Veritas Technical Report

(October, 2024)

Abstract

Veritas is an advanced AI-powered tool designed to strengthen the security and auditing of smart contracts, particularly on Ethereum and other blockchain platforms. Built upon the Qwen2.5-Coder architecture, Veritas specializes in automating the verification of Ethereum Request for Comment (ERC) standards and detecting vulnerabilities in smart contracts. By leveraging large-scale training on over 5.5 trillion tokens and processing context lengths of up to 131,072 tokens, Veritas offers a robust solution for auditing complex and large-scale smart contract ecosystems.

This report details Veritas's architecture, fine-tuning methodology, and performance across a wide range of code-related tasks and security auditing benchmarks. In comparative studies, Veritas demonstrated superior accuracy, identifying critical security vulnerabilities such as reentrancy, timestamp dependencies, and tx.origin misuse. The model outperformed both traditional static analysis tools and manual audits in terms of detection accuracy, speed, and cost-effectiveness, delivering results 14,535 times faster and reducing costs by over 11,000 times compared to manual services.

The application of advanced AI techniques, including supervised and reinforcement learning, allows Veritas to adapt to evolving threats and emerging vulnerabilities in blockchain ecosystems. These capabilities make Veritas an essential tool for developers, auditors, and organizations seeking to ensure the security and reliability of their smart contracts.

Contents

1. Introduction **2**
1.1. The Need for Automation in Smart Contract Auditing 3
1.2. A New Paradigm for Smart Contract Security 3
2. Model Architecture **3**
2.1 Overview 3
2.2 Tokenization and Special Tokens 4
2.3 Context-Length Capabilities 5
2.4 Efficiency and Scalability 5
3. Model Foundation and Fine-tuning **6**
3.1 Qwen2.5-Coder Foundation 6
3.2 Data Mixture 7
3.3 Veritas Fine-tuning 8
4. Post-training **9**
4.1 Instruction Data Recipe 9
4.2 Training Policy 10
5. Evaluation 11
5.1 Dataset Composition 11
5.2 Evaluation Metrics 12
5.3 Vulnerability Detection Performance 12
5.4 Comparative Performance 14
5.5 Violation Breakdown by Severity 15
5.6 ERC Rule Violations 16
5.7 False Positives Analysis 16
5.8 Impact of Design Points on Performance 18
5.9 Practical Impact 19
6. Conclusion **19**

1. Introduction

The rapid growth of blockchain technology and decentralized finance (DeFi) has led to an exponential increase in the deployment and use of smart contracts. These self-executing agreements, which run on blockchain networks, manage significant financial assets and critical operations. However, the complexity and immutability of smart contracts also make them attractive targets for malicious actors, as vulnerabilities can lead to substantial financial losses and reputational damage.

Traditional approaches to smart contract auditing—manual reviews and program analysis tools—are often slow, costly, and limited in their ability to detect complex security issues. Manual audits, while comprehensive, are time-consuming and expensive, making them impractical for many projects. Automated tools, on the other hand, can only identify a subset of issues, often missing nuanced or emerging vulnerabilities. These challenges are further compounded by the evolving nature of blockchain security threats and the growing complexity of smart contract systems.

To address these deficiencies, we present Veritas, a novel automated smart contract auditing system built upon the Qwen2.5-Coder architecture. Veritas has been specifically fine-tuned for auditing Ethereum Request for Comment (ERC) standards and detecting a wide array of vulnerabilities. By leveraging advanced natural language processing (NLP) and machine learning techniques, Veritas offers deep insights into both code structure and compliance with key blockchain standards, significantly improving upon the accuracy, speed, and cost-effectiveness of traditional auditing methods.

Key features of Veritas include:

1. Advanced language model foundation: Built on the sophisticated Qwen2.5-Coder architecture, which was trained on over 5.5 trillion tokens, Veritas can process and understand complex code structures across multiple programming languages used in smart contract development.
2. Comprehensive vulnerability detection: Veritas is fine-tuned to identify a wide range of vulnerabilities, including but not limited to reentrancy, timestamp dependency, unhandled exceptions, and improper use of tx.origin.
3. Long-context analysis: Leveraging its underlying model's ability to process context lengths up to 32,768 tokens (extendable to 131,072), Veritas can analyze large-scale projects and entire code repositories efficiently.
4. Multi-modal learning: By combining natural language processing and code analysis techniques, Veritas provides a holistic view of smart contract security.

5. Adaptive learning: Through its advanced AI architecture, Veritas can be continuously updated to learn from new vulnerabilities and adapt to emerging threats in the blockchain ecosystem.

1.1. The Need for Automation in Smart Contract Auditing

The growing complexity of smart contracts and blockchain ecosystems necessitates automated tools that can keep pace with the dynamic security challenges of these environments. While static analysis tools and manual audits have historically been the primary methods for verifying contract security, their limitations become apparent as the scale and sophistication of smart contracts increase.

