How humans learn to trust machines that check their work

Trust in programmed systems emerges not from absolute certainty but from layered safeguards, social validation, and architectural minimalism. The most trusted systems—whether SQLite, seL4, or the Coq proof assistant—share a common pattern: they reduce the amount of code that must be trusted to a tiny, auditable core, then subject that core to extraordinary scrutiny over many years. For theorem proving systems specifically, mathematicians have developed a remarkably sophisticated epistemology centered on the **de Bruijn criterion**: the principle that proof checking should be separable from proof generation, allowing skeptics to write their own small verification program. MathOverflow This architectural choice, combined with decades of successful use and the discovery that formalization routinely catches errors in human proofs, has gradually transformed computer-verified mathematics from a curiosity into an increasingly accepted standard.

The psychological reality is that no one—not even experts—can fully verify systems they depend on. Instead, trust becomes a social and institutional phenomenon, built through transparent processes, diverse implementations, and the slow accumulation of track record. Ken Thompson's 1984 observation remains foundational: ultimately, we trust people more than code. Carnegie Mellon University

Testing proves presence but never absence of bugs

Edsger Dijkstra's 1969 observation that "program testing can be used to show the presence of bugs, but never to show their absence" Wikiquote defines a fundamental epistemological boundary. Testing is asymmetric—it can falsify claims of correctness but cannot verify them. A system passing millions of tests provides statistical confidence but never logical certainty.

Modern testing has evolved sophisticated techniques to maximize coverage. **Property-based testing**, pioneered by QuickCheck in 1999, automatically generates millions of test inputs to verify that general properties hold universally. Jsverify When QuviQ's commercial QuickCheck tested Ericsson's Megaco protocol, it found faults "early in development that had not been detected by other testing techniques." **Fuzzing** has proven even more powerful for finding crashes—Google's OSS-Fuzz infrastructure has discovered over **28,000 bugs** across 850+ open-source projects Gocodeo by methodically mutating inputs to trigger unexpected behavior. sqlite

Yet coverage metrics create a false sense of security. A test suite achieving 100% line coverage tells us only that every line executed at least once—not that it executed correctly under all conditions. Danielhall The FDA's guidance acknowledges this directly: "decision coverage alone is insufficient for high-integrity applications." Somco Software Even the gold standard **Modified Condition/Decision Coverage (MC/DC)**, required for aviation software under DO-178C, cannot guarantee correctness. As SQLite's creator D. Richard Hipp notes: "Achieving 100% MC/DC does not prove that you always get the right answer. All it means is that your tests are so extensive that you managed to get every machine-code branch to go in both directions at least once." Hacker News

SQLite exemplifies what obsessive testing can achieve—and its limits. The database engine contains **155,800 lines of code** paired with **92 million lines of tests** (a 590:1 ratio), achieves 100% MC/DC coverage, and runs approximately 248.5 million tests before each release. sqlite It may be the most thoroughly tested software in

existence, deployed on virtually every smartphone, browser, and operating system. Nurkiewicz Yet even SQLite's maintainers acknowledge that testing proves nothing about the absence of bugs—it merely demonstrates extraordinary diligence.

Formal verification closes the gap between testing and truth

Where testing samples behavior, formal verification proves properties mathematically. The seL4 microkernel, verified in 2009, represents the gold standard: approximately 8,700 lines of C code ACM Digital Library proven functionally correct against a high-level specification Australian National University using **200,000 lines of Isabelle/HOL proof**. The verification guarantees that "the implementation always strictly follows our high-level abstract specification of kernel behaviour"—meaning seL4 will never crash, never perform an unsafe operation, ACM Digital Library and enforces strict isolation between processes. ResearchGate

But verification proofs verify conformance to specification, not correctness in any absolute sense. As Hillel Wayne emphasizes: "You do not prove something's correct. You prove conformance to specification." If the specification is wrong or incomplete, the proof guarantees nothing about real-world behavior. The seL4 team explicitly lists their assumptions: they trust approximately 340 lines of assembly code, assume hardware operates correctly, exclude 1,200 lines of boot code, and do not address information side-channels. Sel4

CompCert, the verified C compiler developed by Xavier Leroy at INRIA, demonstrates both verification's power and its limits. Its semantic preservation theorem guarantees that compiled code preserves the behavior of source code— ACM Digital Libraryif the source has defined behavior, the compiled binary will match it exactly. In a landmark 2011 study, researchers used the Csmith random program generator for six CPU-years of testing. The result was striking: "CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors." Cornell Computer Science Every bug subsequently discovered in CompCert has been in unverified components—the preprocessor, lexer, or assembly printer—never in the verified core.

