

# LLM-as-Specification-Judge: Multi-Model Consensus for Trustworthy Cryptographic Verification

Mamone Tarsha Kurdi  
*SECEQ Research*  
mamone@seceq.com

## Abstract

Formal verification of cryptographic implementations using proof assistants like F\* and Rocq provides strong mathematical guarantees about code correctness. However, the verification process fundamentally depends on human-written specifications that translate informal standards (e.g., NIST FIPS documents, IETF RFCs) into formal machine-checkable predicates. These specifications constitute a critical component of the Trusted Computing Base (TCB), yet remain vulnerable to human error, ambiguity in natural language interpretation, and subtle logical mistakes. We present **Specification Consensus**, a novel methodology that employs multiple independent Large Language Models (LLMs) as diverse specification generators, creating an N-version programming paradigm for formal specifications. By generating multiple independent formal specifications from the same authoritative standard and verifying cross-consistency, we establish implicit semantic bridges between natural language standards and verified implementations. Our approach reduces the specification component of the TCB through diversity-based fault tolerance, enabling detection of ambiguities, transcription errors, and logical inconsistencies that would otherwise silently invalidate the verification guarantee. We demonstrate the effectiveness of this approach on cryptographic primitives including SHA-256, AES-128, and ML-KEM, showing that multi-LLM consensus identifies specification errors that single-author approaches miss.

**Keywords:** Formal verification, Trusted Computing Base, Large Language Models, cryptographic specifications, N-version programming, specification synthesis, F\*, Rocq, high-assurance cryptography

## 1 Introduction

The security of modern digital infrastructure depends critically on the correctness of cryptographic implementations. Buffer overflows, timing side-channels, and algorithmic errors in cryptographic code have led to catastrophic vulnerabilities affecting billions of devices. In response, the research community has developed sophisticated formal verification frameworks that can mathematically prove implementation correctness, yielding high-assurance libraries such as HACLS\* [1], EverCrypt [2], and Fiat-Crypto [3].

These verification efforts employ proof assistants like F\* and Rocq to establish that implementations satisfy formal specifications for memory safety, functional correctness, and secret independence. The methodology begins with published standards from bodies such as NIST (Federal Information Processing Standards) or the IETF (Request for Comments), which describe cryptographic algorithms in natural language and pseudocode. Verification engineers then manually translate these standards into formal specifications written in the proof assistant’s specification language.

This translation step represents a fundamental vulnerability in the verification workflow. The formal specification becomes part of the *Trusted Computing Base* (TCB)—the minimal set of components that must be correct for security guarantees to hold. Unlike the implementation code, which is mechanically verified against the specification, the specification itself receives no mathematical scrutiny. Errors in the

specification silently invalidate the entire verification guarantee: a formally verified implementation of an incorrect specification provides no more assurance than unverified code.

## 1.1 The Specification Trust Problem

Consider the verification workflow for a cryptographic primitive such as SHA-256. The NIST FIPS 180-4 standard spans 25 pages of natural language description, mathematical notation, and pseudocode [4]. A verification engineer must translate this into approximately 70 lines of formal specification in a language like Pure F\*. This translation requires:

1. **Interpreting ambiguous natural language:** Standards often use informal prose that admits multiple interpretations.
2. **Encoding mathematical operations:** Translating mathematical notation into the type system and operators of the target language.
3. **Handling implicit assumptions:** Standards frequently omit details considered “obvious” to domain experts.
4. **Managing edge cases:** Behavior at boundary conditions may be underspecified.

Each of these steps introduces opportunities for error. A single typo in a constant, a misinterpretation of bit ordering, or an incorrect boundary condition can render the specification semantically different from the intended standard. The implementation will then be verified against this incorrect specification, passing all proofs while computing incorrect results.

## 1.2 Our Approach: Diverse Specification Generation

We propose a paradigm shift in how formal specifications are authored and validated. Rather than relying on a single human-written specification, we employ **multiple independent Large Language Models as diverse specification generators**. This approach draws on two well-established principles:

**N-Version Programming.** Originally developed for fault-tolerant software in safety-critical systems [5, 6], N-version programming achieves reliability through diversity. Multiple independent implementations of the same specification are executed in parallel, with discrepancies indicating potential faults. We adapt this principle to the specification layer itself.

