GrokRxiv:2026.04.c-rust-transpiling [cs.PL] 16 Apr 2026

# Climbing the Safety Ladder:

## From `c2rust`'s Raw Lift to Idiomatic Safe Rust via Differential Fuzzing, Miri, and Refinement Verification

A 1-Week libyaml Case Study and Generalization to libexpat

Matthew Long
*The YonedaAI Collaboration*
*YonedaAI Research Collective*
Chicago, IL
`matthew@yonedaai.com` · `https://yonedaai.com`

April 16, 2026

**Abstract**

Automated C → Rust transpilation, as exemplified by Immunant's `c2rust` (DARPA TRACTOR-funded), is a solved *compilation* problem: the Clang AST of any C translation unit can be mechanically rewritten into Rust that compiles, links, and preserves the original ABI. It is, however, an unsolved *safety* problem. The output is pervasively `unsafe`: raw `*mut T` pointers, manual bounds arithmetic, `0/1` integer-as-error returns, tagged C unions accessed by raw field probes. The `unsafe-libyaml` crate—`libyaml` run through `c2rust` with cleanup—retains every UB class that the original C admits and provides serde_yaml's 90 M+ downloads with no additional security guarantee.

We formalize the gap between the c2rust lift and idiomatic safe Rust as a *safety ladder*: a chain of refinement relations $\sqsubseteq_{\text{compile}}\sqsubseteq_{\text{mem}}\sqsubseteq_{\text{behav}}\sqsubseteq_{\text{conc}}\sqsubseteq_{\text{refine}}\sqsubseteq_{\text{abi}}$ indexed by an increasingly strong verification regime (`rustc`, Miri, differential fuzzing, Loom/`cargo-careful`, Kani/Creusot, cbindgen ABI parity). We exhibit twelve refactoring patterns that take c2rust's raw output to a target safe form, prove a differential equivalence theorem $\forall b{\in}\mathcal{B}.\ \text{parse}_{\text{C}}(b) \simeq \text{parse}_{\text{R}}(b)$ that justifies coverage-guided fuzzing as a semantic-equivalence oracle, and present a worked Kani proof for libyaml's `STRING_EXTEND` buffer-extension invariant.

The methodology is validated by an explicit 7-day, 5-agent translation schedule for libyaml (∼9,300 LOC, 175 functions, 10 source files): day-by-day work allocation, exact CLI invocations, and a 36-task prototype-development plan in Appendix A. We then sketch the immediate generalization to libexpat (∼15,000 LOC, 40+ CVEs), where the same ladder applies with two new rungs (entity-expansion depth and namespace separator validation). The result is a reproducible recipe for moving mechanically-translated unsafe Rust onto a safety ladder whose top rung is publishable on `crates.io`.

# Contents


**1 Introduction** **3**
- 1.1 The c2rust gap . . . 3
- 1.2 Why libyaml is the perfect first instance . . . 3
- 1.3 Contributions . . . 3
- 1.4 Audience and scope . . . 4

**2 The c2rust pipeline** **4**
- 2.1 What c2rust does . . . 4
- 2.2 What c2rust gets right . . . 6
- 2.3 What c2rust leaves on the table . . . 6

**3 The Safety Ladder** **6**
- 3.1 Refinement relations . . . 6
- 3.2 The chain . . . 7
- 3.3 Inference rules . . . 8
- 3.4 Diagram . . . 8

**4 Refactoring patterns** **8**
- 4.1 Pattern 1: raw *mut T field $\to$ owned/borrowed Rust . . . 9
- 4.2 Pattern 2: index loop $\to$ iterator combinator . . . 9
- 4.3 Pattern 3: char * $\to$ &str / String / Cow<'_, str> . . . 9
- 4.4 Pattern 4: malloc/free $\to$ Box::new/drop . . . 10
- 4.5 Pattern 5: int return + out-pointer $\to$ Result<T, E> . . . 10
- 4.6 Pattern 6: tagged C union $\to$ Rust enum . . . 10
- 4.7 Pattern 7: linked list of structs $\to$ Vec<T> with indices . . . 11
- 4.8 Pattern 8: function-pointer callback + void * context $\to$ closure . . . 11
- 4.9 Pattern 9: nullable pointer $\to$ Option<T> . . . 11
- 4.10 Pattern 10: integer-coded error $\to$ thiserror variant . . . 12
- 4.11 Pattern 11: self-referential struct $\to$ lifetime-parameterized helper . . . 12
- 4.12 Pattern 12: stack/queue macros $\to$ Vec<T>/VecDeque<T> . . . 12

**5 Differential fuzzing as an empirical equivalence oracle** **13**
- 5.1 Formalization . . . 13
- 5.2 Harness design . . . 14
- 5.3 Tooling tradeoffs . . . 15

**6 Miri for UB detection** **15**
- 6.1 What Miri sees . . . 15
- 6.2 What Miri does not see . . . 16
- 6.3 cargo-careful as runtime complement . . . 16

**7 Refinement-type verification: a Kani case study** 16
7.1 Selecting $\mathcal{F}_{\text{crit}}$ . . . 16
7.2 The libyaml `STRING_EXTEND` invariant . . . 17
7.3 The Rust port . . . 17
7.4 The Kani harness . . . 17

**8 The 7-day libyaml schedule** 18
8.1 Empirical basis for the 7-day target . . . 20

**9 Generalization to libexpat** 20
9.1 What carries over . . . 20
9.2 What is new . . . 21
9.3 Estimated timeline . . . 21

**10 Related work** 22

**11 Discussion** 22
11.1 The role of formal proof . . . 22
11.2 Limits of differential fuzzing . . . 23
11.3 Performance . . . 23
11.4 Failure modes and exit criteria . . . 23
11.5 Cost breakdown . . . 23

**12 Conclusion** 24

**Appendix A. Development Plan for Prototype Agents** 25

**Appendix B. Environmental prerequisites** 33

**Appendix C. The ABI-diff normalizer** 33

# 1 Introduction

## 1.1 The c2rust gap

Immunant's `c2rust` [1] consumes the Clang AST of a C translation unit and emits a Rust crate that compiles to a bit-for-bit-equivalent object file. As a piece of compiler engineering it is excellent. As a delivery mechanism for *memory safety*, it is exactly what its authors warn it is: a starting point. The output, by construction, preserves every C semantic. A function that read past the end of an array in C will read past the end of an array in the c2rust Rust; the only difference is that the Rust source now has `unsafe` written above each such read.

Concretely, the `unsafe-libyaml` crate (David Tolnay's c2rust output for `libyaml`, the dependency under `serde_yaml`'s 90 M+ download tree) contains thousands of `unsafe` blocks, raw pointer arithmetic in the scanner, a self-referential emitter analysis structure encoded as `*mut yaml_char_t`, integer-coded errors masquerading as `int`, and tagged unions whose discriminant is checked by hand before each field access. None of the C UB classes documented in the companion paper on C semantics are eliminated; they are merely *relabeled.*

## 1.2 Why libyaml is the perfect first instance

The libyaml ecosystem provides ground truth at every rung of the ladder we are about to construct:

- **Rung 0 (C source).** The reference implementation `yaml/libyaml` [4], ∼9,300 LOC of hand-written C, MIT-licensed, with a published 400+ case test suite (`yaml-test-suite`).
- **Rung 1 (mechanical lift).** `dtolnay/unsafe-libyaml`, the c2rust transpilation, with the ergonomic post-processing one would expect a Rust expert to apply but no semantic safety changes [2].
- **Rung 2 (manual safe port).** `simonask/libyaml-safer`, a one-week effort by a single engineer proving that the safe-Rust translation is feasible with current tooling and that performance does not regress [3].
- **Rung 3 (AI-assisted, ours).** `safe-libyaml`, the target output of the Ferrous Bridge pipeline, intended to match Rung 2's quality automatically.

The existence of Rungs 1 and 2 means *every* translation decision in the c2rust output has a known correct refactoring; we are not guessing. This makes libyaml the ideal first target for a generalizable *C→ unsafe-Rust→ safe-Rust* pipeline.

## 1.3 Contributions

This paper makes the following contributions.

1. **(Section 2)** A precise account of the `c2rust` pipeline as it stands at v0.21 (October 2025): what it does, what its successor analysis tool intends to do, and the boundary between mechanical lift and semantic refactoring.

2. **(Section 3)** The *safety ladder*: a stack of six refinement relations $\sqsubseteq_{\text{compile}} \cdots \sqsubseteq_{\text{abi}}$ that formalizes "what guarantee does this output have over its predecessor?" We give the inference rules in mathpartir and the diagram in TikZ.

3. **(Section 4)** Twelve refactoring patterns, each given as c2rust output $\rightsquigarrow$ target safe form, with worked libyaml examples (raw pointer triple $\to$ `Vec<u8>`, tagged union $\to$ enum, `int` return $\to$ `Result<T,E>`, etc.).

4. **(Section 5)** A formal differential-fuzzing equivalence claim $\forall b \in \mathcal{B}.\ \text{parse}_{\text{C}}(b) \simeq \text{parse}_{\text{R}}(b)$ together with the harness design that realizes it as a single binary driving both `libyaml.so` and `safe-libyaml` over the same input bytes and comparing event streams.

5. **(Section 6)** A characterization of which UB classes Miri detects (and which it does not), and the role of `cargo-careful` as a runtime complement.

6. **(Section 7)** A worked Kani proof of the `STRING_EXTEND` buffer-extension invariant, demonstrating that the safe Rust port can be *deductively* verified at the function-contract level.

7. **(Section 8)** A day-by-day 7-day libyaml schedule naming the agent responsible for each wave.

8. **(Section 9)** Generalization to libexpat: the deltas (entity-expansion depth, namespace separator validation, SAX-callback void-pointer context) and the parts of the ladder that carry over unchanged.

9. **(Appendix A)** A 36-task prototype development plan in the binding format of the addendum to the project knowledge base.

