Verifying RustCrypto's Fixsliced AES-128 in ACL2: An Aeneas Backend and an Executable MIR Semantics*

Eric McCarthy
Kestrel Institute https://kestrel.edu

Aeneas translates safe Rust programs into pure functional code by symbolically resolving borrows, with existing backends for F*, Rocq, HOL4, and Lean. We present an ACL2 backend for Aeneas, together with supporting changes to Aeneas and to its Rust front end Charon, and use it to verify a real-world cryptographic implementation: the RustCrypto project's 32-bit fixsliced AES-128, extracted essentially verbatim from the upstream crate. We prove that the extracted encryption and decryption functions, key schedule included, equal the existing ACL2 AES-128 specification for all keys and blocks, with every bounds check and assertion proven to pass. The proofs combine GL bit-blasting for bit-level facts about the bitsliced S-box and state packing with structured rewriting for the round ladders' mask and frame algebra, over fueled recursive models of the upstream loops. To reduce the trust placed in the unverified Rust-to-ACL2 translation chain, we also present an executable MIR semantics in ACL2 covering every operation the crate uses, with a loader for the compiled crate's MIR: the interpreter runs the same AES inside ACL2, reproduces the FIPS-197 vectors, and agrees with the same AES-128 specification on differential tests, independently of the Aeneas translation — a step toward translation validation of the extraction.

1 Introduction

Rust has become a language of choice for security-critical infrastructure: its ownership type system rules out memory errors in safe code by construction. Memory safety, however, is not functional correctness, and the verification of Rust programs has become an active industrial concern—the Rust standard library now runs a verification campaign of over 16,000 proof harnesses per code change in continuous integration [6]. Nearly all of this tooling either verifies Rust in place, with specifications embedded in the source, or extracts it to other proof assistants (Section 7). ACL2 [20], with its executable specifications, industrial-strength rewriting, and an existing library of cryptographic specifications—including an AES-128 specification already used to verify JVM bytecode and x86 implementations [31, 32]—has met Rust only for programs written to fit a restricted subset designed for transpilation [11, 12]; it has had no general path from the ecosystem's existing code.

This paper builds that path and walks it end to end, making three contributions. First, an **ACL2 backend for Aeneas** [14], the borrow-resolving translator from safe Rust to pure functional code, together with supporting changes to Aeneas and its front end Charon [13]: extraction to an untyped, first-order target, with a monomorphization carve-out, defunctionalized iterators, and fueled loop models (Section 3). Second, a **verified real-world cipher**: the RustCrypto project's 32-bit fixsliced AES-128, vendored essentially verbatim from the upstream crate under a mechanical audit (Section 4), with both extracted directions—encryption and decryption, key schedule included—proved equal to the ACL2 AES-128 specification for all keys and blocks, every assertion and bounds check proved to pass (Section 5); our

*Substantial parts of this paper, and much of the work it describes — the Aeneas ACL2 backend, the AES-128 proof development, and the MIR interpreter and loader — were produced by Claude Fable 5, an AI assistant from Anthropic, working under the author's direction; see Section 1.

Submitted to:
ACL2-2026

© E. McCarthy
This work is licensed under the
Creative Commons Attribution License.

emphasis throughout is the ACL2 path—the audit discipline, the proof architecture, and membership in the same-specification lineage as the JVM and x86 verifications. Third, an **executable MIR semantics** in ACL2 with a loader for the compiled crate's MIR, giving the same program an independent second reading that reproduces the FIPS-197 vectors and agrees with the same specification on differential tests (Section 6).

The three artifacts compose into an assurance case (Figure 1): one path proves the extracted program correct; the other path checks the compiled program independently of Aeneas and our backend, sharing only `rustc` and Charon's front end; and the remaining trusted components are explicit, each with a named plan to discharge it (Section 8). Throughout, we emphasize reusable method over one-off proof: the mechanical vendoring audit, the division of labor between GL bit-blasting and structured rewriting, and the probe-then-prove discovery loop are all portable to the next crate and the next algorithm.

In accordance with the workshop's guidance on the use of AI assistants: the Aeneas backend, the proof development, and the MIR semantics and loader described in this paper were developed by Claude Fable 5, an AI assistant from Anthropic, working under the author's direction; the author specified goals, reviewed the work, and certified and integrated all results. This paper was drafted the same way. The supporting materials—all books, certifiable from scratch—accompany the submission (Section 9).

2 Background

2.1 Rust, borrows, and purity

Rust manages memory without garbage collection through *ownership*: every value has one owner, other code sees it only through borrows—shared references (`&T`, read-only, many at once) or mutable references (`&mut T`, exclusive)—and the compiler's borrow checker enforces the discipline statically. The payoff for verification is that safe Rust, despite its imperative surface, is observationally close to a functional language: exclusive mutation with no aliasing means a mutable update can be read as producing a new value. That reading is exactly what Aeneas [14] makes precise, resolving borrows symbolically and emitting pure functions—mutation becomes value threading, and a mutable borrow passed into a call comes back as an ordinary result.

2.2 The pipeline: `rustc`, MIR, Charon, Aeneas

`rustc` lowers source through several intermediate representations to MIR, the mid-level intermediate representation on which borrow checking and optimization run: a small control-flow-graph language of statements, terminators, and places, with monomorphized types available. Charon [13] extracts MIR from the compiler and serializes it (as ULLBC, and structured into LLBC) for external analyzers; Aeneas consumes LLBC, performs its borrow-resolving symbolic execution, and hands each backend a pure AST. Prior backends target F*, Rocq, HOL4, and Lean; Section 3 describes ours. Because Aeneas does not establish termination, recursive functions are extracted with fuel.

