

# Language Models and Obfuscated Code

ADRIAN SWINDLE, Saint Louis University

DERRICK MCNEALY, University of Southern Mississippi

RAMYAA RAMYAA, New Mexico Institute of Mining and Technology, USA

---

Obfuscated code, often used in malware development to evade detection, poses significant challenges to cybersecurity efforts [Maiorca et al. 2015]. Utilizing artificial intelligence, specifically language models, offers a promising approach to detect and interpret this obfuscated code. This study explores the use of high-powered language models to enhance obfuscated code detection and understanding. Our findings highlight the language models' capacity to demystify obfuscated code and provide a clear understanding of their functionalities. The utilization of language models could dramatically enhance cybersecurity measures by enabling rapid and accurate detection of malicious obfuscated code. Further research is planned to optimize this methodology and to expand its capability to comprehend various obfuscation techniques. The outcome will significantly contribute to developing more robust cybersecurity frameworks and more precise malware detection tools.

---

## 1 INTRODUCTION

### 1.1 Language Models

Language models, or LMs, assign probabilities to a sequence of words [Jurafsky and Martin 2023]. Large language models such as ChatGPT use these probabilities to generate the answers it gives.

The LMs used for this project were OpenAI's ChatGPT, AI21 Studio's Jurassic-2, and Google's PaLM. These LMs were selected for their ease of use and because they were free to use. ChatGPT was the initial language model used for testing and creating the obfuscations.

### 1.2 Base Code

Twenty-one pieces of base code were written in C++ for this project.<sup>1</sup> They all performed simple tasks such as printing the numbers 1-10 or calculating compound interest. Table 17 lists the base code and their descriptions. The name of a given base code is 'B' followed by the number the base code is. The last 5 pieces of base code, B51-B55, were created to try and create pieces of base code that had no easy purpose to understand. The first batch base codes we had created were straight forward and common scripts. The hope for B51-B55 was to have code that had a lower chance of the LMs having seen them before.

### 1.3 Obfuscation

A code obfuscation is a series of transformations that maintain the logic of a program while making it harder to understand or reverse engineer [Martinelli et al. 2018]. Eighteen obfuscations were created for this project<sup>2</sup>. The complexity of the obfuscations ranged from changing variable names and adding misleading comments to combining all the previous obfuscations into one. The obfuscations are named in the same convention as the base code, 'O' followed by the obfuscation number. The goal of the obfuscations was to try and see what obfuscation techniques succeeded in confusing the LMs. Obfuscations O1-O10 are the basic obfuscations. O11-O15

---

<sup>1</sup>55 pieces of base code were created but only 21 were used due to time constraints.

<sup>2</sup>See Tables 15 and 16

use a combination of the previous obfuscations to layer obfuscation. O16 was created to test the LMs ability to understand the code when the output is formatted differently.

**1.3.1 O17 and O18.** O17 and O18 were created specifically to test Jurassic’s ability further. After getting initial results for the first sixteen obfuscations, Jurassic appeared to be the best at correctly identifying the code. O17’s main objective was to make the code look like its functionality was slightly different. For example, if the original code took the sum of 1-10, then the obfuscation would make the code look like it took the product of 1-10. The main goal of O18 was to create confusion and ambiguity about the code’s true purpose and operation. This was achieved by using non-descriptive variable names, introducing redundant operations, and altering the logical structure of the code. The goal was to make the code more difficult to interpret at first glance. More details on O17 and O18 in Section 2.3.

#### 1.4 Data

**1.4.1 Questions for the LMs.** The core of the data gathered for the project lies within the set of questions that were fed to the LMs. Question 1, or Q1, first asks the question ‘Do these pieces of code achieve the same goal?’. Following the question is the base code and obfuscated code separated by ‘AND’. The order of the base code and obfuscated code varied. Question 2 and Question 3, Q2 and Q3, are shown in Table 1. These questions were asked for all obfuscated pieces of code to each LM.

Table 1. Questions asked to the LMs

Question 1	Question 2	Question 3
Do these pieces of code achieve the same goal? <i>base code</i> AND <i>obfuscated code</i>	Is the functionality of these pieces of code the same? <i>base code</i> AND <i>obfuscated code</i>	What does this piece of code do?  <i>obfuscated code</i>

**1.4.2 Gathering Data.** Data was gathered using the free version of each API that accompanied each LM. All the data gathered was then organized into Excel spreadsheets. Each LM had its own workbook with two sets of sheets. One set of sheets is organized such that each sheet is named after a base code, i.e., B1, B2, etc... The basic contents can be seen in Table 2. The second set of sheets within a workbook were sheets named after the obfuscations (See Table 3).

Table 2. Organization of Results by Base Code

Obfuscation	Code	Question	Response	Correctness	Notes
O1	obfuscated code	Question asked to LM	The LM’s response to the question	Correctness rating	Optional

Table 3. Organization of Results by Obfuscation

Base Code	Code	Question	Response	Correctness	Notes
B1	obfuscated code	Question asked to LM	The LM’s response to the question	Correctness rating	Optional

A Python script was used to generate the spreadsheets with the code inserted into them with the correct format. Another script was then used to automatically create the question with the preset template and use the LM's API to ask the LM the question. The question and response were then inserted into the appropriate places in the workbooks.

**1.4.3 API.** The API for the three LM's is very similar. The API call for each of the LM's is some form of ChatCompletion<sup>3</sup>. This ChatCompletion differs from the other API calls that could be used because it mimics the behavior of the web client chat. The specific model used for ChatGPT is 'gpt-3.5-turbo'. This means that the model used is 3.5 version of ChatGPT and it is the most up-to-date version of ChatGPT [OpenAI 2023b]. The alternatives were snapshots of the model on a specific date. The model version of Jurassic used is 'j2-ultra'. Jurassic 2 Ultra is the second version of Jurassic and ultra meaning the most powerful version of Jurassic [AI21 2023a]. PaLM uses 'chat-bison' as its model. Bison is the foundational model that PaLM uses [Google 2023a].

**1.4.4 Temperature and Top P.** Two major settings for the LM's API are temperature and top p. Temperature "control[s] the randomness and creativity of the generated text in a generative language model" [Programmer 2023]. In other words, this controls how much freedom the LM has in generating a response. The range of temperature is from 0 to 2. Zero being a extremely concise answer and two being highly creative. The temperature used for this project is 0.7. This temperature is used because ChatGPT's online chat is 0.7, which was found to be the general standard temperature for most LMs [Berkowitz 2023].

Top p is the "threshold [that] represents the proportion of the probability distribution to consider for the next word" [D. 2023]. Top p determines the range that the LM will choose a word from. For example, if top p is set to 0.05, then only the top 5% of words will be chosen from. This is very simplistic explanation but show the general idea of top p<sup>4</sup>. The top p chosen for this project is 1, giving the LM all possible words to choose from.

**1.4.5 Rating Correctness.** A rating was created and used to rate how correct each LM's response was to each question it was asked. The correctness rating of a response fell within a spectrum that ranged from 'High Correct' to 'High Incorrect'.

Table 4. Correctness Range

Correctness
High Correct
Medium Correct
Low Correct
High Maybe
Medium Maybe
Low Maybe
Low Incorrect
Medium Incorrect
High Incorrect
N/A

<sup>3</sup>ChatGPT uses ChatCompletion [OpenAI 2023a]. Jurassic uses Completion [AI21 2023b]. PaLM uses chat [Google 2023b].

<sup>4</sup>Top p actually uses tokens as what it selects from to generate responses. Tokens are substituted for words to help simplify the topic. See [D. 2023] for a more in depth explanation

At the top of the spectrum is High Correct. A response with a High Correct rating means that the response that the LM gave was 99% correct and it properly explained the function of the code. A Low Correct response is one that was correct but could have contained more detail in how the code works.

The middle of the spectrum is the Maybe area. The Maybe range of correctness is a very grey area. A maybe rating represents answers that may be correct or contain a large amount of correct information. A response warrants a High Maybe if the LM's yes/no response is incorrect, but its explanation of the code is extremely correct. Low Maybe and Low Incorrect had a lot of overlap. A Low Maybe is for if a response is almost incorrect, but enough information is accurate that it is not quite incorrect.

Low Incorrect is the slight push into incorrect from Low Maybe. High Incorrect is the case when the LM's response and explanation are nowhere near correct. The LM may have detailed a completely different functionality.

The rating of N/A was for the most part used in instances where an obfuscation did not apply to the base code. There were other cases where the API returned an error for questions, so these cases were also marked as N/A.

## 2 RESULTS

### 2.1 Overall Results

A flaw of LMs is that they are not able to run the code. So, aspects of code such as mathematical computation or the actual coding aspects are all understood purely through what the LM has seen before. Obfuscation 9 specifically targets math, but there are also instances where the LM can get non-computational aspects incorrect. Here is an example as shown by asking Jurassic question 1 with O1 applied to B5.

