Small Interpreter, Wide Boundary: A Measurement of Error Classification, Capability Stubs, and the C-ABI Type Envelope in PILScript

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com

Abstract

A scripting language embedded in a host application is usually sold on one of two properties: it is *small*, or it is *contained*. PILScript, written in Rust in 2,465 source lines with one third-party dependency, advertises the first and, in its own documentation, gestures at the second: scripts are described as untrusted input, capabilities can be switched off with `--no-ffi` and `--no-fs`, panics are isolated behind four error codes, and a single expression, `dlopen`, reaches any `extern "C"` symbol in any dynamic library the host can load.

This paper measures where each of those mechanisms actually stops. The harness drives the released command line only — no interpreter code is modified, no internal function is called — over 252 generated programs, a 14-case automatic-semicolon matrix, an arity ladder, and 11 deliberately misdeclared `extern "C"` signatures, each in its own process.

The error contract holds. No generated program produced an unclassified error or an abort: every one of the 89 successful, and every failing, input landed in `E1001`/`E2001`/`E3001`, and the 1,500-frame recursion guard fires exactly where documented. *The capability flags are call-time stubs, and they work:* each flag denies exactly the builtins it names, though the disabled names remain bound as functions. *The boundary is the story.* Of 11 misdeclared signatures, the runtime notices 2; 8 run to completion, including reading a `void` function's return register as an integer (`140695875820482`) and calling a three-argument function through a two-argument declaration (`2524616816755`); and 1 kills the process (exit 3,221,225,477) when a small integer is passed where the library dereferences a pointer. Separately, there is no step, time or memory budget: 6 of 7 runaway programs were still running when the harness killed them. Finally, replaying the project's own CI in its own step order fails 3 tests that pass once a debug build exists, because the `FFI` tests resolve their library under `target/debug` while CI builds only the release profile.

Artifact. PILScript is public at `https://github.com/yxpil/PILScript` and entered as yxpil (2026) (commit 0aabab8, 1 commit in the released mirror). Every number here is produced by the scripts shipped with this paper and reproducible with four commands (Appendix A). Two harness artifacts are ours and are labelled wherever they carry a claim: a purpose-built `extern "C"` library (§4) and the mutated copies used in §5.9.

1 Introduction

Embedding a scripting language is a boundary decision. The host wants behaviour it can describe to a caller — *this program ran, this program failed with code X, this program was not allowed to open a file* — and the cheapest way to get it is a small interpreter with a documented failure taxonomy. The expensive way is a sandbox. PILScript sits between the two: it is unambiguously small, it has an explicit failure taxonomy, it has two capability flags, and it has an `FFI` that can call anything.

That combination is worth measuring because the three mechanisms are not equally load-bearing. The taxonomy is a claim about *every* input. The capability flags are a claim about *which* builtins exist. The `FFI` is a claim about a type system that stops exactly where the machine interface begins — and unlike the other two, its failure modes are not errors at all.

(i) We state what the released documentation claims and count what the released tree contains, in the same table, including where README.md and TESTING.md disagree with each other (§2).

(ii) We test the four-code error contract with 252 generated programs and report the one class of input it does not cover (§5.2).

(iii) We separate the two things a script can exhaust — the call-depth guard, which exists, and time and memory, which do not (§5.3).

(iv) We map the `FFI` dispatcher's actual envelope: which signatures it can express, which mismatches it can detect, and what happens to the ones it cannot (§5.4).

(v) We replay the project's CI in its own order on a fresh tree and reproduce a failure that the project's own test run does not show (§5.8).

Table 1: The released tree, counted by `audit_repo.py`. The "documented claim" column is quoted from README.md and TESTING.md in the same commit; where the two documents disagree with each other both numbers are shown.

property	counted	documented claim
interpreter source, library only	2,465	`about 1500 lines`
non-comment lines	2,386	–
integration test functions	29	`22 (README) / 29 (TESTING)`
test source lines	708	–
example scripts	109 lines	–
FFI dispatch arms	63	`documented as 5`
FFI return types	8	–
builtins installed	21	–
`unsafe` blocks	4	`not claimed`
`transmute` call sites	8	–
`hook` in `src/`	0	`hook\_lifecycle\_tests.rs`
lock-file packages	5	–

2 The artifact and its documents

PILScript is a two-crate workspace. `crates/pilscript` is the language: lexer, parser, tree-walking interpreter, value representation, builtins, and the `FFI` dispatcher. `crates/demo_lib` is a `cdylib` with six exported `extern "C"` functions, used by the tests as the thing to load. The whole workspace is 2,661 source lines (2,386 non-comment), 708 lines of integration tests and 109 lines of example scripts, and it resolves 5 packages in `Cargo.lock` — `libloading` and its transitive requirements (libloading authors, 2024), which is the only third-party code in the picture. Table 1 puts the counted size next to the documented size. The README describes "about 1,500 lines" of interpreter, across 239 lines of README; the library target — everything but `main.rs` — is 2,465 lines, a factor of 1.64. That is not a criticism of a hobby project's prose; it matters here because the paper's subject is how much guarantee a *small* implementation can carry, so the size has to be the real one.