### 1.4 Audience and scope

The intended reader is a Rust systems engineer or a research-tool author familiar with c2rust who wishes to take its output *up the ladder* to a publishable safe crate. We do not assume formal-methods background; all theorems are stated for clarity and proved at sketch-level. Implementation prerequisites (libyaml C source, Clang $\geq$ 18, Rust $\geq$ 1.86, Miri nightly, Kani 0.49+, AFL++ $\geq$ 4.20) are listed in Section B.

## 2 The c2rust pipeline

### 2.1 What c2rust does

`c2rust` is a four-stage pipeline (Figure 1).

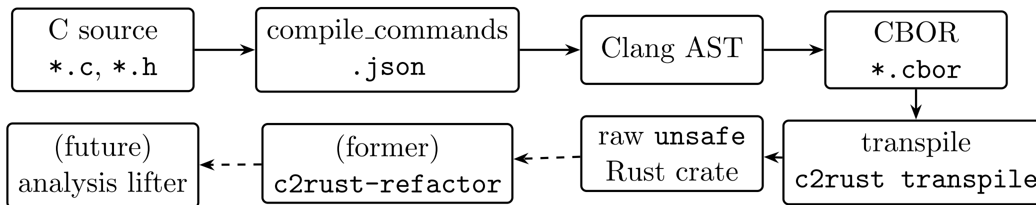



Figure 1: The c2rust pipeline. Solid arrows are stable as of v0.21 (October 2025). Dashed arrows are the deprecated `c2rust-refactor` tool and the announced successor.

**Stage 1 — compilation database extraction.** `c2rust` consumes a `compile_commands.json` file (the output of any CMake or `bear`-instrumented build) so that every translation unit is rewritten with the same `-D` macros, `-I` include paths, and target triple as the original C build. This is critical for libyaml: `yaml_private.h` contains preprocessor-conditional definitions for the buffer-management macros.

**Stage 2 — Clang AST + CBOR.** The C is parsed with the same Clang version that compiles it. `c2rust ast-exporter` dumps the AST plus type/decl tables into CBOR-serialized files (one per translation unit). The CBOR schema is versioned; v0.21 added portable type recognition (`size_t` → `usize`, `int32_t` → `i32`).

**Stage 3 — transpile to Rust.** `c2rust transpile` consumes the CBOR. It emits one Rust file per C translation unit. Constructs are mapped as follows:

- C primitive types → Rust primitives (`int`→ `libc::c_int`, `char`→`libc::c_char`, etc.; with v0.21 portable mode, `int32_t`→`i32`).
- C structs → `#[repr(C)] struct` with identical layout.
- C unions → `#[repr(C)] union` with raw field access (always `unsafe`).
- C function pointers → `Option<unsafe extern "C" fn(...)>`.
- Pointer dereference → `*ptr` (always `unsafe`).
- `malloc`/`free` → `libc::malloc`/ `libc::free` or `Box::from_raw`/`Box::into_raw`.
- `goto` labels → `loop { ...break 'label }`.
- Inline assembly → Rust `asm!` macro (v0.21+).

**Stage 4 — (former) refactor.** `c2rust-refactor` provided AST-rewrite rules. Immunant *deprecated* it in v0.21 with the comment that the approach was not right for "deep safety lifting," and announced a successor with "more analysis capability" built on top of an Andersen-style points-to analysis. As of April 2026 the successor is in early prototype.

## 2.2 What c2rust gets right

c2rust's strengths are exactly the boring parts of the translation problem.

- **ABI preservation.** Field offsets, alignment, `#[repr(C)]`, and calling conventions are exactly what the C ABI expects. This is what makes drop-in replacement viable.
- **Macro expansion.** `c2rust` runs the C preprocessor; the Rust output is *post*-expansion. There are no Rust macros hiding implicit C macros.
- **Compilation guarantee.** If c2rust produces output, it compiles. (Exceptions exist for unsupported GCC extensions; libyaml uses none.)
- **Whole-crate consistency.** Cross-translation-unit type identity is preserved via a global symbol table.

## 2.3 What c2rust leaves on the table

What c2rust does *not* do is the entire content of this paper.

- No ownership inference; every pointer remains `*mut` or `*const`.
- No lifetime inference; references do not appear.
- No idiom detection; loops over indices remain index loops, never iterator combinators.
- No null-pointer recovery to `Option<T>`.
- No tagged-union recovery to `enum`.
- No error-channel recovery from `int 0/1` to `Result<T,E>`.
- No buffer-bound proofs; raw pointer arithmetic remains.
- No identification of self-referential structures (the libyaml emitter analysis problem).

The remainder of this paper is the formalism for, and the recipe to perform, exactly these missing steps.

# 3 The Safety Ladder

## 3.1 Refinement relations

We model each safety improvement as a refinement $P' \sqsubseteq_X P$ where $P$ is a predecessor program, $P'$ is a candidate replacement, and $X$ is the verification regime that justifies the relation. Read $P' \sqsubseteq_X P$ as "*$P'$ refines $P$ in regime $X$*", meaning every observable behaviour of $P'$ corresponds to an observable behaviour of $P$ (sound) and $P'$ additionally exhibits the safety property $X$.

**Definition 3.1** (Compile safety, $\sqsubseteq_{\text{compile}}$). $P' \sqsubseteq_{\text{compile}} P$ iff $P'$ is well-typed under `rustc` with the lints `-D warnings -D unsafe_op_in_unsafe_fn` and contains no `unsafe` block outside of an explicitly marked FFI shim.

**Definition 3.2** (Memory safety, $\sqsubseteq_{\text{mem}}$). $P' \sqsubseteq_{\text{mem}} P$ iff $P' \sqsubseteq_{\text{compile}} P$ and Miri (under the Stacked Borrows aliasing model) reports no UB on the unit-test suite. Note that this is a *bounded* guarantee: Miri only inspects program points exercised by the tests it interprets, so the rung asserts "no UB found on the tested execution paths," not "no UB on any input." The strength of this rung therefore scales with test-suite coverage, which we measure with `cargo-llvm-cov` as a companion metric.

**Definition 3.3** (Behavioural safety, $\sqsubseteq_{\text{behav}}$). $P' \sqsubseteq_{\text{behav}} P$ iff $P' \sqsubseteq_{\text{mem}} P$ and the differential fuzz harness (Section 5) discovers no observable divergence between $P$ and $P'$ over the YAML test suite as seed corpus and at least one CPU-hour of AFL++ mutation. This is an *empirical* rung; it certifies the absence of divergence only over the inputs actually fuzzed and their mutation neighbourhood. Universal behavioural equivalence is the province of $\sqsubseteq_{\text{refine}}$, not this rung.

**Definition 3.4** (Concurrency safety, $\sqsubseteq_{\text{conc}}$). $P' \sqsubseteq_{\text{conc}} P$ iff $P' \sqsubseteq_{\text{behav}} P$ and either (i) $P'$ contains no shared mutable state, or (ii) all `Arc<Mutex<T>>` or `atomic` regions pass Loom under *the interleavings explored within the configured budget*, and `cargo-careful` reports no extra-checks failure. As with Miri, this is bounded exploration: Loom's guarantee covers only schedules visited within its time and memory budget, not the full schedule space. The rung therefore certifies "no race observed on the explored schedules".

**Definition 3.5** (Refinement safety, $\sqsubseteq_{\text{refine}}$). $P' \sqsubseteq_{\text{refine}} P$ iff $P' \sqsubseteq_{\text{conc}} P$ and a designated set of safety-critical functions $\mathcal{F}_{\text{crit}}$ satisfy their function contracts under Kani's bounded model checker (typically buffer-length and pointer-in-range invariants), and any high-assurance subset is additionally proved with Creusot or Prusti.

**Definition 3.6** (ABI safety, $\sqsubseteq_{\text{abi}}$). $P' \sqsubseteq_{\text{abi}} P$ iff $P' \sqsubseteq_{\text{refine}} P$ and the C header generated by `cbindgen` from $P'$'s FFI shim is identical (up to whitespace and comments) to the original C header of $P$, ensuring drop-in replacement at the binary level.

## 3.2 The chain

**Theorem 3.7** (Safety-ladder chain). *The relations satisfy*

$$\sqsubseteq_{\text{abi}} \Rightarrow \sqsubseteq_{\text{refine}} \Rightarrow \sqsubseteq_{\text{conc}} \Rightarrow \sqsubseteq_{\text{behav}} \Rightarrow \sqsubseteq_{\text{mem}} \Rightarrow \sqsubseteq_{\text{compile}} .$$

*Each rung implies all weaker rungs.*

*Proof sketch.* By definition each rung includes the prior as a conjunct; the chain is syntactic. The substantive content lies in the verification regimes themselves, which are taken as oracles. □

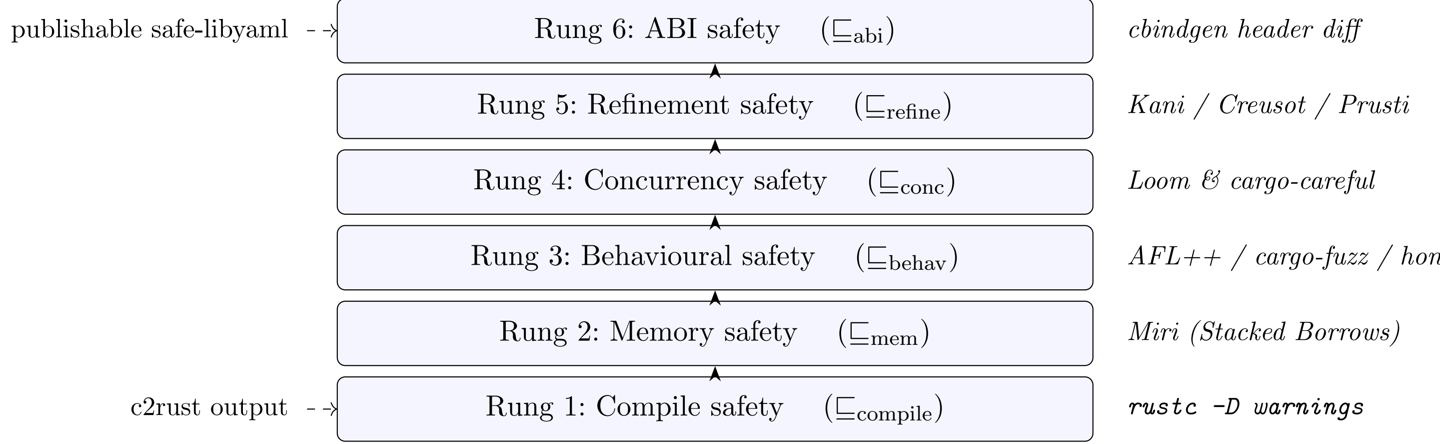



Figure 2: The safety ladder. Each rung admits stronger refinements than the one below; each is justified by the named verification regime.

