Escaping the Quicksand

A Call to Arms

PETER SEWELL, University of Cambridge, UK
JEAN PICHON-PHARABOD, Aarhus University, Denmark

Computing has been an astonishing success – but at a huge cost in business and societal risk Computing has transformed the world since its origins in the 1930s and 1940s. Advances in computer science and engineering have given us systems of remarkable performance, sophistication, and impact, building computing infrastructure that is now fundamental to modern society, across personal, economic, medical, and government spheres.

But it comes at a cost. There is the obvious development cost: building new systems to a sufficient quality to be marketable is expensive. More importantly, we pay hidden costs in technical debt and risk. Our infrastructure – the millions of lines of code that implement everything from our hardware designs to phones to cloud services – works well enough in normal use to be marketable, but it is riddled with flaws that all too often can be exploited, and, in an increasingly adversarial environment, all too often are. We have built systems that are far too complex to be fully understood, on a quicksand foundation of decades of legacy choices. This creates inertia and cost for maintenance and new development; it exposes us all to continual risk of malfunction; and it exposes us all to continual to risk of attack, at every scale from individual criminals to nation states.

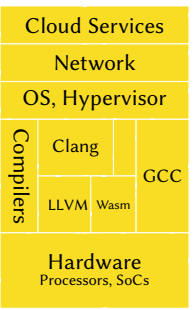

Fig. 1. Parts of the computing infrastructure that we all rely on

Imagine, if you don't want to sleep tonight, how long an outage of our infrastructure we could withstand without business or even societal collapse. Current attempts to mitigate such risks, in technical, social, and legal ways, are vital, but they are also a continual Red Queen's race that can never be won.

How did we get ourselves into this? By relying on test-and-debug development, focussing solely on code while neglecting specification, and misaligning market and societal incentives. Conventional engineering relies on prose specification and on test-and-debug development. We build systems from millions of lines of code, without precise descriptions of what it should achieve, or the reasons why it is correct. And we develop them by testing on concrete inputs, checking the results, and debugging and fixing any issues that reveals.

This is an expensive and ineffective feedback loop. Specifications are used to communicate between human beings, and prose does make them superficially accessible, but prose is intrinsically a poor medium for describing the subtle and complex behaviour of real systems. Prose descriptions are almost inevitably ambiguous and incomplete; they are often in some ways inconsistent or simply incomprehensible; and they do not directly support any mechanised use – one cannot directly test against them, let alone use them for test generation, or coverage analysis, or generation of implementation components, or static analysis, or proof. Prose specifications can thus only be very weakly tensioned against implementation; one needs to manual curate intended test-case results from the prose, and/or fall back to implicit specifications such as the absence of crashes.

Authors' Contact Information: Peter Sewell, University of Cambridge, UK, Peter.Sewell@cl.cam.ac.uk; Jean Pichon-Pharabod, Aarhus University, Denmark, jean.pichon@cs.au.dk.

Testing alone can cover only a tiny fraction of the astronomical number of execution paths and internal states of real systems. It manifestly can suffice to make systems that work well enough in common cases to market and use, but it manifestly cannot make systems robust or secure enough for today's adversarial environment.

Exacerbating the challenge is the scale and pan-industry nature of the fundamental components – the major processor architectures, ABIs (application binary interfaces), programming languages, compilers, network protocols, operating-system APIs, and so on. We have become locked into vast technical debt, based on design decisions taken in the 1960s and 1970s in a much less adversarial environment. Then, computing resources were scarce, we relied on them much less, programming languages and other protection technologies were much less developed, and malicious cyber-attack was almost unknown. The resulting designs were arguably good choices then, but they remain the basis for our infrastructure today: hardware provides protection only at coarse granularities, with virtual memory and privileged execution modes, and systems software remains largely written in C and C++. Recent years have seen a welcome focus on software and hardware memory safety, e.g. with Rust and CHERI[43], but the deployed base is so large that wholesale replacement is infeasible, and, even if we fixed memory safety, higher- and lower-level problems would quickly come to the fore.

These technical challenges are intertwined with mismatched incentives and a market failure. Vendors have huge incentives to bring new systems quickly to market, and thus to use the established infrastructure and engineering skills base, but that locks us in still further, while the risks fall largely on the end-users and on society as a whole. The option of reduced risk is usually not even available, but where it is, if it appears principally as a cost, then organisations often choose not to pay it.