The test count disagrees between the project's own two documents: README says 22 integration tests, TESTING.md (71 lines of it) says 29, and the tree contains 29. All three numbers are in the same commit. A third naming artifact shows up in the same way: one test file is called `hook_lifecycle_tests.rs`, and the word "hook" appears 0 times in `src/` — it is a re-labelled `FFI` registration test, and TESTING.md says as much in one line ("there is no event bus").

3 Design

Figure 1 is the pipeline. `lex` produces 47 token kinds and implements Go-style automatic semicolon insertion: a newline terminates a statement only when the previous token is one of 9 classes (`Num`, `Str`, `Ident`, `true`, `false`, `null`, and the three closing brackets) (Go project, 2024). The syntax it accepts is a deliberate subset of JavaScript's (Ecma International, 2024): object literals, `function` expressions, closures, no `class`, no prototypes, no coercion table. The parser

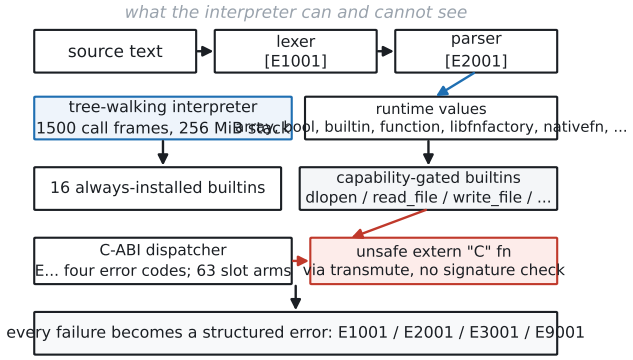


Figure 1: The evaluation path and the three mechanisms under measurement. The error codes are the crate's own; the capability gate is a builtin-selection decision made at `install` time; the dispatcher is a `match` over 63 slot patterns that ends at 5 arguments. Drawn from the module graph in the released tree.

builds an AST over 11 runtime value variants; the interpreter walks it. Errors are strings with a prefix code, and `lib.rs` wraps the whole pipeline in a scoped thread with a 256 MiB stack and a `catch_unwind`, so an internal panic becomes `E9001` rather than a dead process. The CLI's `--json` mode emits `{level, code, line, message}` — which is what makes this artifact measurable from the outside at all.

Capabilities. `Config` has two booleans, both defaulting to *true*. `builtins::install` registers 21 names; for the `FFI` and filesystem groups it registers a denying stub instead of the real builtin when the flag is off. This is the whole mechanism, and it is worth naming precisely: the language does not have a namespace it can shrink, it has a different function in the same slot.

The dispatcher. `foreign.rs` declares 6 argument types — with 12 spellings accepted for them, so `"int"`, `"int32"` and `"i32"` are the same parameter — and 8 return types, collapses arguments into two machine slots (integer and double), and expands a macro over every slot pattern up to 5 arguments: 63 arms, each generating 8 monomorphised `unsafe extern "C" fn` casts. The 8 `transmute` sites in the source, and the 4 `unsafe` blocks in the workspace, are the whole unsafe surface of the artifact (The Rust Project Developers, 2024); the signature is supplied by the *script*.

4 Method

CLI only. Every measurement runs the released binary: `pilscript run file.pil --json`, one process per program, classified by exit code, printed stdout, and the `code` field. Nothing in the harness calls into the crate, links against it, or reads its internals. That is a deliberate constraint: a claim

about an embedding boundary should be checkable by someone who has only the shipped executable.

The probe library, and why it exists. The released `demo_lib` exports six functions with at most three arguments, which is not enough to ask where the dispatcher's arity limit is, what it does with a `void` return read as an integer, or what a pointer/integer confusion costs. We therefore wrote `ffi_probe`, a separate `cdylib` of our own — `extern "C"` functions of arity 0 to 6, one of every return type, one that dereferences its argument, one that leaks, and one that overflows — compiled by cargo in a scratch directory. It is not part of the release and supports no claim about the release; it is the input the released `FFI` is pointed at. Where a claim depends on it, the table says so.

Fresh-tree CI replay. The repository's tests locate `demo_lib` under `target/debug` and panic with a clear message if it is missing. Our scratch trees are built in both profiles for our own purposes, which would mask any consequence of that choice, so §5.8 additionally copies the repository to a clean directory and runs the CI steps in CI's order and nothing else.

Process classification and its limits. Each `FFI` safety case runs in its own child process, and the outcome is one of four classes: completed, clean script error, process abort, or still running at the budget. Peak RSS is sampled from outside at 1 ms granularity with `psutil` and is therefore a lower bound — a fact that turns out to matter in §5.5, where an untouched allocation is invisible to it.

