SENTINEL: Moving Assertions Earlier for Enhanced Python Program Safety

Dinesh Vasireddy, Soham Gupta, Dhruv Patel, Lavik Jain

1 Introduction

Ensuring the correctness and safety of software systems often requires enforcing invariants through runtime assertions. These assertions typically appear at the end of computations, detecting violations only after resources have been consumed or side effects have occurred. This pattern is particularly common in dynamic languages like Python, where permissive semantics lead developers to write post-hoc validation logic, potentially allowing errors to propagate before detection. This raises a critical question: could violations be detected earlier in execution, before unnecessary computation occurs?

Existing approaches to this problem face significant limitations: static analysis tools struggle with Python's dynamic nature, manual refactoring is error-prone, and automated precondition inference often produces approximations that are either too strict or too lenient. The fundamental challenge lies in automatically transforming programs to detect violations earlier while preserving the exact logical guarantees of the original assertions.

We present SENTINEL, a novel approach that combines large language models (LLMs) for assertion generation with a multi-stage verification pipeline that guarantees logical equivalence between early and final checks. Our system enables more proactive error handling while maintaining program semantics. Experimental results demonstrate that early assertions can significantly reduce computational overhead (saving an average of 28.60 bytecode instructions and 2.68 internal or recursive function calls per function) while preserving the original program's logical guarantees. We further detail our motivations, results, and analysis in the subsequent sections.

2 Project Overview

This paper investigates the technical feasibility and formal verification of *early assertions* which were placed earlier in program execution and are logically equivalent to existing later checks (Figure 1). Our research explores how to systematically transform programs to shift violation detection earlier while ensuring transformed assertions are neither overly strict nor overly weak.

We focus specifically on Python programs, where three factors make this problem particularly challenging: (1) dynamic typing that complicates static analysis, (2) implicit type conversions that can mask potential errors, and (3) complex control flow that creates multiple execution paths. Our approach leverages recent advances in LLMs to generate candidate assertions and combines symbolic execution, fuzzing, and formal verification to validate their correctness.

The central research question we address is: ***How can we generate and formally verify early assertions in Python programs that are logically equivalent to existing later checks, ensuring critical security and correctness properties are maintained and potential violations are detected proactively?*** Through our systematic evaluation across functions of varying complexity, we demonstrate both the potential and limitations of automated early assertion transformation.

Logical Equivalence of Assertions

For any input values $\vec{v}$ and program state σ, early assertions ϕ_{early} and final assertions ϕ_{final} must be logically equivalent:

$$\forall \vec{v}, \sigma : \phi_{early}(\vec{v}, \sigma) \Leftrightarrow \phi_{final}(\vec{v}, f(\vec{v}, \sigma)) \tag{1}$$

where f represents the computation between early and final assertion locations.

Figure 1: Formal definition of assertion equivalence property verified in our pipeline.

3 Related Works

3.1 Enforceable Security Policies and Early Assertion Placement

Foundational work on security policy enforcement established that certain policies can be enforced at runtime by monitoring program executions and halting before a violation occurs [5]. **Schneider** formalized this notion, showing that execution monitors can insert runtime checks to stop an execution "when the security policy they enforce is about to be violated" [5]. It is generally desirable to prevent execution as soon as a violation is detectable (although there are some situations in which desired computation occurs before the violation and after detection). Such inline reference monitors and runtime-checking frameworks (e.g., SASI, PoET/PSLang) demonstrated practical techniques for injecting security checks into code to enforce memory safety and access control policies at runtime. This fail-fast approach aligns with the principles of Meyer's Design by Contract, where software components check preconditions at the interface and immediately flag contract violations rather than propagate errors [4]. **Meyer** introduced in the Eiffel programming language advocated that critical conditions be checked before or during execution of an operation, so that a violation is caught as early as possible, improving reliability [4].

These ideas underline our project's goal: by placing correct and equivalent assertions early in Python programs (at the point where a property can be evaluated), one can proactively detect potential correctness or security violations sooner, provided these early assertions enforce the same policy as existing late-stage checks.

3.2 Assertion Synthesis

3.2.1 LLMs for Assertion and Invariant Generation

LLMs have become a common tool for assertion generation, but their approaches and guarantees vary widely. **Watson et al.** introduced ATLAS [6], a neural machine translation model trained on unit test code, achieving 31% exact matches with developer-written assertions. Recognizing its limitations with longer or uncommon assertions, **Yu et al.** [7] proposed a retrieval-augmented method which searches for semantically similar tests and adapts their assertions, yielding better accuracy and more interpretable outputs. While ATLAS attempts to learn general mappings, Yu's method grounds generation in past examples, making it more robust to out-of-distribution code.

Building on this idea, **Zhang et al.** [8] developed RetriGen, combining a fine-tuned CodeT5 model with retrieval-based augmentation. By feeding the model both context and candidate assertions, RetriGen surpassed 57% exact-match accuracy (which is significantly higher than prior models) and showed particular strength in longer, multi-condition asserts. Unlike Yu et al., Zhang incorporates retrieved examples into the model's input, letting it generalize more flexibly while still leveraging concrete cases.

Other directions include **Torkamani et al.** [12], who use chain-of-thought prompting to synthesize inline production assertions directly from code, and further work from Zhang et al. [13], who frame assertion generation as a code-repair problem. Both rely on retrieval-enhanced prompting to boost accuracy and generalization, but mainly target postconditions.

In contrast, **Pulavarthi et al.** [14] test LLMs on synthesizing formal assertions for hardware verification. They find high variability: while models generate syntactically valid assertions, correctness often suffers, especially without sufficient in-context guidance. This highlights the brittleness of unconstrained generation. **Pei et al.** [15] take a broader view, using LLMs to infer loop invariants from Java source code. Their model achieves high precision when trained on Daikon-mined datasets but struggles with pointer-heavy or structural invariants, showing the limits of purely pattern-driven learning.

While these works highlight the promise of LLMs in capturing developer intent, most focus on generating final-state or unit test assertions and evaluate success syntactically or via coverage metrics. Our work instead targets early assertion placement and enforces formal $\phi_{early} \Leftrightarrow \phi_{final}$ equivalence. We don't just synthesize plausible assertions but prove they preserve program behavior, bridging the gap between generation and correctness.

3.2.2 Weakest Pre-Condition Predicate Transformation

A classical alternative to LLM-based assertion placement is *weakest precondition (WP) predicate transformation*, which computes the conditions under which a later assertion would be guaranteed to hold if moved earlier in the program.

Recent work adapts WP reasoning to dynamically typed languages like Python. **Rak-amnouykit et al.** [27] perform interprocedural WP analysis to infer preconditions that prevent runtime failures (e.g., `raise` statements). Applied to libraries like `scikit-learn`, their tool generates constraints such as `penalty="elasticnet"` $\Rightarrow 0 \leq l1\_\texttt{ratio} \leq 1$, capturing or strengthening undocumented requirements. Their follow-up work [28] introduces partial evaluation to simplify constraint expressions and generate machine-checkable schemas.

Other efforts apply WP-style reasoning at the specification level. **Cosler et al.** [29] combine backward constraint propagation with LLMs to extract temporal logic specs from natural language API documentation. **Hassan et al.** [30] use MaxSMT-guided WP reasoning to infer interprocedural type contracts in Python via abstract interpretation.

While these techniques offer a principled way to infer sufficient preconditions, they can struggle with Python's dynamic features — like mutable state and aliasing [31] — and often require costly symbolic reasoning. In contrast, our method leverages LLMs to propose early assertions and then verifies their equivalence to final checks. This avoids precondition synthesis entirely and instead ensures correctness through lightweight forward reasoning and symbolic validation.

3.2.3 Neuro-Symbolic Learning for Assertion Synthesis

Outside of LLMs, neuro-symbolic methods have been developed to generate assertions via learning-guided search. **Si et al.** [22] introduced *Code2Inv*, which uses reinforcement learning to generate invariants with feedback from a verifier as reward. Its successor, LIPus, which was introduced in **Yu et al.** [23], prunes the search space and adds richer reward signals to learn more expressive invariants. **Yao et al.** [24] developed G-CLN, a logic-regression model that fits nonlinear invariant functions to execution traces. It outperformed template-based inference but was limited by the expressiveness of its features.

