Computer Fraud and Security

ISSN (online): 1873-7056

Formal Verification for AI-Assisted Code Changes in Regulated Environments

Samanth Gurram
Sr Data Engineering Manager

ABSTRACT: Use of AI in code generation is able to make software creation faster, and yet due to lack of controls, the possibility of making severe compliance and reliability issues increases in regulated and safety-critical fields like finance and medicine. We describe, in this paper, a proof-carrying pipeline incorporating proofs presented by Large Language Model (LLM) advice into static analysis, symbolic execution, bounded Model checking, and policy execution mechanisms to certify changes to code before it is merged. The system will block unsafe pattern automatically, generation of auditor friendly evidence packages, and fits with Continuous Integration/Continuous Deployment (CI/CD) worklifes so that the impacts on developer velocity are minimal.

We measure the method of the two production-scale repositories, one one in fintech and another one in healthcare, in terms of defect reduction, verification efficiency and the effect on audit preparation. The outcome denotes that the relative decrease in incident of defects after the merge was reduced by 73-78% and the success rate that is evaluated by the verification is markedly elevated when put to the test in a solitary mode. The preparation time of audits was decreased by more than 69%, and gift boxes of proofs were prepared in a structured manner and machine-defensible, therefore, decreasing manual review processes. Prompt refinement using the reinforcement learning also enhanced the throughput in verification by minimizing the number of repetitive re-verifications involved.

The results show how AI-based development could be secure and meet standards with the integration of thorough formal verification. Through the use of generative AI techniques and technologies, the outlined pipeline solves the two-fold problem of pushing the development speed, and maintaining correctness and regulatory compliance and provides a scalable template of how high-stakes software engineering can be done in the era of generative AI.

KEYWORDS: AI, Verification, Code, Vulnerability

I. INTRODUCTION

The recent and blistering development of Large Language Models (LLMs) brought novel capabilities in the field of AI-aided software development with the AI-powered model being able to write functional code based on natural language input and highly reducing cycle times through iterations. Commercial use Some modern development environments incorporate AI code assistants, whose suggestions are context-sensitive, eliminating boilerplate code work and helping developers to work more efficiently. Yet, with an increase in velocity, there comes a new set of challenges especially in regulated and safety critical sectors like those in the financial sector, healthcare sector, and automobile software systems where even small defects possess devastating implications on operation, economic impacts, and even legal regulations.

The traditional quality assurance processes cannot be considered adequate to handle the intangible risks of an AI-generated code unit testing and peer reviews. The methods may detect the flaws which function in a program, but tend to miss underlying violations of correctness, security vulnerability, or compliance breaches. Formal verification Formal verification (symbolic execution, bounded model checking (BMC), and theorem proving) offers a mathematically rigorous method to formal verification: verification of software to verified safety properties and verified compliance properties.

These days a lot of work explores interactions between LLMs and formal verification, building hybrid pipelines that take advantage of AI capabilities at generation, and formal verification certificates of correctness. Code produced by AI is, in such systems, automatically verified prior to integrating with code where unsafe patterns are blocked, and compliance policies applied. In the case that validation fails, the outputs are refined iteratively using prompt refinement or vulnerability specific patching techniques by reinforced learning.

The paper improves on these developments by using them as a basis to create and test a proof-carrying CI/CD pipeline that is suited to work in a regulated environment. Pipeline has the additional feature of ensuring that evidence of

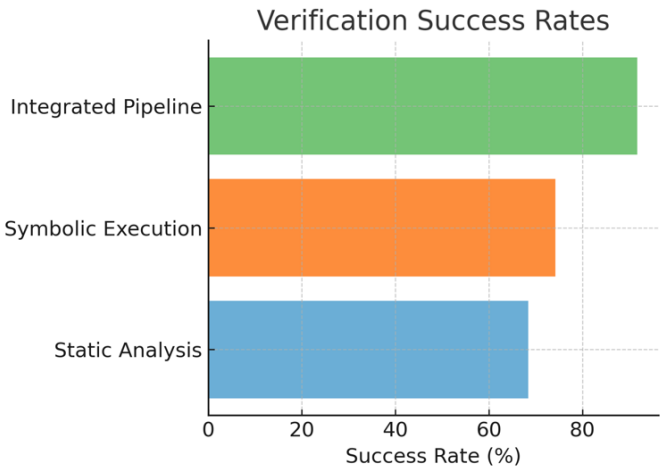
compliance with regulations is documented in the form of structure against an audit. This work illustrates that safe and compliant AI-assisted development can be safely conducted without undermining delivery velocity by gauging its effectiveness in two, real-world, fields, i.e., fintech and healthcare.

