Proof-Carrying Modernization: LLM Translations of COBOL Batch Programs, Each Proved Against the Original in Lean

TEJASVI SINGH TOMAR

Large language models translate legacy code quickly, but a translation can differ from the original on inputs that no test exercises. We show that LLM translations of whole, unmodified COBOL programs into Rust and Java can be proved in Lean 4 to behave like our semantics of the originals on every admissible input: a bounded number of well-formed records, among other premises. The COBOL semantics is executable; it computes record layouts, decimal arithmetic and file handling from our front end's output for the unmodified source. For the eight standalone batch programs of AWS CardDemo covered by that front end, all 16 translations are proved, as equivalences for Rust and as refinements with partial correctness for Java. The statements are generated from the programs except for a few small parts, LLM agents write the proofs, and Lean's kernel checks them. Before any proof, 35% of the first running translations differed from our semantics of the original on a generated input, and no proof found a difference that this testing had missed.

1 Introduction

Large organisations still run batch jobs written in COBOL, and many of them plan to move this code to Java or to memory-safe languages such as Rust. Large language models now write such translations quickly. The translations, however, can be wrong in ways that are easy to miss: a sort order that follows Unicode instead of EBCDIC, a field read before it is set, or a total that the original program carries from one record to the next. Testing finds many of these defects, but a test suite exercises only the inputs that someone thought to write down. When a translation replaces the original, the question is whether the two programs behave the same on every input.

A proof answers that question only if its statement, the meaning of both programs and the proof itself can be trusted. For a whole program, a statement written by hand can be as wrong as the translation it is meant to check. The meaning matters above all for the COBOL program, in which record layouts, decimal arithmetic, the EBCDIC collating order and file handling all decide the outcome. And the proof of a whole program is long. We address each in turn. For the 16 migration theorems, a generator writes each statement from the COBOL program and the translation's interface; only three small input domains and the model of one assembler routine are written directly, by LLM agents, and the generator itself belongs to the trusted base. The COBOL side is an executable Lean semantics of our front end's output for the unmodified source, which we check by differential testing and on the NIST COBOL85 suite in GnuCOBOL's conventions for characters and signs. The IBM conventions used in the theorems are tested only in an exploratory run, with gaps listed in Section 8. LLM agents write the proofs, but Lean's kernel checks every one of them, so no proof step has to be trusted.

We apply this to AWS CardDemo, a public mainframe application, and take the eight batch programs of CardDemo that run on their own and are covered by our COBOL front end. A language model translates each program to Rust and to Java in a repair loop, and nobody edits its output (Section 4). For each of the 16 translations we prove in Lean 4 [17] that it behaves like our semantics of the original COBOL program on every *admissible* input: records whose fields fit their declared formats, bounded in number, together with the further premises listed in Section 2. For Rust the theorem is an equivalence of the two programs'

Author's Contact Information: Tejasvi Singh Tomar, .

outcomes: the lines they display, the records they write and rewrite (compared as field values decoded by the COBOL program's layouts), whether they end abnormally, that is, abend, and the return code where the program sets one. For Java it is a refinement with partial correctness: whenever the COBOL run is safe, no Java run goes wrong, and every Java run that terminates matches the COBOL outcome.

All 16 translations are proved. For three Java programs, the proved translations come from a second style guide, which we introduced after the proofs of those programs' first-guide translations had stalled. In the agents' checks, the unchanged proof of a theorem fails to check or times out when any of the 100 seeded mutants replaces its translation (Section 5). The translation runs also show how the model goes wrong. In 32 of the 92 runs (35%), the first version that ran disagreed with our semantics of the COBOL program on some generated input. Testing on the semantics found every behavioural error we know of, and none of the 16 proofs exposed a further difference. Instead, the proofs required bounds on the number of records and corrections to some of the statements.

Our verification stack (Section 3) also has semantics for C and C++, through ClangIR, and for a subset of TypeScript. Section 6 summarises its uses on code that was not translated from COBOL: Rust ports of C code, fuzzing harnesses, library commits, random samples and Apache Thrift. There the results are more limited, and the technical report [48] gives them in full.

Our contributions are:

- Proved migration: 16 kernel-checked theorems that relate LLM translations of the eight standalone CardDemo batch programs covered by our front end to our semantics of the unmodified COBOL programs on every admissible input, with mostly generated statements and a mutation test of the theorems (Sections 2 to 5).
- A measurement of LLM translation errors in this batch layer, by prompt and language, in which each behavioural error is witnessed by an input on which the translation and the original disagree (Section 4).
- Lean semantics for COBOL, C and C++, Java and a TypeScript subset, joined by one notion of simulation and each tested against a reference implementation (Sections 3 and 5); Section 8 summarizes their known defects.

Section 2 works through one program, Section 3 describes the stack, and Section 4 reports the migration study. Section 5 states what the theorems rely on and how we tested the semantics, and Section 6 summarises the further uses. Sections 7 and 8 discuss related work and the limitations.

2 A Running Example

CBACT04C is the monthly interest calculator of CardDemo, 552 non-comment lines of COBOL. It reads a file of balances, one record per account and transaction category, sorted by account. For each new account it reads the account record and the card cross-reference record. For each balance it looks up an interest rate in the disclosure-group file, keyed by the account's group, the transaction type and the category. When that key is missing, the program displays two lines and retries with the group `DEFAULT`. It then computes the monthly interest as balance times rate divided by 1200 (Section 8 discusses how our semantics places this quotient), adds it to a running total and writes an interest transaction. When the account number changes, it adds the total to the account's balance, zeroes the cycle credit and debit, and rewrites the account record. A missing account, a missing cross-reference record or a missing `DEFAULT` rate ends the run with an abend.

```
theorem rust : ∀ (tcat xref acct disc : List (List FVal)) (link : List FVal)
    (clk : List UInt8),
  (∀ v ∈ tcat, TranCatBalRecord.Valid v) → ... → (∀ v ∈ disc, DisGroupRecord.Valid v) →
  ExternalParms.Valid link → CurrentDate clk →
  Domain tcat xref acct disc link →                          -- written directly
  tcat.length + xref.length + acct.length + disc.length < 2 ^ 28 →
  ∀ n, tcat.length + xref.length + acct.length + disc.length < n →
  ∀ s1 s2 s3 s4 q rclk, TcatbalFileOf s1 tcat → ... → DiscgrpFileOf s4 disc →
    ExternalParms.vals q = link → rclk.val.map (·.val) = rustLine clk →
  Equiv Rel (cbact04c_t.run s1 s2 s3 s4 q rclk)                    -- the Rust translation
            (Cobol.run CardDemo.Cbact04c { clock := fun _ => clk } (n + 60)
               (CardDemo.Cbact04c.initState
                  (Mem.write initMem 1727 (ExternalParms.encode link))
                  [(1, tcat.map TranCatBalRecord.encode), ...]))   -- the COBOL program
```

Fig. 1. The statement for CBACT04C and its Rust translation, abridged. Everything except `Domain` is generated from the COBOL program and the translation's interface (Section 3).

CardDemo's sample data never finds a rate on the first try. All 50 accounts in the sample account file have a blank group, and the disclosure-group file holds only the groups `A000000000`, `DEFAULT` and `ZEROAPR`. So every sample run takes the `DEFAULT` path. The program also has a defect. The call that rewrites the last account sits in a branch that the loop never reaches, so the last account's interest is written as a transaction but never added to its balance. This defect was reported upstream as CardDemo issue #64 [44]. A faithful translation has to keep it. The theorems below say that the proved translations do.

Figure 1 shows the statement for the Rust translation, slightly abridged. The inputs are the records of the four input files, as lists of field values, the linkage record that carries the run date, and the clock. The first hypotheses say that each field fits its declared picture. They are generated from the record layouts of the COBOL program, as is the hypothesis that the clock has the form of `FUNCTION CURRENT-DATE`. The clock is one value. Every read of it in the COBOL run returns `clk`, and the Rust translation receives the same value, converted by `rustLine`, as `rclk`. The hypothesis `Domain` is the only one that is not generated. It asks for fewer than 999,999 balance records, with the records of each account next to each other, as in a file read in key order. It assumes nothing about lookups. The bound of 2^{28} records in all keeps the output vectors within the range of a 32-bit `usize`, and Aeneas (Section 3) allows `usize` to be that narrow. The fuel $n + 60$ bounds how deeply the paragraph transitions, `PERFORM` entries and loop iterations of the COBOL run may nest, and it is enough for the run to finish. The remaining hypotheses tie the Rust arguments to the same records: each input slice holds structs whose fields decode to the given values. Every `OPEN` of the COBOL run succeeds, an assumption discussed in Section 5.

The conclusion is an equivalence up to the relation `Rel`, both defined in Section 3. The COBOL run finishes within the fuel, without undefined behaviour and without leaving the modelled fragment, exactly when the Rust run terminates. The Rust run then returns normally, without a panic, and the two outcomes agree on the displayed lines, as the bytes of their EBCDIC text, on the transaction records written, on the account records rewritten and on whether the run abends. The COBOL side is our semantics of the unmodified program. An equivalence alone would also hold between two programs that both fail, but the proof rules this out along the way: its main lemma states that the Rust translation returns

normally on every input of the domain, with outputs given by a Lean model of the program. A *composite theorem*, with exactly the hypotheses of Figure 1, states in one conclusion that the COBOL run is safe, that the Rust run returns normally and that the two are equivalent.

