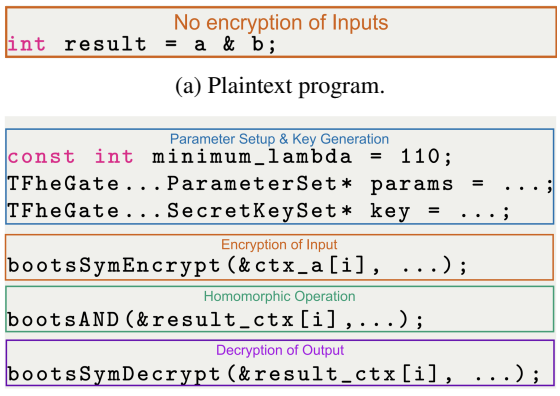
TFHE-CODER: EVALUATING LLM AGENTS FOR SECURE FULLY HOMOMORPHIC ENCRYPTION CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Fully Homomorphic Encryption over the Torus (TFHE) is a cornerstone of confidential computing, yet its adoption is severely limited by a steep learning curve requiring specialized cryptographic expertise. To bridge this skills gap, we investigate the potential of Large Language Model (LLM) agents to automate the generation of secure TFHE code from natural language. We introduce TFHE-CODER, a novel, three-phase agentic framework designed to overcome the critical failure points of this process. Our framework integrates a Prompt Formalizer to structure user intent and configure secure parameters, a specialized RAG retriever for accurate API knowledge , and an automated Security Verifier that provides iterative feedback to correct cryptographic flaws. We comprehensively evaluate our framework by testing four leading LLMs on a benchmark of ten programming tasks of increasing difficulty. Our results demonstrate that while baseline agents consistently produce functionally correct but insecure code, our full agentic framework is uniquely capable of generating solutions that are simultaneously compilable, functionally correct, and verifiably secure. This work establishes the first robust methodology and benchmark for agentic TFHE code generation, demonstrating a viable path toward democratizing secure computation.

1 INTRODUCTION



(a) Plaintext program.

(b) TFHE program.

Figure 1: High-level structure of TFHE programs.

Fully Homomorphic Encryption (FHE) Gentry (2009); Lou & Jiang (2019); Zhang et al. (2024); Lou et al. (2021) allows computing over encrypted data, eliminating the need for decryption during processing Brakerski et al. (2014); Lou & Jiang (2021); Brakerski (2012); Zhang et al. (2023); Fan & Vercauteren (2012); Cheon et al. (2017); Chillotti et al. (2020); Xue et al. (2022); Zhang et al. (2025); Lou et al. (2019a); Zheng et al. (2023). It is, therefore, a promising cryptographic tool to ensure data privacy in the settings of secure computation, such as privacy-preserving machine learning Gilad-Bachrach et al. (2016); Lou et al. (2019b); Santriaji et al. (2024), secure multi-party computation Jin et al. (2023), private blockchain transactions Madathil & Scafuro (2023), and secure medical diagnostic Raisaro et al. (2018). There are various practical FHE schemes have been proposed. Among these, the TFHE scheme stands out. It is unique by offering efficient gate bootstrapping and functional bootstrapping, which allow for the computation of arbitrary functions while refreshing the noise.

However, the adoption of privacy-preserving computation frameworks, such as TFHE, has failed to keep pace with rising industry and academic demand. A principal challenge hindering their widespread implementation is the shortage of developers possessing the necessary specialized skills. Developing applications with TFHE requires a deep understanding of advanced cryptographic and mathematical concepts, a skill set distinct from the general programming expertise (e.g., in Python) held by the broader developer community. This work confronts this challenge by exploring a fundamental question: *can LLM Agents be used to translate Natural Language to secure TFHE code?*

Recent advancements in Large Language Models (LLMs) Jiang et al. (2024); Xue et al. (2024) have showcased their remarkable capacity to comprehend natural language. Regarding coding, LLMs can assist developers by suggesting code snippets and even offering solutions to common programming challenges Mastropaolo et al. (2023); Nijkamp et al. (2022). To this end, leveraging LLMs' capabilities to assist developers implement secure TFHE applications is a promising avenue for addressing the complexities associated with TFHE implementations. It would be valuable to evaluate if LLM Agents could help developers with the TFHE coding, such as encryption parameters configuration and correct API calling, automatically. To this end, the expertise barrier could be significantly lowered, making TFHE more accessible for developer with few related expertise.

However, when regular code generation agents are employed for generating TFHE code in an automated fashion, they often fail to follow the instructions and produce plaintext programs like Fig. 1a. This failure stems from several core challenges inherent to this specialized domain. Models trained on general-purpose code often lack a fundamental understanding of the required TFHE program structure and the critical process of selecting appropriate security parameters. Furthermore, they frequently exhibit poor knowledge of the correct library APIs, leading them to hallucinate functions or misuse existing ones in ways that break homomorphic properties. Moreover, traditional code generation metrics like Pass@k Chen et al. (2021) are ill-equipped for this context, as they only evaluate functional correctness, not cryptographic security. A program can therefore pass such tests by operating on plaintext data, completely failing its primary privacy-preserving objective and highlighting that functional correctness is an insufficient and misleading proxy for success in secure code generation.