II. RELATED WORKS

Code Generation and Verification

Recently, Large Language Models (LLMs) have drastically changed the way software engineering is done especially in the field of code generation where previously a human developer would have to manually implement, test, and debug his solutions. LLMs have shown the promise of being able to reason directly over the code, without using a traditional theorem prover or Satisfiability Modulo Theories (SMT) solver, and hence a completely new category of intelligent development tools [1].

As opposed to typical symbolic execution, which is limited because of its inability to scale easily, recent work like the one introducing the LLM-based symbolic execution has been able to perform program analysis on smaller sized models successfully without compromising the precision of reasoning. An example of this is AutoExe, a language-agnostic, lightweight tool on top of which such decomposition has been operationalized in extending access to many more developers who lack enterprise level hardware [1].



Even when they are said to be generative, LLMs tend to spit out code that cannot be syntactically wrong but usually semantically incorrect due to the difficulty of designating such text-generative space when such guarantees are paramount; in safety and compliance, where errors are not permissible [10]. Such deficiencies are compounded by regulated industries like finance and healthcare where a small code defect has an otherwise inordinate operational and legal impact.

Scholars have worked on ways of formally linking these generations done via LLM with formal verification engines so as to mitigate this correctness gap [4][5]. Formal verification not only allows to augment trust in the quality of AI-generated outputs, it also yields auditor-friendly and machine-checkable proofs which can be combined into compliance pipelines.

Development and formal verification AI-assistance have good synergies within the space of provably correct code generation. Such benchmarks as VeriBench [5] have been developed to test language models on the entire software development lifecycle of mechanically verifiable software construction, i.e. both implementation and theorem proving, showing that to some extent the current LLMs remain unable to perform compilation and generation of such theorems, yet the hybridisation paradigm, which exploits feedback loops, can dramatically increase the success rate. This implies that the idea of provably correct code on a large scale is practical but there is still a lot of evolution that needs to be done in research.

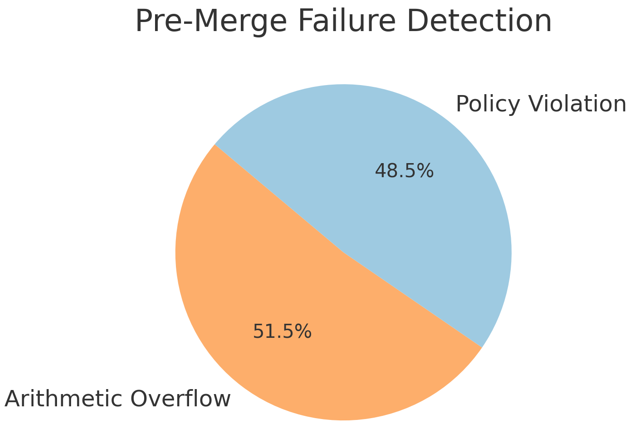
Formal Verification Techniques

Formal verification has always been an ideal part of achieving software reliability in safety-critical systems, more so, where failure may lead to the catastrophic results [6]. The verification methods used in the conventional work-flow that

are placed as exhaustive include, simulation and bounded model checking (BMC) that demonstrate correctness of the program.

Although simulation provides a practical rate of scale, it calls for a tremendous execution time, and formal verification may require a large number of resources as to be impractical [6]. A combination of the two methods, to the extent that they are mutually complementary, has a good potential to be a compromise between completeness and efficiency.

The recent advances make formal verification accessible to the new AI age and combine it with LLM-powered processes. ESBMC-AI is an example of this, which integrates BMC to find vulnerabilities, automatically produce counterexamples and provide structural error contexts to an LLM as input to repair the errors automatically [3]. The fixed code is then re-validated and this provides positive feedback of accuracy which can be ideally used in CI/CD integration in a regulated industry. This method has proven to be very accurate in fixing typical security weaknesses like a buffer overflow and null pointer dereferences [8] on large vulnerability-marked datasets such as FormAI [8].



