Hybrid Algorithm to Protect Java's Code from Reverse Engineering

Asmaa M. Alhakimy and Abu Bakar Md. Sultan

***Abstract*—Reverse engineering (RE) is a process that begins with disassembly, which attempts to translate machine language code to assembly code. This process creates a potential opportunity for theft of source file via software by allowing the attacker to discover secrets of the original code, which leads to financial crisis to the author. Anti-reverse engineering is the implementation of techniques that delay the attempts of prohibited (RE). This study is to discuss the impact of RE, and propose best solution for it. Total five different java applications are used for the experiment. Findings from the experiment reveals that all java applications are possible to be broken and there is no 100% protection, The goal of this research is to propose a new method that protects java's source file during implementation phase to reduce readability, and increase security, by delaying RE to certain time, whereas the reverser feels frustrated to continue breaking.**

***Index Terms*—Anti-reverse engineering, intellectual property reverse engineering, software security.**

I. Introduction

Reverse engineering is mainly used as a systematic methodology to extract design's information, function's specifications and requirements from the program's code. The process of reverse engineering can be used to find the relationships among the system's components, such as, functions, classes and so on. The process of reverse engineering is providing a representation of the system in a higher level of abstraction from lower level. The main benefit of this process is to get new product with new features in less budget, time, better quality, and competitive advantage.

Develop once and run anywhere, is an incredible feature of java that made it very popular. When java program is compiled, a class file will be generated. The class file contains all reserved information in bytecode format. Therefore, illegal reverse engineering becomes easier. Reverse engineering attacks the class file and converts the bytecode to a readable language. This process has cost authors to lose their software's digital copyrights. When reverse engineering curved into a dangers process, many anti-reverse engineering techniques and tools were developed to prevent it. Tools such as packing, Y-guard, Stealthy code, Key hiding and so on.

Reversers was able to break most of these tools, hence they are no longer effective to protect the applications from reverse engineering [1].

In this research, new algorithm will be proposed. It contains a mathematical formula and binary code to convert the Java English written words to symbols. An experiment was conducted to test the algorithm on samples of java class files to observe the effectiveness of it. Total five java applications are used for the experiment. The applications are from different categories to ensure valid and accurate results, the categories are Functional, Object oriented, Mobile, Graphic and Database Two de-compilers were used for the experiment to reveal erroneous codes generated by them. The quality measurements used to verify the algorithm are *Readable variables* to guarantee unreadability when application is reversed, *Error free* to guarantee that application is running perfectly after injecting the new algorithm, *Discovered lines* to count the number of readable lines after reversing, the less lines discovered is the better, *Functionality* to compare before and after injecting the algorithm if the functionality is still the same, if still same, then it is a good sign, *VBP functionality* to match before and after injecting the algorithm, as a good sign they have to match. If any of the quality measurements reveals different results, then amendments of the algorithm will be carried out until getting the accurate results. There will be a description of the mechanism of the new algorithm and how does it work. Including lab test results.

The analysis phase will be carried out to test the outputs of the experiment and match them against the research questions and hypothesis and find out the relationship between them. Systematic analysis will run against the results of systematic literature review to find out the relationship between them and the findings from the experiment as well, to draw the final form of the algorithm.

As for this paper, results are shown for the first experiment with one of the reversing tools. As this research goes further, the new results will be published for the next conference.

The outcome of this research is a new hybrid obfuscation technique that can be injected in to java source file. This technique can prevent reversers from copying the source code, and prevent students from copying code illegally without authorization.

II. Problem Statement of Reverse Engineering

When reverse engineering has become dangers, many cases have filed to the court seeking justice of their stolen codes and copyrights, many companies have fought in the courts, each claiming the authority of the intellectual property, but the one with good lawyer will always win the case[2].

Manuscript received February 12, 2014; revised May 15, 2014. This work was supported by a grant from the Research University Grant Scheme (RUGS) from the Universiti Putra Malaysia, Malaysia. (We do thank UPM University to the support that it is giving for this research).

The authors are with the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, 43400, Selangor, Malaysia (e-mail: selfemoon@ gmail.com, abakar@upm.edu.my).

