical shape of this is worth stating plainly:

```
┌──────────────────────┬──────────────────────────────────────┐
│ classical cipher     │ grammar-protected structure          │
├──────────────────────┼──────────────────────────────────────┤
│ secret is a value    │ secret is a specification            │
│ attack = search key  │ attack = induce a grammar            │
│ cost: exponential in │ cost: NP-hard to infer minimally,    │
│   key length         │   inapproximable, undecidable to     │
│                      │   verify equivalence (CFG)           │
│ secret can leak      │ nothing to leak — no key exists      │
│ compromise is total  │ compromise is per-grammar            │
│   and instant        │   and requires reconstruction        │
└──────────────────────┴──────────────────────────────────────┘
```

The row that matters most is the fourth. There is no key to steal because the defence is not a withheld value; it is a withheld *specification*, and a specification is only recoverable by being re-derived.

6.3 What is NOT claimed

An argument of this shape attracts overstatement, and overstatement is how a real result gets discredited. The following limits are part of the claim, not caveats appended to it.

NP-hardness is worst-case, not average-case. These theorems say that *some* instances are hard. They say nothing about whether the particular grammar you are protecting is one of them. A small, regular, conventionally-structured grammar may be recoverable in an afternoon by a competent engineer with no theory at all. Worst-case hardness has never been a sufficient basis for a security guarantee — it is the same gap that separates lattice problems from lattice *cryptosystems*, and it took the field decades to bridge it properly.

A public grammar destroys the argument entirely. If the source language is JSON, or MeTTa, or any published standard, the adversary simply obtains the grammar. Nothing above applies. This defence exists only for *proprietary* languages and internal representations — which is a real and common case, but a narrower one than "CCS protects your data".

A membership oracle changes the complexity class. This is the sharpest limit. Angluin (1987) showed that regular languages *are* efficiently learnable in polynomial time given a minimally adequate teacher — an oracle answering membership and equivalence queries. If an adversary can feed inputs to a running frontend and observe whether they are accepted, they have exactly such an oracle, and the hardness results above stop protecting anything. **Any deployment that exposes a parser as a queryable service has handed the attacker the teacher the theory assumes they lack.**

Structure is not payload. Grammar-derived hardness protects how an instance is *organised*. It does not protect the bytes inside it. A string literal sitting in an `edpfix_` vector is readable by anyone who opens the file, grammar or no grammar. This is exactly why algorithm 1 exists.

Exact induction is not the attacker's goal. The theorems concern recovering *the* grammar, or a minimal one. An adversary usually needs far less: enough approximate structure to extract one field of interest. Hardness of exact inference is weak evidence about the difficulty of partial, targeted extraction.

6.4 Why the two compose

Read together, §6.2 and §6.3 point at the same conclusion, and it is the thesis of this project: **structural hardness and payload encryption defend different things, fail in different ways, and are therefore worth stacking.**

A cipher protects values and fails catastrophically on key compromise — one disclosure and every artifact ever produced is open. Grammar-derived hardness protects organisation and fails gracefully but partially — it degrades against oracles, side information, and targeted extraction,

yet there is no single disclosure that opens everything at once, because there is no single thing to disclose.

An attacker facing both must obtain the key *and* reconstruct the specification. Neither compromise implies the other. That is the whole argument, and it is why §5's honest catalogue of algorithm 1's weaknesses is not an embarrassment: the cipher layer is the *replaceable* half, which is precisely why this project is built as a platform for writing better ones.

6.5 What this means for an algorithm author

If you are implementing an algorithm on this platform (§1.7), the framing above has direct consequences:

- **Do not rely on the grammar staying secret for anything that must not leak.** Treat structural hardness as defence in depth, never as the primary control. If a value must be confidential, cipher it.
- **Consider what your transformation does to the oracle surface.** An algorithm that makes malformed input fail *differently* from well-formed input is leaking membership answers. Uniform failure is worth real design effort.
- **Structural transformations are additive.** Identifier renaming (algorithm 5), symbol-ID permutation (algorithm 6) and the walk-process ideas of §4 attack the induction problem directly, by removing the human-meaningful regularities that make a grammar guessable in practice. They are the half of the design space where this section's argument actually applies.
- **State your threat model.** Every algorithm in this project should document what it protects against and what it does not. §2 and §5 are the model for that, and they are the reason this document is worth reading.