3.3 Precondition Inference and Logical Equivalence Guarantees

Ensuring that early assertions are logically equivalent to final checks is a central goal of our work. **Schneider** [5] defines enforceable safety properties as those that must hold for all prefixes of execution. This principle motivates early assertion placement when violations are inevitable.

Dinella et al. [18] attempt to synthesize human-readable preconditions by simplifying downstream conditions and lifting them earlier in code. Their tool discovers "natural" preconditions that ensure safe execution but stops short of proving equivalence to later checks. **Menguy et al.** [19] take a black-box approach, using constraint acquisition to learn necessary input conditions that avoid failures. Their system can infer rich safety constraints even in code with third-party calls, but the results may be overly conservative and lack formal guarantees.

Koenig and Shao [25] offer a complementary perspective through their work on *CompCertO*, a modular extension of the CompCert verified C compiler. They formally prove that source-level specifications (including pre- and post-conditions at component boundaries) are preserved across compilation, ensuring semantic correctness even in open-component settings. While their focus lies in compiler verification, the emphasis on preserving input-output behavior and component-level equivalence aligns with our goals of proving $\phi_{early} \Leftrightarrow \phi_{final}$ after transformation. Their work reinforces the idea that transformations across compiler phases or within program execution can maintain behavioral fidelity when guided by formal specification preservation.

Unlike these works, which either approximate preconditions heuristically or verify equivalence at the compiler level, our approach targets Python programs directly, focusing on synthesizing and verifying early run-time checks. We do not infer approximations but instead construct and prove assertion equivalence through a combined pipeline of LLMs, symbolic execution, and static verification.

3.4 Hybrid Verification: Symbolic Execution, Fuzzing, and Static Verification

We adopt a hybrid verification strategy that integrates symbolic execution, fuzzing with random inputs, and static analysis to validate assertion correctness. Symbolic execution tools like **CrossHair** and **PyExZ3** explore feasible execution paths using SMT solvers to check satisfiability [9, 1, 2]. Utilizing CrossHair as a verification method, our system verifies that synthesized early assertions match final ones across all reachable paths. To complement path-based analysis, we use **Hypothesis**, a property-based fuzzing library that generates edge-case inputs.

These methods are inherently incomplete: symbolic tools suffer from path explosion and unsupported features, while fuzzing lacks formal guarantees. We address these limitations with **Nagini** [3], a sound static verifier for Python 3 that proves user-specified assertions via modular reasoning in the Viper intermediate language. Originally used to verify safety in production network code, Nagini supports Mypy-style annotations and handles concurrency properties like memory safety and race freedom.

Our multi-layered approach reflects neuro-symbolic frameworks such as *Driller*, blending broad bug exposure with formal rigor. **Liu et al.** [16] advance this further with LLM-SE, combining LLM-inferred invariants and symbolic execution analysis to outperform either method alone. Similarly, **LLM-Sym** [17] extends symbolic execution to previously unsupported Python features, like dynamic list operations, by translating them into SMT-compatible constraints. While powerful, it still requires external validation — reinforcing the need for both dynamic and static verification layers.

3.5 Ranking, Filtering, and Post-Processing of LLM Assertions

Because LLM-generated assertions vary in correctness, post-processing methods help reduce verification burden. **Chakraborty et al.** [20] propose *iRank*, a contrastive learning model trained to distinguish between provable and unprovable invariants using verification feedback. It prioritizes likely-correct invariants and improves verification throughput.

Hellendoorn et al. [21] applied similar filtering to Daikon's dynamic invariants, using ML models to predict which ones are likely to hold across executions. These rankers reduce noise and guide developers toward useful assertions. **So and Oh** [26] extend these ideas to automated program repair, introducing a verifier-guided system called *SmartFix* that accelerates the generate-and-verify loop using statistical models. Their framework filters candidate code patches by passing each through a safety verifier, retaining only semantically correct fixes. While their domain focuses on smart contract repair, the core methodology, leveraging verification feedback to select high-confidence candidates, parallels the challenge of filtering LLM-generated code. Unlike purely syntactic or heuristic methods, SmartFix incorporates semantic validation to ensure output correctness, aligning with our goals of verification-informed selection of early assertions.

While we do not use a learning-based ranker, our pipeline acts as a verification filter: only assertions that pass equivalence checks are accepted. Nonetheless, iRank- or VERSE-style models could complement our approach by prioritizing candidate assertions before verification, reducing tool burden and guiding LLM output quality, especially when integrated with retrieval-augmented generation.

4 Methodology

4.1 Overview

Our goal is to proactively enforce security and correctness properties (and prevent unnecessary computation) in Python programs by shifting runtime assertion checks earlier in execution, while ensuring that the new early assertions are *logically equivalent* to the original final assertions. To achieve this, we design a three-phase pipeline combining LLM-based synthesis, program transformation, and formal verification methods. This methodology allows us to generate proposal transformations and validate their accuracy.

1. *Assertion Generation:* Use a large language model (GPT-o3-mini) to generate candidate early, equivalent assertions (ϕ_{early}) in specified locations (that are earlier in the program) based on program context and the existing final assertion (ϕ_{final}). We also aid the LLM with context using condensed forms of wikis on Python assertion generation and best practices (including [10] and [11]) for ensuring logical consistency when making program modifications.

2. *Program Transformation:* Create a "transform program" that introduces boolean variables b_{early} and b_{final} representing ϕ_{early} and ϕ_{final}, respectively, and asserts $b_{early} == b_{final}$ to encode logical equivalence of the early and final checks. For programs with multiple assert statements, this boolean representation and combined assert would involve more variables and equivalence checks. See Figure 2.

3. *Testing & Verification Pipeline:* We then apply testing & formal methods on the transform program to verify that the LLM-added early assertion(s) enforce the same properties as the existing later assertions.

   - **Symbolic Execution:** Employ CrossHair to explore execution paths of the transform program, searching for counterexamples where b_{early} and b_{final} differ.
   - **Fuzz Testing:** Systematically generate random and boundary-case inputs designed to violate assertions, confirming that no input triggers a mismatch between early and final assertions.
   - **Static Verification:** Apply Nagini to formally prove that for all inputs and execution paths, $\phi_{early} \Leftrightarrow \phi_{final}$, discharging verification conditions automatically. **Nagini is experimental and lacks support for various program features, so we only apply this step for programs with supported features.*

Input Python Program

```
def process_data(x: int):
    [insert earlier assert here]
    y = x * 2
    if y > 0:
        z = y
    else:
        z = -y
    assert z == 100
```

LLM-Altered Program

```
def process_data(x: int):
    assert x == 50
    y = x * 2
    if y > 0:
        z = y
    else:
        z = -y
    assert z == 100
```

Transformation Program

```
def process_data_transformed(x: int):
    b_early = (x == 50)
    y = x * 2
    if y > 0:
        z = y
    else:
        z = -y
    b_final = (z == 100)

    # Assert that early & final are
        equivalent
    assert b_early == b_final
```

Figure 2: Example Input Program, LLM Assertion, and Transformation Program

4.2 Evaluation & Measuring Success

4.2.1 Primary Pipeline

We're **evaluating our ability to successfully *transform* and *verify* Python programs by adding early assertions** that are equivalent to existing final assertions. We see success as **achieving upwards of 80% success in creating verified and equivalent programs** with earlier assertions **across programs of varying difficulty.** However, even for failed transformations, we also consider **finding failure paths with high explainability** as a success of our approach and thus we evaluate that as well. For each program analyzed, we classify the result into one of three categories shown in Figure 3 below.

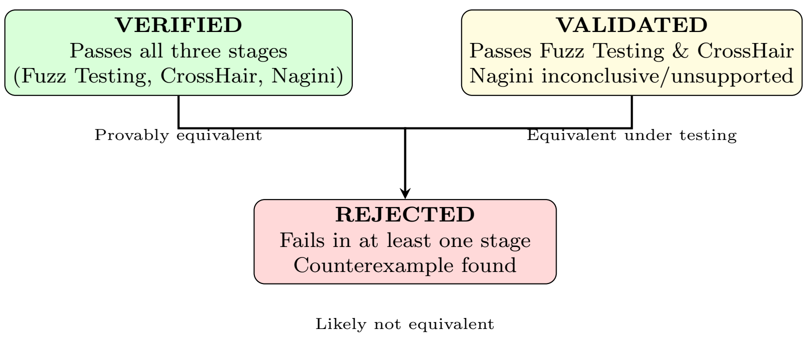

Figure 3: Classification of programs after transformation and verification.

