Contents

[1.Introduction 2](#_Toc482225137)

[2. Overview 2](#_Toc482225138)

[2.1 Usage (Application User Stories) 2](#_Toc482225139)

[3. Implementation 3](#_Toc482225140)

[3.1 Waterfall model 3](#_Toc482225141)

[3.1.1 Requirements 3](#_Toc482225142)

[3.1.2 Design and Analysis 4](#_Toc482225143)

[4. Testing 7](#_Toc482225144)

[4.1 Unit Testing 7](#_Toc482225145)

[4.1.1 Naming Convention 7](#_Toc482225146)

[4.1.2 Testing Strategy 7](#_Toc482225147)

[4.1.3 Assertion Failure Messages 7](#_Toc482225148)

[4.2 Integration Testing 7](#_Toc482225149)

[4.2.1 Integration Testing Convention 7](#_Toc482225150)

[5. Results and Future Work 7](#_Toc482225151)

[5.1 Results 8](#_Toc482225152)

[5.2 Future Work 8](#_Toc482225153)

## **1.Introduction**

A polymorphic engine (sometimes called mutation engine or mutating engine) is a computer program that can be used to transform a program into a subsequent version that consists of different code yet operates with the same functionality [1]. The polymorphic engine developed in this project will convert the source code of the payload into a subsequent new version with essentially the same functionality but with a different signature and some other changes in the program logic which will help the payload evade detection by the antivirus.

Typically, an antivirus contains a database of malware and their signatures, while scanning a file it checks the signature of that file against the signatures stored in its database. If it finds a hit the antivirus will alert the system about a possible malware in the file. These signatures are not just limited to the hashes of the executable, the antivirus can log the behavior or certain system calls or certain type of registers used by a malware. To evade the malware from being detected by the antivirus, it is required to not only change the byte data but also its functionality.

## **2. Overview**

### 2.1 Usage (Application User Stories)

1. User has the source code for payload
   1. User places the payload source code into a txt file named “Payload.txt”
   2. Runs the PE
   3. The new version of payload is stored in “resource.h”
2. User tries to remodify the payload
   1. User runs PE
   2. Payload source code is decrypted from “resource.h”, PE will morph the source code, compiles the new version and runs the payload
   3. Encrypt the new version of payload and store it back in “resource.h”

## **3. Implementation**

### 3.1 Waterfall model

### 3.1.1 Requirements

The most important requirement is the ability of the malware to evade detection from the anti-virus. It should still do the same functionality for example if it’s a keylogger, it should act like a keylogger but without being detected by the antivirus. This is achieved by implementing 2 types of morphing mechanisms:

1. Change in functionality

Consider a payload whose signature is residing in the database of the Anti-Virus. This signature may include not just the overall hash but also enumeration of the functions a malware performs. For example, a key logger

* 1. Has to remain in stealth mode
  2. Binds to the keyboard input and mouse input
  3. Stores the data in a txt file

Now, the Polymorphic engine aims to add functionality to this keylogger payload which don’t essentially do anything. Such functions are called no-operations. In computer science, a NOP, no-op, or NOOP (pronounced "no op"; short for no operation) is an assembly language instruction, programming language statement, or computer protocol command that does nothing. Therefore, adding no ops will make changes to the procedure of payload by adding functions that execute assembly language instructions that make no changes to the real functionality of the payload but helps in evading detection by the anti-virus. When the anti-virus tries to match the pattern of the morphed payload, it is no longer successful in recognizing the pattern due to the difference in functionality created by the no-ops.

1. Change in logical order

Modern anti-virus have the capability to recognize and signature the logical order of execution by a payload, for instance, the order in which variables are stored in memory can determine the payloads identification. To combat this, it is necessary for the polymorphic engine to change the logical order of execution of a payload. In C++, this can be achieved by using labels and goto statements. Inserting labels and the corresponding goto statements will ensure that the execution of a payload will not be out of order even if the order of instructions is changed on the source code level.

Additionally, the payload is stored in the form of encrypted source code which will make it difficult to read using naked eye. Simple encryption will disable most natural language parsers from understanding the contents of the files used as resources.

