CAP6614-HW5

Bradley Coles, Kyle Williams

April 2025

1 Agent Outline

We designed an LLM agent to generate CKKS scheme FHE code for five different tasks (addition, multiplication, vector dot product, matrix multiplication, and convolution) using the OpenFHE library. In order to accomplish this, we tested four different techniques against each other. We used few shot prompting to add examples to the agent to guide generated outputs. We also attempted a simple RAG technique to pass any relevant documentation from the OpenFHE library to the agent. We also tested execution-based consistency decoding from a pool of 15 codes to generate five different outputs. And finally, we also tested self-improvement by executing generated codes against the test cases and iterating through a debug session. Each of these fulfills one of the required techniques: prompting, decoding, self-improvement, and RAG. After getting outputs from these methods, we then collected metrics about the generated codes.

We used GPT4.0, Qwen2.5Coder, Llama, StarCoder2, and Deepseek to generate outputs. We generated 500 total codes, five codes for each combination of the 5 LLMs, 5 tasks, and 4 techniques (5 * 5 * 5 * 4 = 500).

1.1 Few Shot

Few shot prompting was accomplished by augmenting the prompts with a singular exemplar to guide the code generation style and use of OpenFHE methods such as CryptoContext generation using CCParams. These also helped to guide the output style for the code to match what output style is expected for test cases to ensure any errors with test are more likely from computational errors in the code than output formatting.

1.2 Execution-based similarity

Execution-based similarity takes a number of generated codes, compiles and executes them, then groups these based on output similarity. We then select output codes from the groups with the highest frequency, selecting more from the more frequent groups when fewer than 5 groups exist. All codes that produce the same output on test cases are considered semantically similar. Codes that

did not compile were considered invalid if there were codes that compiled and produced output.

1.3 Self-Debugging

For self improvement we perform self-debugging. This is accomplished by compiling and executing the code and gathering the compilation errors or execution outputs, including errors and the output results for each test case along with the expected result for each test case. These results and the original code are then passed back to the model and the model is asked to debug the code to make the code compile or meet test cases. For time constraints we only performed one iteration on the self-improvement.

1.4 RAG

We implemented a simple vector-based RAG technique. This selects some documents to pass in as context to the model based on semantic similarity to the input query. The entire documentation of OpenFHE was passed in as a vector space.

2 Results

Here we outline the results for each technique and then discuss a comparison of the techniques. We used (mostly) very small models, and each technique was applied individually; this meant that matching a very difficult code base that LLMs use infrequently to do relatively complicated tasks was mostly unsuccessful. If we were to combine the techniques, we imagine that the code we produce would be much more successful; for now, we compare who has the better pass@1 compilation and CrystalBleu scores, with some surprising results.

2.1 Few Shot

Task		ADD	CONV	DOT	MAT	MULT
Deepseek	Pass@1 Comp	0.0000	0.0000	1.0000	0.0000	1.0000
	Avg CrystalBLEU	0.2764	0.2301	0.2742	0.2320	0.2916
GPT4.0	Pass@1 Comp	0.0000	1.0000	1.0000	1.0000	0.0000
	Avg CrystalBLEU	0.3231	0.3032	0.4480	0.3372	0.1853
Llama	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.0010	0.1722	0.0005	0.2149	0.0008
Qwen	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2823	0.3000	0.2271	0.3073	0.2806
StarCoder2	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2749	0.2676	0.2273	0.2752	0.2792

Table 1: Few Shot Metrics per LLM and Task. NOTE: pass@1 functionality was 0 for all

Table 1 outlines the metrics gathered for all models using few shot prompting. The few shot section ended up having the highest CrystalBleu by far, out of all the techniques. This should not be surprising. Its method of utilizing existing code to match it means that it's fantastic for being similar to real OpenFHE code, even if it doesn't produce code that works. This was great for pumping up its score, even if a proper corpus for CrystalBleu would disincentivize this. Of these, Qwen and GPT4.0 had the most impressive CrystalBleu, and Llama was mostly terrible. This will be a running trend in the all techniques.

