

Fooling LLM-Based Program Verifiers

Dev Arora, Lalit Aditya Julapalli, with Apart Research

May 2026

Abstract

LLM-based program verification systems promise stronger correctness guarantees than testing or type systems alone, but their robustness to adversarial inputs remains underexplored. We study Clover, a verification framework that checks correctness by enforcing mutual consistency across a program’s natural language docstring, formal specification, and implementation. We ask whether an LLM can systematically generate adversarial programs that evade Clover’s detection. Targeting the C6 corruption class, we build an automated generate-and-verify pipeline that uses Claude Sonnet 4.5 to simultaneously weaken a program’s formal specification and mutate its implementation. Of 62 programs from CloverBench, 23 mutated instances evaded all Clover checks, yielding a 37.1% evasion rate. The failure is concentrated in mutations where the weakened specification is logically equivalent to the original: Clover missed 22 of 30 such cases, a 73.3% evasion rate. Analysis of normalized edit distance reveals that subtle annotation changes ($\text{NED} \leq 0.05$) are the most dangerous, evading detection 52.6% of the time. Our results demonstrate that formal verification passing does not imply correctness when the specification itself can be adversarially weakened, exposing a fundamental blind spot in consistency-based verification frameworks.

1 Introduction

The rise of large language models (LLMs) has made AI assistance ubiquitous in software engineering, with 85% of developers now incorporating these tools into their workflows (JetBrains [2025]). Despite this widespread adoption, concerns over code quality persist: Pearce et al. [2021] found that 40% of code generated by GitHub Copilot contained security vulnerabilities.

Secure program synthesis is a response to the shortcomings of generated code—instead of relying on manual human review, programs are constructed that provably satisfy some formal specification. Tools such as Dafny allow developers to write programs alongside machine-checkable proofs of their properties, providing much stronger guarantees than testing or type systems. Since LLMs can be used to automate this process, this promise has become increasingly practical. The implicit assumption that is drawn from the capability is compelling – a program that a verifier accepts is a program that is completely correct.

However, verification only guarantees correctness with respect to the specification provided, not the programmer’s intent. If the spec fails to fully capture what the code is supposed to do, the verifier is proving the wrong thing.

The Clover system (Sun et al. [2024]) is an attempt to close this gap. Rather than relying on the formal verifier alone, Clover treats correctness as a property dependent on the consistency of three objects of a program: its natural language docstring, formal specification, and implementation.

In this paper, we challenge the interpretation by asking a different question: can an LLM generate adversarial programs that fool Clover? We focus on the C6 corruption class, where the implementation and the annotation are simultaneously weakened. We introduce an automated generate-verify pipeline that uses an LLM to mutate both objects, runs Dafny to confirm their verification, and retries with error feedback if failure occurs.

2 Related Work

2.1 Dafny

Dafny is a programming language that is designed to support deductive program verification: a way to ensure that a program is provably correct with respect to some specifications (Leino [2010]). Dafny programs contain *annotations*, which are constraints on the preconditions, postconditions, and loop invariants of a given function. Then, programs and the specifications are converted to verification conditions, which are checked using a satisfiability modulo theory (SMT) solver.

2.2 Clover

The Clover paradigm defines the correctness of a program in terms of the consistency of its components – the natural language docstring, the specification, and the code implementation. The intuition is that an incorrect program is unlikely to be simultaneously consistent with both a precise natural-language description and a set of formally verifiable annotations, so mutual consistency serves as a strong filter for incorrect code.

Consistency is checked across all six component pairs via six directional checks. Code–annotation consistency is split into soundness, where the Dafny verifier proves that the code satisfies its annotations, and completeness, where an LLM reconstructs code from the annotations and checks equivalence via unit-tests, preventing trivially weak annotations from being accepted. Annotation–docstring consistency is checked by reconstructing each artifact from the other using an LLM, with docstring equivalence judged by a second LLM call and annotation equivalence discharged as a formal Dafny lemma. Code–docstring consistency is established analogously via LLM-based reconstruction in both directions. A program is accepted only if all six checks pass.

3 Methodology

3.1 Setup

The Clover-provided datasets include an arrangement of data containing 62 different topics of programming concepts, each with a *spec*, *annotation*, and *solution*.

- The *spec*, which we referred to as a docstring, contains a short natural language instruction/description of a program that must be implemented.
- The *annotation*, which we referred to as the specification (spec) provides conditions for the program and specific constraints that accompany it.
- The *solution*, which we referred to as the implementation, is the code solution that follows the spec and annotation.