2.3 The ACL2 AES-128 specification

The verification target is the AES-128 specification in the ACL2 community books (`kestrel/crypto/aes`): an executable, audited rendering of FIPS-197 [25], covering both cipher directions and the key expansion. It is not new, and that is the point—the same specification has anchored Axe-based verifications

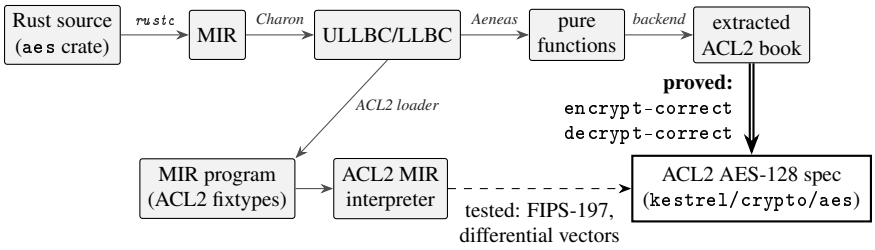


Figure 1: Two paths from the Rust source to the ACL2 AES-128 specification. Gray arrows are trusted (unverified) translations; the doubled arrow is the machine-checked equivalence for all keys and blocks; the dashed arrow is differential testing. The two paths share `rustc` and Charon's translation to its serialized IR; neither Aeneas nor the backend is on the second path.

of JVM bytecode and x86 implementations of AES [31, 32], so a proof against it places extracted Rust in an existing family of verified implementations (Section 7).

3 An ACL2 Backend for Aeneas

Aeneas [14] ends its borrow-resolving translation with a pure abstract syntax tree: functions and algebraic data types with no references, no lifetimes, and no mutation, in which the effects Rust expresses through panics—failed bounds checks, failed assertions, integer overflow—appear as explicit error results. Our backend prints this AST as ACL2. Each crate becomes one generated book; each Rust function becomes a `defun` written in a monadic `b*` style over a `result` sum with `ok` and failure constructors, so that a Rust function of type `fn(...) -> T` whose execution can panic becomes an ACL2 function returning either `(ok v)` or a failure value that all callers propagate. For example, the one-line upstream rotation helper

```
fn rotate_rows_1(x: u32) -> u32 { ror(x, ror_distance(1, 0)) }
```

extracts as

```
(defun aes-fixslice-encrypt-rotate-rows-1 (x)
  (b* (((ok v3) (aes-fixslice-encrypt-ror-distance 1 0)))
  (aes-fixslice-encrypt-ror x v3)))
```

with names systematically derived from the crate and function path.

3.1 The Rust primitives model

Generated books include a handwritten book, `rust-primitives`, that gives first-order ACL2 meanings to the Rust value layer the extracted code manipulates: machine integers of each width with checked arithmetic (`u32-add` returns `ok` on in-range sums and a failure otherwise, with rules characterizing both cases), total bitwise operations, monadic shifts (a shift amount at or beyond the width is an arithmetic overflow, a panic under the overflow checks in force here), rotations, and truncating and widening casts; arrays and slices as lists with bounds-checked monadic `array-index` and `array-update`; range values as structures; and the `Result`/`Option` plumbing of the standard library. Upstream `debug_assert!`s extract as `massert` forms— `ok`-binders on the asserted condition—so a verified theorem about an extracted function establishes that every assertion and bounds check along the way succeeds (Section 5).

3.2 Loops and fuel

Aeneas does not prove termination, so recursive functions—including the recursive loop functions produced from upstream `loop { ... break }` text—take an explicit fuel argument and return a distinguished out-of-fuel failure when it is exhausted. Our correctness theorems instantiate a concrete fuel that the proofs show sufficient: the theorems conclude an `ok` result, so fuel exhaustion is ruled out along with every other failure. The AES crate exercises this on its three upstream loop shapes: the key schedule's `rcon` fold, the encrypt round loop, and the decrypt round loop with its descending offset counter and mid-body `break`.

3.3 Iterators, defunctionalized

The structurally hardest extraction class in the crate is slice iteration: upstream `shift_rows_*` and `add_round_key` are `for` loops over `iter()`, `iter_mut()`, and `zip`. For mutable iteration Aeneas's translation threads *backward functions*: the loop's translated state includes a continuation that writes the mutated elements back into the slice, so a loop formal has arrow type and is not directly first-order. The printer handles this in two steps. First, it models the opaque iterator types first-order: `Iter` and `IterMut` as products of the underlying list and a cursor position, `Zip` as a pair of iterators, with the constructor and `next` operations synthesized against those models. Second, it *defunctionalizes* the loop's write-back continuation into a list of pending writes—the identity continuation becomes `nil`, each wrapping lambda becomes a `cons`—applied by generated companion functions: after k `next`s the cursor sits at position k, and applying the pending writes latest-first lands each value in the slot its `next` read. A dedicated probe crate in the test suite pins this extraction shape and its proof treatment independently of AES.

3.4 Supporting changes to Charon and Aeneas

ACL2's first-order world wants monomorphic code, and Charon [13] can monomorphize—but indiscriminate monomorphization bakes out instantiations that carry borrows (such as `Option<&mut u32>` and the slice iterator types), erasing region information that Aeneas's analysis needs and causing it to reject the input. We therefore run a carve-out configuration, contributed to our Charon fork: types capable of carrying mutable borrows stay polymorphic while everything else monomorphizes, with associated types lifted and ADT clauses removed so the result still prints first-order. The AES crate exercises this on exactly the iterator shapes above; the configuration is general and a candidate for upstreaming.