Yet CompCert still trusts the Coq proof assistant, the OCaml compiler and runtime, and the underlying hardware. The chain of trust extends downward indefinitely, creating what philosophers call the "turtles all the way down" problem.

Theorem provers achieve trust through radical minimalism

The mathematical community's acceptance of theorem provers represents perhaps the most sophisticated case study in trusting programmed systems. These tools—Coq, Lean, Isabelle, HOL Light—verify mathematical proofs by checking that each step follows from previous steps according to formal inference rules. Wikipedia Their trustworthiness derives primarily from architectural minimalism.

The **de Bruijn criterion**, named after the creator of Automath, requires that "a verifying program only needs to see whether in the putative proof the small number of logical rules are always observed." github PLS Lab This means proof generation can be arbitrarily complex—employing sophisticated tactics, heuristics, and automation—while verification remains simple. A skeptic can write their own proof checker trusting only their implementation and the logical calculus itself. As Henk Barendregt explains: "Although the proof may have the size of several Megabytes, the verifying program can be small." github PLS Lab

HOL Light epitomizes this approach with a kernel of approximately **400 lines of OCaml**—small enough for a mathematician to read in an afternoon. Springer Coq's kernel is larger (~27,500 lines) but still auditable. Lean's kernel runs approximately 5,875 lines. These tiny trusted cores exploit a fundamental asymmetry: proof checking is decidable and mechanical while proof finding is undecidable and creative. The Curry-Howard correspondence formalizes this—proofs are programs, propositions are types, Wikipedia and type-checking (proof verification) is decidable. En Academic

The **LCF approach**, developed by Robin Milner in the 1970s, offers an alternative architecture. Instead of generating proof objects for external checking, LCF-style provers (including Isabelle) define an abstract type thm whose values can only be created through valid inference rules. github The ML type system enforces that theorems are constructed correctly—you cannot "forge" a theorem without deriving it. As Larry Paulson notes: "Strict typechecking ensured that the only values that could be created were those that could be obtained from axioms by applying a sequence of inference rules—namely theorems." github

Both approaches achieve trustworthiness through architectural constraint rather than exhaustive verification of implementation code.

Mathematicians' acceptance followed painful discoveries

The mathematical community's embrace of theorem provers followed not from theoretical arguments but from repeated encounters with error. The **Four Color Theorem proof** (1976) by Appel and Haken marked the first major computer-assisted proof and sparked immediate controversy. When Haken presented at the Joint Summer Meeting, Donald Albers expected "the audience to erupt with a great ovation. Instead, they responded with polite applause." The discomfort stemmed from two sources: no human could verify all computational steps, and the proof contributed no conceptual understanding.

Trust emerged slowly. Rumors of errors circulated in the early 1980s, and Ulrich Schmidt found a significant flaw in the discharging procedure. Only in 2005, when Georges Gonthier formally verified the theorem in Coq, did most mathematicians accept the result as settled. The formal verification "removed the last doubt as to whether the proof is valid."

Vladimir Voevodsky's journey proved even more influential. The Fields Medalist discovered in 2013 that a paper he published with Michael Kapranov in 1991 contained a fundamental error—**fifteen years** after Carlos Simpson had provided a counterexample. The experience transformed his practice: "Who would ensure that I did not forget something and did not make a mistake, if even the mistakes in much more simple arguments take years to uncover?" Institute for Advanced Study Matematikdunyasi He estimated that for every hour spent on original mathematics, he needed nineteen hours ensuring correctness—"too steep" a price. Voevodsky devoted his remaining years to Univalent Foundations, a new mathematical framework designed for computer verification, and founded the UniMath library for formalized mathematics.