2.2 Execution Consistency Decode

Task		ADD	CONV	DOT	MAT	MULT
Deepseek	Pass@1 Comp	1.0000	1.0000	1.0000	1.0000	1.0000
	Avg CrystalBLEU	0.1150	0.1468	0.1540	0.1686	0.0705
GPT4.0	Pass@1 Comp	0.0000	0.0000	1.0000	1.0000	0.0000
	Avg CrystalBLEU	0.3494	0.1535	0.4481	0.2937	0.2835
Llama	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.0079	0.0233	0.0744	0.0173	0.0405
Qwen	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2858	0.2177	0.3142	0.2539	0.3141
StarCoder2	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.0006	0.0004	0.0004	0.0004	0.0006

Table 2: Decode Metrics per LLM and Task. NOTE: pass@1 functionality was 0 for all

Table 2 outlines the metrics gathered for all models using Execution Consistency Decode. Less of the code was compiled then we hoped, but DeepSeek impressively had all 5 codes compile. Without Fewshot's matching of given text, this technique was mostly out of luck and proved inefficient. Running on the smaller models, there is very little variety to make use of its gimmick. In many cases the codes that did compile were not even attempting FHE operations as Deepseek. StarCoder2 and Llama struggled to comprehend the use of FHE with no context and versions of themselves small enough to run on our computers.

2.3 Self Debug

Task		ADD	CONV	DOT	MAT	MULT
Deepseek	Pass@1 Comp	1.0000	1.0000	1.0000	1.0000	0.0000
	Avg CrystalBLEU	0.1989	0.0121	0.1879	0.0760	0.2081
GPT4.0	Pass@1 Comp	0.0000	0.0000	1.0000	1.0000	0.0000
	Avg CrystalBLEU	0.2183	0.1674	0.4314	0.3649	0.3496
Llama	Pass@1 Comp	0.0000	0.0000	1.0000	0.0000	0.0000
	Avg CrystalBLEU	0.0004	0.0002	0.1886	0.0002	0.0004
Qwen	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2787	0.0648	0.3015	0.2849	0.2860
StarCoder2	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2104	0.0195	0.0478	0.0211	0.0425

Table 3: Self-Improvement Metrics per LLM and Task. NOTE: pass@1 functionality was 0 for all

Table 3 shows the metrics extracted for each model on each task using the self-debug technique. Few codes compiled and none managed to execute successfully. Since many codes did not compile on the first generation and we only did one generation of debugging due to time constraints the debugged codes often failed to compile or did not manage to read input vectors to execute the test cases. In the future, repeatedly self debugging may be a neat trick, but only one passthrough proved mostly ineffective. This did, however, give it the highest compile rate of any program, though not by much – it actually tied with our decode technique. Errors were not able to successfully resolved for for most models. For example, GPT4.0 insisted on using template functions that do not exist repeatedly. It used them so consistently, however, it may have been trained on a similar repository where such usage is valid. OpenFHE creates CryptoContext using CCParams¡CryptoContextCKKSRNS¿, so genCryptoContext being used so often caused it fail to be functional.

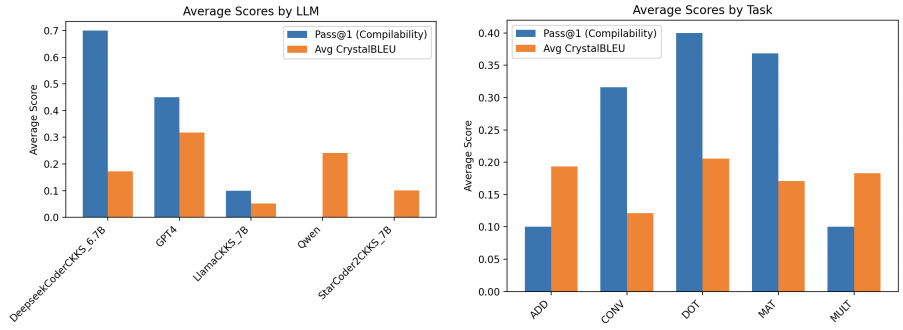
2.4 RAG

Task		ADD	CONV	DOT	MAT	MULT
Deepseek	Pass@1 Comp	0.0000	1.0000	1.0000	1.0000	0.0000
	Avg CrystalBLEU	0.3141	0.0123	0.1567	0.0791	0.2446
GPT4.0	Pass@1 Comp	0.0000	1.0000	0.0000	1.0000	0.0000
	Avg CrystalBLEU	0.3475	0.3269	0.3438	0.3234	0.3667
Llama	Pass@1 Comp	0.0000	1.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.1163	0.0000	0.0071	0.1081	0.0499
Qwen	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.2622	0.0681	0.2795	0.0673	0.2755
StarCoder2	Pass@1 Comp	0.0000	0.0000	0.0000	0.0000	0.0000
	Avg CrystalBLEU	0.0043	0.1199	0.0043	0.1255	0.0914