4 Case Study: RustCrypto's Fixsliced AES-128

4.1 Fixslicing

The RustCrypto `aes` crate's [30] portable software fallback is a *fixsliced* implementation, a constant-time bitsliced representation due to Adomnicai and Peyrin [2]. The 32-bit variant processes two blocks per call: the 256 bits of two AES states are transposed into eight 32-bit words, one per bit position, so that the S-box becomes a fixed Boolean circuit over the words and the whole cipher runs without data-dependent memory access. Classical bitslicing must still permute the representation for ShiftRows every round; fixslicing instead *fixes* the slice layout and rotates the round operations around it. The per-round standalone ShiftRows disappears: the state simply drifts one ShiftRows frame per round, the implementation provides four conjugated variants `mix_columns_0` through `mix_columns_3` that each absorb the current frame, and the cycle closes every four rounds. What remain, and do permute

the words, are boundary and key-schedule corrections: `shift_rows_2` before the final round, and its inverses at decryption's entry and on the key schedule's windows. The key schedule is pre-distorted to match, storing round key j rotated into its frame. The S-box circuit has 113 gates—32 ANDs, 77 XORs, and 4 XNORs; the implementation realizes them as 32 `&` and 81 `^` operations, absorbing the XNORs' inversions into the key schedule—so the stored keys also carry a compensating byte mask, cancelled at each `add_round_key`. Decryption runs the same machinery in reverse, with the inverse S-box circuit taking the mask compensation on its input instead of its output. All of this—bit-level packing, frame and mask distortions threaded through ten rounds and an 88-word key schedule, iterator-based inner loops, subslice borrows—is what makes the crate an exacting subject for extraction and proof, and a representative one: it is real, shipped, optimized cryptographic Rust, not code written to be verified.

4.2 Provenance and mechanical audit

Our subject is vendored by hand from `aes` v0.9.1, `src/soft/fixslice32.rs` (repository `RustCrypto/block-ciphers`, tag `aes-v0.9.1`, commit `507938c`), and the pristine upstream file is checked in beside it so the adaptation is diffable forever. The vendored copy began life with several de-sugarings that the early toolchain required; as backend and Charon support landed, a sequence of eleven restoration passes replaced each de-sugared form with the verbatim upstream text—the re-rolled encrypt and decrypt round loops, the `iter_mut()/zip` loops, the key schedule's subslice borrows and `step_by` fold, upstream's own function signatures, and all thirteen upstream `debug_assert!`s. A script-verified, whitespace- and comment-insensitive, function-by-function audit maintains the current delta classes. At this writing the shared round-operation and key-schedule bodies are statement-identical to upstream; the residue is the adaptation layer around them. The batch drivers, the key schedule's signature, and one `inv_bitslice` initializer replace the types of the `cipher` crate dependency we do not vendor with plain byte arrays (upstream's `Array` blocks and the `FixsliceKeys128` alias; `default()` initializers written out), and four added single-block entry points—`encrypt`, `encrypt_block`, `decrypt`, `decrypt_block`, the functions the theorems name—wrap a batch driver, duplicating the one block into both lanes and returning lane 0. Dropped scope: AES-192/256, the `cipher`-crate trait implementations, and the untaken `cfg` path. The theorems are about the adapted copy; they transfer to shipped RustCrypto exactly modulo these documented deltas.

4.3 Extraction

The crate goes through the carve-out pipeline of Section 3.4:

```
charon rustc --preset=aeneas --monomorphize \
    --monomorphize-mut=except-types --remove-adt-clauses \
    --lift-associated-types='*' \
    -- --edition=2021 aes_fixslice_encrypt.rs --crate-type=rlib
aeneas -backend acl2 -use-fuel -loops-to-rec aes_fixslice_encrypt.llbc
```

From the 863-line vendored source this produces one generated book of 1,852 lines defining 74 functions, including the extracted single-block entry points the main theorems are stated about: `encrypt` and `decrypt` each run the extracted `aes128_key_schedule` and then the two-lane block function, mirroring how the upstream trait implementations invoke the same internals. The generated book certifies as ordinary ACL2, with no axioms and no trust tags; every handwritten proof book of Section 5 builds on it unchanged.

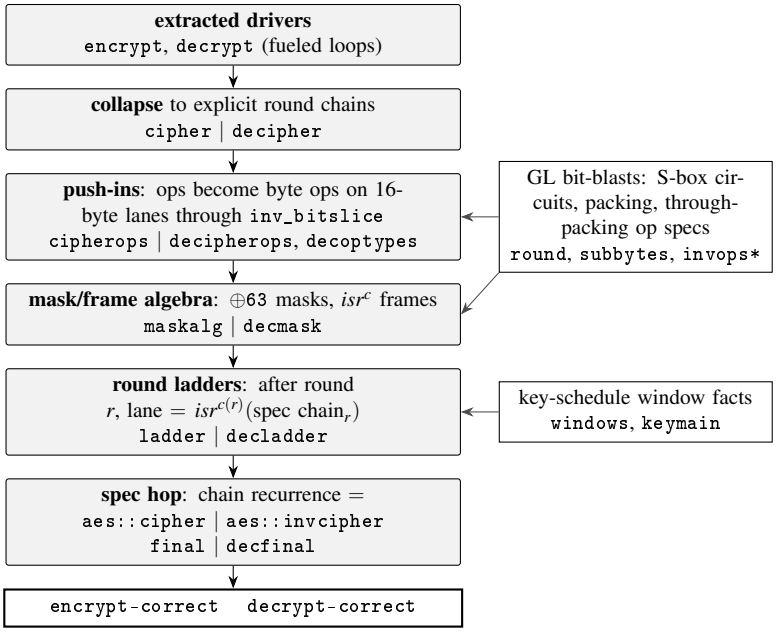


Figure 2: The proof architecture: each layer names its job and the encrypt | decrypt books that implement it. Bit-level facts enter once, as GL bit-blasts, at the push-in and algebra layers; everything above and below is structured rewriting.