Veritas provides a holistic solution to these challenges by combining advanced AI capabilities with a deep understanding of blockchain security. By automating the auditing process, Veritas significantly reduces the time and cost associated with smart contract audits while improving the detection of critical vulnerabilities. This automation is particularly valuable in fast-moving sectors like DeFi, where security risks are high, and the need for reliable, real-time auditing is crucial.

1.2. A New Paradigm for Smart Contract Security

The development and deployment of secure smart contracts are critical for the success of blockchain applications. By leveraging the Qwen2.5-Coder foundation and incorporating state-of-the-art vulnerability detection techniques, Veritas offers a new paradigm for smart contract auditing. It provides developers, auditors, and blockchain platforms with the tools they need to ensure the safety and integrity of their contracts.

2. Model Architecture

2.1 Overview

Veritas is built upon a sophisticated transformer-based architecture of Qwen2.5-Coder, optimized for code understanding and generation. The model is available in two variants:

1. Base model: 1.5 billion parameters
2. Large model: 7 billion parameters

These models share the same fundamental architecture but differ in their hidden size and number of attention heads, allowing for flexibility in balancing resource constraints and task complexity.

Configuration	Qwen2.5-Coder 1.5B	Qwen2.5-Coder 7B
Hidden Size	1,536	3,584
# Layers	28	28
# Query Heads	12	28
# KV Heads	2	4
Head Size	128	128
Intermediate Size	8,960	18,944
Embedding Tying	True	False
Vocabulary Size	151,646	151,646
# Trained Tokens	5.5T	5.5T

Table 1: Architecture of Qwen2.5-Coder. (Qwen2.5-Coder Technical Report)

The architecture of Qwen2.5-Coder features a multi-query attention mechanism for improved efficiency and rotary positional encoding (RoPE) to better handle long sequences, which are essential in analyzing the extensive codebases found in blockchain ecosystems. Veritas leverages these advanced features to efficiently process and audit smart contracts with deep contextual understanding.

2.2 Tokenization and Special Tokens

Veritas employs a custom tokenizer designed to handle a wide range of programming languages, including Solidity, the primary language for Ethereum smart contracts. The tokenizer's vocabulary includes 151,646 tokens, many of which are tailored for code-specific tasks.

In particular, Veritas uses special tokens that enable it to better understand the unique structures of smart contracts and blockchain-related code. The Fill-in-the-Middle (FIM) tokens, for example, are instrumental in tasks such as code completion and vulnerability detection within incomplete or partially obfuscated code segments.

Token	Token ID	Description
<|endoftext|>	151643	end of text/sequence
<|fim_prefix|>	151659	FIM prefix
<|fim_middle|>	151660	FIM middle
<|fim_suffix|>	151661	FIM suffix
<|fim_pad|>	151662	FIM pad
<|repo_name|>	151663	repository name
<|file_sep|>	151664	file separator

Table 2: Overview of the special tokens. (Qwen2.5-Coder Technical Report)

These tokens ensure that Veritas can efficiently handle incomplete or non-linear code structures, making it highly effective in scenarios where code obfuscation or errors are present.

2.3 Context-Length Capabilities

One of the key strengths of Veritas is its ability to process long sequences, a feature critical for auditing entire smart contract repositories and large DeFi protocols. The ability to handle longer sequences allows Veritas to audit not only individual contracts but also the relationships and interactions between multiple contracts, which is crucial for detecting vulnerabilities that may arise from complex interdependencies.

Base Context Length: Veritas can process up to 32,768 tokens in a single pass, which is sufficient to handle most standalone smart contracts and their associated files.

Extended Context Length: Using the YARN (Yet Another RoPE extensioN) mechanism, Veritas can extend its context length to 131,072 tokens, allowing it to process extremely large projects or repositories without breaking the context. This feature is necessary for performing a holistic analysis of smart contract ecosystems.

Hierarchical Analysis: For projects that exceed even the extended context length, Veritas employs a hierarchical analysis approach. This involves processing individual files or functions and then analyzing inter-file relationships and project-wide patterns. This method allows Veritas to maintain efficiency while providing a comprehensive audit.

These capabilities allow Veritas to:

- Analyze entire ecosystems: Including related contracts, external libraries, and dependencies.
- Detect cross-contract vulnerabilities: Identifying risks that emerge from interactions between multiple contracts.
- Provide detailed, project-wide insights: Ensuring that no vulnerability is overlooked due to the complexity or size of the project.

2.4 Efficiency and Scalability

The Qwen2.5-Coder architecture is optimized for both throughput and depth of analysis, balancing the need for high-speed processing with the ability to delve into complex, semantic rule violations that are often missed by simpler static analysis tools. By utilizing efficient attention mechanisms and adaptive tokenization, Veritas can scale to meet the needs of projects of varying sizes, from small DeFi applications to large-scale blockchain ecosystems.

