Full-Stackless Computing:
A Multi-Language Compiler That Builds and Boots Its Own Operating System

B.S. Hartshorn
https://github.com/brentharts/crust

August 3, 2026

Abstract

We present *Crust*, a self-hosted compiler that accepts C, a subset of C++, a subset of Rust and a subset of rpython *in a single translation unit*, together with its own assembler, linker, freestanding runtime and operating system. After bootstrap the chain depends on no external toolchain: not GCC, not LLVM, not GNU `as` or `ld`, and not a host libc. The system is deliberately kept small enough that its entire call graph fits in the context window of a large language model, and we argue this constraint is what makes a system auditable in an era when the reader is increasingly a machine. As a worked example we take the removal of a layer usually treated as part of the machine — the return-address stack — and, with archived and reproducible measurements [20], show both a two-order-of-magnitude pitfall in the naive implementation and a regime in which abandoning the stack is faster than the hardware's own return predictor. The point of the example is not the speed but the epistemics: the pitfall, the fix, and the regime were each found, made, and checked inside a toolchain small enough to read whole.

1 Introduction

The prevailing model of software construction is a tower. A program sits on a libc, on an operating system, built by a toolchain of several million lines, itself built by an earlier version of itself. No participant can read all of it. The tower is trusted because it is old, not because it is understood — a distinction Thompson made unavoidable in 1984 [1].

Large language models change this in a specific way. A model can read, modify and verify a system, but only the part that fits in a context window. A ten-million-line toolchain is, to a model, exactly as opaque as it is to a person. A one-hundred-thousand-line toolchain can be held whole. This suggests a constraint that would have looked eccentric before 2023: *keep the entire system — compiler, assembler, linker, runtime, operating system — small enough to fit inside a single context window.* We call the result *full-stackless*: not a thin stack, but the removal of the distinction between layers.

That small systems are comprehensible is not new; Project Oberon [11] is the existence proof. What is new is the reason to care. Wirth's reader was a person with a semester; ours is a model with a window, and the window is a hard boundary.

2 What Crust is

Crust's front end accepts four languages; its middle and back ends are shared. A single file may contain all of them:

```
#include "mingine.rs"       /* Rust:    types with invariants   */
#include "mingine.py"       /* rpython: integer rules, curves   */

int main(void) {            /* C:       entry point, raw memory */
    Rect r = Rect_new(0, 0, 96, 48);
    mg_fill(r, rgb(30, 40, 90));
    return score_for(1, 3, 24);
}
```

This is not a foreign-function interface. `crust.py` lowers the Rust to C and `py2c.py` lowers the rpython to C, both *before the lexer runs*, so the compiler proper sees one C program. `Rect_clip` called from C is a direct call that inlines. There is no marshalling and no runtime cost for crossing a language that no longer exists by the time code is generated.

The argument for this is not interoperability but *density*: each language is short at something different, and a program written in the shortest available notation for each part is a smaller program — fewer tokens, more of the system in the window at once. C takes raw memory and linkage; Rust takes fixed layouts and index arithmetic; rpython takes rules and parsing; C++ takes destructors. A second criterion emerged during this work and is developed in §2.2: *a function belongs in the language whose checker can say the most about it.*

Position. The formal study of multi-language programs begins with Matthews and Findler [3], who model interoperation as explicit boundary terms between two calculi; the line continues through FunTAL [4], mixing a functional language with typed assembly. Crust occupies the opposite corner: they keep both languages and reason about the seam, whereas Crust *eliminates the seam* by lowering everything before compilation — and so has no boundary terms to reason about, and no cross-language guarantees beyond the target's. The Truffle/GraalVM line [5] also removes marshalling costs, but dynamically, at a shared object representation inside a VM; Crust does it statically, at a shared *source* representation, before any machine exists.

The empirical literature does not flatter multi-language development. Kochhar et al. [6] find across 628 projects that multi-language systems are measurably more bug-prone, and work dissecting cross-language bugs directly [7] locates much of the difficulty at interface boundaries. We take this seriously and observe that the mechanism identified is precisely the one Crust lacks: with no FFI, there is no interface at which two sides can disagree.

2.1 Differential agreement

Wheeler's diverse double-compiling [2] is this argument applied to the trusting-trust attack, and notes in passing that the method "detects … some compiler defects as well"; differential and metamorphic compiler testing in the Csmith [8] and EMI [9] line applies it to randomly generated programs. Our contribution is to note how far the discipline generalises when a system builds all of its own tools.

