Comparative Analysis of Different Prompting Methods for Large Language Model

Hyoyeon Lee[1]

[1] School of Computer Science, University of Bristol

1 Introduction

This document presents a comparative analysis of various prompting techniques based on their memory usage and execution time when applied to Gemini-Flash 1.5. The study evaluates three distinct methods: one-shot prompting, Chain-of-Thought (CoT) prompting, and few-shots prompting. Notably, the current implementation does not incorporate counterexamples from the CBMC (C Bounded Model Checker) with the new prompts. Instead, it uses 'simple restart' method which uses the same prompt and relies on the possibility of obtaining a different response from the LLM than previously returned. Additionally, the number of reattempt has been restricted to three to avoid the hallucination of the LLM and endless reiteration of the software. This comparison aims to provide insights into the efficiency and performance of each prompting strategy, thereby guiding the appropriate usage of prompting methods.

2 Experimental Setup

In this section, I define the overall procedure of the experiment and its environment settings. For the exact prompt, check Section 3.

2.1 Implementation

I tested three different prompting methods to input code into Gemini Flash-1.5, receive the generated response, and verify its correctness using CBMC. The prompt has been refined from looping until it individually checks all proof harness functions generated to create a new function, 'combined_proof_harness()', to verify its correctness all at once. This approach reduced the running time as it removed the need of creating a separate JSON file of CBMC results per every proof harness functions. A 'simple restart' method has been applied when reattempting to query the LLM.

2.2 LLM

Gemini Flash 1.5 has been used for generating proof harness functions, where its temperature has been fixed to default temperature of 0.25 to ensure its determinism. Other hyperparameters have been set to default as well.

2.3 CBMC

CBMC 6.0.1 has been used to verify the correctness of generated proof harness functions. Identical CBMC command '`cbmc FILE_NAME.c --function combined_proof_harness --no-standard-checks --no-malloc-may-fail --verbosity 8 --unwind 3 --trace --json-ui`' has been used for every prompting method. As the updated version of CBMC was released, new flags were used to restore the default value of the previous version of CBMC.

2.4 Benchmarks

The benchmarks were collected from real-world datasets on Kaggle, primarily related to LeetCode problems. All programs were written in C, using only standard libraries to avoid errors.

Prompt Method	Project	Success	#Iteration	Runtime	Memory Usage	LoC	#Functions
One shot	prompt.c	True	2	28.7236	90093	42	5
Few shots	prompt.c	False	3	42.7236	318866	42	5
CoT	prompt.c	True	1	4.8829	81507	42	5

Table 1: Benchmark details and format

Details on the benchmark are provided in Table 1. The verification result, indicated by 'Success', is marked as 'True' for successful verification from the CBMC and 'False' otherwise. The total number of iterations for each project is listed under '#Iteration'. The iteration count is limited to three; projects that reached the maximum iteration number without 'True' success result are considered unsuccessful. The total runtime and memory usage are reported by 'Runtime' and 'Memory Usage' in seconds and bytes, respectively. The runtime and memory usage have not been averaged to better reflect the LLM's capability to generate correct and safe C codes in a given number of iteration.

3 Prompting Methods

The following are the three different sample prompt formats: one-shot prompting, few-shots prompting, and Chain-of-Thought prompting, to query Gemini Flash 1.5 for generating proof harness functions. New constraint regarding library declarations has been added due to frequent failures in proof harness function generation, 'assert.h' in particular.

3.1 One-shot Prompting

```
# Preamble
  You are given a C program. We need to create a proof harness function.
  # Code generation example - You are given three examples.

  Q: Write a method "void proof_harness_withdraw()" that tests method withdraw below for all possible
  inputs.

  // Define the Account structure
  struct Account {{
      unsigned short bal;
  }};

  // Function to withdraw an amount from an account
  void withdraw(struct Account *account, unsigned short amount) {{
      unsigned short de = account->bal;
      account->bal = de - amount;
  }}

  A:
  struct Account {{
  unsigned short bal;
      }};


  void withdraw(struct Account *account, unsigned short amount) {{
      unsigned short de = account->bal;
      account->bal = de - amount;
  }}

  void proof_harness_withdraw() {{
      struct Account *account = (struct Account *)malloc(sizeof(struct Account));
      __CPROVER_assume(account != NULL);  // Ensure account is not NULL

      unsigned short amount;

      __CPROVER_assume(account->bal >= 0);
      __CPROVER_assume(amount > 0);
      __CPROVER_assume(account->bal >= amount);

      unsigned short initial_balance = account->bal;

      withdraw(account, amount);
      assert(account->bal == initial_balance - amount);

      free(account);
  }}


  # Instruction
  Give me a proof harness code of the below C code.

  # Query
  Q: Write method "void proof_harness_{{function_name}}()" that tests every methods including 'main()'
  below for all possible inputs. Then write a method "void combined_proof_harness()" that calls every
  other proof
  harness methods generated.

  {query_code}.

  # Constraints
```

```
  Here are some constraints that you should respect:
  - Give me only the translated code, don't add explanations or anything else.
  - Use only safe C.
  - Do not use custom generics. # fuzzer limitation
  - Do not remove any code from the original code you have received.
  - Ensure that all libraries needed, including <assert.h>, are declared at the beginning of the code.
```