The Java statement has the same hypotheses, except that the bound is 2^{27} records, a margin below the 2^{31} elements at which the `int` size of a Java collection wraps, and except that it also assumes that the classes of the translation are initialized. Its conclusion is the one-way refinement with partial correctness described in Section 1. Its composite theorem, with the same hypotheses, adds that the COBOL run is safe, that no Java run diverges and that every Java run meets its postcondition. This composite theorem is stated on a later version of the Java semantics, the termination version (Section 3). We call a translation *certified* when its migration theorem and its composite theorem are proved; all 16 translations are certified.

Why insist on every admissible input? An earlier pair of statements for CBACT04C assumed instead that each keyed read finds its record. We call statements over a narrower domain than all admissible inputs *restricted*, and those over all admissible inputs *full-domain*. The restricted statements are proved as well, but CardDemo's own data lies outside them, because on the sample data the first disclosure-group read always misses. The mutation test (Section 5) shows the difference. One mutant of the Rust translation drops the `DEFAULT` retry and abends at once when the account's group has no rate. Take an account whose group has no rate for a category while `DEFAULT` has one. The original writes an interest transaction and ends normally, whereas the mutant abends. The proof of the full-domain theorem fails on this mutant. The proof of the restricted theorem still checks against it, since the restricted domain never reaches the retry. The Java translation's full-domain theorem rejects the corresponding Java mutant as well.

3 The Verification Stack

The stack has three layers: semantics for each language, a relational layer that states and proves that two programs behave the same, and an orchestrator that applies both to whole repositories. Each semantics runs on the unmodified output of a front end, so a theorem is about the code as the front end sees it rather than about a hand transcription of it. We call the versions of our semantics used for the migration theorems (Appendix A) the *migration versions*.

3.1 Semantics on front-end output

COBOL. Our COBOL semantics is called coblean. A front end reads the COBOL source with its copybooks, expands `COPY ... REPLACING`, resolves names and writes a JSON representation. Everything else is computed in Lean from that file: data layout, offsets, encodings, `REDEFINES` overlays, level-88 values and the intermediate-scale rules of arithmetic. Working storage is a byte array in which a field is a range of bytes with a format. The codecs cover `DISPLAY` in EBCDIC or ASCII, packed decimal, binary under the truncation options and native binary. Arithmetic is exact, with COBOL's rounding modes and size errors, and `MOVE` implements padding, justification and numeric editing. Control flow covers every form of `PERFORM`, including `THRU` with IBM's return-slot semantics, `GO TO` and sections. Files are sequential or indexed with IBM's file-status codes, and an unhandled I/O error abends. A run is deterministic and has a fuel bound. Running out of fuel, leaving the modelled fragment and behaviour that COBOL leaves undefined (an overlapping `MOVE`, a subscript out of range) are not outcomes but going wrong. Abends, including `CALL 'CEE3ABD'`, are outcomes, since a rewrite has to reproduce them. For proofs, the main tactic executes a

program symbolically, merges the two branches of an `IF` into one state with conditional fields, and tracks value intervals so that truncation tests disappear when a value fits. A loop over a file is replaced by its result, by induction over the records with an invariant.

Java. JimpleLean gives a semantics to Jimple, the typed three-address code that Soot [50] produces from Java bytecode. A command parses Soot's output at elaboration time and defines the program as a Lean term. The state holds a heap, static fields, class-initialization states and monitors. Method selection follows the JVM specification, exceptions raised by the JVM are objects dispatched through the trap table, and class initialization is lazy, in the order of the Java Language Specification. Library types are Lean types: a `String` is a list of UTF-16 code units and a `List` is a Lean list. Where the JDK leaves behaviour unspecified, the semantics is nondeterministic, and anything outside the models goes wrong. Because the semantics is big-step, it cannot observe divergence, so its verification conditions state partial correctness. The condition generator is proved sound against the semantics. A later version of JimpleLean, which we call the *termination* version, adds a termination calculus, also proved sound (Section 4).

Rust. We do not define a Rust semantics. Instead, Charon and Aeneas [27] translate a Rust crate into a pure Lean model in which a function returns a result, a panic or divergence. Integer overflow, out-of-bounds indexing and `Vec` capacity are checked in the model and panic as in a debug build, and we trust this translation.

C and C++. cirlean works on ClangIR [46], the MLIR dialect that Clang emits before lowering, in which loops, scopes and cleanups are still explicit. Memory consists of blocks of structured values, and a pointer is a block with a path of member and element steps. Undefined behaviour is detected and counts as going wrong: signed overflow, out-of-range access, use after free, reads of indeterminate values, writes to read-only blocks and accesses through stale container pointers. Library models cover `std::vector`, `std::string` and parts of the C library. A symbolic executor with invariant-based loop rules proves the judgement `Returns`, which states total correctness. Every function can live in its own Lean module, and a proof then depends only on its function and the contracts of its callees.

TypeScript. StrictTS defines a statically typed subset of TypeScript and emits Lean through the TypeScript compiler API. Its representation is intrinsically typed, and evaluation therefore has no type errors. Numbers are IEEE doubles, strings are UTF-16 code units, and promises, `async` functions and the event loop are part of the semantics.

3.2 Relating programs

A Lean type class gives each semantics two predicates: `Runs c o`, running `c` can end with the observable outcome `o`, and `Safe c`, running `c` cannot go wrong. Simulation and equivalence are defined once for any two such semantics:

```
def Sim (R : OT → OS → Prop) (t : T) (s : S) : Prop :=
  Safe s → Safe t ∧ ∀ ot, Runs t ot → ∃ os, Runs s os ∧ R ot os
def Equiv (R : OT → OS → Prop) (t : T) (s : S) : Prop :=
  Sim R t s ∧ Sim (fun os ot => R ot os) s t
def EquivOn (Bad : X → Prop) (R : OT → OS → Prop) (f : X → T) (g : X → S) : Prop :=
  ∀ x, ¬ Bad x → Equiv R (f x) (g x)
```

Each instance fixes what going wrong means. For COBOL it is running out of fuel, an unsupported statement or an undefined result. For Rust it is divergence, since a panic is

an observable outcome. For C++ it is undefined behaviour or divergence. At the migration version, a Java target gives only partial correctness, as divergence is invisible to its big-step semantics. The termination version adds total correctness (Section 4). `EquivOn` excludes a set of inputs `Bad`. Its contrapositive says that the two programs can differ only on `Bad`, which is the shape of a bug fix when `Bad` is the set of inputs on which the old version went wrong.

Generated statements. For the migration study, a generator writes each statement from four inputs: the translation's interface (which parameter holds which COBOL file, record or linkage item); the representation of the COBOL program, from which it derives the record layouts and the validity predicate of each field; the form of the COBOL-side theorem (its fuel); and the initial storage of the program. It emits the statement of Figure 1: one quantified list of field values per input file; validity hypotheses; the record bound; the optional domain (not generated); the relations between the arguments of the translation and the COBOL records; and the conclusion with the outcome relation. This relation, `Rel`, compares the displayed lines as EBCDIC text, the records written and rewritten per file, decoded by the layouts of the COBOL program, and the abend flag, plus the return code where the program sets one. The parts of the trusted statement that are not generated are the domains of Table 1 and one model: CBACT01C calls COBDATFT, an assembler routine of 84 lines, and the routine's meaning on the COBOL side is a Lean model written from its source. The model leaves out the error path of the routine, on which the call is unsupported; a run that reaches it therefore goes wrong rather than producing an outcome.

How a migration proof goes. A proof has three parts. The first is a theorem of coblean that gives the outputs of the COBOL run as a function of the input records, for every input of the domain. Its proof executes the program symbolically with the tactic described above: an `IF` becomes one state with conditional fields, value intervals remove the truncation tests where a value fits, and each loop over a file is replaced by its result through an induction over the records with an invariant. The second part relates these outputs to the translation. For Rust, loop invariants over the Aeneas extraction show that the extracted function returns the same outputs and never panics on the domain; for Java, verification conditions at the statement indices of the Jimple method body show the same for every terminating run. Glue lemmas relate the encodings on the way: zoned decimal against `i64`, EBCDIC text against Latin-1 bytes, and records against structs or classes. The proofs for five of the eight programs also pass through a field-value model of the program in Lean, shared by both languages. The third part instantiates `Equiv` for Rust and `Sim` for Java with the outcome relation of the statement, and closes the statement from the first two parts. For CBACT04C in Rust, for example, a glue theorem derived from coblean's all-input theorem states that the COBOL run ends with `GOBACK` or with abend 999 and that its displayed lines, written transactions, rewritten accounts and abend flag are those of a Lean model of the program. A specification proved over the Aeneas extraction states that the Rust run returns the same outputs of that model and never panics. A final theorem derives the generated statement from these two facts and from a lemma that relates the Latin-1 bytes of the Rust output to the EBCDIC text; since both runs are deterministic, both directions of `Equiv` reduce to equal outputs. The Lean model of the program and the glue lemmas are proved rather than trusted, because the generated statements do not mention them; an error in them can make a proof fail but cannot make a theorem wrong. The theorems are named `EquivLean.Modernize.`*`Prog`*`.Rust.rust` and `EquivLean.Modernize.`*`Prog`*`.Java.java` in the artifact, those over all admissible inputs

of CBACT04C and CBTRN02C under `Cbact04c.Every` and `Cbtrn02c.Any`, and Table 3 gives the size of each proof.