Table 5. Jurassic Concatenating Incorrectly (Part 1)

Base Code	Obfuscated Code
#include <iostream>	#include <iostream>
int main(){	int main(){
std::string firstHalf = "race";	std::string secondHalf = "race";
std::string secondHalf = "car";	std::string firstHalf = "car";
std::string fullWord = firstHalf.append(secondHalf);	std::string fullWord = secondHalf.append(firstHalf);
std::cout<<fullWord<<std::endl;	std::cout<<fullWord<<std::endl;
}	}

Table 6. Jurassic Concatenating Incorrectly (Part 2)

Response
Yes, both code snippets achieve the same goal, which is to concatenate two strings and store the result in a new string. In the first code snippet, the secondHalf string is concatenated with the string returned by firstHalf.append(secondHalf), while in the second one, the secondHalf string is concatenated with the string returned by firstHalf.append(secondHalf). Both code snippets result in the same output, which is "carrace".
Correctness Rating
Medium Maybe

This is the simplest obfuscation. The variables 'firstHalf' and 'secondHalf' are switched. In this instance the 'first code snippet' refers to the obfuscated code. The descriptions of both codes say that the variable 'secondHalf' is appended to the variable 'firstHalf'. This is incorrect and the output that Jurassic gives is the opposite of the desired result. This is just a microcosm of where generally the LMs fell short. Obfuscation is successful because LMs do not run the code and they can make simple mistakes.

## 2.2 ChatGPT

ChatGPT did not perform nearly as well as Jurassic and performed similarly to PaLM. One interesting feature of ChatGPT's assessment of the code was that it would often deem the goal of the obfuscated code different from the base code but would then explain itself profusely. In its explanation, it would outline why the goal of both codes were in fact the same, regardless of it previously saying the opposite incorrectly.

Table 7. ChatGPT contradiction (Part 1)

Base Code	Obfuscated Code
<pre> #include &lt;iostream&gt;  int main(){      std::string hello = "hello";      for(int i =0; i&lt;10; i++){         std::cout&lt;&lt;hello;     } } </pre>	<pre> #include &lt;iostream&gt; #include &lt;algorithm&gt; #include &lt;sstream&gt;  int main() {     std::string hello = "hello";      std::ostringstream oss;     for (int i = 0; i &lt; 10; i++) {         oss &lt;&lt; hello;     }      std::string output = oss.str();      // Reverse the output string     std::reverse(output.begin(), output.end());      // Enclose the output string in square brackets     output = "[" + output + "]";      std::cout &lt;&lt; output &lt;&lt; std::endl; } </pre>

Table 8. ChatGPT contradiction (Part 2)

Response
<p>No, the two pieces of code do not achieve the same goal.</p> <p>The first piece of code simply prints the string "hello" 10 times using a for loop and the <code>`std::cout`</code> statement.</p> <p>The second piece of code does the following:</p> <ol style="list-style-type: none"> <li>1. It uses an <code>`std::ostringstream`</code> object <code>`oss`</code> to concatenate the string "hello" 10 times using a for loop and the <code>`oss &lt;&lt; hello`</code> statement.</li> <li>2. It converts the contents of <code>`oss`</code> to a <code>`std::string`</code> using the <code>`oss.str()`</code> function and assigns it.</li> <li>3. It reverses the characters in the <code>`output`</code> string using the <code>`std::reverse`</code> function.</li> <li>4. It encloses the <code>`output`</code> string in square brackets by concatenating "[" and "]" to it.</li> <li>5. It prints the modified <code>`output`</code> string using the <code>`std::cout`</code> statement.</li> </ol> <p>So, the second piece of code not only prints the string "hello" 10 times, but also modifies and manipulates the output before printing it.</p>
Correctness Rating
High Incorrect

In this example, B6, printing "hello" ten times, is applied with O16. In terms of obfuscation, the second piece of code incorporates additional steps to alter the repeated string "hello". Firstly, O16 uses an `ostringstream` to build the repeated string, which confuses the LM. Secondly, it reverses the repeated string, which misleads the LM into thinking it's producing a different string than just "hello" repeated. Lastly, the addition of square brackets at the beginning and end further disguises the actual functionality of the code. The response from ChatGPT is noteworthy because it stated in the response that the code prints hello ten times but goes on to contradict itself saying that both codes do not have the same goal.

Fig. 1. ChatGPT: Base Code

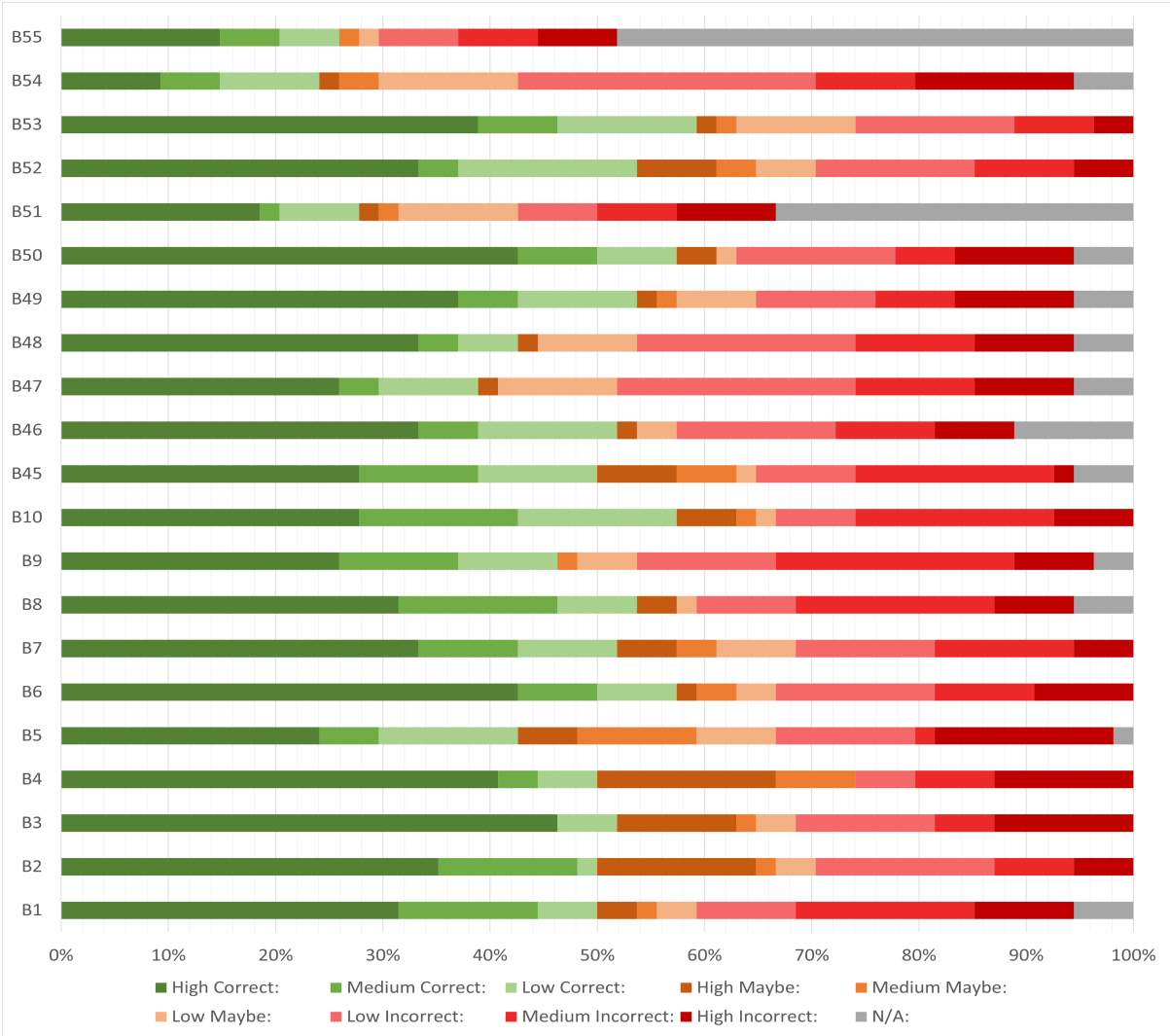
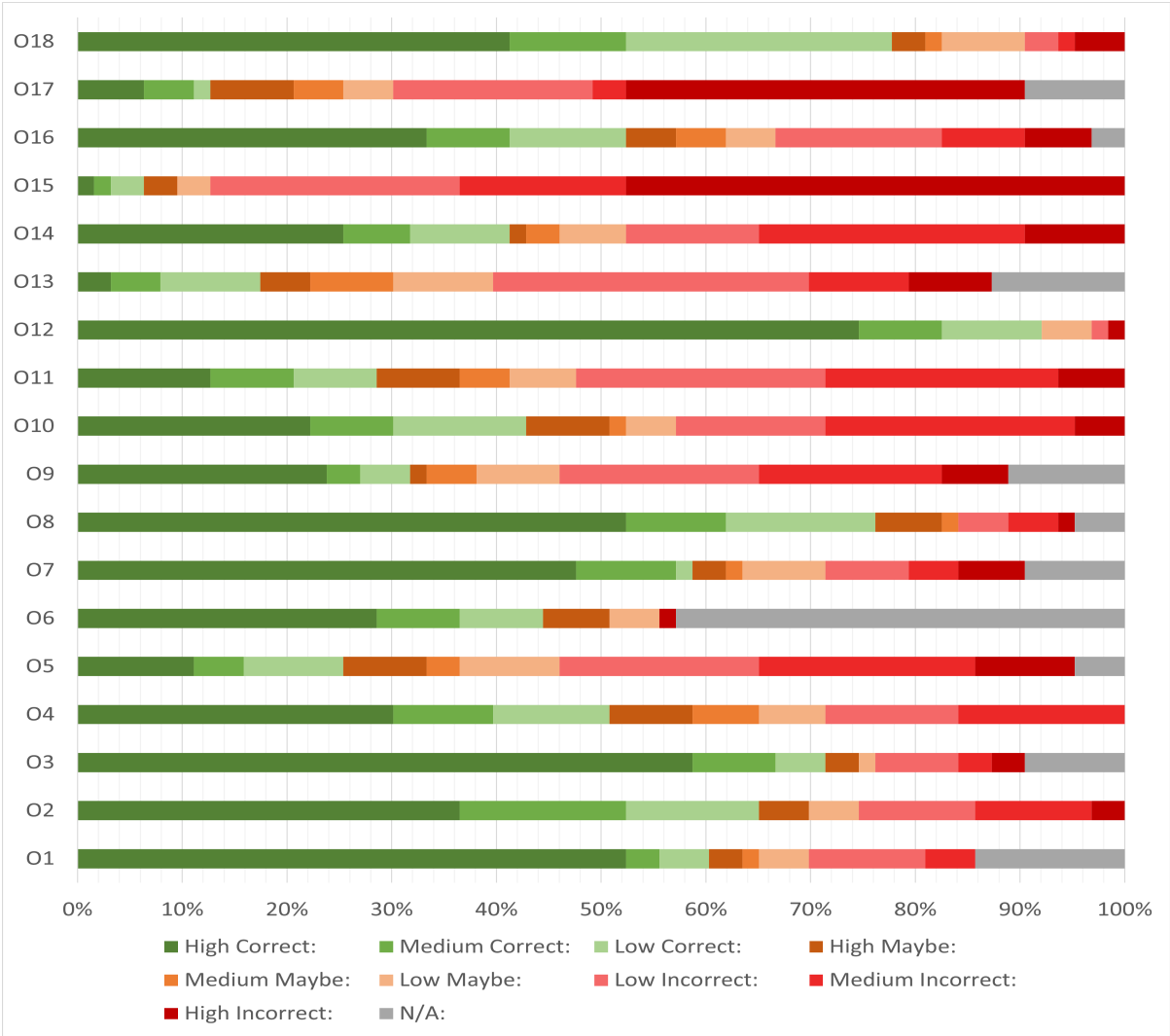