2.2 External checkers without external dependencies

A system that refuses external *dependencies* need not refuse external *opinions*. Three mbos modules — the shell's edit buffer and history ring, the heap's block bookkeeping, the ramdisk's ustar parser — are written in the Rust subset with two toolchains pointed at them doing different jobs. `rustc` **checks** them (`--emit=metadata`), producing nothing the kernel uses; `crust.py` **compiles** them, lowering to C that GCC builds into the image as an ordinary object. The shipped artefact depends on `rustc` not at all: remove it and the kernel still builds. What is lost is a check, not a capability. We think this distinction is underused — a project can take the benefit of a large external analyser while keeping it strictly outside the trusted build path.

The rule that emerged is simple: *if a function can be checked without a machine, it belongs in Rust.* Buffer index arithmetic, free-list splitting and coalescing, fixed-offset header parsing — exactly the code where an off-by-one silently corrupts memory, and none of it needs hardware. Drawing, port I/O and pointer arithmetic over physical memory stay in C, because no checker helps there.

We define a software-level secure system by properties unusual to see stated as requirements: small enough that all source fits in one context window, including the call graph; self-hosted; readable; a single translation unit, so the call graph is complete and static; and built by a system that is itself secure by these criteria. The fifth forces the others and is Thompson's argument [1] as an engineering constraint. The single-translation-unit criterion has precedent in unikernels [12], which specialise application and operating system into one image for much the same reason.

Wheeler's diverse double-compiling [2] shows the trusting-trust attack is detectable by compiling source with a second, independent compiler and requiring bit-identical output — a stronger guarantee than self-hosting alone provides, and one a system agreeing byte-for-byte with an external toolchain is unusually well placed to perform.

Related positions. TempleOS [16] is the closest prior example and underrated as engineering: by refusing the stack, Davis obtained a JIT almost for free and a system one person could hold in mind. Project Oberon [11] makes the case in an academic register. Our addition is a reason the argument is now forced rather than merely attractive, and a multi-language answer to the density problem those systems solved by designing a new language instead. Somlo [14] pursues a trustable self-hosting system down to the hardware, and reproducible builds [15] share our reliance on bit-identity as the acceptance criterion; where those minimise the *seed*, we minimise the *steady state*. seL4 [10] achieves guarantees we do not, and supports the size argument from the other side: verification became feasible precisely because the system is small. The mainstream alternative is to add layers — WebAssembly [13] places a virtual machine inside a browser that already contains several; meta-interpreter stacks add tiers in pursuit of performance they then spend on the tiers [18]. Our position [19, 17] is that these respond to a scale problem better solved by not having the scale. We note plainly what the mainstream has in its favour: GCC and LLVM encode decades of correctness work, hardware support and optimisation Crust does not have. The claim is narrower — that for a system intended to be *understood*, breadth is a cost.

3 The stack as a convention

"Full-stackless" can be read at two depths. The shallow reading is the one above: no tower of software layers. The literal reading reaches a layer usually counted as part of the machine — the call stack. The return-address stack is not a law of the hardware but a convention: a protocol in which `call` pushes an address the matching `ret` later pops, agreed to by caller and callee because a compiler emitted both halves. A system that owns its compiler may choose a different protocol where the default is not the cheapest one, and — the point of this section — may *measure* the choice and reason about it in full, because the code that emits a return is a few lines the reader already holds.

Crust exposes two mechanisms in this direction. `-fstackless-calls` lowers a statically-known, non-reentrant chain into jumps that build no per-frame return record. The `__metamorphic__` specifier goes further: the callee does not use the stack for its return at all. Each caller writes the address it wishes to resume at into a per-function slot and *jumps* into the callee; the callee returns by jumping through that slot. No return address is pushed or popped, and no `ret` executes.

The idea is old — TempleOS obtained a JIT almost for free by refusing the stack [16], and the trampoline is folklore. What a small, self-contained toolchain adds is not the idea but what one can *see* about it: we implemented the mechanism, measured it against ordinary `call`/`ret` on one machine, and found that the naive form is a disaster, the corrected form reaches parity, and there is a specific regime in which it wins outright. Every lowering in the benchmark is required to compute the same result before its timing is recorded — a differential check of the kind in §2.1, here used to keep an optimisation honest rather than to test a compiler. The programs and the full data are archived and reproducible [20]; we draw out the three findings that bear on this paper's argument.