Therefore, to mitigate each of these issues, we introduce the novel agentic code generation workflow and evaluation framework as shown in Fig. 2. Our workflow is composed of three key components designed to address these specific challenges. First, the FHE Prompt Formalizer (Fig. 3) corrects structural and parameterization errors by translating the user's request into a formal specification with secure, correctly calculated cryptographic parameters. Second, to remedy the model's lack of API knowledge, an FHE API RAG Retriever (Fig. 4) provides the agent with relevant documentation and code examples on-demand. Finally, to overcome inadequate evaluation, our FHE Security Verifier (Fig. 5)introduces a multi-faceted check for critical security properties, ensuring the generated code is not only functionally correct but also verifiably secure.

We summarize our contributions as follows:

- We propose a **novel, three-phase agentic workflow for secure TFHE code generation**. This workflow includes an FHE Prompt Formalizer to ensure correct program structure and parameterization, an FHE API RAG Retriever to provide necessary API knowledge, and an FHE Security Verifier to validate the cryptographic integrity of the output.
- We introduce a **new evaluation metric, Pass@1 (security)**, specifically designed to measure the security of generated TFHE code, addressing the shortcomings of traditional metrics like $Pass@k$ that only assess functional correctness.
- To the best of our knowledge, this work presents the **first comprehensive benchmark** and evaluation of large language model agents for the task of generating code **for Fully Homomorphic Encryption over the Torus (TFHE)**.

2 BACKGROUND

The intersection of large language models (LLMs) and fully homomorphic encryption (FHE) presents a unique opportunity to democratize secure computation. While LLMs have shown remarkable prowess in code generation for mainstream languages, their application to specialized cryptographic libraries like TFHE remains unexplored. This section examines the potential of LLMs to

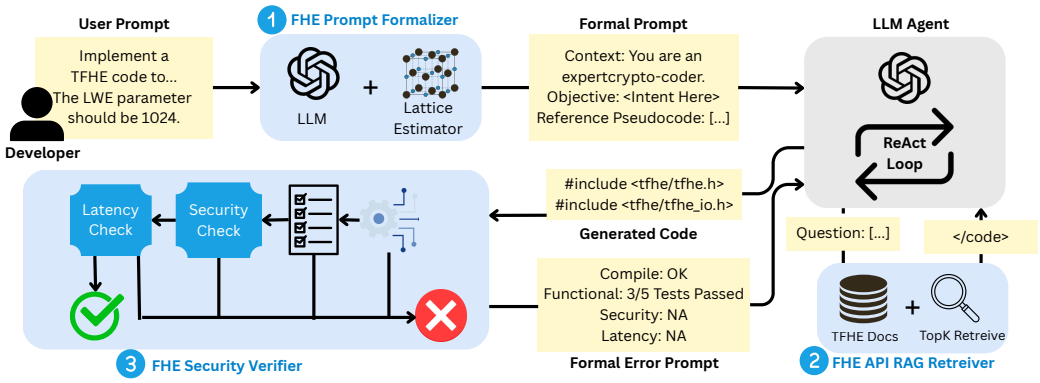


Figure 2: An overview of our workflow for secure FHE code generation. Our key contributions (highlighted with stars) are: (1) the FHE Prompt Formalizer, which enriches a developer's prompt with secure parameters from a Lattice Estimator; (2) the FHE API RAG Retriever, which provides the agent with expert-annotated API usage examples; and (3) the FHE Security Verifier, which provides an automated feedback loop for security and correctness. Maximum Iterations is set to 10.

generate TFHE code, leveraging their understanding of logical operations, and explores the unique characteristics of TFHE that make it both challenging and promising for automated code generation.

2.1 TFHE

Fully Homomorphic Encryption over the Torus (TFHE) Jiang et al. (2022) operates on boolean circuits using logical gates (NOT, AND, OR) with explicit noise management through bootstrapping after each operation. While TFHE requires adherence to strict security parameters, it offers several advantages over schemes like BGV Brakerski et al. (2014); Yudha et al. (2024) and CKKS Cheon et al. (2017) in terms of practical implementation. TFHE's boolean circuit approach aligns more closely with traditional programming paradigms, making it easier for developers to conceptualize and implement encrypted computations. Its efficient gate-by-gate bootstrapping is faster and more straightforward to implement than the complex relinearization and modulus switching procedures required in BGV/CKKS. TFHE's deterministic noise management simplifies handling in code implementation, as noise is reset after each gate operation. Additionally, TFHE's structure allows for efficient hardware acceleration, potentially simplifying high-performance implementations. However, programming TFHE still presents challenges, including the need to carefully manage bootstrapping operations and adhere to specific security parameters to maintain the scheme's integrity.