Along the way, when situation became complicated, the courts have built up a new law to prevent it and protect digital copyrights, but law as standing alone cannot prevent it whereas the ones with good lawyer will always win the case, so what is the guarantee the law will protect copyrights?. So technical solutions have come out, those are obfuscation and encryption techniques. According to Jan M. Memon, 2006. Many Anti-reverse engineering techniques were developed when law was no longer effective to stop the threat of illegal reverse engineering, some of these tools are strong, and some are weak, but both of them are not stopping the reverse engineering, yet they are delaying it [3].

Christoph Treude, 2011, is supporting that these tools are meant to slow down the process, and make it more painful and difficult; then, the reverser would give up. Most known anti-reverse engineering technique is *obfuscation*; this technique has many styles, such as, hiding the classes, changing the flow of the program, changing variable names, and even making the code very complicated and difficult to understand [4].

Our focus in this research is on obfuscation techniques. A. M. Al-Hakimy, K. P. Rajadurai, M. I. Ravi, 2011, have proposed a new way of obfuscation that is using three elements to protect the code, they are mathematical formula to change the messages, binary code for variables, and Unicode for system keywords. The problem with this technique is that the Unicode can be translated easily by using any De-compiler, and the mathematical technique does not perform well in nested loops [5].

According to S. M. Darwish, S. K. Guirguis, 2010, obfuscation known as the best way to protect the code, but it depends to the form used to protect the code, when the obfuscation technique is very complicated is the best to delay the time of breaking the code and confuse the reverser while reading reversed code. Not all obfuscation techniques work very well. For example changing the form of the code is no longer protecting the code, as the reversing tools can also reform the code. Hiding classes as a stand-alone technique will not help protecting the code, there must another technique to supported, such as encrypt the hidden classes [6].

Xiang Guangli, Cai Zheng, 2010, have proposed class combination obfuscation, this way can only join several java class into one class, but it does not change anything in the code, which means reverses can just un-join the classes since they are very good in programming [7].

Christian S. Collberg, 2002, also agreed that obfuscation is wildly used and proven to be best technique to use to protect java application whether java applets for online use or networking or even desktop applications. But it depends to the way it is used as well. One of the bad obfuscation technique is Lexical Transformations, this technique is used for java applications, typically, they do nothing more than scramble identifiers. Such lexical transforms will surely be annoying to a reverse engineer and, therefore, will prevent some thievery of intellectual property in software. However, any determined reverse engineer will be able to read past the scrambling of identifiers in order to discover what the code is really doing [8].

By looking at the current state of technology evolution, we can see that reversers have advance tools that can break and extract code in short time, which means that even using trail key protection is no longer effective, because the reverser is able to break the code with in few seconds for small programs and less than an hour for bigger programs. Most of the obfuscation techniques are based on one element to protect the code, which is not enough to delay the process of reverse engineering, and it is not enough to confuse the reverser. Furthermore the evaluated obfuscation techniques are not looking into transforming the code into another form in the source file before compilation, for example ("welcome to java world") becomes ***("²¶**´¶µ Êþº¾ 2 k 8 9 : ; <=@ ¶ µ **´ *´ cµ ² » Y")***. To achieve this kind of transformation, there must be more than one technique used and injected into the code. To achieve such transformation, the algorithm must be hybrid of more than two elements to ensure perfect changing of the form and wording of the code.

III. Proposed Solution

According to previous studies, it was obvious that using a technique to protect java's source file rather than class file is most effective, even though it has some problems, but these problems can be solved as development grows further.

Proposed technique is still under reviewing. It is based on several steps; first step, is to convert system or key words to Unicode which will help to transform the look and form of the code which will confuse the reader. Second step is to use mathematical formula to convert messages into unreadable form in which can be read while printing. The mathematical formula helps transforming the code while reversing. Third step, is to convert variables into binary code to confuse the reader. These steps will give the revere very hard and long time to understand the obfuscated code. Fig. 1 shows the process of conversion.