This evaluation ensures that only **early statements that are provably equivalent or validated through extensive testing are accepted**. To support this, we define a structured process for evaluating the pipeline across diverse Python programs, sourced from open-source code, algorithm textbooks, and hand-crafted examples. Each program is annotated with metadata including line count, number of functions, control flow complexity (e.g., conditionals, loops, recursion), and data structure usage.

Using this information, each program is systematically assigned **a difficulty rating (1-5)** after being evaluated on nine weighted components detailed in Table 1. The scoring system applies specific scaling factors for each component to enable consistent evaluation of function difficulty and help identify correlations between program complexity and failure explanation quality (see Table 1 below). The weights were established through over 50 iterations of tuning using data from 120 benchmark programs, where each metric's weight was adjusted to maximize correlation with difficulty ratings from 5 human annotators and performance slowdowns with static analyzers; metrics like control flow depth (0.70) and loop complexity (0.60) received higher weights due to their strong, consistent impact on reasoning complexity observed across both human assessments and tool behavior in early testing.

Parameter	**Weight**
Parameter Complexity	0.30
Operation Density	0.40
Assertion Count	0.20
Control Flow Depth	0.70
Data Type Diversity	0.25
Assertion Complexity	0.80
Mathematical Sophistication	0.50
Loop Complexity	0.60
Function Call Density	0.35
Branching Complexity	0.50
Bytecode Complexity	0.45

Table 1: Complexity Metrics and Weights (matching implementation)

4.2.2 Explaining Failures

For **rejected and not verified transformations**, we introduce a **Failure Explanation Quality (FEQ) score (0-1)** that measures counterexample specificity, diagnostic clarity, and actionability of the failure information. We evaluate the stack traces, failure logs, discovered counterexamples, and conditions from failed executions on Hypothesis, CrossHair, and Nagini. This provides an additional metric to evaluate our verification pipeline's effectiveness even when assertions are incorrect or transformations fail, with a target average FEQ score of 0.5 or higher across all failed cases being preferred. Our scoring system assesses failure messages using four weighted criteria detailed in Equation 2.

$$\text{FEQ} = 0.3 \cdot S_{\text{spec}} + 0.3 \cdot S_{\text{action}} + 0.2 \cdot S_{\text{context}} + 0.2 \cdot S_{\text{tech}}, \quad \begin{cases} S_{\text{spec}} = \text{specificity} \\ S_{\text{action}} = \text{actionability} \\ S_{\text{context}} = \text{context} \\ S_{\text{tech}} = \text{technical detail} \end{cases} \quad \text{where all } S_i \in [0,1] \tag{2}$$

Ultimately, we aim to evaluate the **feasibility of automatically generating and verifying equivalent early assertions in Python programs** across a diverse set of real-world examples. While conducting FEQ evaluations, we also identify ***false positives*** where **assertions are deemed not equal due to a non-logical error** with the method, an unsupported feature, type errors, etc. In this case, the assertion equivalence for that method would be deemed inconclusive.

4.3 Practical Implementation

We evaluated the pipeline on 50 Python functions, each implementing basic arithmetic, complex control flow, or structural operations such as multiplication, discount calculation, factorial calculation, sine function, and data transformation. These functions span operations from basic mathematical computations to complex string processing and data transformations. We also developed automated scripts to synthesize early assertions and transform programs, computed difficulty metrics, ran testing and verification, and generated failure-explanation scores for each program. In particular, we **identified Nagini-unsupported programs using GPT-4.1** (with Nagini's wiki as context) to assess compatibility with its specification language, followed by manual verification [3]. We also used a similar LLM-as-a-judge approach to score failures across all methods to calculate Failure Explanation Quality (FEQ) scores and false positives.

4.3.1 Adaptations for Testing & Verification Methods

Hypothesis Testing. Using the Hypothesis property-based testing framework, we executed each transformed program over 20 randomized and edge-case inputs. The transformed programs encode early (ϕ_{early}) and final (ϕ_{final}) assertions as boolean expressions and assert their equivalence. Our implementation uses Hypothesis's *@given* decorator with appropriate strategy generators tailored to each function's domain (integers, floats, ranges). We configured testing parameters with *@settings(max_examples* $= 20$*)* to limit test cases while maintaining sufficient coverage. Each test function wraps the transformed program in a try-except block to cleanly capture and report assertion failures. For numerical functions involving floating-point calculations, we employed $st.floats()$ with carefully selected boundaries. The framework effectively demonstrates assertion equivalence in practice by encountering fewer failures than symbolic execution, highlighting the complementary nature of dynamic testing versus static verification approaches.

Symbolic Execution. We ran CrossHair symbolic execution on each of the transformed programs. The transform programs are written to disk files with special docstring contracts using pre: and post: annotations that CrossHair understands. Our implementation captures both the early and final assertion conditions as boolean variables (b_{early} and b_{final}) and explicitly checks for their equivalence. The code includes carefully defined preconditions that establish reasonable bounds for input variables (e.g., $-1000 \leq x \leq 1000$) to constrain the search space while ensuring comprehensive verification. Postconditions formally express the property that early and final assertions should be logically equivalent. The implementation invokes CrossHair through a subprocess call, captures its output, and reports any counterexamples found.

Nagini Static Verification. For transformed programs with supported features, we use **Nagini** to formally verify the logical equivalence $\phi_{\text{early}} \Leftrightarrow \phi_{\text{final}}$. Built on the Viper intermediate verification language, Nagini supports PEP 484-style type annotations and permission-based heap access control, but imposes strict constraints: programs must be statically typed and avoid dynamic features like `eval` or runtime field creation. Adapting programs for Nagini required **adding type annotations and contracts, minimally reformatting control flow, and eliminating unsupported language features**. Our verification process follows a two-phase strategy: first testing the original program with dynamic and symbolic methods, then adapting it for Nagini while preserving logical structure. This approach enables us to distinguish between true semantic failures (indicating assertion inequivalence) and compatibility issues (reflecting tooling limitations rather than transformation failures).

```python
def process_data_transformed(x: int):
    '''
    pre: -1000 <= x <= 1000
    post: (x == 50) == ((x * 2 if x * 2 > 0 else -x * 2) == 100)
    '''
    b_early = (x == 50)
    y = x * 2
    z = y if y > 0 else -y
    b_final = (z == 100)
    assert b_early == b_final, "Early and final assertions are not equivalent"
    return z
```

(a) Transform Program with Cross Hair Annotations

```python
from nagini_contracts.contracts import *

@Pure
def is_even(x: int) -> bool:
    return x % 2 == 0

@Pure
def double(x: int) -> int:
    return x * 2

@Pure
def is_multiple_of_4(y: int) -> bool:
    return y % 4 == 0

def check_even_equivalence(x: int) -> int:
    Requires(-1000 <= x and x <= 1000)
    Ensures(is_even(x) == is_multiple_of_4(Result()))
    Ensures(Result() == double(x))

    b_early: bool = is_even(x)
    y: int = double(x)
    b_final: bool = is_multiple_of_4(y)

    Assert(b_early == b_final)

    return y
```

(b) Transform Program in Nagini Specification Language

Figure 4: Annotation Examples in CrossHair and Nagini

5 Results

5.1 Assurance Coverage

Of the 50 target functions, 5 (10%) are *Verified*; 30 (60%) are *Validated*; and 11 (22%) are *Rejected* (Detailed results in Table 2). Notably, the Verified group is dominated by pure-arithmetic functions with minimal control flow, likely contributing to their success across all three methods, whereas functions that rely on randomness, floating-point operations, or complex stdlib calls appear exclusively in the Rejected category (See Table 4). *All relevant code, figures, and tables with results are in this paper's Appendix.*

5.2 Fail-Fast Security Impact

Moving assertions earlier in the program yields tangible security and efficiency benefits. While bytecode instructions don't perfectly capture execution time or system-level resource usage, they provide a consistent, implementation-independent metric for quantifying computational work in Python programs. Across our benchmark, early assertion placement saved an average of 28.6 Python bytecode instructions per function (median 20), with over 80% of failed inputs halted within the first 10 instructions (Figure 6). This reduction in executed instructions before error detection directly shrinks the window for side-effects, resource exhaustion, or side-channel leakage. The greatest savings were observed in pure arithmetic and simple loop functions, which typically halt in under 15 instructions, while string manipulation and floating-point operations required more computation before reaching the assertion. We also see in Figure 7 that there's an average of 3.38 computational elements (loops, function calls, and branches) bypassed per function when assertions are moved earlier, with function calls (mean: 2.68) representing the most significant portion of computational work avoided.. These results highlight the practical value of fail-fast enforcement, especially in security-sensitive or resource-constrained contexts.