Thomas Hales faced a different problem with his proof of the Kepler conjecture. After announcing the result in 1998, the Annals of Mathematics convened a twelve-referee panel that worked for four years before reporting they were "99% certain" of correctness but could not certify all computer calculations. Wikipedia Frustrated, Hales launched the Flyspeck project to formalize the proof completely. Wolfram MathWorld The effort took eleven

years and twenty-two collaborators—but revealed **hundreds of small errors** in the original proof and at least one significant mistake requiring substantial revision. Notably, the formal proof could not certify the original: "Because the original proof was not used for the formalization, we cannot assert that the original proof has been formally verified to be error free." arXiv Vanity

Formalization routinely catches errors that humans miss

The pattern of formalization discovering errors has repeated across mathematics. When Sébastien Gouëzel attempted to formalize Vladimir Shchur's work on the Morse Lemma for Gromov-hyperbolic spaces—published in the Journal of Functional Analysis, a reputable venue—he "found an actual inequality which was transposed at some point causing the proof to collapse." The fix required "a new and correct (and in places far more complex) argument," published as a joint paper. MathOverflow

The experience generalizes. As formalization experts observe: "The process of formalizing any non-trivial mathematical argument is likely to reveal some minor gaps, such as degenerate cases for which the main argument doesn't quite work as stated." MathOverflow Common errors include missing cases, glossing over "uninteresting" details, and missing assumptions.

The **Feit-Thompson theorem** formalization in Coq (2012) took six years and produced 170,000 lines of code covering 4,000+ sub-theorems. The remarkable finding: "Feit and Thompson's original proof was really correct! Coq says it's correct, and Coq is much pickier about details than any human could ever be." Mathlesstraveled This success story—a major theorem surviving complete formalization—built confidence that formalization could confirm as well as correct.

Meanwhile, bugs in proof assistants themselves remain rare and manageable. While Coq has had soundness bugs—one 2008 report described a vulnerability the developers called an "exploit"—these bugs are difficult to trigger accidentally. Open Access Government As community members note: "To my knowledge, no machine checked proof of a complex mathematical development has ever been retracted" due to a prover bug.

Humans trust through delegation and social proof

The psychology of trusting systems too complex to verify personally reveals a fundamental reliance on social epistemology. Experts employ what researchers call **trust calibration**—balancing self-confidence against trust in decision support systems. Those confident in their abilities defer less to automation; those trusting systems become more dependent on them.

For proof assistants specifically, Robert Pollack's 1997 paper "How to Believe a Machine-Checked Proof" decomposes trust into verifiable sub-problems: ensuring the formal proof is actually a valid derivation, and ensuring the formal statement matches its intended informal meaning. Stack Exchange This separation parallels engineering's distinction between verification (did we build it right?) and validation (did we build the right thing?).

Social proof plays an outsize role. The "many eyes" hypothesis—Eric Raymond's claim that "given enough eyeballs, all bugs are shallow"—captures the intuition that open source enables community verification. TechRepublic But the hypothesis faces empirical challenges. The Shellshock bug lurked in bash for **25 years**

undetected. Erratasec The XZ Utils backdoor (2024) nearly compromised Linux infrastructure despite open development. As one critic observed: "The average programmer writes 10x more code than they read." The pool of qualified reviewers for security-critical code—especially cryptographic implementations—remains shallow. Erratasec

Transparency creates a paradox: being auditable does not mean being audited. Open source makes verification possible without guaranteeing anyone performs it. The Dual_EC_DRBG controversy, where trusting NSA recommendations led to a potentially backdoored cryptographic standard, prompted NIST to acknowledge "the core issue was trusting other organizations and individuals without verifying their work."

The expert consensus in cryptography—**"Don't implement cryptography yourself"**—reflects accumulated wisdom about the impossibility of individual verification. The history of encryption is "littered with broken software, some authored by eminent expert cryptographers." Trust flows instead through institutional channels: NIST's transparent standardization process, years of cryptanalytic scrutiny, and reputation built through survival.

Thompson's trusting trust reveals the infinite regress

Ken Thompson's 1984 Turing Award lecture, "Reflections on Trusting Trust," remains the definitive statement of the trust problem's depth. Wikipedia cmu Thompson demonstrated that a compiler binary could be modified to insert a backdoor into the Unix login program when compiling it—and simultaneously reinsert the backdoor-insertion code when compiling any new compiler from clean source. The attack is self-perpetuating and invisible to source code inspection. The Morning Paper

The implication is devastating: "You can't trust code that you did not totally create yourself. No amount of source-level verification or scrutiny will protect you from using untrusted code." The Morning Paper And Thompson extended the point further: "I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect." The Morning Paper cmu

David Wheeler's diverse double-compiling (DDC) provides a partial defense. ResearchGate The technique compiles a compiler's source code using two independently-written compilers, then compares results. Wikipedia If the outputs match bit-for-bit, the compiler binary corresponds to its source (under certain assumptions). Wheeler demonstrated DDC successfully with TinyCC, Lisp compilers, and GCC. The **Bootstrappable Builds** project extends this approach, reducing the minimal trusted binary required to bootstrap a toolchain to approximately **6 kilobytes**.