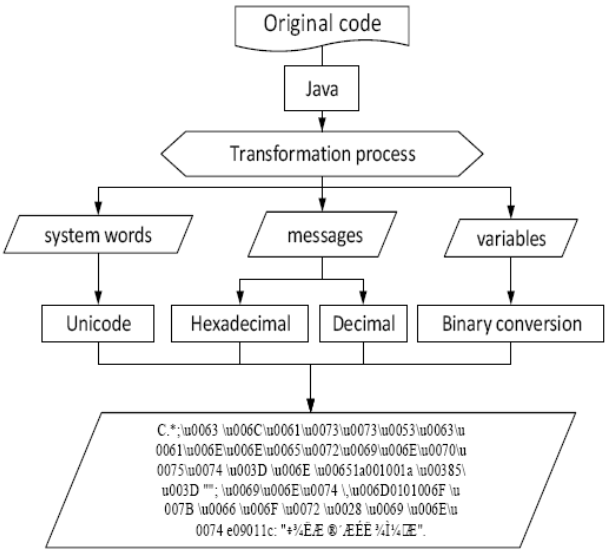

Fig. 1. Transformation of java's code process.

IV. Technique Implementation

Java programming language will be used to implement the technique. The first sample of code will be functional math application. The application contains some calculation and

then display some text on the screen. After that, the code will be compiled into class file which contains garbage code, then a reversing tool will be used to test how much can be uncovered from the code, then will inject the new algorithm into the original code, and test again by using the same reversing tool to find the difference with and without using the algorithm. The environment used for this implementation is Netbeans 7. Fig. 2 shows the original code before injecting the algorithm and before reversing.

A. *Sample of Code before Using the Algorithm*

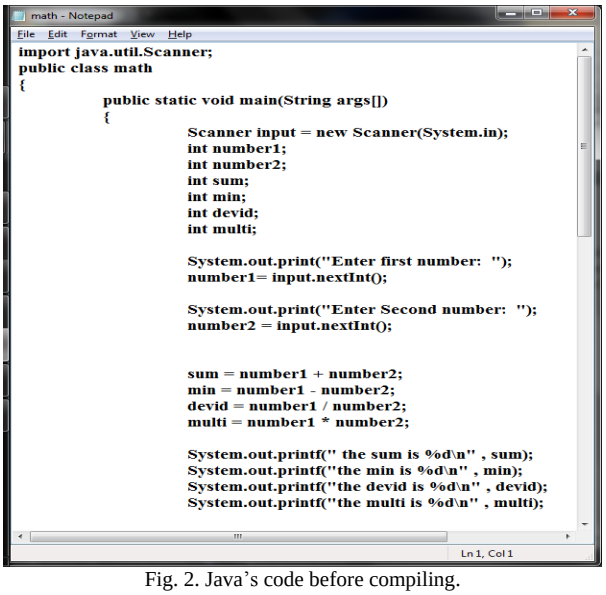

Fig. 2. Java's code before compiling.

- When this program is compiled, a class file will be generated, it contains the garbage code which is unreadable. The displayed garbage code is a mixture English words, symbols and numbers. Fig. 3 shows the code after compiling.

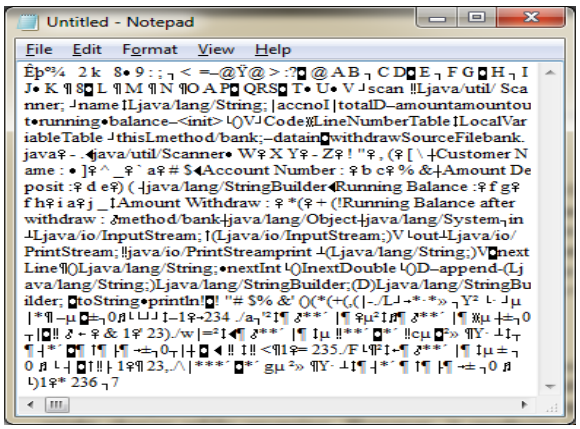

Fig. 3. Java's class file.