5.3 Complexity vs. Verifiability

Verification success rates are strongly tied to program complexity. As shown in Figure 8, functions with overall difficulty scores below 2.5 passed CrossHair over 80% of the time and Nagini over 40%, while those with scores above 4.0 rarely succeeded under either tool. The average difficulty of Verified functions was 2.0, compared to 2.7 for Rejected ones (Table 5).

To further quantify these trends, we analyzed the correlation between difficulty subscores and verification outcomes (Table 6). For CrossHair, control flow depth (0.22) and operation density (0.08) showed weak positive correlations, indicating symbolic execution is only mildly affected by these factors. Nagini exhibited strong positive correlations with operation density (0.82) and assertion complexity (0.45), likely because programs with more explicit operations and assertions align better with static verification. Fuzzing was largely insensitive to these metrics, with the largest effect being a negative correlation with control flow depth (–0.38), suggesting only highly complex control structures reduce fuzzing success. Overall, static and symbolic tools are more sensitive to program structure than fuzzing.

A closer look at the rejected functions (Table 2) reveals several recurring patterns. Nearly all rejected cases feature at least one of: deep or nested control flow (e.g., multiple loops or branches), floating-point arithmetic, or use of randomness and non-deterministic library calls. For example, functions like `binary_search_iterations`, `matrix_determinant`, and `polygon_area_calculator` combine high control-flow depth with complex data manipulations, while `convert_temperature` and `calculate_discount` rely on floating-point comparisons that are notoriously difficult for SMT-based tools like CrossHair and Nagini. Randomness-heavy functions (e.g., `random_mod_calculator`) are systematically rejected, as symbolic and static verifiers cannot reason about non-deterministic outcomes.

Additionally, list comprehensions, dynamic data structures, and external library calls (such as math or datetime) are overrepresented among failures. These features increase the search space and introduce behaviors that are hard to model or prove correct in a general way. This analysis underscores that, while our pipeline is robust for simple, linear code, it currently struggles with the kinds of dynamic, data-rich, or numerically sensitive logic that typifies real-world Python programs. Addressing these limitations is a key direction for future work in automated formal verification for security.

5.4 Diagnostic Richness of Failures

We were able to analyze 18 separate failure cases across fuzzing, CrossHair, and Nagini, where we investigated syntax errors, conditional failures, counterexamples, and proof errors. Across all failed cases, the mean Failure Explanation Quality (FEQ) score is 0.53 (Table 7), indicating that most counterexamples and error messages are specific, actionable, and reasonably context-rich. Failures involving floating-point operations or complex data manipulations tend to yield the most detailed explanations (e.g., `convert_temperature`, `digit_sum_processor`), with FEQ scores as high as 0.74. In contrast, failures in randomness-heavy or highly dynamic code are less informative, as the tools struggle to generate concrete counterexamples or detailed failure points in program execution.

Not all failures, however, reflect true logical inequivalence of early and final assertions. A subset of five cases, marked as inconclusive (false positive) in Table 2 and detailed in Table 7, arise from tool limitations, unsupported language features, or type errors rather than genuine assertion mismatches. For example, `isbn_validator` and `matrix_determinant` both failed due to dynamic data manipulations and unsupported operations, while `date_difference_calculator` and `day_of_week_calculator` triggered false positives because of reliance on Python's datetime library, which is not modeled by the verification tools. `convert_temperature` produced a spurious failure due to floating-point precision issues. In our analysis, we distinguish these false positives from genuine counterexamples, ensuring that only true logical failures are counted as rejections. This distinction is critical for accurately assessing both the strengths and the current boundaries of automated formal verification in practice.

Notably, functions that fail both CrossHair and Nagini typically receive the highest FEQ scores, benefiting from the complementary strengths of symbolic and static analysis. This multi-tool approach not only increases the likelihood of catching subtle errors but also improves the quality of diagnostic feedback, helping developers quickly identify and address the root causes of assertion inequivalence.

5.5 Tool Coverage Issues

We also found that a number of the programs in our 50-program corpus contained features that were unsupported by Nagini, preventing them from being modeled in the verifier's specification language (See Table 8 for a detailed list). The most common unsupported features are floating-point arithmetic (80% of unsupported cases), list comprehensions (65%), randomness (40%), and datetime or external library calls (30%). These features either introduce non-determinism, require complex heap modeling, or depend on external semantics that are not captured by the verification engines.

Representative examples include `date_difference_calculator` and `day_of_week_calculator` (unsupported due to datetime operations), `isbn_validator` and `loop_string_hash` (dynamic comprehensions and string/list indexing), `matrix_determinant` (modular arithmetic and matrix ops), and `random_mod_calculator` (randomness). Many mathematical and data-rich programs, such as `factorial_root_calculator`, `polygon_area_calculator`, and `mean_absolute_deviation`, are also unsupported due to their use of advanced math functions, rounding, or complex control flow.

6 Discussion

6.1 Contributions

SENTINEL's primary contribution lies in its integration of LLM-based assertion synthesis with a multi-layered verification pipeline that combines symbolic execution, fuzzing, and static analysis. While prior systems tend to favor either generative approaches without guarantees [6, 12] or heavyweight static verification [3], SENTINEL bridges both, offering a realistic yet formally grounded architecture for increasing assurance. Crucially, it treats early assertion synthesis not as a generation task, but as a transformation within an existing program.

1. We demonstrate that early assertions, synthesized by LLMs and relocated from downstream checks, can be formally validated to preserve program behavior ($\phi_{\text{early}} \Leftrightarrow \phi_{\text{final}}$) across a diverse set of Python functions. Notably, 70% of our benchmark functions were either fully verified or validated by at least two independent methods, illustrating the practical feasibility of our approach. This systematic, multi-tool strategy for discharging equivalence obligations goes beyond prior weakest precondition tools [28, 27] or natural precondition synthesis [18], which often lack proof of semantic equivalence.

2. We empirically demonstrate that moving assertions earlier in Python programs enables fail-fast enforcement of safety properties, with over 80% of failures intercepted almost immediately after input. This proactive approach enhances runtime security, supports Design by Contract principles [5, 4], and offers practical benefits for debugging and vulnerability containment.

3. We provide a robust framework for *identifying practical breakpoints* of existing formal testing and verification tools in Python security: by reliably screening for Nagini incompatibility via static analysis and LLMs, deterministically logging and categorizing failure reasons across all methods, and distinguishing true counterexamples from tool-induced *false positives* and precisely identify residual verification blind spots. This expands on the capabilities of iRank's failure categorization and SmartFix's verifier-guided repair in [20, 26].

6.2 Limitations

Our empirical results show that 82% of our benchmark programs contain features unsupported by Nagini's static verification such as floating-point arithmetic, list comprehensions, randomness, and external library calls —creating persistent "blind spots." These limitations, observed across our evaluation, fundamentally constrain the security guarantees achievable with tools like Nagini [3] and CrossHair [9], and underscore the current boundaries of formal methods for security in dynamic languages, particularly in reasoning about non-determinism, complex heap structures, and external semantics.

SENTINEL is also limited by the capabilities of LLM-based assertion synthesis, which does not guarantee soundness or completeness and may introduce semantic drift, especially for complex or state-dependent assertions as found in [8, 12, 14]. The pipeline further assumes that assertions are syntactically extractable and comparable, and our evaluation relies on a benchmark of 50 functions that may not capture the full complexity of large-scale or multi-module Python systems. Some steps, such as adapting code for static verification, require manual intervention, limiting automation and introducing the risk of unintended changes to program logic.

