Key Observation on the Effectiveness of LLMs in Decompilation

1. Clean Sample (1.3B and 6.7B Versions)

Key Observations:

- Memory Usage: The 1.3B model consumes 4610 MiB while the 6.7B model uses significantly more, 16046 MiB. This is expected, given the larger size of the 6.7B model.
- Decompiled Output Differences:
    - The 1.3B model generates plausible decompiled code, though it occasionally simplifies the function or misses certain details (fabs(a[i] - 1.0f) appearing across all optimizations).
    - The 6.7B model generally produces more accurate results that closely match the structure of the original function, especially for the higher optimization levels (O2 and O3).
    - The function for O3 optimization in the 6.7B model captures more details, such as correct usage of thresh in the final function.

Interpretation:

- Accuracy: The 6.7B model generally produces a more faithful decompilation for complex patterns and higher optimization levels. The 1.3B model is less precise, especially in generating higher-level logic, but is still competent at simpler optimization levels.
- Memory and Performance: The increase in memory usage for the 6.7B model (3.5x more) is expected due to its size, but it yields improved accuracy, especially at higher optimization levels.

2. Complex Sample Output (1.3B and 6.7B Versions)

Key Observations:

- Recursive Function Handling: Both models can generate a recursive version of the function. The 6.7B model shows better handling of recursive function flow, especially for O3, where it correctly decompiles a recursion with conditions on the input array.

- Memory Usage: 1.3B model memory usage is 4718 MiB, while 6.7B model memory is 16252 MiB.
- Errors in Lower Optimizations: At O0 and O1, the models tend to simplify the recursion logic, sometimes duplicating or missing recursive calls. However, the higher optimizations (O2, O3) produce more correct function structures.

Interpretation:

- Accuracy: The 6.7B model significantly outperforms the 1.3B model in decompiling recursive structures accurately. It manages complex logic and dependencies better, especially in higher optimizations.
- Resource Consumption: The 6.7B model's higher memory consumption is justified by its ability to handle recursion and optimize the structure better than the 1.3B model.

3. Incomplete/Malformed Sample (1.3B and 6.7B Versions)

Key Observations:

- Handling of Incomplete Code: Both models struggle with incomplete code but manage to generate some plausible functions. The 6.7B model generates more structured output at all optimization levels (O0-O3), handling the incomplete code better by filling gaps (using fwrite correctly).
- Logic Generation: The 1.3B model introduces some repetition and incorrect logic in its outputs (e.g., repeated calls to fwrite), while the 6.7B model is more accurate in replicating the correct output.

Interpretation:

- Accuracy: The 6.7B model is better equipped to handle incomplete or malformed code. It fills in logical gaps more effectively, although both models struggle to some extent with malformed inputs.
- Hallucinations: Both models show a tendency for hallucination in such cases, where they generate unnecessary or repetitive logic.

4. Obfuscated Sample (1.3B and 6.7B Versions)

Key Observations:

- Handling of Obfuscated Code: Both models struggle with obfuscated inputs. However, the 6.7B model produces more reasonable decompiled functions at higher optimization levels, while the 1.3B model struggles with incorrect logic (repetitive fabs(a[0] - 1.0) conditions).

- Logic Simplification: The models simplify obfuscated code by breaking it into its logical components, with the 6.7B model preserving more of the structure and correctly capturing the intent of the code.

Interpretation:

- Accuracy: The 6.7B model is better at decompiling obfuscated code, maintaining more of the original logic and structure, though both models introduce simplifications.
- Complexity Handling: Obfuscated input causes issues with both models, though the 6.7B model is better at preserving the logic flow and reducing hallucinations.

5. Perturbed Code Sample (1.3B and 6.7B Versions)

Key Observations:

- Decompiled Logic: Both models handle perturbed inputs reasonably well. The 1.3B model has more difficulty maintaining the precise logic of the function, while the 6.7B model handles it with better accuracy.
- Functionality Preservation: At higher optimization levels (O3), the 6.7B model produces better outputs, maintaining the original function logic. The 1.3B model simplifies some of the conditions unnecessarily.

Interpretation:

- Accuracy: The 6.7B model performs better with perturbed code, preserving the original structure and logic, especially at higher optimization levels.
- Resource Usage: Memory consumption is in line with expectations for both models, with the 6.7B model using more memory but delivering better performance.