- To demonstrate theory of the ability to break java's class file, a reversing tool is used to break the application after compiling, this tool is CAVAJ. The class file will be dragged into CAVAJ platform, then, the original code will be generated. Fig. 4 shows the original code after reversing.
- The code after reversing has changed in form, variables names have changed as well. But both shows the same result for output. The reversing tool was able to read the variables and classes' names. In addition it had included extra class for the application. Number of lines methods and libraries have changed as well. Table I shows the differences between original and reversed file.

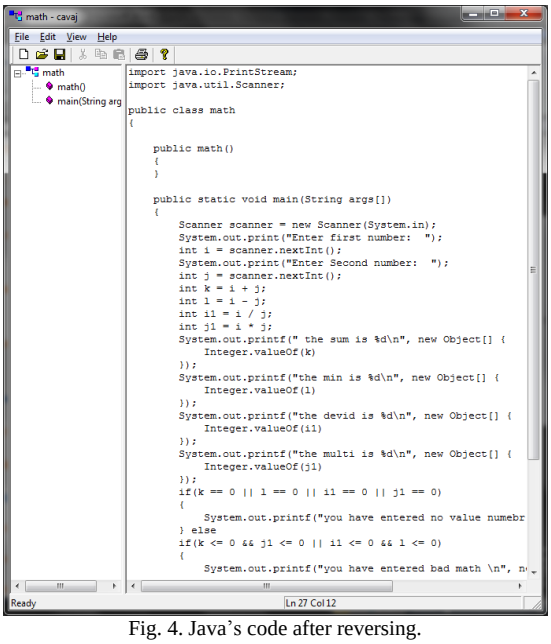

Fig. 4. Java's code after reversing.

TABLE I: COMPARISON BETWEEN ORIGINAL AND REVERSED FILE

	Original File	Reversed File
Elements	Quantity	Quantity
Number of lines	44	48
Number of Libraries	1	2
Number Of methods	1	2
	46	52
	6	

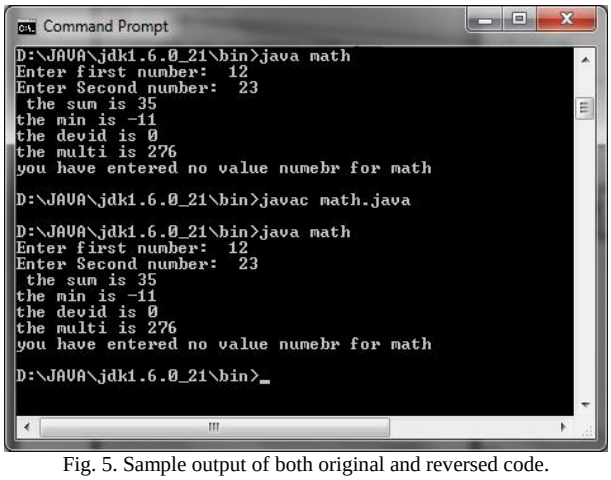

Fig. 5. Sample output of both original and reversed code.

There is major differences between the original and reversed code. Next step is to find the percentage of the differences between the two codes as follows;

$$\frac{6\times 100}{46+52}=\frac{600}{98}=\%6.12$$

The differences between the original file and reversed file are 6 points extra information given by the engineering tool. Therefore, the reverser is able to do good analysis on the code,

moreover, the reversing tool usually does detailed analysis on the code for further amendments.

Fig. 5 shows the same output from original and reversed code. The first set of output is from original file, second set is from reversed file.

- Since the tool is able to read the system key words, then they have to be converted into Unicode. It is not possible to play with these words, as the compiler will be confused while reading them, so converting them to Unicode will have an impact with less expert reversers who are still new in this area, as they will not be able to read it. Unicode works with letters not words so we can convert every letter for example **(\u0069 \u006D \u0070 \u006F)** these are four letters which the compiler can read perfectly.
- If the program contains messages, they can be converted as well using different way of conversion, it can be mathematical or binary. Mathematical is used for this case, but still under implementation and testing. First result is **(◆Á¾Ë°¬¤Î¬).** This sample contains alphabets only. Currently, in order for this sample to be readable by the compiler it must be in an array. The reverser will not be able to read this sample during implementation phase from the source, it is only readable when the program is executed.
- The variables of the program can be converted as well. The possible way is now under testing. Every character can be converted into a binary form without considering the arrangement of the word. Example for this kind of conversion is **(001001)**. In order for this conversion to be done we have to use an array to store the characters. Next section will show some sample of code after conversion.