Looking forward, these challenges motivate concrete directions for future research. Extending the capabilities of static verification frameworks and developing more robust hybrid approaches - combining symbolic execution, fuzzing, and LLM-guided synthesis [16] - will be essential to close the gap between soundness and practical coverage. Advances in specification preservation across program transformations [25] offer promising strategies for ensuring that security properties are maintained as code evolves. Integrating our pipeline into secure development workflows, such as CI/CD pipelines for Python, could make formal security guarantees more actionable and accessible to practitioners. Addressing these challenges is critical for scaling formal methods to the full diversity of Python programs encountered in security-sensitive contexts.

6.3 Future Directions

SENTINEL opens several promising paths for future work, particularly in improving assertion quality, verification integration, and developer usability. (1) One direction involves incorporating richer contextual signals, such as contracts, test results, or natural language documentation, into the assertion synthesis process. Prior work like NL2Spec [29] and RetriGen [8] has shown that retrieval-augmented generation can ground LLMs in auxiliary context, but typically stops at producing formal specifications rather than executable program assertions. SENTINEL could extend these techniques to generate in-place, verifiable checks directly from high-level intent. (2) A second direction concerns tighter integration between synthesis and verification. While many current systems treat verification as a passive downstream filter [12, 27], SENTINEL could enable a more interactive verification loop, where failed proofs trigger real-time refinement or simplification of assertions. As noted in Section 6.1, SENTINEL already builds on the loose coupling present in ASSERTIFY [12] and the more rigorous guarantees of CompCertO [25], but further integration would improve compatibility with tools like Nagini [3]. (3) Improving developer usability remains essential for broader adoption. Tools like CrossHair [9] and SmartFix [26] prioritize correctness and repair but often lack transparency or actionable feedback for non-experts. SENTINEL could close this gap through natural language diagnostics, failure visualizations, and editor-integrated explanations, features designed to make formal reasoning more intuitive and accessible for general-purpose programmers.

References

[1] Ball, Thomas, and Jakub Daniel. "Deconstructing Dynamic Symbolic Execution." *Microsoft Research Technical Report MSR-TR-2015-95*, Jan. 2015. https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/

[2] Bruni, Alessandro Maria, Tim Disney, and Cormac Flanagan. "A Peer Architecture for Lightweight Symbolic Execution." *University of California, Santa Cruz*, 2011.

[3] Eilers, Marco, and Peter Müller. "Nagini: A Static Verifier for Python." *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018*. Ed. Hana Chockler and Georg Weissenbacher. Springer, 2018, pp. 596–603. Lecture Notes in Computer Science, vol. 10981.

[4] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 1992.

[5] Schneider, Fred B. "Enforceable Security Policies." *ACM Transactions on Information and System Security*, vol. 3, no. 1, 2000, pp. 30–50.

[6] Watson, Cody, et al. "On Learning Meaningful Assert Statements for Unit Test Cases." *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, Association for Computing Machinery, 2020, pp. 1398–1409. https://doi.org/10.1145/3377811.3380429

[7] Yu, Han, et al. "Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning." *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 163–174. https://doi.org/10.1145/3510003.3510149

[8] Zhang, Quanjun, et al. "Improving Deep Assertion Generation via Fine-Tuning Retrieval-Augmented Pre-trained Language Models." *ACM Transactions on Software Engineering and Methodology*, just accepted, Feb. 2025, Association for Computing Machinery. https://doi.org/10.1145/3721128

[9] "CrossHair." https://github.com/pschanely/CrossHair

[10] "UsingAssertionsEffectively." Python Wiki, wiki.python.org, https://wiki.python.org/moin/UsingAssertionsEffectively. Accessed 7 May 2025.

[11] "GCP 5 - Using Assertions." Geo-Python 2023, geo-python-site.readthedocs.io/en/lih/notebooks/L6/gcp-5-assertions.html. Accessed 7 May 2025.

[12] Torkamani, Mohammad, et al. "ASSERTIFY: Utilizing Large Language Models to Generate Assertions for Production Code." *arXiv preprint arXiv:2411.16927*, 2024. https://doi.org/10.48550/arXiv.2411.16927

[13] Zhang, Quanjun, et al. "Exploring Automated Assertion Generation via Large Language Models." *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, 2025, pp. 1–25. https://doi.org/10.1145/3699598

[14] Pulavarthi, Vaishnavi, et al. "AssertionBench: A Benchmark to Evaluate Large-Language Models for Assertion Generation." *arXiv preprint arXiv:2406.18627*, 2024. https://arxiv.org/abs/2406.18627

[15] Pei, Kexin, et al. "Can Large Language Models Reason About Program Invariants?" *Proceedings of the 40th International Conference on Machine Learning (ICML)*, Honolulu, HI, USA, 2023, pp. 27496–27520. https://doi.org/10.5555/3618408.3619552

[16] Liu, Chang, et al. "Towards General Loop Invariant Generation: A Benchmark of Programs with Memory Manipulation." *arXiv preprint arXiv:2311.10483v2*, 2024. https://arxiv.org/abs/2311.10483v2

[17] Wang, Wenhan, et al. "Python Symbolic Execution with LLM-powered Code Generation." *arXiv preprint arXiv:2409.09271*, 2024. https://arxiv.org/abs/2409.09271

[18] Dinella, Elizabeth, et al. "Program Structure Aware Precondition Generation." *arXiv preprint arXiv:2310.02154*, 2024. https://arxiv.org/abs/2310.02154

[19] Menguy, Grégoire, et al. "A Query-Based Constraint Acquisition Approach for Enhanced Precision in Program Precondition Inference." *Journal of Artificial Intelligence Research*, vol. 82, 2025. https://doi.org/10.1613/jair.1.16206

[20] Chakraborty, Saikat, et al. "Ranking LLM-Generated Loop Invariants for Program Verification." *Findings of the Association for Computational Linguistics: EMNLP*, 2023. arXiv:2310.09342. https://doi.org/10.48550/arXiv.2310.09342

[21] Hellendoorn, Vincent J., et al. "Deep Learning Type Inference." *Proceedings of the 2018 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2018)*, 26 Oct. 2018, Association for Computing Machinery. https://doi.org/10.1145/3236024.3236051

[22] Si, Xujie, et al. "Code2Inv: A Deep Learning Framework for Program Verification." *Computer Aided Verification*, 2020, pp. 151–164. Lecture Notes in Computer Science, vol. 12225. https://doi.org/10.1007/978-3-030-53291-8_9

[23] Yu, Shiwen, et al. "Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning." *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, 2023, pp. 175–187. https://doi.org/10.1145/3597926.3598047

[24] Yao, Jianan, et al. "Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks." *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 15–20, 2020, pp. 106–120. https://doi.org/10.1145/3385412.3385986

[25] Koenig, Jérémie, and Zhong Shao. "CompCertO: Compiling Certified Open C Components." *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, Virtual Event, 2021, pp. 1095–1110. `https://doi.org/10.1145/3453483.3454097`

[26] So, Sunbeom, and Hakjoo Oh. "SmartFix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models." *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, CA, USA, 2023, pp. 185–197. `https://doi.org/10.1145/3611643.3616341`

[27] Rak-amnouykit, Ingkarat, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. "Principled and Practical Static Analysis for Python: Weakest Precondition Inference of Hyperparameter Constraints." *Software: Practice and Experience*, vol. 54, no. 3, 2024, pp. 363–393. `https://doi.org/10.1002/spe.3279`

[28] Rak-amnouykit, Ingkarat, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. "The Raise of Machine Learning Hyperparameter Constraints in Python Code." *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 580–592. `https://doi.org/10.1145/3533767.3534400`

[29] Cosler, Matthias, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. "nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models." *arXiv preprint arXiv:2303.04864*, 2023. `https://doi.org/10.48550/arXiv.2303.04864`

[30] Hassan, Mostafa, Caterina Urban, Marco Eilers, and Peter Müller. "MaxSMT-Based Type Inference for Python 3." *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, 2018, pp. 12–19. `https://doi.org/10.1007/978-3-319-96142-2_2`

[31] Laursen, Mathias Rud, Wenyuan Xu, and Anders Møller. "Reducing Static Analysis Unsoundness with Approximate Interpretation." *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, 2024, pp. 1165–1188. `https://doi.org/10.1145/3656424`

Appendix

`https://github.com/dhruvtpatel/2540_finalproj`

Figure 5: GitHub Repository with Code Files, Scripts, Results, etc.