3.3 The orchestrator

For the studies of Section 6, an orchestrator, veriforch, applies the semantics to whole repositories. It extracts the call graph, gives each function its own Lean module and works from the leaves upwards. A function that uses a construct outside its semantics is recorded as blocked, with the reason. Statements come in two kinds. Mechanical statements are produced without a model, for example a test run from the module's initial memory or the statement that a Java method throws only what it declares. Model-stated statements, such as contracts, are kernel-checked as well but reported separately. An LLM proposes contracts, invariants and proof scripts, and a candidate counts only if Lean checks it and the claimed theorem uses only the standard axioms. A function that cannot be proved does not block its callers: its contract becomes a hypothesis of each caller, and the caller's result is then reported as conditional. Proofs are cached under a key of the function body, the callee contracts and the library version, so an unchanged function is not re-proved.

For a change, the same machinery gives each changed function one of four verdicts: `equivalent`, that is, `Equiv` over the well-typed inputs of the function; `changed_only_where_old_unsafe`, that is, `EquivOn` with `Bad` the inputs on which the old version is unsafe; `behaviour_changed`, with a distinguishing input found by running both versions on the semantics; and `unknown`. The first two are kernel-checked theorems.

Who did the work. Our proofs, our own tools (the semantics, front ends and orchestrator) and the runs come from LLM agents, each working in its own session under the author's direction. Each of our four semantics has a maintainer session, and these sessions can start helper agents for larger proofs. The session that runs the orchestrator starts further agents that write contracts and proofs directly in Lean where the orchestrator's own model calls fall short. Coordinating sessions assign the work and collect the results. The costs we report are taken from the usage records of these sessions and, for the orchestrator's own model calls, from its usage logs and unit records.

4 Certified Migration of the Batch Layer

CardDemo has 44 COBOL programs. By a source-text rule (no CICS, DB2 or IMS calls), 14 of them are batch programs, and our front end covers 10 of these. Of the other four, CBEXPORT and CBIMPORT declare their record key outside the file's record, which our front end does not accept and IBM's manual does not allow. COBSWAIT, which reads from `SYSIN`, and CSUTLDTC are covered only by later snapshots (Section 4.4). The study takes the eight that run on their own. Table 1 lists them; together they have 2,878 non-comment lines of COBOL, not counting copybooks. The remaining two, CBSTM03A and CBSTM03B, form a pair: CBSTM03A reads its four input files only through calls to the subroutine CBSTM03B and writes its two report files itself. We left this pair out of the translation study, and Section 4.4 reports the theorems proved about it later.

4.1 Translation protocol

Each program is translated in a loop with a fixed protocol. Every attempt is one model call. The first attempt sends the COBOL source, its copybooks and a style guide for the target language. The guide is the same for every program. It fixes the shape of the result: one entry point that takes the records of each input file and returns the displayed lines,

Table 1. The eight programs, the part of each statement that is not generated, and the proved translations. "vk" is the attempt within its run that produced the proved translation; "g2" marks the second Java style guide. Each proved translation is the final version of the run of its guide with input-generator seed 1. All other hypotheses are generated: field validity, the clock form, and the record bounds 2^{28} (Rust) and 2^{27} (Java).

Program	What it does	Lines	Domain (not generated)	Rust	Java
CBACT01C	copies accounts to three files	358	none[a]	v3	v1 g2
CBACT02C	lists the card file	129	none	v1	v1
CBACT03C	lists the card cross-references	130	none	v2	v1
CBACT04C	computes monthly interest	552	$<$ 999,999 balances, grouped by account	v3[b]	v1
CBCUS01C	lists the customer file	130	none	v1	v1
CBTRN01C	checks transactions against cards and accounts	415	none	v1	v1
CBTRN02C	posts the daily transactions	619	$< 10^9$ transactions[c]	v2	v2 g2
CBTRN03C	prints the transaction report	545	$< 10^7$ transactions, a date parameter	v1	v2 g2

[a] CBACT01C calls the assembler routine COBDATFT, whose COBOL-side meaning is a model written directly from its source (Section 5). [b] The first attempt of this run failed because the runner's resource cap killed the extraction, not because of the model's output. [c] Implied by the record bound; it is the hypothesis of the COBOL-side theorem, whose transaction and reject counters have nine digits.

the records written and rewritten, and an abend flag. It restates IBM Enterprise COBOL's file-status rules, gives a naming rule for record fields, and restricts the code to the subset accepted by the verifiers (for Rust, loops over indices and no iterator adapters, closures or strings). It also asks the model to reproduce the program's quirks and defects. Each later attempt sends back the current translation with either the compiler's message or up to three counterexamples. A counterexample is a generated input on which the translation's output differs from that of the COBOL program under our semantics, sent together with both outputs. Nothing else goes back, neither hints about causes nor edits by a person. A version passes when it agrees with the COBOL run on 60 fresh cases and on every earlier counterexample. It must then agree on 300 further fresh cases. A run ends when a version passes this final check, after 10 attempts, or when it has spent $15. Each guided translation runs both natively and inside the verifier: Java on the JVM and in JimpleLean's interpreter, Rust natively and through its Aeneas extraction evaluated in Lean.

A plain arm asks only for an idiomatic Java or Rust translation, with the input and output contract needed to run it. Its translations use real file I/O, which is outside the languages of the verifiers. They run natively and serve only to measure errors. For three Java programs we proved translations made with a second version of the Java guide. It adds four rules on structure: each search over a file's records in its own helper method, no large constant tables, little state carried across the main loop, and output text built with a `StringBuilder`. The translations from the first guide passed testing as well, but their proofs were not completed.

Every proved source equals, byte for byte, the final version of its run in the translation records. The Java bytecode and the Jimple imported by the proofs regenerate identically from these sources. All 20 generated statements regenerate identically as well: one per program and language, and a second, restricted one for CBACT04C and CBTRN02C. Three of the four restricted statements are proved. For the Java translation of CBTRN02C, only the full-domain statement was proved.

4.2 Results

All 16 theorems are proved, use only Lean's standard axioms and pass the kernel replay and the mutation test (Section 5). Each Rust proof also shows that the translation returns normally on every input of its domain.

The need for full-domain statements arose on the COBOL side. Proving the COBOL programs on all admissible inputs brought up three paths that are easy to leave out: CBACT04C's `DEFAULT` retry, which CardDemo's own data always takes; the `NEXT SENTENCE` in CBTRN03C, which ends the report at the first transaction dated outside the requested range; and the abend of CBTRN02C on a repeated transaction id. The statements in Table 1 cover all three, and Section 5 shows two Rust mutants that the full-domain statements reject and the restricted ones accept.

The translations keep the defects of the original programs, as the theorems require. CBACT04C never adds the last account's interest to its balance (reported as CardDemo issue #64 [44]). At end of file, CBTRN03C adds the last transaction's amount to the totals a second time (described in CardDemo pull request #32 [20]). CBACT01C sets the cycle debit of its output record only when the account's own debit is zero. An account with a debit therefore inherits the value written for the previous one.

Termination of the Java translations. The Java theorems above state partial correctness. The termination version of JimpleLean adds a calculus that uses loop variants. Its soundness theorem (`TermVC.mterm`) derives the semantic termination predicate `MTerm` from the conditions of the calculus, under the contracts of the callees. `MTerm` holds when every execution terminates, whatever the nondeterministic choices. With this calculus, on the termination version, all eight Java translations are also proved totally correct. For every input admitted by the migration theorem, one theorem per program states `MTerm` for the Java program, and a total-correctness version of the migration theorem, proved from that theorem, states that an execution exists and that every execution ends normally with the outputs related to the COBOL run by the migration theorem. So no Java run diverges, and every run matches the COBOL outcome. These theorems belong to a separate commit that uses the termination version of the Java semantics. The migration theorems also check in that commit, and the results above refer to the earlier version.

4.3 How the model went wrong

Table 2 summarizes the 92 counted runs: 8 programs, two languages, the plain arm and the first guide with seeds 1 to 3, and the second Java guide with seed 1. The plain arm does not apply to CBACT01C, whose output record has a packed field outside the contract of that arm. One guided Rust run of CBACT04C is left out, because the runner's resource cap killed its first attempt (Table 1, note b). That run produced the proved translation, so 15 of the 16 proved translations come from the counted runs. Every run ended with a passing version. The first version that ran differed from the COBOL program on some generated case in 32 runs (35%, 95% interval [26, 45]). The rate was 20/42 with the plain prompt and 10/47 with the first guide. With the guide, 12 of the 23 first Rust versions did not build, because Charon or Aeneas rejected a construct. Counting first attempts instead of first running versions, 43 of the 92 did not pass: the 12 that did not build and 31 that differed. The two arms differ in more than the guide, since the translations of the plain arm read and write real files instead of using the interface of the verifier. The differing first versions also cluster by program: none for CBACT01C, CBCUS01C and CBTRN01C, but

Table 2. Translation errors over 92 runs. "Differed": the first version that ran differed from the COBOL program on a generated case. Collating sequence: causes marked fixed whose kind is the collating sequence, labelled after the runs from a fixed list of 15 kinds. Intervals are Wilson intervals over runs treated as independent.