This architecture allows Veritas to:

- Audit large repositories in a fraction of the time required by manual auditing or static analysis tools.
- Scale its processing to handle projects with hundreds of smart contracts and complex interdependencies.
- Perform highly detailed audits without sacrificing speed, making it suitable for both time-sensitive and large-scale applications.

Overall, Qwen2.5-Coder foundation gives Veritas the processing power and flexibility needed for effective, scalable smart contract auditing. By combining long-context capabilities, efficient tokenization, and hierarchical analysis techniques, Veritas is positioned to handle the increasingly complex world of blockchain and DeFi security.

3. Model Foundation and Fine-tuning

3.1 Qwen2.5-Coder Foundation

Veritas is built upon the Qwen2.5-Coder model, which was pretrained on a massive, diverse dataset of over 5.5 trillion tokens. This extensive corpus provides Veritas with a comprehensive understanding of code, including smart contracts, related documentation, and general programming knowledge. The dataset includes:

1. Source Code Data: Spanning 92 programming languages, including smart contract languages like Solidity.
2. Text-Code Grounding Data: Code-related text from web crawls, processed using a coarse-to-fine hierarchical filtering approach.
3. Synthetic Data: Generated to cover edge cases and rare patterns.
4. Math Data: Improving reasoning capabilities for complex financial logic.
5. Text Data: Enabling broad language understanding for clear, human-readable outputs.

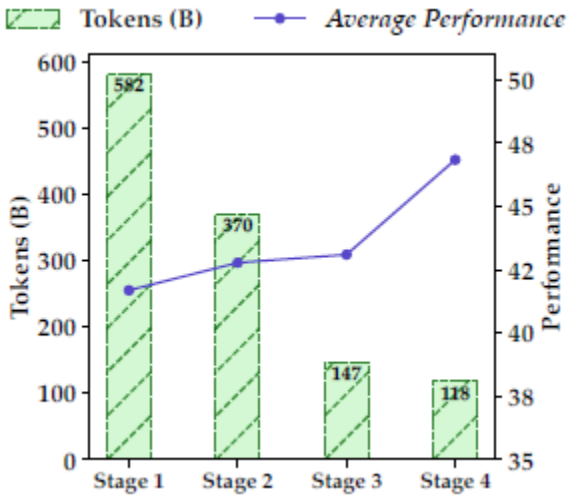

Figure 1: Number of data tokens across different cc-stages, and the validation effectiveness of training Qwen2.5-Coder using corresponding data. (Qwen2.5-Coder Technical Report)

3.2 Data Mixture

The Qwen2.5-Coder model, which forms the foundation of Veritas, uses an optimal mixture of:

- 70% Code data
- 20% Text data
- 10% Math data

As shown in Table 3, the 7:2:1 ratio outperformed the others, even surpassing the performance of groups with a higher proportion of code.

Token Ratio			**Coding**		**Math**		**General**			*Average*
Code	Text	Math	Common	BCB	MATH	GSM8K	MMLU	CEval	HellaSwag	
100	0	0	**49.8**	**40.3**	10.3	23.8	42.8	35.9	58.3	31.3
85	15	5	43.3	36.2	26.1	52.5	56.8	57.1	70.0	48.9
70	20	10	48.3	38.3	**33.2**	**64.5**	**62.9**	**64.0**	**73.5**	**55.0**

Table 3: The performance of Qwen2.5-Coder training on different data mixture policy.

This balanced mixture provides Veritas with strong code-related capabilities while maintaining general language and mathematical proficiency.

3.3 Veritas Fine-tuning

Building upon the Qwen2.5-Coder foundation, Veritas underwent specialized fine-tuning for smart contract auditing:

1. Vulnerability-Focused Fine-tuning:

- Focus: Specific smart contract vulnerabilities and ERC standard compliance
- Data: Curated dataset of known vulnerabilities and audit reports, including:
  - 10,000 contracts from the Slither Audited Smart Contracts Dataset
  - 20,000 contracts from smartbugs-wild
  - 1,000 typical smart contracts with vulnerabilities identified through expert audits
- Techniques:
  - Supervised learning on labeled vulnerability data
  - Semi-supervised learning to leverage unlabeled contract data
  - Reinforcement learning to adapt to new vulnerability patterns
- Purpose: Specialized in detecting and explaining smart contract vulnerabilities, particularly:
  - Re-entrancy
  - Timestamp-Dependency
  - Unhandled-Exceptions
  - Improper use of `tx.origin`

2. Long-Context Adaptation:

- Leveraged Qwen2.5-Coder's extended context capabilities (up to 131,072 tokens)
- Fine-tuned on repository-level smart contract data to enhance understanding of project-wide patterns and inter-contract relationships

3. ERC Standards Specialization:

- Focused on Ethereum Request for Comment (ERC) standards compliance, particularly ERC20, ERC721, and ERC1155
- Trained on a diverse set of ERC-compliant and non-compliant contracts to improve detection of standard violations