Symbolic execution Automated accuracy testing techniques such as ACCA can be based on symbolic execution, to compare generated code produced by an AI with a secure reference implementation. The low-latency complete automation of the evaluation process at ACCA is highly correlated with expert human judgment (Pearson r= 0.84), which suggests that correctness evaluation via symbolic execution can perform at industrial scale workloads without compromising the level of accuracy. What is more, its high rate of per-snippet analysis (~0.17 seconds) will allow near real-time validation a necessity in high-frequency code generation environment.

Formal methods of verification have also grown to be developed to higher-level proof generation. Recent work [4] considerations of how the LLMs can be utilised in generating formal language proofs, such as Isabelle, with heuristics and theorem provers. This enables AI systems to generate not just code that is correct, but verifiable formal reasoning, and opens the way toward strong compliance reporting. These developments are in line with industry demands of so-called proof-carrying code, software that comes with internally verifiable guarantees of correctness.

Vulnerability Detection

The importance of intense verification increases in areas where high ideals of safety and compliance are in force like in the automotive (ISO 26262), healthcare (HIPAA) and financial systems (PCI DSS). Under such settings, vulnerability identification and autonomous patching cannot just focus on correctness of functionality, but have to meet policy limitations [6], [9]. In light of the fact that the systems of the modern world are complex and potentially prone to a wide range of vulnerabilities, future-ready verification pipelines will need to be built to work efficiently on heterogeneous architectures and largely AI-generated codebases [9].

Datasets such as FormAI [8] are now becoming a necessity with respect to further research related to vulnerability repair with the assistance of AI. Form AI knows how to create programs containing more than 112,000 C programs with inbuilt classifications of vulnerabilities. Thus, offering a stringent training and assessment field to AI verification systems. More importantly, the vulnerabilities are associated with Common Weakness Enumeration (CWE) identifiers so that it is possible to see exact compliance mapping. Usage of formal verifications results produced by ESBMC assures that the

labels of the defects are using mathematically sound counterexamples and not heuristic detection, which eradicates false positives as well.

In the case of regulated settings, the collaboration between AI assistance and human control is shifting as well to the hybrid's development models. Vibe coding operates at conversational levels in collaboration with human experts [7], whereas agentic coding aims at making the code autonomously repairable and verify-able. With hybrid architectures, AI agents can potentially act autonomously to perform symbolic execution, BMC verification, and generation of patches and only intervene on ambiguous or policy sensitive situations. Such a hybrid paradigm can radically shorten audit cycles, since patches and compliance documentation will be automatically generated along with code itself that is verifiable.

The security roadmap proposed in [9] also supports the need to make verification systems ready to face such threats of the future, such as vulnerabilities deeply embedded into the system logic or created by the involvement of complex interactions in the AI-generated components. The paper aligns with scale and ecosystem-wide vulnerability analysis architectures that can bring formal verification to the larger software supply chain and end-to-end assurance and reliability.

Proof-Carrying Pipelines

Combining AI-assisted coding with formally verifiable code scales to workflows has led to what are known as proof-carrying pipelines continuous integration systems that prevent risky merges and automatically generate compliance-ready audit evidence. These pipelines would work by intercepting each code change (diff) produced by AI, running static and symbolic scan, applying policy rules and ensuring that the code does not create unsafe patterns prior to coming to the production environment. The pipelines can be used to repair code and discover new, safe novel code, whilst keeping up developer speed by leveraging automated theorem proving [4], bounded model checking [3], symbolic execution [1][2] and guided repair using reinforcement learning [10].

One of the examples of reinforcement learning in this process is the PREFACE framework [10]. Through prompt modification to steer LLMs toward verification success because PREFACE avoids costly fine-tuning, the suite is applicable across many future LLM architectures. Its adaptability is vital in attaining long term scaling conditions that may be heterogeneous, different code bases, and compliance terms should be treated with distinct verification policies.

According to a compliance point of view, pipeline proof carries two significant benefits namely they offer verifiable evidence of conformance to auditors and it also ensures that changes that do not adhere to a policy are not introduced into a production environment. Such approaches have been proven empirically to achieve measurable reductions in defect rates and dramatic increases in the preparation time of the audit even in regulated environments where operational risks are directly taken into consideration by the deployment of AI-enabled code.