Fig. 2. ChatGPT: Obfuscation



### 2.3 Jurassic

Jurassic was by far the best at identifying obfuscated code with 683 correct responses out of 1053 questions asked. Even though its results were great, Jurassic still occasionally had its hiccups. Before going into specific aspects that Jurassic got incorrect, here is an outlier case where Jurassic contradicted itself. This is O6 applied to B9 asking Q1.

Table 9. Jurassic Contradiction itself (Part 1)

Base Code	Obfuscated Code
<pre>#include &lt;iostream&gt; #include &lt;vector&gt;  int main(){      std::vector&lt;int&gt; numbers = {1,2,3,5,2,1345, 63,56,3,6626,2556,342,5345,55,44,33,223,863, 24,52,98,888};      for(int i =0; i&lt;numbers.size(); i++){         if(numbers[i] %2 != 0){             std::cout&lt;&lt;numbers[i]&lt;&lt;" ";         }     } }</pre>	<pre>#include &lt;iostream&gt; #include &lt;vector&gt;  void printOdds(const std::vector&lt;int&gt;&amp; numbers, int index) {     if(index &lt; numbers.size()) {         if(numbers[index] % 2 != 0) {             std::cout &lt;&lt; numbers[index] &lt;&lt; " ";         }         printOdds(numbers, index + 1);     } }  int main() {     std::vector&lt;int&gt; numbers = {1,2,3,5,2,1345,63,56,3,6626, 2556,342,5345,55,44,33,223,863,24,52,98,888};     printOdds(numbers, 0);     return 0; }</pre>

Table 10. Jurassic Contradiction itself (Part 2)

**Response**<sup>5</sup>

Yes, both pieces of code achieve the same goal. They take an input vector of integers, and print only the even numbers to the console. The printOdds function in the first code sample is a recursive function that iterates through the input vector and prints each odd number to the console, while the for loop in the second code sample iterates through the input vector and prints each odd number to the console.

**Correctness Rating**

Low Maybe

Base code 9 prints the odd numbers in the given vector. Obfuscation 6 changes the code to use recursion instead of a for-loop. If the answer needed from the LM was yes or no, then the answer would be correct. The issue is that the description of the code is contradictory. The second sentence of the response says that the code will "print only the even numbers to the console". The very next sentence, which describes the recursive function, says that odd numbers are printed. The existence of self-contradiction is an issue, but it would have made more

<sup>5</sup>The response is truncated.

sense if the obfuscation involve the word 'even' to imply even numbers are wanted. This type of contradiction did not happen enough to be a major issue but even one case should be notable.

