The Proof Must Go On

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

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

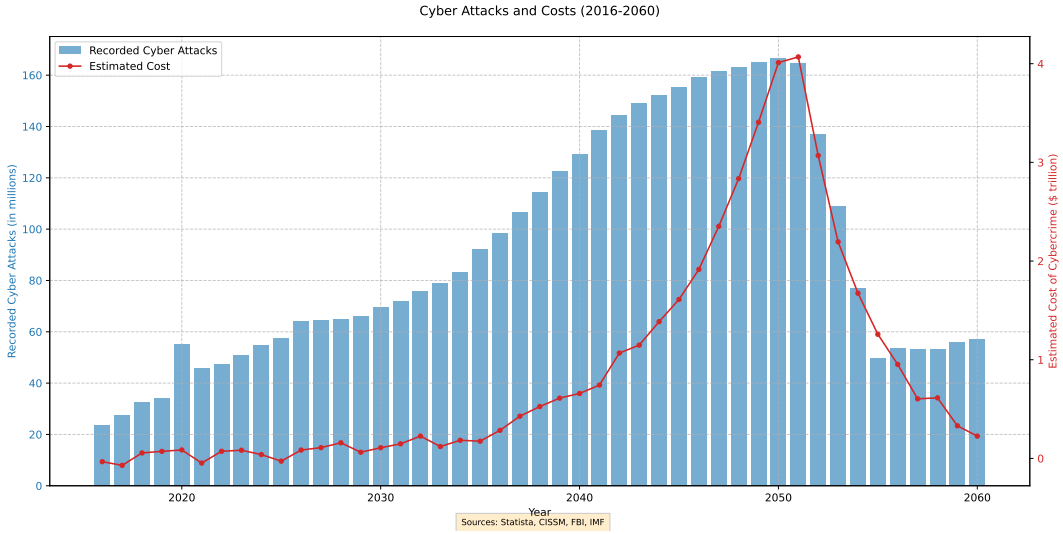

Figure 1. Recorded cyberattacks and their cumulative costs over the first half of the 21st century.[1]

Abstract

Formal Methods (FM) will not arrive with fanfare. It will spread quietly, not as a revolution, but as a patch: reviewed, merged, and dismissed. In the coming century, the decades-old practice of simply testing code will collapse under the weight of cyberattacks and development complexity.

It is succeeded by a slower, more methodical approach 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. 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 a fictional, yet historically-grounded, story of how formal verification goes from lofty fantasy to invisible standard throughout the century, and how, without anyone noticing, the proof goes on.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

ACM Reference Format:
Charles Averill. 2025. The Proof Must Go On: Formal Methods in the Theater of Secure Software Development of the Future. In . ACM, New York, NY, USA, 4 pages.

Prologue: The Wild West

In the early 21st century, FM was an overlooked aspect of software development for most practitioners. The typical developer's primary objective was speed. Finishing current tasks quickly allowed one to address the backlog of features and bugs that had accumulated over months. In this unending race against time, verification became a luxury only a few large companies and governments could afford. This created a security inequality: the most critical systems benefited from rigorous verification, while software used by schools, hospitals, and small businesses relied on hope, patches, and best-effort testing.

These early formal methods encompassed a spectrum of guarantees. At one end lay "lightweight formal methods [1]," collections of dedicated tools for checks of typing, bounds, and memory safety. At the other end are machine-checked mathematical arguments that programs conform to formal specifications. Between these poles existed a diverse toolkit:

[1]Interpolated from historical FBI cybercrime data

SAT solvers, symbolic execution engines, proof checkers, and refinement type systems, all attempting to answer the fundamental question: "Does this program behave how we want it to?"

Many FM initiatives pursued a high-level strategy: developing new languages, compilers, and operating systems verified from first principles. This "top-down" formal methods approach aimed for theoretical perfection, building upward from pure foundations. However, real-world systems tended to resist such clean-slate approaches. Production software was remarkably complex, legacy-bound, and interconnected. These complexities were often trusted and hand-waved away in the name of common sense and development time. Due to their technical elegance, purely top-down efforts commonly reached deployment, but their reliance on trust meant they were built on potentially-faulty guarantees.