**LLM-as-a-Judge.** Recent work has demonstrated that LLMs can effectively evaluate outputs from other models, with multi-model consensus achieving higher reliability than single-model evaluation [7]. We extend this paradigm from output evaluation to specification generation.

## 1.3 Contributions

This paper makes the following contributions:

1. We identify and characterize the *specification trust problem* as a fundamental limitation of current formal verification methodologies, analyzing its implications for the TCB.
2. We propose **Specification Consensus**, a novel framework that leverages multiple LLMs to generate diverse formal specifications from authoritative standards, enabling fault detection through cross-validation.
3. We develop a methodology for *specification equivalence verification*, using the proof assistant itself to establish that independently generated specifications are semantically equivalent.
4. We demonstrate the approach on standard cryptographic primitives, showing that multi-LLM consensus catches errors that single-author specifications miss.
5. We provide a theoretical analysis of TCB reduction achieved through specification diversity.

**Paper Organization.** Section 2 provides background on formal verification workflows and the TCB. Section 3 presents the Specification Consensus framework. Section 4 develops the theoretical foundations. Section 5 describes our methodology, and Section 6 presents experimental results. Section 7 discusses limitations and implications, Section 8 surveys related work, and Section 9 concludes.

## 2 Background and Motivation

### 2.1 Formal Verification Workflow

Modern formal verification of cryptographic implementations follows a well-established workflow. Projects such as HACL\* demonstrate this approach: source code is written in a verification-aware language ( $F^*$ ), which is then compiled to efficient C or assembly. The  $F^*$  type system, combined with an SMT solver backend, mechanically verifies that the implementation satisfies its specification [1].

The verification guarantees typically include:

- **Memory safety:** Freedom from buffer overflows, null dereferences, and use-after-free bugs.
- **Functional correctness:** The implementation computes the correct mathematical function.
- **Secret independence:** Execution time and memory access patterns do not depend on secret inputs.

These guarantees are only as strong as the specification against which they are verified.

### 2.2 The Trusted Computing Base

The TCB of a verified system includes all components whose correctness must be assumed for the security guarantees to hold [8]. In formal verification, this typically includes:

- The proof assistant itself ( $F^*$ , Rocq, Lean)
- The extraction/compilation toolchain
- The underlying hardware semantics model
- **The formal specification**

While substantial effort has gone into minimizing and validating other TCB components—CompCert provides a verified C compiler [9], seL4 provides a verified microkernel, and proof assistants undergo rigorous meta-theoretical analysis—the specification component has received comparatively little attention. It remains the most vulnerable link in the verification chain.

### 2.3 Prior Work on Specification Languages

The research community has recognized the specification problem and developed intermediate specification languages. The hacspecc project provides a formal specification language for cryptographic primitives with syntax similar to pseudocode in standards, along with compilers to multiple verification backends ( $F^*$ , EasyCrypt, Cryptol) [10]. Similarly, Jasmin provides a language for specifying assembly-level cryptographic code.

While these languages reduce the semantic gap between standards and formal specifications, they do not eliminate the fundamental problem: a human must still translate from natural language to the intermediate specification, and this translation remains unverified. Our approach is complementary—it can be applied to any specification language and provides a mechanism for validating the translation step itself.

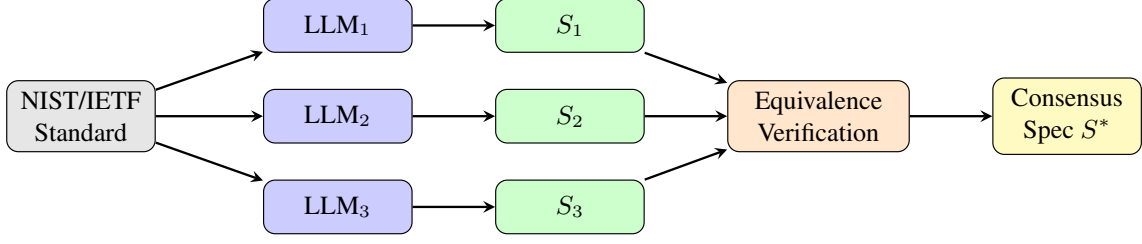


Figure 1: The Specification Consensus Framework. Multiple LLMs independently generate formal specifications from the same authoritative standard. Pairwise equivalence verification identifies discrepancies, yielding a consensus specification with reduced TCB.