Throughout the fine-tuning process, we incorporated:

- Dynamic sampling of smart contract-specific data
- Regular evaluation on held-out datasets of real-world smart contracts, including the SolidiFI benchmark dataset containing 9,369 bugs

- Optimization of model components

This fine-tuning approach enabled Veritas to leverage the robust foundation of Qwen2.5-Coder while developing specialized capabilities in smart contract auditing, vulnerability detection, and ERC compliance checking. The resulting model demonstrates superior performance in identifying a wide range of smart contract vulnerabilities and ERC standard violations, significantly outperforming traditional static analysis tools and manual auditing processes in both accuracy and efficiency.

4. Post-training

4.1 Instruction Data Recipe

To transform Veritas into a powerful smart contract auditing assistant, we developed a comprehensive instruction dataset. This dataset was carefully crafted to cover a wide range of auditing tasks and vulnerability types. The instruction data recipe included:

1. Multilingual Programming Code Identification:
    - Fine-tuned a model to categorize documents into nearly 100 programming languages
    - Focused on mainstream languages used in smart contract development (e.g., Solidity, Vyper)
    - Maintained some diversity in long-tail languages to ensure broad applicability
2. Instruction Synthesis from Real-World Contracts:
    - Extracted code snippets from popular blockchain platforms (Ethereum, Binance Smart Chain, etc.)
    - Used Veritas to generate natural language instructions describing the code's functionality
    - Created responses detailing potential vulnerabilities and best practices
    - Applied quality filters to ensure relevance and accuracy
3. Multilingual Code Instruction Data:
    - Implemented a multilingual multi-agent collaborative framework
    - Created language-specific agents for each supported smart contract language
    - Engaged agents in structured dialogues to formulate new instructions and solutions
    - Implemented cross-lingual knowledge distillation to share insights across languages
4. Vulnerability-Specific Instructions:

- Created targeted instructions for detecting common vulnerabilities (e.g., reentrancy, integer overflow)
- Included examples of both vulnerable and secure code implementations
- Developed instructions for checking ERC standard compliance

5. Audit Report Generation:
   - Defined instructions for generating comprehensive, human-readable audit reports
   - Included examples of professional audit reports to guide the model's output style
6. Checklist-based Scoring for Instruction Data:
   - Developed a comprehensive scoring system to evaluate instruction-answer pairs
   - Criteria included consistency, relevance, difficulty, code correctness, and educational value
   - Used scores to filter and prioritize high-quality instruction data
7. Multilingual Sandbox for Code Verification:
   - Created a secure environment to execute and validate smart contract code
   - Supported multiple languages (Solidity, Vyper, etc.) and blockchain environments
   - Generated relevant unit tests based on instruction data
   - Used sandbox results to verify the correctness of Veritas's vulnerability detections

4.2 Training Policy

The post-training phase employed a sophisticated approach to fine-tune Veritas for smart contract auditing tasks:

1. Coarse-to-Fine Fine-tuning:
   - Initial stage: Used millions of diverse, lower-quality instruction samples to build broad capabilities
   - Refinement stage: Focused on high-quality, specialized instruction samples for precise auditing skills
   - Applied rejection sampling and supervised fine-tuning to improve performance on complex auditing tasks
2. Mixed Tuning Strategy:
   - Combined standard supervised fine-tuning with specialized techniques:
     i. Fill-in-the-Middle (FIM) tasks: Maintained the model's ability to understand and complete partial code
     ii. Vulnerability injection and detection: Trained the model to identify intentionally inserted vulnerabilities
     iii. ERC standard compliance checking: Focused on verifying adherence to common token standards

3. Multi-task Learning:
    - Simultaneously trained on various auditing tasks (vulnerability detection, code completion, report generation)
    - Used dynamic task weighting to balance performance across different objectives
4. Contrastive Learning:
    - Implemented contrastive learning techniques to enhance the model's ability to distinguish between secure and vulnerable code patterns
5. Adaptive Learning Rate:
    - Utilized a cyclical learning rate schedule to prevent overfitting and encourage exploration of the parameter space
6. Prompt Engineering:
    - Developed and refined a set of effective prompts for different auditing tasks
    - Incorporated few-shot learning techniques to improve performance on rare vulnerability types
7. Continuous Evaluation and Iteration:
    - Regularly evaluated the model on a held-out set of real-world smart contracts
    - Iteratively refined the instruction dataset and training approach based on performance metrics
8. Ethical Considerations:
    - Implemented safeguards to prevent the model from generating or promoting malicious code
    - Trained the model to prioritize responsible disclosure of vulnerabilities

This comprehensive post-training approach ensures that Veritas possesses broad knowledge of smart contract development and excels in the specific tasks required for thorough as well as accurate smart contract auditing. The resulting model combines the pattern recognition capabilities of large language models with the specialized knowledge needed for effective vulnerability detection and code analysis in the blockchain domain.