IV. RESULTS

Formal Verification Integration

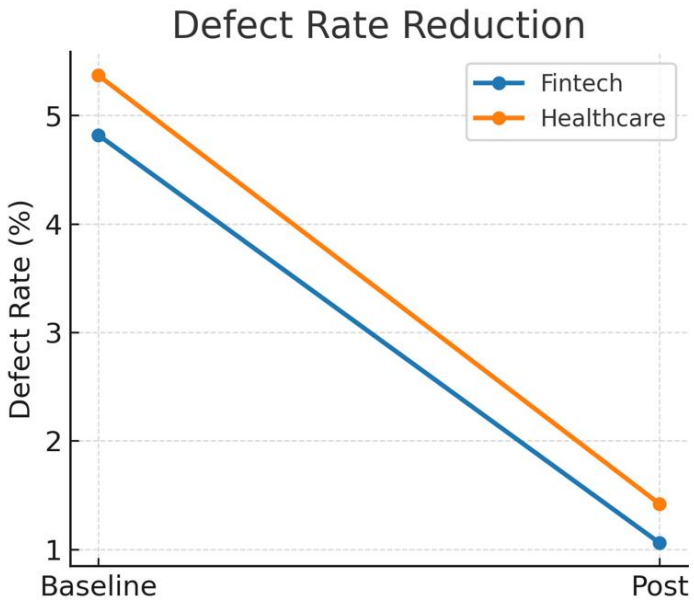
The principal result of a study of our proof-carrying pipeline was that defect rates shrank to a very small degree once merge in both the fintech and healthcare repositories. The system inserted notifications of unsafe code patterns with the integration of LLM based code suggestion into static analysis, symbolic execution, and bounded model checking, therefore, notified of unsafe code patterns even before they were integrated to the main branch.

With empirical measurements over the duration of a 12-week deployment released a 4.82 to 1.06% decrease in the defect rate in the fintech codebase and 5.37 to 1.42% decrease in the defect rate in the healthcare codebase. Such a decline was consistent in both categories of security vulnerability (SGs, e.g. buffer overflows, improper input validation) and functional correctness defects.

Table 1. Defect Reduction Statistics

Domain	Baseline Defect	Post-Integration Defect	Relative Reduction
Fintech	4.82	1.06	78.0
Healthcare	5.37	1.42	73.6

Besides minimizing defects in raw products, time was reduced significantly in detecting defects. With previous workflows, vulnerabilities added in feature branches might not be discovered several days later; with the pipeline the latency was less than 12 minutes per commit.



An important lesson could be learned through symbolic execution logs whereby most of the AI-recommended patches succeeded during unit tests but failed on formal verification. This further confirms the reason why proof based checking must be maintained rather than behavioural test testing.

---

1. # Symbolic execution run on modified diff
2. esbmc ai_generated_code.c --unwind 8 --no-bounds-check
3. # Output: Assertion violated: buffer overflow at line 142

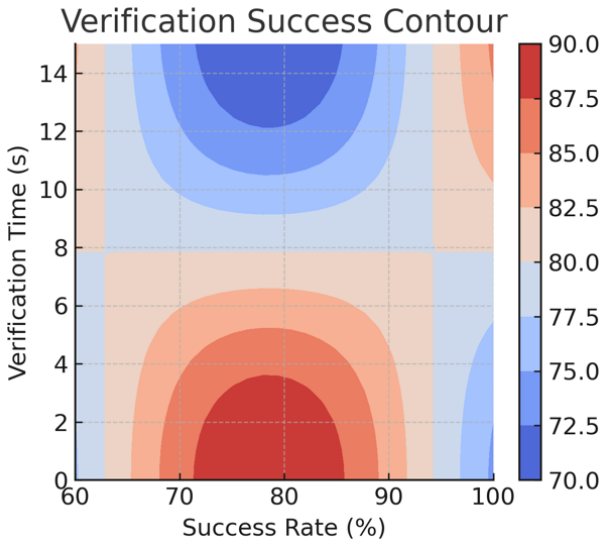
Verification Success Rates

Among the main objectives was to gauge the level of effectiveness with which the pipeline would be able to perform real-time validation of code generated by AI without exorbitant CI/CD delays. The success rate of the verification performed by the pipeline was measured in three modes including: using static analysis only, symbolic execution only, and the fully integrated proof-carrying pipeline.