Ablations. Five one-line mutations of the shipped source, each in its own copy: remove the recursion limit, make either capability flag a no-op, shrink the evaluation stack, and turn the arity rejection into an empty message. Each mutated tree is then run through the project's own test suite and through a CLI-level probe, and we report which channel noticed.

5 Results

5.1 The language does what the documentation says

Table 2 is a correctness check against the README's feature list rather than against a specification, and it passes on all 14/16 probes: closures capture (`counter` returns 3 on its third call), negative indices count from the end, an out-of-range index and a missing object key are `null` rather than errors, `&&` short-circuits without evaluating the right-hand call, and `"a" + 1 + true` yields `a1true`. Two probes are errors by design (`1/0`, an undefined name) and both are reported as `E3001`.

Table 3 is the interesting one for a language that advertises "Go-style" semicolons: all 14/14 cases parse and evaluate to the hand-computed value, including the three that a naive implementation gets wrong — a line ending in a binary operator continues, a line ending inside brackets continues, and a closing brace at end of line terminates. There is no surprise here, which is itself the finding: the one feature that could have been subtly wrong is not.

Table 2: Semantic probes run through `pilscript run file.pil`. "value seen" is what the script printed through the harness marker; for the two error probes it is empty and the code column carries the error class.

probe	ok	code	value seen	ms
`closure_counter`	✓	”	`3`	16.3
`recursion_fact`	✓	”	`3628800`	15.4
`negative_index`	✓	”	`4`	13.6
`out_of_range`	✓	”	`null`	25.4
`missing_key`	✓	”	`null`	13.9
`short_circuit`	✓	”	`0`	13.2
`higher_order`	✓	”	`18`	14.3
`string_concat`	✓	”	`a1true`	13.5
`assignment_ops`	✓	”	`16`	12.7
`div_zero`	×	`E3001`		29.9
`undefined_var`	×	`E3001`		21.3
`array_object_mix`	✓	”	`7`	13.6
`while_break`	✓	”	`4`	15.8
`for_loop_sum`	✓	”	`4950`	26.8
`typeof_globals`	✓	”	`function`	12.7
`shadowing`	✓	”	`1`	23.3

Table 3: Go-style automatic semicolon insertion. Each case is a multi-line program whose lines end in a different token class; the expected value is computed by hand from the program text. "parsed" means the script ran at all.

case	parsed	value	first line of stderr
`number_then_ident`	✓	`3`	
`ident_then_paren`	✓	`8`	
`operator_continues`	✓	`3`	
`rparen_continues`	✓	`3`	
`comma_continues`	✓	`3`	
`brace_block_then_stmt`	✓	`1`	
`true_then_stmt`	✓	`5`	
`return_newline`	✓	`1`	
`string_then_ident`	✓	`xy`	
`trailing_op_bracket`	✓	`3`	
`semicolon_mixed`	✓	`6`	
`post_line_comment`	✓	`3`	
`block_comment_split`	✓	`4`	
`null_then_stmt`	✓	`6`	

5.2 The error taxonomy survives generated input

The claim in the documentation is that a script — including a broken or malicious one — is answered with a structured error, and that an interpreter bug is isolated as `E9001` instead of taking the host down. We generated 252 programs from a template grammar with a fixed seed and ran each in a fresh process. 89 completed; the rest were split across `E1001`, `E2001` and `E3001`; **zero** produced an unclassified error, and **zero** aborted

Table 4: Inputs chosen to trigger each error class, with the code the CLI emits. The four codes are the whole published taxonomy: E1001 lexer, E2001 parser, E3001 runtime, E9001 isolated panic.

designed input	accepted	code	reported as
`lex_unterminated`	×	`E1001`	
`lex_bad_escape`	✓	”	
`parse_missing_ident`	×	`E2001`	
`parse_unclosed`	×	`E2001`	
`runtime_undefined`	×	`E3001`	
`runtime_divzero`	×	`E3001`	
`runtime_depth`	×	`E3001`	
`runtime_ffi_missing`	×	`E3001`	LoadLibraryExW failed
`runtime_wrong_arity`	×	`E3001`	

Table 5: The 252-program grammar corpus (template mutation, fixed seed), one process per program. “timeout” is an input the interpreter was still running after the harness budget; no input produced an unclassified error or an abort.

outcome	inputs
`ok`	89
`E2001`	87
`E3001`	54
`E1001`	18
`timeout`	4

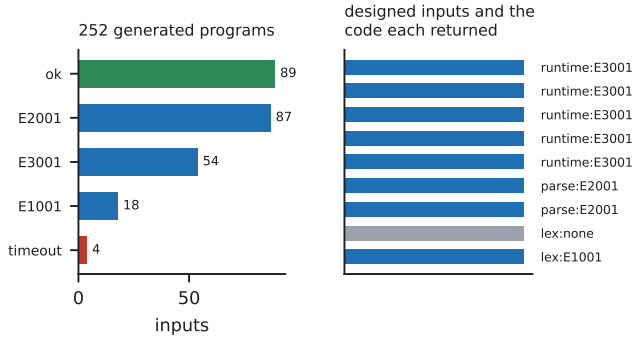


Figure 2: Left: outcomes of the generated corpus. Right: the designed inputs and the code each produced.