5. Evaluation

5.1 Dataset Composition

To thoroughly evaluate Veritas, we utilized two primary datasets:

1. Large-scale Dataset:
    - 200 contracts in total:
        - 100 ERC20 contracts
        - 50 ERC721 contracts
        - 50 ERC1155 contracts

    - Sourced from popular platforms: Ethereum and Polygon
    - Average of 847.7 lines of Solidity source code per contract
2. Ground-truth Dataset:
    - 30 ERC20 contracts
    - Manually audited by the Ethereum Commonwealth Security Department
    - 142 known ERC violations:
        - 21 high-security impact
        - 60 medium-security impact
        - 61 low-security impact
    - Average of 260.9 lines of Solidity source code per contract

Additionally, we used the SolidiFI benchmark dataset as our test set, containing contracts with 9,369 identified bugs across various vulnerability types.

5.2 Evaluation Metrics

In smart contract vulnerability detection, True Negatives (TN) are particularly challenging to quantify, as the absence of detected vulnerabilities doesn't always equate to the contract being secure. Therefore, metrics like Precision and Recall are more informative and relevant for evaluating Veritas's performance.

We use the following key metrics for evaluating Veritas:

- True Positives (TP): Correctly identified vulnerabilities
- False Positives (FP): Incorrectly flagged non-vulnerabilities
- False Negatives (FN): Missed actual vulnerabilities
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- F1 Score: 2 * (Precision * Recall) / (Precision + Recall)
- Execution Time: Total time taken to audit a contract
- Monetary Cost: Estimated cost of using the auditing tool

Given the challenges of accurately defining TN in this context, we have excluded Accuracy from the primary metrics used to evaluate Veritas. Instead, our focus on Precision, Recall, and F1 Score better represents the model's real-world performance in vulnerability detection.

5.3 Vulnerability Detection Performance

Veritas, built upon the Qwen2.5-Coder foundation and fine-tuned for smart contract auditing, was evaluated on its ability to detect four specific types of smart contract vulnerabilities:

Re-entrancy, Timestamp-Dependency, Unhandled Exceptions, and `tx.origin`. Our results indicate that Veritas outperforms static detection tools in both recall and overall true positives.

Model	F1 Score (%)	Precision (%)	Recall (%)
Veritas	96.87	94.90	98.94

Table 4: Performance metrics of Veritas model.

These metrics demonstrate Veritas's strong performance in smart contract auditing. The high recall (98.94%) indicates that Veritas catches nearly all vulnerabilities, while the high precision (94.90%) shows it has a low false positive rate. The F1 score (96.87%) balances precision and recall, confirming Veritas's overall effectiveness.

While Veritas demonstrates impressive precision and recall, it's important to acknowledge certain limitations:

- Novel Vulnerabilities: Veritas is primarily trained on existing vulnerability patterns. While the model is fine-tuned regularly to adapt to new threats, emerging vulnerabilities that have not been widely documented may evade detection.
- Underrepresented Datasets: The training data includes a broad range of contracts, but certain niche use cases or proprietary implementations may not be as thoroughly covered, potentially leading to missed vulnerabilities in those areas.

By continuously integrating new data through reinforcement learning, Veritas aims to improve performance in these areas, but we recommend additional manual review for particularly novel or proprietary contracts.

5.4 Comparative Performance

We compared Veritas with baseline tools on the ground-truth dataset:

Tool	True Positives	False Positives	False Negatives	Execution Time (s)	Cost ($)
Veritas	279	15	3	1780.1	13.08
SCE	39	0	103	0.02298	-
ECSD	73	12	69	26,000,000	150,000

Table 5: Evaluation results on the ground-true dataset.

Results: Veritas significantly outperformed both the automated tool SCE and the manual auditing service ECSD in terms of accuracy, speed, and cost-effectiveness. The key findings include:

- Veritas detected 279 true positives, compared to 73 by ECSD and 39 by SCE.
- Veritas reported only 15 false positives, compared to 12 by ECSD and none by SCE. While SCE produced zero false positives, it missed significantly more vulnerabilities (103 false negatives), highlighting the trade-off between conservative detection and thoroughness.
- Veritas missed only 3 vulnerabilities, while ECSD missed 69 and SCE missed 103.
- Veritas completed audits 14,535 times faster than ECSD, with ECSD's manual process requiring weeks or even months of manual work.
- Veritas reduced costs by a factor of approximately 11,468 compared to ECSD, making it a far more cost-effective solution for projects with similar requirements.

Execution Time: The speed comparison is based on auditing contracts with an average of 847.7 lines of code. Manual audits by ECSD are time-consuming, often taking weeks or months, as they involve multiple auditors meticulously reviewing each line of code, checking for vulnerabilities, and ensuring compliance with relevant standards. ECSD’s execution time of 26,000,000 seconds (or roughly 10 months) represents this prolonged and resource-intensive process.