AI will save us, right? AI-enabled engineering promises to amplify all of this, both the success and the costs and risks. AI coding promises to reduce coding costs, AI bugfinding finds many security vulnerabilties, and AI translation might reduce our dependence on legacy languages. All are very attractive, but together they seem much more likely to accelerate the Red Queen's race than to end it. Cheaper coding will lead to ever-larger codebases, and AI coding currently relies on the same ineffective test-and-debug feedback loop: ever-larger codebases will mean ever more opportunities for exploitable errors. AI bugfinding can catch some of these, but we have no reason to believe that it will catch all. The rapid pace of automated vulnerability detection serves the attacker just as well as (or even better than) the defender, and will place automated discovery of zero-days in the hands of many. Meanwhile, dependence on all these will mean that the human knowledge of the codebase, and the human skills to understand, engineer, and debug it, will atrophy: when the AI engineering is not up to the task, we will have no alternative. Above we argued that human engineering needs better feedback loops – but AI-based engineering needs them even more acutely.

Formal mathematics will save us, right? A long history of work, also from the 1930s–40s onwards, explores more rigorous development methods: mathematical specification of the desired behaviour of computer systems, more cleanly structured and less error-prone programming languages and other abstractions, and more discriminating ways to check that implementations have the desired properties, with richer type systems, static analysis, and proof.

For many decades, despite consistent advances and some impressive results, this struggled to gain widespread industry traction. Some ideas fed into practice, but mainstream engineering continued to rely on test-and-debug development, prose specifications (or none at all), and legacy languages. Our abilities to specify and verify systems improved massively, but were outstripped by the huge increases in system complexity; and there was a cultural disconnect between the classic formal methods and semantics community and mainstream practice – some of the former (from Dijkstra

onwards) emphasising formal approaches and proof *in opposition* to testing, requiring a radical shift to mainstream practice and quite different skills, rather than methods that complement and smoothly extend existing practice. The lack of approaches and tools that could be directly applied to mainstream development, the weight of legacy systems and skills, and the incentive structure emphasising rapid development and viewing improvements in engineering methods as a cost rather than a benefit, combined to impede improvement.

Pragmatic semantics, specification, and verification More recently, it has become clear that it is possible to do better, at scale, in pragmatic ways that can be more smoothly adopted into and impact mainstream engineering. Fig. 2 shows some recent success stories, rigorously defining abstractions at various levels in the stack, and using those to improve engineering – both in lightweight ways and in some cases with full correctness proofs, of the implementation of one interface above another. In this short paper we can mention just a sample, so apologies to those omitted. The 2024 Big Specification programme at the Isaac Newton Institute[1] included talks on many of these.

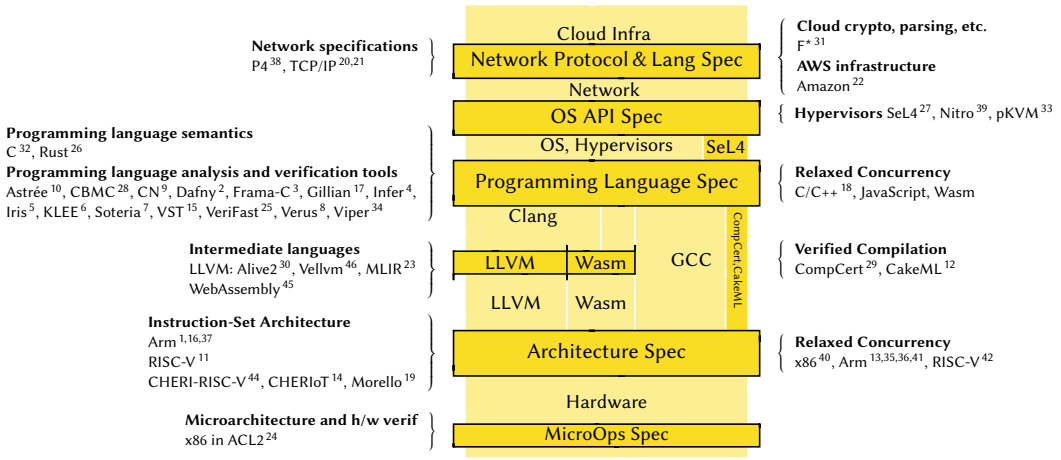

Fig. 2. Selected semantics success stories

Several advances have come together to permit this. Some are technical: advances in tools for mechanised mathematics (ACL2, CVC5, HOL4, Isabelle, Lean, Rocq, Z3, etc.), in logics, analysis methods, and tools for reasoning about programs, and in greater computing capacity. But a key advance is simply a change of mindset: focussing on rigorous specification, on the feedback loops that can enable, and on methods that can be applied to existing systems not just clean-slate redesigns.