2.2 LLMs for Code Generation

Code generation is a key application of large language models (LLMs), with models such as CodeGen Nijkamp et al. (2022), CodeX Chen et al. (2021), and CodeT5 Wang et al. (2021) excelling in widely used languages like C, C++, Python, and Java due to the availability of extensive training corpora. However, generating code for specialized libraries like TFHE, implemented in C, presents challenges due to its cryptographic complexity and niche API. Recent studies on LLMs for High-Level Synthesis (HLS) and Register Transfer Level (RTL) design Thakur et al. (2023); Liao et al. (2024); Xiong et al. (2024) demonstrate that LLMs can effectively model logical operations such as AND and OR gates. Given that TFHE operations also rely on gate-level computations, it is reasonable to hypothesize that LLMs, with appropriate improvement techniques, could generate functional TFHE code by leveraging their learned logical reasoning capabilities.

3 Our Method

Our Agentic workflow 2 has three main components: **FHE Prompt Formalizer**, **FHE API RAG Retriever** and **FHE Security Verifier**. The Agent is implemented using the ReAct Yao et al. (2023) prompting strategy.

3.1 FHE Prompt Formalizer

A novice user might not be aware of how to choose the appropriate security parameter for a particular FHE scheme (λ in case of TFHE). First, we use an Intent extraction LLM which separates the intent (the overall goal) from any specification that the user may have give. This (partial) specification is then passed to the Lattice Estimator Albrecht et al. (2015) which solves for λ parameter. Next, the intent and the parameter are passed to a Formal Specification LLM which outputs a Dafny Leino (2010) pseudo-code and code requirements. Finally the pseudo-code, the requirements and the intent is set into a Formal Prompt template which is finally fed to the agent. Note that the formal prompt is much more precise compared to the initial user prompt. The ensure statements in the Dafny code guide the agent write `assert` statements in actual code which ensure correctness of the generated solution at all intermediate steps.

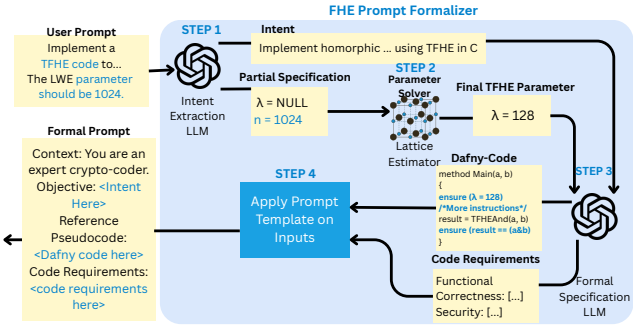


Figure 3: LLMs and a Lattice Estimator transform a developer's prompt into a secure and structured set of instructions. The process extracts the user's intent, solves for correct cryptographic parameters, and generates a final formal prompt containing Dafny-based pseudocode and security requirements to guide the agent.

3.2 FHE API RAG Retriever

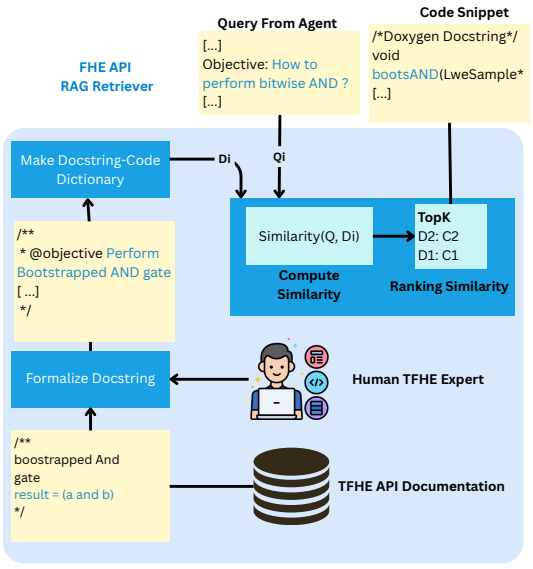


Figure 4: An offline, human-in-the-loop process creates a dictionary mapping expert-enriched docstrings to code snippets from the TFHE documentation. When the agent queries this dictionary with a natural language objective, a similarity search returns the most relevant and accurate code snippet.

The agent might not be aware of the correct TFHE API and its usage. Additionally, in our initial experimentation, we found that NL to code retrieval is not very optimal. To mitigate this issue, we adapt the vanilla RAG for TFHE by extracting the docstring of the methods and enhancing them into more descriptive text following the Doxygen format [1]. We chose Doxygen specifically because its structured tags, such as @objective, embed clear, machine-readable semantic metadata into the documentation. This structure allows for a more precise similarity search between the agent's intent and the function's documented purpose, overcoming the ambiguity of plain natural language. This docstring is then paired with the corresponding code-snippet. When the agent queries the retriever, the similarity is computed between the query and the docstring, and the most relevant docstring along with the corresponding code-snippet is returned. The chunking-size was set to 600, and the chunk-overlap was set to 120.