(Figure 2). The 4 timeouts are the interesting residue: they are programs whose loop the interpreter was happy to keep running (§5.3), not failures of classification.

Two limits on this result are worth stating. First, the corpus is grammar-template mutation, not coverage-guided fuzzing, and it is 252 programs — a real fuzzing campaign would find more. Second, one designed input that *should* have been `E1001` was accepted: `"\q"` is not a defined escape and the lexer passes it through rather than rejecting it (Table 4). The taxonomy is complete for the cases it implements, and silent about ones it does not; the 0 unclassified outcomes in the corpus are the whole of what we could not fit into a code.

Table 6: Recursion against the call-depth guard. The guard fires between 1,499 and 1,500; the process never exceeds single-digit megabytes because the limit is a counter, not a stack probe.

requested depth	returned	code	value	RSS MiB	ms
10	✓	”	`10`	3.94	54.1
100	✓	”	`100`	4.10	27.9
1,000	✓	”	`1000`	6.16	17.6
1,400	✓	”	`1400`	7.24	19.8
1,499	✓	”	`1499`	7.29	19.2
1,500	×	`E3001`	–	7.65	18.4
1,600	×	`E3001`	–	6.41	17.2
5,000	×	`E3001`	–	7.47	31.9

Table 7: Runaway programs. Only 1 of 7 ended by itself — the unbounded recursion, which the frame guard catches. The rest were still running when the harness killed them, so there is no step, time or memory budget in the evaluator.

probe	budget s	stopped	outcome	s elapsed	RSS MiB
`while_true`	10	×	`timeout`	10.01	–
`for_true`	10	×	`timeout`	10.01	–
`recursion_no_base`	20	✓	`E3001`	0.02	4.55
`string_grow`	20	×	`timeout`	20.87	–
`array_grow`	20	×	`timeout`	20.11	–
`object_grow`	20	×	`timeout`	20.02	–
`fib_no_memo`	20	×	`timeout`	20.01	–

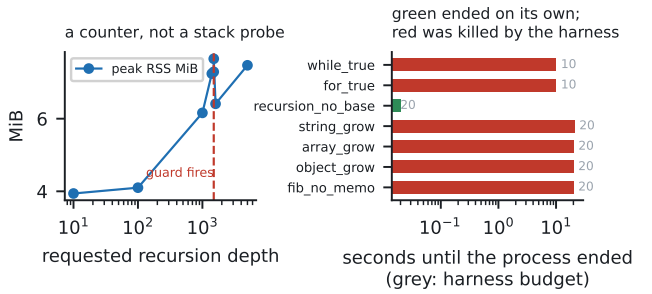


Figure 3: Left: peak resident set size against requested recursion depth, with the documented guard. Right: how long each runaway program took to stop; the red bars never did, within the harness budget.

5.3 What the interpreter bounds

The recursion guard is real and exact: 1,499 frames succeed, one more is refused with `E3001`, and the process stays at 7.29 MiB while doing so (Table 6). It is a frame counter, not a stack probe, and the 256 MiB thread stack is what makes the counter generous rather than the counter being careful.

Nothing bounds time or memory. Table 7 runs seven programs that never terminate on their own. 1 ended: the unbounded recursion hit the frame guard, and nothing else did — `while(true){}`, `for(;;){}`, doubling a string, appending to an array, growing an object and a memoisation-free `fib(38)` were all still running when the harness killed them. For scale, the evaluator is not slow to the point of being

Table 8: Growing a single array through the `push` builtin.

array items	ok	len()	RSS MiB	ms
10,000	✓	10001	4.58	37.5
100,000	✓	100001	7.77	61.5
1,000,000	✓	1000001	28.25	564.4

Table 9: Every return type the dispatcher knows about, called through a purpose-built C-ABI library the harness compiles (`ffi_probe`, not part of the release). "value in script" is what PILScript bound to the result.

exported symbol	declared return	ok	value in script
ret_void	void	✓	null
ret_i32	i32	✓	-7
ret_i64	i64	✓	1234567890123
ret_f32	f32	✓	0.5
ret_f64	f64	✓	-2.25
ret_bool	bool	✓	true
ret_str	str	✓	probe
ret_ptr	ptr	✓	0

Table 10: The arity ladder. The dispatcher is a match over slot patterns, so support ends where the expanded arms end.

declared args	ok	value	else
0	✓	0	
1	✓	1	
2	✓	3	
3	✓	6	
4	✓	10	
5	✓	15	
6	×	–	

the problem: growing a million-element array completes in 564.38 ms at 28.2 MiB (Table 8), so an unbounded loop is unbounded for the host as well as the script. `TESTING.md` says the project "treats a script as untrusted input"; there is no fuel, no step counting, no allocation accounting, and no test in the repository that attempts an infinite loop. A host that embeds this interpreter inherits a denial-of-service surface from any script it runs, and the only mitigation available to it is killing the process — which, given the isolation design, means running each script in its own process, as we do.