Making specifications executable as test oracles One of the simplest, and perhaps the most under-appreciated, use of more rigorous specifications is to use them as test oracles. We say a specification is *executable as a test oracle* if, given some behaviour that might be exhibited by the system, the specification can be used to compute whether or not that behaviour is allowed.

Given an executable-as-test-oracle specification, and instrumentation to record the externally observable behaviours of an implementation, one can mechanically test whether the specification and an implementation are consistent, simply by running whatever tests one has and checking whether the specification allows the observed behaviours. This might be to check whether the

[1] https://www.newton.ac.uk/event/bsp/

implementation correctly implements the specification, or to validate whether the specification soundly abstracts from the implementation, or both. It makes the specification a live engineering artifact, that can be integrated into conventional development practice.

It also radically simplifies the construction of test suites: given such a specification, one can make tests automatically (systematically or randomly), without needing a manually written check or manually curated set of allowed results for every test.

A specification that is executable as a test oracle could take many forms. It could simply be executable assertions, or executable pre- and post-conditions or other contracts, or a program that takes a representation of a system behaviour (perhaps a trace) as input and checks whether it has some desired properties, or, in some cases, a reference implementation. It has to be mechanised in some way, but this need not be in an exotic specification language. Any of the above could be expressed in a conventional programming language – either the ambient language of the implementation, or a functional language chosen for its clarity – or in some mechanised mathematical form, equipped with some execution mechanism.

Executable and testable abstraction functions and invariants Specification and specification testing can be just for the top-level behaviour of some component, but it can also be much more fine-grained: one can define computable abstraction functions, that compute the intended abstract state from the current concrete implementation state, and check these step-by-step. This can be very discriminating, capturing and checking – even without proof – many of the reasons why a system behaves as intended. If one can examine and diff the abstract states, it also gives a powerful way to inspect a complex system.

Partial specification and a gentle on-ramp to full verification Specifications that support both testing and proof offer an approach to incrementally increasing assurance: improving both code and specifications with feedback from testing and property-based testing, which check that the code and specification are consistent in concrete executions, and then (if more assurance is desired) attempting proof. This is a simple and obvious idea that dates back to the 1970s and should have become routine practice long ago, but somehow has not. A long history of lightweight formal methods has advocated aspects of it and put some of the pieces in place, but it has not been part of main-line thinking. It could be done either code-first, incrementally adding partial specifications for existing code; or specification-first, as in some classic formal-methods approaches; or co-developing both code and specification. It could be done either where both code and specification are human-written, or to provide strong feedback loops for AI tooling. It provides a gentle on-ramp to full verification, with incremental benefits (and incremental demands on costs and skills) as one goes, rather than the more all-or-nothing character of traditional proof-focussed formal methods.

Design clarification Perhaps most importantly, writing a precise specification, tensioned against implementation, can help clarify the intended abstraction. It is all too easy to add a paragraph to a prose specification without considering all of its implications, and very hard to check that a prose specification is self-consistent, or that it covers all that it should. In contrast, when writing a more rigorous specification, one is forced to consider what it says in all cases. When coupled with any serious use of the specification, by testing or proof, this tends to expose open questions in the intent. We and others have found this in multiple cases: the relaxed-memory concurrency of architectures and programming languages, the semantics of C, the aliasing and provenance semantics of C, LLVM IR, and Rust, the undef and poison semantics of LLVM IR, and even the definition of PDF files. In each case the historical lack of precise specification has led to widespread misunderstandings and deep-seated mismatches baked into our infrastructure, which are expensive or even infeasible to fix up after the fact. For example, the ‘middle-end’ of GCC assumes that

the provenance of a pointer value matters, while the back-end assumes they are just machine integers – which are fundamentally incompatible choices of what optimisations are permitted. And no high-performance high-level language has a satisfactory definition of the allowed concurrent behaviour of "relaxed-atomic" racy programs.

