SENTINEL: Moving Assertions Earlier for Enhanced Python Program Safety

Research Question

Many security and correctness violations in software are detected only after critical operations execute, making remediation costly and often too late to prevent harm. By inserting assertions earlier in program execution that capture the same logical condition as a later final check, we can detect violations sooner, reduce error propagation, and improve program safety and efficiency. We aim to pursue this line of research for Python programs. We break down existing research and motivations in detail in the later related works section.

We work towards answering: *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?*

Implementation (How do we plan to address this?)

To address our research question, we implement a three-phase approach combining LLM-based assertion generation with systematic program transformation and formal verification. This approach **verifies the logical equivalence between early and final assertions**, ensuring that critical properties are preserved. In doing so, we **effectively move equivalent assertions earlier in the program**, **avoiding unnecessary execution and computation** on inputs while **still enforcing the same properties as the original later assertions** in the program.

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}).

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.*

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.

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
    z = y if y > 0 else -y
    b_final = (z == 100)

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

Evaluation & Measuring Success

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.

To check whether a generated early assertion (ϕ_{early}) is equivalent to the final assertion (ϕ_{final}) in a given program, we apply the aforementioned pipeline consisting of symbolic execution, fuzz testing, and static verification **on the transform program**. A transformation is considered successful if the transform program satisfies all of the following checks:

1. We execute the transformed program with symbolic inputs using CrossHair. CrossHair systematically explores feasible paths and attempts to find counterexamples where the assertion $b_{early} == b_{final}$ fails. A successful run contains no such counterexamples.

2. We subject the same transformed program to fuzz testing using Hypothesis. This generates randomized and edge-case inputs in search of violations of assertion equivalence. Success here means that no input generated (in the limited cases we define) causes a mismatch between ϕ_{early} and ϕ_{final}.

3. If applicable, we run Nagini on the transformed program to statically verify the logical equivalence $b_{early} == b_{final}$ holds for all possible inputs and paths. A verification pass from Nagini is a strong formal guarantee of correctness.

For each program analyzed, we classify the result as:

- **Verified:** All three stages (symbolic execution, fuzzing, static verification) pass.
- **Validated:** Symbolic execution and fuzz testing pass, but static verification is inconclusive or not applicable.
- **Rejected:** Any stage produces a counterexample or fails to prove equivalence.

This evaluation ensures that only **early assertions 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.

We **assign each program a difficulty rating (1–10)** based on these features. The full pipeline—from assertion generation to verification—is applied to each program, and we log equivalence results, failure stages, and verification time. While we use an early version of this rating system now, we plan to **refine it through further study of how different features impact difficulty**.

For **rejected and not verified transformations**, we introduce a **Failure Explanation Quality (FEQ) score (1-10)** that measures counterexample specificity, diagnostic clarity, and actionability of the failure information. This provides an additional metric to evaluate our verification pipeline's effectiveness even when transformations fail, with a target average FEQ score of 7 or higher across all failed cases.

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. Success is measured not only by **correctness and soundness** (through symbolic execution, fuzz testing, and static verification), but also by the method's **ability to scale** to programs of varying size, complexity, and structure—while maintaining **low false positive rates and manageable performance overhead.**

Preliminary Results

Code & Results: `https://colab.research.google.com/drive/1Bs9N_T-TsuBAiaKfM7JNh1M23fJOSJXB?usp=sharing`

For this checkpoint, we implemented a prototype version of our assertion equivalence pipeline on a small, curated set of Python programs. The goal was to evaluate whether early assertions generated via a large language model (GPT-o3-mini) could be verified to be logically equivalent to final assertions via symbolic execution, fuzz testing, and explore future static verification methodology (which we plan to implement thoroughly by checkpoint 3).

Test Programs. We evaluated the pipeline on five programs, each implementing basic arithmetic, control flow, or structural operations such as multiplication, discount calculation, and data transformation. These programs were chosen to vary in complexity and represent distinct structural features. We manually rated each program's difficulty using a scoring rubric that considers: (1) Number of Parameters, (2) Depth of control flow (e.g., conditionals, loops), (3) Number of mathematical operations and use of rounding/complex functions, and (4) Data types and structure usage.