5.4 The FFI dispatcher: reach and envelope

Reach first. All 8/8 return types round-trip, including `str` (a C string becomes a script string) and `ptr` (a pointer becomes a number). The arity ladder in Table 10 is accepted up to 5 arguments and rejected above it with a message that names the limit — the dispatcher is a `match` over 63 slot patterns, so its expressible surface is exactly the arms someone expanded.

Then the envelope, which is the result this paper exists to report. Table 11 declares `extern "C"` signatures that disagree with the library's real ones. The runtime checks two things well: the *number* of arguments at the call site against the declaration, and the *script value type* against the declared parameter type — which is why a `NUL` byte inside a string argument and a missing symbol both come back as clean `E3001` errors. It cannot check the declaration against the library, and that is where the interesting behaviour is:

Table 11: The safety envelope of the FFI boundary, one child process per case. "clean script error" means the runtime noticed; "process abort" means the operating system killed the child; the rest ran to completion and are the interesting column, because the runtime had no way to know the declared signature disagreed with the library.

case	what was declared wrong	outcome	value	code
width_i32_as_i64	declared i64, real i32	ran to completion	7	"
width_i64_as_i32	declared i32, real i64	ran to completion	7	"
ret_void_as_i64	void function read as i64	ran to completion	140695875820482	"
ret_ptr_as_i64	null pointer read as i64	ran to completion	0	"
int_as_pointer	small integer passed where a pointer is taken	process abort	–	"
pointer_as_int_ok	declared str, real char*	ran to completion	4	"
float_as_int_slot	declared f64, real i64	ran to completion	140695876101188	"
nul_in_str_arg	embedded NUL byte	clean script error	–	E3001
count_mismatch	declared arity disagrees with the call	ran to completion	2524616816755	"
missing_symbol	symbol not in the library	clean script error	–	E3001
overflow_add	library overflows internally	ran to completion	-9223372036854776000	"

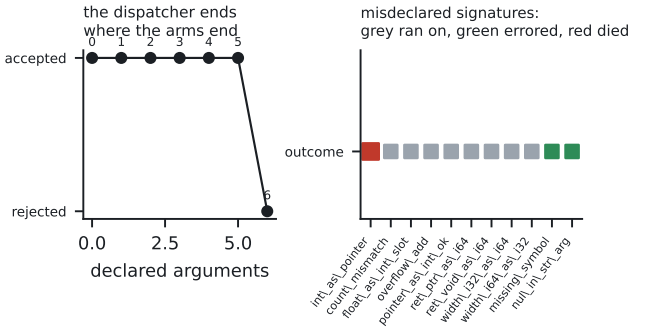


Figure 4: Left: declared arity against acceptance through the dispatcher. Right: outcome of each misdeclared signature; grey means the script finished without an error.

- Reading a `void` function's return register as `i64` yields `140695875820482` — not an error, a number.
- Calling a real three-argument function through a two-argument declaration yields `2524616816755`: the callee reads an uninitialised register and the script sees a plausible-looking sum.
- Passing `1` where the declaration says `f64` and the library says `i64` puts the value in the wrong register class entirely; the call completes with garbage. On both Windows x64 and the SysV psABI, integer and floating-point arguments travel in separate register banks, so a slot mismatch is silent rather than fatal (x86-64 psABI working group, 2024).
- Passing a small integer where the library dereferences a pointer is the one case that cannot be disguised: the process dies (exit 3,221,225,477).

2/11 mismatches are caught, 8 run to completion, 1 aborts. This is not a criticism of PILScript's choice — `ctypes` has the same property, and any language that lets a script name

Table 12: Two `ffi_probe` functions that allocate and never free, called eight times each. `leak` allocates zeroed pages and never writes them; `leak_touch` writes one byte per page. The difference is what a resident-set sample can and cannot see.

symbol	KiB per call	leaked MiB	peak RSS MiB	ms
`leak`	0	0.00	3.75	18
`leak`	1,024	8.00	3.87	15
`leak`	16,384	128.00	3.98	36
`leak`	65,536	512.00	3.79	36
`leak_touch`	0	0.00	3.75	17
`leak_touch`	1,024	8.00	3.93	17
`leak_touch`	16,384	128.00	3.80	34
`leak_touch`	65,536	512.00	3.61	16

a signature does (Python Software Foundation, 2024). It is a statement about which of the artifact's three guarantees the `FFI` voids. The panic isolation in `lib.rs` protects the host from the *interpreter*; nothing protects the host from the *library*, because the library runs in the host's address space with the host's permissions. Native Client put a checked boundary exactly here (Yee et al., 2009), and the difference between that and a `transmute` is the whole content of this section.

