

Proof Harness Generation for C-programs: LLMs and CBMC

Hyoyeon Lee¹

¹ School of Computer Science, University of Bristol

This document provides a result of proof harness function generation for C programs using the given template and example, intended for use in verifying the correctness of C programs. The created proof harness function has been fed to the CBMC (C Bounded Model Checker) to prove its correctness. If any errors are returned from CBMC, the errors are fed back to the LLMs until no errors are returned.

Query Format

```
1 # Preamble
2 You are given a C program. We need to create a proof harness function.
3
4 # Code generation example
5 Q: Write a method "void proof_harness_withdraw()" that tests method withdraw below for all possible
   inputs.
6
7 // Define the Account structure
8 struct Account {
9     unsigned short bal;
10 };
11
12 // Function to withdraw an amount from an account
13 void withdraw(struct Account *account, unsigned short amount) {
14     unsigned short de = account->bal;
15     account->bal = de - amount;
16 }
17
18 A:
19 void proof_harness_withdraw() {
20     struct Account *account;
21     int amount;
22     CPROVER_assume(account->bal >= 0);
23     CPROVER_assume(amount > 0);
24     CPROVER_assume(account->bal >= amount);
25
26     // Save the initial balance for verification
27     unsigned short initial_balance = account->bal;
28
29     withdraw(account, amount);
30
31     // Check that the new balance is the expected value after withdrawal
32     assert(account->bal == initial_balance - amount);
33 }
34
35 # Instruction
36 Give me a proof harness code of the below C code.
37
38 # Query
39 Q: Write method "void proof_harness_newNode()" that tests method transfer below for all possible inputs.
40 // Insert your code here
41
42 # Constraints
43 Here are some constraints that you should respect:
44 - Give me only the translated code, dont add explanations or anything else.
45 - Use only safe C.
46 - Do not use custom generics. # fuzzer limitation
47 - Make sure there is an entry point in main.
```

Listing 1: Preamble and Example

Due to recurring errors from the absence of an entry point in the main function, a new constraint has been added accordingly. The language models used are ChatGPT-4 and Gemini Pro, with code examples sourced from Kaggle.

ChatGPT4

Example 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct node{
5     struct node *leftNode;
6     int data;
7     struct node *rightNode;
8 };
9
10 struct node *newNode(int data){
11     struct node *node = (struct node *)malloc(sizeof(struct node));
12
13     node->leftNode = NULL;
14     node->data = data;
15     node->rightNode = NULL;
16
17     return node;
18 }
19
20 int main(void){
21     return 0;
22 }
```

Listing 2: Code for prompt

```
1 void proof_harness_Node(){
2     int data;
3     struct node *n = newNode(data);
4
5     assert(n != NULL);
6     assert(n->leftNode == NULL);
7     assert(n->data == data);
8     assert(n->rightNode == NULL);
9 }
10
11 int main(void)
12 {
13     proof_harness_Node();
14     return 0;
15 }
```

Listing 3: Generated proof harness function

```
1 Violated property:
2 function __CPROVER__start thread 0
3 dynamically allocated memory never freed in __CPROVER_memory_leak == NULL
4 __CPROVER_memory_leak == NULL
5
6 ** 1 of 73 failed (2 iterations)
7 VERIFICATION FAILED
```

Listing 4: 1st CBMC result: Verification Failed

The CBMC has been run with the command: `cbmc newNode.c -function proof_harness_Node -memory-leak-check -bounds-check -trace -pointer-check`. The error prompt has been fed back to the LLM until we obtain successful verification.

```
1 void proof_harness_Node() {
2     int data;
3     __CPROVER_assume(data >= 0);
4     struct node *node = newNode(data);
5     assert(node != NULL);
6     assert(node->data == data);
7     assert(node->leftNode == NULL);
8     assert(node->rightNode == NULL);
9 }
```

```

9
10     free(node);
11 }

```

Listing 5: Final proof harness function: Successful Verification

Example 2

```

1  #include <stdio.h>
2
3  void inOrderTraversal(struct node *node){
4      if (node == NULL)          return;
5
6      inOrderTraversal(node->leftNode);
7      printf("\t%d\t", node->data);
8      inOrderTraversal(node->rightNode);
9  }
10
11 void preOrderTraversal(struct node *node){
12     if (node == NULL)          return;
13
14     printf("\t%d\t", node->data);
15     preOrderTraversal(node->leftNode);
16     preOrderTraversal(node->rightNode);
17 }
18
19 void postOrderTraversal(struct node *node){
20     if (node == NULL)          return;
21
22     postOrderTraversal(node->leftNode);
23     postOrderTraversal(node->rightNode);
24     printf("\t%d\t", node->data);
25 }
26
27 int main(void){
28     return 0;
29 }

```

Listing 6: Code fed as a prompt