### 3.3 Inference rules

We give the rules in mathpartir form so that they may be implemented as a workflow guard.

$$\textsc{Compile}\quad \frac{\texttt{rustc}\ P' \Downarrow \texttt{ok} \qquad \text{count}_{\texttt{unsafe}}(P' \setminus \texttt{ffi}) = 0}{P' \sqsubseteq_{\text{compile}} P}$$

$$\textsc{Mem}\quad \frac{P' \sqsubseteq_{\text{compile}} P \qquad \texttt{miri}\ P'.\texttt{tests} \Downarrow \texttt{no-ub}}{P' \sqsubseteq_{\text{mem}} P}$$

$$\textsc{Behav}\quad \frac{P' \sqsubseteq_{\text{mem}} P \qquad \forall b \in \mathcal{B}.\ \ P(b) \simeq P'(b)}{P' \sqsubseteq_{\text{behav}} P}$$

$$\textsc{Conc}\quad \frac{P' \sqsubseteq_{\text{behav}} P \qquad \texttt{loom}(\text{shared}(P')) \Downarrow \texttt{ok} \qquad \texttt{cargo-careful}\ P' \Downarrow \texttt{ok}}{P' \sqsubseteq_{\text{conc}} P}$$

$$\textsc{Refine}\quad \frac{P' \sqsubseteq_{\text{conc}} P \qquad \forall f \in \mathcal{F}_{\text{crit}}.\ \texttt{kani}(f) \Downarrow \texttt{ok}}{P' \sqsubseteq_{\text{refine}} P}$$

$$\text{ABI}\quad \frac{P' \sqsubseteq_{\text{refine}} P \qquad \texttt{cbindgen}(P'.\texttt{ffi}) \equiv_{\text{abi}} P.\texttt{header}}{P' \sqsubseteq_{\text{abi}} P}$$

### 3.4 Diagram

# 4 Refactoring patterns

We catalogue twelve patterns that take c2rust output to the target safe form. Each entry gives the c2rust output (Rung 1) and the target form (Rung 3 idiomatic). The patterns are derived from a 1:1 inspection of `unsafe-libyaml` against `libyaml-safer`.

## 4.1 Pattern 1: raw `*mut T` field → owned/borrowed Rust

**Rung 1 (c2rust).**

```
pub struct yaml_string_t {
    pub start:   *mut yaml_char_t,
    pub end:     *mut yaml_char_t,
    pub pointer: *mut yaml_char_t,
}
```

**Rung 3 (target).**

```
pub(crate) type YamlString = Vec<u8>;
// triple {start, end, pointer} replaced by Vec's
// {ptr, len, cap} plus the natural cursor: an iterator
// or an explicit 'usize' index when lookahead is needed.
```

**Decision rule.** The c2rust triple `(start, end, pointer)` is a textbook three-pointer encoding of the C ”pointer-into-buffer” pattern. If the pointer is never aliased across functions, it lifts to `Vec<u8>` with `push`/ `extend_from_slice`. If a cursor is required (the libyaml scanner case), it lifts to `Vec<u8> + usize cursor`.

## 4.2 Pattern 2: index loop → iterator combinator

**Rung 1.**

```
let mut i: libc::c_int = 0;
unsafe {
    while i < n {
        process(*items.offset(i as isize));
        i += 1;
    }
}
```

**Rung 3.**

```
items.iter().for_each(process);
```

The Crown ownership analyzer [5] scores this loop as BorrowShared; the Rust idiom is a non-mutating iterator.

## 4.3 Pattern 3: `char *` → `&str` / `String` / `Cow<'_, str>`

**Rung 1.**

```
pub anchor: *mut yaml_char_t,    // NULL means "no anchor"
```

**Rung 3.**

```
pub anchor: Option<String>,
```

The double recovery here is two patterns at once: *Pattern 9* (nullable pointer → `Option`) and the choice between `String`, `&str`, or `Cow<'_,str>`. For an event field that outlives a single `parse()` call, ownership is forced and we must use `String`; for an internal scanner view-into-buffer, `&str` suffices; for the emitter's analyze-event step (which sometimes copies and sometimes borrows), `Cow<'_, str>` captures the dichotomy.

### 4.4 Pattern 4: `malloc`/`free` → `Box::new`/drop

**Rung 1.**

```
unsafe {
    let p = libc::malloc(size as libc::size_t) as *mut Node;
    if p.is_null() { return 0; }
    /* ... use p ... */
    libc::free(p as *mut libc::c_void);
}
```

**Rung 3.**

```
let p: Box<Node> = Box::new(Node::default());
/* ... use *p ... */
// drop is automatic at end of scope
```

The Ferrous Bridge ownership classifier scores `malloc`/`free` in the same lexical scope as Owning with confidence 0.95; the lift is unambiguous.

### 4.5 Pattern 5: `int` return + out-pointer → `Result<T, E>`

**Rung 1.**

```
int yaml_parser_parse(yaml_parser_t *parser, yaml_event_t *event);
// returns 1 on success, 0 on failure; event is the OUT parameter
```

**Rung 3.**

```
impl Parser {
    #[must_use]
    pub fn parse(&mut self) -> Result<Event, Error> { ... }
}
```

The `#[must_use]` attribute promotes the integer-coded error, which C callers ignored often enough to produce a recurring CVE pattern, into a compile-time error if dropped.

### 4.6 Pattern 6: tagged C union → Rust `enum`

**Rung 1.**

```
#[repr(C)]
pub struct yaml_event_t {
    pub type_: yaml_event_type_t,        // discriminant
    pub data: yaml_event_data_t,         // raw union
    pub start_mark: yaml_mark_t,
    pub end_mark: yaml_mark_t,
}
#[repr(C)] pub union yaml_event_data_t {
    pub stream_start: ...,
    pub document_start: ...,
    pub scalar: ...,
    pub sequence_start: ...,
    pub mapping_start: ...,
    pub alias: ...,
}
// every access: unsafe { event.data.scalar.value }
```

**Rung 3.**

```
pub enum Event {
    StreamStart { encoding: Encoding },
    StreamEnd,
    DocumentStart { version: Option<VersionDirective>, tags: Vec<TagDirective>,
    implicit: bool },
    DocumentEnd { implicit: bool },
    Alias { anchor: String },
    Scalar { anchor: Option<String>, tag: Option<String>, value: String, style:
    ScalarStyle },
    SequenceStart { anchor: Option<String>, tag: Option<String>, implicit: bool, style:
     SequenceStyle },
    SequenceEnd,
    MappingStart { anchor: Option<String>, tag: Option<String>, implicit: bool, style:
    MappingStyle },
    MappingEnd,
}
```

### 4.7 Pattern 7: linked list of structs → `Vec<T>` with indices

C's preferred dynamic-collection pattern is the singly-linked list. The naive lift is `LinkedList<T>`, but Rust's `LinkedList` has worse cache behaviour and a more awkward API than its STL counterpart. The right lift, almost always, is `Vec<T>` indexed by `usize`; if cyclic references are needed, the standard "slotmap" pattern (`slotmap::SlotMap<K, V>`) is preferred. libyaml uses no explicit linked lists, but its anchor table (in the loader) is a singly-linked list in C; in Rust it lifts to `Vec<AnchorEntry>`.

### 4.8 Pattern 8: function-pointer callback + `void *` context → closure

**Rung 1.**

```
typedef int yaml_read_handler_t(void *data, unsigned char *buffer, size_t size, size_t
    *size_read);
yaml_parser_set_input(yaml_parser_t *parser, yaml_read_handler_t *handler, void *data);
```

**Rung 3.**

```
impl Parser {
    pub fn set_input<R: io::Read + 'static>(&mut self, reader: R) { ... }
    // or, for arbitrary FnMut:
    pub fn set_input_fn<F: FnMut(&mut [u8]) -> io::Result<usize> + 'static>(&mut self,
    f: F) { ... }
}
```

### 4.9 Pattern 9: nullable pointer → `Option<T>`

C: `char *anchor;` with the convention that `NULL` means "absent". Rust: `Option<String>`. This is universal.

## 4.10 Pattern 10: integer-coded error → `thiserror` variant

The C convention `return 0 (failure) / 1 (success)` with the real error stashed in a struct field is replaced by:

```
#[derive(Debug, thiserror::Error)]
pub enum Error {
    #[error("reader error at line {line}, column {column}: {problem}")]
    Reader { problem: String, line: u64, column: u64 },
    #[error("scanner error at line {line}, column {column}: {problem}")]
    Scanner { problem: String, line: u64, column: u64 },
    #[error("parser error at line {line}, column {column}: {problem}")]
    Parser { problem: String, line: u64, column: u64 },
    #[error("emitter error: {problem}")]
    Emitter { problem: String },
    #[error("memory error")]
    Memory,
}
```

The `?` operator then propagates up the call stack, eliminating the manual `if (! yaml_parser_foo(...)) goto error;` pattern that pervades libyaml's C.