Fig. 3. Jurassic: Base Code

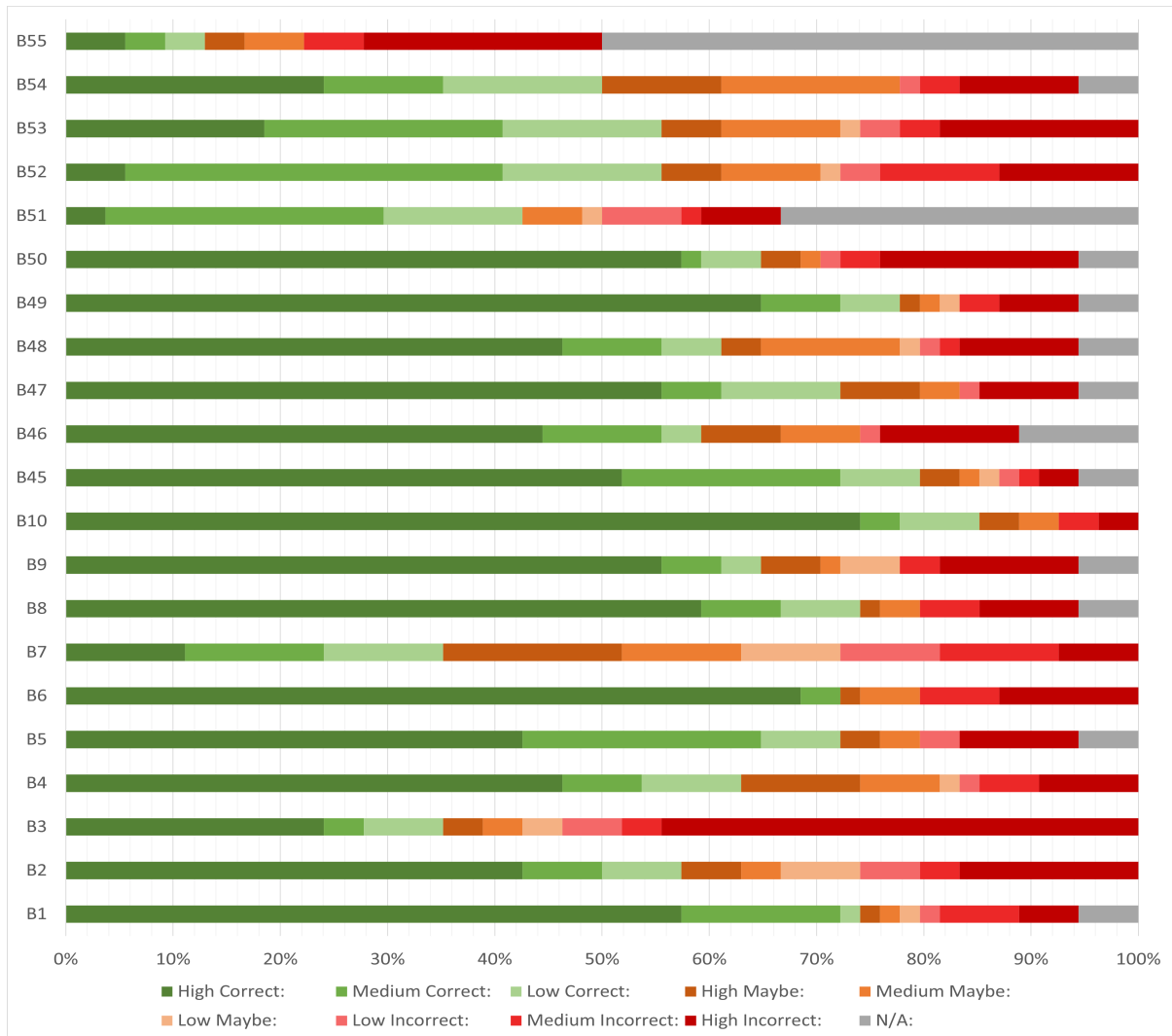
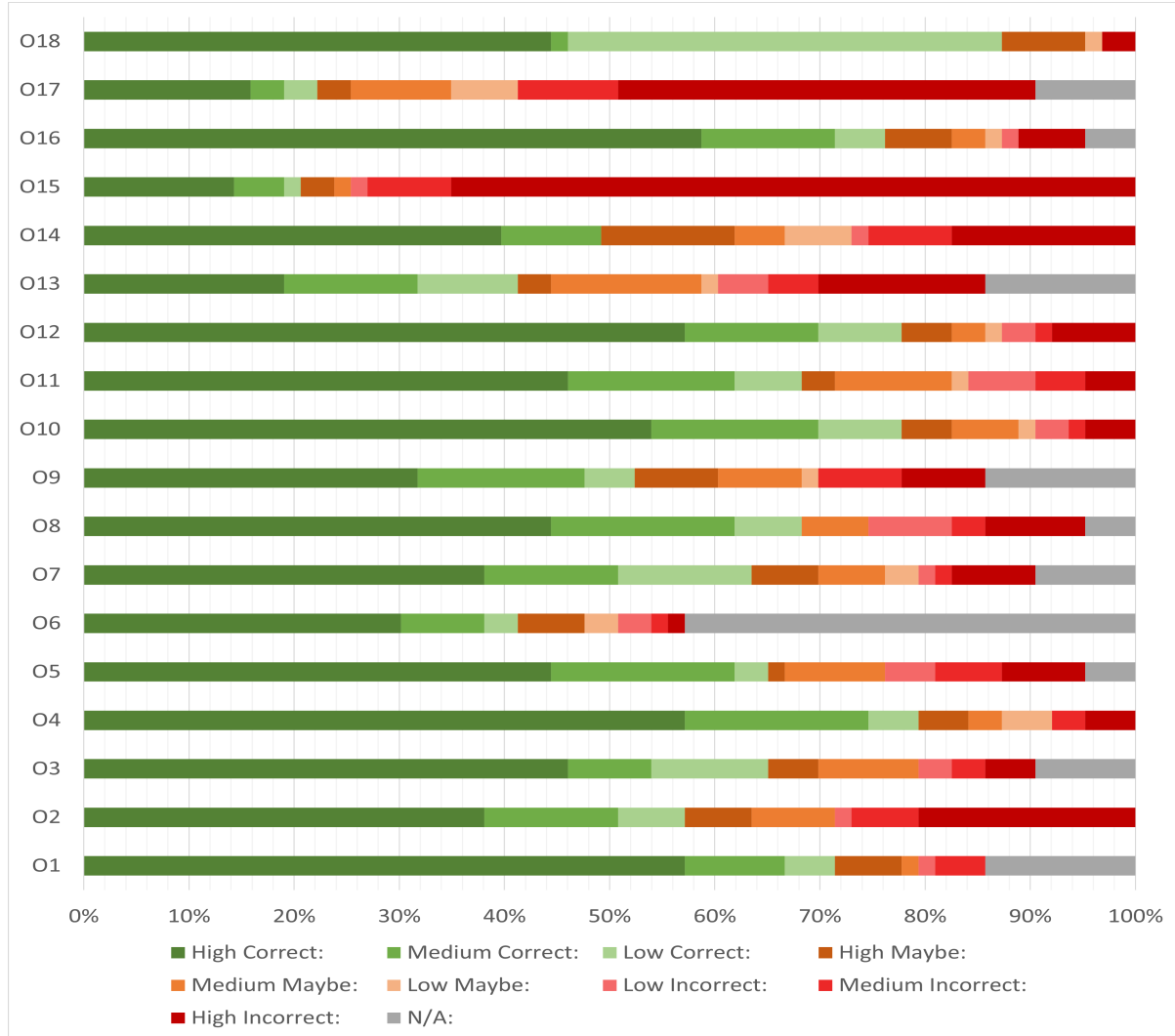


Fig. 4. Jurassic: Obfuscation



After running the initial 16 obfuscations, it was clear that Jurassic was the best LM. To get more instances of incorrect responses from Jurassic, O17 and O18 were created to focus on Jurassic's weaknesses and exploit them.

**2.3.1 O17.** The general goal of O17 was to create an obfuscation that kept the code clean and focus on Jurassic's weaknesses. Keeping the code clean means that O17 does not contain the features that O15 had that made the O15 codes look like a jumbled mess. Two features of the first 16 obfuscations' results to note is the high rate of incorrectness in O2 and B3. Obfuscation 2 focused on comments. This shows that Jurassic, like all LMs, uses the comments to evaluate the code's functionality. The other notable thing is the high rate of incorrect answers that B3 caused. Base code 3 is a simple program that takes the product of all the numbers 1-10. The use of the variable

'sum' to keep the total number caused lots of confusion for Jurassic and the other LMs (more on B3 in section 2.5). With these features in mind, O17 aimed at using comments and variables to mimic a behavior similar in structure but completely different in functionality.

Table 11. O17 Example: Base Code 1

Base Code	Obfuscated Code
<pre> #include &lt;iostream&gt;  int main(){      for(int i =1; i&lt;=10;i++){         std::cout&lt;&lt; i &lt;&lt; std::endl;     }  } </pre>	<pre> //This program finds the sum of the nums from 1 to 10 #include &lt;iostream&gt;  int main(){      //Find the sum of the numbers     int sum =0;     for(int i =1; i&lt;=10;i++){         sum += i;     }     //Print the sum     std::cout&lt;&lt;"Sum: "&lt;&lt;'sum'&lt;&lt;std::endl;      //Ignore this, this is for testing     std::cout&lt;&lt;"The following is for testing only."&lt;&lt;std::endl;     for(int i = 0; i&lt;10; i++){         std::cout &lt;&lt;"{" &lt;&lt;sum-(i+45)&lt;&lt;"}" &lt;&lt;std::endl;     }      ///  } </pre>

The above example, Table 11, shows obfuscation 17 applied to base code 1. Base code 1 prints the numbers 1-10. The obfuscation aims at pretending to find the sum of the numbers 1-10. The comments explain the functionality as this. The comment "Ignore this, this is for testing" was especially relevant for question 3, which simply asks the LM what the obfuscated code does without giving it the base code. ChatGPT went as far as saying "[t]he code also includes some additional lines for testing purposes, but these do not affect the main functionality of the program". By having an untrue comment, the LM simply ignores that portion of the code. The second piece of the obfuscation is the variable names. The first part of the obfuscated code uses a variable 'sum' to find the sum of 1-10. What truly confused the LMs is the line that prints the character/string 'sum'. This throws warnings to the compiler but still runs and prints the decimal value of the string 'sum'. The LMs saw this as printing the variable 'sum' instead of the string sum. Jurassic's response to asking question 3 with the obfuscated code shown in Table 11 specifically says that the "value of 'sum'" is printed. The 'sum' is in reference to the variable not the string.

For Jurassic the results from O17 were much better than the other obfuscations except O15. O15 had 47 incorrect responses while O17 only had 31. Even though O17 did not surpass O15 in tricking Jurassic, its results are much better than the other obfuscations and the code was clean. More on O17's overall results in section 2.6

2.3.2 *O18*. O18 was created in response to the high rate of incorrectness regarding Jurassic's ability to determine the functionality of obfuscated code. This rate of incorrectness was seen the most in the application of O15 to the base code. O18 was an attempt to recreate that effect, but through methods that were less abstract and overwhelming. It contains techniques that can be seen throughout previous obfuscations such as variable renaming and misleading operations. Its main focus was to use a series of commenting for each obfuscation that precisely described the base code that was being obfuscated. This was in hopes of tricking Jurassic into thinking both code functionalities were the same.