IN GENERAL,

1. Memory Usage:
    - The memory usage scales with the size of the model, with the 6.7B model using approximately 3.5x more memory than the 1.3B model across all input types. This increased memory consumption allows for better handling of complex, recursive, and obfuscated code.
2. Accuracy Across Optimization Levels:
    - The 6.7B model consistently performs better across all optimization levels (O0, O1, O2, O3), especially for complex, recursive, and obfuscated code.
    - The 1.3B model performs well at simpler optimization levels (O0 and O1), but struggles with more complex logic and higher optimizations.

3. Hallucinations:
    - Both models show signs of hallucination, particularly with incomplete and obfuscated code. The 6.7B model, while not immune to hallucinations, tends to generate more logical and plausible code.

Clean Code Sample Hallucination:

1.3B Model:

- Example: In the O3 optimization level, the 1.3B model decompiled the function as follows: Hallucination: The original code logic does not include the idea of calculating or returning a "minimum" value. The model hallucinates the logic of finding the smallest value in the array, which is not part of the original func0. The decompiled output fabricates this new functionality instead of simply checking for the proximity of two values.

6.7B Model:

- Example (less severe): Hallucination: The model is slightly closer to the original logic. However, the condition fabsf(x[i] - x[j]) > eps is a hallucination as well. The original intent is to check for values that are *less* than a threshold, but the model reversed the condition to greater than. This subtle hallucination changes the core logic of the function.

2. Complex Code Sample Hallucination:

1.3B Model:

- Example (O1 Optimization): Hallucination: The model introduces repetitive recursive calls like recursiveFunc(a, n, i + 1) unnecessarily. This introduces extra recursion checks that aren't present in the original function. Also, the condition fabs(a[i] - 1.0f) is hallucinated—it is nowhere in the original code, which dealt with comparing values in a different context.

6.7B Model:

- Example (O0 Optimization): Hallucination: Here, the model generates a condition to check if an element is smaller than two previous elements (a[i] < a[i-1] && a[i] < a[i-2]), which doesn't exist in the original recursive function logic. This is an example of a hallucination, where the model fabricates logic that isn't relevant to the task it was supposed to handle.

3. Incomplete/Malformed Code Hallucination:

1.3B Model:

- Example (O2 Optimization): Hallucination: The model introduces a repetitive pattern of checking if arrays a, b, and c are equal using memcmp. This code does not exist in the original incomplete code provided. The original code didn't even use arrays in such a manner, so this output is purely hallucinated logic to fill in gaps due to incomplete inputs.

6.7B Model:

- Example (O1 Optimization): Hallucination: This is a clear hallucination where the model generates a nonsensical series of memcmp operations that compare a floating-point value f to itself. The model introduces a repetitive pattern that doesn't logically belong in the original code. While it tries to generate reasonable behavior, it ends up fabricating an illogical loop with self-comparisons.

4. Obfuscated Code Hallucination:

1.3B Model:

- Example (O1 Optimization): Hallucination: The function generates repetitive and redundant logic involving filling the b array with 1.0 multiple times, without any meaningful difference between these operations. This suggests the model struggles to handle obfuscated inputs, often fabricating unnecessary or repetitive logic.

6.7B Model:

- Example (O3 Optimization): Hallucination (subtle): Although this is a more accurate attempt, the logic of checking differences between consecutive elements (samples[i] - samples[i-1]) and returning 1 if they exceed the threshold is hallucinated. The original code did not specifically focus on this type of comparison; the model introduces this pattern to handle obfuscated input.

5. Perturbed Code Sample Hallucination:

1.3B Model:

- Example (O3 Optimization): Hallucination: The introduction of logic to find the minimum value in the array is another clear hallucination. The original code did not perform any such operation. The perturbed input seems to cause the model to hallucinate this minimum-finding behavior.

ADDITIONS:

INPUT TYPE	OPTIMIZATION TYPE	1.3B MEMORY CONSUMPTION	6.7B MEMORY CONSUMPTION
Clean	O0	4610	16046
Clean	O3	4718	16252
Complex	O0	4718	16252
Complex	O3	4718	16252
Obfuscated	O0	4610	16046
Obfuscated	O3	4610	16046
Perturbed	O0	4610	16046
Perturbed	O3	4610	16046
Malformed	O0	4610	14556
Malformed	O3	4610	14556

1. CLEAN SAMPLE: The original function func0 takes an array of floating-point numbers (numbers[]), a size (size), and a threshold (threshold). It compares every pair of numbers in the array to check if the absolute difference between them is smaller than the threshold. If such a pair is found, the function returns 1; otherwise, it returns 0. This is a well-structured function that checks for proximity between elements in the array using the fabs function from the C standard library, which calculates the absolute difference between floating-point values.