Listing 1: One-shot Prompting Format

3.2 Few-shots prompting

```

# Preamble
  You are given a C program. We need to create a proof harness function.

  # Code generation example
  ## Example 1
      Q: Write a method "void proof_harness_withdraw()" that tests method withdraw below for all
  possible inputs.
      // Define the Account structure
      struct Account {{
          unsigned short bal;
      }};

      // Function to withdraw an amount from an account
      void withdraw(struct Account *account, unsigned short amount) {{
          unsigned short de = account->bal;
          account->bal = de - amount;
      }}

      A:
      struct Account {{
      unsigned short bal;
          }};


      void withdraw(struct Account *account, unsigned short amount) {{
          unsigned short de = account->bal;
          account->bal = de - amount;
      }}

      void proof_harness_withdraw() {{
          struct Account *account = (struct Account *)malloc(sizeof(struct Account));
          __CPROVER_assume(account != NULL);  // Ensure account is not NULL

          unsigned short amount;

          __CPROVER_assume(account->bal >= 0);
          __CPROVER_assume(amount > 0);
          __CPROVER_assume(account->bal >= amount);

          unsigned short initial_balance = account->bal;

          withdraw(account, amount);
          assert(account->bal == initial_balance - amount);

          free(account);
      }}
## end Example 1

## Example 2
  Q: Write method "void proof_harness_{{function_name}}()" that tests every methods below for all
  possible inputs.
       #include <stdio.h>
  #include <stdlib.h>

  struct node{{
      struct node *leftNode;
```

```
        int data;
        struct node *rightNode;
    }};

    struct node *newNode(int data){{
        struct node *node = (struct node *)malloc(sizeof(struct node));
        node->leftNode = NULL;
        node->data = data; node->rightNode = NULL;
        return node;
    }}

    int main(void){{
        return 0;
    }}

    A:
    #include <stdio.h>
    #include <stdlib.h>
    #include <assert.h>

    struct node{{
        struct node *leftNode;
        int data;
        struct node *rightNode;
    }};

    struct node *newNode(int data){{
        struct node *node = (struct node *)malloc(sizeof(struct node));
        node->leftNode = NULL;
        node->data = data; node->rightNode = NULL;
        return node;
    }}

    int main(void){{
        return 0;
    }}

    void proof_harness_newNode() {{
        int data;
        __CPROVER_assume(data >= 0);
        __CPROVER_assume(data <= 2147483647);
        struct node *node = newNode(data);
        assert(node != NULL);
        assert(node->leftNode == NULL);
        assert(node->data == data);
        assert(node->rightNode == NULL);
        free(node);
    }}

## end Example 2

## Example 3
   Q: Write method "void proof_harness_{{function_name}}()" that tests every methods below for all
  possible inputs.

        #include <stdio.h>

    struct node{{
        struct node *leftNode;
        int data;
        struct node *rightNode;
    }};

    void inOrderTraversal(struct node *node){{
        if (node == NULL) return;
        inOrderTraversal(node->leftNode);
        printf("\t%d\t", node->data);
        inOrderTraversal(node->rightNode);
```

```
}}

void preOrderTraversal(struct node *node){{
    if (node == NULL) return;
    printf("\t%d\t", node->data);
    preOrderTraversal(node->leftNode);
    preOrderTraversal(node->rightNode);
}}

void postOrderTraversal(struct node *node){{
    if (node == NULL) return;
    postOrderTraversal(node->leftNode);
    postOrderTraversal(node->rightNode);
    printf("\t%d\t", node->data);
}}

int main(void){{
    return 0;
}}

A:

#include <stdio.h>
    #include <assert.h>

struct node{{
    struct node *leftNode;
    int data;
    struct node *rightNode;
}};

void inOrderTraversal(struct node *node){{
    if (node == NULL) return;
    inOrderTraversal(node->leftNode);
    printf("\t%d\t", node->data);
    inOrderTraversal(node->rightNode);
}}

void preOrderTraversal(struct node *node){{
    if (node == NULL) return;
    printf("\t%d\t", node->data);
    preOrderTraversal(node->leftNode);
    preOrderTraversal(node->rightNode);
}}

void postOrderTraversal(struct node *node){{
    if (node == NULL) return;
    postOrderTraversal(node->leftNode);
    postOrderTraversal(node->rightNode);
    printf("\t%d\t", node->data);
}}

int main(void){{
    return 0;
}}

void proof_harness_inOrderTraversal() {{
    struct node *node = (struct node *)malloc(sizeof(struct node));
    __CPROVER_assume(node != NULL);
    node->leftNode = NULL;
    node->rightNode = NULL;
    __CPROVER_assume(node->data >= 0 && node->data <= 100);
    inOrderTraversal(node);
    free(node);
}}

void proof_harness_preOrderTraversal() {{
    struct node *node = (struct node *)malloc(sizeof(struct node));
```

```
        __CPROVER_assume(node != NULL);
        node->leftNode = NULL;
        node->rightNode = NULL;
        __CPROVER_assume(node->data >= 0 && node->data <= 100);
        preOrderTraversal(node);
        free(node);
    }}

    void proof_harness_postOrderTraversal() {{
        struct node *node = (struct node *)malloc(sizeof(struct node));
        __CPROVER_assume(node != NULL);
        node->leftNode = NULL;
        node->rightNode = NULL;
        __CPROVER_assume(node->data >= 0 && node->data <= 100);
        postOrderTraversal(node);
        free(node);
    }}

  ## end Example 3

# Instruction
    Give me a proof harness code of the below C code.

# Query
    Q: Write method "void proof_harness_{{function_name}}()" that tests every methods including 'main()'
    below for all possible inputs. Then write a method "void combined_proof_harness()" that calls every
    other proof
    harness methods generated.

    {query_code}

    # Constraints
    Here are some constraints that you should respect:
    - Give me only the translated code, don't add explanations or anything else.
    - Use only safe C.
    - Do not use custom generics. # fuzzer limitation
    - Do not remove any code from the original code you have received.
    - Ensure that all libraries needed, including <assert.h>, are declared at the beginning of the code.
```