3.3 FHE Security Verifier

The FHE Security Verifier implements a four-stage automated validation process (Security, Functional, Compile, and Latency checks) to

[1] https://www.doxygen.nl/

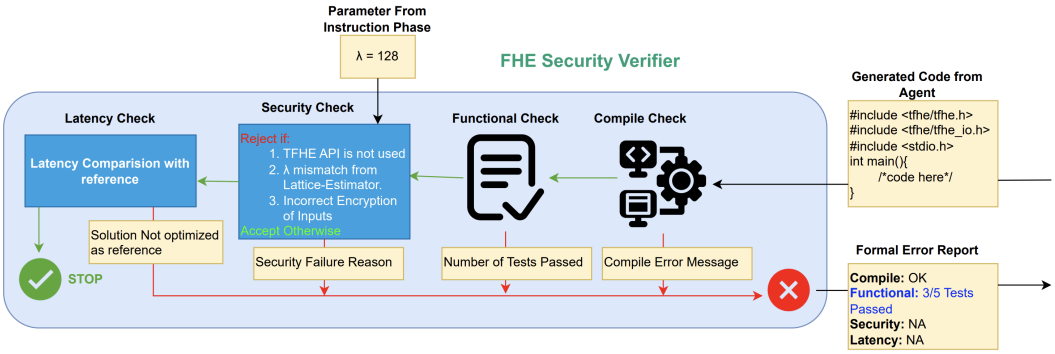


Figure 5: This automated pipeline validates generated code across four stages: Compile, Functional, Security, and Latency. The critical Security Check verifies correct API usage, secure parameter configuration, and proper input encryption. If any stage fails, a consolidated Formal Error Report is generated and returned to the agent for iterative correction; otherwise, the solution is accepted.

ensure generated TFHE code meets both functional and security requirements. The Security Check serves as the primary gate, analyzing code for three critical vulnerabilities: improper TFHE API usage, incorrect lattice parameter configuration, and plaintext data leakage. When violations are detected, the system provides diagnostic feedback through an automated correction loop with up to 10 iterations, addressing the fundamental challenge that baseline LLMs often produce functionally correct but cryptographically insecure implementations.

4 Experiment Design Section

4.1 Problem Definition

We introduce the LLM-Agentic TFHE Generation and Evaluation Framework, illustrated in Figure 2. In this framework, each TFHE task is formulated as a natural language prompt and provided to the agent. Leveraging its reasoning capabilities, the agent may (optionally) consult external documentation through retrieval-augmented generation (RAG) before implementing the corresponding code. The generated code is subsequently evaluated by the Security Verifier, which consolidates any compilation or verification errors into structured feedback and returns it to the agent for refinement. The agent then iteratively revises its solution, continuing this process until all security checks are satisfied or a predefined iteration limit is reached.

4.2 Workload Selection

We describe the workloads for the code generation breifly in Table 1.

Table 1: Benchmark workloads for evaluating TFHE code generation, progressing from elementary boolean operations to sophisticated machine learning architectures.

Workload	**Description**
AND	Bitwise AND between two 32-bit integers.
ReLU	ReLU on a signed 32-bit integer.
Adder	Adding two between two 32-bit integers.
Multiplier	Multiplying two 32-bit integers.
Vector Addition	Vector addition between two integer vectors of length 5.
Vector Dot Product	Inner-product of two integer vectors of length 5.
Matrix-Vector Multiplication	Multiplication between an encrypted vector and a plaintext matrix.
Matrix-Matrix Multiplication	Multiplication between an encrypted matrix and a plaintext matrix.
MLP	A simple 3-layer MLP with ReLU activation.
CNN	A small Convolutional Neural Network with a fully connected layer.

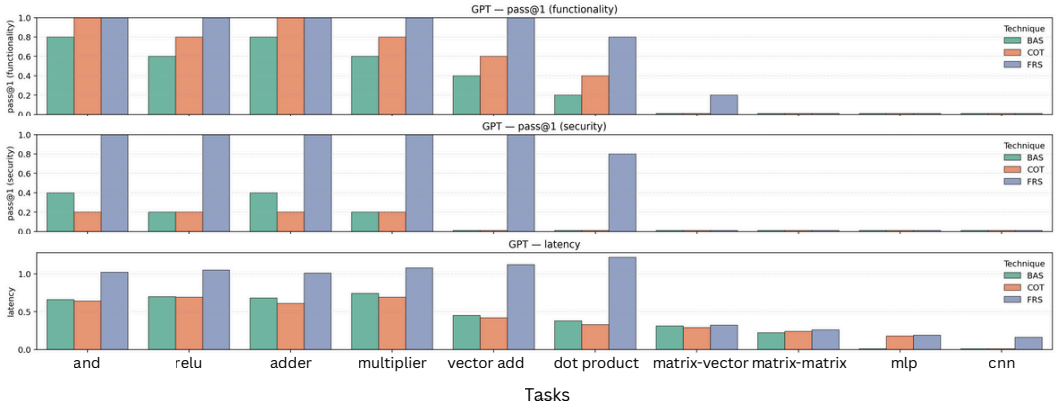


Figure 6: Performance of GPT-5 across all tasks. A comparison of our framework (FRS) against Baseline (BAS) and Chain-of-Thought (COT) techniques. While the baseline methods fail on complex tasks and are fundamentally insecure (near-zero security pass rate), FRS consistently produces code that is both functionally correct and verifiably secure.

These tasks collectively evaluate LLMs' ability to synthesize both low-level cryptographic primitives and high-level machine learning components using TFHE's gate-level programming paradigm.