Yet the regress cannot be eliminated entirely, only managed. As the Bootstrappable Builds community acknowledges: "Trust is not transitive... if the sister of a friend knows someone who verified this, it is not as much trust as 'I verified this.'" Modern computing stacks embed decades of trust decisions, each link in the chain adding uncertainty.

Hardware vulnerabilities bound what software verification can guarantee

Software verification ultimately depends on hardware correctness, and hardware has repeatedly proven

untrustworthy. The **Intel Pentium FDIV bug** (1994) arose from missing values in a lookup table, causing incorrect floating-point division in approximately one in nine billion random cases. The discovery cost Intel $475 million and triggered industry-wide adoption of formal verification for floating-point units.

Spectre and Meltdown (2018) revealed far deeper problems. These vulnerabilities exploit speculative execution and out-of-order processing—fundamental performance optimizations present in virtually all modern processors since 1995. Meltdown allows user-space programs to read kernel memory; Spectre enables side-channel attacks through branch prediction. Neither vulnerability is visible to software-level analysis because the exploited behaviors were considered implementation details, not part of the instruction set architecture.

Verified hardware exists but remains research-stage. The **VAMP processor** (Verified Architecture Microprocessor) implemented a DLX instruction set with full formal verification at the gate level. ARM's **ISA-Formal** framework has verified multiple commercial ARM processors, catching bugs involving "complex sequences of instructions" that simulation misses. But no commercially deployed processor has been fully formally verified, and physical manufacturing defects or tampering remain outside any formal model.

The practical consequence: even a fully verified software stack running on real hardware cannot achieve the certainty that formal proofs suggest. The verification establishes correctness relative to an idealized model that abstracts away transistor physics, timing variations, and side-channel leakage.

Architectures for trust minimize what must be trusted

The central insight emerging from decades of work on trustworthy systems is architectural: **minimize the trusted computing base ruthlessly**. Every line of code that must be trusted represents a potential vulnerability. The strategy appears across domains:

- **Microkernels** move services out of the kernel into user-space, reducing TCB from millions of lines (Linux) to thousands (seL4) HandWiki
- **LCF-style provers** restrict theorem creation to a small kernel that can be audited
- **Proof-carrying code** separates proof generation (complex, untrusted) from proof checking (simple, trusted)
- **Diverse double-compiling** uses independent implementations to cross-validate trust assumptions

Layered verification decomposes complex systems into abstraction levels, proving correctness at each level independently. The **CertiKOS** kernel (Yale, 2016) extends seL4's approach to concurrent systems USENIX using over 30 certified abstraction layers, each adding one feature with its own proof. The result: 6,500 lines of verified concurrent kernel code with 90,000 lines of proofs. Alastairreid

End-to-end verification chains remain the holy grail—connecting verified source through verified compilation through verified operating system to verified hardware. Individual pieces exist: CompCert provides verified compilation, seL4 provides a verified kernel, and various projects have verified processors. But no production system combines them all, and each junction in the chain introduces assumptions that must be trusted.

Conclusion: Trust is social, not mathematical

The search for trustworthy programmed systems reveals a fundamental truth: trust is ultimately a social phenomenon, not a mathematical one. Even the most carefully verified systems rest on assumptions about hardware, specifications, and the correctness of verification tools. The infinite regress of trust—Thompson's turtles all the way down—cannot be resolved through purely technical means.

What makes systems trustworthy is not perfection but the accumulation of confidence through multiple independent channels: small auditable kernels, diverse implementations, transparent processes, track records of successful operation, communities of experts who stake their reputations on using them, and the slow accretion of time. SQLite is trusted because it has run correctly for decades on billions of devices. Coq is trusted because mathematicians have formalized major theorems in it without discovering soundness bugs. CompCert is trusted because years of random testing found bugs only in unverified components.

The lesson for anyone building or evaluating trustworthy systems is architectural: design for verification from the start, minimize what must be trusted, make assumptions explicit, and invest in the institutional processes—review, auditing, transparency, time—that allow trust to accumulate. As Thompson concluded in 1984: "Perhaps it is more important to trust the people who wrote the software." Carnegie Mellon University cmu The most trustworthy systems are those built by communities whose incentives, reputations, and practices align with getting things right.