A) 1.3B VERSION RESULTS:

CLEAN CODE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

int func0(float numbers[], int size, float threshold) {
  int i, j;
 	
  for (i = 0; i < size; i++)
  for (j = i + 1; j < size; j++)
  if (fabs(numbers[i] - numbers[j]) < threshold)
  return 1;
 	
  return 0;
}
	static int func0(float a, float b, float c)
	{
	int i, j;
	for (i = 0; i < N; i++)
	for (j = 0; j < N; j++)
	if (fabs(a * i + b * j - c) > 1e-6f)
	return 1;
	return 0;
	}
		int func0(float *a, int n)
		{ int i; float min = 1.0f;
		for (i = 0; i < n; i++)
		if (fabs(a[i] - 1.0f) < fabs(min))
		min = a[i];
		return min;
		}

B) 6.7B VERSION RESULTS:

CLEAN CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

int func0(float numbers[], int size, float threshold) {
  int i, j;
 	
  for (i = 0; i < size; i++)
  for (j = i + 1; j < size; j++)
  if (fabs(numbers[i] - numbers[j]) < threshold)
  return 1;
 	
  return 0;
}
	int
	func0(float *x, int n, float eps)
	{
	  int i, j;
	
	 for (i = 0; i < n; i++)
	  for (j = i + 1; j < n; j++)
	  if (fabsf(x[i] - x[j]) > eps)
	  return 1;
	 return 0;
	}
		static int func0(const float *samples, int num_samples, float thresh)
		{
		  int i;
		
		  for (i = 1; i < num_samples; i++)
		  if (fabsf(samples[i] - samples[i-1]) > thresh)
		  return 1;
		  return 0;
		}

- What Went Right:

a) Accurate Floating-Point Operations: Both models correctly identified that fabs (or fabsf) was used to compare floating-point numbers. This shows the models understood that absolute value calculations are crucial to the function's logic.

b) Loop Structures: The decompiled outputs correctly use looping structures to compare elements in an array, which is central to the original function's behavior.

- What Went Wrong (Hallucinations and Errors):

a) Hallucinations in the 1.3B Model:

  - The 1.3B model introduced significant hallucinations, especially in the O0 output, where variables like a, b, and c are fabricated and mathematical operations (a * i + b * j - c) are introduced. These elements don't exist in the original code, representing a major deviation from the intended behavior.
  - In the O3 output, the hallucination of calculating the minimum value (min = a[i]) is irrelevant and not part of the original function's logic.

b) Inverted Logic in the 6.7B Model:

  - The 6.7B model produced an inversion of the comparison logic (fabsf(x[i] - x[j]) > eps instead of fabs(numbers[i] - numbers[j]) < threshold). While this is a minor issue, it changes the behavior of the function.
  - The model also simplified the logic in the O3 output by comparing consecutive elements rather than all pairs, which, while functional, deviates slightly from the original design.

FINAL THOUGHTS:

1.3B Model: The 1.3B model suffers from significant hallucinations, especially at higher optimization levels. These hallucinations cause the model to introduce entirely new, irrelevant logic that changes the core function of the code.

The performance of the 1.3B model is weaker in terms of maintaining the accuracy of the original logic, especially when the input code increases in complexity.

6.7B Model: The 6.7B model provides a more accurate decompilation overall, with only minor hallucinations or simplifications. The errors it introduces (e.g., inverted comparison or simplifying consecutive comparisons) are relatively minor and do not drastically change the behavior of the code.

This model performs well at both lower and higher optimization levels, making it a more reliable option for decompiling complex code.

Hallucinations and Errors: Hallucinations were present primarily in the 1.3B model and caused severe deviations from the original logic.

The 6.7B model occasionally simplified or inverted logic but remained closer to the intended behavior of the original function.

2. COMPLEX SAMPLE : The original code is a recursive function recursiveFunc that checks if any pair of elements in the array numbers[] are within a certain threshold distance of each other. The function operates recursively by checking each element against the subsequent elements in the array, using the fabs function for comparing floating-point values. If a close pair is found, the function returns 1; otherwise, it continues recursively until the base case is reached.