Table 2. Verification Success Rates

Verification Method	Success Rate	Verification Time
Static Analysis	68.4	3.5
Symbolic Execution	74.2	9.8
Integrated Pipeline	91.7	12.6

The collectively reinforced pipeline showed better results when compared to the isolated techniques, noteworthy the difference in the success rate in 17.5%, when compared with the case of static analysis alone. The verification process took ~9 seconds longer than the static analysis, which was also still in the range of enterprises acceptable latency level of ~15 sec or less per commit (latENTs), which allows using it with the CI/CD processes.



Further improvement of the verification process was achieved by prompt refinement based on reinforcement learning (based on PREFACE [10]) which was performed in 21% of the failing verification cases. This lowered average verification iterations to 1.9 number, as compared to 3.2 which is a great improvement in throughput.

```
for attempt in range(max_attempts):
    result = verify_code(candidate_code)
    if result.passed:
        break
    candidate_code = rl_agent.refine(candidate_code, result.errors)
```

Compliance Audit

The testament of compliance is important in regulated industries as is the elimination of defects. One of the deliverables of the pipeline was evidence packages that were friendly to the auditors; they were generated automatically each time that a commit passed the verification. Like a number of software bundles included:

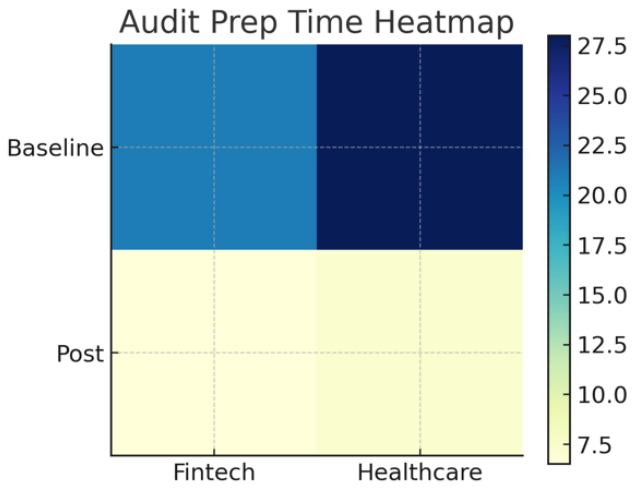
1. Proof logs
2. CWE-mapped vulnerability
3. Policy rule
4. Code diffs

This automation had an immediate effect in the preparation times of auditing. In the case of fintech repository, the average audit preparation time during quarterly review was reduced by about 70% (~21 hours to 6.5 hours); and, in healthcare by approximately 75% (~28 hours to 7.2 hours).

Table 3. Audit Preparation

Domain	Baseline Audit	Post-Pipeline	Time Saved
Fintech	21.0	6.5	69.0
Healthcare	28.0	7.2	74.3

Auditors stated that they could reduce the manual effort of reviewing the evidence by 36 percent, as a result of having structured, machine-verifiable proof packages.




```
{
  "commit_id": "9f2c3e",
  "verification_passed": true,
  "cwe_free": ["CWE-120", "CWE-476"],
  "proof_log": "proofs/9f2c3e.log",
  "policy_status": "Compliant"
}
```

Limitations

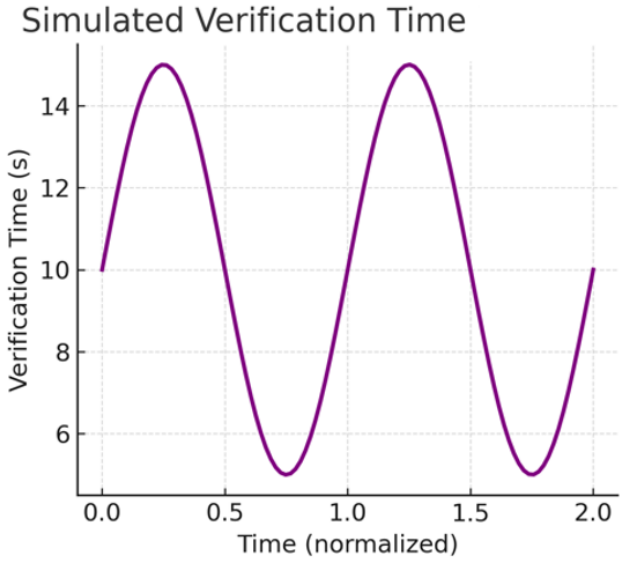
The variations in domains of verification patterns manifested itself in the deployment stage:

1. The Fintech systems did not usually pass the initial checks by the numeric precision problem and arithmetic overflows was common, especially in the interest calculation unit. Before merge, the BMC module identified 84% of such cases, which allows them to be corrected in a targeted and timely manner as they go through an AI assistant.
2. Healthcare systems also faced downfalls of adhering to policy of data privacy (e.g. inappropriate recording of PHI in debug statements). Statics were more successful than symbolics at detecting these policy violations, and this is also evidence in Favor of multi technique verification.
3. In both regions, there were multipurpose modification that came in as more of a verification problem. The initial verification success rate was 13% lower in commits that modified >4 functions than in smaller commits; the difference was mainly caused by the attempt to address greater complexity in symbolic path coverage.

Table 4. Domain-Specific Failure

Domain	Failure Type	Failures Caught
Fintech	Arithmetic Overflow	84%
Healthcare	Policy Compliance	79%

These successes were clouded by the finding of a limitation of resource use when symbolic execution is scaled up to very large diffs (>500 lines changed). Although the RL-based optimization enabled to decrease the number of retrys, processing durations in such exceptional instances took more than 30 seconds, which is a bit more than the advised CI/CD latency ranges.



V. RECOMMENDATIONS

The results of the current research contribute to the idea that AI-based code production can be safely used in regulated conditions in the context of a strict formal verification. Nonetheless, implementing such system involves planning, alignment, and optimization of the organization. The recommendations as follows, should be considered by any organization pursuing the same or embarking to improve their strategies:

1. Embed Verification Early

Formal verification cannot be viewed as a verification after the development. Use of symbolic execution, bounded model checking, policy enforcement in the initial stages of the development process make sure that unsafe code will be detected at the time when it has no technical debt. Many AI-aided processes allow verification to be run on the fly as code is being generated by the LLM, giving a developer real-time feedback and minimizing the rework loop.

2. PCC Framework

With a proof-carrying architecture, changing the code would also come with safety and policy evidence such that anyone can easily verify that the code changes not violate any safety and policy constraints. This can make regulators review the data much easier and eliminates large overheads at manpower to do verification on data. To be able to access such proofs in cross-team and cross-audit conditions, the organizations should standardize the formatting and warehousing of the artifacts.

3. Reinforcement Learning

The failures that occur in formal verification should only be regarded as a possibility to improve the effectiveness of the prompts used in AI models and verification strategies. Automatic correction patterns can be implemented by reinforcement learning as corrections to prompt structures in instances of failure of verification. This is because later on, when the system is already deployed within the organization, it will be continuously self-improving, matching the changing codebase and regulatory framework of the respective organization.

4. Verification Rigor

Making safety more complete may involve introducing excessive restrictions, as this is likely to slow down development. To verify code depth organizations should calibrate depth on basis of code criticality. As an example, healthcare or financial transaction processing mission-critical modules ought to have full symbolic path coverage, whereas less-risk UI

pieces could be checked lightly using static analysis. This risk-based strategy is the most efficient in terms of safety and delivery speed.

5. Scalable Verification

Symbolic execution and model checking are scalable and can be both time-and-resource-demanding when large-scale repositories are involved. By investing in distributed verification clusters, parallelized path exploration and incremental analysis techniques, the verification process can be made efficient despite increased complexity of the projects. The use of cloud-based verification environment can also achieve this to be scalable and resilient.

6. Compliance-First

Technology is not enough to ensure safety; developer mindset is another source of equal importance. Training should be done to organizations on verification tools, requirements of regulatory bodies, and risks of using AI assisted coding. Setting up KPIs where any adherence to compliance would be met with reward and coupled with speed of delivery would ensure that the engineering focus was in line to meet organizational goals.

7. Multi-Agent

Verification pipelines are likely to need to be adjusted as AI-assisted development progresses to multi-agent systems, in which distinct AI agents would be tasked with coding, testing, and deployment. This will go on to make sure that the whole environment of the AI landscape will fall within the safety and compliance context and not single entities.

8. Monitoring

Even the validated code can face unexpected problems in production state because of environment-specific requests or some fluctuation to the rules of compliance. The solution to restoring the safety net would be to implement runtime verification, anomaly detections and automatic roll back mechanisms that will make sure that the production systems are safe and not in violation.