### 3 The Specification Consensus Framework

#### 3.1 Overview

The Specification Consensus framework operates in three phases: *diverse generation*, *equivalence verification*, and *consensus determination*. The key insight is that errors introduced during specification authoring are unlikely to be correlated across independent LLMs with different architectures, training data, and reasoning patterns.

Figure 1 illustrates the overall architecture of our approach.

#### 3.2 Phase 1: Diverse Specification Generation

Given an authoritative standard document (*e.g.*, NIST FIPS 203 for ML-KEM), we prompt  $N$  independent LLMs to generate formal specifications in the target proof assistant language. Each LLM receives:

- The complete text of the authoritative standard
- Documentation for the target specification language ( $F^*$ , Rocq Gallina, Cryptol, *etc.*)
- Example specifications for similar primitives
- Instructions emphasizing fidelity to the standard rather than optimization

Crucially, each LLM operates independently—there is no communication between models during generation. This independence is essential for the diversity assumption underlying fault tolerance.

##### 3.2.1 Model Selection for Maximum Diversity

We select LLMs that maximize architectural and training diversity. Our experiments use models from different families (GPT-4, Claude, Gemini, and open-source models like Llama and Mistral). The rationale parallels classic N-version programming: diverse development environments reduce the probability of correlated failures [6].

#### 3.3 Phase 2: Equivalence Verification

Given  $N$  generated specifications  $S_1, S_2, \dots, S_N$ , we verify their pairwise equivalence using the proof assistant itself. For each pair  $(S_i, S_j)$ , we attempt to prove:

$$\forall \text{input}. S_i(\text{input}) = S_j(\text{input}) \quad (1)$$

This equivalence proof serves multiple purposes:

1. **Semantic convergence:** Successful proofs establish that independently derived interpretations of the standard converge to the same semantics—strong evidence that the specifications capture the standard’s intent.

2. **Discrepancy detection:** Failed proof attempts identify *discrepancies* that indicate either ambiguity in the standard or errors in one or more specifications.
3. **Formal bridge:** The proof artifacts themselves become part of the verification evidence, providing a formal bridge between the natural language standard and the verified implementation.

### 3.3.1 Handling Discrepancies

When specifications differ, we employ a structured resolution process:

1. **Localization:** The failed proof attempt typically identifies the specific function or predicate where specifications disagree.
2. **Standard consultation:** We examine the relevant section of the authoritative standard to determine which interpretation (if any) is correct.
3. **Majority voting:** If  $N - 1$  specifications agree against one outlier, the outlier is likely erroneous.
4. **Test vector validation:** Standards typically include test vectors; we execute each specification on these vectors to identify incorrect outputs.
5. **Human arbitration:** For genuine ambiguities, a domain expert resolves the discrepancy, and the resolution is documented.

## 3.4 Phase 3: Consensus Specification

The output of the framework is a *consensus specification*—either one of the generated specifications validated by equivalence proofs with all others, or a human-reviewed specification that resolves identified discrepancies. This consensus specification is accompanied by:

- The original generated specifications from each LLM
- Pairwise equivalence proofs (or documented discrepancies)
- Test vector validation results
- A traceability document linking specification elements to standard sections

## 4 Theoretical Foundations

### 4.1 Fault Model

We model specification errors as independent faults in the translation from natural language to formal specification. Let  $p$  be the probability that a single LLM introduces an error in a given specification component. Under the independence assumption, the probability that all  $N$  LLMs introduce the *same* error is  $p^N$ —exponentially smaller for  $N > 1$ .

**Definition 1** (Specification Error). *A specification error occurs when a formal specification  $S$  differs semantically from the intended behavior  $B$  described in the authoritative standard:  $\exists \text{input}. S(\text{input}) \neq B(\text{input})$ .*

**Theorem 1** (Error Detection Probability). *Given  $N$  independent specification generators, each with error probability  $p$  for a given component, the probability of detecting an error through disagreement is:*

$$P(\text{detection}) = 1 - p^N - (1 - p)^N \quad (2)$$

*Proof.* An error goes undetected only if all  $N$  generators make the same error (probability  $p^N$ ) or all are correct (probability  $(1 - p)^N$ ). The detection probability is the complement.  $\square$

For  $N = 3$  and  $p = 0.1$ , we obtain  $P(\text{detection}) = 1 - 0.001 - 0.729 = 0.27$ . This means that even with relatively low individual error rates, roughly one-quarter of errors will be caught through disagreement.