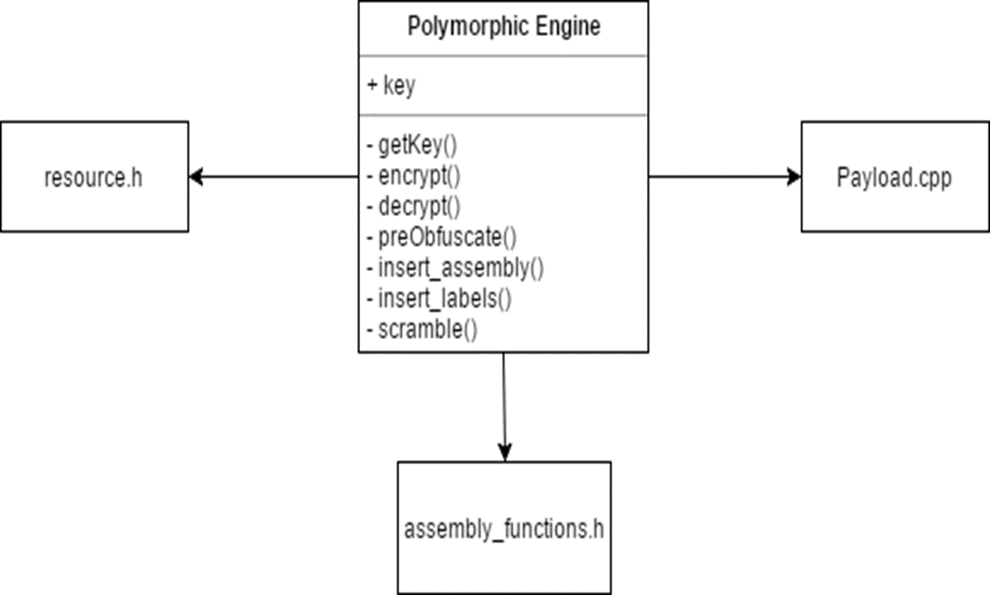
### 3.1.2 Design and Analysis

The polymorphic engine developed comprises of 4 objects:

1. Polymorphic Engine: The class containing all the related functions
2. Resource.h: The header file used to store the morphed payload
3. Payload.cpp: The source code of the original payload
4. Assembly\_functions.h: A list of all the assembly functions used as no-ops.

#### Design Considerations:

One object is created for the polymorphic engine containing all the functionality because it is easier if all the functions remained at one place and the polymorphic engine need not have to access other parts of memory to carry out its execution. A header file is used to store the morphed payload keeping in mind the future considerations of the project where the plan is to import the header file and create one object overall. Another header file is used to store the assembly functions for the same reason. This will make the polymorphic engine to be modular and adding assembly functions will be hassle free and not require remodifying any other part of the code.



#### Modules:

Separate function is used to implement each functionality in the class object of polymorphic engine. Each module is explained in this section.

1. Encrypter

As the name of the function suggests, Encrypter simply converts the plain text source code into cipher text. The input to the function is line of string which means the parsing of the text will be line by line. The encryption key used is a random number between 1 to 10 and the encryption carried out is a simple ascii rotation. This simple encryption should be enough to disable the anti-virus from being able to read the contents. The Encrypter returns the cipher text line of data as a string.

1. Decrypter

The functionality of decrytper is just opposite to that of the encrypter, the input parameters are a line of data in string format and the key used to encrypt the cipher text. The function returns the line of decrypted plain text in string format.

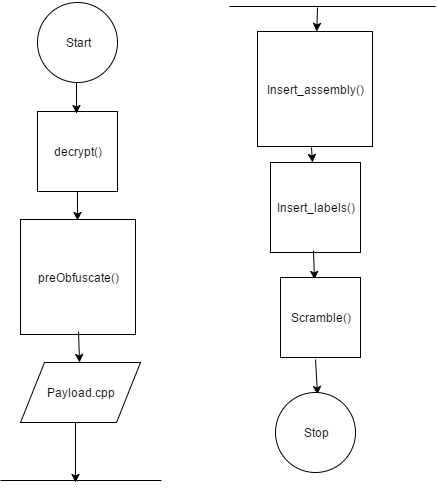
1. preObfuscator

To successfully implement obfuscation at the source code level and still be able to compile it, comments need to be added to the source code itself that can shed light on what’s coming next. For instance, labels cannot be added just before a closing bracket(end of block), opening bracket(beginning of block) and white space outside of main function or any function definitions. The preObfuscator has numerous such cases where labels and goto statements cannot be added, for each case, it will add specific comments. The function takes a line in string format and returns the line adding specific comments if necessary.

1. Insert\_assembly   
   As the name of the function suggests, the main responsibility of this function is to add assembly instructions to the source code of the payload. The function checks for the comments added by the preObfuscator and only if it is safe, the function adds calls to the assembly functions. Further, the assembly functions definitions are added at the very bottom of the soiurce code after everything and they are added the same way they appear in the assembly\_functions.h header file. The function takes input parameter as a line in string format and
2. Insert\_labels  
   Before the logical order of the payload is scrambled or reordered, it is necessary to add labels and goto functions. That is what the insert\_labels function does, add the labels and corresponding goto functions. The labels are of the convention “MyLabel“ followed by a sequential number and the labels are added at the beginning of each statement. The goto statements are added at the end of every instruction, “goto” followed by an increment of the sequential number is the convention implemented. All labels and goto statements are added only to the instructions where it is safe to modify i.e. the comments added by the preObfuscator are checked before adding labels and goto statements. The function takes a line in string format as an input parameter and returns the modified line in string format as the output parameter.
3. Scrambler  
   The Scrambler function simply reorders the statements in the source code where there are labels and goto statements. The reordering is carried out by storing a statement in a variable, wait for the next statement to appear and add the stored statement before or after the next statement.

