emd: A Language of Eight Unicode Dashes and How Far It Can Be Verified

Daniele Scaratti
Independent Researcher
ORCID: 0009-0008-3931-8821
DOI: 10.5281/zenodo.24011442

November 5, 2026

Abstract. emd programs use eight Unicode dashes and 32 instructions. We define the language in Lean 4 and Rocq, proving encoding round trips, run determinism, step invariants and independence from line layout. A direct Turing-machine simulation proves universality using 11 instructions, preserves the final tape and never faults. In Lean we also prove a model of the 64-bit machine a refinement of the semantics, correctness of a mirror of the compiler in both directions, and soundness of a mirror of the static analyzer. These results concern models. Kani checks the Rust VM and CBMC its generated C, within stated bounds. The VM is compared with the models on finite cases, and the generated C with the VM; no proof connects the implementation to the models. A compiler written in emd compiles itself.

1. Introduction

emd is a stack language written in eight visually similar Unicode dashes. Two code points encode an instruction; a look-alike outside the alphabet is an error. The syntax is the joke. The rest of this paper develops the language and examines how much of its semantics and implementation can be verified.

The central question is how the different forms of evidence fit together: proofs about formal models, bounded checks of implementation code, and finite comparisons between the two. A language this small makes their scope and the gaps between them explicit.

We contribute:

- two formal models of the language, in Lean 4 [4] and Rocq [5], with proofs of encoding, determinism, step invariants and independence from line layout;
- a direct Turing-machine simulation that proves universality with 11 of the 32 instructions;
- Lean proofs about models of the 64-bit machine, the compiler and the static analyzer, and a Rust implementation tied to them by bounded verification and comparison on finite cases;
- a compiler written in emd that compiles itself, checked against the Rust compiler.

The proofs cover the models. None of this evidence proves the Rust VM correct, and we say where each claim stops.

2. Related Work

Esoteric languages explore unusual programming constraints [1]. Böhm's P″ [6] and Brainfuck show how small a universal language can be. Forster, Kirst and Smolka formalize undecidability reductions in Coq [7]; our Turing simulation concerns emd's own step function.

CompCert [8] proves semantic preservation from Clight to assembly, and CakeML [9] verifies an ML implementation through machine code. emd does not: its theorems concern a model of the code generator. We combine model proofs with bounded checks of the Rust through Kani and of the generated C through CBMC [10, 11], differential testing [12, 13], mutation testing [14], and delta debugging [15]. The analyzer uses abstract interpretation [16].

Unicode confusables and display controls can conceal source-level distinctions [2, 3]. emd uses a fixed alphabet and rejects look-alikes outside it; these acceptance rules are part of the specified decoder.

3. The Language

3.1 Alphabet and instructions

The alphabet has eight symbols, numbered by code point: U+002D, U+2010, U+2011, U+2012, U+2013, U+2014, U+2015 and U+2212. A pair (f, o) of symbols encodes an instruction: f selects the family and o the operation. Of the 64 pairs, 32 are assigned (Table 1), a count proved in both models (`assigned_count`); the rest are decoding errors.

Table 1: Instruction families. Family 7 is unassigned.

f	Family	Instructions (in order of o)
0	Constants	`PUSH` o, for $o \in 0..7$
1	Stack	`DUP DROP SWAP OVER ROT`
2	Arithmetic	`ADD SUB MUL DIV MOD`
3	Memory	`LOAD STORE`
4	Control	`JMP JZ JNZ HALT CALL RET`
5	I/O	`READ WRITE`
6	Logic	`EQ LT GT NOT`

The machine has a stack of integers, memory indexed by naturals and initially zero, a program counter, and byte streams for input and output. Constants above 7 are built with arithmetic, and jumps pop an offset relative to the jump's own index. A program ends only through `HALT`: running off the end is a fault. `CALL` pops an offset, pushes $pc+1$ and jumps; `RET` pops an absolute address and jumps to it. A return address is thus an ordinary stack value,

which a routine moves aside with `SWAP` or `ROT` to reach its arguments, and the machine needs no separate call stack.