The counter to these initiatives, "bottom-up" formal methods, entailed reasoning exclusively about the behavior of machines rather than the languages and systems that abstracted on top of them. Despite their power and generality, the use of these techniques required deeper knowledge and significantly more effort than their counterparts due to the complexity that lies at the base of physical computation. This high barrier to entry meant that bottom-up FM was often overlooked or dismissed as unproductive or unnecessary.

The following history of formal methods details not revolutionary replacement, but methodical, incremental transformation. This essay traces FM's trajectory from academic idealism and intense effort to deeply integrated, production-level infrastructure. A backstage revolution that continued while the audience was none the wiser.

Act I: Meeting in the Middle

Top-down formal methods had long been championed by researchers as the cleanest path to correctness. They offered compelling visions: verified languages, compilers, and kernels, all constructed from first principles. Yet as these systems were deployed, their limitations became clearer.

Several large bodies of critical code were found to be impossible to verify using these methods. The foremost example of this issue was in C runtime libraries: colossal collections of code built to facilitate communication between programs and the machines they ran on. Due to their low-level nature, they contained instructions that interfaced directly with hardware: an interaction impossible to formally quantify with the techniques of the time. As a result, the only sound way these instructions could be modeled was by assuming they wreak havoc on the state of the machine, making any application of existing FM essentially impossible.

As top-down approaches encountered their natural boundaries, bottom-up techniques began to advanced rapidly. New government initiatives [2, 3] into language security sparked interest in FM targeting low-level systems. As researchers targeted low-level systems, they inevitably reached bare metal, prompting collaborations with experts in the already-decades-old field of hardware verification. Although the study of this paradigm had not made the arduous process of reasoning about a machine's behavior at the binary level any easier, the increased interest set a strong course for success. It began to be clear to both researchers and engineers that a convergence of top-down and bottom-up FM would yield astronomical results.

This realization gave rise to a hybrid approach: "full-stack proof frontends," which allowed users to reason about both the high- and low-level behavior of code at unprecedented levels of rigor. Following a wave of cyberattacks exploiting vulnerabilities in how memory allocators stored pointer metadata, DARPA launched the *VERIFEX*[2] project in 2033, encouraging the development of formally-verified software components to replace trusted low-level components such as allocators, parsers, encoders, and protocol handlers. These Minimal Verified Components (MVCs) were first utilized on a large scale by hosting providers and financial institutions, but quickly disseminated to the majority of organizations by replacing existing components of popular libraries and toolchains.

By the late 2030s, the first formally verified C standard library, *Veritas*, entered production, catalyzing a gradual migration through the open-source ecosystem. Developed to use the mass of newly-created MVCs to inplace-modify existing standard libraries, the GNU C Library soon provided options to use the verified alternatives of functions it shared with Veritas, and embedded firmware followed. Soon, billions of devices relied on formally verified routines for memory management, string manipulation, and concurrency without their developers needing to understand the underlying proofs.

Despite the explosion of use of software touched by FM, integration of these techniques into everyday development could not be accomplished due to the general public's lack of awareness of them. The first step towards everyday integration had already been plotted out decades before: building expressive constructs into commonly-utilized programming languages. Large amounts of work now went into providing these popular languages with expressive types and patterns to better communicate developer intention. With compilers allowing for heightened expressiveness, new verification pipelines arose that chained the compiler and the source program with large language models (LLMs) and proof assistants, enabling unprecented, untrusted, automated verification.

Verified components proliferated, with public repositories of proven algorithms replacing ad-hoc implementations. Open-source contributions began to include not just tests but

[2] "VERified, Reusable Infrastructure for Everything eXecutable"

proofs of machine-checkable properties. Developers increasingly reused verified components instead of re-implementing common routines. FM remained resource-intensive, but no longer out of reach. By infiltrating trusted modules, FM had silently transformed from an academic curiosity to the infrastructural default.