Each program was assigned a difficulty rating on a scale of 1–10. For example, `process_data` received a score of 5/10 due to moderate branching and arithmetic; `process_complex_number` rated higher due to use of magnitude and complex numbers. This system is relatively arbitrary at the moment, but as we assemble a larger dataset of programs, we aim to tune this ranking system to be more representative of relative difficulty, which will help us better quantify results. **Please see the attached google colab for the programs and the formal code behind our difficulty rating system.**

Fuzz Testing Results. 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. **All five programs passed fuzz testing** without failing any of the random input tests, increasing our confidence in the equivalence of the LLM-inserted early assertions. These results are noted in the attached code notebook.

Symbolic Execution Results. We ran CrossHair symbolic execution on each of the transformed programs. For three of the programs—`process_data`, `convert_temperature`, and `calculate_discount`—CrossHair identified counterexamples where $b_{early} \neq b_{final}$, suggesting that the early assertion does not hold equivalently for all inputs. However, for `process_complex_number` and `transform_data`, CrossHair found no counterexamples, suggesting strong behavioral equivalence under bounded symbolic exploration.

Program	Difficulty (1–10)	Fuzz Testing	CrossHair Result	Counterexample Input
`process_data`	5	Pass	Fail	$x = 0$
`convert_temperature`	5	Pass	Fail	$celsius = 38.0$
`calculate_discount`	5	Pass	Fail	$price = 120,\ discount = 0.4$
`process_complex_number`	6	Pass	Pass	N/A
`transform_data`	4	Pass	Pass	N/A

Table 1: Summary of preliminary results from fuzz testing and symbolic execution (Also in attached notebook)

Planned Next Steps. By the next checkpoint, we plan to integrate Nagini for full static verification to address inadequacies in solely using symbolic execution and fuzz testing. These methods, as noted previously, only provide us initial insight into potential bugs, but we can't deterministically verify all paths and provably show assertion equivalence, hence the need for Nagini's static verification capabilities.

This will allow us to formally prove the logical equivalence $\phi_{early} \Leftrightarrow \phi_{final}$ across all paths and inputs. We are also expanding our program corpus to include significantly longer and more complex functions (with multiple assertions), allowing us to evaluate the scalability of the pipeline. Additionally, we aim to analyze per-stage failure paths (by implementing the Failure Explanation Quality score detailed above) and runtime overhead on more a more diverse set of Python programs.

Discussion & Related Works

Enforceable Security Policies and Early Assertion Placement

Foundational work on security policy enforcement established that certain policies (notably safety properties) can be enforced at runtime by monitoring program executions and halting before a violation occurs [5]. **Schneider's paper on enforceable security policies** 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]. This implies that if a violation of a desired property will inevitably occur later in an execution, a monitor (or inline assertion) should trigger as soon as that condition becomes detectable, effectively shifting the check earlier without changing which executions are deemed safe [5]. 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 Design by Contract, where software components check preconditions at the interface and immediately flag contract violations rather than propagate errors [4]. **Meyer's Design by Contract paradigm** 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.

Static Verification Tools for Python and User-Defined Assertions

Fully static formal verification of Python programs is challenging due to dynamic typing and constructs. Nonetheless, recent research has produced tools that support user-specified assertions and specifications in Python. **Eilers' et. al's Nagini paper** [3] is a notable example – it introduces a **sound static verifier for Python 3** that can prove memory safety, data race freedom, and user-supplied assertions in code [3]. Nagini requires type annotations (leveraging Python's gradual typing via Mypy) and uses a contract-like specification library to allow preconditions, postconditions, and assertions to be embedded in Python programs [3]. Internally, Nagini translates Python and its specifications into the Viper intermediate verification language, enabling automated reasoning via verification condition generation [3]. This modular verification approach

scales to real-world concurrent Python code and was used to verify a production networking architecture in Python [3]. Nagini was one of the first tools to provide automatic formal verification for Python beyond type checking [3]. Its ability to handle user-defined assertions is directly relevant to our project, as it indicates how final assertion conditions can be formally verified. We are using Nagini to to mitigate inadequacies of symbolic execution and fuzz testing and perform more formal assertion verification on the transform programs detailed above.