4.3 Model Selection

We select the latest LLMs to drive our agent. For open-source LLMs, we choose Qwen3-Coder-480B-A35B(Yang et al. (2025)) (QWE) and Deepseek-V3.1(Liu et al. (2024))(DSK). For closed-source LLMs, we select Gemini-2.5-Pro(Comanici et al. (2025))(GEM) and GPT-5(OpenAI (2025))(GPT). For all studied LLMs, we set the temperature to 0.5. Note that, to mitigate issues stemming from the randomness of model generation, the experimental results presented in this paper are obtained by conducting five repeated experiments and averaging the results. The embedding model used for RAG was `text-embedding-3-small` [2] from OpenAI.

4.4 Metrics

In our framework, we employ three key metrics to judge the quality of the generated codes. Following prior works on code-generation, we use **1. *Pass@k(func)*** to denote the fraction of generated codes that pass the unit tests. We present our novel metric **2. *Pass@k(security)***, which denotes the fraction of generated codes that are secure; a program is considered **secure** only if it passes an automated analysis verifying: **(i)** exclusive use of TFHE APIs to prevent plaintext data leakage, **(ii)** correct configuration of cryptographic parameters against secure values from the Lattice Estimator, and **(iii)** proper encryption of all inputs before their use. A failure in any of these checks renders the code insecure. Our third metric is **3. *Latency***, compared to expert-written reference codes.

4.5 Baselines

Our first baseline denotes the **regular code generation** workflow. This can be constructed by removing the FHE Prompt Formalizer, FHE API RAG Retreiver and removing the security and latency checks from the proposed workflow in Fig. 2. We abbreviate it as BAS. Our second baseline is **Zero-shot Chain-of-Thought** agent, which builds upon the regular workflow by appending a step-by-step worked example of correct TFHE code generation. We abbreviate it as COT.

5 Evaluation Results

This section presents a comprehensive empirical analysis of our proposed agentic framework (denoted as FRS) in comparison to a regular code generation workflow (BAS) and a Zero-shot Chain-of-

[2] https://platform.openai.com/docs/models/text-embedding-3-small

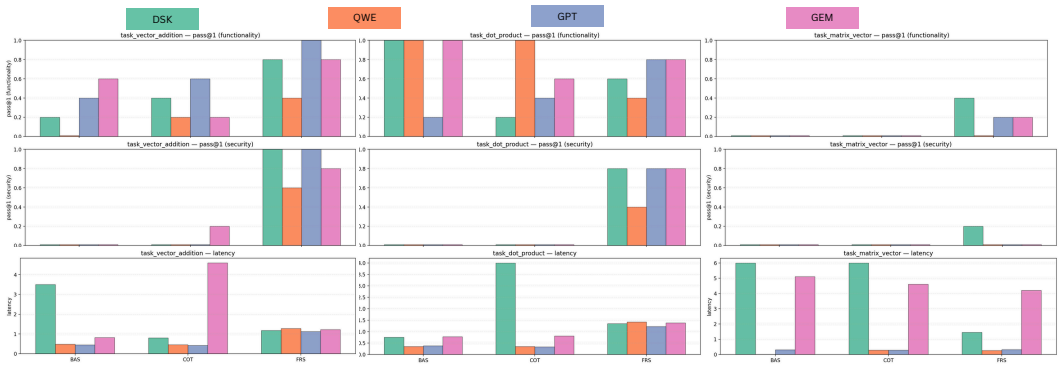


Figure 7: A comparison of our framework (FRS) against baselines (BAS, COT) using four different LLMs. The results show that the security failures of baseline methods are universal; all models produce insecure code with BAS and COT across all tasks. In contrast, our FRS framework is the only technique that provides a consistent and essential security uplift, proving its benefits are model-agnostic.

Thought agent (COT). We evaluated four leading LLMs on a benchmark of ten TFHE programming tasks with varying complexity.

5.1 In-Depth Analysis on a State-of-the-Art Model

We first conduct a detailed analysis using the state-of-the-art GPT-5 model to illustrate the core performance differences between our framework and the baselines across all tasks. The results are presented in Fig. 6.

Functional Correctness: As shown in the top chart of Figure 6, the baseline methods (BAS and COT) demonstrate partial success on tasks with low complexity, such as `and` and `relu`. However, their performance exhibits a notable decline as task complexity increases, particularly for compositional tasks like `matrix-vector multiplication` and `CNN`. In contrast, our FRS framework maintains high functional correctness across the majority of tasks, indicating a greater robustness to increasing complexity.

Security: A critical distinction between our framework and the baselines is revealed in the security evaluation. Both the BAS and COT methods yield a `pass@1(security)` approaching zero for all tasks evaluated. This finding indicates that they consistently fail to produce secure code, often generating plaintext implementations that, while sometimes functionally correct, do not adhere to the required cryptographic protocols. Conversely, the FRS framework achieves near-perfect security scores across the entire benchmark. This outcome underscores the necessity of a guided, multi-phase process—encompassing the proposed prompt formalization, accurate API retrieval, and security verification—to meet the specific requirements of secure code generation.