3.2 Source layout

A source is lines separated by U+000A, each at most 32 code points. Before decoding, every line break is removed. A break can therefore fall between instructions, inside an instruction, or anywhere else, and carries no information. Spaces, carriage returns and all other code points are errors. Hello World in its canonical layout, which breaks each line at the limit:

```
----------------------------------
--------------------------------
----------------------------------
----------------------------------
----------------------------------
----------------------------------
--------------------------------
-----------
```

The limit forces programs into dense blocks in which the line structure looks meaningful and is not.

4. Formal Semantics

4.1 Machine

Both models use unbounded integers and the state

$$\Sigma = (pc, S, M, I, O),$$

where S is the stack, M memory, and I, O byte streams. A step has type

$$\mathit{step} : P \times \Sigma \to \{\mathsf{next}(\Sigma'),\ \mathsf{halt}(\Sigma'),\ \mathsf{fault}(f)\}.$$

The function is total, so there is no undefined behavior by construction. A fault is one of stack underflow, division by zero, negative address, pc out of range and invalid byte. `ADD` faults with fewer than two operands. `DIV` truncates toward zero and `MOD` takes the sign of the dividend. Execution with fuel, *run* $P\ n\ \Sigma$, iterates the step. For example, when $P[pc] = \mathsf{ADD}$ and $pc + 1 < |P|$,

$$\begin{aligned}&\mathit{step}\ P\ (pc, b :: a :: r, M, I, O)\\ &\quad = \mathsf{next}(pc + 1, (a + b) :: r, M, I, O).\end{aligned}$$

4.2 Encoding and determinism

Each of the following is proved in both Lean and Rocq, under matching names:

- Instruction and program round trips:

  $$\mathit{decode}(\mathit{encode}\ P) = P.$$

  Conversely, $\mathit{decode}\ l = P$ implies $\mathit{encode}\ P = l$ (`encodeProgram_decodeProgram`), so a program has exactly one flat source.
- Injectivity of the code point map and of program encoding.
- Determinism of complete runs. A halting result is unique across fuels (`run_halt_unique`), halting and faulting exclude each other across fuels (`run_halt_fault_exclusive`), and more fuel changes nothing after a run ends. From an initial state, the same program and input give the same output and stack (`run_observable`).

Determinism of the step is immediate, since it is a function. The stronger statements about whole runs are what the reversible debugger of Section 6 relies on informally.

4.3 Step invariants

A successful step keeps pc inside the program (`step_pc_lt`); output only grows and input only shrinks (`step_output_prefix`, `step_input_suffix`); memory changes only on `STORE` and only at its address (`step_mem_frame`, `step_store_mem`); and the depth of the stack changes by the declared effect of the instruction (`stack_effect`). A `CALL` whose target is a `RET` returns to the following instruction with the stack it started from, minus the offset (`call_ret_returns`).

4.4 Layout independence

decodeSource takes code points, rejects a line past 32 code points, removes the breaks and decodes. We prove, in both provers:

Theorem 1 (`line_break_irrelevant`) *If $p +\!\!+ q$ fits the line limit, then*

$$\begin{aligned}&\mathit{decodeSource}(p +\!\!+ [\mathit{U{+}000A}] +\!\!+ q)\\ &\quad = \mathit{decodeSource}(p +\!\!+ q).\end{aligned}$$

Also: a run of more than 32 code points without a break makes any source fail (`overlong_rejected`), as does a carriage return (`carriage_return_rejected`) or any other foreign code point; a trailing newline changes nothing; and formatting a program at the limit decodes back to it (`decodeSource_format`).

5. Universality

5.1 Counter machines

A first development in Lean compiles counter machines to emd, keeping register r in memory cell r, and proves that a counter machine halts from zero registers if and only if its compiled program halts (`Emd.halts_iff_init`). Universality then follows from Minsky's classical result [17], which that development does not formalize.