Assertion Generation with Deep Learning and Large Language Models

There is a rich vein of recent research on using machine learning to automatically generate program assertions (often in the context of test oracles). Early work by **Watson et al. introduced ATLAS** (Automatic Learning of Assert Statements), which applies a Neural Machine Translation model to generate meaningful assert statements for unit tests [6]. Given a test method body and the focal method under test, ATLAS predicts an assert statement that a developer might write to check correctness [6]. Impressively, ATLAS could exactly reproduce the manually written assert in about 31% of cases (50% within top-5 guesses) on thousands of real tests [6], suggesting that learning-based approaches can capture common correctness properties. However, subsequent studies identified limitations: neural sequence-to-sequence models like ATLAS sometimes generate unexplainable or incorrect assertions, especially when the assertion is long or involves unseen tokens [7]. To address this, **Yu et al. proposed an Information Retrieval-based method** that retrieves existing asserts from similar tests and adapts them to the new context [7]. Their approach provides more explainable results and handles longer assertions better, outperforming the pure neural approach by a significant margin [7]. More recently, researchers like have begun leveraging pre-trained large language models (LLMs) for assertion generation. **Zhang et. al** conducted an extensive study on using **state-of-the-art LLMs to generate unit test assertions** [8]. They developed **fine-tuned models like RetriGen that incorporate external knowledge to boost assertion accuracy**. For instance, they improve assertion generation by augmenting a CodeT5 model with retrieved candidate assertions (RetriGen), achieving over 57% exact-match accuracy and substantially outperforming prior techniques.

Symbolic Execution and Fuzz Testing for Python (and Limitations)

Several tools have adapted symbolic execution to Python's dynamic semantics. One one example is PyExZ3, a symbolic execution engine built to explore possible execution paths in Python programs using Z3 as an SMT backend, detailed in **Microsoft's *Deconstructing Dynamic Symbolic Execution* research paper** [1]. In our project, we use **CrossHair**, a more actively maintained symbolic execution tool for Python that utilizes a similar architecture. This approach—**lightweight symbolic execution using SMT backends**—was formalized by **Bruni et al**. [2], and serves as the basis for both PyExZ3 and CrossHair. We use CrossHair to check whether LLM-generated early assertions (ϕ_{early}) are behaviorally equivalent to final assertions (ϕ_{final}) across all execution paths in the transformed program. To complement symbolic execution, we use **Hypothesis**, a property-based fuzz testing framework that generates randomized and edge-case inputs to identify assertion mismatches.

However, both symbolic execution and fuzzing are **incomplete for verification**: they can discover counterexamples but cannot conclusively prove that none exist as they cannot individually reason over programs. Symbolic engines may miss paths due to state explosion or solver limitations, while fuzzing lacks exhaustive coverage—particularly in complex branching or recursive code. Thus, we treat these tools as **bug-finding mechanisms**, not proof systems. A passing result increases confidence but does not constitute formal equivalence. To close this gap, we incorporate the static verifier **Nagini**, which—under supported specifications—can formally prove that $\phi_{early} \Leftrightarrow \phi_{final}$ holds across all inputs and paths. While more restrictive, static verification provides stronger guarantees and a principled foundation for our pipeline.

Logical Equivalence of Conditions and Impact on Safety Verification

A core challenge is ensuring that an early inserted assertion is logically equivalent to the original final assertion. Schneider's work shows that a runtime monitor must halt execution when the monitored condition, equivalent to the policy, is violated [5]. In runtime verification terms, any enforceable property is a safety property that must hold for all execution prefixes [5]. Our project treats the final assertion as detailing a safety property and the early LLM-generated assertion as a monitor that finds potential violations. We aim to prove that EarlyAssertion ⇒ FinalAssertion and FinalAssertion ⇒ EarlyAssertion for all inputs, establishing equivalence. This guarantees that **proactive checking does not reject any safe execution nor overlook any true violation**, thus enhancing robustness without altering program semantics.

References

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

[2] Bruni, Alessandro Maria et al. "A Peer Architecture for Lightweight Symbolic Execution." (2011).

[3] Eilers, Marco, and Peter Muller. "Nagini: A Static Verifier for Python." Lecture Notes in Computer Science, vol. 10981, Springer Nature, 2018, pp. 596-603.

[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." 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 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