Clover introduced CloverBench — a dataset of annotated Dafny programs paired with natural-language specs, along with handwritten adversarial variants organized by corruption class. The classes show the different ways that we can create adversarial injections to these data-points, our adversarial examples falling under the C6 class.

3.2 LLM Mutation Generator-Verifier Loop

Our point of attack will be weakening the spec that accompanies the topic. We expect weakening the spec to pose the problem of allowing looser implementations to pass through the Clover detection module. We will then use the weakened spec to generate corresponding mutated implementations.

Table 1: CloverBench Corruption Classes and Components

Class	Code (Implementation)	Specification	Docstring
C1	Correct	Correct	Mutated
C2	Correct	Weakened	Correct
C3	Correct	Weakened	Mutated
C6	Mutated	Weakened	Correct

We send original specifications and implementations to Anthropic’s Claude Sonnet 4.5 model, which is instructed to subtly mutate the spec and implementation.

Then, Dafny is used to verify if the mutated implementation corresponds to the weakened spec. If the Dafny verification fails, then the LLM tries again up to five times, until it passes. If all five attempts fail, then that topic will be reported to not have viable mutations for that experiment cycle.

4 Results

Of the 62 programs, 23 mutated instances passed all Clover checks, yielding an overall evasion rate of 37.1%.

To better understand where Clover succeeds and fails, we categorized each mutation by the logical relationship between its spec and the original, as determined by Dafny. A **strictly-weakened (SW)** spec is one where the original logically implies the mutation but not vice versa. An **equivalent (EQ)** spec is one where Dafny proves a biconditional — the mutation is logically indistinguishable from the original despite syntactic differences. An **incomparable (IC)** spec is one where Dafny can establish neither relationship.

Across the 62 mutations, we found 15 SW, 30 EQ, and 17 IC specs. Clover caught all 15 SW mutations and 16 of the 17 IC mutations. The failure was concentrated almost entirely in the EQ category: Clover missed 22 of 30 equivalent-spec mutations, a 73.3% evasion rate. This indicates that when a weakened spec is logically indistinguishable from the original, Clover’s docstring-facing checks find nothing wrong, and the implementation flaw goes largely undetected.

Table 2: 2x2 Cross-Tabulation of Clover Evasion by Mutation Category

	Clover MISSED	Clover CAUGHT
Strictly weaker spec	0	15
Equivalent / other	23	24

4.1 Evasion Prediction

To quantify mutation amounts, we measured the normalized edit distance between the original and the mutated program, which is the character-level Levenshtein distance divided by the length of the longer string, giving us a score between 0 to 1 where lower values indicate subtler mutations. We find opposing trends: the evasion rate falls sharply with annotation change: mutations that changed the annotation by less than 5% evaded Dafny 52.6% of the time. However, for code changes, the pattern reverses: evasion rate rises from 20.0% for near-identical bodies to 58.8% for bodies with NED above 0.20.

Table 3: Dafny evasion rate for generated C6 mutations by annotation and body normalized edit distance. Fooled = Dafny incorrectly accepted the mutated program as correct.

NED Range	Foiled	Caught	Evasion Rate
<i>Annotation NED</i>			
[0.00, 0.05)	20	18	52.6%
[0.05, 0.10)	1	7	12.5%
[0.10, 0.20)	2	10	16.7%
[0.20, 1.00]	0	4	0.0%
<i>Body NED</i>			
[0.00, 0.05)	5	20	20.0%
[0.05, 0.10)	3	5	37.5%
[0.10, 0.20)	5	7	41.7%
[0.20, 1.00]	10	7	58.8%
Overall			37.1%

5 Discussion, Limitations, and Future Work

Our central finding is that Clover is vulnerable when a weakened specification remains logically equivalent to the original. In this regime, the docstring-facing checks find nothing wrong — the natural language still accurately describes what the spec says — and detection depends entirely on the implementation $\rightarrow$ specification reconstruction test. In this check, an LLM reads the mutated code and generates a specification from scratch; the bug surfaces only if the reconstruction happens to be tighter than the weakened spec, which it frequently is not.

This highlights a critical gap: currently, we cannot guarantee that LLM reconstruction of a flawed spec. can surface a bug when the spec. itself is not consistent with its initial implementation. It further adds that relying on a formal checker claiming a program is provably correct is not viable, especially if the spec. and docstring (natural language) seem consistent, but the spec. and implementation are not. The Evasion Prediction statistics back up this claim, especially since the implementations that have higher chances of bypassing Dafny are more modified from the original implementation.