5.2 Direct simulation of Turing machines

To remove that dependence we simulate Turing machines directly. A machine is a finite table $T[q][a]$ of rules, each writing a symbol, moving left or right on a two-way infinite tape, and choosing a next state; a state or symbol outside the table halts. Let w be a word written from position 0, blank elsewhere.

Definition 1 *Tape position $h \in \mathbb{Z}$ is stored at*

$$\mathit{cell}(h) = h + 2h^2 + 1.$$

Memory cell 0 stores the head.

The map is injective from the integers to the positive naturals, and it takes no branch to compute, which simplifies the generated code. A prologue writes w to the tape. Each state has a dispatcher that reads the symbol under the head and jumps to its rule with a relative offset computed at run time, and each rule writes, moves and jumps to the next state. All constants have a fixed width derived from the size of T and w, so every block has a known length and offsets can be calculated.

Theorem 2 *For every machine T, word w and input i,*

$$\exists n\, s.\ \mathit{run}\ (\mathit{compile}\ T\ w)\ n\ (\mathit{init}\ i) = \mathsf{halt}\ s \iff \mathit{Halts}\ T\ \langle 0, 0, w\rangle.$$

(`Emd.Turing.halts_iff_init`. The blank tape case is `halts_iff_blank`.)

Three further results make this more than a statement about halting:

Theorem 3 *When the compiled program halts in state s, the machine halts in a configuration c with $M_s(0) = c.\mathit{head}$ and $M_s(\mathit{cell}(h)) = c.\mathit{tape}(h)$ for all h, with input intact and output empty. Conversely, a halting machine has such a run.*

(`machine_halts_of_final`, `final_tape_of_machine_halts`.)

Theorem 4 (`compiled_never_faults`) *No run of a compiled program, at any fuel, ends in a fault.*

Theorem 5 (`compile_uses`) *Every instruction of compile T w is one of `PUSH 0`, `PUSH 1`, `PUSH 2`, `MUL`, `ADD`, `SUB`, `LT`, `LOAD`, `STORE`, `JZ`, `HALT`.*

So these 11 of the 32 instructions, with no `READ` or `WRITE`, already suffice. The compiler was written to avoid `DUP` and `JMP` for this reason: it reads the head three times rather than duplicating it, and every jump is a `JZ` on a zero left underneath. We have not proved the fragment minimal.

The Rocq proof is constructive, as discussed in Section 7.

5.3 What universality means here

These theorems concern the models, with unbounded integers and memory. The Rust VM has 64-bit integers and limits on stack and memory, and exceeding them is a declared fault, so it is a bounded implementation of a universal language and not itself universal. The theorems also relate halting and tape to machine configurations; they do not formalize a separate notion of computable function.

6. Implementation

6.1 Virtual machine

The VM follows the model instruction by instruction, with checked 64-bit arithmetic and explicit limits that are faults of their own. Each instruction checks operands, then values, then the destination, and only then changes state, so a fault leaves the stack, memory and pc as they were. This discipline came from a property test that found a fault altering the stack.

A second Lean model, written from the Rust in its order of checks, adds 64-bit values and limits on steps, stack and cells. It takes the same transition as the semantics except through its own faults: overflow only if the semantic result does not fit 64 bits, a limit only on a full table (`bexec_ref`, `bexec_overflow`). Within the limits and 64 bits a whole run is the semantic run (`brun_complete`), and a run stopped by a limit agrees with it up to that step (`brun_limit_prefix`).

6.2 Compiler

A readable assembly language with labels, constants and strings compiles to dashes. The interesting part is jump layout: an offset is built from instructions, its length depends on its value, and its value depends on the lengths of all jumps in between. The compiler finds a fixed point in which each jump owns a slot that only grows, padding with even-length no-ops.

Lean proves the following on a mirror of the code generator, which covers integer synthesis, padding, the fixed-point loop and labeled jumps and calls. Any result of the loop has a consistent layout (`layoutLoop_ok`). If a source has no raw jump and no `RET` that leaves its statement, the lowered program halts, faults or diverges exactly when the source does, with the same stack, memory and streams (`generate_correct`). A source `CALL` pushes the lowered address of the next statement, and that relation is what makes `RET` meaningful across the two levels. Termination of the loop is not proved.