```

1  void proof_harness_traversal() {
2      struct node *root = malloc(sizeof(struct node));
3      struct node *left = malloc(sizeof(struct node));
4      struct node *right = malloc(sizeof(struct node));
5
6      __CPROVER_assume(root != NULL);
7      __CPROVER_assume(left != NULL);
8      __CPROVER_assume(right != NULL);
9
10     root->data = 1;
11     root->leftNode = left;
12     root->rightNode = right;
13
14     left->data = 2;
15     left->leftNode = NULL;
16     left->rightNode = NULL;
17
18     right->data = 3;
19     right->leftNode = NULL;
20     right->rightNode = NULL;
21
22     inOrderTraversal(root);
23     preOrderTraversal(root);
24     postOrderTraversal(root);
25
26     free(root);

```

```

27     free(left);
28     free(right);
29 }

```

Listing 7: Generated proof harness function: Successful Verification

Example 3

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Node{
5      int data;
6      struct Node *llink;
7      struct Node *rlink;
8  } node;
9
10 node *create_node(int data){
11     node *ptr = (node *)malloc(sizeof(node));
12     ptr->rlink = ptr->llink = NULL;
13     ptr->data = data;
14     return ptr;
15 }
16
17 void insert_bt(node **root, int data){
18     node *new_node = create_node(data);
19     node *temp;
20     node *prev;
21     if (*root == NULL){
22         *root = new_node;
23     }
24     else{
25         temp = *root;
26         prev = NULL;
27         while (temp != NULL){
28             if (new_node->data > temp->data){
29                 prev = temp;
30                 temp = temp->rlink;
31             }
32             else if (new_node->data < temp->data){
33                 prev = temp;
34                 temp = temp->llink;
35             }
36             else{
37                 return;
38             }
39         }
40
41         if (new_node->data > prev->data){
42             prev->rlink = new_node;
43         }
44         else{
45             prev->llink = new_node;
46         }
47     }
48 }
49
50 void search(node *root, int ele){
51     node *temp = root;
52     while (temp != NULL){
53         if (temp->data == ele){
54             break;
55         }
56         else if (ele > temp->data){
57             temp = temp->rlink;
58         }
59         else{
60             temp = temp->llink;
61         }
62     }
63
64     if (temp == NULL){
65         printf("%s\n", "Element not found.");
66     }
67     else printf("%s\n", "Element found.");
68 }
69
70 void inorder_display(node *curr){
71     if (curr != NULL){
72         inorder_display(curr->llink);
73         printf("%d\t", curr->data);
74         inorder_display(curr->rlink);
75     }
76 }
77
78 void postorder_display(node *curr){
79     if (curr != NULL){
80         postorder_display(curr->llink);
81         postorder_display(curr->rlink);
82         printf("%d\t", curr->data);
83     }
84 }
85
86 void preorder_display(node *curr){
87     if (curr != NULL){
88         printf("%d\t", curr->data);
89         preorder_display(curr->llink);
90         preorder_display(curr->rlink);
91     }
92 }
93
94 void delete_bt(node **root, int ele){
95     node *temp;
96     node *prev;
97     if (*root == NULL)
98         return;
99     else{
100         temp = *root;
101         prev = NULL;
102         while (temp != NULL){
103             if (temp->data == ele){
104                 break;
105             }
106             else if (ele > temp->data){
107                 prev = temp;
108                 temp = temp->rlink;
109             }
110             else{
111                 prev = temp;
112                 temp = temp->llink;
113             }
114         }

```

```

115 }
116
117 if (temp == NULL)
118     return;
119 else
120 {
121     node *replacement;
122     node *t;
123     if (temp->llink == NULL && temp->rlink ==
124         NULL){
125         replacement = NULL;
126     }
127     else if (temp->llink == NULL && temp->
128         rlink != NULL){
129         replacement = temp->rlink;
130     }
131     else if (temp->llink != NULL && temp->
132         rlink == NULL){
133         replacement = temp->llink;
134     }
135     else{
136         replacement = temp->rlink;
137         t = replacement;
138         while (t->llink != NULL){
139             t = t->llink;
140         }
141         t->llink = temp->llink;
142     }
143
144     if (temp == *root){
145         free(*root);
146         *root = replacement;
147     }
148     else if (prev->llink == temp){
149         free(prev->llink);
150         prev->llink = replacement;
151     }
152     else if (prev->rlink == temp){
153         free(prev->rlink);
154         prev->rlink = replacement;
155     }
156 }