Arm, language	Runs	Differed	First version did not build	Collating sequence
plain, Java	21	7	0	6
plain, Rust	21	13	0	11
guide 1, Java	24	6	0	8
guide 1, Rust	23	4	12	4
guide 2, Java	3	2	0	2
all	92	32 (35%, [26, 45])	12	31

11 of the 13 runs of CBTRN03C and 8 of the 13 of CBTRN02C had one. The study was not pre-registered, and its intervals treat the runs as independent and are therefore too narrow.

The causes were labelled after the runs, from a fixed list of 15 kinds. For each run whose repairs fixed behaviour, further model calls named the causes that each diff fixed: one call per run in 30 of the 32 runs, 37 calls in all. These calls saw the COBOL program and, for the repairs, the version before each, the diff and the counterexamples. Each repair that only fixed a build failure, in the 12 guided Rust runs whose first version did not build, was labelled from its build error, without a model call. The repair loop never saw these labels. Of the 50 causes marked fixed in the 52 repairs, 31 are the collating sequence: the model had compared or sorted alphanumeric fields in Unicode order where the mainframe uses EBCDIC. Next come constructs outside the verifier's language (12, all guided Rust) and uninitialized storage (6 causes in 4 repairs). No repair was labelled rounding, truncation, sign, file status or control flow.

Three findings concern the checking itself. First, for every guided translation, the native run and the verifier's execution agreed on all cases of all attempts, so any discrepancy was also visible on the formal semantics. The translations of the plain arm ran only natively. Second, take the 10 first versions from seed 1 (plain arm and first guide) that failed on generated inputs. The records of the translation runs, which we did not re-run, show that none of them differs from the original when run on CardDemo's own sample data. Third, no proof found a behavioural difference that testing had missed (0 of 16). Instead, the proofs found two things that tests cannot reach. The statements needed bounds on the number of records, so that collection sizes fit a 32-bit `usize` in Rust and `int` indices in Java. And some statements had to be corrected, for example to model the JVM's class initialization for a translation that reads a static table, or to pad a `READ INTO` a record shorter than the file's. In all, the generator and the domains of the eight programs changed in 26 commits after their first versions (CBTRN03C's first domain counts as its first version). The sessions that wrote the generator and the domains classified each change against the earlier statement of the same program, in a list that accompanies the paper: 9 made statements weaker (the record bounds, premises that the translation's classes are initialized, and the domains of the restricted statements), 6 made them stronger, mostly by widening a domain to every admissible input, and 11 were neutral (first statements of further programs and of Rust, new features, representation changes, a superseded change, and fixes of statements that were ill-formed before any proof). The same list also covers CBSTM03B.

4.4 The pair CBSTM03A and CBSTM03B

The pair CBSTM03A and CBSTM03B, which the translation study left out, is covered by later theorems on the migration version of the COBOL semantics. COBOL-side theorems give a call of CBSTM03B, and a run of CBSTM03A with CBSTM03B linked, by functional models of their outputs. They assume, among other premises, record lengths and keys of the expected forms, a fixed model of the z/OS address space that CBSTM03A inspects, and successful opens. For the pair, they also assume a nonempty transaction file sorted by card, with at most 50 card groups of at most 10 records each, whose customers and accounts are all found. This assumption makes the pair's main theorem restricted in the sense of Section 2, and two more theorems give the abend when a customer or account is missing. Run under GnuCOBOL without the walk through the address space, the pair writes the same 1,212 account-statement records and 6,082 HTML records as our semantics.

Both programs then went through the translation protocol (Section 4.1), in runs that are not part of the statistics above. Theorems with mostly generated statements relate the first-guide translations of CBSTM03B to the COBOL call, by an equivalence for Rust and by a refinement with partial correctness for Java. Further theorems relate the Rust and the first-guide Java translations of CBSTM03A to the linked pair on every input in a stated domain, including the runs that abend; the Java one states partial correctness. Their relation leaves out the 11 lines displayed by the walk through the address space, and their domain, the address-space model, the patterns of those lines and the map from files to record layouts are written directly. The theorems of this subsection are *exploratory*: they lie outside the planned studies and the pre-registrations (Appendix B); those for CBSTM03A are proved on later versions of our packages (Appendix A). Composite theorems, for CBSTM03B's Java translation and for these two translations of CBSTM03A, add the safety of the COBOL run and the normal termination of the translation. Two exploratory theorems on later snapshots of coblean cover CSUTLDTC and COBSWAIT on the COBOL side. They leave the external routines called by these programs arbitrary, and they allow faults named in the theorems. With them, 12 of CardDemo's 14 batch programs have a theorem on the COBOL side. The technical report [48] states every hypothesis of these theorems.

4.5 Cost and effort

The model calls of all translation runs, in every arm and seed, and of the labelling of root causes cost $50.57 over 191 calls; the runs that produced the 16 proved translations cost between $0.08 and $1.30 each (Table 3). The proofs cost far more. By the usage records of the sessions that did it, the proof work cost about $3,130 in model usage from its start until shortly before its proofs were frozen. This figure includes the shared bridges and the COBOL side but not the earlier development of the semantics and the bridges. The Lean builds and checks logged in two of the three proof sessions took at least 52 core-hours, part of them for the tiny-regex-c port (Section 6). A clean build of all migration proofs, with the libraries of the semantics already built, takes 7.9 core-hours on one core. The program-specific modules have 54,413 lines of Lean, 1,233 of them generated statements, against 2,878 lines of COBOL. No person wrote any of these Lean lines.

5 What the Theorems Rely On

A kernel-checked theorem is only as good as its statement and its trusted base. This section lists both for the migration theorems, reports how we tested the semantics, and describes three checks of the theorems: axioms, kernel replay and a mutation test. We re-checked every

Table 3. Size of the proofs and cost of the translations, per program. Lean lines are non-blank, non-comment lines of the program-specific modules in the import closure of each theorem; shared lines (model, glue, domain) serve both languages. A module that several programs' theorems import counts in each of their rows and once in the last row. Translation cost: model calls of the run that produced the proved translation.

	COBOL	Lean lines				Translation $	
Program	lines	Rust	Java	shared	statements	Rust	Java
CBACT01C	358	1,286	2,296	1,149	98	0.69	0.32
CBACT02C	129	226	600	745	70	0.08	0.09
CBACT03C	130	241	1,008	188	71	0.16	0.10
CBACT04C	552	4,224	9,519	2,336	306	1.30	0.45
CBCUS01C	130	225	893	188	70	0.10	0.11
CBTRN01C	415	729	2,298	261	145	0.23	0.21
CBTRN02C	619	3,880	10,364	2,907	326	0.95	1.10
CBTRN03C	545	2,768	7,030	1,720	147	0.52	1.00
all, each module once	2,878	13,579	33,222	6,379	1,233	4.03	3.38

number in this paper on a frozen copy of the artifacts. The exceptions are numbers taken from the records of the sessions that produced the results, and for each of these we name its source.

5.1 Trusted base

Each theorem relies on Lean's kernel and its three standard axioms (`propext`, `Classical.choice`, `Quot.sound`), and on what the definitions mean. Table 4 lists, per language, the front end that turns source text into Lean terms, the semantics, the library models and how we tested the semantics. Tactics, lemmas and program logics are not in it, since the kernel checks everything they produce. For Rust, the trusted base includes Aeneas's translation, which we test only through our own programs: the native runs of the Rust translations and of the tiny-regex-c port (Section 6) agreed with their Aeneas extractions on every generated case, while the jsmn and Adler-32 ports were compared with the C code only in native runs.

By contrast, the binary-level proofs about OSS-Fuzz harnesses by the company Theorem [47] rely on a smaller trusted base. Because they are about the compiled binary, neither the compiler nor the library sources are part of it. They trust a model of a Linux process that runs RISC-V code (a model its authors call best effort and not complete), the tools that load a binary into it, and the Rocq kernel. Our trusted base is larger. It includes a front end, a semantics and library models per language, and Aeneas's translation of Rust. In return, our theorems are about the source program as our front ends present it, not about a compiled binary. They cover undefined behaviour that the binary does not show (Section 6), and they relate programs written in different languages.

For the migration theorems, the statement fixes what is claimed and therefore has to be read. The generated part follows one template for all programs, whereas the part that is not generated, listed in Table 1, consists of the input domains and the model of the assembler routine COBDATFT. The statement generator, the definitions of `Equiv`, `Sim` and `Rel`, the record encoders and these parts that the agents wrote directly are therefore trusted as well. Agents read them during our checks, and we have no record of a line-by-line review by a person. Last, the theorems abstract file I/O. Each input file is the list of its records, and

Table 4. What the theorems trust, per language, at the versions of Appendix A and Table 5. Lean lines count the files that define the language (syntax, values, memory or data layout, evaluation rules, number and string models) and the library models.