A) 1.3B VERSION RESULTS:

COMPLEX CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

// Recursive version of the function to check pairwise closeness
int recursiveFunc(float numbers[], int size, float threshold, int index) {
  if (index >= size) return 0; // Base case
 	
  for (int j = index + 1; j < size; j++) {
  if (fabs(numbers[index] - numbers[j]) < threshold) {
	static int recursiveFunc(float *a, int n, float eps)
	  {
	 if (n < eps)
	return 0;
	int i;
	 for (i = 1; i < n; i++)
	if (fabs(a[i] - a[0]) > eps)
	return 1;
	if (recursiveFunc(a, n - 1, eps))
	 return 1;
		int recursiveFunc(float *a, int n, int m)
		{
		 int i;
		 for (i = m; i < n; i++)
		if (a[i] < a[i-1] && a[i] < a[i-2])
		return 1;
		return 0;
		 }

return 1; // Close numbers found
}
}
 	
return recursiveFunc(numbers, size, threshold, index + 1); // Recursive case
}
	if (recursiveFunc(a, n - 1, eps))
	return 1;
	return 0;
	}	

B) 6.7B VERSION RESULTS:

COMPLEX CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

// Recursive version of the function to check pairwise closeness
int recursiveFunc(float numbers[], int size, float threshold, int index) {
if (index >= size) return 0; // Base case
 	
for (int j = index + 1; j < size; j++) {
if (fabs(numbers[index] - numbers[j]) < threshold) {
return 1; // Close numbers found
}
}
 	
return recursiveFunc(numbers, size, threshold, index + 1); // Recursive case
}
	int main()
	{
	float a[6] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f};
	float b[5] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f};
	float c[5] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f};
	
	if (recursiveFunc(a, 6, 0.0f) != 1)
	error("recursiveFunc(a, 6, 0.0f) != 1", "", 21, "");
	if (recursiveFunc(a, 6, 1.0f) != 0)
	error("recursiveFunc(a, 6, 1.0f) != 0", "", 22, "");
	
	if (recursiveFunc(b, 5, 0.0f) != 1)
	error("recursiveFunc(b, 5, 0.0f) != 1", "", 25, "");
	if (recursiveFunc(b, 5, 1.0f) != 1)
	error("recursiveFunc(b, 5, 1.0f) != 1", "", 26, "");
	
	if (recursiveFunc(c, 5, 0.0f) != 1)
		static int recursiveFunc(float *data, int n, int i, float thresh)
		{
		if (i >= n) return 0;
		if (fabsf(data[i] - data[i+1]) < thresh) return 1;
		return recursiveFunc(data, n, i+1, thresh);
		}

	error("recursiveFunc(c, 5, 0.0f) != 1", "", 29, "");
	if (recursiveFunc(c, 5, 1.0f) != 1)
	error("recursiveFunc(c, 5, 1.0f) != 1", "", 30, "");
	return 0;
	}	

- 1.3B Model Issues:

a) Hallucinations: In the O0 output, the fabricated base case (if (n < eps) return 0) and redundant recursive calls are significant hallucinations. These elements deviate from the original function's logic, which had a clear base case (index >= size) and only a single recursive call.

In the O3 output, the hallucinated comparison a[i] < a[i-1] && a[i] < a[i-2] has no basis in the original function. This deviation makes the output code almost entirely incorrect.

b) Structural Deviations: The removal of the recursive structure in the O3 output is a major issue. The function no longer behaves as a recursive function, which was key to the original logic. Instead, it becomes an iterative function with incorrect comparisons.

- 6.7B Model Strengths and Weaknesses:
  a) Preserving Recursion: The 6.7B model is much closer to the original in both O0 and O3 optimizations. It correctly identifies and preserves the recursive structure, which is central to the original function. The base case (if (i >= n) return 0) and recursive call (recursiveFunc(data, n, i+1, thresh)) are correctly decompiled in the O3 output.
  b) Simplifications: In the O3 output, the model simplifies the logic by only comparing consecutive elements (data[i] - data[i+1]). This deviates from the original function's goal of comparing each element with subsequent elements, slightly altering the logic.
  c) Hallucinations in O0: The embedding of the recursive function within a main function, as seen in the O0 output, is a hallucination. The additional error-checking logic (error(...)) is unnecessary and irrelevant to the task at hand.

FINAL THOUGHTS:

- 1.3B Model: The 1.3B model introduces substantial hallucinations and fails to capture the recursive nature of the original function, especially at higher optimization levels. The significant deviations in logic and repeated recursive calls make this model unreliable for decompiling complex recursive functions.
- 6.7B Model: The 6.7B model performs significantly better, especially at the O3 optimization level, where it retains the recursive structure and closely mimics the original

logic. However, some simplifications (such as comparing consecutive elements) slightly alter the behavior of the function.

- Hallucinations in the O0 output (e.g., embedding the function in a main test case) should be noted, but they do not drastically affect the recursive logic itself.

OBFUSCATED SAMPLE:

1.3B VERSION RESULTS: The original function f checks whether any two elements in the array X[] differ by less than a threshold Z. The main logic involves two nested loops that compare each element with subsequent elements using the fabs function (absolute value of the difference between two floating-point numbers). The function returns 1 if a close pair is found, and 0 otherwise. Additionally, the main function tests this logic with various arrays and asserts that the output matches the expected results.