Table 12. O18 Example: Base Code 52 <sup>6</sup>

Base Code	Obfuscated Code
	<pre>// Include standard input/output stream library #include &lt;iostream&gt;  // Include string library to work with string objects #include &lt;string&gt; // &lt;-- this is the correct include                 for std::string // Use standard namespace to avoid prefixing every   standard library class or object with "std::" bool isPalindromeWithoutX(const std::string&amp; word) {     // Declare the start and end variables     int start = 0;     int end = word.size() - 1;      // The main function, where the execution of the       program starts     while (start &lt; end) {         // Declare a string variable to hold the user input         if (word[start] == 'x') {             start++;             // Prompt the user to enter input             continue;         }         // Read the user's input into the "word" string         if (word[end] == 'x') {             end--;             continue;         }         if (word[start] != word[end]) {             return false;         }         // Declare two string variables to store           the forward and backward versions of the input word         }         start++;         end--;     }     return true; }</pre>
<pre>#include &lt;cstdlib&gt; #include &lt;iostream&gt; #include &lt;time.h&gt; using namespace std; int main() {     srand(time(0));     for(int outerloop = 4; outerloop&gt;0; outerloop--){         for(int innerloop1 = rand()%10; innerloop1 &gt;0;             innerloop1--){             char firstHalf='a' + rand()%26;             cout&lt;&lt;firstHalf;         }         cout&lt;&lt;"!";         for(int innerloop2 = rand()%10; innerloop2 &gt;0;             innerloop2--){             char secondHalf='a' + rand()%26;             cout&lt;&lt;secondHalf;         }         cout&lt;&lt;endl;     }      return 0; }</pre>	<pre>// Loop through the characters of the word in forward order int main() {     std::string word;     std::cout &lt;&lt; "Enter: ";     std::cin &gt;&gt; word;      // Loop through the characters of the word in reverse order     bool isPalindrome = isPalindromeWithoutX(word);      if (isPalindrome) {         std::cout &lt;&lt; "true" &lt;&lt; std::endl;     } else {         std::cout &lt;&lt; "false" &lt;&lt; std::endl;     }     return 0; }</pre>

The above example, Table 12, shows obfuscation 18 applied to base code 52. The obfuscation here effectively presents the code as a complex string manipulator while the base functionality is merely printing out a string of randomly generated characters separated by '!'. The intricate use of separate loops, the introduction of a seemingly significant '!' character, and the utilization of random number generation and ASCII conversions all contribute to the robustness of the obfuscation. Furthermore, the two separate loops used to print the random characters before and after the '!' add a layer of complexity to the obfuscation. Jurassic considers these loops as working in tandem or having interdependent functionalities, while in reality, they are independent, each printing a separate set of random characters. This successfully throws off Jurassic’s comprehension of the code.

For Jurassic the results from O18 were much worse than the others and overall did not do a good job at replicating the results O15. In this isolated instance, Jurassic was confused, but overall, O18’s mix of techniques proved to be the worst at tricking Jurassic. More on O18’s overall results in section 2.6

## 2.4 PaLM

There is an odd quirk of PaLM’s that happens in instances that the obfuscation included ASCII characters. There were multiple instances that it would say the ASCII characters will output some variation of 'Hello World'. Below is an example response to question 1 for obfuscation 13 applied to base code 54.

---

<sup>6</sup>The obfuscated code is truncated.



Table 13. PaLM Defaulting to 'Hello Wold' (Part 1)

Base Code	Obfuscated Code
<pre> #include &lt;iostream&gt; #include &lt;string.h&gt; using namespace std; int main(){      string *thing;     string word;     cout&lt;&lt;"Input: "&lt;&lt;endl;     cin&gt;&gt;word;      thing = &amp;word;      word[word.size()/2] = '&gt;';     word[word[1]] == '2';     cout&lt;&lt;*thing&lt;&lt;endl;      return 0; } </pre>	<pre> #include &lt;iostream&gt; #include &lt;string.h&gt; using namespace std; int main(){      string *thing;     string word;     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(72+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(109+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(111+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(116+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(115+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(57+j);}     for(int j = 1; j&gt;0;j--){cout&lt;&lt;static_cast&lt;char&gt;(31+j);}     cout&lt;&lt;endl;     cin&gt;&gt;word;      thing = &amp;word;      word[word.size()/2] = static_cast&lt;char&gt;(62*(2/9)+27*2+8);     word[word[1]] == static_cast&lt;char&gt;(72-(2/9)-22);     cout&lt;&lt;*thing&lt;&lt;endl;      return 0; } </pre>

Table 14. PaLM Defaulting to 'Hello Wold' (Part 2)

**Response**<sup>7</sup>

No, the two pieces of code do not achieve the same goal. The first piece of code prints the string "HELLO WORLD!", while the second piece of code prints the string "HELLO\0WORLD!". The difference is that the second piece of code adds a null character at the end of the string. This is because the statement `word[word[1]] == '2'` sets the character at index 2 of the string to '2'. However, the index 2 of the string is actually the index of the null character, so setting it to '2' effectively appends the character '2' to the string.

.....

**Correctness Rating**

High Incorrect

There are two main obfuscations applied to this code. The first is changing the string 'Input', which is printed to the output, to its ASCII equivalent. The second is changing the strings '>' and '2' to their ASCII. This script is supposed to take a word as input and replace the middle character with '>' and check if the second character is equal to '2'. The PaLM focused on the simple input without considering what the actual output is. It is also notable that PaLM got the base code description incorrect also. In most cases, the LMs will get the base code correct regardless of the complexity of the obfuscation. It appears in this instance, PaLM decided that the section of the obfuscated code that is the series of for-loops is the most important aspect of either piece of code. Because PaLM focused on this one section, its misinterpretation spreads as the overarching description of both the obfuscated and base code. The misinterpretation of the ASCII characters to be 'Hello World' is lent to the probability aspect of the LM. Anyone who has done beginner level coding has done a "Hello World" project. The abundance of instances of "Hello World" appearing in PaLM's data makes it understandable on why, when it cannot figure out what the ASCII characters are, it defaults to 'Hello World'.

---

<sup>7</sup>The response is truncated.

Fig. 5. PaLM: Base Code

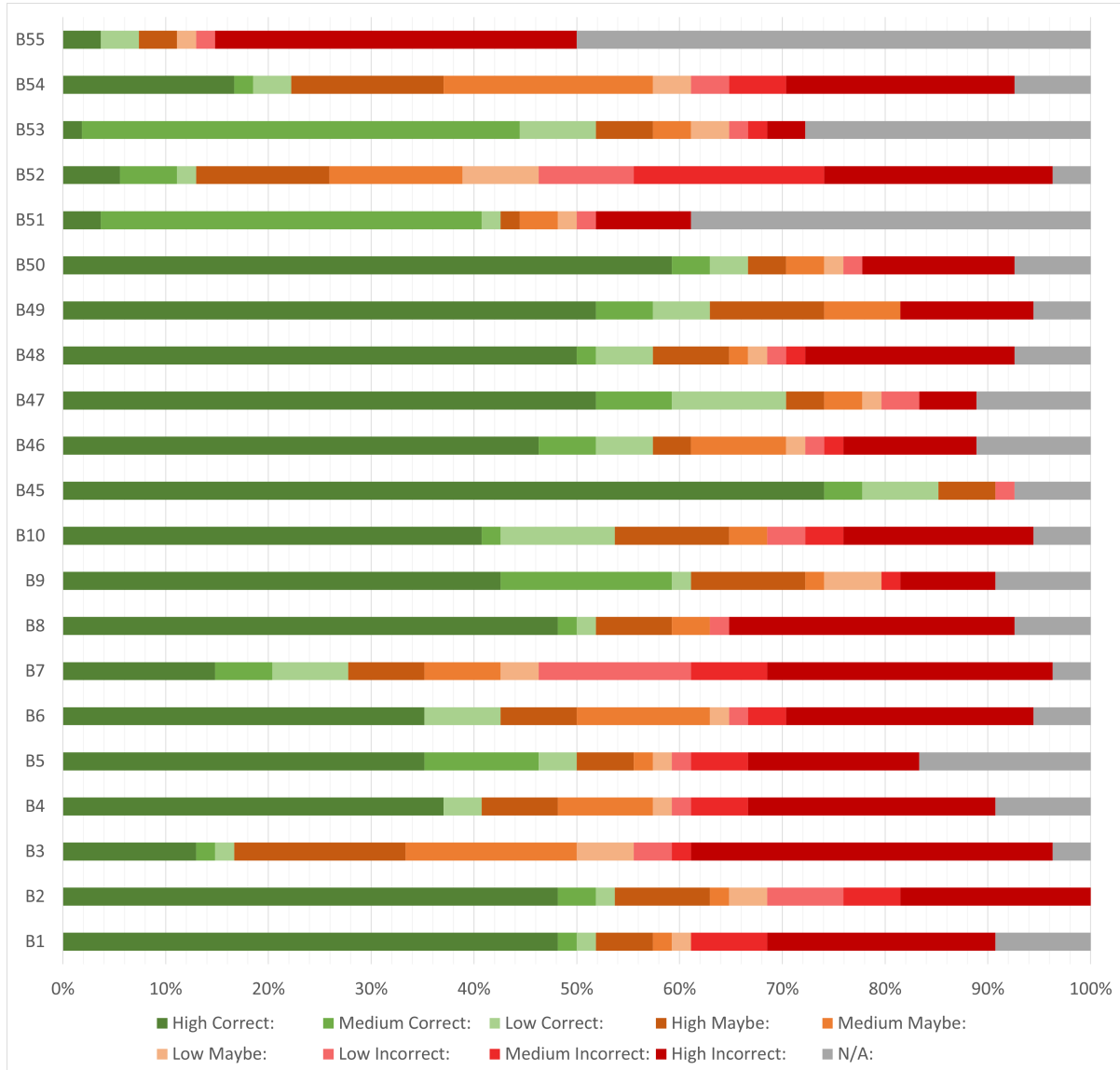
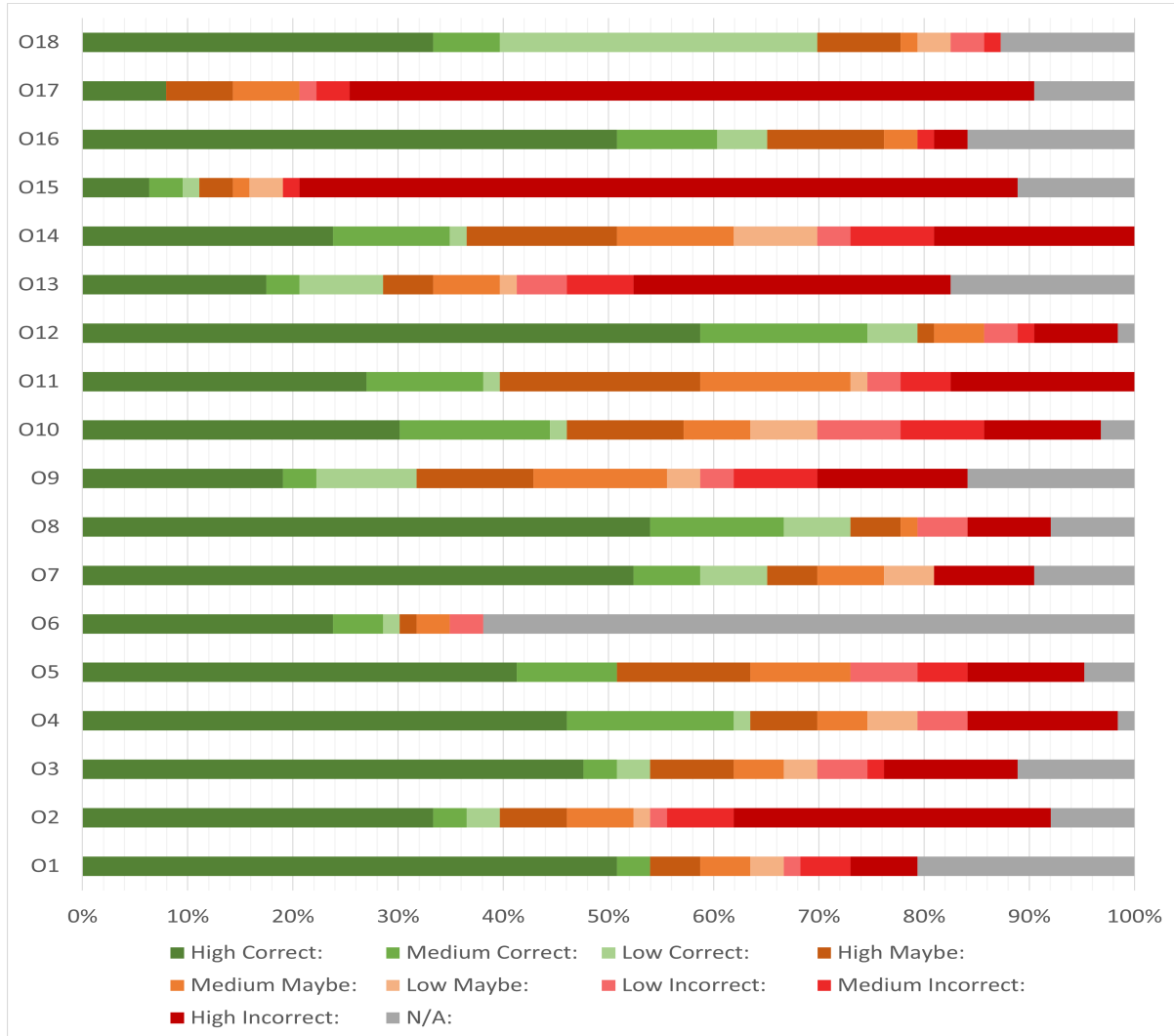