3.1 The naive form is dominated by self-modification

The obvious layout places the return slot immediately before the function, in a writable, executable section — "memory near the function." It is catastrophic, and instructively so. The caller's store into the slot lands in the same cache line the processor is fetching the callee's instructions from, and the hardware's self-modifying-code detection answers with a machine clear on *every* call. In a tight loop this layout runs about two orders of magnitude slower than an ordinary call — on our test machine roughly 410 cycles per call against 2.7. The lesson is easy to miss without a cycle-level number one can attribute to a single line: the cost is not the extra store, it is *where* the store lands.

Moving the slot off the code page — into ordinary writable data, a separate page from any instruction — removes the machine clear entirely. The return then costs about 4.4 cycles, within a small constant of `call`/`ret`, and, as a side effect, the executable no longer needs a writable-and-executable segment at all: the body is plain read-execute text and the slot is plain read-write data. A further refinement — patching the return address into the immediate field of a register-load instruction and returning through the register — reaches about 3.5 cycles when the patch can be hoisted out of the loop; performed per call it re-enters the self-modification regime and its cost, which is why the hoist, not the narrow write, is the load-bearing part. The corrected mechanism is, in the end, a change of *which section a word is emitted into*. That such a change is both the whole fix and directly visible in the generated assembly is the property the paper is about: a reader with the back end in view can make it with confidence; a reader without it cannot even see it.

3.2 Where a jump beats a return

Parity is the ceiling one expects, and for a reason. A matched `call`/`ret` pair is predicted almost perfectly by the processor's return-stack buffer, a small hardware stack that shadows the software one; nothing that also touches memory beats a correctly predicted return. So the corrected mechanism ties, and one might stop there, banking a security and code-size property — no writable-executable memory — at no cost in speed.

The tie holds only while the chain is shallow. The return-stack buffer has a fixed depth, on the order of sixteen to thirty-two entries on current parts, and a chain deeper than the buffer evicts its own oldest entries; when it unwinds, the returns past the buffer's capacity are mispredicted, each paying a branch-misprediction penalty. A metamorphic return is an indirect jump whose target, for a given call site, is stable: it is predicted by the ordinary indirect-branch predictor and never enters the return stack, so it is immune to the depth limit. Table 1 shows the consequence. The two mechanisms cross near the buffer's capacity, and beyond it the stackless chain pulls away, to roughly a fivefold advantage at depth sixty-four. The crossover depth is itself a reading of the buffer's size, taken with nothing more than the compiler already in hand.

We state the limits plainly: the numbers are for one microarchitecture, the win requires call chains deeper than most code contains, and a production return predictor is a moving target. But the shape of the result is the paper's shape. It was found, corrected, and explained inside

chain depth	`call`/`ret`	metamorphic	faster
16	29	44	`call`/`ret`
24	84	64	jump
32	254	87	jump (2.9×)
48	567	127	jump (4.5×)
64	940	171	jump (5.5×)

Table 1: Cycles per outer iteration for a chain of N nested calls whose return addresses are loop-invariant, lowered two ways: ordinary `call`/`ret` and stackless metamorphic returns with the patch hoisted out of the loop. The crossover near the return-stack buffer's capacity is visible directly. One microarchitecture; full data and programs in [20].

a toolchain small enough to read; the pitfall of §3.1, the fix, and the crossover here are each a few lines of emitted assembly; and the reader who audits the compiler audits the claim in the same sitting.

3.3 Against inlining

The conventional way a compiler removes call-and-return overhead is to *inline* — to paste the callee into the caller until no transfer of control remains. It is the right tool for a small, hot, statically-known callee, and Crust uses it. But inlining is a technique whose cost grows with the size of what it removes. A large callee inlined at many sites multiplies code; a deep chain inlined through multiplies it multiplicatively; and every compiler stops, past a threshold, precisely to bound the instruction-cache cost of having succeeded. Inlining also cannot flatten a chain that is deep because its *logic* is deep — an interpreter's dispatch, a driver stack, a chain of continuations — without either failing to fit or paying back the cache cost it was meant to save.