Act II: The Unverifiable Core and its Consequences

As MVCs became widespread, their trustworthiness stood in stark contrast to Artificial Intelligence, which had steadily evolved since the 2010s. Despite its powerful capabilities, AI continued to struggle with hallucination and susceptibility to manipulated inputs. AI functionality remained firmly categorized as "untrustworthy but useful," with developers implementing hard-coded bounds and limiting the scope of the behavior of their models to prevent erroneous outputs from affecting sensitive deterministic systems. Many FM researchers viewed this cautious approach favorably.

Concurrently, low-level I/O operations interacting with hardware became the frontier for verifying high-level code. These operations, essential to every computing aspect, presented unique challenges due to their interaction with the physical world. Researchers began formally describing these effects where possible, sharing machine-parseable libraries of hardware specifications. GIANT Labs, (formed by Google, Intel, AMD, NVIDIA, and TSMC), assembled large datasets of these hardware specifications, allowing automatic generation of further formal specifications for new devices through AI-powered proof automation. Alongside AI, the remaining undocumented pieces of these interactions made up the the trusted computing base for huge volumes of software.

The combination of MVCs and this pragmatic approach to hardware-level components raised the security baseline for many critical applications. For the first time, the annual number of successful cyberattacks utilizing hardware side-channels decreased for several consecutive years.

However, the increased scrutiny into unverifiable hardware component effects alerted a new generation of attackers that there exists an attack surface that the world's best researchers could not reason about. Although exploits of hardware side-channels had decreased, there remained a large swath of organizations that fail to properly validate the behavior of unverifiable components, and large-scale breaches of medical and financial systems continued. Increasing societal costs due to cybercrime reminded the world that, while opportunistic attacks had decreased, sophisticated cybercriminals remained active. The piecemeal replacement of components could not fully secure software built on fundamentally vulnerable foundations.

Act III: Education, the Backstage Revolution

As FM further permeated into industry software development practice, a quiet revolution unfolded in classrooms worldwide. The old industrial practices of developing first and testing later had shown to be completely ineffective, leading to governments intervening to force the industry to adapt their incentives to meet modern-day problems. Formal methods, once confined to graduate-level seminars and specialized industry sectors, permeated educational curricula government and industry institutions support such studies in their own future interest. Verification became not just an ideal, but a necessary skill for the next generation of developers.

By the mid-40s, formal methods had been reintegrated into core computer science curricula, returning to the vision of the field's founders. Universities revamped their teaching models to reflect the new reality: software is not merely a product, but a promise to its users. Every line of code must carry the weight of trust, and every developer must be equipped to uphold that promise. These educational models elegantly blended theoretical frameworks for program analysis and verification with practical software development.

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

In 2049, Rowan Carter, a Cornell University graduate student, achieved a breakthrough marking a turning point for the world. Combining lifetimes' worth of existing research into type theory, proof automation, and automata theory, Carter demonstrated 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, could now be automatically analyzed and verified.

This advancement was monumental. Loops, present in virtually all programs, 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 needed to manually reason through termination, bounds, or safety properties of large swaths of loops.

Continuous integration systems with formal methods capabilities were quickly updated with this capability, enabling automatic verification of billions of lines of code for critical security and correctness properties. These new analysis techniques immediately began preventing cyberattacks on a large scale, reducing yearly incidents to levels not seen in decades. By 2060, a generation of formal methods-educated

developers has established itself in development and security positions, now required to defend only against attacks targeting the smaller category of code for which decidability was still impossible with Carter's developments.

The integration of formal methods into education did not just improve software safety, it transformed a generation of developers and the way in which they interacted with software. They no longer simply wrote code to clear backlogs; they became engineers, architects, and maintainers of the trustworthy systems upon which society depends. Carter's 2049 breakthrough symbolized this transformation: not merely a singular achievement but a milestone in an ongoing revolution reshaping how we conceptualized code and correctness.

Epilogue

The success of formal methods was not 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 nearly 200 years of persistent, transformative work.

References

[1] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Van Geffen, J., and Warfield, A. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 836–850.

[2] DARPA. TRACTOR: Translating All C to Rust.

[3] DARPA. V-SPELLS: Verified Security and Performance Enhancement of Large Legacy Software.

Received 24 April 2025; revised 17 July 2025; accepted 12 August 2025