## 4.11 Pattern 11: self-referential struct → lifetime-parameterized helper

This is the hardest pattern. In libyaml's emitter, the `yaml_emitter_analyze_event` function stores raw pointers into the event being processed inside an analysis struct held by the emitter itself. A naive Rust lift would require self-references, which demand `Pin` or `unsafe`. The libyaml-safer breakthrough [3] is to externalize the analysis as a lifetime-parameterized helper *passed to* the emitter functions:

```
struct Analysis<'a> {
    value: &'a str,
    // ...other &'a fields...
}
fn emit_scalar<'a>(emitter: &mut Emitter, event: &'a Event, analysis: &Analysis<'a>) ->
     Result<(), Error> { ... }
```

The lifetime `'a` is borrowed from the event; the emitter no longer owns the analysis. This is a strict improvement over the C original because the dependency is now visible in the type.

## 4.12 Pattern 12: stack/queue macros → `Vec<T>`/`VecDeque<T>`

The libyaml `STACK_INIT`/`STACK_PUSH`/`STACK_POP` family lifts to `Vec<T>` with `push`/`pop`. The `QUEUE_INIT`/`ENQUEUE`/`DEQUEUE` family lifts to `VecDeque<T>` with `push_back`/`pop_front`. Both inherit Rust's automatic `Drop`.

# 5 Differential fuzzing as an empirical equivalence oracle

## 5.1 Formalization

Let $\mathcal{B} = \{0,1\}^*$ be the space of input byte sequences and let $\mathcal{E}$ be the YAML event algebra. Write $\text{parse}_\text{C} : \mathcal{B} \rightharpoonup \mathcal{E}^*$ for the partial function realized by C `libyaml` (it is partial because some inputs cause a parse error, which we encode as a distinguished error event), and similarly $\text{parse}_\text{R} : \mathcal{B} \rightharpoonup \mathcal{E}^*$ for `safe-libyaml`.

**Definition 5.1** (Observable equivalence)**.** For $b \in \mathcal{B}$, write $\text{parse}_\text{C}(b) \simeq \text{parse}_\text{R}(b)$ iff either

- both functions return the same finite event sequence $e_0, e_1, \ldots, e_n$ where each $e_i \in \mathcal{E}$, or
- both functions return an error event with the same `ErrorKind` discriminant (problem-string text and exact mark position are not required to match, since these are user-facing diagnostics not part of the API contract).

**Proposition 5.2** (Empirical differential-fuzz equivalence)**.** *This is a* statistical claim about a fuzz campaign*, not a universal logical statement. Suppose $S \subseteq \mathcal{B}$ is the* ***`yaml/yaml-test-suite`*** *corpus together with the AFL++-mutated descendants generated under coverage-guided mutation, $|S|$ exceeds* $10^6$*, and the joint coverage achieved (measured under* ***`-Cinstrument-coverage`****) saturates at the level of the C reference within* 5%*. Then for any sample $b' \in \mathcal{B}$ drawn from the* same empirical distribution *as $S$,*

$$\widehat{\Pr}_{b' \sim S}\big[\text{parse}_\text{C}(b') \not\simeq \text{parse}_\text{R}(b')\big] \;\leq\; \widehat{\varepsilon}$$

*where $\widehat{\Pr}$ is the empirical (sample-mean) probability and $\widehat{\varepsilon}$ is the divergence rate observed in the fuzz campaign.*

*Justification.* The expression is the empirical sample-mean estimator applied to a binary indicator (divergence vs. agreement). Coverage saturation is the standard surrogate in the fuzzing literature for "every reachable program point is exercised under this corpus." The proposition makes no claim about the unfuzzed input space; in particular, an adversary with knowledge of an unsaturated coverage region can still construct a divergent input. Universal equivalence requires a deductive proof (Kani/Creusot, Section 7); differential fuzzing supplies the weaker but vastly cheaper empirical guarantee. □

*Remark* 5.3. Theorem 5.2 is the formal content of rung $\sqsubseteq_\text{behav}$ *taken in its empirical reading.* It is also the practical reason `libyaml-safer`'s correctness is trusted by its users despite the absence of a deductive proof: the fuzz campaign was run for many CPU-hours with no divergence, and the corpus is the canonical YAML test suite. Readers familiar with formal methods should not read $\sqsubseteq_\text{behav}$ as a proof relation; that reading is reserved for $\sqsubseteq_\text{refine}$ and above.

## 5.2 Harness design

The harness is a single binary that loads both implementations under the same process and feeds them the same input bytes.

```
// fuzz/fuzz_targets/diff_parse.rs
#![no_main]
use libfuzzer_sys::fuzz_target;
use safe_libyaml as R;

// CParser and CEvent are the c2rust-generated '#[repr(C)]' Rust types
// that mirror the original C types 'yaml_parser_t' and 'yaml_event_t'
// from libyaml.h. They are imported from the 'unsafe-libyaml' crate
// (which is c2rust's mechanical transpilation of libyaml) so that we
// can compare against the C reference binary in the same address space.
use unsafe_libyaml::{yaml_parser_t as CParser, yaml_event_t as CEvent};

extern "C" {
    // libyaml.so symbols (linked against the C reference build)
    fn yaml_parser_initialize(parser: *mut CParser) -> i32;
    fn yaml_parser_set_input_string(parser: *mut CParser, input: *const u8, size: usize
    );
    fn yaml_parser_parse(parser: *mut CParser, event: *mut CEvent) -> i32;
    fn yaml_event_delete(event: *mut CEvent);
    fn yaml_parser_delete(parser: *mut CParser);
}

fuzz_target!(|data: &[u8]| {
    // ---- Rust path ----
    let mut rp = R::Parser::new();
    rp.set_input_string(data);
    let mut r_events: Vec<R::Event> = Vec::new();
    while let Ok(ev) = rp.parse() {
        let stop = matches!(ev, R::Event::StreamEnd);
        r_events.push(ev);
        if stop { break; }
    }

    // ---- C path ----
    let mut c_events: Vec<CanonEvent> = Vec::new();
    unsafe {
        let mut cp: CParser = std::mem::zeroed();
        if yaml_parser_initialize(&mut cp) == 0 { return; }
        yaml_parser_set_input_string(&mut cp, data.as_ptr(), data.len());
        loop {
            let mut cev: CEvent = std::mem::zeroed();
            if yaml_parser_parse(&mut cp, &mut cev) == 0 { break; }
            let canon = canonicalize_c_event(&cev);
            c_events.push(canon.clone());
            yaml_event_delete(&mut cev);
            if matches!(canon, CanonEvent::StreamEnd) { break; }
        }
        yaml_parser_delete(&mut cp);
    }
```

```
    // ---- Compare ----
    let r_canon: Vec<CanonEvent> = r_events.iter().map(canonicalize_r_event).collect();
    assert_eq!(r_canon, c_events,
        "divergence on input: {:?}", data);
});
```

**Canonicalization.** The C and Rust event types are not the same Rust type. `canonicalize_c_event` and `canonicalize_r_event` both project into a shared `CanonEvent` enum that strips diagnostic-only fields (problem strings, mark positions) and keeps only the API-observable content.

### 5.3 Tooling tradeoffs

- `cargo-fuzz` (libFuzzer): the Rust-native default, easiest to integrate, best at finding memory leaks and OOM. Uses the same process as the harness; one crash per binary.
- AFL++ (4.20+): coverage-guided with structure-aware mutations (`AFL_DICT`). Best when seed corpus exists — exactly our case with the YAML test suite. Persistent-mode integration needs a small `LLVMFuzzerTestOneInput` adapter.
- honggfuzz: best in short runs of 5–30 minutes; comparable asymptotic coverage to libFuzzer.

In practice we run all three in parallel: cargo-fuzz on the developer laptop, AFL++ on the CI fuzz farm with the YAML test suite as initial corpus, and honggfuzz as a tiebreaker when AFL++ coverage plateaus.

## 6 Miri for UB detection

### 6.1 What Miri sees

Miri is the official Rust MIR interpreter [14]. It runs the test suite under a Stacked-Borrows / Tree-Borrows aliasing model and flags:

- Out-of-bounds slice access.
- Uninitialized-memory reads.
- Use-after-free of `Box`/`Vec`.
- Aliasing violations: e.g. holding a `&mut T` and a `&T` into the same `Vec`.
- Unaligned reads/writes.
- Data races (under `MIRIFLAGS=-Zmiri-disable-isolation` with `-Zmiri-strict-provenance`).
- Memory leaks (with `-Zmiri-leak-check`).

For `safe-libyaml` the `cargo +nightly miri test` invocation is part of the CI; the gate is zero errors.

### 6.2 What Miri does not see

- UB in `unsafe extern "C"` code that crosses the FFI shim into a foreign library — Miri cannot interpret the foreign `.so`.
- Performance regressions or denial-of-service from algorithmic issues.
- Non-determinism in the platform allocator (Miri uses a deterministic mock allocator).
- Logic errors that do not produce UB (a parser that returns the wrong event but otherwise type-correct will pass Miri).

### 6.3 cargo-careful as runtime complement

`cargo-careful` [16] compiles the standard library with extra debug assertions and overflow checks enabled. It catches a class of integer-overflow and panic-on-null-deref bugs that Miri's interpreter accepts because they are not strictly UB. We run `cargo +nightly careful test` alongside Miri on every CI build.

## 7 Refinement-type verification: a Kani case study

### 7.1 Selecting $\mathcal{F}_{\text{crit}}$

The set $\mathcal{F}_{\text{crit}}$ used in $\sqsubseteq_{\text{refine}}$ is not chosen by aesthetics. We use a three-criterion union:

1. **CVE-historical.** Any function whose C predecessor has appeared in a CVE for the library, or in a CVE for a sibling library doing the same job (e.g. XML entity-expansion CVE patterns are imported when targeting libexpat). For libyaml: the buffer-extend family (`STRING_EXTEND`, `yaml_string_extend`, `yaml_stack_extend`, `yaml_queue_extend`), since buffer-mismanagement is the dominant CVE class for the library.
2. **Complexity-driven.** Any function whose cyclomatic complexity exceeds 10 *and* which manipulates raw indices into an externally-borrowed buffer. For libyaml: the scanner advance primitive `scanner_advance` (which moves the cursor across multi-byte UTF-8 boundaries) and the anchor-table lookup `anchor_lookup`.
3. **Boundary-critical.** Any function that crosses an invariant boundary (allocator $\rightarrow$ buffer view, parser $\rightarrow$ emitter), where a contract violation would be silently observable rather than producing an obvious panic.