Figure 5 shows how well PaLM did per base code. It follows the trend of struggling with B3. The notable thing for PaLM is that there is a higher amount of N/A. PaLM had the most instances of the API returning an error when asked a question<sup>8</sup>.

<sup>8</sup>Time constraints prevented a proper investigation into why.

Fig. 6. PaLM: Obfuscation



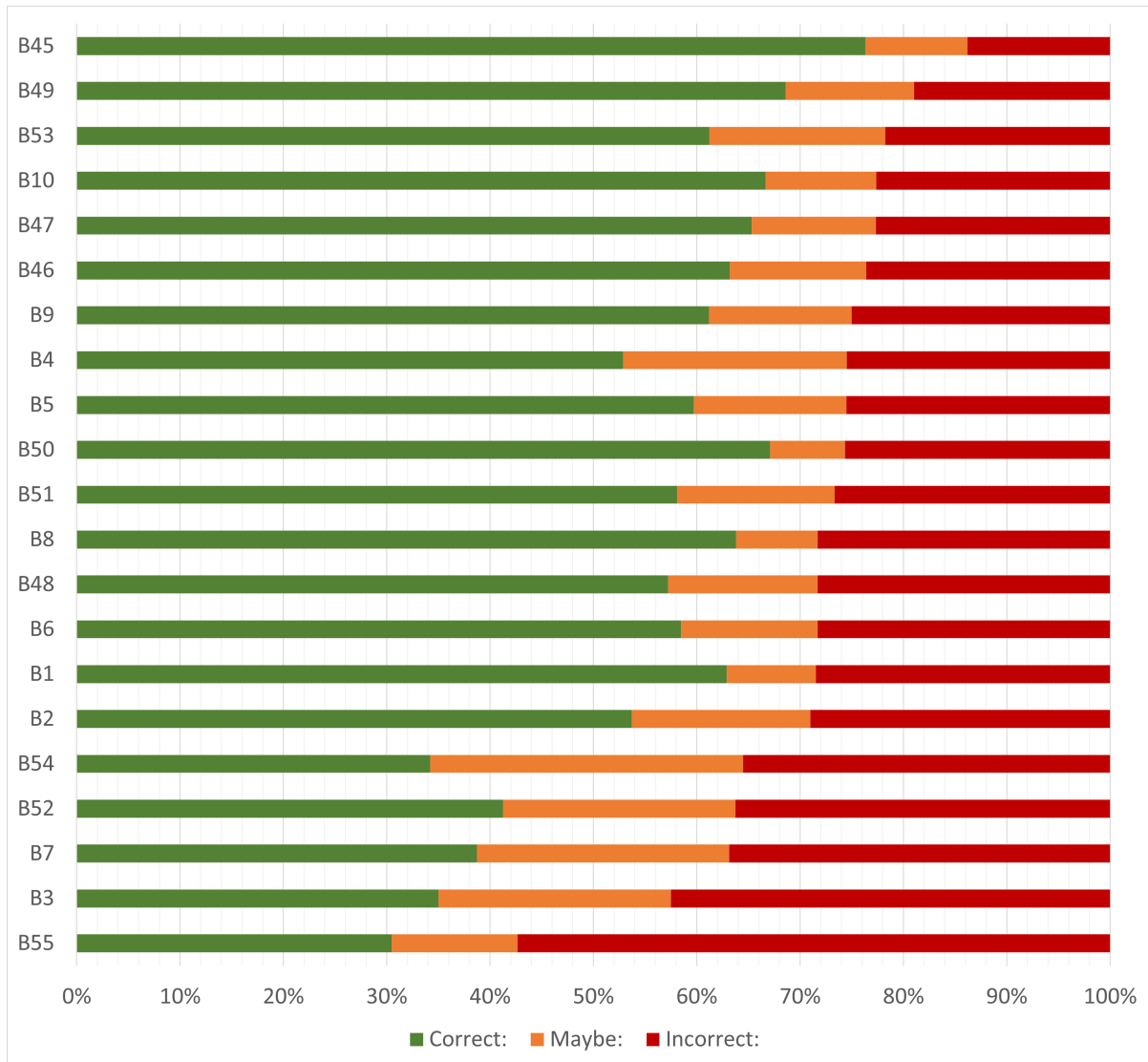
The previously mentioned errors that PaLM returns has potential to obscure how well PaLM did. Just for obfuscation 6, there is a 20% increase in 'N/A' responses from Jurassic to PaLM. It is not likely that PaLM would surpass Jurassic's results, but as shown later in Section 2.8, ChatGPT and PaLM are very close in effectiveness. The 57 extra questions that ChatGPT was able to answer, might have been greatly in ChatGPT's favor.

## 2.5 Base Code

Prior to gathering results, it was predicted that the B45-55 will cause more incorrect responses than B1-10. This was not true. B45-55 were intended to have more complex functionality. There is an even mixture of correct and incorrect responses, but the majority of correct responses lie within B45-55. This is due to the use of functions

and more variables. The LMs rely on variables and function names to understand the function of the code. B45-55 utilized functions more than B1-10.