5 Correctness Proofs

The main theorems equate the extracted `encrypt` and `decrypt`—key schedule included—with the audited AES-128 specification in the ACL2 community books, for all keys and blocks:

```
(defthm encrypt-correct
  (implies (and (aes::inp key) (aes::inp block))
           (equal (aes-fixslice-encrypt-encrypt 100 key block)
                  (ok (aes::aes-128-encrypt block key)))))

(defthm decrypt-correct
  (implies (and (aes::inp key) (aes::inp block))
           (equal (aes-fixslice-encrypt-decrypt 100 key block)
                  (ok (aes::aes-128-decrypt block key)))))
```

Here `aes::inp` recognizes arbitrary 16-byte blocks, and `100` is the concrete fuel. The subjects are the crate's single-block entry points (Section 4.2): each expands the key, runs the two-block batch driver with its block in both lanes, and returns lane 0, so the equalities are single-block statements, with the two-lane internals covered by component lemmas along the way. Because the conclusion is an `ok` result, the theorems also establish that no execution path fails: every one of the thirteen upstream assertions, every bounds check, and every fuel decrement succeeds, for every input.

5.1 Architecture

The proof is a stack of layers (Figure 2), each stated in the vocabulary the layer below establishes, with the decrypt stack a structural mirror of the encrypt stack. The *collapse* layer turns each fueled driver

into an explicit chain of round operations: the recursive loop function at fuel 100 is expanded at each of its descending call fuels under a pinned minimal theory, so ten rounds unroll deterministically without the rewriter exploring anything else. The *push-in* layer moves reasoning from the packed representation to bytes: for each round operation `op`, a rule $\texttt{inv_bitslice}(\texttt{op}(s)) = \textit{op-bytes}(\texttt{inv_bitslice}(s))$ rewrites the chain into per-lane byte transformations, where each *op-bytes* is the operation's byte-level specification. The *algebra* layer normalizes the two fixslice distortions of Section 4: mask lemmas move the S-box constant between state, keys, and round operations, and frame lemmas telescope one ShiftRows per round through the conjugated MixColumns variants. The *ladder* layer proves the per-round invariant—after round r, a lane of the un-bitsliced state is the byte-level specification chain in the ShiftRows frame of round r—as nine explicit round lemmas plus a final one, no induction needed. The *spec hop* layer, finally, is a fact about the Kestrel specification alone: its cipher loop, read through the flat 16-byte representation, is exactly the ladder's chain recurrence, with the extracted key schedule's windows already bridged to `aes::keyexpansion`.

5.2 GL where bits live, rewriting where structure lives

Two proof styles carry the stack, with a deliberate division of labor. Facts that are true *because of bit-level computation*—what the 113-gate S-box circuit computes through the packing, that `bitslice` and `inv_bitslice` are inverses, what each conjugated MixColumns variant does to a lane of bytes—are proved by GL [33] bit-blasting, as symbolic BDD computations over all 2^{256} packed states or all 2^{128} byte lanes. Everything else is structured rewriting under pinned theories: collapses, push-in folds, mask and frame algebra, ladders, and spec hops never open a bit-level definition.

The blasts required their own engineering. The forward MixColumns facts certify with variables in natural order, but the conjugated *inverse* MixColumns facts—coefficients 9, 11, 13, 14 rather than 1, 2, 3—exhausted memory at every book split and clearing discipline we tried. The fix was a variable *ordering*: for the variant conjugated into frame c, interleaving the sixteen byte variables along the frame's diagonals makes each output column read contiguous BDD variables, and the same blasts that died at seven gigabytes certify in under a minute. Memory discipline matters elsewhere too: heavy blasts live in books with the slimmest possible include chains, memoization tables are cleared between them, and the heaviest blast in a session is sequenced last.

5.3 The decrypt mirror

Decryption mirrors the encrypt stack layer for layer, but the mask flows the other way, and the difference is instructive. On the encrypt side the state and the stored round keys both carry the S-box constant, and the two cancel at each `add_round_key`. On the decrypt side the state is mask-free: each `add_round_key` *deposits* the key's mask, and the following `inv_sub_bytes` *consumes* it, because the inverse S-box circuit expects the constant on its input. So where the encrypt algebra cancels masks across an exclusive-or, the decrypt algebra associates the mask outward, commutes it through the conjugated inverse MixColumns (whose coefficients also sum to 1, so the uniform mask is fixed), and cancels it into the true inverse S-box map. The frames likewise telescope in reverse, and the final round key window is stored unmasked and unrotated—so the last `add_round_key` lands lane 0 on the specification's output exactly.

5.4 Probe, then prove

The invariants above were discovered before they were proved. For each round operation, and then for each state of the round chains, we ran the extracted code and the specification on FIPS-197 data and matched the extracted value against a table of candidate spec stages under every frame rotation and mask combination. The match table read off the entire ladder invariant—including the frame schedule and exactly where masks appear and vanish—after which each equation became a lemma to certify, by GL if bit-level and by rewriting otherwise. The first run of the decrypt probe was also the extracted decrypt's first end-to-end known-answer test. We found this *probe, then prove* loop fast and robust: executability turned invariant discovery into observation, and no discovered equation later failed to certify.

5.5 The suite

The complete development is 78 books: the generated book, the handwritten proof stack, and books certifying FIPS-197 known-answer vectors by execution—the Appendix C.1 cipher vector in both directions with round trips, and the Appendix A.1 key-expansion example against `aes::keyexpansion`. All books are in logic mode with no axioms, no `skip-proofs`, and no trust tags, and the suite recertifies from scratch in about twelve minutes with two parallel jobs in our development environment.

6 An Executable MIR Semantics

The verification of Section 5 is only as trustworthy as the translation chain that produced the extracted functions. Our second line of assurance gives MIR itself a formal semantics in ACL2, so the same compiled program can be read a second way, independently of Aeneas and our backend; the two readings share `rustc` and Charon's translation of MIR into its serialized IR (Figure 1).