Effective feedback loops for humans and AI All this suggests that more effective and efficient feedback loops for computer engineering are possible, for any combination of human and AI-driven development: much more discriminating testing loops, to reduce error and to improve assurance that specifications are as intended; and incremental development of specifications and code (either together or in either order) as a smooth on-ramp towards fine-grained specifications that support mechanised proofs of correctness. Mechanised proof can leverage the rapid advances in AI proof – and mechanised proofs are checkable by small non-AI prover kernels, to provide actual assurance even for AI-coded systems.

That sounds good – so what's missing? Some of what we advocate can be done immediately, with existing tools. For example, instead of conventional coding, or vibe-coding an implementation from a prose prompt, one can write – or in some cases generate – a rigorous specification in the ambient programming language, and generate both code and an executable abstraction function. Continuously testing that the code and specification are related by the abstraction function helps a lot to keep either human or AI coding on the straight and narrow.

But much requires a body of *semantics infrastructure*: well-validated and usable semantics of the abstractions on which the implemented system rests – be they assembly, C, Rust, OS APIs, or whatever, and testing/analysis/proof tooling for them. If one wants proof (AI or human) of C code, one needs a well-validated semantics of C, a specification language, and robust tooling for testing, test generation, and proof that the code and specification are consistent (as we've prototyped[9]). Or if one wants proof of Rust systems code that manages hardware-architecture security features such as virtual memory, down to the binary, one needs well-validated semantics of the underlying architecture, of the LLVM IR intermediate language, of linking and loading, and of the Rust static and dynamic semantics, and, again, tooling for testing, test generation, and proof that lets one relate each layer. We arguably should have developed this over the last 50 years, in sync with the evolving architectures, programming languages, compilers, protocols, etc., but we did not. For example, there is no specification language for the behaviour of C code that is remotely as well-established as C itself. The success stories in Fig. 2 provide some of what's needed, and they demonstrate convincingly that it's possible to build it, even for some of the gnarliest of legacy abstractions – it is possible to catch up. But each has demanded a major project by a substantial group of researchers over many years.

This is a challenge of scale, incentives, and market failure:

The normal **academic incentives and funding models** are at odds with the demands of such projects. They need long-term investment and commitment, to develop semantic artifacts at a larger scale, engineer them and the associated tooling to make them usable in production, and maintain them into the indefinite future.

The normal **industry incentives** are at odds with the fact that we depend on many *pan-industry* abstractions – for example, no one vendor controls or stands to benefit from improvements to the C specification, even though many would (directly or indirectly). A well-resourced company might build parts of what is needed for their own use, and some have, but that is not enough.

It needs **community consensus** on how to do it, to make integrated solutions. Only when one tries to integrate and use specifications does one discover how they should be phrased to make them actually usable, and only when one tries to relate multiple abstractions – e.g. to prove

correctness of compilation from Rust via LLVM to assembly and binary code – can one understand how best to set up all the definitions to make that possible. Isolated projects tackling different abstractions in different ways will not lead to semantic components that can be integrated later.

We now know much of how to do this, but many **research questions** remain. For example, ongoing work has integrated architecture-level instruction-set and concurrency semantics[35]; other ongoing work is examining how the aliasing and provenance models of C, Rust, and LLVM IR should be related; and the design of really effective *tool support for semantics* is an active topic.

It needs **education** of undergraduates, PhD students, postdocs, and industry practitioners, both of those who can develop this semantic infrastructure (a relatively skilled activity) and of those who need to use it in day-to-day development. Programming is widely taught, but where are the courses on specification, specification-based testing, and the whole gamut of rigorous engineering?

And it needs **care and support to maintain the research culture**. Both AI and semantics and verification are currently facing a success disaster: skilled researchers are in such high demand from well-resourced companies that it is becoming impossible to retain the next generation of researchers, to advance the fundamentals and educate the future. Traditionally limited academic salaries cannot compete with industry offers at many times their levels, for intellectually interesting and challenging work.

Other disciplines realised long ago that they needed to support "big science", for major biology, physics, or engineering research – e.g. the human genome project, or particle physics, or aeronautics, but computer science has not. The semantics infrastructure we need demands large resources by the standards of any one academic or industry research team – but only small resources by the standards of industry as a whole, and an infinitesimal fraction of the current AI spend. It is larger than (say) a typical ERC call, or UK ARIA or UKRI programme, or a DARPA programme, or academic funding from the large vendors (Amazon, AMD, Apple, Arm, IBM, Intel, Google, Meta, NVIDIA, etc.), or philanthropic funding, but it does not need another CERN: *fairly big science* would suffice to make a big step forwards: coordination of some or all of those potential funders and users, new social structures among researchers, and 100s of M$, not of B$, of funding.