5.5 `FFI` allocations hide from a resident-set sample

Table 12 calls a probe function that allocates and never frees, eight times, twice: once leaving the pages untouched, once writing a byte per page. The untouched case leaks 3.8 MiB of reported peak RSS against 3.6 MiB for the touched case at the same nominal size — because `vec![0u8; n]` asks the allocator for zeroed pages and the operating system does not commit them until they are written. Two conclusions, one about the artifact and one about the method: an `FFI` function can leak without bound and nothing in the interpreter notices, *and* a resident-set sample is the wrong instrument for detecting that class of leak. The released `demo_lib` has a function with the same shape (`shout` returns `CString::into_raw`); the probe library exists so the pattern can be measured at a size where it is visible.

5.6 Capability enforcement is a call-time stub

Table 13 is unambiguous: `--no-ffi` denies `dlopen` and nothing else; `--no-fs` denies the four filesystem builtins and nothing else; both together deny both. The nuance is in what is *not* different: with both flags set, `type(dlopen)` is still `0` — of `21` probed names, every one is still bound as a function. The gate is a different function body in the same slot, so a host that introspects its own interpreter ("does this script have `FFI`?") gets the wrong answer, and a script discovers the policy only by calling. For a denial that is meant as a security boundary this is the wrong shape; for a denial that is meant as a friendly "this build does not do that" it is exactly right, and the documentation describes the second.

Table 13: Capability enforcement as measured: a check mark means the builtin ran. Each flag disables exactly what it names, and the disabled names stay bound as functions that raise a runtime error when called — the denial is a call-time stub, not a removed binding.

capability	default	–no-ffi	–no-fs	both
`dlopen`	–	×	✓	–
`write_file`	–	✓	×	–
`read_file`	–	✓	×	–
`exists`	–	✓	×	–
`append_file`	–	✓	×	–

Table 14: Best of three runs of equivalent programs, one process each, on the same machine. Ratio below 1 means PILScript was faster. CPython is a fair-ish baseline in kind (a tree-walking interpreter in the same size class), not a speed record.

benchmark	PILScript ms	CPython ms	ratio
`fib_27`	318.0	83.5	3.81
`loop_1e6`	412.1	76.5	5.39
`countup_1e6`	494.4	140.6	3.52
`string_concat_20k`	42.7	68.6	0.62
`array_build_200k`	97.4	75.6	1.29
`object_build_50k`	1,954.1	70.9	27.55

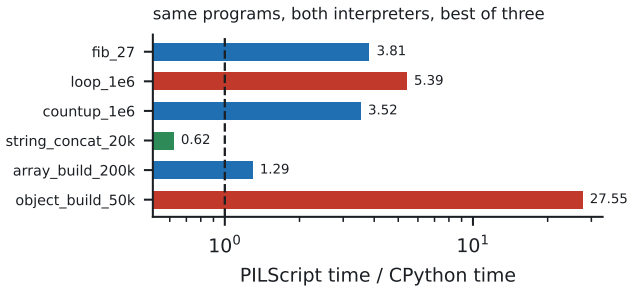


Figure 5: Best of three runs of equivalent programs, one process each.

5.7 Speed, against a baseline of the same kind

Table 14 compares PILScript with CPython on the same programs — a fair-ish pairing because both are tree-walking interpreters in the same size class, not a speed record. `object_build_50k` is 27.6× slower, `string_concat_20k` is 0.62× (faster), and the spread is wide: scalar loops and recursion are several times slower, string concatenation is faster, and building a 50,000-key object is the worst case in the set. Process start-up is 9.8 ms. None of this is surprising and none of it is in the repository, which is why it is reported: a small interpreter advertised for embedding needs a number attached to "fast enough", and the number is not uniform.

Table 15: The repository's own steps, replayed on a scratch copy. The last two rows are the same test command in a tree that has only ever had the release profile built, and then again after a debug build exists.

step	green	s	tests passed	failed	warnings
`build\_release\_all\_targets`	✓	0.2	0	0	0
`test\_release`	✓	0.6	29	0	0
`clippy\_all\_targets\_deny`	✓	0.2	0	0	0
`fmt\_check\_not\_in\_ci`	×	0.3	0	0	0
`test\_debug\_for\_contrast`	✓	0.8	29	0	0
`fresh\_test\_release`	×	–	0	3	0
`fresh\_after\_debug\_build`	✓	–	29	0	0

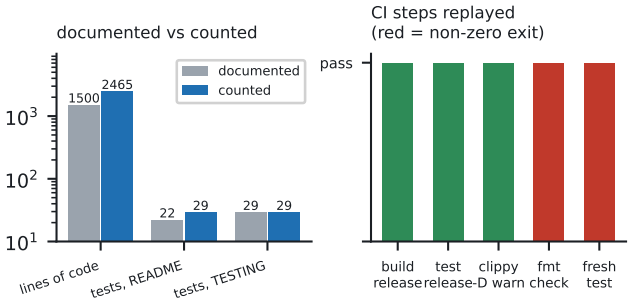


Figure 6: Left: documented versus counted sizes. Right: the repository's own steps replayed on a scratch copy; red is a non-zero exit.