6.1 The MIR subset and its semantics

We formalize, as fixtypes and functions in the ACL2 community books, a subset of MIR sufficient for the AES crate: values, places and their projections, rvalues, statements, terminators, bodies, and a machine state. The interpreter is small-step at the statement and terminator level, with big-step evaluation of operands, places, and rvalues, which are functional. Coverage was chosen empirically from the crate's own MIR, and the interpreter handles a superset of what the crate exercises: six terminator kinds (`Call`, `Assert`, `Return`, `SwitchInt`, `Goto`, `Unreachable`, plus the `Abort` that Charon produces for diverging panic calls), `Assign` statements, eight rvalue forms, and twelve binary operators including the overflow-checked pairs that debug assertions compile to. Thirty-three named standard-library functions are given ACL2 models, twenty-five of which the imported AES program uses—wrapping arithmetic, rotations, little-endian byte conversions, slice indexing with its panics, the slice iterators including `zip`, `TryInto`, and panic machinery modeled as abort states. The model also fixes its operating assumptions: a 64-bit target for `usize`, abort-on-panic with unwind edges dropped at import, moves that read without deinitializing, and no-op drops—each adequate for this crate, whose state is `Copy` throughout. The semantics is executable, and its pieces were exercised by dedicated probe programs as they were added—development checks, not part of the committed suite.

6.2 Loading compiled MIR

The interpreter runs the actual crate. Plain Charon—no monomorphization, none of the extraction pipeline's reshaping flags—translates the crate's MIR into its unstructured IR and serializes it as 6.9 MB of ULLBC JSON, normalization passes included (storage markers, pointer metadata, unit returns); an ACL2 importer turns that into a program value. The importer first expands Charon's serialization sharing (hash-consed values with one global counter in document order; 19,249 markers in the AES dump), then maps JSON into the MIR fixtypes with nil-tolerant navigator functions made measure-transparent by a handful of linear lemmas—a reusable pattern for termination over semi-structured input. A classifier names crate functions by identifier and standard-library declarations by implementation self-type and trait; because nothing is monomorphized, one generic `Index` implementation serves every slice-index call, so shim names are finalized per call site from the call's generic arguments. The whole import runs in under a second; an import test book parses the committed sample at certification time and checks that the result is exactly the program fixture the interpreter tests run on.

6.3 Running AES

The interpreter runs the imported crate end to end: `encrypt` reproduces the FIPS-197 ciphertexts of Appendix C.1 and Appendix B byte for byte, and `decrypt` inverts both. The vector constants are the ones the specification's own test book uses. Two further inputs, all zeros and all ones, are checked differentially: on both, the interpreted `encrypt` agrees with `aes::aes-128-encrypt` itself, the same specification function the theorems of Section 5 equate the Aeneas extraction to. One interpreted encryption takes about 0.27 seconds.

Live execution also exposed one real semantic gap, in iterator coercion: `add_round_key` zips against a bare slice, relying on the standard library's `zip` to convert it to an iterator (`IntoIterator`) internally—a coercion the interpreter had not modeled. It also drove the `Abort` terminator and slice-window dereference paths that unit probes had not reached. The fix came with a new probe test.

6.4 What this adds to the assurance case

The two readings of the compiled crate agree: one is proved equal to the specification for all inputs, the other reproduces the specification on the tested vectors. The readings share `rustc` and Charon's translation to ULLBC, normalization passes included, but neither Aeneas's borrow resolution nor our backend; agreement on the tested vectors is therefore evidence against errors in the unshared stages—evidence on those inputs, not a proof for all of them. The point generalizes beyond extraction: automation-first tools meet Rust at the same level—Kani, for instance, compiles MIR proof harnesses to CBMC through a translation that is itself trusted [6]—so an independently tested MIR semantics serves the ecosystem broadly, not only our pipeline. The remaining steps are to validate the MIR semantics itself by differential testing against `rustc`'s own MIR interpreter, and to replace testing by proof— either proving the interpreted MIR equal to the extracted functions (translation validation), or proving it equal to the specification directly (Section 8).

7 Related Work

Rust verification is an active field. RustBelt [16] gave the language's ownership discipline its foundational semantic model. Most practical tools verify Rust *in place*, with specifications embedded in the

source: Prusti [3] builds on Viper, Creusot [7] translates to Why3, and Verus [22] extends Rust itself with a verification dialect; Kani [6] model-checks Rust by compiling MIR proof harnesses to CBMC, with function and loop contracts extending its bounded core, and RefinedRust [9] derives foundational proofs in Iris. Miri [17], `rustc`'s own MIR interpreter, serves the ecosystem as a de facto dynamic semantics—and serves us as the differential-testing target for ours. A second school *extracts* Rust into a prover's native language. The prior path into ACL2 is Hardin's Restricted Algorithmic Rust (RAR) [11, 12]: programs are written in a restricted subset of Rust, transpiled source-to-source into Restricted Algorithmic C, and reach ACL2 through the RAC translator of industrial floating-point practice, an approach demonstrated on array-backed data structures, cryptographic routines, and Knuth's Dancing Links. Where RAR asks the program to be written in the subset, our pipeline consumes what the ecosystem already ships, borrows included. In the same school, hax [4] translates a restricted Rust subset for security-critical code (mutation is supported; mutable borrows are restricted), and Aeneas [14], on which we build, handles borrows themselves by translating them away, with backends for F*, Rocq, HOL4, and Lean. Our backend adds an untyped, first-order target to that family, which is what forces the monomorphization carve-out and the iterator defunctionalization of Section 3—and what buys ACL2's executability and industrial-strength rewriting for the proofs.