The stackless return is orthogonal and size-independent. It is a few bytes regardless of the callee's size, it composes across indirect and separately-compiled calls, and its advantage appears in exactly the deep-chain case where inlining and the return predictor both give out. The claim is not that one replaces the other; it is that owning the whole toolchain lets a system inline the small leaves *and* abandon the stack for the deep structural chain, choosing per function and on evidence, rather than accept a single strategy fixed by a compiler it cannot see into. That freedom is a consequence of the size discipline, not an addition to it: the same property that lets a reader hold the system whole lets the system hold two strategies at once and pick between them where the machine, measured, says to.

4 Conclusion

Systems small enough to be read, that build themselves and check themselves against anything willing to disagree with them, do not merely make auditing possible. They make certain classes of bug unable to hide. The stackless-return study is the same point at the scale of a single optimisation: a two-order pitfall and a fivefold win that were only ever a few lines apart, both legible to the reader who holds the back end whole, neither reachable by one who does not. In an era where the reader is increasingly a model with a finite window, that legibility is not an aesthetic preference. It is the constraint that decides whether a system — or a claim about it — can be understood at all.

References

[1] K. Thompson, *Reflections on Trusting Trust*, Communications of the ACM 27(8), pp. 761–763, 1984.

[2] D.A. Wheeler, *Countering Trusting Trust through Diverse Double-Compiling*, Proc. 21st Annual Computer Security Applications Conference (ACSAC), pp. 33–48, IEEE, 2005.

[3] J. Matthews and R.B. Findler, *Operational Semantics for Multi-Language Programs*, Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 3–10, 2007. Extended version: ACM TOPLAS 31(3), Article 12, 2009.

[4] D. Patterson, J. Perconti, C. Dimoulas and A. Ahmed, *FunTAL: Reasonably Mixing a Functional Language with Assembly*, Proc. PLDI, pp. 495–509, 2017.

[5] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján and H. Mössenböck, *Cross-Language Interoperability in a Multi-Language Runtime*, ACM TOPLAS 40(2), Article 8, 2018. Earlier version: DLS 2015, pp. 78–90.

[6] P.S. Kochhar, D. Wijedasa and D. Lo, *A Large Scale Study of Multiple Programming Languages and Code Quality*, Proc. IEEE SANER, pp. 563–573, 2016.

[7] *Dissecting Real-World Cross-Language Bugs*, Proceedings of the ACM on Software Engineering, 2025. https://doi.org/10.1145/3715777

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

[9] V. Le, M. Afshari and Z. Su, *Compiler Validation via Equivalence Modulo Inputs*, Proc. PLDI, pp. 216–226, 2014.

[10] G. Klein et al., *seL4: Formal Verification of an OS Kernel*, Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 207–220, 2009.

[11] N. Wirth and J. Gutknecht, *Project Oberon: The Design of an Operating System and Compiler*, Addison-Wesley, 1992; revised edition 2013.

[12] A. Madhavapeddy et al., *Unikernels: Library Operating Systems for the Cloud*, Proc. ASPLOS, pp. 461–472, 2013.

[13] A. Haas et al., *Bringing the Web up to Speed with WebAssembly*, Proc. PLDI, pp. 185–200, 2017.

[14] G.L. Somlo, *Toward a Trustable, Self-Hosting Computer System*, IEEE Security and Privacy Workshops (SPW), pp. 136–143, 2020.

[15] C. Lamb and S. Zacchiroli, *Reproducible Builds: Increasing the Integrity of Software Supply Chains*, IEEE Software 39(2), pp. 62–70, 2022.

[16] T.A. Davis, *TempleOS*, 2005–2018.

[17] B.S. Hartshorn, *Beyond WebAssembly: Rethinking the Web Browser with Post-JavaScript, VM-Free Page Execution*, 2026. https://ai.vixra.org/abs/2607.0013

[18] B.S. Hartshorn, *Rethinking Meta-Interpreters for High-Performance Execution*, 2026. https://ai.vixra.org/abs/2606.0084

[19] B.S. Hartshorn, *Foundational Problems with Compilers and Operating Systems*, 7 July 2025. https://ai.vixra.org/abs/2507.0081

[20] B.S. Hartshorn, *Crust: The C + C++ Rust Multi-Compiler* https://doi.org/10.5281/zenodo.21777457