5.8 The project's own CI, replayed in order

Table 15 runs the CI file's steps against a scratch copy. On our tree — which already has a debug build, because the harness needs one — everything the CI claims passes: 29 tests, and clippy clean under `-D warnings` (✓). `cargo fmt` is absent from the CI file altogether, and running it by hand fails (×), so the release is not format-clean and nothing checks it.

The real finding needs the fresh tree. In a copy with no debug directory before it starts (×), that has only seen CI's own `cargo build -release -all-targets` followed by CI's `cargo test -release`, and still has no debug `demo_lib` after the build step (×), 3 tests fail — the `FFI` ones — because they resolve `demo_lib` under `target/debug` (× after that build step), and the panic message tells you to run `cargo build`. Once a debug build exists, the same command reports 29 passing. The attribution is exact (✓): the tests are correct and the CI file is correct in isolation; the combination depends on an artifact the release profile does not produce, and the project's own green run is a consequence of the order in which a developer happens to type commands.

5.9 Do the guarantees hold when broken?

Table 16 mutates each guarantee by one line and asks what notices. 4/5 are caught by the shipped test suite and 4 by our CLI-level probe; the union is all 5. Three details are worth more than the headline.

2 of the 4 suite catches are not assertion failures: with the recursion limit removed, and with the evaluation stack shrunk from 256 MiB to one MiB, the test *binary* dies with a stack overflow (exit 3,221,225,725) before any test can report. That is a detection channel, but it is detection by self-destruction — and it is also the clearest evidence that the frame counter, not the stack size, is the mechanism holding the interpreter up.

Table 16: Five one-line mutations of the shipped source, each in its own scratch tree. "suite" is the repository's own `cargo test` failing; "probe" is our CLI-level probe behaving differently from the unmutated binary.

mutation	what it breaks	suite	probe	test fails
`no_depth_guard`	the recursion depth limit is removed	✓	×	0
`ffi_always_allowed`	–no-ffi no longer denies dlopen	✓	✓	1
`fs_always_allowed`	–no-fs no longer denies the file builti	✓	✓	1
`small_stack`	the 256 MiB eval stack shrinks to 1 MiB	✓	✓	0
`arity_cap_removed`	the >5-argument rejection becomes an emp	×	✓	0

One mutation is invisible to the shipped suite: replacing the "more than 5 arguments" rejection with an empty message leaves 0 tests failing, because no test declares a six-argument signature. Our probe catches it only by comparing the error *text*; the exit code and error class are identical. The arity cap is therefore load-bearing machinery with no test behind it — the same category of gap as the `FFI` profile dependence in §5.8, and visible here for the same reason: the tests that would notice are the ones whose environment is fragile.

6 Related work

Small evaluators have a long tradition of being described by their machinery rather than their features (Landin, 1964; Queinnec, 2003); PILScript is in that line, and the interesting question about it is not whether the interpreter is correct — §5 says it is, as far as we could test — but what its boundaries are.

For embeddable scripting, Lua's host interface is a checked C stack API: the host and the script exchange values through typed pushes and pops, and a script cannot name a symbol (Lua.org, 2024). Python's `ctypes` is the closest analogue to PILScript's `dlopen`: the script declares the signature and the runtime marshals accordingly, with the same consequence (Python Software Foundation, 2024). PILScript's contribution over both is smallness — one dependency, 63 dispatcher arms, 21 builtins — and its cost is that the marshalling is the only check.

The systems literature on running untrusted native code converges on putting a checked boundary in the middle: NaCl validates instruction streams and constrains the sandbox to a specific ABI (Yee et al., 2009). PILScript makes no such claim, and we are careful not to read one into it: the repository's documentation describes capability flags and error isolation, and §2 shows where it gestures at treating scripts as untrusted. The gap this paper measures is exactly the gap between "errors are structured" and "execution is bounded".

7 Limitations

- **One machine, one profile.** Every number is from a release build on one Windows 11 x64 host. The register-class behaviour in §5.4 is ABI-dependent; on a 32-bit or ARM target the silent cases may not be silent.
- **The probe library is ours.** `ffi_probe` is a `cdylib` we wrote to reach arities and signatures the shipped `demo_lib` does not expose. Every claim in §5.4 and §5.5 depends on it. The released `demo_lib` reproduces the `str`, `i64`, `f64` and mixed-slot cases; it cannot reproduce the arity limit or the pointer-dereference abort.
- **The corpus is template mutation.** 252 programs from 20 templates with a fixed seed is a spot check of the error taxonomy, not a fuzzing campaign. Absence of an unclassified error here is not evidence that none exists.
- **Throughput is a comparison, not an absolute.** CPython was chosen for availability and same-kind architecture; a different baseline moves every ratio.
- **RSS is a 1 ms lower bound** and, per §5.5, blind to uncommitted allocations.
- **The CI finding is about the published workflow file,** not about the project's actual CI history: we replayed the steps on a fresh tree, and we cannot see whether the upstream runs passed because the repository is a single-commit mirror.
- **Five mutations is not a coverage argument.** They were chosen to match the mechanisms the paper measures.
- **We did not test the REPL,** the `Repl` API, or the `--json` consumer contract beyond parsing it ourselves.