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

int f(float X[],int Y,float Z){
int A,B;
for(A=0;A<Y;A++)
for(B=A+1;B<Y;B++)
if(fabs(X[A]-
X[B])<Z)
return 1;
return 0;
}
 	
int main(){
float
e[]={1.0,2.0,3.9,4.0,5.0,2.2};
assert(f(e,6,0.3)==1);
assert(f(e,6,0.05)==0);
 	
float
g[]={1.0,2.0,5.9,4.0,5.0};
assert(f(g,5,0.95)==1);
assert(f(g,5,0.8)==0);
 	
float
h[]={1.0,2.0,3.0,4.0,5.0};
assert(f(h,5,2.0)==1);
 	
float
i[]={1.1,2.2,3.1,4.1,5.1};
assert(f(i,5,1.0)==1);
assert(f(i,5,0.5)==0);
	int f(float a, float b, float c)
	{
	int i, j;
	float x[6], y[6];
	for (i = 0; i < 6; i++) {
	for (j = 0; j < 6; j++) {
	x[j] = a + j * 0.01; y[j] = b + i
	* 0.01;
	}
	if (f(x, y, c) != 1) return 1;
	}
	return 0; }
		int f(float *a, int n) { int i;
		float d = 1.0;
		for (i = 0; i < n; i++)
		if (a[i] < d)
		d = a[i];
		return d; }

return 0; }		

6.7B VERSION RESULTS:

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

int f(float X[],int Y,float Z){
int A,B;
for(A=0;A<Y;A++)
for(B=A+1;B<Y;B++)
if(fabs(X[A]-X[B])<Z)
return 1;
return 0;
}
 	
int main(){
float e[]={1.0,2.0,3.9,4.0,5.0,2.2};
assert(f(e,6,0.3)==1);
assert(f(e,6,0.05)==0);
 	
float g[]={1.0,2.0,5.9,4.0,5.0};
assert(f(g,5,0.95)==1);
assert(f(g,5,0.8)==0);
 	
float h[]={1.0,2.0,3.0,4.0,5.0};
assert(f(h,5,2.0)==1);
 	
float i[]={1.1,2.2,3.1,4.1,5.1};
assert(f(i,5,1.0)==1);
assert(f(i,5,0.5)==0);
 	
return 0;
}
	int f(float *x, int n, float eps)
	{
	int i, j;
	
	for (i = 0; i < n; i++) {
	for (j = i + 1; j < n; j++)
	{
	if (fabsf(x[i] - x[j]) > eps)
	return 1;
	}
	}
	return 0;
	}
		static int f(float *samples, int nb_samples, float thres)
		{
		int i;
		
		for (i = 1; i < nb_samples; i++)
		if (fabsf(samples[i] - samples[i-1]) > thres)
		return 1;
		return 0;
		}

1.3B Model Issues:

a) Hallucinations: In both the O0 and O3 decompiled outputs, the 1.3B model introduces hallucinations that drastically alter the logic of the function.

The variables a, b, x[], and y[] do not exist in the original code. The addition of new, irrelevant calculations (a + j * 0.01, b + i * 0.01) completely changes the purpose of the function.

The introduction of a recursive call in O0 is another significant hallucination, as the original function has no recursive behavior.

b) Loss of Core Logic: In the O3 output, the function has been transformed into a minimum-value calculation (d = a[i]). This is entirely unrelated to the original function's task of comparing pairs of elements using fabs.

The 1.3B model fails to preserve the original structure and logic of the function, making it unreliable for decompiling this obfuscated code.

6.7B Model Strengths and Weaknesses:

a. Preservation of Logic: The 6.7B model retains the key aspects of the original function. The nested loop structure, floating-point comparisons, and return conditions are all preserved in both the O0 and O3 decompiled outputs.

Despite being obfuscated, the model successfully decompiles the code with minimal hallucinations and stays close to the original logic.

b. Logic Inversion: Both the O0 and O3 outputs from the 6.7B model invert the comparison logic. Instead of checking if the absolute difference is less than the threshold (< thres), the decompiled code checks if the difference is greater than the threshold (> thres). This changes the behavior of the function but is a relatively minor issue that can be corrected.
c. Simplification: in the O3 output, the model simplifies the logic by only comparing consecutive elements (samples[i] and samples[i-1]). This is a deviation from the original function, which compares each element with subsequent elements. While this simplifies the code, it reduces the thoroughness of the pairwise comparison.