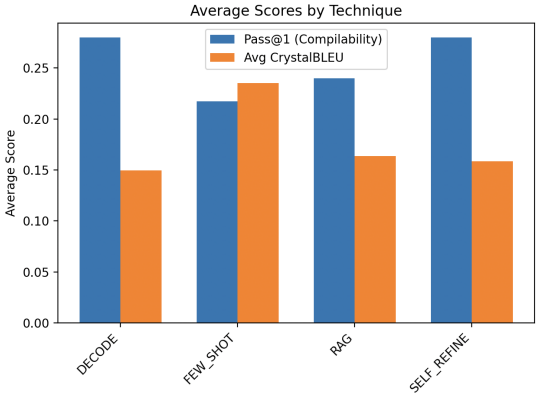
Table 4: RAG Metrics per LLM and Task. NOTE: pass@1 functionality was 0 for all

The RAG method was less performative than we expected as seen by the results in Table 4. We implemented the entire OpenFHE code into a vector space, but this was poorly aimed; it would have been much better if we found the most useful functions and provided it, such as EvalAdd or EvalMult. The technique was instead selecting other contexts that were more removed from the immediately necessary code such as serialization. As such we theorize that a graph based RAG technique would have seen better results and could have outperformed few shot prompting. We believed that a basic RAG may have still been able to succeed so we attempted this instead of graph based, but our hopes proved too optimistic; document size, similarity comparison for selection, and number of documents retrieved may have needed to be optimized.

3 Conclusion



(a) Average Performance Metrics by LLM (b) Average Performance Metrics by Task



(c) Average Performance Metrics by Technique

Figure 1: Averaged Performance Metrics

Overall, as shown in Fig 1, the most effective technique at compilation was self debugging and execution based decoding. The decode is likely artificially inflated due to prioritizing code that compiles and Deepseek did end up producing lots of compiling code, though none of this code performed FHE operations. This seems like a significant weakness in the pass@k metric, to give the highest scores to a model completely off the mark compared to, say GPT4.0. A graph based RAG technique that retrieves better context may have outperformed this technique, but our optimistic use of making a vector space for all of OpenFHE's documentation proved unhelpful. Our implementation of a basic RAG technique was not able to get relevant enough context to guide the generation of effective codes as compared to using example codes that were similar to the goal prompt to guide the LLMs to use proper OpenFHE syntax. A large part of the errors within the generated codes that were close to OpenFHE format and attempted to perform FHE operations was from models hallucinating about syntax and

insisting on using methods that either did not exist or were configured with the wrong inputs. RAG should be able to provide context to aid with the syntax. However, our use of the technique did not display this behavior.

The best performing model for compilation appears to be Deepseek, but most Deepseek codes did not perform any FHE operations and would therefore be invalid and not useful. The scores for Llama and StarCoder2 are low across all techniques as the models often failed to produce anything resembling code. Deepseek outputs that did compile often did not perform FHE operations. Overall, only GPT4.0 and Qwen produced codes attempting FHE operations consistently, even if Qwen codes did not compile.

Surprisingly, the best performing task for compilation was dot product. GPT was able to produce a good number of compiling codes for this task, and though these codes often correctly performed a dot product under FHE they did not pass functionality testing as they all used hard coded vector values or failed to read input values correctly.

FewShot prompting methods did sometimes produce code that compiled, but the code did not always correctly process the input from the test cases resulting in execution failures. Some of the code that did compile was properly performing some of the operations required for the task but was using hard coded vectors for the operations so failed test cases. This would be usable code for users looking to take and modify the code to write some solution incorporating FHE, but it does not manage to score well on our functionality metric. A CoT method with FewShot examples may have proved more successful here due to the complexity of the logic in FHE code. If the models were informed to take it step by step and provide a method to read inputs, setup context, encode vectors, encrypt vectors, perform the operations in question, decode the result, and print the result they may have been more likely to generate code that could break down and execute these steps effectively.

FewShot prompting most consistently produced the highest CrystalBleu scores. An evaluator in the loop framework that combines prompting benefits of producing more often compile-able code using CoT with FewShot examples with the self-improvement loop of regenerating until test cases are satisfied would likely produce a more effective agent than the methods proposed here. Deeper analysis using fewer LLMs, more models, and a more robust extraction and evaluation process to better ensure the generation of codes attempting FHE operations would be an interesting future extension of this project.

It's clear the models we selected, mostly able to run on our computers, were far too small to handle something as complex as OpenFHE. We prioritized models that could run on our local computers, and as such, chose poor models. The LLM that produced the most sensible code was, intuitively, GPT4.0 which produced code that was almost right, although it hallucinated functions far too

often.

While current LLM agents show promise in scaffolding FHE code, achieving reliable functional correctness for complex schemes like CKKS remains a significant challenge, one which almost certainly requires combining techniques instead of testing them separately against each other.