Verified AES implementations have a long history. Ours joins a lineage attached to this same ACL2 specification: with the Axe toolkit [31, 32], JVM bytecode and x86 implementations of AES were proved equivalent to it, and the present work adds fixsliced Rust as a third verified implementation family (with Axe over our MIR semantics as a prospective fourth route, Section 8). Elsewhere, SAW [8] verifies AES implementations against Cryptol specifications by symbolic execution, and projects like HACL* [37] take the complementary clean-slate approach, writing new cryptographic code in verification-first languages. Our distinguishing constraint is that the subject is shipped, third-party, optimization-heavy code, verified essentially as-is under a mechanical audit against upstream.

Rust and MIR have also been formalized directly. In the K framework [28], two contemporaneous 2018 semantics made source-level subsets executable—KRust [35] and K-Rust, since grown into RustSEM [19], whose ownership-and-borrowing model is validated against the compiler on around 700 tests—and KMIR [29] now gives the compiler's Stable MIR JSON export a K semantics with concrete and symbolic execution, of the existing efforts the nearest relative of our loader-plus-interpreter. Calculi such as Oxide [36] and Featherweight Rust [26] distill the borrow checker for metatheory. The Rust project itself maintains MiniRust [18], a precise specification pitched just above MIR, and `a-mir-formality` [23], a model of MIR and the trait system, while the Stacked and Tree Borrows aliasing models [15, 34] supply the rules Miri enforces on unsafe code.

Finally, the MIR interpreter continues ACL2's long tradition of executable formal semantics used for code proofs—the x86 ISA [10], RISC-V [21], and the Yul intermediate language [5] being recent and classic examples—brought here to a compiler's internal representation, with translation validation [27] as the intended endgame for connecting the two readings of a program.

8 Future Work

Two engineering items continue the current arc. *Complete de-vendoring*: whole-crate extraction would retire the vendored copy entirely, and the residual driver-signature delta of Section 4.2 dissolves once the `cipher`-crate API can be consumed. *Guard verification*: the suite is logic mode with no axioms, `skip-proofs`, or trust tags, but the generated books declare no guards; emitting real `:guard` annotations (for example, `unsigned-byte-p` argument types) and discharging the call-site obligations would add

well-formedness checking and execution efficiency.

The larger program strengthens Figure 1 arrow by arrow. The MIR semantics itself will be validated by differential testing against Miri [17], `rustc`'s own MIR interpreter. Testing then gives way to proof: using the Axe equivalence checker [31] to prove the interpreted MIR for AES equal to the ACL2 AES-128 specification would upgrade the figure's dashed arrow to a doubled one—Axe's third AES representation, after JVM bytecode and x86—and proving the interpreted MIR equal to the extracted functions (translation validation [27]) would remove Aeneas and our backend from the trusted base per program.

Finally, the chain extends beyond MIR in both directions. Above it, a formal model of the Rust front end is underway—executable, tested specifications of Rust's lexical grammar and lexer, including token trees, exist today—toward independently checking `rustc`'s translation from source to MIR. Below it, Axe can lift the *compiled* AES binary from RISC-V machine code to logic, using ACL2's executable RISC-V model [21], and prove that equal to the same specification—checking `rustc`'s back half just as the front-end model checks its front half. The same AES-128 specification then anchors verified readings of the same program at every level from source to binary.

9 Conclusion

There is now a working path from shipped, third-party Rust code to ACL2 theorems, and it has been walked end to end on a demanding subject: a shipped, optimization-heavy, bitsliced AES-128, verified in both directions against a specification with a decade of prior use, with an executable MIR semantics providing an independent reading of the same compiled program. Beyond the theorems, we hope the reusable pieces travel: the mechanical vendoring audit as a discipline for verifying third-party code essentially as-is, the GL-versus-rewriting division of labor with its memory craft, and the probe-then-prove loop that turns executable specifications into an invariant-discovery instrument.

Supporting materials accompany the submission, and the whole pipeline is reproducible from a pinned toolchain. The extraction and the complete book suite—the generated book, the proof stack, and the known-answer tests—build and certify from scratch inside a published development-environment image [24] that fixes the exact Charon fork, Aeneas branch, and ACL2 build together with the Kestrel certified books; re-running the extraction there reproduces the generated books byte-for-byte, and the full suite then certifies. The MIR semantics and importer, with the serialized crate sample and its end-to-end tests, are part of the ACL2 community books' Rust development (`master`, commit `ada59446`, `books/kestrel/rust`) [1]. Appendix A records the exact identifiers, the commands, the expected output, and the measured cost.

References

[1] ACL2 Community: *ACL2 Community Books*. `https://github.com/acl2/acl2`.

[2] Alexandre Adomnicai & Thomas Peyrin (2020): *Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V*. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021(1), pp. 402–425, doi:10.46586/tches.v2021.i1.402-425. Artifact available at `https://artifacts.iacr.org/tches/2021/a6`.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli & Alexander J. Summers (2019): *Leveraging Rust Types for Modular Specification and Verification*. *Proceedings of the ACM on Programming Languages* 3(OOPSLA), doi:10.1145/3360573.

[4] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch & Bas Spitters (2025): *hax: Verifying Security-Critical Rust Software Using Multiple Provers*. In: *Verified Software: Theories, Tools and Experiments (VSTTE 2024)*, *LNCS* 15525, Springer, doi:10.1007/978-3-031-86695-1_7.

[5] Alessandro Coglio & Eric McCarthy (2025): *A Formalization of the Yul Language and Some Verified Yul Code Transformations*. *Electronic Proceedings in Theoretical Computer Science* 423. Proceedings of the 19th ACL2 Workshop.