6.6 References for this section

- Kerckhoffs, A. (1883). *La cryptographie militaire.* Journal des sciences militaires.
- Gold, E. M. (1967). Language identification in the limit. *Information and Control* 10(5).
- Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control* 37(3).
- Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control* 39(3).
- Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation* 75(2).

- Pitt, L., & Warmuth, M. K. (1993). The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM* 40(1).

---

7. Appendices

A. Glossary

Term	Definition
SCB	Syntax-Controlled Binary. Serialized form of a CCS grammar; `.fgr` is decimal text, `.bfgr` is raw native bytes.
SCR	Syntax-Controlled Runtime. In-memory form of the grammar; navigable via cnms_ / kwns_ / edps_ containers.
Phase 2.1a	Source-text → SCR (parsing / building).
Phase 2.2a	SCR → SCB (serialization). cppcc surfaces.
Phase 2.2b	SCB → SCR (deserialization). ccs_obfuscator surfaces; cppcc does not.
Phase 2.3	SCB → text (decompile). cppcc -u surfaces.
cnm	Context name. Top-level container in SCR.
kwn	Keyword. One per grammar rule / terminal.
edp	Edge production. One per alternation branch of a rule.
edpfix_	Fixed-arity rule-body terms of an edp.
edpdyn_	Dynamic-arity rule-body terms of an edp (lists, repetitions).
tag::Long	The scalar type used for SCR/SCB words. 64-bit in the default build; 32-bit under _CPPCC_TAG_32BITS_.
XTEA	eXtended Tiny Encryption Algorithm. Wheeler & Needham, 1997. 64-bit block, 128-bit key.
ECB	Electronic Code Book — cipher mode that processes each block independently.
CTR	Counter mode — cipher mode that XORs plaintext with a cipher-of-counter keystream.
KDF	Key Derivation Function. Maps a low-entropy secret (a password) to a high-entropy key.
CCT Repository	The repository-of-grammars data model

```
|                   |   maintained by `repocli` (cct_repo).          |
| R1-R6             | The six TODO.md entries for CCT Repository    |
|                   |   ciphering. See §5.2.                         |
└───────────────────┴────────────────────────────────────────────────┘
```

B. References

Algorithm 1 — XTEA. David J. Wheeler, Roger M. Needham. "TEA extensions". Computer Laboratory, Cambridge University, 1997. Public-domain. The 30-line cipher loop in `xtea.h` follows the published spec.

Algorithm 3 — ChaCha20. Daniel J. Bernstein. "ChaCha, a variant of Salsa20." 2008. IETF RFC 8439 (2018) gives the canonical pseudocode and test vectors.

Algorithm 4 — AES. Joan Daemen, Vincent Rijmen. "AES Proposal: Rijndael." NIST AES competition, 1998. Standardized as FIPS 197 (2001). AES-NI instruction set added to x86_64 in 2008 (Intel Westmere).

5.1.2 KDF. Burt Kaliski. "PKCS #5: Password-Based Cryptography Specification Version 2.0." RFC 2898 (2000), updated as RFC 8018 (2017) — PBKDF2. Alex Biryukov, Daniel Dinu, Dmitry Khovratovich. "Argon2: the memory-hard function for password hashing and other applications." 2015 (RFC 9106, 2021).

CCS substrate. `cppcc/CLAUDE.md` is the canonical entry point. The Phase taxonomy comes from `ccs_python_dev/UML-DIAGRAMS.md` § 3.1 and `ccs_scb/docs/CCS_Syntax-Controlled_Binary_Users_Guide.md`.

Algorithm-1 implementation. `ccs_obfuscator/include/xtea.h`, `ccs_obfuscator/include/cipher.h`, `ccs_obfuscator/src/obfuscator.cc`. Landing commits in step8.22 (`8192b12` cppcc + `9296d1f` ccs_obfuscator), step8.24 (`1ea43d2` ccs_obfuscator), step8.25 (`79a3811` ccs_obfuscator).

CCT Repository ciphering concept. User-proposed, recorded 2026-05-25 in `cct_boot/TODO.md` section R (commit `0630aa5`). This compendium's §5.2 mirrors that material.

End of CCS Obfuscator Compendium, Draft 0.1.