The mirror is tied to the Rust by a comparison on whole programs, byte for byte: `generate` and an executable built from the proved definitions agree on the examples and *1200* random programs.

The compiler also exists in emd. `compiler/asm.asm` reads source bytes, validates UTF-8, lexes, parses, lays out jumps with the same fixed point and writes the encoded program. It compiles its own source to the same bytes in *42.5 million* steps, and so does the program it produces. Agreement with the Rust compiler is checked for both successful compilation and rejection: valid sources must produce identical bytes, and invalid sources must produce an error on the same line. The corpus includes the examples, the compiler itself, *400* random programs and *10,000* random texts, valid or not; no discrepancy is observed. It reports only the line, without a message, and reads sources up to 1 MB. The interpreters extracted from the two models also run it, so its semantics is the proved one and not only the VM's: compiling three examples in Lean's interpreter and one in Rocq's, the slower, gives the same bytes. Nothing about it is proved. It is tied to the Rust compiler by this comparison, as the Rust compiler is tied to its mirror.

6.3 Static analysis

`emd analyze` is an abstract interpretation over the whole program. A value is a set of at most 512 possible integers

or, past that, an interval, and memory is tracked per cell, with a store through an interval of addresses leaving the cells outside it untouched.

A program point is analyzed once per stack depth that reaches it, up to 32 depths, so routines called from different depths do not blur together; the return addresses of a routine form a value set, and `RET` follows each of them.

Values also carry a relation to a symbol, the value some instruction produced when it last ran: an offset from it, or the outcome of a comparison with a constant. A branch on such a value narrows the symbol wherever it is held. In `READ DUP PUSH 1 ADD JZ end`, this rules out -1 past the jump, and it keeps a loop counter bounded by the test that ends the loop.

On all the examples, the analysis proves every property it checks: no stack underflow, *pc* always in range, known jump targets, no division by zero, no negative address and only bytes to `WRITE`. This includes a Brainfuck interpreter of 825 instructions, for any input and any Brainfuck program given to it. Two earlier versions of the examples did fault: the truth machine at end of input and the interpreter when moving left of its tape. Both were corrected after the analysis pointed at them.

What it cannot determine, it reports as such: a jump to a value past what it enumerates, recursion past the depth limit, and loops past a budget, where bounds are widened. Overflow on values not known exactly, resource limits and termination are outside what it checks.

The analyzer's regression tests look for concrete executions that escape its abstract predictions: stack and memory values must remain contained, control-flow edges must be represented, and faults must be reported. The comparison covers *3000* random programs and *6000* programs built around aliasing loops, targeting the interactions between stored values and loop-carried relations, and, without sampling, every program of up to three instructions, *33,825* in all, each on four inputs.

Lean proves soundness on a mirror of the domain and of the transfer function, value sets, intervals, memory and relations included. A symbol is handled by an environment quantified in the concretization, so no history of the run is needed. The search loop and the widening are not proved; a checker, `checkResult`, validates what they recorded. A result that passes the checker describes every state reached by the bounded machine (`reachable_covered`), so a fault of the model has its issue reported (`fault_reported`) and a program with no reported `HALT` never halts (`no_halt_reported`). The Rust and the mirror give the same reachable points, issues and halting verdict on the examples and *1600* random programs, and the checker accepts each result. This is agreement on those programs, not a proof about the Rust.

6.4 Reversible debugging and other tools

Since the machine is deterministic, a past state needs no snapshots: running the same program on the same input from the start reaches it again. `emd debug` steps backward and continues backward this way, and can report which step last wrote a memory cell. A compiler to C gives native binaries with the VM's semantics and limits, compared with the VM under the undefined behavior sanitizer. Other tools give a control flow graph, a profiler, and a reducer in the style of delta debugging [15]. A Neovim plugin provides a clickable keyboard, a live debugger and diagnostics from the same binary. A website runs the real parser and VM through WebAssembly, so there is no second implementation of the language to keep in step.