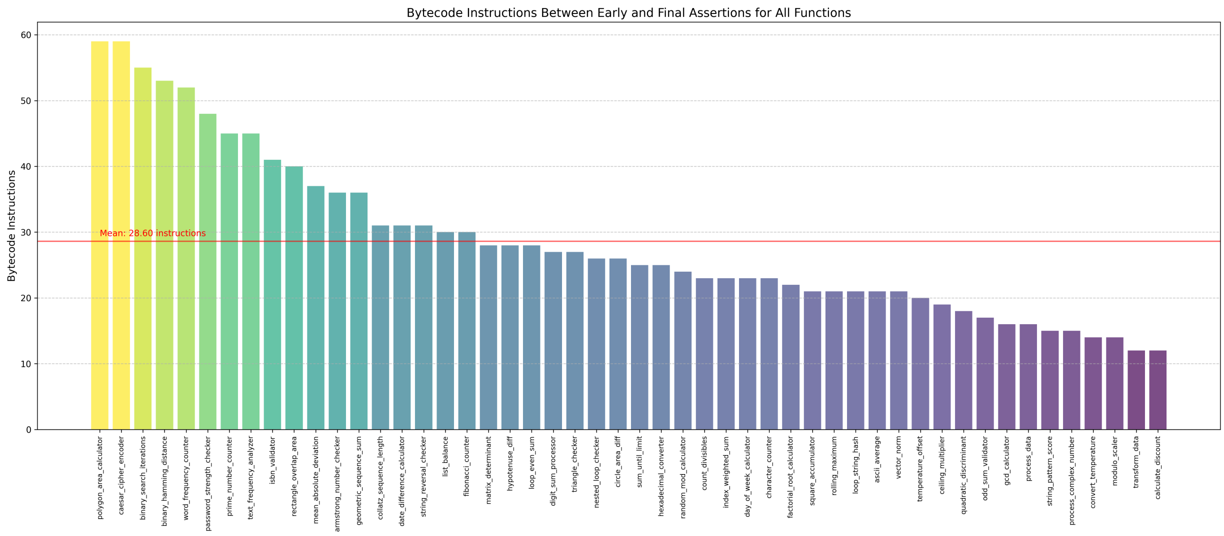

Figure 6: Bytecode Instructions Between Early and Final Assertion Locations in Each Program

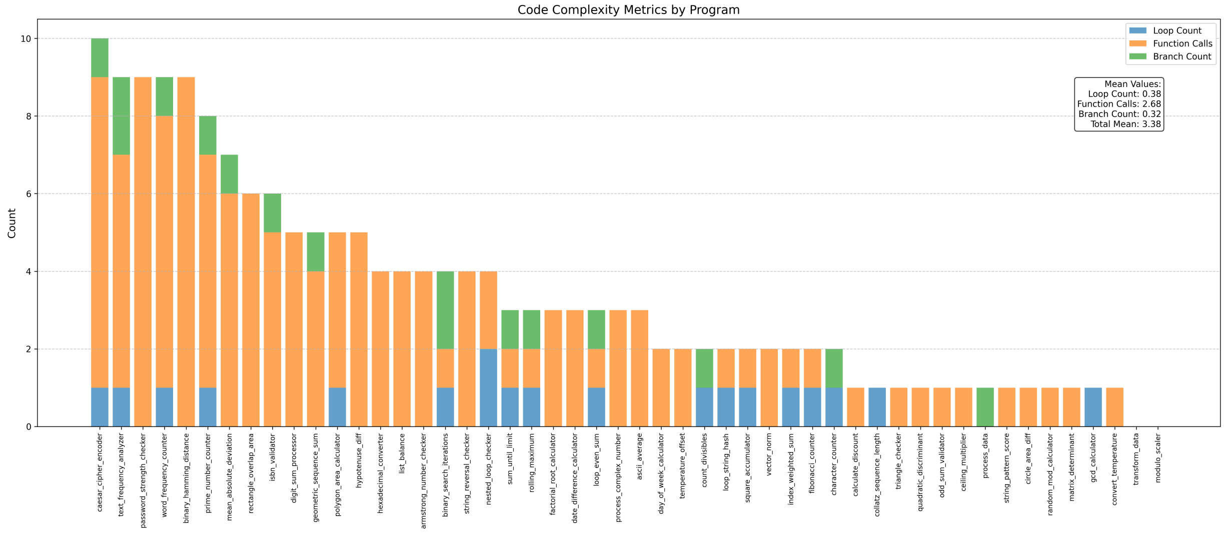

Figure 7: Code Operations Between Early and Final Assertion Locations in Each Program

Program	Difficulty	Fuzz	CrossHair	Nagini	FEQ (method)
armstrong_number_checker	7	pass	pass	unsupported	NA
ascii_average	6	pass	pass	unsupported	NA
binary_hamming_distance	10	pass	pass	unsupported	NA
binary_search_iterations	10	fail	pass	unsupported	NA
caesar_cipher_encoder	10	pass	pass	unsupported	NA
calculate_discount	6	pass	fail	unsupported	0.66 (crosshair)
ceiling_multiplier	6	pass	pass	fail	0.39 (nagini)
character_counter	6	pass	pass	unsupported	NA
circle_area_diff	7	pass	pass	unsupported	NA
collatz_sequence_length	7	pass	pass	pass	NA
convert_temperature	6	inconclusive (false positive)	fail	unsupported	~~0.20(fuzz)~~, 0.74 (crosshair)
count_divisibles	6	pass	pass	unsupported	NA
date_difference_calculator	8	inconclusive (false positive)	pass	unsupported	~~0.66 (fuzz)~~
day_of_week_calculator	8	inconclusive (false positive)	pass	unsupported	~~0.75 (fuzz)~~
digit_sum_processor	9	pass	fail	unsupported	0.74 (crosshair)
factorial_root_calculator	7	pass	fail	unsupported	0.68 (crosshair)
fibonacci_counter	6	pass	pass	unsupported	NA
gcd_calculator	7	pass	pass	unsupported	NA
geometric_sequence_sum	10	pass	pass	unsupported	NA
hexadecimal_converter	7	pass	pass	unsupported	NA
hypotenuse_diff	9	pass	pass	unsupported	NA
index_weighted_sum	6	pass	pass	unsupported	NA
isbn_validator	9	inconclusive (false positive)	pass	unsupported	~~0.50 (fuzz)~~
list_balance	8	pass	pass	fail	0.57 (nagini)
loop_even_sum	8	pass	pass	unsupported	NA
loop_string_hash	6	pass	pass	unsupported	NA
matrix_determinant	7	inconclusive (false positive)	pass	unsupported	~~0.53 (fuzz)~~
mean_absolute_deviation	10	pass	pass	unsupported	NA
modulo_scaler	6	pass	pass	pass	NA
nested_loop_checker	7	pass	pass	unsupported	NA
odd_sum_validator	6	pass	pass	unsupported	NA
password_strength_checker	10	pass	pass	unsupported	NA
polygon_area_calculator	9	pass	pass	unsupported	NA
prime_number_counter	10	pass	pass	unsupported	NA
process_complex_number	6	pass	pass	fail	0.45 (nagini)
process_data	5	pass	fail	fail	0.61 (crosshair), 0.72 (nagini)
quadratic_discriminant	9	pass	pass	pass	NA
random_mod_calculator	7	pass	fail	unsupported	0.54 (crosshair)
rectangle_overlap_area	10	pass	pass	pass	NA
rolling_maximum	6	pass	pass	unsupported	NA
square_accumulator	6	pass	pass	unsupported	NA
string_pattern_score	5	pass	pass	unsupported	NA
string_reversal_checker	8	fail	fail	unsupported	0.54 (crosshair)
sum_until_limit	6	pass	pass	unsupported	NA
temperature_offset	7	pass	pass	unsupported	NA
text_frequency_analyzer	10	pass	pass	unsupported	NA
transform_data	6	pass	pass	pass	NA
triangle_checker	7	pass	pass	unsupported	NA
vector_norm	8	pass	pass	unsupported	NA
word_frequency_counter	10	pass	pass	unsupported	NA

Table 2: Program Execution Results Across Verification Methods

Method	Pass	Fail	Unsupported	Inconclusive
Fuzzing	43 (86%)	2 (4%)	0 (0%)	5 (10%)
CrossHair	43 (86%)	7 (14%)	0 (0%)	0 (0%)
Nagini	5 (10%)	4 (8%)	41 (82%)	0 (0%)

Table 3: Summary of verification outcomes across all 50 programs. Each row sums to 50 total programs. Inconclusive results were initially categorized as failures but later identified as false positives due to tool limitations rather than logical inequivalence.