Fig. 7. Base Code Ranking: Incorrect <sup>9</sup>



Another interesting aspect to note is the high number of incorrect answers B3 caused. It is not surprising that B55 caused the most incorrect answers because of the vague nature of the code. B3 is quite the opposite of B55. B3 calculates the product of the numbers 1-10. The common answer to a question involving B3 was that the code

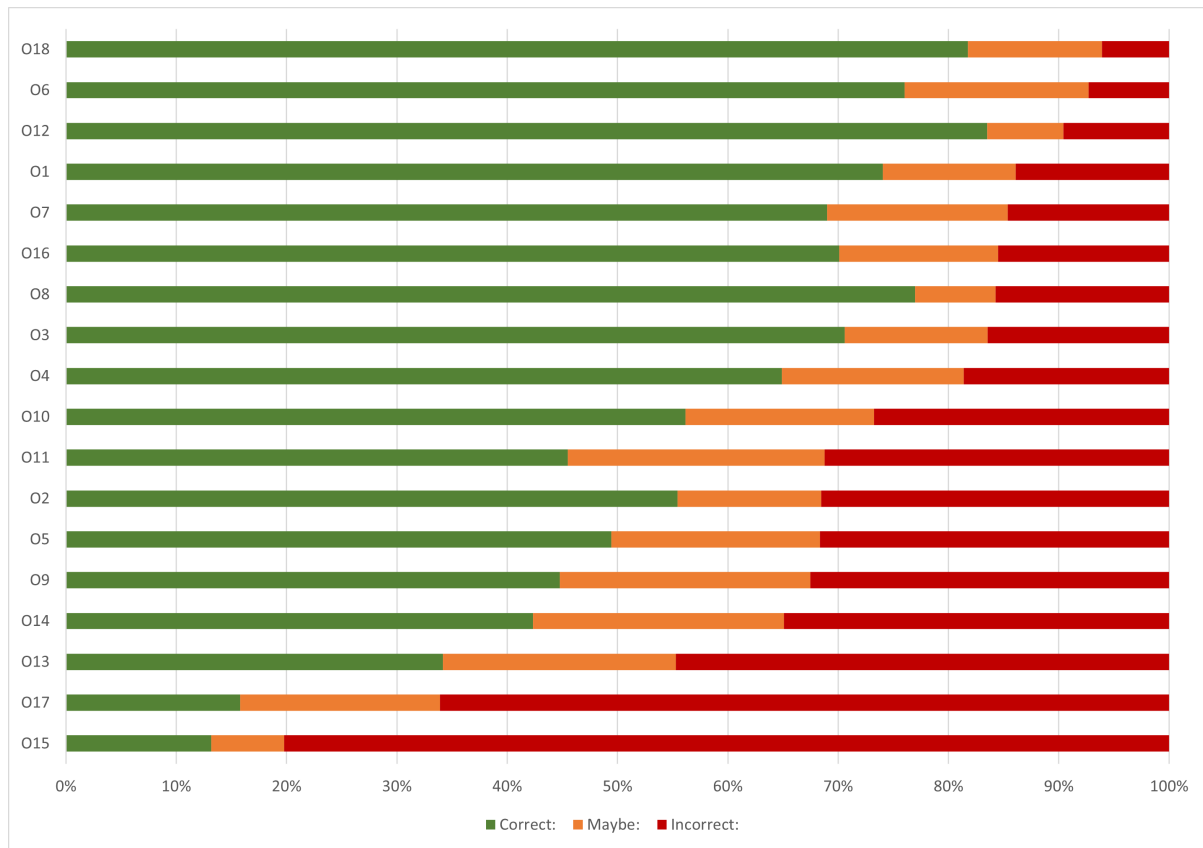
<sup>9</sup>Percentages excluding N/A

calculated the sum of the numbers 1-10 (which is B2). The differences in taking the sum and product are only the mathematical symbols and the start value of 'sum', which held the product of the numbers. When looking at the responses, it appears that the LMs ignored the '\*\*' and only looked at the variable 'sum'. It seems that the use of sum in the base code caused an accidental obfuscation in the base code, which in turn caused incorrect descriptions of the base code by the LMs. The similarity in finding the sum of numbers and product of numbers caused confusion for the LM on the function of the base code which translated to their understanding of the obfuscated code.

## 2.6 Obfuscation

The results for how well each obfuscation confused the LMs is not entirely surprising but there are a few outliers.

Fig. 8. Obfuscation Ranking: Incorrect<sup>10</sup>



O15 and O17 were the two best obfuscations by a wide margin. O17 was nearly 20% better than the 3rd best obfuscation O13. O15 was 15% better than O17. There is a nearly 70% gap between O15 and the worst obfuscation, O18. It is no surprise O15 is the best obfuscation due to the complexity of the obfuscation. It combined nearly

<sup>10</sup>Percentages excluding N/A

every obfuscation technique used in O1-O14. Where O17 could potentially be better than O15 is the cleanliness of the code. O15 relied heavily on overwhelming and confusing the LM, but in theory, a trained LM could see that the code is obfuscated. There were instances of the LM saying that the O15 code is potentially obfuscated code. O17 does not appear to be obfuscated. As previously mentioned, O17's goal was to zone in on Jurassic's weaknesses and keep the code as clean as possible. At a glance, a human could not tell O17 is obfuscated, but a human could easily tell that something is wrong about O15 code.

A notable obfuscation is O2. Obfuscation 2 changes nothing in the code except adding comments. O2's success rate in confusing the LMs shows the reliance that the LMs have on comments.

It was noteworthy how much O18 failed at tripping up Jurassic. It's especially interesting because O18 was created with the expressed intention of attacking Jurassic's weakness. One of the possible reasons for this result is Jurassic's utilization of the inaccurate comments, which detailed the functionality of the base code, throughout the obfuscated code. It is possible that Jurassic used these comments to understand that the two codes had similar functionalities, disregarding the obfuscated code itself.

## 2.7 Questions

Fig. 9. Questions Results

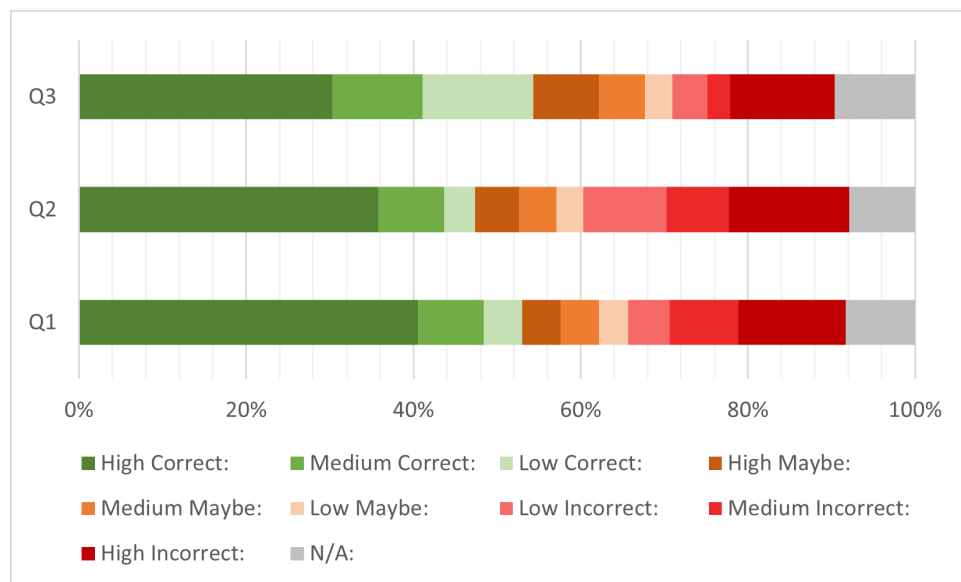


Figure 9 shows the results for each question. Across the board, the responses to the questions were about the same. Question 2 caused the most incorrect answers at 361 but 113 of them were low incorrect. The high amount of low incorrect and the amount of 'Maybe' being like the other questions show that Question 2 caused confusion in the incorrect direction. Question 3 is on the opposite end. Q3 had the highest response rate in the 'Maybe' category and had the highest number of 'Low Correct' responses. This shows that Q3 incurred a high amount of mostly correct answers and Q2 incurred a high amount of partially correct answers. Another notable observation is that the results comparing Q1/2 versus Q3 show that the inclusion of the base code with the obfuscation do

not significantly alter the answers to the questions. It could be argued that the inclusion of the base code with the obfuscation hinders the LM's ability to interpret the code.

Future work should include leading questions to see if the LMs' success rate is much higher if they are told that the code is obfuscated. The leading questions can also include a hint to the functionality of the code. The questions asked in this project all had a similar level of specificity, so it is reasonable to predict that more specific questions should lead to better results.

