The Proof Must Go On

Formal Methods in the Theater of Secure Software Development of the Future

Charles Averill
charles@utdallas.edu
University of Texas at Dallas
Dallas, Texas, USA

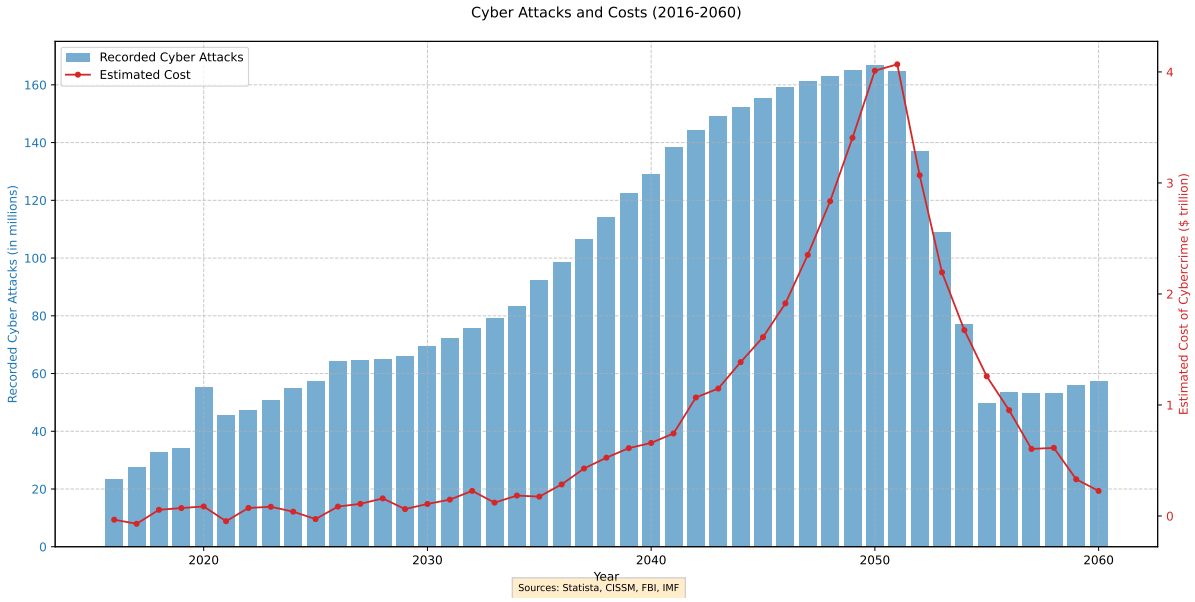

Figure 1. Recorded cyberattacks and their cumulative costs over the first half of the 21st century.

Abstract

Formal methods does not arrive with fanfare. It spreads quietly, not as a revolution, but as a patch: reviewed, merged, and dismissed. In the coming decades, the dream of top-down correctness collapses under the weight of real-world software, replaced by a slower, messier reality as FM becomes infrastructure. Verified C libraries thread their way into the systems that run the internet. Model checkers embed themselves in CI pipelines. Modern type systems reach the masses. But AI and IO remain stubbornly unverifiable and insecure, forcing industry to work towards a deeper form of trust. Meanwhile, a generation of developers learns to write specifications not as a niche academic exercise, but as a matter of professional survival. This paper tells the story of how formal verification goes from lofty fantasy to invisible standard — and how, without anyone noticing, the proof goes on.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

SPLASH '25, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Charles Averill. 2018. The Proof Must Go On: Formal Methods in the Theater of Secure Software Development of the Future. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (SPLASH '25).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

Prologue

Formal methods (FM) remain an overlooked aspect of software development for most practitioners. The typical developer's primary objective is speed — finishing current tasks quickly to address the backlog of features and bugs that has accumulated over months. In this race against time, verification becomes a luxury few can afford.

Consequently, formal methods have become a privilege exclusive to aerospace companies, defense contractors, and elite infrastructure teams within tech giants — entities that can invest in correctness by construction. This creates a security inequality: the most critical systems benefit from rigorous verification, while software used by schools, hospitals, and local governments relies on hope, patches, and best-effort testing. The same mathematical principles that ensure the safety of aviation and banking systems remain inaccessible to most developers.

After a century of advancement, only well-resourced organizations can leverage formal methods to protect their infrastructure and users. This disparity continues to grow, with increasingly severe consequences for those left behind.

Today's formal methods encompass a spectrum of guarantees. At one end lie simple proofs of invariants, types, bounds, and memory safety. At the other, we find machine-checked mathematical arguments that programs conform to formal specifications. Between these poles exists a diverse toolkit: SAT solvers, symbolic execution engines, abstract interpreters, and refinement type systems — all attempting to answer the fundamental question: "Does this program do what we believe it does, and only that?"

Many FM initiatives have pursued a top-down strategy: developing new languages, compilers, and operating systems verified from first principles. This "top-down formal methods" approach aims for theoretical perfection, building upward from pure foundations. However, real-world systems resist such clean-slate approaches. Production software is inevitably complex, legacy-bound, and interconnected. Despite their technical elegance, purely top-down efforts rarely reach deployment.

In our modern digital landscape, software orchestrates everything from embedded devices to global infrastructure. Behind the scenes, correctness, security, and safety form the invisible foundation ensuring these systems function reliably. Formal methods, once primarily academic, increasingly provide these critical properties.

The future of formal methods lies not in revolutionary replacements but in methodical, incremental transformation. This essay traces FM's trajectory from top-down idealism to deeply integrated, production-level infrastructure — a backstage revolution that continues even when unnoticed by most observers.