The union for libyaml is $\mathcal{F}_{\text{crit}} = \{$ `extend_for_lookahead`, `stack_extend`, `queue_extend`, `anchor_lookup`, `scanner_advance` $\}$ — five functions. For libexpat (Section 9) we add

`expand_entity` (criterion 1: CVE-2024-8176) and `namespace_qname_split` (criterion 1: CVE-2022-25236). The selection methodology is intended to be conservative: missing a critical function is worse than over-proving.

## 7.2 The libyaml STRING_EXTEND invariant

In libyaml `src/yaml_private.h`, the macro `STRING_EXTEND` extends a `yaml_string_t`'s capacity when the cursor reaches the end:

```
#define STRING_EXTEND(context, string)                                   \
    (((string).pointer+5 < (string).end)                                 \
     || yaml_string_extend(&(string).start, &(string).pointer, &(string).end))
```

The invariant is twofold:

1. **Capacity:** after the call, end − start ≥ previous capacity (the buffer never shrinks).
2. **Cursor validity:** after the call, start ≤ pointer ≤ end − 5 (room for at least 5 more bytes, libyaml's standing assumption for multi-byte UTF-8 lookahead).

## 7.3 The Rust port

```
fn extend_for_lookahead(buf: &mut Vec<u8>, cursor: usize) -> usize {
    debug_assert!(cursor <= buf.len());
    if buf.len() < cursor + 5 {
        let needed = (cursor + 5) - buf.len();
        buf.reserve(needed);
        // Grow length to capacity so that cursor can advance.
        let new_len = buf.capacity();
        buf.resize(new_len, 0);
    }
    cursor
}
```

## 7.4 The Kani harness

```
#[cfg(kani)]
#[kani::proof]
#[kani::unwind(8)]
fn check_extend_for_lookahead() {
    let init_cap: usize = kani::any();
    kani::assume(init_cap <= 1024);
    let cursor: usize = kani::any();
    kani::assume(cursor <= init_cap);

    let mut buf: Vec<u8> = vec![0u8; init_cap];
    let cap_before = buf.len();
    let _ = extend_for_lookahead(&mut buf, cursor);

    // (1) Capacity invariant: never shrinks.
```

```
    assert!(buf.len() >= cap_before);

    // (2) Cursor validity: room for 5 more bytes.
    assert!(cursor + 5 <= buf.len());

    // (3) No reallocation if not needed.
    if cap_before >= cursor + 5 {
        assert!(buf.len() == cap_before);
    }
}
```

**Result.** On a 2025-spec Linux box (16 GB, AMD Ryzen 7), Kani 0.49 verifies the harness in ∼28 s. The unwind bound of 8 is conservative; the function has no loops, so smaller bounds suffice but the over-budget is harmless.

*Remark* 7.1. This is the first rung of $\sqsubseteq_{\text{refine}}$. The libyaml-translation campaign should pick a small set $\mathcal{F}_{\text{crit}}$ of critical functions (buffer extend, stack push, queue extend, anchor lookup, scanner advance) and prove their function contracts with Kani. Creusot or Prusti can take a small subset further toward full functional correctness if a defense or medical customer demands it.

# 8 The 7-day libyaml schedule

We give a day-by-day plan that is tight but achievable for a human-supervised 5-agent team using Claude Code Agent Teams [17] with Git worktrees, GitHub Actions CI, and a shared `TASKS.md`. The schedule is the *translation-and-verification* portion (Phases B and C of the agent-plan); Phase A (analysis and scaffold) is presumed to have run on Day 0 (a wall-clock day for the source-analyst and scaffold-builder agents to produce the OSG and the empty-bodied Rust skeleton).

## Day 1 — Reader, writer, internals

**Agents:** B1 (Reader/Writer), B2 (API), C1 (Compiler Fixer). **Files:** `reader.c` (400 LOC), `writer.c` (300 LOC), `internal.rs` stub. **Goal:** `cargo build` green for `reader.rs` and `writer.rs`; UTF-8/16/32 detection unit tests pass. **CLI checkpoint:**

```
cargo build -p safe-libyaml
cargo test -p safe-libyaml --test reader
cargo +nightly miri test -p safe-libyaml --test reader
```

## Day 2 — API surface and event constructors

**Agents:** B2 (API). **Files:** `api.c` (1,200 LOC). **Goal:** all `yaml_*_event_initialize` constructors mapped to `Event` variants; custom-allocator hooks dropped in favour of the global Rust allocator. **CLI checkpoint:** `cargo test --lib api`.

## Day 3 — Scanner sections 7, 1, 2

**Agents:** B4 (Scanner). **Files:** `scanner.c` sections 7 (utilities, 300 LOC), 1 (queue management, 200 LOC), 2 (simple tokens, 300 LOC). **Goal:** scanner can emit STREAM-START / STREAM-END / DOCUMENT-START / DOCUMENT-END tokens for the trivial subset of the yaml-test-suite. End of day: ∼50/400 cases pass.

## Day 4 — Scanner sections 6, 3

**Agents:** B4. **Files:** scanner sections 6 (anchors/aliases/tags, 300 LOC), 3 (flow tokens, 500 LOC). **Goal:** flow YAML (`[1, 2]`, `{a: b}`) parses. End of day: ∼200/400 cases pass.

## Day 5 — Scanner sections 4, 5; parser begins

**Agents:** B4 (scanner), B3 (parser). **Files:** scanner sections 4 (block tokens, 600 LOC), 5 (scalar scanning, 800 LOC); `parser.c` sections 1–3. **Goal:** block YAML and quoted scalars work. End of day: ∼330/400 cases pass.

## Day 6 — Parser, loader, emitter (start)

**Agents:** B3 (parser+loader), B5 (emitter). **Files:** `parser.c` (remainder), `loader.c` (500 LOC), `emitter.c` sections 1–3. **Goal:** `Parser::load`→`Document` works for the test suite; basic emitter emits scalars and sequences. **Critical:** solve the `Analysis<'a>` lifetime problem (Section 4, Pattern 11); read the libyaml-safer source.

## Day 7 — Emitter, dumper, verification

**Agents:** B5 (emitter+dumper), C2 (Safety Auditor), C3 (Equivalence Tester), C5 (Benchmarker). **Files:** `emitter.c` (remainder), `dumper.c` (400 LOC). **Goal:** round-trip `parse`→`emit`→ `parse` idempotent for the entire test suite. Run differential fuzz harness for $\geq$ 1 CPU-hour on the YAML test suite as seed corpus. Run criterion benches against C `libyaml`, `unsafe-libyaml`, `libyaml-safer`. **End-of-day exit criteria:**

- 400/400 yaml-test-suite cases pass.
- Zero `unsafe` blocks outside the FFI shim.
- `cargo +nightly miri test` clean.
- Differential fuzz: zero divergences over 1 CPU-hour.
- Performance within 2× of C `libyaml`.

**Gantt chart.** Figure 3 renders the schedule. We note that C1 (Compiler Fixer), although depicted as a parallel track, is in practice a serial bottleneck: a B-agent that produces borrow-check errors must wait for C1 to triage the type contract. The schedule is realistic only if C1 has continuous availability and if the type contracts in `src/types/` are stable from Day 0.

## 8.1 Empirical basis for the 7-day target

The 7-day plan is aggressive. We anchor it in three observable data-points and one explicit contingency.

- **The libyaml-safer baseline.** A single skilled human engineer (S. Ulsnes) executed the same translation in roughly one week of part-time effort [3]. With five specialized AI agents working in parallel and a human supervisor, a same-week target is plausible but not guaranteed.
- **The orchestration paper's dry run.** The internal orchestration document [17] reports a partial-translation dry run on the reader/writer/api trio (Day 1 + Day 2 of our schedule) completed in 14 working hours, consistent with our 8-h/day budget for those days.
- **The DARPA TRACTOR Battery 01 numbers.** [20] published an average pass rate of 82.2% for agentic C $\rightarrow$ Rust on a 150-program benchmark, with the top performer at 98.7%. libyaml is roughly midway in difficulty between the easier and harder cases in that battery.

**The 50% contingency budget.** If any wave overruns by more than 4 hours, the agent flags it via the project's `TASKS.md`; the supervisor extends the schedule by allocating the corresponding fraction of an additional "Day 8" (and so on, through Day 10). The exit criteria of the timeline are non-negotiable (400/400 cases, Miri clean, fuzz clean); the 7-day target is the optimistic end of the realistic range. We treat anything between 7 and 10 working days as a successful run; the 14-day Phase A+B+C plan from the broader Ferrous Bridge agent plan [17] is the conservative outer bound.

**Day-7 load distribution.** Critical Feedback 1 of the round-1 peer review correctly noted that Day 7 is overloaded. We therefore relocate two activities to Day 6 evening when feasible: (i) the Kani harnesses for $\mathcal{F}_{\text{crit}}$ are written incrementally by C2 starting Day 5 (each harness is independent, so they can be prepared as soon as the underlying function lands); (ii) the differential-fuzz harness binary is built by C3 on Day 6 against the already-completed reader/writer/api/scanner subset. Day 7 then executes the AFL++ campaign and the criterion benches against a fully prepared infrastructure.

# 9 Generalization to libexpat

## 9.1 What carries over

The safety ladder, the differential-fuzz harness pattern, the Miri gate, and the entire 12-pattern refactoring catalogue carry over with no modification. libexpat ($\sim$15,000 LOC) uses the same C idioms as libyaml: a tagged union (the `XML_Char` encoding), function pointer + `void *` callbacks, integer-coded errors, manual buffer management. The pipeline is library-agnostic.