	Front end	Semantics	Library models	Tested against
COBOL	our front end (3,468 lines of Python), Lean loader (1,137 lines)	4,461 Lean lines	`CEE3ABD` (an abend); `COBDATFT`, from its assembler source	Tables 5–6
Java	`javac` (JDK 17), Soot 4.7.1, our Jimple reader (1,543 lines)	2,180	449 library methods (4,961 lines)	Tables 5–6
Rust	Charon `de8736f`, Aeneas `fdebf20`	Aeneas's library, 14,899 lines with its lemmas	in Aeneas's library	native runs
C/C++	ClangIR (llvm-project `4955d32b`), our CIR reader (3,150 lines)	2,578	102 library operations (1,285 lines)	Tables 5–6
TypeScript	our front end on the TypeScript 5.9.3 compiler (10,287 lines of TypeScript)	5,283, built-ins included	98 built-in operations	Tables 5–6
Statements	the statement generator	`Equiv`, `Sim`, `Rel`, record encoders	the domains and the COBDATFT model, written directly	mutation test

each output is the list of records written or rewritten; the semantics represents an indexed file as a list in ascending key order with unique keys, a documented limit. The file statuses of the semantics are those of operations on these lists, such as end of file, a missing or duplicate key and the logic errors of IBM's status table, and those reported by the environment for a failing `OPEN`. In the migration statements the environment lets every `OPEN` succeed, and the semantics assumes that I/O never fails for physical reasons.

5.2 Testing the semantics

Table 5 lists the differential tests of each semantics against a reference implementation, and Table 6 two existing collections of test programs and, under a pre-registration, random mutants of the semantics. The COBOL differential tests run the representation with GnuCOBOL's character set and sign conventions. The theorems use IBM's conventions, the *IBM variant* of the representation, and only an exploratory run of the same testers covers it. That run, on 964 generated and hand-written programs of its own, finds no disagreement and declines 11 of them as outside the modelled subset; some programs agree only up to a negative packed zero or at the same abend. Section 8 lists what the run leaves untested.

The Java and C++ suites were also run at versions other than those of Table 5. At the migration version of the Java semantics, 2 of the 41 JVM programs stop at a method without a model. All 41 agree at each later version that we tested. These are the versions of the table, of a sample of Thrift functions in the technical report and of the joint build (Appendix A), and the termination version. The C++ suite gives the result of the table at five other versions used for C and C++ theorems that we report. At the earlier version of the first sample run and the security-fix studies, it had 167 cases, of which 166 agree and 1 is reported as undefined behaviour. Other results, among them some of the re-checks in the technical report, use versions not covered by these runs.

Table 5. Differential tests of the semantics against reference implementations, for COBOL at the coblean snapshot 20261024T2306Z of the migration theorems and for the others at the versions of the second run on random samples: cirlean d9ae8e5, JimpleLean 20261025T0300Z and StrictTS 44d6b9b.

Semantics	Reference	Suite	Result
COBOL	GnuCOBOL 3.2	random arithmetic, `MOVE` and `IF` programs, `TRUNC(STD)`: 200	200 agree (5 only up to the stored sign)
COBOL	GnuCOBOL 3.2	the same generator, `TRUNC(BIN)`: 200	198 agree; 2 differ (binary items beyond their `PICTURE`, not claimed)
COBOL	GnuCOBOL 3.2	random `PERFORM`/`GO TO` programs: 100	88 agree, 12 declined (recursive or abandoned `PERFORM`)
COBOL	GnuCOBOL 3.2	front end on random programs: 100 arithmetic, 60 control flow	100 agree (2 only up to a sign); 58 agree, 2 declined
COBOL	GnuCOBOL 3.2	sequential and indexed files: 100; `UNSTRING`, `INSPECT`, `STRING`: 60 each	all agree (one at the same abend)
COBOL	GnuCOBOL 3.2	CBSTM03A with CBSTM03B on CardDemo's data, without the z/OS walk	both output files equal, 1,212 and 6,082 records
C++	native	175 cases (arithmetic, control flow, RAII, exceptions, classes, containers, the C library)	174 agree, 1 reported as undefined behaviour, none differs
Java	JVM (JDK 17)	41 programs	all agree; at the migration version, 39 agree and 2 stop at a method without a model
TypeScript	Node 22	138 programs; 40 programs outside the subset; 1,854,483 checks of number formatting and string methods	all agree; all rejected; no mismatch

Table 6. Collections of test programs and mutants of the semantics. The NIST programs ran at the coblean snapshot of the migration theorems and the LLVM programs at cirlean `ffab71e`. Intervals are 95% Wilson intervals.

Semantics	Reference	Suite	Result
COBOL	GnuCOBOL 3.2	NIST COBOL85: 309 programs of six modules (pre-registered)	84 pass; 152 outside the subset; 46 fail; 25 front-end errors; 1 undefined behaviour; 1 timeout. In the 130 that run to the end, 2,419 of 2,525 test verdicts agree
C	native	LLVM test programs: 177 with the reference status natively, of 200 drawn (pre-registered)	95 same status (53.7%, 46.3–60.9); 3 another; 25 undefined behaviour (7 confirmed); 31 outside the subset; 13 `exit`; 10 timeouts. With a verdict: 95 of 123 (77.2%, 69.1–83.8)
all	differential tests	30 random mutants of each semantics (pre-registered)	killed: C 7, Java 16, COBOL 15, TypeScript 20; COBOL front end 18
COBOL, Java, C	differential tests	extended suites on 30 fresh mutants each	killed, new suite against old: COBOL 21 vs 19; front end 23 vs 21; Java 24 vs 20; C 24 vs 10 (23 without one memory-limit kill)

The test programs. None of the constructs that make NIST programs fail or end with errors occurs in the eight programs of the migration theorems. Because the front end of our version would stop every program at its first report line, a `WRITE ... ADVANCING` statement, these runs treat that statement as a plain `WRITE`. Of the 46 failures, 34 are caused by constructs dropped by the front end with only a warning, 33 of them declaratives. Six, in statements with `NOT ON SIZE ERROR` but no `ON SIZE ERROR`, arise from IBM's documented rule that a size error without an `ON SIZE ERROR` phrase stores the truncated result, where GnuCOBOL and the suite leave the receiving item unchanged. The others are due to three front-end bugs, a defect of the semantics in `INSPECT`, a file status that our version could not express and an earlier program that did not run. In an exploratory census, later snapshots pass more of the suite (technical report). Because our C semantics does not model printed output, the LLVM comparison covers only the exit status and the absence of undefined behaviour. Of the 25 stops at undefined behaviour, the sanitizer confirms 7, and in the others the semantics is stricter than C or has a gap; our version also reports `exit` as undefined behaviour. The three mismatches are due to a front-end bug in hexadecimal floating-point constants of type `float` or `long double` (the ClangIR of our theorems contains no such constant), to type punning through a union (Section 8) and to C++ dynamic initializers, which the semantics does not run. Two of the ten timeouts follow a change to the interpreter's stack, recorded in the deviation log (Appendix B). Exploratory re-runs on three later versions of the C semantics raise both the share of programs with the reference status and the share of verdicts that match at each step (technical report).

Mutants of the semantics. Each mutant changes an operator, a literal or a condition in the definitions of a semantics, or for COBOL a rounding mode or a file-status code; it was rebuilt with its broken proofs replaced by `sorry` and counted as killed when some differential test failed. The survivors cluster in features the suites barely exercise: floating-point formatting and parsing in C (13 of 23 survivors), floating-point operations in Java (9 of 14), and less common statements and clauses in COBOL; a few are equivalent to the original. Suites extended in these areas kill more of a fresh set of 30 mutants each: for COBOL under a pre-registered addendum, for Java and C in runs that were not pre-registered. An agent whose tool log shows that it had not seen the mutants wrote the C tests.

5.3 Axioms and kernel replay

We printed the axioms of every theorem that we report as a result, from our own copies, and only the standard axioms appear. Printing the axioms is not enough. In a control, a false theorem added with kernel checking switched off compiled and reported no axioms at all, and only a kernel replay rejected it. We therefore replayed, with Lean's `leanchecker`, every module of our packages and of Aeneas in the import closure of each such theorem, except modules that merely import others. Where a module and its closure were byte-identical to ones replayed earlier, we reused that result. Mathlib, the packages it depends on and Lean's core library were not replayed and are at the revisions pinned in the lake manifest. For the migration theorems, the replays cover 924 modules, together with the 68 modules of the COBOL package that holds the COBOL-side theorems. One theorem of that package uses the native-evaluation axiom of `bv_decide`. We do not report it, and it lies outside the closure of the migration theorems. The tool that computed the import closures missed the 17 modules of Aeneas's metaprogramming library. Byte-identical copies of these modules pass a replay for the migration, composite and CBSTM03A theorems. For the Adler-32, jsmn and Thrift wire-format theorems of Section 6, they pass a replay only in a later build

that contains those theorems. The technical report [48] gives the replay behind every result, including exploratory re-checks on later versions of the semantics.