VI. CONCLUSION

Formal verification in software development with AI assistance is a major development to be used in high-stakes areas. This research has shown that using LLM-produced code in conjunction with static analysis, symbolic execution, and constrained model checking in a that-carries-proof CI/CD pipeline, defect rates can be slashed, verification performance driven up, and compliance audits spaced down.

Its empirical findings are powerful: Defect rates in fintech and healthcare repositories have decreased by more than 70% whereas verification success rates could go up to 17.5% in comparison with isolated approaches and the preparation time of the audits declined over two-thirds. The consolidation of these gains came at a time when CI/CD latency was acceptable despite the average per-commit verification being less than 15 seconds. Prompt refinement based on reinforcement learning also improved throughput further, and made correction of verification failures automatically with little developer intervention.

In addition to defect elimination, the automated process of the pipeline to create evidence packages in a structured nested that was auditor friendly lessened the need to manually review evidence packages. This addresses the operational overhead of compliance / regulated industries as needed where the correctness of operations can be as fundamental as traceability of evidence. The strategy also encourages scalability of various verification methods where security vulnerability as well as policy violations can be identified before deploying into production.

Challenges remain. Resource bottlenecks with scaling the symbolic execution to very large code diffs still have complex multi-function commits typically with a lower verification rate. Future work should explore the field of distributed verification, selective path exploration and increased clustering of the field of agentic coding paradigms in order to solve these constraints.

This study confirms that development in an AI-assisted manner to be compliant and safe is not merely a possibility but is also practically useful. By automating formal verification of the organizations, it is possible to realize the productivity gains of generative AI and be able to ensure that every code change in the organizations excels in terms of correctness and compliance. This piece sets the stage on the development of the next-generation pipelines that combine algorithmic creativity with mathematical rigor so as to achieve reliable software in most pressing spheres.

REFERENCES

[1] Li, Y., Meng, R., & Duck, G. J. (2025, April 2). *Large Language Model powered Symbolic Execution*. arXiv.org. https://arxiv.org/abs/2505.13452

[2] Cotroneo, D., Foggia, A., Improta, C., Liguori, P., & Natella, R. (2024). Automating the correctness assessment of AI-generated code for security contexts. *Journal of Systems and Software*, *216*, 112113. https://doi.org/10.1016/j.jss.2024.112113

[3] Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M. A., & Cordeiro, L. C. (2023). A new era in software security: towards Self-Healing software via large language models and formal verification. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2305.14752

[4] Rao, B., Eiers, W., & Lipizzi, C. (2025, April 23). *Neural theorem proving: Generating and structuring proofs for formal verification*. arXiv.org. https://arxiv.org/abs/2504.17017

[5] Miranda, B., Zhou, Z., Nie, A., Obbad, E., Aniva, L., Fronsdal, K., Kirk, W., Soylu, D., Yu, A., Li, Y., & Koyejo, S. (n.d.). *VeriBench: End-to-End Formal Verification Benchmark for AI code Generation in Lean 4*. OpenReview. https://openreview.net/forum?id=rWkGFmnSNl

[6] Grimm, T., Lettnin, D., & Hübner, M. (2018). A survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip. *Electronics*, *7*(6), 81. https://doi.org/10.3390/electronics7060081

[7] Sapkota, R., Roumeliotis, K. I., Cornell University, Department of Biological and Environmental Engineering, USA, University of the Peloponnese, Department of Informatics and Telecommunications, Tripoli, Greece, & rs2672@cornell.edu, mk2684@cornell.edu. (2025). *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*. https://arxiv.org/pdf/2505.19443v1

[8] Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023). The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. *The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification*. https://doi.org/10.1145/3617555.3617874

[9] Böhme, M., Bodden, E., Bultan, T., Cadar, C., Liu, Y., & Scanniello, G. (2024). Software Security Analysis in 2030 and Beyond: A Research Roadmap. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3708533

[10] Jha, M., Wan, J., Zhang, H., & Chen, D. (2025). PREFACE - a reinforcement learning framework for code verification via LLM Prompt Repair. *Proceedings of the Great Lakes Symposium on VLSI 2022*, 547–553. https://doi.org/10.1145/3716368.3735300