[6] Rémi Delmas, Zyad Hassan, Qinheping Hu, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Adrián Palacios, Celina Val, Michael Tautschnig, Justus Adam, Daniel Schwartz-Narbonne & Carolyn Zech (2026): *Kani: A Model Checker for Rust*. ArXiv:2607.01504; to appear at ASE 2026, Industry Showcase.

[7] Xavier Denis, Jacques-Henri Jourdan & Claude Marché (2022): *Creusot: A Foundry for the Deductive Verification of Rust Programs*. In: *Formal Methods and Software Engineering (ICFEM 2022)*, Springer, pp. 90–105, doi:10.1007/978-3-031-17244-1_6.

[8] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee & Aaron Tomb (2016): *Constructing Semantic Models of Programs with the Software Analysis Workbench*. In: *Verified Software: Theories, Tools, and Experiments (VSTTE 2016)*, *LNCS* 9971, Springer, doi:10.1007/978-3-319-48869-1_5.

[9] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers & Derek Dreyer (2024): *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*. *Proceedings of the ACM on Programming Languages* 8(PLDI), doi:10.1145/3656422.

[10] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2017): *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. In: *Provably Correct Systems*, Springer.

[11] David S. Hardin (2022): *Hardware/Software Co-Assurance using the Rust Programming Language and ACL2*. *Electronic Proceedings in Theoretical Computer Science* 359, pp. 202–216, doi:10.4204/EPTCS.359.16.

[12] David S. Hardin (2023): *Verification of a Rust Implementation of Knuth's Dancing Links using ACL2*. *Electronic Proceedings in Theoretical Computer Science* 393, pp. 161–174, doi:10.4204/EPTCS.393.13.

[13] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz & Jonathan Protzenko (2025): *Charon: An Analysis Framework for Rust*. In: *Computer Aided Verification (CAV)*, doi:10.1007/978-3-031-98685-7_18.

[14] Son Ho & Jonathan Protzenko (2022): *Aeneas: Rust Verification by Functional Translation*. *Proceedings of the ACM on Programming Languages* 6(ICFP), doi:10.1145/3547647.

[15] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang & Derek Dreyer (2020): *Stacked Borrows: An Aliasing Model for Rust*. *Proceedings of the ACM on Programming Languages* 4(POPL), doi:10.1145/3371109.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers & Derek Dreyer (2018): *RustBelt: Securing the Foundations of the Rust Programming Language*. *Proceedings of the ACM on Programming Languages* 2(POPL).

[17] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer & Qian Wang (2026): *Miri: Practical Undefined Behavior Detection for Rust*. *Proceedings of the ACM on Programming Languages* 10(POPL), doi:10.1145/3776690.

[18] Ralf Jung et al.: *MiniRust: A Precise Specification for "Rust lite/MIR plus"*. `https://github.com/minirust/minirust`.

[19] Shuanglong Kan, Zhe Chen, David Sanán & Yang Liu (2024): *Formally Understanding Rust's Ownership and Borrowing System at the Memory Level*. *Formal Methods in System Design* 64, pp. 200–236, doi:10.1007/s10703-024-00460-3.

[20] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

[21] Carl Kwan (2025): *RV32I in ACL2*. *Electronic Proceedings in Theoretical Computer Science* 423, pp. 46–50, doi:10.4204/EPTCS.423.4.

[22] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno & Chris Hawblitzel (2023): *Verus: Verifying Rust Programs using Linear Ghost Types*. *Proceedings of the ACM on Programming Languages* 7(OOPSLA1), doi:10.1145/3586037.

[23] Nicholas D. Matsakis et al.: *a-mir-formality: A Model of MIR and the Rust Type/Trait System*. `https://github.com/rust-lang/a-mir-formality`.

[24] Eric McCarthy (2026): *A Reproducible Development Environment for the Aeneas ACL2 Backend*. `https://github.com/bendyarm/aeneas-acl2-devenv`, doi:10.5281/zenodo.22365956. Prebuilt image `ghcr.io/bendyarm/aeneas-acl2-dev`, built by continuous integration from pinned toolchain commits.

[25] National Institute of Standards and Technology (2001): *Advanced Encryption Standard (AES)*. FIPS PUB 197.

[26] David J. Pearce (2021): *A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust*. *ACM Transactions on Programming Languages and Systems* 43, doi:10.1145/3443420.

[27] Amir Pnueli, Michael Siegel & Eli Singerman (1998): *Translation Validation*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS* 1384, Springer.

[28] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[29] Runtime Verification (2023): *Introducing KMIR: Concrete and Symbolic Execution of Rust MIR*. `https://runtimeverification.com/blog/introducing-kmir`. Semantics at `https://github.com/runtimeverification/mir-semantics`.

[30] RustCrypto Developers: *RustCrypto: `aes` crate, version 0.9.1*. `https://github.com/RustCrypto/block-ciphers`.

[31] Eric Whitman Smith (2011): *Axe: An Automated Formal Equivalence Checking Tool for Programs*. Ph.D. thesis, Stanford University. `https://www.kestrel.edu/research/axe/`.

[32] Eric Whitman Smith & David L. Dill (2008): *Automatic Formal Verification of Block Cipher Implementations*. In: *Formal Methods in Computer-Aided Design (FMCAD 2008)*, IEEE, doi:10.1109/FMCAD.2008.ECP.10.

[33] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. *Electronic Proceedings in Theoretical Computer Science* 70, pp. 84–102, doi:10.4204/EPTCS.70.7.

[34] Neven Villani, Johannes Hostert, Derek Dreyer & Ralf Jung (2025): *Tree Borrows*. *Proceedings of the ACM on Programming Languages* 9(PLDI), doi:10.1145/3735592.