5.4 Mutation test of the theorems

Are the theorems vacuous, or too weak to tell a right translation from a wrong one? We seeded 100 mutants into the proved translations, 50 in Rust and 50 in Java, five to eight per program and language. Each mutant is a small edit, usually of one line. The categories are rounding mode, loop bound, truncation width, swapped fields, and a dropped step, fallback, check or abend; a few mutants change the handling of end of file or the state carried from one record to the next. For each mutant, a native harness runs the original and the mutant on one admissible input and shows a difference in the output, while the original run against itself shows none. Since the statement fixes the outputs on every admissible input, it is false for the mutant and has no proof. At the migration versions, that falsity rests on the native run, not on a Lean proof. When the extraction is regenerated from the mutant (Aeneas for Rust, `javac` and Soot for Java) and checked with the unchanged statement and proof scripts, the build of the theorem fails. It does so in every run reported by the session that ran these checks, and in each of the six runs that we repeated. All 100 mutants are thus rejected, 97 with an error and 3 because a module did not finish within the time limit of its run (about 30 minutes for two of them and 83 minutes for the third). The three mutants that ran out of time are Java mutants. For each, a later exploratory Lean theorem refutes the mutant's statement on the termination version of JimpleLean, also used by the composite theorems. That statement is the generated one with the mutant program in place of the translation. On one input with one record per file that meets every premise, a run of the mutant ends with an outcome different from the COBOL program's. The run is computed in the kernel by an evaluator proved sound for the Jimple semantics.

The mutation test also checks the domains. Two mutants change the output only on inputs outside the earlier, restricted domains: the CBACT04C mutant from Section 2, and one in which the duplicate-id abend of CBTRN02C can no longer happen. For the Rust translations, the restricted theorems still check against both mutants, whereas the full-domain theorems reject them.

6 Further Uses of the Stack

The semantics and the orchestrator also apply to code that was not translated from COBOL. The technical report [48] gives these studies in full and names the pre-registrations behind them; their C and C++ results depend on a C semantics with known defects (Section 8). The method also applies to small pieces of C code. Theorems with statements written by LLM agents relate safe Rust ports of tiny-regex-c, jsmn and zlib's Adler-32 checksum to the ClangIR of the unmodified C code (for zlib, compiled with the flags named in the technical report). They state that the Rust code returns normally with the same results (for jsmn, the return value and the tokens that it counts) on bounded inputs on which the C run terminates without undefined behaviour in our C semantics, under further premises. A model translated tiny-regex-c in the same kind of loop as the COBOL programs, and agents wrote the other two ports directly. Elsewhere the results are weaker. Under a pre-registration, no commit among 63 security fixes of cJSON and tiny-regex-c received a kernel-checked verdict. On a TypeScript library, a semantic diff tool proves 46 of 415 changed functions equivalent, many of them only in restricted forms such as at one type instance, and shows 86 to differ, 76 of them on calls that are well-typed under the declared types. This study was designed after a first look at 31 pairs, which stay in the counts, and was not pre-registered.

On pre-registered random samples of functions from four code bases, the stack proves at most 29% of the functions in any run, mostly against statements that the model wrote and no person has reviewed. On Apache Thrift, the strongest result concerns the wire formats: under contracts for the transport, theorems with statements written by agents show eight departures of the C++, Java and Rust libraries from a Lean specification of the binary and compact encodings, each confirmed on the real libraries, and in seven of them the libraries also differ from each other. Java, for example, rejects an empty list of UUIDs, and on the real library it cannot read back any list, set or map of UUIDs that it wrote itself.

Undefined behaviour and fuzzing harnesses. Run on our C semantics, the test programs of tiny-regex-c and cJSON reach two cases of undefined behaviour that GCC 15.2 builds with the default sanitizers do not show: forming a pointer one element before a string literal in `matchstar`, and comparing pointers into distinct objects in `cJSON_SetValuestring` [28]. On the OSS-Fuzz harnesses that Theorem proved at the binary level [47], agents and the orchestrator proved in Lean 4 that, in our C semantics (where allocation always succeeds and library calls run our models), the unmodified cJSON 1.7.18 harness terminates without undefined behaviour below 2^{61} bytes (proved across four versions of the semantics, after a parse-only variant below 2^{63} bytes) and the unmodified yajl-ruby 1.4.3 harness does so below 2^{31} bytes. Three further theorems cover copies with a small source change: with seven lines added to `print_string_ptr` in cJSON 1.7.18 [10], the whole-harness statement holds below 2^{63} bytes on 64-bit and below 2^{31} bytes on a 32-bit build, and with one line changed in `yajl_lex_number`, that function's contract holds across all `unsigned int` lengths below 2^{32}. All of these harness theorems lie outside the pre-registration (which covered a seed-corpus sweep and an unfinished attempt on yajl-ruby), use only Lean's three standard axioms and pass kernel replay; the technical report gives the effort tables, the static-scan bounds and the per-version module counts.

7 Related Work

LLM translation with formal checks. Several systems check LLM translations formally. Heimdall [16] migrates eBPF programs from C to Rust and checks, with symbolic execution of both binaries and Z3, that they are equivalent on all inputs that satisfy a safety condition, for 109 of 115 programs. VERT [53] checks LLM translations to Rust against an oracle derived from WebAssembly, with property tests or bounded model checking. LLMLift [5] has the model write a program in a domain-specific language together with invariants and a proof that an SMT solver checks. Browne et al. [8] prove in Lean that LLM-generated GPU kernels match a specification under assumptions they state, and Lazaropoulos and Paraskevopoulou [32] have LLM agents write Lean refinement proofs for EVM bytecode over a semantics that passes the Ethereum conformance tests. IRONPROOF [6] checks COBOL-to-Python translations with Z3 against its own representation of the COBOL program, and reports that proofs of that kind can disagree with GnuCOBOL. Equivalence modulo an explicit set of behaviours, as in our domains, appears in Heimdall and in Partial Contracts [12]. Our setting differs in the programs and in the checker. Our programs are whole unmodified COBOL programs with loops, files, packed decimals and `REDEFINES`, translated into two languages, and our proofs are kernel-checked over a COBOL semantics that is tested against GnuCOBOL. Testing a semantics against GnuCOBOL, another implementation or a conformance suite is itself common practice [1, 6, 32].

COBOL migration and CardDemo. Industrial COBOL-to-Java migration is validated by testing. Ferenczi et al. [23] certify repeatable entry-replay witnesses for COBOL-to-Java

migration and report that none of the 45 seeded precision or rounding mutants was killed, although the witnesses reached 35 of them, which shows how hard such errors are to catch by tests. A Python model of COBOL numerics has been compared with GnuCOBOL [1], and a formal description of COBOL's control constructs dates from 1993 [4]. CardDemo has been analysed before. Range assertions were checked on four of its programs [45], and its defects are reported in issue #64 [44] and pull request #32 [20], which also describes the `NEXT SENTENCE` and end-of-file behaviour of CBTRN03C. We do not claim to have found these defects. Our contribution for these programs is the theorems.

Errors in LLM translations. Pan et al. [40] classify 1,748 bugs in LLM translations between five languages, Eniser et al. [21] measure translation to Rust on real projects with differential fuzzing, Chang et al. [11] study vulnerabilities that translation introduces, and Du et al. [19] study hallucinations in code translation. Our measurement is narrower in scope. It gives rates and kinds of errors for one batch layer, where each behavioural error comes with an input on which the translation and the original disagree, and 16 translations, one per program and language, carry a proof.

Semantics in proof assistants. For ClangIR, circe [3] translates a list of program shapes into Lean through a verified emitter, and VeIR [38] has an integer-only `cir` dialect. Java bytecode is verified through Soot's Jimple in ByteBack [39], and K-Java [7] tested an executable Java semantics against the JDK on far more programs than we did. Tarski and Thales [2] give JavaScript a Lean semantics, and LemmaScript [37] translates annotated TypeScript to Dafny or Lean. Aeneas [27] and Charon [26] give us Rust. We found no mechanized COBOL semantics in a proof assistant before ours.

Relational verification. Language-parametric equivalence is not new. Keq [29] and Ciobâcă et al. [15] relate programs across two semantics, and SAW [18] has proved C and Java code equivalent since 2013. Lock-step loop rules are standard [31, 34, 42, 43], as is regression verification of changes within one language [22, 24, 30]. Alive2 [35] lets a target differ from its source only where the source has undefined behaviour, which is the shape of our `EquivOn` verdicts.

Other uses of verification. That compiled code need not show source-level undefined behaviour is well known [51, 52], and kcc [25] detects examples of all 77 categories of core-language undefined behaviour in C11. Proofs about OSS-Fuzz harness binaries [47] rule out the errors that the sanitizer builds detect, and their authors state that errors the sanitizers cannot detect are not excluded; Section 6 shows two such cases in the projects' own tests. Code-level proofs have run in continuous integration at AWS [13, 14], DiffKemp [36] checks semantic equivalence between versions of large C projects, Sarker et al. [41] quantify the impact of CVE patches symbolically, and verification results have been cached across runs [9, 33]. We found no evaluation of verification tools on random samples of functions with confidence intervals or a pre-registration, in recent papers on LLM-based verification and in the literature they cite; one evaluation samples 70 lemmas at random for an ablation [49].

8 Limitations

What the migration theorems cover. The theorems are about records in memory, so they say nothing about opening real data sets, record locking or the job control that runs the programs. The bounds (2^{28} and 2^{27} records, and the domains of Table 1) are part of every statement. The total-correctness theorems for Java are proved on the termination version of JimpleLean, and those of CBSTM03A on a later version (Appendix A). The eight programs

are small batch programs, 129 to 619 lines each, and the online programs of CardDemo, which use CICS, are out of scope.

Trusted front ends and translations. A theorem is about the terms that a front end and a generator produced. None of the front ends is verified, and Aeneas's translation of Rust is trusted. The differential tests exercise each pipeline end to end, but a front-end error that the tests miss would make a theorem concern a different program.