B. Expected Sample of Code after Using the Algorithm

\u0069\u006D\u0070\u006F\u0072\u0074\u006A\u0061\u0076\u0061.\u0075\u0074\u0069\u006C.*;\u0063\u006C\u0061\u0073\u0073\u007601010101010r\u007B\u0053\u0063\u0061\u006E\u006E\u0065\u0072\u0069\u006E\u0070\u0075\u0074\u003D\u006E\u0065\u0077\u0020\u0053\u0063\u0061\u006E\u006E\u0065\u0072\u0028System.in\u0029;\u0053\u0074\u0072\u0069\u006E\u0067a001001a\u003D"";\u0053\u0074\u0072\u0069\u006E\u0067b001100\u0185\u003D"";\u0069\u006E\u0074\u00730111\u0075010\u006D12,\u006D010100\u00690101000\u006E0101088,\u006440011 ¤\u0069 ¢\u0076\u0069 ¥\u0064160\u0065,\u006D100100100\u0069;\u0069\u006E\u0074n101010r1;\u0069\u006E\u0074n01010102;\u0070\u0075\u0062\u006C\u0069\u0063\u0020\u0076\u006F\u0069\u0064\u0020\u006Doo111\u006D\u0028\u0029\u007B\u0064\u006F\u007B\u0066\u006F\u0072\u0028\u0069\u006E\u0074e09011c:

"◆ Ë Æ®ÆÉË Ì ◆ Æ .\u0074\u006F\u0043\u0068\u0061\u0072\u0041\u0072\u0072\u0061\u0079\u0028\u0029\u0029\u007B\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074\u0028\u0028\u0063\u0068\u0061\u0072\u0029\u0028e09011c/2\u002B17-2\u0029\u0029;\u007D\u0074\u0072\u0079\u007Ba001001a=input.nextLine\u0028\u0029;\u0069\u0066\u0028a001001a.equals\u0028""\u0029\u0029\u007B\u0066\u006F\u0072(\u0069\u006E\u0074u767d:

:" ¬ É¬◆ Ë Æ®ÆÉË Ì ◆ Æ◆Á Ë ¬ Î¬Ë◆

\u0074\u006F\u0043\u0068\u0061\u0072\u0041\u0072\u0072\u0061\u0079\u0028\u0029\u0029\u007B\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074\u0028\u0028\u0063\u0068\u0061\u0072\u0029\u0028u767d/2\u002B\u0029\u0029;\u007D\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074("\n\n");\u007D\u0065\u006C\u0073\u0065\u007Bn101010r1=Integer.parseInt\u0028a001001a\u0029;\u007D\u007D\u0063\u0061\u0074\u0063\u0068(Exceptione)\u007B\u0066\u006F\u0072\u0028\u0069\u006E\u0074i878g:

" ¬ É¬◆ Ë Æ Ì ◆ Æ®ÁÆ ËÁ Õ ÁË ◆ ◆¬Ë◆ .\u0074\u006F\u0043\u0068\u0061\u0072\u0041\u0072\u0072\u0061\u0079\u0028\u0029\u0029\u007B\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074\u0028\u0028\u0063\u0068\u0061\u0072\u0029\u0028i878g/2\u002B17\u0029\u0029\u007D\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074\u0028"\n\n"\u0029;\u006Doo111\u006D\u0028\u0029;\u007D\u007D\u0077\u0068\u0069\u006C\u0065\u0028a001001a.equals\u0028""\u0029\u003D\u003D\u0074\u0072\u0075\u0065\u0029;\u007D\u0070\u0075\u0062\u006C\u0069\u0063\u0076\u006F\u0069\u0064\u006D00112\u006D\u0028\u0029\u007B\u0064\u006F\u007B\u0066\u006F\u0072\u0028\u0069\u006E\u0074g1111g:

"◆ Ë ÆÉ¬◆Á ◆ Ì ◆ Æ Ì ◆ Æ .\u0074\u006F\u0043\u0068\u0061\u0072\u0041\u0072\u0072\u0061\u0079\u0028\u0029\u0029\u007B\u0053\u0079\u0073\u0074\u0065\u006D.\u006F\u0075\u0074.\u0070\u0072\u0069\u006E\u0074\u0028\u0028\u0063\u0068\u0061\u0072\u0029\u0028g1111g/2\u002B17-2\u0029\u0029;\u007D\u0074\u0072\u0079\u007Bb001100\u0185\u003Dinput.nextLine\u0028\u0029;\u0069\u0066\u0028b001100\u0185.equals\u0028""\u0029\u0029\u007B\u0066\u006F\u0072\u0028\u0069\u006E\u0074d10010d:

" ¬ É¬◆ Ë ÆÉ¬◆Á ◆ Ì ◆ Æ ◆Á Ë ¬ Î¬Ë◆

This is not full version, since it is still under testing and implementation. Results are not confirmed yet. However, we can provide sample output to compare with original code. Fig. 6 shows the output after injecting the algorithm into the code.

C. The Sample of Output after Injecting the Algorithm

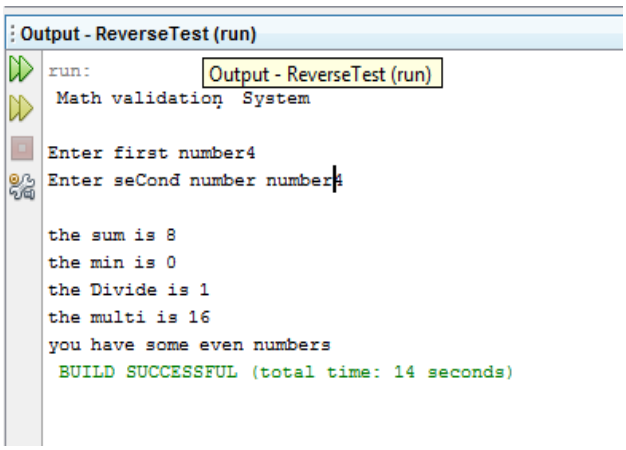

Fig. 6. Sample output after injecting the algorithm.

The output after injecting the new algorithm is similar to

the output before the injection. The results of the calculation was accurate as well.

Next step is to reverse the obfuscated code to test the ability of reversing tool to read it, and how much percentage can the reversing tool uncover.

D. The Results after Testing the Obfuscated Code

The converted code was reversed using CAVAJ tool to try to break obfuscated code. Surprisingly the tool was helping to increase the complication while reversing. This means, the reverser will have hard time to read the code, especially when the technique and reversing tool are help to transform the code more than once. The reversing tool was not able to find tree classes as well. Fig. 7 shows the output when the reversing tool is used against the obfuscated code.

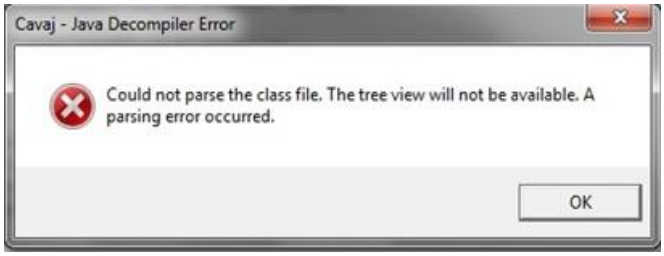

Fig. 7. Message appears after reversing obfuscated code.

The above message appears when the class file is dragged into the CAVAJ tool to read the code, the reverser has to click OK to proceed. Fig. 8 shows the generated code after clicking OK on the error message.