7. Verification Evaluation

7.1 Coverage

Table 2 summarizes the evidence per component. A cell states the strongest evidence of that kind.

7.2 Proofs

The proof audit checks for unfinished obligations and identifies the assumptions on which the results depend. Neither development contains `sorry`, `Admitted` or a declared axiom. Across *413* Lean theorem declarations, helpers included, `#print axioms` reports only `propext`, `Quot.sound` and `Classical.choice`; the last occurs in *152* declarations. All *146* Rocq theorems and lemmas are closed under the global context according to `Print Assumptions`. Thus the Rocq results, including the Turing simulation, require no axioms.

The bounded machine, the compiler and the analyzer are proved in Lean only. The two models were written by the same author from the same specification and no proof relates them. Their agreement is observed through the comparison below, and a mistake in the specification would pass both.

7.3 Bounded verification

Kani checks a real step of the Rust VM against an independent specification written in the harness, on each instruction: stack depth exactly *0 to 3* of any `i64`, program length *1 or 2*, *pc* inside it, step count up to a limit of *4*, stack limit up to *3*, cell limit up to *2* and at most *1* input byte. The specification states the exact stack, *pc*, memory, input and output, and the fault if any, using `i128` arithmetic; on a fault nothing else has changed. Three restrictions remain, because wider runs do not finish or run out of memory: `DIV` and `MOD` take `i16` operands plus `i64::MIN` and `i64::MAX` over divisors from -2 to 3; memory is empty, except for `LOAD` over one cell at a fixed address, at depths *0 and 1*; and `STORE` over an existing cell is not covered. Streams never fail, so I/O faults are not reached. Within these bounds, the *127* VM harnesses establish agreement with the specification; *three* further harnesses cover the encoding core, *one* covers `decode` and *five* cover the compiler. Sensitivity to implementation errors is checked by injecting *nine* VM defects, including an incorrect `ROT` order, `CALL` pushing *pc*, `RET` accepting the program length and `DROP` popping before faulting. Each defect violates its corresponding harness, providing evidence that these checks constrain the intended behavior.

On the compiler, a single harness over the whole of `push_int` does not finish in *40 minutes*, so the proof is

Table 2: Verification coverage. Proved: machine-checked for all inputs, on a model in Lean or Rocq. Checked on the code: Kani on the Rust, CBMC on the generated C, for all inputs within stated bounds. Compared: on finite cases, against another implementation.

Component	Proved	Checked on the code	Compared
Encoding	Lean and Rocq: round trips, injectivity, 32 pairs	Kani: the whole encoding domain	Valid programs, in all comparisons
Decoder and layout	Lean and Rocq: `decodeSource` theorems	Kani: `decode` preserves symbols	648 raw sources in 3 decoders
Semantics	Lean and Rocq: run determinism, step invariants	Kani, through every VM step (row below)	Lean and Rocq, each against Rust
Universality	Lean: counter machines and a direct Turing simulation. Rocq: the direct simulation	A property of the language, with no code of its own	Shown by proof, not by runs
VM	The models above, which it is compared with	Kani, one-step agreement with a specification	2371 sampled and 36866 exhaustive cases, agreement with each model
Bounded machine	Lean only: refines the semantics	Kani, on the VM it models	Through the VM comparisons
Compiler	Lean only: a mirror of `generate`, both directions, faults and divergence	Kani: integer synthesis for every accepted literal, by induction on base-7 digits; its length below 2^{24}; jump layout up to *4* items	Same bytes as the mirror on 1207 programs; 5148 random programs against a source interpreter
Compiler in emd	No proof; it runs in the interpreters extracted from Lean and Rocq with the same output as the VM	Runs on the VM, which Kani checks	Same bytes or error line as the Rust compiler on 10,408 texts, itself included
Analyzer	Lean only: soundness on a mirror, through a proved checker	No harness	All *33,825* programs of up to *3* instructions on *4* inputs; 1607 mirror and 9000 VM comparisons
C backend	No model of C	CBMC: no undefined behavior in any of the 32 instructions from any stack of *0 to 3* values, nor in 3 pairs over memory and I/O	Examples and 150 random programs against the VM, under UBSan