So, this is a call to arms, to the research community and all those organisations, to collectively organise and make this happen: to build solid foundations, not quicksand, for future computing.

Acknowledgments

We thank all our academic and industry colleagues. This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER (Sewell). This work was funded in part by an AUFF starter grant (Pichon-Pharabod). We thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification. This work was supported in part by EPSRC grant EP/Z000580/1.

References

[1] 2026. ASL1 Architecture Specification Language. https://developer.arm.com/architectures/architecturespecificationlanguage, accessed 2026-07-27.
[2] 2026. The Dafny Programming and Verification Language. https://dafny.org/, accessed 2026-07-27.
[3] 2026. Frama-C: A platform to make your C code safer and more secure. https://frama-c.com/, accessed 2026-07-27.
[4] 2026. Infer: A tool to detect bugs in Java and C/C++/Objective-C code before it ships. https://fbinfer.com/, accessed 2026-07-27.
[5] 2026. Iris. https://iris-project.org/, accessed 2026-07-27.
[6] 2026. KLEE Symbolic Execution Engine. https://klee-se.org/, accessed 2026-07-27.
[7] 2026. Soteria. https://soteria-tools.com/, accessed 2026-07-27.

[8] 2026. Verus. https://github.com/verus-lang/verus, accessed 2026-07-27.

[9] Zain K Aamer[1], Rini Banerjee[1], Hiroyuki Katsura[1], David Kaloper-Meršinjak[1], Dimitrios J. Economou, Kayvan Memarian, Dhruv Makwana, Neel Krishnaswami, Benjamin C. Pierce, Christopher Pulte, and Peter Sewell. 2026. Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows. *Proc. ACM Program. Lang.* 10, PLDI (June 2026). https://doi.org/10.1145/3808278 [1] these authors contributed equally..

[10] AbsInt. 2026. Astrée: Fast and sound static analysis. https://www.absint.com/astree/index.htm, accessed 2026-07-27.

[11] ahadali5000, Alasdair Armstrong, Alexander Richardson, Aril Computer Corp. (for contributions by Scott Johnson), Ben Marshall, Bicheng Yang, Bilal Sakhawat, Brian Campbell, Chris Casinghino, Christopher Pulte, Martin Berger Codasip (for contributions by Tim Hutt, Ben Fletcher), dylux, eroom1966, Google LLC (for contributions by its employees), Hesham Almatary, Jan Henrik Weinstock, Jessica Clarke, Jon French, Martin Berger, Michael Sammler, Microsoft (for contributions by Robert Norton-Wright, Nathaniel Wesley Filardo), Muhammad Bilal Sakhawat, Nathaniel Wesley Filardo, Paul A. Clarke, Peter Rugg, Peter Sewell, Philipp Tomsich, Prashanth Mundkur, Rafael Sene, Rishiyur S. Nikhil (Bluespec, Inc.), Robert Norton-Wright, Shaked Flur, Thibaut Pérami, Thomas Bauereiss, VRULL GmbH (for contributions by its employees), William McSpaddden, and Xinlai Wan. 2014–2024. Sail RISC-V instruction-set architecture (ISA) model.

[12] Nikos Alexandris, Johannes Åman Pohjola, Hrutvik Kanabar, Ramana Kumar, Magnus Myreen, Irvin Ng, Daniel Nezamabadi, Michael Norrish, Yong Kiam Tan, Miki Tanaka, Oskar Abrahamsson, Anthony Fox, Alejandro Gómez-Londoño, and Scott Owens. 2026. CakeML. https://cakeml.org/, accessed 2026-07-27.

[13] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. https://doi.org/10.1145/3458926

[14] Saar Amar, David Chisnall, Tony Chen, Nathaniel Filardo Wesley, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture* (Toronto, Canada). Association for Computing Machinery. https://doi.org/10.1145/3613424.3614266

[15] Andrew W. Appel, Lennart Beringer, William Mansky, and Qinshi Wang. 2026. Verified Software Toolchain. https://vst.cs.princeton.edu/, accessed 2026-07-27.