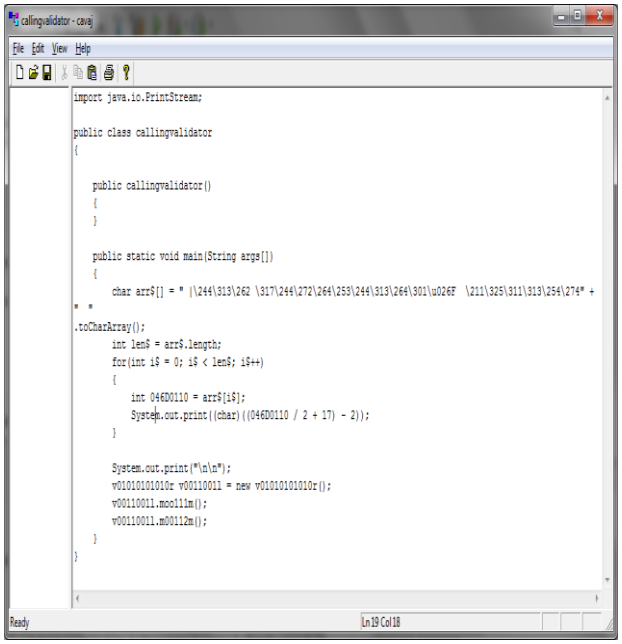

Fig. 8. Obfuscated code after reversing.

From the above results, following points are recorded

- The Tool is not able to define public, private, and protected class.
- Variables names are not clear.
- The tool is not able to read the messages, they have been converted to numbers.
- Over all flow have been changed.
- The tool is not able to read the file name.
- Symbol $ is added with variables.
- Util library is removed and replaced with (io).
- Form of object declaration has changed.

The code after reversing was tested to see if it is possible to be compiled. After testing, total lines and errors are recorded, sample of errors messages while reversing are recorded as well. Table II shows number of entities covered by the reversing tool.

TABLE II: RECORDING OF DATA AFTER REVERSING OBFUSCATED FILE

Item	Quantity
Total lines	197
Total Errors	33
Differences	164
Percentage	21%

From the above table we can see the final result is 21% far from accurate and success. After reversing the obfuscated code, a test conducted to examine if Netbeans 7 can read this code or not. The result of the test was that the compiling tool was unable to read the obfuscated code after reversing, and while running the application several errors occurred, Fig. 9 shows the errors that appeared after compiling. The errors are generated by Netbeans 7 compiler, and copied exactly as they are.

C:\Users\Asmaa\Documents\NetBeansProjects\UniReverse\src\Main.java:185: cannot find symbol
symbol: variable e
location: class v01010101010re="\325\301\314\262
\24\316\255\311\301\274\255\255\316\255\276\276\314\274\u02EC"+C:\Users\Asmaa\Documents\NetBeansProjects\UniReverse\src\Main.java:188: cannot find symbol
symbol : variable e
location: class v01010101010r arr$ = e.length;
int,char[] for(int i$ = 0; i$ < arr$; i$++)
C:\Users\Asmaa\Documents\NetBeansProjects\UniReverssrc
symbol : variable e
location: class v01010101010r int k110110s = e[i$];
33 errors (this is only a small part of the errors)

Fig. 9. Errors occurred when compiling reversed obfuscated code.

As noticed, neither the compiler nor the reversing tools were able to read the obfuscated code and the messages. This conversion does not work with complicated code, as it will change the outputs of the program, only if the program uses deep nested if-else loops. Current work is to enhance the algorithm to work with simple and deep complicated nested loops.

V. CONCLUSION

Implementing a defensive technique to protect the code might not be very useful, although it can protect the code up to some level. The value of the protection level will fluctuate to how strong or weak the technique is. It is very important to admit that all applications need to be protected specially the ones with special business rules and secret ideas, reverser usually attacks these applications. Current state indications

that all reveres and companies are more interested to break complicated expert systems rather than implementing fresh ones, to save the writing time. Implementing such technique will make them to struggle to understand the obfuscated code and it will require more time to find out the lifecycle of the implemented logic as well. Therefore, by adding all this time together the total will be as follows;

Total lines of original code = 44

Total lines after encryption = $\frac{142}{3} = 47.33$

Total hours to translate lines = 47

Total lines after reversing = $\frac{197}{3} 65.66$

Total hours to fix reversed file = 56

Result = 44

According to this calculation, it is possible to know the total hours taken if the original program takes many more lines than what was tested. The methodology used to gather the information was experiment. As this research drives further, a systematic literature review will be used to collect more information about anti-reverse engineering tools and techniques. Moreover, mathematic algorithms are being used to prove the statement in an accurate way and according to the questions type and the purpose of it. More results will be exposed with different reversing tool and then will be compared with previous results from CAVAJ.