Latency: The performance trade-offs are detailed in the bottom chart of Figure 6, which shows the latency overhead. The FRS framework naturally incurs higher latency due to its iterative feedback loop and verification stages. This overhead is a deliberate design choice, representing a practical trade-off for the significant improvements in security and functional reliability.

5.2 Generalizability Across Diverse LLMs

To ensure our findings are not model-specific, we assessed the generalizability of our framework by applying it to four different LLMs. Fig. 7 presents a comparative analysis on three representative tasks. The results confirm that the performance patterns persist across all models. The security deficiencies of the baseline methods are model-agnostic; both BAS and COT fail to generate secure code regardless of the LLM used. In contrast, the FRS framework is the only approach that enables the models to consistently produce secure outputs. While the overall performance ceiling is influenced by the base model's intrinsic capabilities—with GPT-5 and Gemini-2.5-pro generally outperform-

ing Deepseek-V3.1 and Qwen3-Coder —our framework provides a consistent and essential security impoves for all tested models.

5.3 Solving Complex Tasks with Structured Decomposition

While our FRS framework demonstrates strong performance on atomic tasks, Figure 6 shows that its effectiveness diminishes on complex, compositional problems that require multi-step reasoning (e.g., matrix-vector multiplication, MLP). To address this limitation, we evaluate a structured decomposition approach where a complex task is broken down into simpler, solvable subtasks.

In this approach, the agent first generates secure code for the necessary primitives (such as dot product). A final "composer" agent is then provided with a structured prompt containing these verified subroutines and a high-level goal to compose them into the final solution.

As detailed in Table 2, our structured decomposition approach effectively solves complex tasks where the direct method fails. It achieves high functional and security scores across most tasks, reaching a perfect 1.0 security pass rate for matrix-vector and matrix-matrix multiplication. The increased compositional complexity of the MLP task lowers both its functional (0.4) and security (0.6) pass rates, suggesting a link between logical complexity and the agent's ability to maintain security protocols. Overall, this hierarchical strategy of composing solutions from verified sub-tasks significantly extends the agent's capabilities to a previously unsolvable class of problems.

Table 2: Performance using structured decomposition.

Task	Pass@1(func)	Pass@1(sec)
Matrix-Vector	0.8	1.0
Matrix-Matrix	0.8	1.0
MLP	0.4	0.6
CNN	0.8	0.8

5.4 Ablation Study

To quantitatively assess the contribution of each component within our framework, we conduct an ablation study by systematically removing each of our three key modules: the FHE Prompt Formalizer (FP), the FHE API RAG Retriever (RAG), and the FHE Security Verifier (SC). The results for the GPT-5 model on the representative Vector Addition task, shown in Figure 8, reveal a clear performance hierarchy.

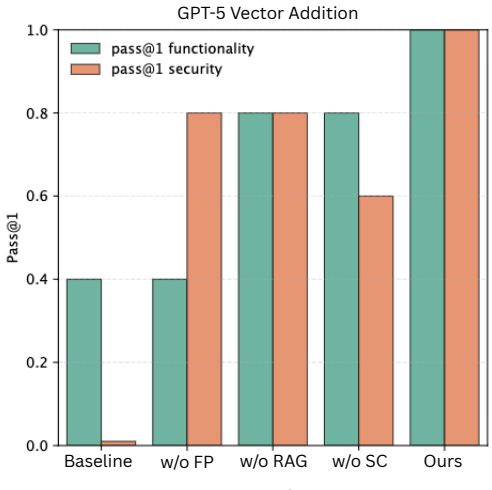


Figure 8: Ablation study on the Vector Addition task. The results reveal the specific contribution of each module. Removing the Security Verifier (w/o SC) eliminates the security guarantee, while removing the Prompt Formalizer (w/o FP) reduces functionality to baseline levels. This demonstrates that all components are essential for the perfect functional and security performance of the full framework (Ours).

The study establishes that the **Baseline** agent, lacking our modules, is unable to produce secure TFHE code, achieving a low functional pass rate of 0.4 and zero security pass rate. Removing the **FHE Security Verifier (w/o SC)** from our full system causes the most critical security degradation, with the security pass rate falling from 1.0 to 0.6. This highlights that the iterative feedback from the verifier is indispensable for correcting cryptographic flaws. Similarly, ablating the **FHE API RAG Retriever (w/o RAG)** degrades both functionality and security to 0.8, confirming that access to correct API knowledge is crucial. Removing the **FHE Prompt Formalizer (w/o FP)** causes the functional pass rate to collapse to baseline levels (0.4), demonstrating its vital role in achieving functional correctness. Ultimately, the **Ours (Full)** configuration is the only one to achieve a perfect 1.0 score on both metrics, confirming that all three components are essential and com-

plementary, working in concert to generate code that is simultaneously secure and functionally correct.