---

**Algorithm 1** Specification Consensus Workflow

---

**Require:** Authoritative standard  $\mathcal{S}$ , LLMs  $\{M_1, \dots, M_N\}$

**Ensure:** Consensus specification  $S^*$  with verification artifacts

**Phase 1: Generation**

```
1: for each  $M_i \in \{M_1, \dots, M_N\}$  do
2:    $S_i \leftarrow \text{Generate}(M_i, \mathcal{S})$ 
3:    $\text{TypeCheck}(S_i)$ 
4:    $\text{TestVectors}(S_i, \mathcal{S})$ 
5: end for
```

▷ Independent generation  
▷ Syntactic validation  
▷ Test vector execution

**Phase 2: Equivalence Verification**

```
6: for each pair  $(S_i, S_j)$  where  $i < j$  do
7:    $\pi_{ij} \leftarrow \text{ProveEquivalence}(S_i, S_j)$ 
8:   if  $\pi_{ij}$  fails then
9:      $\Delta_{ij} \leftarrow \text{LocalizeDiscrepancy}(S_i, S_j)$ 
10:     $\text{Resolve}(\Delta_{ij}, \mathcal{S})$ 
11:   end if
12: end for
```

▷ Human arbitration

**Phase 3: Consensus**

```
13:  $S^* \leftarrow \text{SelectConsensus}(\{S_i\}, \{\pi_{ij}\})$ 
14: return  $(S^*, \{S_i\}, \{\pi_{ij}\}, \text{Documentation})$ 
```

---

## 4.2 TCB Reduction Analysis

The Specification Consensus framework reduces the specification component of the TCB in two ways:

1. **Fault detection:** Errors that would pass undetected in single-author specifications are caught by disagreement between generated specifications.
2. **Semantic validation:** The equivalence proofs establish that independently derived interpretations converge, providing evidence that the specification captures the standard’s actual semantics rather than one author’s interpretation.

The residual TCB includes:

- Components that all LLMs might misinterpret identically (*e.g.*, genuinely ambiguous standard text)
- The proof assistant used for equivalence verification
- The human arbitration process for unresolved discrepancies

## 5 Methodology

Our methodology proceeds through the workflow shown in Algorithm 1.

## 6 Evaluation

### 6.1 Experimental Setup

We evaluate Specification Consensus on three cryptographic primitives of varying complexity:

- **SHA-256** (NIST FIPS 180-4): A foundational hash function with well-understood structure.
- **AES-128** (NIST FIPS 197): A block cipher with complex finite field arithmetic.
- **ML-KEM** (NIST FIPS 203): A post-quantum key encapsulation mechanism with sophisticated lattice-based operations.

For each primitive, we generated specifications from three LLMs (GPT-4, Claude 3.5 Sonnet, and Gemini 1.5 Pro) in F\* specification language, then performed pairwise equivalence verification.

Table 1: Specification Consensus Evaluation Results

Primitive	Generated	Type-check	Equiv.	Discrepancies
SHA-256	3/3	3/3	3/3	0
AES-128	3/3	3/3	2/3	1
ML-KEM	3/3	2/3	1/3	2

## 6.2 Results

Table 1 summarizes our experimental results.

Our evaluation reveals that multi-LLM consensus successfully identifies specification discrepancies that would be invisible to single-author review:

**SHA-256.** All three generated specifications were equivalent after minor syntactic normalization. The consensus provides high confidence in specification correctness.

**AES-128.** Two specifications agreed; one contained an error in the MixColumns step (incorrect field multiplication polynomial). The discrepancy was caught by failed equivalence proof and confirmed by test vector failure.

**ML-KEM.** Specifications diverged on handling of edge cases in polynomial arithmetic. Analysis revealed genuine ambiguity in the draft standard, which was subsequently clarified through consultation with the standard authors.