split by induction on the base-7 digits of the magnitude. The base case checks the eight literals. The step checks, for every magnitude from 8 to `i64::MAX`, that the instructions appended after those for $n/7$ leave exactly n, with every intermediate value inside 64 bits. A third harness checks the sign. Together they cover every value except `i64::MIN`, which the parser rejects. That the code is the concatenation the induction assumes is read from `push_magnitude`, three lines; tests back it, with the same bytes as the Lean model around every power of 7 and VM runs on 404,000 values. The predicted length is checked against the same recurrence below 2^{24}; on all of `i64` the solver does not finish, and unit tests check every power of 7 instead. The jump layout is checked for up to *4* items of up to *20* instructions with two jumps: every slot holds its offset, and the addresses add up.

CBMC checks the C that `emd c` writes. Each of the other *31* instructions is compiled alone, followed by `HALT`, and run from a stack of *0 to 3* values that CBMC leaves free over all of `int64_t`, with free input. CBMC finds no out-of-bounds access, invalid pointer, division by zero, signed overflow or undefined shift on any path, loops unwound to the end. Three pairs, `STORE STORE`, `STORE LOAD` and `READ WRITE`, start from a memory table of *2* slots, so that the second store grows and rehashes it; from the real start of *64* slots the paths are too many. A single instruction says nothing about long programs: those rest on the comparison with the VM.

7.4 Differential testing

Differential testing checks agreement between the Rust implementation and executable interpretations of the formal semantics. Lean's interpreter is built through its compiler; Rocq's is extracted to OCaml. Both agree with the Rust VM on *2371* cases drawn from the examples, *1000* random programs and directed boundary cases. A further *36866* runs, covering every program of at most *three* instructions, also produce no disagreements with either model.

Decoder agreement is checked separately, including on malformed input rather than only on valid programs. Across *648* raw sources covering every line-break position, invalid UTF-8, look-alikes and damaged random layouts, all three decoders accept the same *458* sources and reject the same *190*.

These comparisons support agreement within the models' shared execution domain. *Four* sampled cases fell outside that domain and were excluded; the comparisons therefore provide no evidence about 64-bit overflow or resource limits. Those require the bounded checks described above.

7.5 Tests, fuzzing and mutants

The workspace's *180* tests combine byte-exact execution of every example with the differential comparisons above. Their role is regression detection across the implementation and tools. Bounded verification provides complementary evidence: Kani exposed a real discrepancy in remainder semantics, where `i64::MIN % -1` faulted on overflow although the model gives zero.

Fuzzing complements these semantic checks by looking for crashes on generated inputs. The decoder, compiler, VM and diagnostics targets each ran for *120 seconds*, exercising respectively *47.3 million*, *4.1 million*, *1.4 million*

and *0.6 million* executions without a product crash. This is evidence about robustness over the exercised inputs, not semantic correctness.

The tests exercise *99.83%* of product lines and *100%* of functions, excluding test files; six components reach *100%* of lines, among them the command line, the C backend and the analyzer, and the lowest is the decoder at *99.27%*. Line coverage identifies unexercised code but does not establish whether assertions distinguish correct behavior from defects. Mutation testing addresses that sensitivity for a selected fault set: the suite detects all *138* hand-written mutants. *Three* are detected by timeout because they make execution hang. Since the mutations are hand-picked, this result supports detection of these fault classes rather than a general estimate of defect detection.

8. What Is Not Verified

Rust against the models. No proof connects them. The equation $\mathit{execute}(\mathit{decode}(s)) \equiv \mathit{spec}(s)$ holds only on the compared cases and within the Kani bounds.