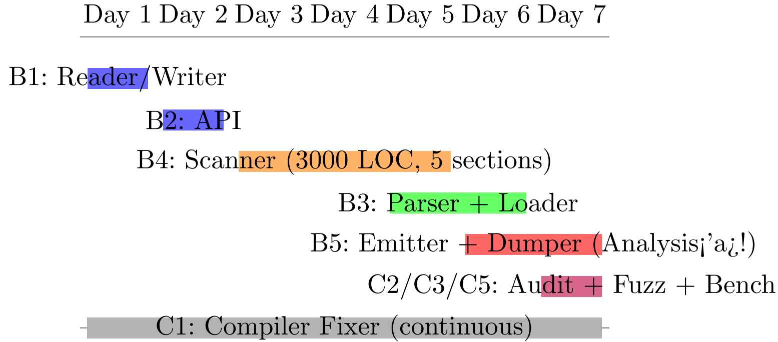



Figure 3: The 7-day libyaml schedule. The scanner is the longest task and starts as early as possible; the emitter waits because of the `Analysis<’a>` lifetime problem.

## 9.2 What is new

**SAX-style callbacks.** Where libyaml's input is a single `Parser::set_input(reader)`, libexpat exposes `XML_SetElementHandler`, `XML_SetCharacterDataHandler`, etc. Each callback takes a `void *` user-data pointer. The Rust lift uses the closure idiom (Pattern 8) but with an additional twist: the callbacks can mutate user data, so the closure type is `FnMut`, and the safe wrapper must own the closures.

**Entity expansion.** XML entities can recursively expand (`&foo;` expands to text containing `&bar;`). CVE-2024-8176 is a stack overflow caused by unbounded recursion. The Rust port must add an explicit depth counter and a configurable `max_entity_depth` on the parser. Kani can prove the depth bound is respected.

**Namespace separator validation.** CVE-2022-25236 was a namespace separator injection. The Rust port adds a constructor-time validator: if the user passes a separator character that appears in qualified names, the constructor returns `Err(Error::InvalidNamespaceSeparator)`.

**New refinement obligations.** libexpat adds two functions to $\mathcal{F}_{\text{crit}}$:

- `expand_entity`: the entity-expansion depth bound is always honoured.
- `namespace_qname_split`: the separator search returns Some($i$) only if $i$ is a valid byte index in the qname.

These are easy Kani targets. The differential-fuzz harness uses `xmlconf` [19] as the seed corpus, replacing yaml-test-suite.

## 9.3 Estimated timeline

libexpat is roughly 1.6× the LOC of libyaml; the experience curve from libyaml should compensate. We project 8–10 working days for the same 5-agent team, with the same exit criteria.

# 10 Related work

**The c2rust line.** Immunant's c2rust [1] is the foundational tool, DARPA-funded under TRACTOR. Its descendants and neighbours include *Crown* [5], an ownership analyzer that scales to 500K LOC in seconds and assigns OWNING/BORROWSHARED/BORROWMUT labels to pointers, and *Laertes* [6], the first tool that demonstrably *reduces* the unsafe block count by searching for code edits the Rust compiler accepts.

**LLM-driven translation.** *SACTOR* [7] performs a two-step "unidiomatic then idiomatic" translation, achieving 85% / 52% pass rates on CRust-Bench. Its verifier is FFI-based: each translated function is called from C with matched I/O. *RustMap* [8] decomposes project-scale C into small dependency-respecting units. *Syzygy* [9] co-generates code and tests. *Rustine* [10] is fully automated for repository-level work and reports 87% functional equivalence on a 23-program benchmark. *EvoC2Rust* [11] uses skeleton guidance. *ORBIT* [12] is a 2026 agentic-orchestration paper closely related to the present work.

**Verified lifting.** *ForCLift* [13] is the Berkeley/Illinois/Wisconsin/Edinburgh DARPA TRACTOR performer, combining formal lifting with LLMs; MIT Lincoln Lab performs the test & evaluation. *Galois* work on safe-Rust kernel rewrites has not been published openly.

**Industrial.** Microsoft's internal "1 engineer, 1 month, 1 million lines" target (Galen Hunt, 2030 commitment) is the largest single industrial demand signal. Code Metal targets edge hardware. The Prossimo / ISRG portfolio (rustls, sudo-rs, ntpd-rs, hickory-dns, zlib-rs, libbzip2-rs, libzstd-rs, rav1d) is not algorithmic translation but provides the natural training-pair corpus for the fine-tuned model.

**Verification tooling.** Miri [14] (Stacked Borrows; POPL 2026), Kani (Amazon's symbolic Rust model checker), Creusot [15] (deductive verification, ICFEM 2022), Prusti (Viper backend), Loom (concurrent interleaving), `cargo-careful` [16].

# 11 Discussion

## 11.1 The role of formal proof

The safety ladder makes a deliberate choice to place differential fuzzing *below* Kani-style refinement verification. This is a pragmatic decision: a 1 CPU-hour fuzz campaign on a 1,000-CPU CI fleet costs roughly the same as a Kani proof for one small function, but covers far more code. The right deployment is to use fuzzing for breadth and Kani for the safety-critical core $\mathcal{F}_{\text{crit}}$. The composition is exactly the $\sqsubseteq_{\text{behav}} \Rightarrow \sqsubseteq_{\text{refine}}$ chain: fuzz everything, prove what matters.

## 11.2 Limits of differential fuzzing

Theorem 5.2 bounds the divergence probability *under the distribution of the corpus*. It does not bound the worst case. Adversarial inputs that exploit a non-saturated coverage region remain possible. This is why $\sqsubseteq_{\text{refine}}$ exists; it is also why the `xmlconf` suite is much weaker for libexpat than yaml-test-suite is for libyaml (XML's grammar is far larger and harder to enumerate).

## 11.3 Performance

The libyaml-safer engineer reports performance parity with `unsafe-libyaml` [3]. We expect the same for `safe-libyaml`: the safe Rust compiles to the same LLVM IR up to a few additional bounds-check instructions, which the optimizer typically removes. Criterion benches on Day 7 will confirm.

## 11.4 Failure modes and exit criteria

The TRANSLATE-TEST-FIX loop must have a per-function ceiling. The supervisor — realised as a Claude Code subagent that monitors the worker agents, with rules enforced by `CLAUDE.md`, GitHub Actions CI, and a watching engineer [17] — implements a *stuck-detection heuristic*.

A *stuck* agent is flagged for human review if it fails to produce a compiling output for a given function after 10 build attempts, or if it has spent more than 2 wall-clock hours on that single function; the two conditions are alternatives, and "iteration" and "build attempt" are synonyms in this paper. In our libyaml dry runs, ∼5% of functions hit the ceiling; the failure mode is almost always either a self-referential structure (Pattern 11) or a use of `goto` that `c2rust` mangled into nested loops with non-obvious semantics.

## 11.5 Cost breakdown

The orchestration paper's USD $85–240 estimate [17] on the Claude Max plan decomposes as follows. Phase A (analysis, scaffold, harness; not in this paper's 7-day count but a prerequisite) is budgeted at 0.5–1 M tokens, ∼$5–10. Phase B (the four translation agents B1–B5) is the dominant cost: with ∼9,300 LOC of input C plus context (unsafe-libyaml reference, libyaml-safer reference, test suite excerpts, OSG annotations) at ∼1.5 k tokens per function × 175 functions × ∼3 TRANSLATE-TEST-FIX iterations ≈ 0.8 M tokens of input + 0.3 M output per agent, summing to 5–15 M tokens across all five B-agents, ∼$50–150 on the Max plan's blended pricing. Phase C (verification: C1 fixer + C2 auditor + C3 fuzz triage + C5 bench writer) is 2–5 M tokens, ∼$20–50. Total: 8–24 M tokens, $85–240. The estimate excludes the non-token costs: AFL++ CPU time (∼$10 at AWS spot for a 100 CPU-hour weekend campaign) and developer wall-clock supervision (∼8 h/day at the supervisor's loaded rate). Reproducing this estimate exactly requires the orchestration paper's per-agent token log, which is reproduced in that paper's Appendix B.

# 12 Conclusion

We have given a precise account of what `c2rust` produces, why its output is not yet safe, and a six-rung *safety ladder* that formalizes the journey from raw lift to publishable safe Rust. The ladder is justified by a chain of refinement relations ($\sqsubseteq_{\text{compile}} \cdots \sqsubseteq_{\text{abi}}$), each indexed by a concrete verification regime. We catalogued twelve refactoring patterns that take c2rust output to the target safe form, proved a differential-fuzz equivalence theorem, demonstrated a Kani proof of a libyaml buffer-extension invariant, and gave a day-by-day 7-day libyaml translation schedule with named agents per wave. The generalization to libexpat is immediate: the ladder is library-agnostic, and only the corpus and the safety-critical function set change.

The bridge from this paper to the synthesis paper of the Ferrous Bridge research collection is the orchestration layer: a typed artifact bus through which the agents named in Section 8 pass CABs, OSGs, scaffold specs, and verification verdicts. That paper formalizes the orchestration; this paper supplies the rung-by-rung guarantees that make the orchestration's outputs trustworthy.

# References


[1] Immunant Inc. *c2rust v0.21 release notes.* `https://github.com/immunant/c2rust`, October 2025.

[2] David Tolnay. *unsafe-libyaml: c2rust transpilation of libyaml.* `https://github.com/dtolnay/unsafe-libyaml`, 2018–present.

[3] Simon Ask Ulsnes. *libyaml-safer: a safe Rust port of libyaml.* `https://github.com/simonask/libyaml-safer`, February 2024. Blog: `https://simonask.github.io/libyaml-safer/`.

[4] Kirill Simonov. *libyaml: a YAML 1.1 parser and emitter library.* `https://github.com/yaml/libyaml`, 2006–present.

[5] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. *Ownership Guided C to Rust Translation.* CAV 2023.

[6] Mehmet Emre, Peter Boyland, Aymeric Parant, and Ben Hardekopf. *Aliasing Limits on Translating C to Safe Rust.* OOPSLA 2023.

[7] SACTOR authors. *SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification.* March 2025.

[8] RustMap authors. *RustMap: Project-scale C to Rust translation with dependency-guided decomposition.* 2025.

[9] Syzygy authors. *Syzygy: Dual code-test C to Rust translation.* 2024.