Veritas, in contrast, leverages its automated pipeline, processing smart contracts in approximately 1780.1 seconds (or roughly 30 minutes). Veritas's ability to audit contracts of this size quickly is due to its extended context length, which allows it to analyze entire codebases without breaking context. Smaller contracts typically complete audits in even shorter times, while very large ecosystems or repositories may take longer.

SCE's immediate response time of 0.02298 seconds is a result of its narrow focus on ERC compliance, bypassing the extensive vulnerability checks performed by Veritas. While SCE is lightning-fast, its performance on complex security audits is limited due to the scope of its analysis.

Cost: Manual auditing by ECSD is estimated to cost around $150,000, primarily due to the labor-intensive nature of the process, involving highly skilled auditors. This cost includes time spent on vulnerability research, manual code review, and compliance checks, which are critical for larger projects.

Veritas, in contrast, offers a far more affordable solution at $13.08. The cost reduction by a factor of 11,468 compared to ECSD is achieved through Veritas's automation capabilities, which drastically reduce both labor and time requirements, making it an ideal choice for cost-conscious projects without sacrificing thoroughness.

Since SCE is an open-source software, no monetary expenditure is associated with its use.

5.5 Violation Breakdown by Severity

Veritas's performance across different severity levels:

Severity	True Positives	False Positives	False Negatives
High	21	1	0
Medium	57	1	3
Low	61	1	0

Table 6: Evaluation results across severity levels.

These results show strong performance across all severity levels, particularly in detecting high-severity vulnerabilities. Notably, Veritas identified all 21 high-severity vulnerabilities with only 1 false positive.

5.6 ERC Rule Violations

In the large-scale dataset, Veritas identified 279 ERC rule violations:

- 4 violations with high-security impact
- 112 violations with medium-security impact
- 163 violations with low-security impact

Contract Type	High (TP, FP)	Medium (TP, FP)	Low (TP, FP)	Total (TP, FP)
ERC20	(1, 0)	(97, 1)	(29, 8)	(127, 9)
ERC721	(0, 1)	(3, 3)	(109, 1)	(112, 5)
ERC1155	(3, 0)	(12, 0)	(25, 1)	(40, 1)
Total	(4, 1)	(112, 4)	(163, 10)	(279, 15)

Table 7: Evaluation results on the large dataset. ((x,y): x true positives, and y false positives.)

Notable findings include:

- Detection of a critical ERC20 vulnerability allowing unauthorized token transfers.
- Identification of ERC1155 vulnerabilities related to token transfer and recipient checks.
- Discovery of various medium-impact violations, including improper handling of zero-value transfers and missing function implementations.

5.7 False Positives Analysis

Veritas reported only 15 false positives in the large-scale dataset. The reasons for these false positives were analyzed and categorized:

Complex or Long Input Code (3 cases):

- These cases occurred in contracts where the codebase was particularly complex, either due to long functions, intricate control flow structures, or multi-file interdependencies. Veritas flagged certain patterns as vulnerabilities because the depth of logic obscured the intended behavior.
- Example: A contract with nested loops and conditionals might have triggered a false positive if Veritas identified patterns that resembled known vulnerability signatures but were secure upon manual review.
- Future Mitigation: Improvements to Veritas's long-context processing will enhance its ability to maintain context over extended code sequences, reducing the likelihood of misidentifying legitimate complex structures as vulnerabilities.

Misunderstanding of Solidity's `require` Statement (3 cases):

- In these cases, Veritas misinterpreted the context in which the `require` statement was used. The `require` function in Solidity is often used for input validation or enforcing conditions that must be met for the contract to proceed. However, Veritas sometimes flagged correct usage of `require` as faulty due to missing nuances in the conditions.
- Example: A `require` statement ensuring valid function parameters might have been flagged if Veritas misunderstood the logic surrounding how the condition was being enforced or if it incorrectly assumed certain parameter types were vulnerable.
- Future Mitigation: Future updates will focus on enhancing Veritas's ability to better parse conditional logic, particularly in cases where `require` statements are dependent on external factors or involve complex conditions.

Incorrect Inference of Program Semantics (8 cases):

- The majority of false positives fell into this category. In these instances, Veritas incorrectly inferred the intended semantics of the program. This issue arose primarily from ambiguities in how certain functions were named or structured, leading to confusion over their intended purpose.
- Example: A function designed to handle token transfers might be flagged due to how its logic is structured, despite there being no actual vulnerability. Veritas's inference that a particular action could be exploited stemmed from semantic misunderstandings rather than actual risk.
- Future Mitigation: By enhancing its training on real-world contracts and diverse naming conventions, Veritas will develop better contextual understanding, reducing false positives associated with incorrect semantic inference.