Validation by testing. Agreement with a reference implementation supports, but does not prove, that a semantics matches its language. GnuCOBOL is the only COBOL implementation we could run. Where GnuCOBOL deviates from IBM's documented behaviour, the semantics follows the IBM manuals, and the generators of the differential tests leave out the classes of programs in which the two differ. Each tester also compares only part of a program's state: the arithmetic tester keeps one flag for the outcomes of conditions and size errors, the file tester compares only the failing statement of a program that stops at an abend, the `UNSTRING` tester compares only receivers, delimiters and counts, and the string testers start their counters at zero. A later version of the testers without these limits finds one more difference, in a class where GnuCOBOL departs from IBM and the semantics follows IBM.

The COBOL tests of Table 5 use GnuCOBOL's conventions for characters and signs, not the IBM variant used in the theorems. The exploratory run in the IBM variant tests the construct classes used in the theorem programs, with the exceptions below. Apart from CBSTM03A with CBSTM03B, it does not run the programs themselves. Both statements and the list that follows come from the coverage list of the COBOL maintainer session. The run leaves untested the clock in CBACT04C and CBTRN02C, the `INITIALIZE` statements of three programs, CBACT01C's assembler call and variable-length file, the exact edited pictures of CBTRN03C and the walk of CBSTM03A through the address space. Two gaps remain in every version of the testers. GnuCOBOL's indexed files keep record keys in native byte order under every flag set we tried, so the order of keys is not tested. And seven `DISPLAY` operands of the theorem programs, five in CBACT01C and two in CBTRN03C, are signed zoned-decimal items, and the semantics prints their stored bytes with the sign overpunched on the last digit, as IBM Enterprise COBOL does under its default option `DISPSIGN(COMPAT)`. GnuCOBOL prints a separate sign under every option we tried, the IBM-variant run included, so no differential test checks these outputs, only the stored bytes; a build with `DISPSIGN(SEP)` would print them differently.

Known defects of the COBOL and Java semantics. At the version used for our theorems, the NIST suite and the differential tests found front-end bugs and defects of the COBOL semantics. The COBOL maintainer session reported more, some of them after our checks, among them departures from IBM's rules for the digits that intermediate results carry. Of the ten programs covered by this version, only CBACT04C reaches one of these defects. One of its statements divides under a rule for the places of a quotient that differs from IBM's, and the stored value is the same under both rules for every input, provided that IBM truncates the extra places of the intermediate result. We base the claim that only CBACT04C reaches one of these defects on our own scans, except for six defects and departures reported after our checks. For those six, the claim rests on the maintainer session's report, and our scan of the divisions supports that report for one of them. The technical report [48] describes each defect, its exposure and the later snapshots of coblean. In our exploratory re-checks, these snapshots fix each defect or stop with a fault where it would apply, with two exceptions.

They keep the rule for quotients in the form of CBACT04C's computation, and they fix the read position only for files declared with dynamic access. A later Java tester found a wrong library model of `Double.compare` for NaN, which no Java theorem that we report calls.

The C semantics. The C and C++ results of Section 6 rest on a C semantics with known defects. At the versions behind them, memory holds typed elements, not bytes, so a load or store of a scalar of another width than the element it reaches, such as a word read from a character array, reads or writes one element where C reads or writes bytes; we call this the one-element meaning. Type punning through a union, some pointer arithmetic, the copying of objects of classes with non-trivial copy or move constructors or destructors when they are passed or returned by value, and reads of globals that a unit only declares are not modelled as the languages define them either; the C library models have gaps, and the semantics defines several operations that C leaves undefined. Those results hold only for code without such accesses and operations. The technical report lists these and the other defects, what scans of the ClangIR on the paths related by the results find, and where the scans cannot decide; a later version of the semantics fixes the defects on our tests, up to gaps that it labels.

The model and the protocol. The translations were produced by one commercial LLM, accessed through its API with fixed settings (Appendix C). Its output is not deterministic, and a newer model may err differently. We ran three seeds of the input generator per arm (one for the second Java guide), each with its own translation run. We report every run, and the statistics leave out only one of these runs, because the runner's resource cap killed its first attempt. The error rates are computed over 92 runs on eight programs, and no interval in this paper accounts for the clustering of units or of runs by program, or for run-to-run variation. Most root causes were labelled by a model and the others from their build errors, from a fixed list, and we did not check the labels against a second labeller. The style guides are the same for every program and restate IBM's file-status rules, which a plain prompt leaves to the model. The second Java guide was added after the proofs of three first-guide translations stalled, but it constrains only the structure of the code.

9 Conclusion

Kernel-checked proofs that LLM translations of whole COBOL batch programs behave like our semantics of the originals on every admissible input are within reach. Every standalone batch program of CardDemo that our front end covered at the version of the study now has a Rust and a Java translation with such a theorem. The theorems are relative to our semantics of the original. For Java they state partial correctness, and on a later version of our Java semantics the Java translations are also proved totally correct. The proofs did not replace testing, which found every behavioural error of the translations that we know of. Instead, they turned translations that passed the tests into translations with a guarantee over all admissible inputs, and the proof work, not the translation, dominated the cost. The main limitation is the trusted base. The theorems trust the front ends, our statement generator, our semantics of COBOL and Java and Aeneas's translation of Rust. The COBOL semantics is tested against GnuCOBOL and, in the IBM conventions used in the theorems, only in an exploratory run with listed gaps. It is not tested against IBM's compiler. Covering the remaining batch programs of CardDemo would widen the study, and testing the semantics against an IBM compiler would narrow its main limitation.

Data Availability

The artifact [48] (https://doi.org/10.5281/zenodo.23996137) contains the technical report, with the full text of the study, every re-check and the kernel replay behind each result; the source snapshots of our semantics, tools and proofs at the versions named in Appendix A and the technical report; the pre-registrations and the deviation log; and the summaries and hash lists of the checks. The LaTeX source of this paper is at https://github.com/tejasvi/proof-carrying-modernization.

Acknowledgements

Generative AI tools were used to write code, run analyses and draft text; the author directed the work and takes full responsibility for its content.

References

[1] Abdulaziz Abdujalilov. 2026. cobol-equiv: differential testing harness and a Python runtime for COBOL numeric semantics. https://github.com/abdulazizbalu/cobol-equiv, created 2026-10-13.

[2] Jesse Alama. 2026. Thales and Tarski: a proof tool for TypeScript and a JavaScript semantics in Lean 4. https://github.com/lakatos-ts/thales.

[3] Petr Bauch. 2026. circe: Lean-based verification tool for C++. https://github.com/petr-bauch/circe, public since 2026-09-21.

[4] Peter Baumann, Jürg Fässler, Markus Kiser, and Zafer Öztürk. 1993. *Beauty and the Beast or A Formal Semantic Description of the Control Constructs of Cobol and its Implementation.* Technical Report IFI-93.39. Universität Zürich.

[5] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, and A. Cheung. 2024. Verified Code Transpilation with LLMs. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*.

[6] Dominik Blain. 2026. IRONPROOF: COBOL-to-Python Transpilation with SMT-Based Equivalence Checking. arXiv:2610.11073.

[7] Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Principles of Programming Languages (POPL)*. ACM.

[8] A. Browne, E. Lee, and C. Salvi. 2026. Proving LLM-Generated GPU Kernels Correct and Safe: Floating-Point Verification in Lean and VerCors. arXiv:2610.24548.

[9] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (NFM) (LNCS)*. Springer.

[10] CERT Polska, Michal Majchrowicz, and Marcin Wyczechowski. 2026. CVE-2026-16554: cJSON `print_string_ptr()` 32-bit `size_t` wraparound. https://cert.pl/en/posts/2026/07/CVE-2026-16554/. Published 2026-07-27.

[11] H. Chang, G. Meng, S. Xiao, K. Chen, Y. Li, and K. Sun. 2025. When Code Crosses Borders: A Security-Centric Study of LLM-based Code Translation. arXiv:2509.06504v2.

[12] Yiannis Charalambous, Rafael Menezes, Youcheng Sun, and Lucas C. Cordeiro. 2026. Partial Contracts Suffice: Sound, LLM-Inferred Regression Verification. arXiv:2607.10291.

[13] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. doi:10.1145/3377813.3381347

[14] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification (CAV)*. doi:10.1007/978-3-319-96142-2_26

[15] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Roşu. 2016. A Language-Independent Proof System for Full Program Equivalence. *Formal Aspects of Computing* 28, 3 (2016), 469–497.

[16] V. A. Dasu, M. Santra, M. R. U. Rashid, A. Kumar, S. Tizpaz-Niari, and G. Tan. 2026. Heimdall: Formally Verified Automated Migration of Legacy eBPF Programs to Rust. arXiv:2605.25411v2.

[17] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 (LNCS, Vol. 12699)*. Springer, 625–635.

[18] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software: Theories, Tools, and Experiments (VSTTE) (LNCS)*. Springer.

[19] Y. Du, Y. Zhang, and M. Li. 2026. Hallucinations of Large Language Models in Cross-Language Code Translation: Phenomena and Mitigation. arXiv:2610.27868.