VI. Current and Future Enhancement

The problem with the current technique is that it does not work with the complicated code that contains nested loops, it changes the output of the program this problem is because of the mathematical formula used. Therefore, the plan is to enhance this technique to be applied with nested loops with accurate output and to make it work with object-oriented cod. Next step of the plan is to implement the tool that contains the technique and add encryption and transformation options for java, C, and C++. It can be applied for open source as well for the copyrighted code. The research can be further enhanced to protect mobile video conferencing, mobile applications, online applications, enterprise applications, software design and further more.

Acknowledgment

First of all Thanks to god for giving us the strength to complete this work. Many thanks for Good supervisor Dr. Abu Bakar. And to all participants who participated in the research. To Mr. Del Myers for his participation in the research. To my parents who were always supportive, and finally to my friends and colleagues.

References

[1] C. Cifuentes and A. Fitzgerald. (March/April 1999). Is Reverse Engineering Always Legal? *IEEE computer society*. [Online]. pp. 42-48. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=774940

[2] F. Bo, Z. Miao *et al*., "Runtime protecting system for java applications with dynamic data flow analyzing," presented at the IEEE, China Wuhan, May 21-24, 2010.

[3] J. M. Memon *et al*., "Preventing reverse engineering threat in java using byte code obfuscation techniques," presented at IEEE Pakistan, Nov. 13-14, 2006

[4] C. Treude, F. F. Filho, and M.-A. Storey, "An exploratory study of software reverse engineering in a security context," presented at IEEE Republic of Ireland Limerick, Oct. 17-20, 2011.

[5] K. P. Rajadurai *et al*., "Formulating a defensive technique to prevent the threat of prohibited reverse engineering," *International Conference and Workshop on Current Trends in Information Technology*, p. 83, Dubai CTIT, 2011.

[6] S. M. Darwish, S. K. Guirguis, M. S. Zalat, "Stealthy code obfuscation technique for software security," presented at the IEEE, Egypt Cairo, Nov. 30-Dec. 2, 2010.

[7] G.-L. Xiang and Z. Cai, "The code obfuscation technology based on class combination," presented at IEEE, Hong Kong, August 10-12, 2010.

[8] C. S. Collberg, "Watermarking, tamper-proofing, and obfuscation tools for software protection," *IEEE Transactions on Software Engineering*, August 2002.

Asmaa Mahfoud was born in Egypt, in 1985. She will get her PhD in software engineering. She obtained diploma in computer programming, University of Science and Technology, Sanaa, Yemen, June 2005, and B.Sc. (Hons) computer studies, Northumbria University, United Kingdom, July 2008. She received her M.Sc. in software engineering, Staffordshire University, United Kingdom August 2011. Currently she is a PhD student in the Faculty of Computer Science and Information System at UPM University, Serdang, Malaysia. Her research interest is software engineering and security.

She worked in Calvalley Petroleum as an IT technician, Versatile Paper Boxes as MIS Administrator, Ahlan Press as Graphic designer, and currently as a lecturer for programming and security in APU University, Bukit Jalil, Malaysia since 2011. She Had published a paper in 2011 for master dissertation Al-Hakimy, A. M. Rajadurai, K. P. Ravi, "Formulating a Defensive Technique to Prevent the Threat of Prohibited Reverse engineering," in *Proc. International Conference and Workshop on Current Trends in Information Technology (CTIT)*, 2011, p. 83. Ms. Al-Hakimy is a member of student ACM.

Abu Bakar Md. Sultan was born in Melaka, Malaysia in 1965. He held a PhD in artificial intelligence from University Putra Malaysia (UPM) in 2007.

His main research areas are artificial intelligence and software engineering, particularly search-based software engineering (SBSE). He had published several articles in conferences and various journals related to SBSE. Associate professor Dr Abu Bakar Md. Sultan currently is the head of Information System Department, Faculty of Computer Science and Information Technology, UPM.