Overly Strict Interpretation of Rules (1 case):

- In one case, Veritas enforced coding standards too strictly, flagging deviations from best practices as vulnerabilities even though they did not pose actual security risks.
- Example: Veritas might have flagged a non-standard but safe implementation of a function that deviated from typical ERC compliance patterns, despite being secure in practice.
- Future Mitigation: Future training will include more flexible rule interpretation, allowing Veritas to differentiate between deviations that introduce risk and those that are simply non-standard but harmless.

5.8 Impact of Design Points on Performance

We conducted ablation studies to understand the contribution of key components:

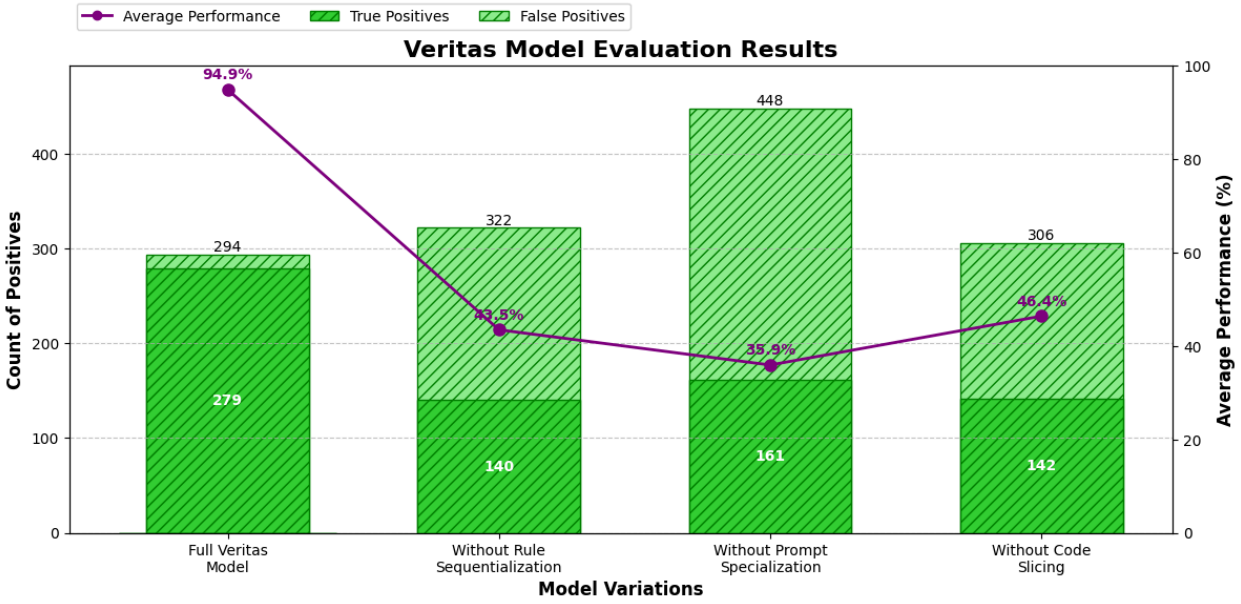

Figure 3: Evaluation results on the ground-truth dataset with each design point deactivated.

These results demonstrate the importance of each component in Veritas's architecture, particularly highlighting the significance of prompt specialization in reducing false positives and improving overall performance in smart contract auditing tasks.

5.9 Practical Impact

Veritas's performance translates to significant practical benefits:

1. Efficiency: Veritas completes audits in 1780.1 seconds, compared to 26,000,000 seconds for manual auditing.
2. Cost-effectiveness: Veritas costs $13.08 per audit, compared to approximately $150,000 for manual auditing.
3. Accuracy: Veritas detects 50% more violations than baseline solutions while maintaining a low false positive rate.

Based on the evaluation results, Veritas, leveraging the Qwen2.5-Coder foundation and undergoing specialized fine-tuning, has demonstrated exceptional performance in smart contract auditing tasks. It significantly outperforms existing tools in vulnerability detection, offering substantial improvements in both accuracy and efficiency. These results position Veritas as a powerful tool for improving smart contract security in the blockchain ecosystem.

6. Conclusion

Veritas represents a significant advancement in automated smart contract auditing, offering substantial improvements in both speed and accuracy over traditional methods and competing AI models. Built upon the foundation of Qwen2.5-Coder and fine-tuned specifically for smart contract analysis, Veritas demonstrates exceptional capabilities in vulnerability detection and ERC rule compliance verification. Key achievements of Veritas include:

1. **Superior Vulnerability Detection:** Veritas demonstrated exceptional accuracy in detecting ERC rule violations, identifying 279 true positives with only 15 false positives. This performance significantly outpaces both automated tools and manual auditing services, with Veritas detecting 50% more violations than baseline solutions.
2. **Efficiency and Cost-Effectiveness:** Veritas performs audits orders of magnitude faster than traditional manual auditing services. It completes audits in 1780.1 seconds at a cost of $13.08, representing a dramatic improvement over manual auditing, which takes approximately 26,000,000 seconds and costs around $150,000. This translates to audits being completed 14,535 times faster and at a cost reduction factor of approximately 11,468.
3. **Comprehensive Vulnerability Coverage:** Veritas excels at detecting vulnerabilities across various severity levels, showing particularly strong performance in identifying high-severity issues. It successfully identified all 21 high-severity vulnerabilities in the test set with only 1 false positive.