[20] DukeDeSouth. 2026. Fix arithmetic and control flow bugs in batch programs. CardDemo pull request #32, https://github.com/aws-samples/aws-mainframe-modernization-carddemo/pull/32. Opened 2026-02-24.

[21] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514.

[22] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Automated Software Engineering (ASE)*. ACM.

[23] A. L. Ferenczi, M. Bessonov, A. Ponnudurai, C. R. Mamillapalli, K. K. Maddula, and K. K. Lingamneni. 2026. Certifying Repeatable Entry-Replay Witnesses for COBOL-to-Java Migration. arXiv:2610.27272.

[24] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Design Automation Conference (DAC)*. ACM.

[25] C. Hathhorn, C. Ellison, and G. Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2737979

[26] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. 2025. Charon: An Analysis Framework for Rust. In *Computer Aided Verification (CAV) (LNCS)*. Springer, 377–391. doi:10.1007/978-3-031-98685-7_18

[27] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 711–741.

[28] imaami. 2024. v1.7.19 added Undefined Behavior to `cJSON_SetValuestring()`. cJSON issue #919, https://github.com/DaveGamble/cJSON/issues/919. Opened 2024-12-25; see also pull request #924 and issue #1106.

[29] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[30] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SymDiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification (CAV) (LNCS)*. Springer.

[31] Peter Lammich. 2012. Refinement for Monadic Programs. Archive of Formal Proofs, https://www.isa-afp.org/entries/Refine_Monadic.html.

[32] E. Lazaropoulos and Z. Paraskevopoulou. 2026. Foundational Refinement Proofs for EVM Bytecode, at the Price of Tokens. arXiv:2607.26306v2.

[33] K. Rustan M. Leino and Valentin Wüstholz. 2015. Fine-Grained Caching of Verification Results. In *Computer Aided Verification (CAV) (LNCS)*. Springer.

[34] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

[35] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Programming Language Design and Implementation (PLDI)*. ACM.

[36] V. Malík and T. Vojnar. 2021. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*. doi:10.1109/ICST49551.2021.00045

[37] Midspiral. 2026. LemmaScript: a verification toolchain for TypeScript. https://github.com/midspiral/LemmaScript.

[38] opencompl. 2026. VeIR pull request #1346: cir: add the CIR integer and core dialect. https://github.com/opencompl/veir/pull/1346, merged 2026-09-02.

[39] Marco Paganoni and Carlo A. Furia. 2023. Verifying Functional Correctness Properties at the Level of Java Bytecode. In *Formal Methods (FM) (LNCS)*. Springer.

[40] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639226

[41] L. Sarker, A. Satter, and T. Bultan. 2026. Quantitative Symbolic Patch Impact Analysis. In *NASA Formal Methods (NFM 2026) (LNCS)*. Springer, 367–390. doi:10.1007/978-3-032-28079-4_17
[42] seL4 project. 2026. l4v: seL4 proofs, `lib/CorresWhile.thy`. https://github.com/seL4/l4v.
[43] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Programming Language Design and Implementation (PLDI)*. ACM.
[44] son risa. 2026. CBACT04C (INTCALC): the last account's interest is written as a transaction but never added to the account. CardDemo issue #64, https://github.com/aws-samples/aws-mainframe-modernization-carddemo/issues/64. Opened 2026-10-03.
[45] Michael Tautschnig. 2026. Bound externally-controlled reference-modification lengths and subscripts. CardDemo pull request #54, https://github.com/aws-samples/aws-mainframe-modernization-carddemo/pull/54. Opened 2026-06-17.
[46] The LLVM Project. 2026. ClangIR: a high-level MLIR dialect for C and C++. https://llvm.github.io/clangir/; upstream llvm-project commit 4955d32b used here.
[47] Theorem Labs, PBC. 2026. OSS Verify: proofs and theorem statements about OSS-Fuzz fuzz targets, release 2026.10.0. https://github.com/theorem-labs/oss-verify-proofs. Tag v2026.10.0 (commit 8f5f467); read at commit 98e995d, 2026-10-23.
[48] Tejasvi Singh Tomar. 2026. Proof-Carrying Modernization: Artifact and Technical Report. Zenodo. doi:10.5281/zenodo.23996137
[49] H. Tu, H. Zhao, Y. Song, M. Zafar, R. Meng, and A. Roychoudhury. 2026. Agentic Verification of Software Systems. *Proceedings of the ACM on Software Engineering* 3, FSE, Article FSE157 (2026). doi:10.1145/3808164
[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot – a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*.
[51] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*. doi:10.1145/2349896.2349905
[52] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/2517349.2522728
[53] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. arXiv:2404.18852v2.

A Versions

Lean is v4.35.0 throughout. The versions listed are those of our frozen copies, on which our own checks ran; the technical report [48] lists the versions behind most other results.

- Migration theorems: equiv-lean `751764f`, whose migration modules are byte-identical to those we checked at `a572cf6`, with the COBOL semantics of the coblean snapshot `20261024T2306Z`, JimpleLean `20261025T0149Z`, Charon `de8736f` and Aeneas `fdebf20`. AWS CardDemo at `59cc6c2f`. GnuCOBOL 3.2.0, JDK 17, Soot 4.7.1.
- Termination of the Java translations: equiv-lean `9f0e5fb` with JimpleLean `20261025T2312Z`, the termination version. The composite theorems: equiv-lean `ba05f6e`, on the same versions. The refutations of three mutants' statements: equiv-lean `47d9ce3`, with JimpleLean `20261025T2312Z`.
- CBSTM03B: coblean snapshot `pin2306Z-cbstm03b-20261027T0112Z`; its translations: equiv-lean `cc9e349` (partial correctness for Java) and `0fd209e` (total correctness), with JimpleLean `20261025T2312Z` and Aeneas `fdebf20`. CBSTM03A with CBSTM03B: snapshot `pin2306Z-cbstm03ab-20261027T1251Z`; its translations: equiv-lean `a50f10e` (Rust) and `8ed869e` (Java), on the versions of the joint build. CSUTLDTC: snapshot `pinM3-csutldtc-20261027T2010Z`, on the semantics of snapshot `20261027T2008Z`. COBSWAIT: snapshot `pinM7-cobswait-20261028T0554Z`, on the semantics of snapshot `20261028T0551Z`.

- The joint build, which builds the migration theorems together with later results: equiv-lean `04d2599`, with cirlean `e6ac99a`, JimpleLean `20261026T2035Z`, StrictTS `a15926a`, coblean `20261024T2306Z` and Aeneas `fdebf20`; the composite theorems and the refutations are added in `89806a8` and `b46c478`, which change no file of `04d2599`.
- C semantics: the three later versions of Sections 5.2 and 8 are cirlean `e6ac99a`, `424e821` and `7c73739`; the last is the one frozen after the runs behind our theorems.
- C to Rust: equiv-lean `751764f`, cirlean `28e2380`, ClangIR from llvm-project `4955d32b`, Charon `de8736f` and Aeneas `fdebf20`, tiny-regex-c `f2632c6`. jsmn: equiv-lean `373874f` with the same tools, jsmn `25647e6`. Adler-32: equiv-lean `ff07aac` with the same tools, zlib 1.3.1 (`51b7f2ab`).

B Pre-registrations

The migration study and its measurement of translation errors followed a fixed protocol (Section 4.1) but had no pre-registration. The studies in the table below are pre-registered: a file with the procedure, the outcomes and the analysis was frozen before the runs it governs, together with a SHA-256 manifest of its input files (samples, commit lists, selection tools), and addenda fixed the samples and inputs left open by a file, each frozen before the runs it governs. Later decisions went into an append-only deviation log, and each is reported where it affects a number. For a run with *inputs recorded*, the inputs and the tool commit were fixed and hashed before it started, but no analysis was frozen. A result is *exploratory* if it lies outside the migration study and every pre-registration and we logged it as such, with its date, in the deviation log, where such results are tagged "labelled". The files, the log and their SHA-256 hashes are in the artifact. Because no hash was deposited with a third party, this record is self-attested.

File	Study	Frozen (UTC)
`PREREG_C_v2.md`	security fixes, proof reuse	2026-10-24
`PREREG_D_v2.md`	random samples, first run	2026-10-24
`PREREG_C1b_v1.md`	security fixes, second pass	2026-10-25
`PREREG_D2_v1.md`	random samples, second run	2026-10-25
`PREREG_T_v1.md`	Thrift	2026-10-25
`PREREG_M_v1.md`	mutants of the semantics	2026-10-25
`PREREG_N_v1.md`	NIST COBOL85	2026-10-25
`PREREG_H_v1.md`	fuzz harnesses	2026-10-25
`PREREG_V_v1.md`	LLVM test programs	2026-10-25

C Model settings

All model calls go to one commercial LLM through its API, in October 2026, with a fixed reasoning-effort setting and at most 64,000 output tokens per call. Translation runs allow at most 10 attempts and $15 each, with 60 fresh generated cases per round and 300 for the final check, and seeds 1 to 3 for the input generator. The root-cause labels were produced by further calls with a fixed prompt, one per run in 30 of the 32 runs with behaviour repairs; repairs that only fixed build failures were labelled from the build error, without a model call. In the security-fix and sample studies, the orchestrator uses the same model for contracts, invariants and proof scripts, with three proposals per proof. The proofs of the migration theorems were written by LLM agents and checked by Lean (Section 4.5).