6 DISCUSSION

Our work demonstrates a viable path toward automating the generation of secure TFHE code, a task where standard Large Language Model agents consistently fail. The core finding is that for security-critical domains, a simple prompt-to-code workflow is insufficient. Success requires a structured, multi-phase agentic framework that guides the model from instruction to generation and, most critically, through iterative, security-aware feedback. Our results show that baseline agents, while sometimes achieving functional correctness, have a near-zero success rate in producing secure code. This highlights a fundamental misalignment: traditional code generation metrics like *Pass@k(func)* are misleading proxies for success in cryptographic programming. Our framework, particularly through the FHE Security Verifier, directly addresses this by making verifiable security a primary objective of the generation process.

The ablation study underscores that our framework's success is not attributable to a single component but to their synergistic effect. The FHE Prompt Formalizer and RAG Retriever provide the necessary *a priori* knowledge of structure and API usage, preventing a significant number of errors. However, it is the *a posteriori* feedback from the Security Verifier that proves indispensable, correcting subtle but critical flaws that persist even with a well-formed prompt. Furthermore, our experiments with structured decomposition reveal a key insight : while the agents struggle with end-to-end reasoning for complex, compositional tasks, they excel as "composers" when provided with a structured prompt containing verified, functional sub-components.

7 CONCLUSION

The adoption of powerful cryptographic tools like TFHE is severely limited by a steep learning curve and the need for specialized expertise. In this work, we investigated whether LLM agents could bridge this skills gap by automatically translating natural language specifications into secure TFHE code. Our findings reveal that while standard agents consistently fail due to a lack of domain-specific knowledge and the inadequacy of traditional evaluation metrics, a structured agentic approach can successfully overcome these challenges. We introduced TFHE-CODER, a three-phase framework that integrates prompt formalization, retrieval-augmented generation, and a critical security verification loop to reliably guide LLM agents.

Our comprehensive evaluation demonstrates that our framework enables the generation of code that is simultaneously functional and verifiably secure—a task at which baseline methods consistently fail. Furthermore, we showed that a structured decomposition strategy can extend this capability to complex, compositional tasks that are otherwise unsolvable. By establishing a robust methodology and benchmark for secure TFHE code generation, this work takes a significant step toward democratizing secure computation. Our framework effectively lowers the barrier to entry, enabling developers without deep cryptographic knowledge to correctly and safely utilize advanced privacy-preserving technologies. Ultimately, TFHE-CODER serves as a blueprint for developing LLM agents in other security-critical domains, proving that with the right scaffolding, these models can be transformed from unreliable assistants into valuable partners for secure software engineering.

8 ETHICS STATEMENT

Large language models (LLMs) were used to check grammer and polish writing.

9 REPRODUCIBILITY STATEMENT

In order to ensure reproducibility, we fixed the seed while inferencing the models.

REFERENCES

Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual cryptology conference*, pp. 868–886. Springer, 2012.

Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23*, pp. 409–437. Springer, 2017.

Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pp. 201–210. PMLR, 2016.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *The Design Automation Conference (DAC 2022)*, 2022.

Weizhao Jin, Yuhang Yao, Shanshan Han, Jiajun Gu, Carlee Joe-Wong, Srivatsan Ravi, Salman Avestimehr, and Chaoyang He. Fedml-he: An efficient homomorphic-encryption-based privacy-preserving federated learning system. *arXiv preprint arXiv:2303.10837*, 2023.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pp. 348–370. Springer, 2010.

Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. Are llms any good for high-level synthesis? *arXiv preprint arXiv:2408.10428*, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Qian Lou and Lei Jiang. She: A fast and accurate deep neural network for encrypted data. In *Advances in Neural Information Processing Systems (NeurIPS) 2019*, pp. 10035–10043, 2019.

Qian Lou and Lei Jiang. Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. *ICML 2021*, 2021.

Qian Lou, Bo Feng, Geoffrey C Fox, and Lei Jiang. Glyph: Fast and accurately training deep neural networks on encrypted data. *NeurIPS 2020 (Advances in Neural Information Processing Systems)*, 2019a.

Qian Lou, Feng Guo, Lantao Liu, Minje Kim, and Lei Jiang. Autoq: Automated kernel-wise neural network quantization. In *International Conference on Learning Representations (ICLR) 2020*, 2019b.

Qian Lou, Yilin Shen, Hongxi Jin, and Lei Jiang. Safenet: A secure, accurate and fast neural network inference. 2021.

Varun Madathil and Alessandra Scafuro. Prifhete: Achieving full-privacy in account-based cryptocurrencies is possible. *Cryptology ePrint Archive*, 2023.

Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2149–2160. IEEE, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

OpenAI. Introducing GPT5. `https://openai.com/index/introducing-gpt-5/`, August 2025. Accessed: 2025-09-21.

Jean Louis Raisaro, Gwangbae Choi, Sylvain Pradervand, Raphael Colsenet, Nathalie Jacquemont, Nicolas Rosat, Vincent Mooser, and Jean-Pierre Hubaux. Protecting privacy and security of genomic data in i2b2 with homomorphic encryption and differential privacy. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(5):1413–1426, 2018.

Muhammad Husni Santriaji, Jiaqi Xue, Qian Lou, and Yan Solihin. Dataseal: Ensuring the verifiability of private computation on encrypted data. *arXiv preprint arXiv:2410.15215*, 2024.

Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6. IEEE, 2023.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. Hlspilot: Llm-based high-level synthesis. *arXiv preprint arXiv:2408.06810*, 2024.