Act I: The Decline of Top-Down Verification

Top-down formal methods begin to fade from prominence. Though once championed by researchers as the cleanest path to correctness, this approach falters under real-world complexity. Efforts to verify entire systems from scratch — operating systems, browsers, or compilers — prove too expensive, fragile, and disconnected from the chaotic environments they aim to secure.

In its place, modular verification thrives. Verified components — allocators, parsers, encoders, protocol handlers — circulate widely and integrate seamlessly into existing codebases. These components use assembly, C, and Rust — not because verification demands it, but because historical momentum dictates it. These Modular Verified Components (MVCs) become formal methods' first mainstream offering.

By the late 30s, the first formally verified C standard library enters production systems, beginning a deliberate migration through the open-source ecosystem. Linux adopts portions of it, followed by embedded firmware. Billions of devices begin to rely on formally verified routines for memory management, string processing, and concurrency — without their developers needing to understand the underlying proofs.

Throughout this transformation, researchers focus on verified boundaries — the interfaces defining trust relationships. More libraries arrive with formal contracts. While most compilers remain unverified, new tools emerge to bridge the gap: symbolic execution engines, model checkers, and static analyzers built on formal methods principles, integrated directly into continuous integration (CI) pipelines. Binary-level analysis becomes standard practice due to its utility and automation.

Meanwhile, programming languages evolve to support verification. High-level languages adopt features like affine types, refinement types, and purity annotations. However, the vision of fully verified high-level programs remains elusive due to real-world complexities.

Nevertheless, verification tools become increasingly accessible. Verified components proliferate, with public repositories of proven algorithms replacing ad-hoc implementations. Open-source contributions begin to include not just tests but machine-checkable properties.

Formal methods remain expensive — the domain of the patient, well-funded, and risk-averse. But they are no longer unreachable, silently transforming from academic curiosity to infrastructural default by infiltrating trusted modules.

Act II: The Unverifiable Core and its Consequences

As MVCs become widespread, their trustworthiness stands in stark contrast to Artificial Intelligence, which has steadily evolved since the 2020s. Despite its capabilities, AI continues to struggle with hallucination and susceptibility to manipulated inputs. AI functionality remains firmly categorized as

"untrustworthy but useful," with developers implementing hard-coded bounds to prevent erroneous outputs from affecting deterministic systems. Many formal methods researchers view this cautious approach favorably, as it aligns with their practice of proving properties of outputs without reasoning about their generation.

Concurrently, low-level I/O operations interacting with hardware become the frontier for verifying high-level code. These operations, essential to every computing aspect, present unique challenges due to their interaction with the physical world. Researchers begin formally describing these effects where possible, sharing machine-parseable libraries of specifications. Like AI, these fundamental operations and their documented effects become necessary components of the trusted computing base for all software.

The combination of MVCs and this pragmatic approach to unverifiable components raises the security baseline for many critical applications. For the first time, the annual number of successful cyberattacks decreases for several consecutive years.

However, this progress proves insufficient. Continued large-scale breaches of medical and financial systems with exponentially increasing societal costs remind the world that while opportunistic attacks have decreased, sophisticated cybercriminals remain active. The piecemeal replacement of components cannot fully secure software built on fundamentally vulnerable foundations.

Act III: Education, the Backstage Revolution

A quiet revolution unfolds in classrooms worldwide. Formal methods, once confined to graduate-level seminars and specialized industry sectors, permeate educational curricula as cyberattacks force institutions to support such studies. Verification becomes not just an ideal but a necessary skill for the next generation of developers.

By the mid-40s, formal methods have been reintegrated into core computer science curricula, returning to the vision of the field's founders. Universities revamp their teaching models to reflect the new reality: software is not merely a product but a promise. Every line of code carries the weight of trust, and every developer must be equipped to uphold that promise. These educational models elegantly blend theoretical frameworks for program analysis and verification with practical software development. Students learn to use verification tools within their standard development environments.

Within existing developer communities, knowledge of formal methods becomes standard. Online forums, local meetups, and open-source projects become invaluable resources as a culture of collaboration around verified software emerges. Practitioners increasingly view formal methods not as a barrier but as a tool for improving their daily work.

In 2049, Rowan Carter, a Cornell University graduate student, achieves a breakthrough that marks a turning point. Carter demonstrates the theoretical ability to automatically decide the behavior of a broader class of common loops than previously thought possible. These loops, which had been considered impossible to verify without manual intervention, can now be automatically analyzed and verified.

This advancement is monumental. Loops, present in nearly every program, had long been a source of uncertainty in verification. While formal methods had progressed significantly in verifying control flow, data dependencies, and memory access, loops remained an enigmatic and computationally expensive challenge. With Carter's method, developers no longer need to manually reason through termination, bounds, or safety properties of these loops.

Continuous integration systems with formal methods capabilities are quickly updated with this functionality, enabling automatic verification of billions of lines of code for critical security and correctness properties. These new analysis techniques immediately begin preventing cyberattacks on a large scale, reducing yearly incidents to levels not seen in decades. By 2055, a generation of formal methods-educated developers has established itself in development and security positions, now required to defend only against attacks targeting truly undecidable loop behavior.

The integration of formal methods into education doesn't just improve software safety — it transforms a generation of developers. They no longer simply write code to clear backlogs; they become engineers of trust, architects, and maintainers of the trustworthy systems upon which society depends. Carter's 2049 breakthrough symbolizes this transformation: not merely a singular achievement but a milestone in an ongoing revolution reshaping how we conceptualize code and correctness.

Epilogue

The success of formal methods wasn't the result of dramatic breakthroughs but incremental transformation. What began as a niche academic pursuit gradually became integral to mainstream software development through quiet, deliberate changes. Today, formal methods are embedded in everything from simple applications to critical infrastructure, demonstrating the power of steady, piecemeal progress. The future of formal methods, though still evolving, stands on a foundation of 150 years of persistent, transformative work.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009