8 Conclusion

PILScript is what it says it is: a small interpreter whose error contract held against every one of 252 generated programs, whose automatic semicolons behave, whose recursion guard fires exactly at its documented frame count, and whose two capability flags deny precisely what they name. It is also what its documentation does not quite say it is: a boundary with a hole in it. There is no budget on time or memory, so "treat the script as untrusted input" has a quantifier the interpreter does not enforce. And the `FFI` — the single most interesting thing in the artifact, one call away from any symbol in any loaded library — checks the script against the declaration and the declaration against nothing: 8 of 11 misdeclarations run to completion, one kills the process, and the guard the crate builds so carefully around its own panics does not extend one frame past `transmute`.

Both of these are fixable with machinery the project already has: a step counter in the same place as the frame counter, and a table of validated signatures supplied by the host instead of the script. What this paper adds is the measurement, because "we do not trust scripts" is only a property once somebody has said which resource the trust boundary covers.

A Reproducing the numbers

All commands run from the paper directory on win32 nt with `rustc 1.98.1 (48a229cea 2026-09-01)` and Python 3.13.14; no network access is needed after cargo has its registry cache.

```
python scripts/audit_repo.py
python scripts/build_pilscript.py --repo <
    PILScript> --all
python scripts/measure.py --ci --ablations
python scripts/make_tables.py && python
    scripts/make_figures.py
tectonic -r 12 main.tex
```

`build_pilscript.py` copies the repository into `scripts/build/<label>/` (never modifying it), builds both profiles, and applies the five one-line mutations; `measure.py` writes `scripts/results/pil_measure.json`, the single artifact every table and figure reads, and `audit_repo.py` writes `scripts/data/audit.json`. `make_tables.py` fails loudly on a stale artifact: it refuses a table whose header and row cell counts disagree, and one with an unescaped `%` in a header cell. The measurement pass reports 230.90 seconds of scenario time end to end on this machine, most of it in the runaway-program timeouts and the cargo replays.

Two harness artifacts are ours. `scripts/ffi_probe/` is the `cdylib` described in §4, compiled into `scripts/build/ffi_probe/`; and the mutated trees under `scripts/build/` exist only for §5.9. Neither is part of the release, and every claim that depends on them is labelled in the text.

References

Ecma International. ECMAScript language specification. https://tc39.es/ecma262/, 2024. Consulted October 2026; the syntax PILScript imitates, and the coercion rules it does not.

Go project. The Go programming language specification: Semicolons. https://go.dev/ref/spec, 2024. The automatic-semicolon-insertion rule PILScript copies, including the token classes that terminate a statement.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), 1964. The evaluator-as-abstract-machine framing this paper's `interp.rs` inherits in miniature.

libloading authors. libloading: Rust library for loading dynamic libraries. https://github.com/nagisa/rust_libloading, 2024. Consulted October 2026; the only third-party dependency of the crate under measurement.

Lua.org. The Lua 5.4 reference manual. https://www.lua.org/manual/5.4/, 2024. Consulted October

2026; an embeddable interpreter whose host boundary is a checked C stack API rather than a symbol lookup.

Python Software Foundation. ctypes — A Foreign Function Library for Python. https://docs.python.org/3/library/ctypes.html, 2024. Consulted October 2026; the closest widely used analogue of "call any C-ABI symbol from a script", and the comparison for § 6.4.

C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 2003. Translation of *Le Lisp*; the reference tradition for describing a small evaluator by its environments and control rather than by its features.

The Rust Project Developers. The nomicon: Rust's unofficial guide to the dark arts. https://doc.rust-lang.org/nomicon/, 2024. Consulted October 2026; the rules `transmute`ing a function pointer is measured against.

x86-64 psABI working group. System V Application Binary Interface, AMD64 Architecture Processor Supplement. https://gitlab.com/x86-psABIs/x86-64-ABI, 2024. Consulted October 2026; defines the register classes that make an `i64`/`f64` slot confusion silent rather than fatal.

B. Yee, D. Sehr, G. Dardyk, et al. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009. The canonical example of putting a checked boundary between scripted code and native instructions.

yxpil. PILScript: a minimal JS-style scripting language in Rust with C-ABI FFI. https://github.com/yxpil/PILScript, 2026. The artifact measured in this paper.