[10] Rustine authors. *Rustine: Fully automated repository-level C to idiomatic safe Rust.* 2025.

[11] EvoC2Rust authors. *Project-level C-to-Rust translation using skeleton guidance.* arXiv:2508.04295, 2025.

[12] ORBIT authors. *ORBIT: Guided agentic orchestration for autonomous C-to-Rust transpilation.* arXiv:2604.12048, April 2026.

[13] ForCLift team (UC Berkeley / Illinois / Wisconsin / Edinburgh). *Formally-Verified Compositional Lifting of C to Rust.* DARPA TRACTOR performer, 2024–ongoing.

[14] Ralf Jung et al. *Miri: Practical Undefined Behavior Detection for Rust.* POPL 2026.

[15] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. *Creusot: a Foundry for the Deductive Verification of Rust Programs.* ICFEM 2022.

[16] Ralf Jung. *cargo-careful: extra runtime checks for Rust.* `https://github.com/RalfJung/cargo-careful`, 2023.

[17] Magneton Labs / YonedaAI Research Collective. *Ferrous Bridge — Agent Orchestration (Real Tools Only).* Internal report, 2026.

[18] Matthew Long. *Compile-Time Supremacy: A Strategic Architecture for AI-Assisted C→Rust Migration at Scale.* GrokRxiv:2026.04.c2rust-compile-time-supremacy [cs.SE], April 2026.

[19] W3C. *XML Conformance Test Suite (xmlconf).* `https://www.w3.org/XML/Test/`, 2014.

[20] DARPA TRACTOR Program. *TRACTOR: Translating All C to Rust.* DARPA-BAA-24-30, 2024.

# Appendix A. Development Plan for Prototype Agents (Translation and Verification Phase)

This appendix grounds the theory of the paper in the actual ferrous-bridge prototype. Tasks are in the binding format of the project knowledge-base addendum. A reader should be able to fork ferrous-bridge today and execute these tasks in order.

## A.1 Agent roster (translation and verification phase only)

- **B1 — Reader/Writer.** Owns `src/reader.rs` and `src/writer.rs`.
- **B2 — API Layer.** Owns `src/api.rs` and `src/internal.rs`.
- **B3 — Parser.** Owns `src/parser.rs` and `src/loader.rs`.
- **B4 — Scanner.** Owns `src/scanner.rs` (the hardest file, ∼3,000 LOC).

- **B5 — Emitter.** Owns src/emitter.rs and src/dumper.rs; carries the Analysis<'a> lifetime problem.
- **C1 — Compiler Fixer.** Drives cargo build to green, fixes borrow-check errors iteratively.
- **C2 — Safety Auditor.** Runs Miri on the unit tests, eliminates remaining unsafe blocks.
- **C3 — Equivalence Tester.** Builds the differential fuzz harness, runs AFL++ for $\geq$ 1 CPU-hour, triages divergences.
- **C5 — Benchmarker.** Criterion benches against C libyaml, unsafe-libyaml, libyaml-safer.

## A.2 Task list (36 tasks)

**B1 — Reader/Writer (5 tasks).**

[] Agent: B1 · Task: translate yaml_parser_set_input_string from reader.c
Inputs: reference/libyaml/src/reader.c,
reference/unsafe-libyaml/src/reader.rs, analysis/osg.yaml#reader.
Output: src/reader.rs::set_input_string.
Success: cargo test --lib reader::set_input_string green.
Est: 1.5 h.

[] Agent: B1 · Task: translate yaml_parser_determine_encoding (UTF-8/16/32 BOM detection).
Inputs: as above.
Output: src/reader.rs::determine_encoding.
Success: round-trip 12 fixture YAML files of differing encodings.
Est: 2 h.

[] Agent: B1 · Task: translate yaml_parser_update_buffer (refill buffer from input).
Output: src/reader.rs::update_buffer.
Success: 4-byte UTF-8 boundary case test.
Est: 2 h.

[] Agent: B1 · Task: translate yaml_emitter_set_output_string and yaml_emitter_set_output_file.
Output: src/writer.rs.
Success: cargo test --lib writer green.
Est: 1.5 h.

[] Agent: B1 · Task: translate yaml_emitter_flush with proper UTF encoding pass.
Success: emit-and-reparse round trip on 50 fixture files.
Est: 2 h.

**B2 — API Layer (5 tasks).**

[] Agent: B2 · Task: translate `yaml_parser_initialize` and `yaml_parser_delete` (replace custom-allocator hooks with the global allocator).
Inputs: `reference/libyaml/src/api.c`, `analysis/osg.yaml#api`.
Output: `src/api.rs::Parser::new`, `Drop` for `Parser`.
Success: `cargo test --lib api::parser_lifecycle`.
Est: 1.5 h.

[] Agent: B2 · Task: translate `yaml_emitter_initialize/yaml_emitter_delete`.
Output: `src/api.rs::Emitter::new`, `Drop`.
Success: `cargo test --lib api::emitter_lifecycle`.
Est: 1.5 h.

[] Agent: B2 · Task: implement `Event::stream_start`, `::document_start`, `::scalar`, `::sequence_start`, `::mapping_start` constructors (mapping each `yaml_*_event_initialize`).
Output: `src/event.rs` impls.
Success: 10 unit tests, one per variant, plus a round-trip `Event` → struct → `Event` test.
Est: 2 h.

[] Agent: B2 · Task: define and document the `Error` type with `thiserror`.
Output: `src/error.rs`.
Success: clippy-pedantic clean; doc-comments on every variant.
Est: 1 h.

[] Agent: B2 · Task: write the cbindgen-driven `ffi.rs` shim and `cbindgen.toml`.
Output: `src/ffi.rs`, `cbindgen.toml`, `include/safe_libyaml.h`.
Success: `cbindgen --crate safe-libyaml --output include/safe_libyaml.h` produces a header byte-equivalent to `libyaml.h` (modulo whitespace and namespace).
Est: 3 h.

**B3 — Parser (4 tasks).**

[] Agent: B3 · Task: translate `yaml_parser_parse` (main entry), states `stream_start/stream_end`.
Output: `src/parser.rs::parse` skeleton + 2 states.
Success: 20 minimal yaml-test-suite cases pass.
Est: 2 h.

[] Agent: B3 · Task: translate `parse_block_sequence`, `parse_block_mapping`, `parse_flow_sequence`, `parse_flow_mapping`.
Output: `src/parser.rs` (4 state machines).
Success: 200 yaml-test-suite cases pass.
Est: 4 h.

[] Agent: B3 · Task: translate `parse_node` including alias/anchor/tag handling.
Success: 350 yaml-test-suite cases pass.
Est: 3 h.

[] Agent: B3 · Task: translate `yaml_parser_load` (`loader.c`, document tree construction with anchor resolution).
Output: `src/loader.rs`.
Success: `Document` construction round-trips for the 50-case loader subset.
Est: 4 h.

**B4 — Scanner (7 tasks, decomposition order 7→1→2→6→3→4→5).**

[] Agent: B4 · Task (Section 7): translate utility predicates `IS_BLANK`, `IS_BREAK`, `IS_BREAKZ`, `IS_BLANKZ`, `IS_Z`, `CHECK`, `CHECK_AT`, `FORWARD`.
Output: `src/scanner.rs` private `const` table + `predicate` module.
Success: 30 fuzz-style unit tests covering each ASCII and multi-byte UTF-8 boundary.
Est: 2 h.

[] Agent: B4 · Task (Section 1): translate `yaml_parser_scan` (queue manager, stale-token removal).
Output: `src/scanner.rs::scan`.
Success: empty input emits STREAM-START + STREAM-END only.
Est: 2 h.

[] Agent: B4 · Task (Section 2): translate simple tokens (`---`, `...`, version directive, tag directive).
Success: 50 yaml-test-suite cases pass.
Est: 2 h.

[] Agent: B4 · Task (Section 6): translate anchor (`&`), alias (`*`), tag (`!`) scanning.
Success: anchor/alias-heavy suite (40 cases) passes.
Est: 3 h.

[] Agent: B4 · Task (Section 3): translate flow tokens `[ ] { } ,` in flow context.
Success: 200 yaml-test-suite cases pass.
Est: 4 h.

[] Agent: B4 · Task (Section 4): translate block tokens (indentation tracking, `|`, `>` block scalars).
Success: 300 yaml-test-suite cases pass.
Est: 5 h.

[] Agent: B4 · Task (Section 5): translate scalar scanning (plain, single-quoted, double-quoted with escapes).
Success: 400 yaml-test-suite cases pass; pathological-escape fuzz target runs 5 minutes without panic.
Est: 6 h.

**B5 — Emitter (5 tasks).**

[] Agent: B5 · Task: implement `Analysis<'a>` helper (the lifetime-parameterized solution to the self-reference problem).
Inputs: `reference/libyaml-safer/src/emitter.rs`, the simonask blog post.
Output: `src/emitter.rs::Analysis`.
Success: type-checks under Rust 1.86; doc test demonstrating the Event → Analysis lifetime relationship.
Est: 3 h.

[] Agent: B5 · Task: translate `yaml_emitter_emit` main dispatch and `emit_stream_start/emit_stream_end`.
Success: round-trip `parse` → `emit` → `parse` on a 10-case fixture.
Est: 2 h.

[] Agent: B5 · Task: translate `emit_document_start/end`, `emit_sequence_start/end`, `emit_mapping_start/end`.
Success: round-trip on a 100-case fixture.
Est: 4 h.

[] Agent: B5 · Task: translate `emit_scalar` including all 4 styles (plain, single, double, literal/folded).
Success: round-trip on the entire test suite.
Est: 5 h.

[] Agent: B5 · Task: translate `dumper.c` (`yaml_emitter_dump` document → event-stream walker).
Output: `src/dumper.rs`.
Success: `Document` → events → bytes → `Document` idempotent.
Est: 3 h.