```

```

154 }
155
156 int main(){
157     node *root = NULL;
158     int choice, n;
159     do{
160         scanf("%d", &choice);
161
162         switch (choice){
163             case 1:
164                 printf("%s\n", "Enter a no:");
165                 scanf("%d", &n);
166                 insert_bt(&root, n);
167                 break;
168             case 2:
169                 inorder_display(root);
170                 printf("\n");
171                 break;
172             case 3:
173                 preorder_display(root);
174                 printf("\n");
175                 break;
176             case 4:
177                 postorder_display(root);
178                 printf("\n");
179                 break;
180             case 5:
181                 printf("%s\n", "Enter a no:");
182                 scanf("%d", &n);
183                 delete_bt(&root, n);
184                 break;
185             case 6:
186                 printf("%s\n", "Enter a no:");
187                 scanf("%d", &n);
188                 search(root, n);
189                 break;
190         }
191     } while (choice != 0);
192     return 0;
193 }

```

Listing 8: Code fed as a prompt

```

1 void proof_harness_newNode() {
2     node *root = NULL;
3     int data;
4
5     CPROVER_assume(data >= -1000 && data <= 1000);
6
7     insert_bt(&root, data);
8
9     assert(root != NULL);
10    node *temp = root;
11    while (temp != NULL) {
12        if (temp->data == data) {
13            break;
14        }
15        if (data > temp->data) {
16            temp = temp->rlink;
17        } else {
18            temp = temp->llink;
19        }
20    }
21    assert(temp != NULL);
22    assert(temp->data == data);
23
24    free_tree(root);

```

```
25 }
```

Listing 9: Generated Proof Harness Function

The initial proof harness function developed for our project had a memory leak issue. To address this, we used the error feedback to iteratively refine the function. This approach led to the creation of a final proof harness function that resolved the memory leak problem effectively.

Gemini Pro

The following are the proof harness functions generated by Gemini Pro.

Example 1

```
1 void proof_harness_newNode(int data) {
2     struct node *n = newNode(data);
3     assert(n->leftNode == NULL);
4     assert(n->data == data);
5     assert(n->rightNode == NULL);
6     free(n);
7 }
```

Listing 10: Generated Proof Harness Function: Verification Successful

CBMC passed with a successful verification without any memory leaks, etc.

Example 2

```
1 void proof_harness_traversal() {
2     struct node *root = newNode(1);
3     root->leftNode = newNode(2);
4     root->rightNode = newNode(3);
5     root->leftNode->leftNode = newNode(4);
6     root->leftNode->rightNode = newNode(5);
7
8     inOrderTraversal(root);
9     preOrderTraversal(root);
10    postOrderTraversal(root);
11
12    freeTree(root); // added
13 }
```

Listing 11: Generated Proof Harness Function: Verification Successful

The initial proof harness function encountered a memory leak error. This issue was addressed by feeding the error details back to the LLM, resulting in a final proof harness function that works effectively.

Example 3

```
1 void proof_harness_traversal() {
2     node *root = NULL;
3     insert_bt(&root, 5);
4     insert_bt(&root, 3);
5     insert_bt(&root, 8);
6     insert_bt(&root, 1);
7     insert_bt(&root, 4);
8     insert_bt(&root, 7);
9     insert_bt(&root, 9);
10
11    inorder_display(root);
12    preorder_display(root);
13    postorder_display(root);
14
15    search(root, 4);
16    delete_bt(&root, 4);
17    search(root, 4);
18 }
```

```

19   free_tree(root);
20 }

```

Listing 12: Generated Proof Harness Function

Example 4

```

1  #include <stdio.h>
2
3  int main(){
4      int t;
5      printf("Enter number of times you want to try");
6      scanf("%d", &t);
7      while (t--) {
8          unsigned long long N, p = 0, sum = 0;
9          printf("Enter the value of N ");
10
11          scanf("%lld", &N);          p = (N - 1) / 3;
12          sum = ((3 * p * (p + 1)) / 2);
13
14          p = (N - 1) / 5;
15          sum = sum + ((5 * p * (p + 1)) / 2);
16
17          p = (N - 1) / 15;
18          sum = sum - ((15 * p * (p + 1)) / 2);
19          printf("%lld\n", sum);
20      }
21      return 0;

```

Listing 13: Code Fed as a Prompt

```

1  #include <stdio.h>
2  #include <assert.h>
3
4  int main() {
5      int t = nondet_int();
6      __CPROVER_assume(t > 0 && t < 10);
7      while (t--) {
8          unsigned long long N = nondet_ulong_long();
9          __CPROVER_assume(N > 0 && N < 1000000);
10         unsigned long long p, sum;
11
12         p = (N - 1) / 3;
13         sum = ((3 * p * (p + 1)) / 2);
14
15         p = (N - 1) / 5;
16         sum = sum + ((5 * p * (p + 1)) / 2);
17
18         p = (N - 1) / 15;
19         sum = sum - ((15 * p * (p + 1)) / 2);
20     }
21
22     return 0;
23 }

```

Listing 14: Generated Proof Harness Function