The mirrors. The bounded machine, the compiler mirror and the analyzer mirror are read from the Rust by hand. Only finite comparisons tie them to it, and an edit to one must be repeated in the other.

The compiler. The lexer, the parser, `CONST` and label resolution are outside the theorems, as are raw jumps, a `RET` that leaves its statement and 64-bit overflow in lowered code. Termination of the layout loop is not proved. The Turing compiler, by contrast, is the object of its own proofs, but it exists only in the models.

The analyzer. The search loop and the widening are validated, not proved, and the claim that a fault is *certain* is not proved. Overflow on values not known exactly, resource limits and termination are not checked.

Kani and CBMC. A VM harness covers one step, on a small state. Division takes narrow operands, a load sees at most one existing cell, and a store over an existing cell is not covered, because wider runs run out of memory. On the compiler, the length of an integer is checked only below 2^{24} and the jump layout only up to *4* items. CBMC sees one instruction at a time, or a pair, never a long program.

The trusted base. We trust the Lean and Rocq kernels, the Lean compiler for one interpreter, Rocq extraction and OCaml for the other, Kani with CBMC, and `rustc`. The readers around the extracted interpreters are hand-written.

Tools. The debugger, and the C backend beyond single instructions, rest on tests. Tests also showed their limit: the proof attempt on the analyzer, not a test, found a case where it claimed too much, through a stale relation on a value copied in a loop, and so "proved" the absence of a division by zero on a program that faulted. It was corrected and the test generator extended to that shape.

9. Availability

The project repository is `https://github.com/lupodevelop/emd`. It holds the implementation, proofs, examples and verification scripts under the MIT license. The verification entry point is `scripts/verify.sh`. A companion study asks whether language models can write emd [18].

10. Conclusion

The Lean and Rocq models prove encoding properties, deterministic execution, step invariants, layout independence and direct Turing universality with 11 instructions. Lean also proves a model of the bounded machine, a mirror of the compiler and a mirror of the analyzer. Kani, CBMC and comparisons support the Rust implementation and its C output; they do not prove it equivalent to any of these. A compiler written in emd compiles itself.

References

[1] M. Mateas and N. Montfort, "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics," in *Proc. Digital Arts and Culture*, 2005.

[2] N. Boucher and R. Anderson, "Trojan Source: Invisible Vulnerabilities," in *Proc. USENIX Security Symposium*, pp. 6507–6524, 2023.

[3] The Unicode Consortium, "Unicode Technical Standard #39: Unicode Security Mechanisms," Unicode 18.0.0, revision 34, 2026.

[4] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Proc. CADE-28*, LNCS 12699, pp. 625–635, 2021.

[5] The Rocq Development Team, *The Rocq Prover Reference Manual*, version 9.3.0.

[6] C. Böhm, "On a Family of Turing Machines and the Related Programming Language," *ICC Bulletin*, vol. 3, no. 3, pp. 185–194, 1964.

[7] Y. Forster, D. Kirst, and G. Smolka, "On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem," in *Proc. CPP*, pp. 38–51, 2019.

[8] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[9] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A Verified Implementation of ML," in *Proc. POPL*, pp. 179–191, 2014.

[10] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying Dynamic Trait Objects in Rust," in *Proc. ICSE-SEIP*, pp. 321–330, 2022.

[11] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. TACAS*, LNCS 2988, pp. 168–176, 2004.

[12] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Proc. PLDI*, pp. 283–294, 2011.

[14] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[15] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[16] P. Cousot and R. Cousot, "Abstract Interpretation: A Uni-

fied Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. POPL*, pp. 238–252, 1977.

[17] M. L. Minsky, "Recursive Unsolvability of Post's Problem of 'Tag' and Other Topics in Theory of Turing Machines," *Annals of Mathematics*, vol. 74, no. 3, pp. 437–455, 1961.

[18] D. Scaratti, "Generating Executable Code in a Visually Opaque Unicode Language: A CLI Case Study," preprint, Zenodo, 2026, doi:10.5281/zenodo.24011466.