## 2.8 Summary of Results

Fig. 10. LM v. LM

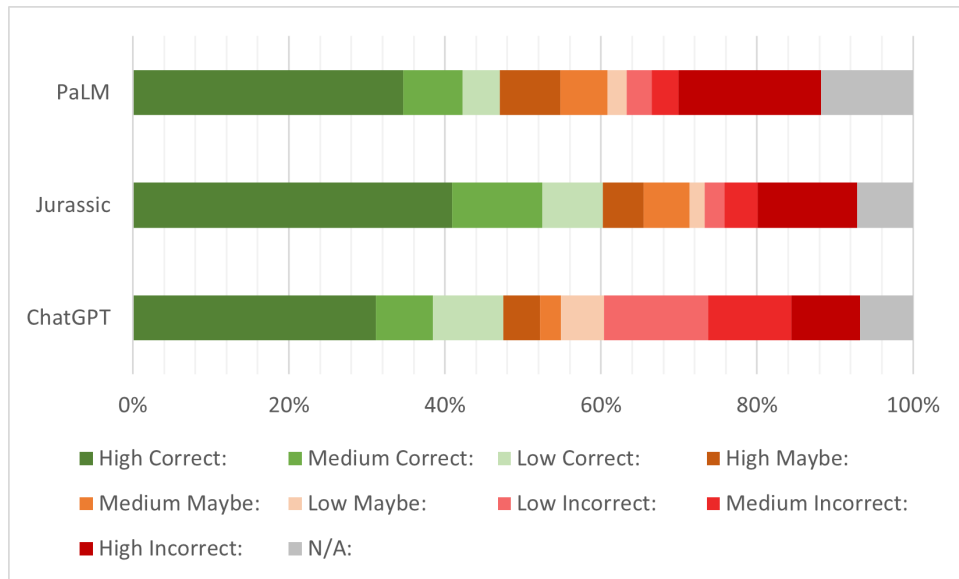


Figure 10 shows how each LM did. Jurassic is the clear winner with around 60% of its responses being correct. ChatGPT and PaLM had a similar rate of correct responses, but ChatGPT had a much lower rate of 'High Incorrect' which gives it the edge over PaLM. It is currently unclear why one LM is better than the other. This would require research into how each LM specifically functions. What can be understood from the results is that Jurassic and PaLM rely heavily on the variable names to identify the code. This enabled them to ignore obfuscations that included complex math. ChatGPT would get caught up in what the numbers represented, causing it to misinterpret the code. Jurassic's usage of variable names was not as much of a crutch as it was for PaLM, so instances where PaLM would only look at the variables and fail, Jurassic could ignore the variable names and see the big picture. ChatGPT's strength over PaLM is that its non-reliance on variable names led to it having the least amount of 'High Incorrect' responses out of all three LMs.

## 3 CONCLUSION

The results shown here showcase the potential that LMs have to help detect obfuscated code, but it also highlights the issues that need to be addressed. Simple things like variable names and comments throw the LMs accuracy off. Jurassic has the potential to be trained to identify the obfuscations shown here and better its ability to be correct.



There are two concerns that should be identified about this research. The first concern is the high rate of answers not asked by PaLM. The number of questions asked to PaLM were lower than the other two, so those unanswered questions could have been the difference between PaLM and ChatGPT. There is also a potential flaw in the correctness rating with the human difference of judgement. Two people rated the correctness for the responses which is two different perspectives. There is a general description for what constitutes each rating but there is still lots of room for subjectivity. To better rank the LMs and obfuscations, a more objective method needs to be created to rate the LMs responses.

Future directions for this research should have different focuses with the end goal of having a LM that can identify obfuscate malware. Research with focus on the questions will need to have questions that are more specific to the problem. Leading questions that hint at the function of the code or that the code is obfuscated. The use of compiled code instead of C++ can help push the LMs ability to understand a range on programming languages. Compiled code is also harder to follow and understand than the normal programming languages. Using obfuscation software. In this project, all obfuscation was done by hand. Obfuscation software will allow the amount of data gathered to expand but also better the obfuscations done. Changing the base code to actual malware will help understand the features of malware that the LMs struggle to understand. Finally, once the previous work has been finished, research can be done to train an LM to identify obfuscated malware.

The research done in this project is just the first step to creating technology that will strengthen cybersecurity efforts which will in turn make the world a safer place.<sup>11</sup>

---

<sup>11</sup>All obfuscated code, base code, and code used for automation can be found here: <https://github.com/SwindleA/ObfuscationDatabase/tree/main>

## REFERENCES

- AI21. 2023a. Jurassic-2 models.
- AI21. 2023b. Python SDK.
- David Berkowitz. 2023. Temperature check: A guide to the best CHATGPT feature you're (probably) not using.
- Hanane D. 2023. Temperature and  $Top_p$  in CHATGPT.
- Google. 2023a. Generative AI: Text Chat.
- Google. 2023b. Module: Google.generativeai: palm API: Generative AI for developers.
- Dan Jurafsky and James H. Martin. 2023. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* (3 draft ed.).
- Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers and Security* 51 (2015), 16–31.
- Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Arun Kumar Sangaiah, and Aniello Cimitile. 2018. Evaluating model checking for cyber threats code obfuscation identification. *J. Parallel and Distrib. Comput.* 119 (2018).
- OpenAI. 2023a. Create Chat Completion.
- OpenAI. 2023b. GPT-3.5.
- Lazy Programmer. 2023. What is temperature in NLP / LLMS?

Table 15. Obfuscation Categories (Part 1)

Obfuscation Identifier	Obfuscation Name	Description
O1	Change Mapping of Variables	Use the same variable names, but switch the data that it applies to.
O2	Incorrect/Misleading Documentation	Write comments that are incorrect or misleading when describing the function of the code.
O3	Change variable names to imply different data type.	Example: std::string numberOfPeople; int name. New names may imply different function but focus was put on different data type.
O4	Insert Unused Variables	
O5	Insert Unneeded Print Statements	Have print statements that either, print variables that do not need to be printed or print information that is incorrect or irrelevant.
O6	Change For-Loops to Recursion.	
O7	Change Variables to Match a Functionality Completely Different	Example: A train booking system: int findTrain; int arrivalTime; int ticketPrice; bool bookTrain;
O8	Change Strings to use ASCII Characters	
O9	Represent Numbers as Unnecessary Math.	Instead of 10, write: (((((1+1+1)*3*5)+5)/10)+5)+(2/3). For integers, the (2/3) truncates to 0. There were instances that the LM would not truncate.
O10	Insert Unnecessary If/Else Statements	Add if-statements that will have no effect on the outcome. For example, an if statement that always evaluates to true.
O11	Insert Unnecessary For-loops	This can be combined with O9 to do unnecessary math.
O12	Remove Spaces and Newlines	

Table 16. Obfuscation Categories (Part 2)

Obfuscation Identifier	Obfuscation Name	Description
O13	Combine O8,O9,and O11	Represent strings in their ASCII form. Represent ASCII characters through confusion math to get the ASCII decimal number you are needing, use loops to further confuse the math. At times, extra variables were used for conversions.
O14	Combine O9 and O11 with Unnecessary Variables	Use complex math, for-loops, and extra variables to represent numbers.
O15	Combine O2-O14	More or less combining every obfuscation method. Also using different languages and alphabets. Excludes O6. Not all obfuscation types may be shown in a given obfuscation. The goal is to try and combine all methods up to this point but not explicitly. When obfuscating, typically started with O13 and when backwards apply obfuscations. O5 was not always used.
O16	Change Output Presentation	Change the way the output is presented. For Example: "123" instead of "321" or "[231]"
O17	Combine O2, O7, O14, and O16	Aimed at making the code function appear different, without having messy code. The comments and variables should match. The for-loops and math can be used to add fake functionality. Changing the output to completely change the look of the info. The goal is to keep it clean, so as to make sure it does not think that it is an obfuscation. Focus was put on the variable names and the comments.
O18		This obfuscation could be categorized as a mix of in-congruent commenting, variable renaming, character replacement, and redundant or misleading operations. It is an attempt at making the code harder to understand at a glance.

Table 17. Base Code

Name	Description
B1	Print the integer numbers 1 to 10, each on a newline.
B2	Print the sum of the numbers 1 to 10.
B3	Print 10 factorial.
B4	Find the factors of 10 and print them on the same line separated by a space.
B5	Concatenate the two strings "race" and "car" together and print the result.
B6	Print the string "hello" 10 times, one line, no spaces.
B7	Concatenate four "Hello" and seven "There" together and print the result.
B8	Print the number of odd numbers in a vector of integers.
B9	Print the odd numbers from a vector of integers on one line separated by a space.
B10	Print the number of vowels in the string "alphabet".
B45	Swap the values of two variables. Print the before and after values of the variables.
B46	Check whether an inputted character is a vowel, upper or lower case. Print true or false.
B47	Find the compound interest for a principal of 10,000, rate of 5 and time of 2. Print the result of the calculation.
B48	Calculate the result of taking two numbers as input and raising one to the power of the other.
B49	Calculate and print the product of two inputted numbers.
B50	Check whether or not the inputted integer is prime. Print "true" or "false".
B51	Print these four strings on separate lines: "adsf!fjelnbo./23@#45jalkd", "as;lkdjfoine!,djfoekngrn", "apple!a;lkdjfoie", "This is the fourth line!"
B52	Four lines are printed. Each line contains an exclamation point preceded and followed by a random amount,0-10, of random lowercase letters.
B53	Checks if a word is a palindrome, excluding the letter 'x'. Prints "true" or "false".
B54	Takes the inputted word and changes the middle character with '>'. Prints the result.
B55	Prints a space followed by 6 newlines.

12

<sup>12</sup>Base codes B11 through B46 were created but not used due to time constraints