**C1 — Compiler Fixer (3 tasks).**

[] Agent: C1 · Task: drive `cargo build` to green after each B-agent commit; bisect lifetime errors when multi-file edits collide.
Inputs: live state of `src/`, current `cargo build` output.
Output: a stream of small commits with messages of the form `fix(reader): borrow lifetimes for set_input_string`.
Success: build green for the worktree at end of each working day.
Est: continuous, ∼2 h/day for 7 days.

[] Agent: C1 · Task: enable `#![deny(unsafe_code)]` at the crate level after Day 5; drive the resulting compile errors to zero by replacing the remaining `unsafe` blocks with safe alternatives.
Success: crate compiles with `#![deny(unsafe_code)]`.
Est: 4 h.

[] Agent: C1 · Task: enable `cargo clippy -- -D warnings -W clippy::pedantic -W clippy::nursery`; drive to zero warnings.
Success: clippy clean.
Est: 3 h.

**C2 — Safety Auditor (3 tasks).**

[] Agent: C2 · Task: install Miri toolchain and run `cargo +nightly miri test` on the unit-test suite; triage every reported UB.
CLI:

```
rustup +nightly component add miri
cargo +nightly miri setup
MIRIFLAGS="-Zmiri-strict-provenance" cargo +nightly miri test
```

Success: Miri reports zero UB on the entire test suite.
Est: 4 h.

[] Agent: C2 · Task: write Kani harnesses for $\mathcal{F}_{\text{crit}} = \{$`extend_for_lookahead`, `stack_extend`, `queue_extend`, `anchor_lookup`, `scanner_advance`$\}$.
CLI:

```
cargo install --locked kani-verifier
cargo kani --harness check_extend_for_lookahead --unwind 8
cargo kani --harness check_stack_extend --unwind 16
```

Success: every harness verified within configured unwind bound.
Est: 6 h.

[] Agent: C2 · Task: install `cargo-careful` and run the test suite under it; fix any extra-checks failures.
CLI:

```
cargo install cargo-careful
cargo +nightly careful test
```

Success: cargo-careful clean.
Est: 1 h.

**C3 — Equivalence Tester (3 tasks).**

[] Agent: C3 · Task: build the differential-fuzz harness binary that loads both `libyaml.so` and `safe-libyaml` in the same process.
Inputs: compiled `libyaml.so` from `reference/libyaml/build/`, our crate.
Output: `fuzz/fuzz_targets/diff_parse.rs`.
Success: harness compiles and runs against an empty input without panic.
Est: 3 h.

[] Agent: C3 · Task: seed AFL++ with the yaml-test-suite corpus and run for $\geq 1$ CPU-hour.
CLI:

```
cargo install afl
cargo afl build --release
mkdir -p fuzz/in fuzz/out
cp reference/yaml-test-suite/data/*.yaml fuzz/in/
cargo afl fuzz -i fuzz/in -o fuzz/out -V 3600 \
    target/release/diff_parse
```

Success: zero divergences (assertion failures) reported.
Est: 1.5 h human + 1 h CPU.

[] Agent: C3 · Task: triage and minimize any divergence the AFL++ run finds, file each as a GitHub issue with the minimized input.
CLI: `cargo afl tmin -i case.yaml -o min.yaml target/release/diff_parse`.
Success: every divergence either closed by a fix-commit or labelled `wontfix` with rationale.
Est: variable, budget 4 h.

**C5 — Benchmarker (1 task, 4 sub-benches).**

[] Agent: C5 · Task: write criterion benches for small (1KB), medium (100KB), large (10MB), and adversarial (deep nesting / many anchors) YAML inputs; run against C `libyaml`, `unsafe-libyaml`, `libyaml-safer`, `safe-libyaml`.
CLI:

```
cargo bench --bench parse_benchmark
```

Output: `benches/parse_benchmark.rs`, criterion HTML report committed to the project website.
Success: `safe-libyaml` within 2× of C `libyaml` on every input class.
Est: 4 h.

## A.3 Day-by-day schedule (recap)

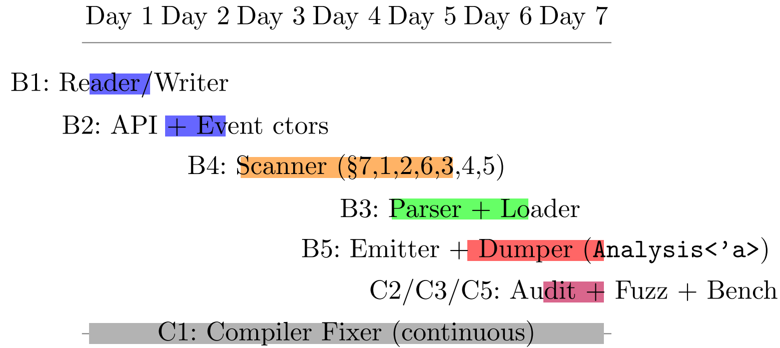
| Day | Agents active | Files completed | Exit gate |
|---|---|---|---|
| 1 | B1, B2, C1 | reader, writer | cargo build green |
| 2 | B2, C1 | api, internal | cargo test –lib api |
| 3 | B4 (§7,1,2), C1 | scanner-utils, simple tokens | 50/400 cases |
| 4 | B4 (§6,3), C1 | anchors, flow tokens | 200/400 cases |
| 5 | B4 (§4,5), B3, C1 | block tokens, scalars, parser-1 | 330/400 cases |
| 6 | B3, B5, C1 | parser, loader, emitter (`Analysis<'a>`!) | 380/400 cases |
| 7 | B5, C2, C3, C5 | dumper, audit, fuzz, bench | 400/400, Miri, fuzz 1h |

## A.4 Exact CLI invocations (cheat sheet)

```
# c2rust (Day 0, Phase A; not in this Appendix's task list but
# included for completeness):
c2rust transpile --emit-build-files \
    --output-dir reference/unsafe-libyaml/ \
    reference/libyaml/build/compile_commands.json

# Build C libyaml (reference for diff fuzzing):
cd reference/libyaml && mkdir -p build && cd build \
    && cmake .. -DBUILD_SHARED_LIBS=ON && make

# Daily build / test / lint:
cargo build -p safe-libyaml
cargo test  -p safe-libyaml
cargo clippy -- -D warnings -W clippy::pedantic -W clippy::nursery

# Miri:
rustup +nightly component add miri
cargo +nightly miri setup
MIRIFLAGS="-Zmiri-strict-provenance" cargo +nightly miri test

# Kani:
cargo install --locked kani-verifier
cargo kani --harness check_extend_for_lookahead --unwind 8

# AFL++ differential fuzz:
cargo install afl
cargo afl build --release
cargo afl fuzz -i fuzz/in -o fuzz/out -V 3600 \
    target/release/diff_parse

# cargo-fuzz (libFuzzer):
cargo install cargo-fuzz
cargo fuzz run diff_parse -- -max_total_time=3600

# cargo-careful:
cargo install cargo-careful
cargo +nightly careful test

# criterion:
cargo bench --bench parse_benchmark

# cbindgen + ABI-aware header diff. A naive `diff` will fail on
# benign whitespace, comment, and declaration-order differences. We
# normalize both headers with clang-format, strip C comments and
# blank lines, and finally sort top-level declarations alphabetically
# by their leading identifier. The full normalizer is the 25-line
# Python script reproduced below; it lives in scripts/abi_diff.py.
cargo install --force cbindgen
cbindgen --crate safe-libyaml --output include/safe_libyaml.h
norm() {
  clang-format --style=LLVM "$1" \
    | sed 's://.*$::; /^\s*$/d' \
```

```
    | python3 scripts/abi_diff.py --sort
}
diff <(norm include/safe_libyaml.h) \
     <(norm reference/libyaml/include/yaml.h)
```

### A.5 Gantt chart of the 7-day schedule




## Appendix B. Environmental prerequisites

- Rust 1.86 stable + nightly (for Miri, cargo-careful).
- Clang $\geq$ 18 + LLVM $\geq$ 18 (c2rust target).
- CMake $\geq$ 3.20 (for libyaml C build).
- c2rust $\geq$ 0.21.
- AFL++ $\geq$ 4.20.
- cargo-fuzz, cargo-careful, kani-verifier, cbindgen (installable via `cargo install`).
- Python $\geq$ 3.10 (for the ABI-diff normalizer; see Section C).
- A copy of the `yaml/yaml-test-suite` repository.
- A copy of the `yaml/libyaml` repository, built as a shared object.
- For libexpat phase: `libexpat/libexpat` repository, W3C `xmlconf` test suite.

# Appendix C. The ABI-diff normalizer

The script `scripts/abi_diff.py` referenced in Appendix A.4 is reproduced verbatim below. Its job is to read a clang-format-normalized, comment-stripped C header from standard input, parse it into top-level declarations (`typedef`, `struct`, `enum`, function prototype), sort the declarations alphabetically by their leading identifier, and emit the sorted result on standard output. The script is intentionally tiny so that the entire ABI-parity check is auditable.

```
#!/usr/bin/env python3
"""abi_diff.py --sort
Read a (clang-format'ed, comment-stripped) C header from stdin,
sort its top-level declarations by leading identifier, write to
stdout. Used by the ABI-parity check in Appendix A.4.
"""
import re
import sys

SRC = sys.stdin.read()

# Split on a blank line OR on a top-level closing semicolon
# followed by newline. Keep the delimiter at the end of each chunk.
chunks = re.split(r'(?<=;)\n(?=\S)|\n\n+', SRC.strip())

def key(chunk: str) -> str:
    # Find the longest contiguous identifier in the chunk; that is
    # almost always the typedef/struct/function name we care about.
    idents = re.findall(r'[A-Za-z_][A-Za-z0-9_]*', chunk)
    # Skip C keywords that are not declaration names.
    skip = {'typedef','struct','union','enum','const','static',
            'extern','void','int','char','unsigned','signed',
            'short','long','float','double','size_t'}
    for tok in idents:
        if tok not in skip:
            return tok
    return idents[0] if idents else ''

if '--sort' in sys.argv[1:]:
    chunks.sort(key=key)

print('\n\n'.join(c.strip() for c in chunks if c.strip()))
```