Code Structure	Fuzz Success	CrossHair Success	Nagini Success
Pure functions without loops	100%	81%	50%
Pure functions with simple loops	100%	80%	33%
Functions with single nested loops	100%	100%	0%
Functions with mathematical operations	100%	67%	40%
Functions with string operations	100%	80%	0%
Functions with list/array operations	86%	86%	25%
Functions with randomness or non-determinism	100%	0%	-

Table 4: Impact of different code structures on verification success rates across methods. Calculated using (passes)/(passes+fails), excluding the inconclusive and false positive cases

Program	Diff.	Params	Ops	CF	DType	Assert	Math	Loop	Calls	Branch
binary_search_iterations	3.72	1.20	5.05	7.85	0.93	0.66	0.33	2.48	0.48	2.42
caesar_cipher_encoder	3.61	1.20	4.65	3.22	0.93	0.60	0.33	4.60	10.04	0.50
rectangle_overlap_area	3.59	19.20	5.40	0.00	0.25	0.54	0.09	0.00	13.04	0.00
prime_number_counter	3.28	0.30	4.18	3.22	0.25	0.54	0.71	3.83	8.66	0.03
geometric_sequence_sum	3.20	2.70	5.40	0.70	0.93	0.57	5.11	0.00	2.80	0.92
polygon_area_calculator	2.85	0.30	4.65	0.70	0.25	0.56	2.24	1.40	2.80	0.00
collatz_sequence_length	2.83	0.30	4.65	0.70	0.25	0.72	1.22	1.73	0.00	0.00
text_frequency_analyzer	2.76	0.30	2.85	3.22	0.25	0.63	0.09	1.91	0.78	0.59
loop_even_sum	2.72	1.20	3.60	3.22	0.25	0.59	0.33	0.83	0.62	0.14
mean_absolute_deviation	2.69	0.30	4.18	0.70	0.25	0.48	1.53	0.00	6.95	0.20
nested_loop_checker	2.62	0.30	1.80	3.22	0.25	0.60	0.09	2.90	1.55	0.00
word_frequency_counter	2.61	0.30	2.85	3.22	0.25	0.62	0.09	1.40	0.95	0.03
binary_hamming_distance	2.59	1.20	1.80	0.00	0.25	0.64	0.09	0.00	15.22	0.00
quadratic_discriminant	2.58	2.70	4.65	0.00	0.25	0.69	2.43	0.00	0.11	0.00
isbn_validator	2.55	0.30	3.60	0.70	0.25	0.58	0.33	0.00	5.04	0.50
count_divisibles	2.49	0.30	2.85	3.22	0.25	0.63	0.33	0.32	0.00	0.14
circle_area_diff	2.46	0.30	4.65	0.00	0.25	0.64	2.43	0.00	0.11	0.00
character_counter	2.39	0.30	1.80	3.22	0.25	0.60	0.09	0.17	0.00	0.80
sum_until_limit	2.34	0.30	1.80	3.22	0.25	0.57	0.00	0.17	0.48	0.41
rolling_maximum	2.32	0.30	1.80	3.22	0.25	0.54	0.00	0.17	0.48	0.41
fibonacci_counter	2.28	0.30	2.85	0.70	0.25	0.64	0.09	0.60	0.48	0.00
index_weighted_sum	2.27	0.30	2.85	0.70	0.25	0.62	0.33	0.60	0.48	0.00
vector_norm	2.27	1.20	3.60	0.00	0.25	0.50	3.54	0.00	0.78	0.00
square_accumulator	2.26	0.30	2.85	0.70	0.25	0.60	0.33	0.60	0.48	0.00
matrix_determinant	2.26	0.30	4.18	0.00	0.25	0.61	1.08	0.00	0.11	0.00
temperature_offset	2.26	0.30	4.18	0.00	0.25	0.58	0.60	0.00	0.86	0.00
digit_sum_processor	2.25	0.30	2.85	0.00	0.25	0.58	0.25	0.00	5.96	0.00
loop_string_hash	2.24	0.30	2.85	0.70	0.25	0.57	0.33	0.60	0.48	0.00
modulo_scaler	2.23	0.30	4.18	0.00	0.25	0.60	0.71	0.00	0.00	0.00
random_mod_calculator	2.22	0.30	4.18	0.00	0.25	0.58	0.71	0.00	0.07	0.00
convert_temperature	2.22	0.30	3.60	0.00	0.25	0.78	0.60	0.00	0.11	0.00
gcd_calculator	2.21	1.20	2.85	0.70	0.25	0.52	0.33	0.50	0.00	0.00
hypotenuse_diff	2.20	1.20	2.85	0.00	0.25	0.46	1.08	0.00	4.86	0.00
password_strength_checker	2.20	0.30	1.80	0.00	0.25	0.62	0.09	0.00	7.36	0.00
transform_data	2.14	0.30	3.60	0.00	0.25	0.68	0.33	0.00	0.00	0.00
list_balance	2.13	0.30	2.85	0.00	0.25	0.57	0.25	0.00	3.85	0.00
date_difference_calculator	2.11	2.70	2.85	0.00	0.25	0.58	0.25	0.00	0.86	0.00
armstrong_number_checker	2.11	0.30	1.80	0.00	0.25	0.64	0.33	0.00	5.04	0.00
hexadecimal_converter	2.10	0.30	1.80	0.00	0.25	0.63	0.00	0.00	5.40	0.00
calculate_discount	2.08	1.20	2.85	0.00	0.25	0.73	0.25	0.00	0.11	0.00
day_of_week_calculator	2.07	2.70	2.85	0.00	0.25	0.59	0.09	0.00	0.15	0.00
factorial_root_calculator	2.06	0.30	1.80	0.00	0.25	0.60	2.24	0.00	2.80	0.00
string_reversal_checker	2.03	0.30	2.85	0.00	0.25	0.57	0.05	0.00	2.14	0.00
process_data	2.02	0.30	1.80	0.70	0.25	0.47	0.09	0.00	0.00	0.92
ceiling_multiplier	1.96	0.30	2.85	0.00	0.25	0.54	0.82	0.00	0.41	0.00
odd_sum_validator	1.94	0.30	2.85	0.00	0.25	0.56	0.33	0.00	0.48	0.00
ascii_average	1.92	0.30	1.80	0.00	0.25	0.54	0.09	0.00	3.09	0.00
triangle_checker	1.88	2.70	1.80	0.00	0.25	0.51	0.00	0.00	0.48	0.00
process_complex_number	1.84	1.20	0.00	0.00	0.25	0.70	1.08	0.00	1.44	0.00
string_pattern_score	1.76	0.30	1.80	0.00	0.25	0.55	0.09	0.00	0.48	0.00

Table 5: Component-wise complexity scores for each program. Diff. = Overall Difficulty, Ops = Operations, CF = Control Flow, DType = Data Type, Assert = Assertions, Math = Mathematical Complexity.

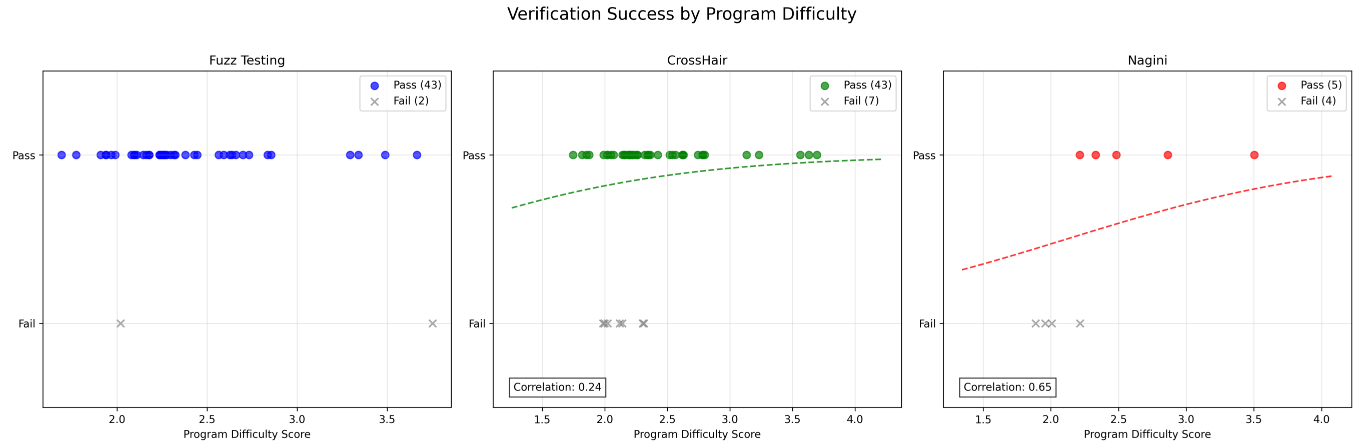

Figure 8: Correlation between difficulty scores and verification method outcomes.