#### Flow of execution:

Expanding on the user stories, the polymorphic engine begins execution by displaying a prompt for the user to enter choice of morphing a new payload or morph a payload that was morphed at least once before. In case the user chooses to morph the payload for the second time, the polymorphic engine will first fetch the morphed payload from the resource.h header file but before it can do anything further it must decrypt the contents. The key for the decrypter is fetched from the resource.h, the first line of the header file contains the key. After the cipher text is decrypted to plain text by the decrypter, the preObfuscator adds comments to the source code wherever necessary. Then, the insert\_assembly function parses through the pre obfuscated source code and adds calls to assembly functions wherever it is safe to do so. At the end of the source code, the definitions to the assembly functions are added. Further, the insert\_labels method will add labels and goto statements in appropriate parts of the source code therefore making the payload ready for scrambling. Labels and goto statements are added only inside the main function and nowhere else, this is to avoid any type of scrambling among the definitions of other functions. It is also not necessary as the main function is where the execution is carried out. Scrambler can now carry on with reordering of the statements in the main method of payload. Now, the morphing of the payload is considered complete and the polymorphic engine will compile and execute the morphed payload. This is carried out according to the statements present in the main function of polymorphic engine. This comprises of one complete cycle of morphing and the next step is to encrypt the newly morphed payload and store the encrypted source code in the resource.h header file.



## **4. Testing**

Since there is only one class object for the entire polymorphic engine, testing was carried out on individual modules before integrating and during the sequential integration of each module.

## 4.1 Unit Testing

### 4.1.1 Naming Convention

For every function that is a module in the polymorphic engine, a number of testing functions are developed to verify the accuracy of results and functionality. The name of each testing function follows the convention : function\_name.test\_name. For instance, the testing function for encrypter that encrypts a simple line using key 2 is Encrypter.encryptsWithKey2

### 4.1.2 Testing Strategy

After each module was developed, testing was carried out simultaneously. Since each module is a function, the testing function calls the module using appropriate parameters and assertions are made regarding the correctness of the result. The results must correspond to the intended functionality of each module. Various testing functions were developed in order to consider all of the different cases/anomalies that might arise in real time.

### 4.1.3 Assertion Failure Messages

When a test case does not pass, a message is raised with the following format:  
  
PolymorphicEngine.module\_name: <errorMessage>

The error message usually explains the functionality being tested by a test function.

## 4.2 Integration Testing

Once all the unit test cases are developed for each module, the module is integrated into the project. During integration, testing is carried out to verify the compatibility of the new module integrated with the already existing modules. For instance, the decrypter module must decrypt using the same key as that of the encrypter so while integrating the decrypter test functions were developed that call the encrypter and decrypter, and the results are verified.

### 4.2.1 Integration Testing Convention

To easily differentiate between the unit testing functions and the integration testing functions, “I” is added in front of the integration testing function :  
  
IPolymorphicEngine.module\_name: <errorMessage>

The error message lists the modules involved in the integration testing function and the functionality being tested.

## **5. Results and Future Work**

### 5.1 Results

The polymorphic engine was used to morph a payload containing the source code for a keylogger that uses GetAsyncKeyState method to get the state of the keyboard. This particular method was recognized by various anti-viruses, the proof of this was established by compiling a version of the same payload without this particular method. The resulting payload after the polymorphic engine morphed the original payload could successfully evade detection by most antiviruses listed in the website “virustotal.com”. Upon further research related to the morphed payload, it was understood that the main reason most antiviruses could not detect the morphed payload was the call to an assembly function right before the method call for GetAsyncKeyState.

### 5.2 Future Work

Although the polymorphic engine developed in this project could successfully evade detection by most antiviruses there is still a lot of room to perfect. The future work related to polymorphic engine can involve further reordering of the variables used. The easiest way to achieve this is to build a better parser that can identify variables and change the order in which they were declared. Additionally, the sizes and types of the variables may be changed which would result in a signature that is much more different from the original payload.

Also, the polymorphic engine is very language centric, meaning, it can only morph source code that was written in C++. There exist better parsers that can recognize compile code and morph the functionality of a program at the binary level. Using such parsers will result in more obfuscation and change in functionality.