FINAL THOUGHTS:

1.3B Model: The 1.3B model suffers from significant hallucinations, introducing new variables and logic that do not exist in the original code.

The core logic is lost, especially in the O3 output, where the function has been transformed into a minimum-value calculation. This makes the 1.3B model unsuitable for decompiling obfuscated code like this.

6.7B Model: The 6.7B model performs significantly better, retaining the core logic of the function and accurately handling the nested loops and floating-point comparisons.

Logic Inversion: The inversion of the comparison condition is a minor issue that can be easily corrected.

Simplification: The simplification of logic in the O3 output (comparing consecutive elements only) deviates from the original intent but is still logically valid within the context of the function.

4. **MALFORMED/INCOMPLETE SAMPLE:** The original code provided is incomplete. It includes a declaration for the function func0, which takes an array of floats (numbers[]), an integer size, and a float threshold but is missing the function body. The main function calls func0 several times with different inputs, using assert to check if the return values are correct. However, since the function body of func0 is missing, this would result in compilation or runtime errors in a real-world scenario.

1.3B VERSION RESULTS:

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
 	
// Function declaration but missing body
int func0(float numbers[], int size, float threshold);
 	
int main(){
  float a[] = {1.0, 2.0, 3.9, 4.0, 5.0, 2.2};
  assert(func0(a, 6, 0.3) == 1);
  assert(func0(a, 6, 0.05) == 0);
 	
  float b[] = {1.0, 2.0, 5.9, 4.0, 5.0};
  assert(func0(b, 5, 0.95) == 1);
  assert(func0(b, 5, 0.8) == 0);
 	
  return 0;
}
 	
// Missing function body
	int main(void)
	{
	float f = 1.0f; double d = 1.0;
	int i = 1;
	char c = 'a';
	char *p = "abc";
	if (fwrite(&f, sizeof(f), 1, stdout) != 1)
	perror("fwrite");
	if (fwrite(&d, sizeof(d), 1, stdout) != 1)
	perror("fwrite");
	if (fwrite(&i, sizeof(i), 1, stdout) != 1)
	perror("fwrite");
	if (fwrite(&c, sizeof(c), 1, stdout) != 1)
	perror("fwrite");
	if (fwrite(p, sizeof(p), 1, stdout) != 1)
	perror("fwrite"); return 0;
	}
		int main(void)
		{
		float f = 0.0f; double d = 0.0;
		int i = 0;
		if (fwrite(&f, sizeof(f), 1, stdout) != 1) fail("fwrite", "f", 11, "fwrite");
		if (fwrite(&f, sizeof(f), 1, stdout) != 1) fail("fwrite", "f", 12, "fwrite");
		f = 1.0f; d = 1.0; if (fwrite(&f, sizeof(f), 1, stdout) != 1) fail("fwrite", "f", 13, "fwrite");
		if (fwrite(&f, sizeof(f), 1, stdout) != 1) fail("fwrite", "f", 14, "fwrite"); i = 1; if (fwrite(&i, sizeof(i), 1, stdout) != 1) fail("fwrite", "i", 15, "fwrite");
		if (fwrite(&i, sizeof(i), 1, stdout) != 1) fail("fwrite", "i", 16, "fwrite"); return 0;
		}

6.7B VERSION RESULTS:

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
 	
// Function declaration but missing body
int func0(float numbers[], int size, float threshold);
 	
int main(){
  float a[] = {1.0, 2.0, 3.9, 4.0, 5.0, 2.2};
  assert(func0(a, 6, 0.3) == 1);
  assert(func0(a, 6, 0.05) == 0);
 	
  float b[] = {1.0, 2.0, 5.9, 4.0, 5.0};
  assert(func0(b, 5, 0.95) == 1);
  assert(func0(b, 5, 0.8) == 0);
 	
  return 0;
}
 	
// Missing function body
	int
	main(void)
	{
	float f[6] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
	double d[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
	
	if (ps_csd_sp(f, 6, stdout) != 1)
	fail("ps_csd_sp", "f", 11, "1");
	if (ps_csd_dp(f, 6, stdout) != 0)
	fail("ps_csd_dp", "f", 12, "0");
	if (ps_csd_dp(d, 5, stdout) != 1)
	fail("ps_csd_dp", "d", 16, "1");
	return 0;
	}
		int
		main(void)
		{
		float f32[3] = { 1.0, 2.0, 3.0 };
		double f64[3] = { 1.0, 2.0, 3.0 };
		
		if (setvbuf(stdout, (char *)NULL, _IONBF, 0) != 0)
		write(2, "setvbuf: ", 9);
		perror("setvbuf");
		}
		
		if (memcmp(&f32, &f64, sizeof(f32)) != 0
		test("float", f32, sizeof(f32), "data_float.raw");
		
		if (memcmp(&f64, &f64, sizeof(f64)) != 0)
		test("double", f64, sizeof(f64), "data_double.raw");
		
		exit(0);
		}

1.3B Model Issues:

- Severe Hallucinations: Both the O0 and O3 decompiled outputs introduce completely irrelevant code that involves printing floating-point numbers, integers, and characters using fwrite. This fabricated functionality does not exist in the original input and represents a significant deviation.
- The addition of unnecessary variables (f, d, i, c, p) further exemplifies the hallucination issues.
- Loss of Core Logic: The missing function func0 should have been part of the logic reconstructed by the model. However, the decompiled output focuses on fabricated printing operations instead of re-creating or handling the incomplete function declaration.

6.7B Model Strengths and Weaknesses:

- More Accurate Array Handling: The 6.7B model, particularly in the O0 output, recognizes the need for arrays and initializes arrays like the original code. This is an improvement over the 1.3B model, which didn't handle arrays properly.
- Hallucinations Persist: Despite the better handling of data structures, the decompiled outputs contain significant hallucinations (e.g., ps_csd_sp, ps_csd_dp, setvbuf, and error-handling logic). These hallucinations detract from the original intent of the main function, which was to test func0.
- Missing Core Logic: The assert statements and calls to func0, which were part of the original code, are entirely missing from the decompiled output. This represents a failure to recover the core logic and functionality of the original program.

FINAL THOUGHTS:

- 1.3B Model: The 1.3B model suffers from severe hallucinations that introduce irrelevant logic and printing operations. It fails to reconstruct any part of the original code's purpose, and the missing function func0 is not handled at all.
- This model is unsuitable for decompiling incomplete or malformed code, as it does not attempt to re-create or infer missing functionality.
- 6.7B Model: The 6.7B model performs better in terms of recognizing data structures and initializing arrays. However, it also introduces hallucinations and fabricated logic that deviate from the original purpose of the program.
- Hallucinations: While the model does a better job handling arrays, it still fabricates unnecessary functionality, such as ps_csd_sp and ps_csd_dp, along with error handling that didn't exist in the original code.
- Loss of Core Logic: The missing function func0 is never reconstructed, and the assert statements from the original program are entirely absent. The core logic of the original program remains unrecovered.
- Recommendations: Neither model handles the malformed input effectively, though the 6.7B model does better in terms of handling data structures.
- Post-decompilation manual intervention is necessary to re-create missing logic or handle incomplete function declarations. The hallucinations should be carefully reviewed and removed to restore the original functionality.

5. **PERTURBED CODE SAMPLE:** The original code is a function func0 that checks if any two elements in the array numbers[] are closer than a given threshold. It uses a nested loop to compare each element with the rest of the elements in the array, using the fabs function to compute the absolute difference between two floating-point numbers. The function returns 1 if such a pair exists, and 0 otherwise. The main function contains several test cases that assert the correctness of func0.

1.3B VERSION RESULTS:

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

// This function checks if any two numbers in the array are closer than the threshold
int func0(float numbers[], int size, float threshold) {
  int i, j;
 	
  // Nested loop to compare each element with the rest
  for (i = 0; i < size; i++) {
  for (j = i + 1; j < size; j++) {
  if (fabs(numbers[i] - numbers[j]) < threshold) {
  return 1; // Close numbers found
  }
  }
  }
 	
  return 0; // No close numbers found
}
 	
int main() {
  float a[] = {1.0, 2.0, 3.9, 4.0, 5.0, 2.2}; // Array of float values
  assert(func0(a, 6, 0.3) == 1); // Assert close numbers exist
  assert(func0(a, 6, 0.05) == 0); // Assert no close numbers
 	
  float b[] = {1.0, 2.0, 5.9, 4.0, 5.0}; // Another test case
	static int func0(float a, float b, float c)
	{
	  int i, j;
	
	  for (i = 0; i < (int)c; i++)
	  for (j = 0; j < (int)b; j++)
	  if (a - 0.0001f < func1(i, j) && func1(i, j) < a + 0.0001f)
	  return 1;
	  return 0;
	}
		int func0(float *a, int n)
		{
		  int i;
		  float min = 1.0f;
		  for (i = 0; i < n; i++)
		  if (fabs(a[i] - 1.0f) < fabs(min))
		  min = a[i];
		  return min;
		}

assert(func0(b, 5, 0.95) == 1);
assert(func0(b, 5, 0.8) == 0);
 	
float c[] = {1.0, 2.0, 3.0, 4.0, 5.0};
assert(func0(c, 5, 2.0) == 1);
 	
float d[] = {1.1, 2.2, 3.1, 4.1, 5.1};
assert(func0(d, 5, 1.0) == 1);
assert(func0(d, 5, 0.5) == 0);
 	
return 0;
}		

6.7B VERSION RESULTS:

INPUT CODE SAMPLE	DECOMPILED O0 CODE	DECOMPILED O3 CODE

// This function checks if any two numbers in the array are closer than the threshold
int func0(float numbers[], int size, float threshold) {
int i, j;
 	
// Nested loop to compare each element with the rest
for (i = 0; i < size; i++) {
for (j = i + 1; j < size; j++) {
if (fabs(numbers[i] - numbers[j]) < threshold) {
return 1; // Close numbers found
}
}
}
	int
	func0(float *x, int n, float eps)
	{
	int i, j;
	
	for (i = 0; i < n; i++)
	for (j = i + 1; j < n; j++)
	if (fabsf(x[i] - x[j]) > eps)
	
	return 1;
	return 0;
	}
		static int
		func0(const float *samples, int num_samples, float thresh)
		{
		int i;
		
		for (i = 1; i < num_samples; i++)
		if (fabsf(samples[i] - samples[i-1]) > thresh)
		return 1;
		return 0;
		}

return 0; // No close numbers found
}
 	
int main() {
float a[] = {1.0, 2.0, 3.9, 4.0, 5.0, 2.2}; // Array of float values
assert(func0(a, 6, 0.3) == 1); // Assert close numbers exist
assert(func0(a, 6, 0.05) == 0); // Assert no close numbers
 	
float b[] = {1.0, 2.0, 5.9, 4.0, 5.0}; // Another test case
assert(func0(b, 5, 0.95) == 1);
assert(func0(b, 5, 0.8) == 0);
 	
float c[] = {1.0, 2.0, 3.0, 4.0, 5.0};
assert(func0(c, 5, 2.0) == 1);
 	
float d[] = {1.1, 2.2, 3.1, 4.1, 5.1};
assert(func0(d, 5, 1.0) == 1);
assert(func0(d, 5, 0.5) == 0);
 	
return 0;
}		

1.3B Model Issues:

1. Hallucinations: In both the O0 and O3 outputs, the 1.3B model introduces hallucinations that significantly alter the logic of the function.

   The fabricated comparisons in the O0 output (e.g., a - 0.0001f < func1(i, j)) and the introduction of irrelevant variables (a, b, and c) represent major hallucinations. The

original function's purpose—comparing elements of an array for proximity—is entirely lost.

The O3 output introduces the calculation of a minimum value (min = a[i]), which has no connection to the original function's logic of comparing array elements using fabs.

2. Loss of Core Logic: The 1.3B model fails to preserve the core logic of the function. The comparisons between elements of numbers[] are replaced with fabricated logic, and the function's return value is incorrectly changed to a floating-point value rather than the original binary return (1 or 0).

6.7B Model Strengths and Weaknesses:

1. Preserving Core Structure: The 6.7B model does a much better job of preserving the core structure of the function. It retains the nested loops and correctly uses fabsf for floating-point comparisons.

   The return logic is also close to the original, returning 1 when a condition is met and 0 otherwise.

2. Logic Inversion: In both the O0 and O3 outputs, the condition for the comparison is inverted (fabsf(x[i] - x[j]) > eps instead of fabs(numbers[i] - numbers[j]) < threshold). While this is a small issue, it changes the behavior of the function and would require correction.

3. Simplification in O3: The O3 decompiled output simplifies the comparison by only checking consecutive elements (samples[i] and samples[i-1]). While this is not strictly incorrect, it deviates from the original function, which compared each element with subsequent elements.

FINAL THOUGHTS:

1. 1.3B Model: The 1.3B model introduces major hallucinations in both the O0 and O3 outputs, replacing the original logic with irrelevant calculations and comparisons. The core purpose of the function is lost, making the 1.3B model unsuitable for decompiling this perturbed code.

2. 6.7B Model: The 6.7B model performs much better, retaining the core logic and structure of the function. However, it inverts the comparison logic and simplifies the comparisons in the O3 output. These issues can be corrected with post-decompilation validation.

   Logic Inversion: The inverted comparison (> eps instead of < threshold) is a minor issue but would change the function's behavior. This should be manually corrected after decompilation.

Simplification: The simplification in the O3 output (comparing consecutive elements only) is another deviation but can be corrected if the full pairwise comparison logic is required.