Method	Param Comp.	Op Dens.	CF Depth	Type Div.	Assert Comp.	Math Soph.	Loop Comp.	Call Dens.	Branch Comp.
Fuzz	0.02	-0.16	-0.38	-0.37	-0.09	0.11	NA	NA	NA
CrossHair	0.10	0.08	0.22	0.10	-0.13	0.03	NA	NA	NA
Nagini	0.34	0.82	-0.06	NA	0.45	0.28	NA	NA	NA

Table 6: Correlation between component scores and verification success for each method. "NA" indicates insufficient data for correlation.

Program	Test Method	Input	FEQ	False Positive	Specificity	Actionability	Context	Technical Detail
day_of_week_ calculator	fuzzing	year=48, month=89, day=69	0.75	True	0.90	0.80	0.70	0.50
convert_ temperature	crosshair	convert_ temperature_ transformed(38.055...)	0.74	False	0.80	0.60	0.90	0.70
digit_sum_ processor	crosshair	digit_sum_ processor_ transformed(298)	0.74	False	0.80	0.60	0.70	0.90
process_data	nagini	N/A - static verification	0.72	False	0.8	0.6	0.7	0.8
factorial_root_ calculator	crosshair	factorial_root_ calculator_ transformed(5)	0.68	False	0.80	0.60	0.70	0.60
calculate_ discount	crosshair	calculate_ discount_ transformed(101.0, 0.502...)	0.66	False	0.80	0.60	0.50	0.70
date_difference_ calculator	fuzzing	year=56, month=78, day=7	0.66	True	0.70	0.50	0.70	0.80
list_balance	nagini	N/A - static verification	0.57	False	0.60	0.50	0.50	0.70
process_data_ module.py	crosshair	process_data_ transformed(-50)	0.61	False	0.70	0.60	0.50	0.60
process_complex _number	nagini	N/A - static verification	0.45	False	0.50	0.40	0.60	0.30
random_mod_ calculator	crosshair	random_mod_ calculator_ transformed(13)	0.54	False	0.50	0.30	0.70	0.80
string_reversal_ checker	crosshair	string_reversal_ checker_ transformed('\x00\x01')	0.54	False	0.60	0.40	0.50	0.70
matrix_ determinant	fuzzing	matrix=[0]	0.53	True	0.60	0.70	0.40	0.30
ceiling_multiplier.py	nagini	N/A - static verification	0.39	False	0.40	0.30	0.40	0.50
binary_search_ iterations	fuzzing	arr=[59, 4, 91, ...], target=83	0.50	False	0.50	0.50	0.60	0.40
isbn_validator	fuzzing	isbn='^{3}\x95 \U00014f7b...'	0.50	True	0.70	0.50	0.30	0.40
string_reversal_ checker	fuzzing	text='ÖnÄ...'	0.36	False	0.40	0.40	0.30	0.30
convert_ temperature	fuzzing	celsius=37.558...	0.20	True	0.30	0.10	0.20	0.20

Table 7: Failure Explanation Quality (FEQ) scores and component metrics for each failure case. Higher scores indicate more informative failure explanations.

Program	Reason Nagini Cannot Support
armstrong_number_checker	Cannot represent comparison of logical transformations internally within a function; complex operations with string conversions and arithmetic don't map neatly into Nagini's permission system
ascii_average	Does not support while loops with collection indexing ('while $(i < \text{len}(s))$' with operations like 'total$+= \text{ord}(s[i])$')
binary_hamming_distance	Does not support direct manipulation of binary representations or string operations such as bin() and zfill()
binary_search_iterations	Does not support while loops with collection indexing with operations like 'if $(arr[(i-1)] > arr[i])$'
caesar_cipher_encoder	Cannot represent complex string transformations and character encoding operations using list comprehensions and built-in string methods
calculate_discount	Does not support floating-point arithmetic and checks for equality with small tolerances
character_counter	Does not support while loops with collection indexing ('while $(i < \text{len}(\text{text}))$' with operations like 'if is_vowel(text[i])')
circle_area_diff	Cannot support floating-point arithmetic, rounding functions, and mathematical constants like math.pi
convert_temperature	Involves floating-point computations which can introduce precision errors not supported by Nagini
count_divisibles	Does not support while loops with collection indexing ('while $(i < \text{len}(\text{nums}))$' with operations like 'if $((\text{nums}[i] \bmod 4) == 0)$')
date_difference_calculator	Cannot express operations related to datetime and date calculations
day_of_week_calculator	Cannot represent logic involving Python's datetime module
digit_sum_processor	Does not support while loops with collection indexing ('while $(i < \text{len}(s))$' with operations like 'total$+= \text{int}(s[i])$')
factorial_root_calculator	Cannot handle functions from the math module like factorial and sqrt
fibonacci_counter	Cannot support asserting equivalence between two conditions which is inherently a runtime concept
gcd_calculator	Cannot represent or utilize dynamic values like math.gcd(a, b)
geometric_sequence_sum	Does not support floating point arithmetic and rounding, and dynamic conditions like summing a geometric sequence when $\|r\| \geq 1$
hexadecimal_converter	Does not support string manipulation and arithmetic operations on strings transformed into integers
hypotenuse_diff	Cannot translate operations like math.hypot and round due to limitations on numeric precision
index_weighted_sum	Does not support verification of properties like equivalence between early and final assertion checks within functions
isbn_validator	Cannot model dynamic list comprehensions and dynamic assertions that depend on runtime evaluation
loop_even_sum	Does not support inline assertions and complex operations like summing over a range with conditions
loop_string_hash	Does not support while loops with collection indexing ('while $(i < \text{len}(\text{text}))$' with operations like 'hash_val$+= (\text{ord}(\text{text}[i]) \cdot 3)$')
matrix_determinant	Cannot support computing matrix determinant and performing modular arithmetic
mean_absolute_deviation	Does not support rounding operations, list comprehensions, and transformations within assertion checks
nested_loop_checker	Cannot verify computation of exact value of sum within nested loops with complex condition equivalence checks
odd_sum_validator	Cannot handle dynamically computed properties such as sum and filtering operations directly
password_strength_checker	Does not support dynamic features like list comprehensions and str methods like isupper(), islower(), isdigit()
polygon_area_calculator	Does not support operations like the Shoelace formula involving iteration over list of tuples with arithmetic operations
prime_number_counter	Cannot express or verify properties about specific numbers being prime due to mathematical complexity
random_mod_calculator	Does not support randomness through random.randint function
rolling_maximum	Cannot represent comparing early computation with final computation for verification of equivalence
square_accumulator	Cannot verify properties involving complex calculations, modular arithmetic, and ensuring equivalence of early and final state checks
string_pattern_score	Cannot represent for loops to filter characters and list comprehensions to compute sums
string_reversal_checker	Does not provide built-in support for string slicing, reversal, and stripping operations
sum_until_limit	Cannot handle complex intermediate computation with list comprehensions and nested conditions
temperature_offset	Does not handle floating-point arithmetic and operations like rounding and converting to integer
text_frequency_analyzer	Does not support dictionary operations and dynamic updates to dictionaries
triangle_checker	Cannot represent list sorting operations
vector_norm	Cannot handle operations like math.sqrt and round on float numbers
word_frequency_counter	Cannot represent counting occurrences within a set and a dictionary

Table 8: Programs Not Convertible to Nagini and Their Limitations