Listing 2: Prompt Format

The prompt has been set to include three examples to prevent excessive length and minimise the risk of hallucinations from the LLM. As it will be demonstrated in the results, for few-shot prompting, which is the longest prompt format, longer codes significantly increase the likelihood of hallucinations and failure rates.

3.3 Chain-of-Thought (CoT)

```
  # Preamble
    You are given a C program. We need to create a proof harness function.
    # Code generation example
    Q: Write a method "void proof_harness_withdraw()" that tests method withdraw below for all possible
    inputs.

    // Define the Account structure
    struct Account {{
        unsigned short bal;
    }};

    // Function to withdraw an amount from an account
    void withdraw(struct Account *account, unsigned short amount) {{
        unsigned short de = account->bal;
        account->bal = de - amount;
    }}

    A: You first identify which functions there are. In this case, we have a single function called '
    withdraw'. Since
```

```
    withdraw is a function that needs to be tested, we create a proof_harness_withdraw function for this
     purpose and
    implement to test the correctness of the function 'withdraw'.

    struct Account {{
    unsigned short bal;
        }};


    void withdraw(struct Account *account, unsigned short amount) {{
        unsigned short de = account->bal;
        account->bal = de - amount;
    }}

    void proof_harness_withdraw() {{
        struct Account *account = (struct Account *)malloc(sizeof(struct Account));
        __CPROVER_assume(account != NULL);  // Ensure account is not NULL

        unsigned short amount;

        __CPROVER_assume(account->bal >= 0);
        __CPROVER_assume(amount > 0);
        __CPROVER_assume(account->bal >= amount);

        unsigned short initial_balance = account->bal;

        withdraw(account, amount);
        assert(account->bal == initial_balance - amount);

        free(account);
    }}


    # Instruction
    Give me a proof harness code of the below C code.

    # Query
    Q: Write method "void proof_harness_{{function_name}}()" that tests every methods including 'main()'
    below for all possible inputs. Then write a method "void combined_proof_harness()" that calls every
     other proof
    harness methods generated.

    {query_code}

    # Constraints
    Here are some constraints that you should respect:
    - Give me only the translated code, don't add explanations or anything else.
    - Use only safe C.
    - Do not use custom generics. # fuzzer limitation
    - Do not remove any code from the original code you have received.
    - Ensure that <assert.h> is declared when using assert.
    """
```

Listing 3: Generated Proof Harness Function

4 Results

I run an experiment for each of the three prompts for a total of 366 proof harness functions generation and correctness verification experiment. The input C programs were validated to ensure them using safe C codes by identifying those that returned conversion errors and the pre-processing them accordingly. The results have been rounded to four decimal places.