Jiaqi Xue, Lei Xu, Lin Chen, Weidong Shi, Kaidi Xu, and Qian Lou. Audit and improve robustness of private neural networks on encrypted data. *arXiv preprint arXiv:2209.09996*, 2022.

Jiaqi Xue, Mengxin Zheng, Ting Hua, Yilin Shen, Yepeng Liu, Ladislau Bölöni, and Qian Lou. Trojllm: A black-box trojan prompt attack on large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Ardhi Wiratama Baskara Yudha, Jiaqi Xue, Qian Lou, Huiyang Zhou, and Yan Solihin. Boostcom: Towards efficient universal fully homomorphic encryption by boosting the word-wise comparisons. In *(PACT'24) The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024.

Yancheng Zhang, Xun Chen, and Qian Lou. Hebridge: Connecting arithmetic and logic operations in fv-style he schemes. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pp. 23–35, 2023.

Yancheng Zhang, Mengxin Zheng, Yuzhang Shang, Xun Chen, and Qian Lou. Heprune: Fast private training of deep neural networks with encrypted data pruning. *Advances in Neural Information Processing Systems*, 37:51063–51084, 2024.

Yancheng Zhang, Jiaqi Xue, Mengxin Zheng, Mimi Xie, Mingzhe Zhang, Lei Jiang, and Qian Lou. Cipherprune: Efficient and scalable private transformer inference. *arXiv preprint arXiv:2502.16782*, 2025.

Mengxin Zheng, Qian Lou, and Lei Jiang. Primer: Fast private transformer inference on encrypted data. *DAC 2023*, 2023.

A Appendix

A Full Code Example: Bitwise ReLU

Below is a full working example for a bitwise ReLU operation. Listing 1 provides a complete, runnable C program demonstrating the plaintext bitwise logic, which reads an integer from standard input. Listing 2 shows the full, equivalent implementation using the TFHE library.

```c
#include <stdio.h>
#include <stdint.h>

// Plaintext bitwise ReLU for a 32-bit signed integer.
int32_t relu(int32_t input) {
    int32_t mask = input >> 31;
    int32_t output = input & ~mask;
    return output;
}

int main() {
    int32_t input_val;

    scanf("%d", &input_val);

    printf("%d\n", relu(input_val));

    return 0;
}
```

Listing 1: Complete, runnable plaintext C code for bitwise ReLU.

```c
#include <tfhe/tfhe.h>
#include <tfhe/tfhe_io.h>
#include <stdio.h>
#include <assert.h>

// The plaintext relu function would be included here
int32_t relu(int32_t input);

int main() {
    // 1. Generate a keyset
    const int minimum_lambda = 110;
    TFheGateBootstrappingParameterSet* params =
    new_default_gate_bootstrapping_parameters(minimum_lambda);

    // Generate a deterministic key for reproducibility
    uint32_t seed[] = { 314, 1592, 657 };
    tfhe_random_generator_setSeed(seed, 3);
```

```
    TFheGateBootstrappingSecretKeySet* key =
    new_random_gate_bootstrapping_secret_keyset(params);

    int32_t plaintext1;
    scanf("%d", &plaintext1);

    // 2. Encrypt the 32-bit signed integer bit by bit
    LweSample* ciphertext1 = new_gate_bootstrapping_ciphertext_array(32,
    params);
    for (int i = 0; i < 32; i++) {
        bootsSymEncrypt(&ciphertext1[i], (plaintext1 >> i) & 1, key);
    }

    // 3. Homomorphically compute ReLU: result = input & ~mask
    LweSample* result = new_gate_bootstrapping_ciphertext_array(32,
    params);
    LweSample* mask = new_gate_bootstrapping_ciphertext_array(32, params)
    ;

    // 3a. Create a 32-bit mask from the encrypted sign bit (bit 31)
    for (int i=0; i<32; i++){
        bootsCOPY(&mask[i], &ciphertext1[31], &key->cloud);
    }

    // 3b. Invert the mask homomorphically
    for (int i=0; i<32; i++){
        bootsNOT(&mask[i], &mask[i], &key->cloud);
    }

    // 3c. Compute the final result: input & ~mask
    for (int i = 0; i < 32; i++){
        bootsAND(&result[i], &ciphertext1[i], &mask[i], &key->cloud);
    }

    // 4. Decrypt the result for verification
    int32_t final_result = 0;
    for (int i = 0; i < 32; i++) {
        int bit = bootsSymDecrypt(&result[i], key);
        final_result |= (bit << i);
    }
    printf("%d\n", final_result);

    // Verify against the plaintext function
    assert(final_result == relu(plaintext1));

    // Cleanup
    delete_gate_bootstrapping_ciphertext_array(32, mask);
    delete_gate_bootstrapping_ciphertext_array(32, result);
    delete_gate_bootstrapping_ciphertext_array(32, ciphertext1);
    delete_gate_bootstrapping_secret_keyset(key);
    delete_gate_bootstrapping_parameters(params);

    return 0;
}
```

Listing 2: Full secure TFHE implementation for bitwise ReLU.