[16] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019), 71:1–71:31. https://doi.org/10.1145/3290384

[17] Sacha Ayoun, José Fragoso Santos, Philippa Gardner, Nat Karmios, and Petar Maksimović. 2026. Gillian. https://vtss.doc.ic.ac.uk/research/gillian.html, accessed 2026-07-27.

[18] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[19] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.). Springer, 174–203. https://doi.org/10.1007/978-3-030-99336-8_7

[20] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. 2019. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *J. ACM* 66, 1 (2019), 1:1–1:77. https://doi.org/10.1145/3243650

[21] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, Roch Guérin, Ramesh Govindan, and Greg Minshall (Eds.). ACM, 265–276. https://doi.org/10.1145/1080091.1080123

[22] Marc Brooker and Ankush Desai. 2024. Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods. *ACM Queue* 22, 6 (2024), 60. https://doi.org/10.1145/3712057

[23] Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1466–1490. https://doi.org/10.1145/3729309

[24] Shilpi Goel, Anna Slobodová, Rob Sumners, and Sol Swords. 2020. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 47–60. https://doi.org/10.1145/

3372885.3373811
[25] Bart Jacobs, Jan Smans, and Frank Piessens. 2026. VeriFast. https://github.com/verifast/verifast, accessed 2026-07-27.
[26] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL (2026), 1383–1411. https://doi.org/10.1145/3776690
[27] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574
[28] Daniel Kroening. 2026. CBMC. https://www.cprover.org/cbmc/, accessed 2026-07-27.
[29] Xavier Leroy, Sandrine Blazy, Zaynah Dargaye, Jacques-Henri Jourdan, Michael Schmidt, Bernhard Schommer, and Jean-Baptiste Tristan. 2026. CompCert. https://compcert.org/, accessed 2026-07-27.
[30] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 65–79. https://doi.org/10.1145/3453483.3454030
[31] Guido Martínez, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2026. https://fstar-lang.org, accessed 2026-07-27.
[32] Kayvan Memarian. 2023. *The Cerberus C semantics.* Technical Report UCAM-CL-TR-981. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-981 PhD thesis.
[33] Kayvan Memarian, Ben Simner, David Kaloper-Mersinjak, Thibaut Pérami, and Peter Sewell. 2025. Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs (Eds.). ACM, 685–700. https://doi.org/10.1145/3731569.3764817
[34] Peter Müller and Alex Summers. 2026. Viper. https://www.pm.inf.ethz.ch/research/viper.html, accessed 2026-07-27.
[35] Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell. 2026. ArchSem: Reusable Rigorous Semantics of Relaxed Architectures. *Proc. ACM Program. Lang.* 10, POPL (2026), 204–234. https://doi.org/10.1145/3776650
[36] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107
[37] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 161–168. https://doi.org/10.1109/FMCAD.2016.7886675
[38] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz (Eds.). ACM, 136–151. https://doi.org/10.1145/3603269.3604834
[39] Ali Saidi. 2026. AWS Nitro Isolation Engine: Formally verifying the hypervisor in the AWS Nitro System. AWS Compute Blog. https://aws.amazon.com/blogs/compute/aws-nitro-isolation-engine-formally-verifying-the-hypervisor-in-the-aws-nitro-system/, accessed 2026-07-26.
[40] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443
[41] Ben Simner. 2025. *Arm systems semantics.* Ph. D. Dissertation. University of Cambridge. https://doi.org/10.17863/CAM.121319
[42] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.* Document Version 20191213. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Ken Dockser, Roger Espasa, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams,

Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

[43] Robert N. M. Watson, John Baldwin, David Chisnall, Tony Chen, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Brett F. Gutstein, Graeme Jenkinson, Ben Laurie, Alfredo Mazzinghi, Simon W. Moore, Peter G. Neumann, Hamed Okhravi, Alex Richardson, Alex Rebert, Peter Sewell, Laurence Tratt, Murali Vijayaraghavan, Hugo Vincent, and Konrad Witaszczyk. 2025. It Is Time to Standardize Principles and Practices for Software Memory Safety. *Commun. ACM* 68, 2 (2025), 40–45. https://doi.org/10.1145/3708553

[44] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-987

[45] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1559–1584. https://doi.org/10.1145/3656440

[46] Steve Zdancewic, Yannick Zakowski, Calvin Beck, Irene Yoon, Gary (Hanxi) Chen, and Roger Burtonpatel. 2026. Vellvm. https://vellvm.github.io/vellvm/, accessed 2026-07-27.