4.1 Success Rate with regards to Line of Code and number of functions

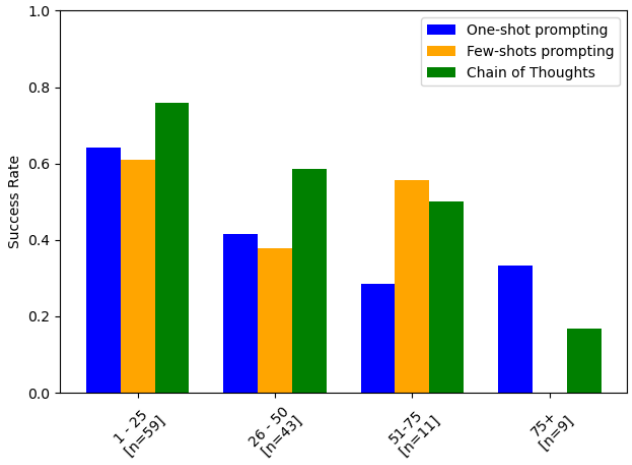

Figure 1: Success Rate of Prompting Methods by Line of Code

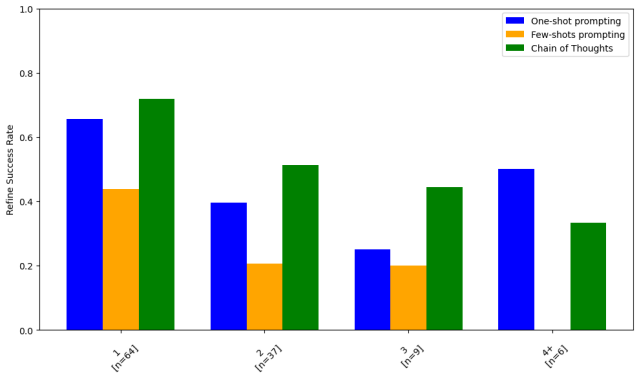

Figure 2: Success Rate of Prompting Methods by Number of Functions

Prompting methods \ LoC	1 ~ 25	26 - 50	51-75	75+
One-shot prompting	35.9375	58.5366	71.4286	66.6667
Few-shots prompting	38.8889	62.0690	44.4444	100
Chain of Thoughts	24.0741	41.4634	50	83.3333

Table 2: Failure Rates of Prompting Methods by Line of Code (percentage)

Figure 1 illustrates the success rates of different prompting methods in relation to the length of the input code, measured in lines of code. The success rates are presented for four ranges of code length: 1-25 lines, 26-50 lines, 51-75 lines, and 75+ lines. The results indicate a general trend where longer prompts and more extensive input code correlate with lower success rates. Specifically:

- **One-shot prompting** demonstrates a significant decrease in success rate as the line of code increase. It achieves its highest success rate of 64.06% with the shortest code line group and drops to its lowest rate of 28.57% within the 51-75 lines range. Regarding the number of functions, one-shot prompting shows consistent performance across different numbers of functions, with the highest success rate observed for single-function code (64.71%) and the lowest for code with three functions (33.33%).
- **Few-shots prompting** exhibits its highest success rate of 61.11% with 1-25 lines and the lowest with the longest line of codes. Notably, there exists an anomaly in the 51-75 lines group, where the success rate increases by approximately 18.25% from the previous group. This indicates that the few-shots prompting performs better performance with moderately long code. Apart from the line of code, few-shots prompting shows lower performance as the number of functions increases. Its success rate significantly drops compared to other prompting methods, with the lowest performance observed for code with four or more functions (16.67%).
- **Chain-of-Thought prompting** maintains the highest success rates among the tested prompting methods for the first three groups. Similarly to other prompting methods, the performance gradually declines for longer code, but shows a highest average success rate among all three. In terms of the number of functions, Chain-of-Thought consistently shows high performance across different function groups. Although there is a decline in success rate as the number of functions increases, Chain-of-Thought remains the most effective method compared to the others.

4.2 Refine Success Rate

One-shot prompting	Few-shots prompting	CoT prompting
5.8333 %	1.8692 %	3.7383 %

Table 3: Refine success rate

Adopting the 'simple restart' strategy, the success rate of refined proof harness function verification was much lower than expected for all three methods. One-shot prompting showed the best ability to return valid proof harness functions, followed by Chain-of-Thoughts prompting with a 3.74% lower success rate. The few-shots prompting method demonstrated the lowest probability of fixing bugs in its proof harness function generation.