A key limitation of this work is that both the mutation generation and Clover’s internal reconstruction checks rely on Claude Sonnet 4.5. A more capable model — such as Claude Opus 4 — might produce tighter reconstructions and catch inconsistencies that Sonnet 4.5 misses. Evaluating whether model capability closes this gap is an important direction for future work.

Additionally, future work should demonstrate how the adversarial examples generalize when scaled to harder problems. The problems in CloverBench are of introductory difficulty, so it would be interesting to see if the evasion patterns hold when faced with complex problems.

6 Conclusion

In this paper, we attempt to beat Clover’s LLM-based verification of specification and implementation. We tested the novel C6 category of adversarial data, leveraging LLMs to subtly mutate program specifications and their corresponding implementations. We paired Anthropic’s Claude 4.5 Sonnet with the Dafny formal verifier in a closed loop and successfully automated the generation of subtle, verified program mutations that break the original intent of the program while still satisfying a weakened specification.

Our evaluation across 62 benchmark tasks yielded an overall Clover evasion rate of 37.1%. Clover proved to be highly effective at catching strictly weaker or incomparable specification changes, but vulnerable to mutations of Dafny-verified logically equivalent specifications paired with corrupted code. That subsection

of mutations revealed a staggering 73.3% evasion rate. We also analyzed the normalized edit distance, which showed minor syntactic adjustments to formal specifications are adept at bypassing these consistency filters.

From these findings, we cannot purely rely on automated consistency checking, as formal verification cannot guarantee structural or intentional correctness if an LLM can systematically align a corrupted implementation with a modified specification. Future software verification methods require more resilient semantic cross-checks between natural language and formal code, particularly when scaling to complex, production-grade settings.

6.1 LLM Usage Statement

We used Claude as a research and coding assistant, and to write drafts for this paper.

References

- JetBrains. The state of developer ecosystem 2025. Industry survey report, JetBrains s.r.o., October 2025. URL <https://blog.jetbrains.com/research/2025/10/state-of-developer-ecosystem-2025/>. Survey conducted April–June 2025; 24,534 respondents from 194 countries.
- K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.
- Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot’s code contributions. *CoRR*, abs/2108.09293, 2021. URL <https://arxiv.org/abs/2108.09293>.
- Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation, 2024. URL <https://arxiv.org/abs/2310.17807>.

7 Appendix

7.1 Code and Links:

<https://github.com/lalitj5/formalization-proj/tree/main>

7.2 Prompts and Examples

Prompt for C6 Generation:

```
prompt = f"""
<role_assignment>
Trickster
</role_assignment>

<task>
You will produce a subtly corrupted Dafny program that:
1. Makes a small change to the implementation logic so it no longer satisfies the
   original spec.
2. Simultaneously weakens or adjusts the formal annotations (ensures/requires/
   invariants) just enough that Dafny still verifies the mutated code.
3. Keeps the change subtle enough that a human reviewer would not immediately
   notice the divergence from the spec.

The docstring/spec is NOT part of the Dafny file - do not add it. Return only the
Dafny method.
</task>

<original_spec>
{spec}
</original_spec>

<dafny_program>
{strong_dfy}
</dafny_program>
{feedback_block}
<return_instructions>
```

```
Return ONLY the modified Dafny method - no explanation, no markdown fences, no
commentary.
</return_instructions>
"""
```

Example of mutated spec vs initial spec:

```
mutated
method append(a:array<int>, b:int) returns (c:array<int>)
  ensures  a[..] + [b] == c[..] || c[..] == a[..] + [b+1]
{
  c := new int[a.Length+1];
  var i:= 0;
  while (i < a.Length)
    invariant 0 <= i <= a.Length
    invariant forall ii::0<=ii<i ==> c[ii]==a[ii]
  {
    c[i] := a[i];
    i:=i+1;
  }
  c[a.Length]:=b+1;
}

initial
method append(a:array<int>, b:int) returns (c:array<int>)
  ensures  a[..] + [b] == c[..]
{
  c := new int[a.Length+1];
  var i:= 0;
  while (i < a.Length)
    invariant 0 <= i <= a.Length
    invariant forall ii::0<=ii<i ==> c[ii]==a[ii]
  {
    c[i] := a[i];
    i:=i+1;
  }
  c[a.Length]:=b;
}
```

Note that the mutated spec contains an additional 'or' operator, which logically loosens the post-condition of the method.