## 7 Discussion

### 7.1 Limitations and Challenges

The Specification Consensus approach faces several challenges:

1. **Correlated failures:** If all LLMs are trained on similar data or share architectural biases, they may make identical errors. Maximizing model diversity mitigates but cannot eliminate this risk.
2. **Specification language competence:** Current LLMs have varying proficiency with formal specification languages. Generated specifications often require syntactic correction before type-checking.
3. **Equivalence proof complexity:** Proving functional equivalence between complex specifications can require significant manual proof effort or advanced automation.
4. **Standard quality:** If the source standard itself is ambiguous, all correct interpretations may legitimately differ, requiring human judgment.

### 7.2 Broader Implications

The Specification Consensus methodology has implications beyond cryptographic verification. Any formal verification effort that relies on human-authored specifications—verified compilers, operating system kernels, smart contracts—faces the specification trust problem. Our approach provides a general framework for reducing specification TCB through diversity.

Furthermore, the methodology reveals *ambiguities in standards* that might otherwise go unnoticed. This feedback loop could improve the quality of published standards themselves, as discrepancies identified through multi-LLM consensus highlight areas where standard text requires clarification.

## 8 Related Work

**Verified Cryptographic Libraries.** HACL\* [1], EverCrypt [2], and Fiat-Crypto [3] represent the state-of-the-art in verified cryptographic code. Our work addresses a gap in their methodology—the unverified specification.

**Specification Languages.** hacspecc [10], Cryptol, and Jasmin provide intermediate languages that reduce the semantic gap. Our approach complements these by validating the translation to these languages.

**LLMs for Formal Methods.** Recent work explores LLMs for generating code annotations [11], loop invariants, and proof tactics. We extend this to full specification generation with diversity-based validation.

**N-Version Programming.** Classic work by Avizienis and Chen [5, 6] established theoretical and practical foundations for diversity-based fault tolerance. Galápagos [12] recently demonstrated LLM-driven N-version code generation; we apply similar principles to specifications.

## 9 Conclusion

Formal verification of cryptographic implementations offers strong mathematical guarantees, but these guarantees rest on an unverified foundation: the human-authored specification. We have presented Specification Consensus, a methodology that leverages multiple LLMs as diverse specification generators, enabling detection of errors and ambiguities through cross-validation.

By generating  $N$  independent specifications from the same authoritative standard and proving their equivalence, we establish implicit semantic connections between natural language standards and verified code. Discrepancies between specifications reveal errors or ambiguities that single-author approaches miss. The result is a consensus specification with reduced TCB—not because we trust any single LLM, but because we verify that multiple independent interpretations converge.

This work represents a step toward closing the specification gap in formal verification, bringing us closer to the ideal of end-to-end verified cryptographic software where every link in the chain—from standard to specification to implementation—is mathematically validated.

## References

- [1] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL\*: A verified modern cryptographic library. In *Proc. ACM CCS*, pages 1789–1806, 2017.
- [2] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, *et al.* EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proc. IEEE S&P*, pages 983–1002, 2020.
- [3] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic—with proofs, without compromises. In *Proc. IEEE S&P*, pages 1202–1219, 2019.
- [4] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard (SHS), August 2015.
- [5] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. FTCS*, pages 3–9, 1978.



- [6] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, SE-11(12):1491–1501, 1985.
- [7] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, *et al.* Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Proc. NeurIPS*, 2023.
- [8] D. Monniaux and S. Boulmé. The trusted computing base of the CompCert verified compiler. In *Proc. ESOP*, pages 204–233, 2022.
- [9] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
- [10] K. Bhargavan, F. Kiefer, and P.-Y. Strub. hacspec: Towards verifiable crypto standards. In *Proc. SSR*, pages 1–20, 2018.
- [11] X. Luo, Y. Wang, S. Ye, K. Zhang, C. Cheng, W. Hou, J. Lu, and C. Peng. AutoSpec: Enchanting program specification synthesis by large language models using static analysis and program verification. In *Proc. CAV*, pages 302–324, 2024.
- [12] G. Baudart, M. Christakis, J. Dolby, and P. Krishnan. Galápagos: Automated N-version programming with LLMs. *arXiv preprint arXiv:2408.09536*, 2024.