4.3 Average Runtime and Memory Usage

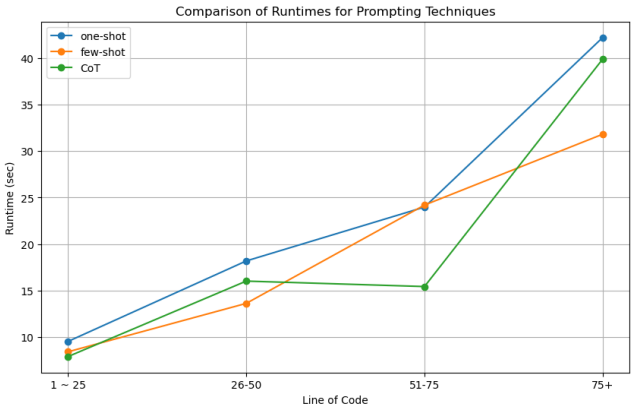

Figure 3: Average Runtime (sec)

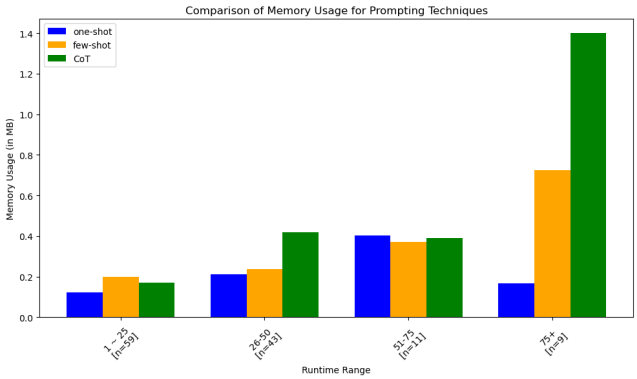

Figure 4: Average Memory Usage (MB)

Before the results are delved, it must be noted that the Chain-of-Thought prompting strategy had two extremely large outliers, which caused the average runtime to increase dramatically. The runtime and memory usage figure has been extracted from the very start until the program finished its final iteration.

The overall average runtime increased as the length of the input code got longer, as well as the memory usage for all three prompting strategies. One-shot prompting shows a gradual increase in runtime as the line of code increases, with a significant jump for the longest code group (75+ lines). Few-shots prompting maintains a relatively stable runtime, with it showing almost a linear relationship between the line of code and the runtime. Chain-of-Thought (CoT) prompting has the shortest runtime for most code lengths but also exhibits a sharp increase for the longest code group, potentially influenced by two extreme outliers. In terms of memory usage, one-shot prompting displays the lowest memory usage across all code lengths, few-shots prompting has moderate memory usage increasing steadily with longer code, and Chain-of-Thought (CoT) prompting shows the highest memory usage, especially for the longest code group, indicating a significant increase in resource consumption.

5 Conclusion

The comparative analysis of different prompting methods for proof harness function verification using Gemini Flash 1.5 reveals significant insights into their performance. One-shot prompting emerged as the most effective method in terms of both success rate and resource efficiency, although its performance notably decreased with longer lines of code and multiple functions. Chain-of-Thought prompting, despite having two large outliers that affected the average runtime, maintained a high success rate across varying code lengths and function numbers, showing consistent performance. Few-shots prompting, while relatively stable in runtime, demonstrated the lowest success rate and the highest memory usage, particularly as the code length increased. Given these results, when LLMs are not provided with feedback from previous failures, they are more likely to succeed if the input is of moderate length and contains fewer functions. The 'simple restart' strategy yielded lower-than-expected success rates for all three methods, with One-shot prompting leading, followed by Chain-of-Thought prompting, and Few-shots prompting trailing behind. Overall, these findings suggest that One-shot prompting is generally preferable for tasks requiring efficient and reliable proof harness function verification, while Chain-of-Thought prompting offers a balanced alternative, especially when runtime and memory usage are less of a concern.

6 Further Discussions

6.1 Using Different Feedback Strategies

Since the current feedback strategy does not show commendable performance in generating valid and safe proof harness functions, trying different feedback strategies such as providing counterexamples could be considered. Incorporating counterexamples into the

feedback prompt may help in guiding the model to correct its previous mistakes more effectively, potentially increasing the overall success rate and robustness of the generated proof harness functions. By leveraging more detailed and constructive feedback, it may be possible to enhance the model's learning process and improve the quality of its outputs.