[35] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu & Jun Zhang (2018): *KRust: A Formal Executable Semantics of Rust*. In: *Theoretical Aspects of Software Engineering (TASE 2018)*, IEEE, pp. 44–51, doi:10.1109/TASE.2018.00014.

[36] Aaron Weiss, Olek Gierczak, Daniel Patterson, Amal Ahmed & Nicholas D. Matsakis (2019): *Oxide: A Formal Foundation for Rust's Borrow Checker*. ArXiv:1903.00982.

[37] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko & Benjamin Beurdouche (2017): *HACL*: A Verified Modern Cryptographic Library*. In: *ACM Conference on Computer and Communications Security (CCS)*.

A Artifact: Reproducing the Results

The artifact is a Linux (amd64) container image with the proof path of Figure 1 built and pinned: Aeneas with the ACL2 backend, the Charon fork with the monomorphization carve-out (Section 3.4), their Rust and OCaml toolchains, and ACL2 with Kestrel's certified community books. It is built by continuous integration from the repository of [24], which also provides scripts for using the image without Docker;

the v1.0.0 release is archived on Zenodo, DOI `10.5281/zenodo.22365956`. The image records its own provenance in `/etc/aeneas-dev-manifest`:

```
aeneas = 23cf8c88b7a13ee26b3c6bea772d494b98ef64ce  (bendyarm/aeneas, acl2-backend)
charon = 58b7d543566f212cd2dabd723b06ccc23ea34075  (bendyarm/charon: upstream pin
                                                    340b1af4 + the carve-out)
acl2   = 0612757bb56f5491f2f4bab76da18fdab4ec93b6  (4,492 certified books; SBCL 2.6.1)
rustc  = nightly-2026-06-01    ocaml = 5.3.0
image  = ghcr.io/bendyarm/aeneas-acl2-dev:aeneas-23cf8c8_charon-58b7d54_acl2-0612757
digest = sha256:1931376654f7d49ce6c52bc1914ce3ea9e606491a46f1cfca65d2dff049389ae
```

The verification subject and all books live in the Aeneas repository. `tests/acl2` holds the 19 generated books (one per test crate; `aes_fixslice_encrypt.lisp` is the whole cipher, both directions, and the rest are extraction probes and two TEA examples), the 489-line handwritten runtime `rust-primitives` (Section 3.1), and 58 handwritten proof books, 42 of which are the AES stack of Section 5, topped by `encrypt-correct` and `decrypt-correct`. The vendored subject and its pristine upstream reference sit in `tests/src`, with the audit ledger (`PORTING-NOTES-ACL2.md`, at the repository root) and its delta classes documented as in Section 4.2.

Reproducing. With Docker, `docker run -it` on the image above. Without Docker (as root on Linux amd64), the repository's `scripts/bootstrap-sandbox.sh` pulls and unpacks the image over plain HTTPS and `scripts/in-dev.sh` enters it. Inside, the shell starts in `/work/aeneas`:

```
grep -l 'GENERATED BY AENEAS' tests/acl2/*.lisp | xargs rm
make -C tests/acl2 regen \
  CHARON=/work/charon/bin/charon AENEAS=/work/aeneas/bin/aeneas
git status --short tests/acl2
make -C tests/acl2 verify JOBS=1
```

Expected results: the `rm` deletes the 19 generated books, and `regen` re-runs Charon and Aeneas on all 19 crates, rewriting them; `git status` then prints nothing, which shows both that every book was actually regenerated and that each is byte-identical to the committed one. (Deleting first matters: the Makefile tolerates extraction failures so that the expected-failure probe crates can run, so with the books left in place a failure could leave a stale book and a clean status.) Then `verify` certifies all 78 books, leaving a `.cert` beside every `.lisp`, including `aes_fixslice_final.cert` and `aes_fixslice_decfinal.cert`. (Aeneas prints a few `[Error]` diagnostics during `regen`, from probe crates that deliberately exercise constructs the backend does not translate; they are expected and do not affect the generated books.)

Cost. Measured on a 2-core, 7 GB machine without Docker (2026-09-05): fetching and unpacking the image, about five minutes and about 10 GB of disk; `regen`, about $2\frac{1}{2}$ minutes; `verify`, about 16 minutes wall clock. Certification is memory-bound: the GL bit-blasting books need several gigabytes each, so `JOBS=1` is required on machines with less than about 12 GB of RAM (with more parallelism the affected book is killed silently); with two parallel jobs and more memory the suite takes about twelve minutes (Section 5.5).

What is and is not checked. `regen` checks that the pinned toolchain reproduces the committed books from the Rust sources; `verify` checks that ACL2 accepts every definition and proof. The theorems are about the ACL2 functions the translation produces: the correspondence to the Rust source rests on

Charon, Aeneas, and our backend printer, which are not themselves verified, and is tested by the known-answer books (Section 5.5). On the ACL2 side the trusted base is the ACL2 core plus the specifications: the community books' AES-128 specification (`kestrel/crypto/aes`, validated by its own FIPS-197 test book and by prior verifications) and the handwritten `rust-primitives` model of Rust's integers, arrays, slices, and result monad. Each verification against the specification is also evidence for it: independently written implementations can agree with a wrong specification only by being wrong in the same way, and this paper adds a third, after the JVM and x86 verifications. The lemma libraries the proofs use (`kestrel/bv` and `kestrel/bv-arrays`, `centaur/gl` and `centaur/fty`, `std`, `arithmetic-5` and relatives) are themselves proved, adding nothing to that trusted base: across the two main theorems' full `include-book` closure (338 books), there are no `defaxioms` and no trust tags. The theorems are functional correctness only: nothing is claimed about timing or the compiled binary. The MIR-interpreter artifact is separate from this image: its committed sample, tests, and generation recipe live with the ACL2 books at the commit pinned in the conclusion (see `books/kestrel/rust/samples/README.md`).