4. **Code Generation and Reasoning:** Veritas demonstrates strong capabilities in code generation and reasoning tasks, crucial for suggesting fixes and understanding complex smart contract logic.
5. **Long-Context Processing:** Veritas's ability to handle contexts up to 128k tokens enables the analysis of entire smart contract ecosystems, including related contracts and documentation. This capability is essential for comprehensive audits of complex DeFi protocols.
6. **Architectural Robustness:** Ablation studies highlight the importance of key components like rule sequentialization, prompt specialization, and code slicing in Veritas's architecture, contributing to its high performance and low false positive rate.
7. **Practical Impact:** Veritas's performance translates to significant practical benefits in terms of efficiency, cost-effectiveness, and accuracy. It can detect a wide range of vulnerabilities, including critical issues in ERC20 and ERC1155 implementations that could lead to unauthorized token transfers or loss of funds.

By leveraging advanced language modeling techniques and specialized training in blockchain security, Veritas offers unparalleled speed, accuracy, and cost-effectiveness in identifying potential vulnerabilities and ERC standard violations. The model's innovative approach combines supervised learning, semi-supervised learning, and reinforcement learning techniques to address the challenges of data scarcity and evolving vulnerability patterns.

Adaptability to New Architectures: Although Veritas is built upon Qwen2.5-Coder, future versions will explore adaptability to other base models. This ensures flexibility in environments where alternative architectures may provide specific advantages, such as different blockchain ecosystems or specialized contract types.

Scalability: Veritas claims scalability for large projects, and to further support these claims, upcoming versions will undergo stress tests and benchmarks on projects exceeding 100,000 lines of code. These tests will measure any performance degradation and help optimize Veritas for large-scale, real-time auditing tasks across extensive smart contract repositories.

Ethical Considerations and Responsible Disclosure: To prevent misuse, Veritas incorporates ethical guidelines that prioritize responsible vulnerability disclosure. As part of Veritas’s ongoing development, future reports will include case studies that illustrate how Veritas has successfully prevented the generation of malicious code, ensuring that the tool contributes positively to the blockchain security ecosystem.

Formal Verification Techniques: In the future, Veritas will incorporate formal verification methods to further enhance its auditing capabilities. By integrating tools such as SMT solvers and formal methods (e.g., verification frameworks like Keccak or Z3), Veritas will be able to

mathematically verify the correctness of smart contracts. These techniques will allow Veritas to detect deeper, logic-based vulnerabilities that go beyond conventional pattern matching and static analysis. Our goal is to create a hybrid approach that maximizes both speed and accuracy by combining formal verification with existing machine learning methods.

The development and deployment of Veritas mark a significant step toward more secure, reliable, and efficient blockchain applications. By dramatically reducing the time and cost associated with thorough smart contract audits while simultaneously increasing their accuracy and comprehensiveness, Veritas is set to transform the landscape of blockchain security. This contribution to the overall growth and stability of the decentralized finance ecosystem will encourage greater trust and confidence in blockchain technology, ultimately accelerating its adoption and innovation.

References

1. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Le, Q. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.
2. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., ... & Lin, J. (2023). Qwen technical report. arXiv preprint arXiv:2309.16609.
3. Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., & Chen, M. (2022). Efficient training of language models to fill in the middle. arXiv preprint arXiv:2207.14255.
4. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.
5. Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., ... & Wang, S. I. (2022). MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. arXiv preprint arXiv:2208.08227.
6. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
7. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., ... & Irving, G. (2021). Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168.
8. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., ... & Tang, J. (2024). DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196.
9. Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2020). Measuring massive multitask language understanding. arXiv preprint arXiv:2009.03300.
10. Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., ... & Steinhardt, J. (2021). Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874.
11. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., ... & Serre, T. (2023). StarCoder: May the source be with you! arXiv preprint arXiv:2305.06161.
12. Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. arXiv preprint arXiv:2305.01210.
13. Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., ... & Serre, T. (2024). StarCoder 2 and The Stack v2: The Next Generation. arXiv preprint arXiv:2402.19173.
14. Peng, B., Quesnelle, J., Fan, H., & Shippole, E. (2023). YARN: Efficient context window extension of large language models. arXiv preprint arXiv:2309.00071.
15. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.
16. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., ... & Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887.
17. Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., ... & Velloso, E. (2024). BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv preprint arXiv:2406.15877.