ISSN No. 0976-5697

Volume 4, No. 3, March 2013 (Special Issue)

International Journal of Advanced Research in Computer Science

CASE STUDY AND REPORT

Available Online at www.ijarcs.info

A Case Study: Code Befuddlement Proficiencies for Assisting Software Enigmas

Jitendra Sharma*
M.Tech. Scholar, Dept. of Computer Science & Engg.
Swami Keshvanand Institute of Technology
Jaipur, India
jitendra0511@gmail.com

Amit Solanki
M.Tech. Scholar, Dept. of Computer Science & Engg.
Swami Keshvanand Institute of Technology
Jaipur, India
amit.solanki48@gmail.com

Karishma Sharma
P.G.D.M. Scholar, Dept. of Decision Science (IT & Operations)
Shanti Business School, Ahmedabad, India
sharmakarishma91@gmail.com

Abstract: To prevent unauthorized reverse-engineering of programs and algorithms is a major problem for the software industry. To access unauthorized such codes are easy to decompile, they increase the risks of malicious reverse engineering attacks. Reverse-engineers search for security holes in the program to exploit or try to steal competitor's vital algorithms. Obfuscating code is, therefore, also a compensating control to manage the risks. It also provides several code obfuscation techniques that have been reviewed for technical protection of software secrets. Code obfuscation is viable method for preventing reverse engineering. The obfuscator is based on the application of code transformations, similar to compile optimizers. It also gives description about large number of such transformations and their classification. The transformations are evaluated with respect to their potency, stealth, resilience and cost. As the internet evolves rapidly, software piracy is rampant in the world; as a result software protection becomes a vital issue in computer industry. The code obfuscation can also use to increase the level of software secrecy with integration of other technology. Programs known as obfuscators operate on source code, object code, or both mainly for the purpose of deterring reverse engineering, disassembly, or de-compilation. Obfuscating code to prevent reverse engineering is typically done to mange risks that stem from unauthorized access to source code.

Keywords: Code obfuscation, code tamper-proofing, obfuscator, functionality, efficiency, potency, resilience.

I. INTRODUCTION

As the interest evolves rapidly, software piracy is rampant in the world; as a result, software protection becomes a vital in current computer industry and a hot research topic [1], [2], [3]. Software piracy has been causing enormous losses for software vendors [4]. Given enough time and resources, even a protected program can eventually be reverse-engineered. As a result, having gained physical access to the application, the reverse engineer can decompile it (using decompiles/LEX/YACC) and then analyze its data structure and control flow with the program source code.

Over the past ten years, reverse engineering research has produced a number of capabilities for analyzing code, including subsystem decomposition, concept synthesis , design, program and change pattern matching , program slicing and dicing, analysis of static and dynamic dependencies, object-oriented metrics , and software exploration and visualization. In general, these analyses have been successful at treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps [5]. This can either be done manually or with the aid of reverse engineering tools. Several research issues, formulated as questions, need to be addressed to enable this capability for "continuous program understanding".

a. What are the long-term information needs of a software system?
b. What patterns of change do software systems undergo?
c. What mappings need to be explicitly recorded?
d. What kind of software repository could represent the required information?
e. What are the requirements of tool support to produce and manipulate the mappings?
f. How can this support coexist with traditional, code dominated tools, users, and processes?

In addition to an emphasis on "continuous program understanding," it is important to focus efforts on a better definition of the reverse engineering process. Reverse engineering has typically been performed in an ad hoc manner.

To discourage reverse-engineering, developers use a variety of static software protections to obfuscate programs [6]. Three techniques have been used for software copyright protection [7][8], code obfuscation, software watermarking and code tamper-proofing. Obfuscation code (or beclouding) is the hiding of intended meaning in communication, making communication confusing, willfully ambiguous, and harder to interpret. It is the process of transforming byte code to al less human-readable format; making it hard to be decompiled or analyzed even it is decompiled. It

CONFERENCE PAPER
II International Conference on
"Advance Computing and Creating Entrepreneurs (ACCE2013)"
On 19-20 Feb 2013
Organized by
2nd SIG-WNs, Div IV & Udaipur Chapter , CSI , IEEE Computer Society Chapter India Council ,
IEEE Student Chapter Geetanjali Institute of Technical Studies, Udaipur, Rajasthan, India

includes stripping all unnecessary information, local variable names, line numbers and source file names used by debuggers from the classes, and renaming classes, interface fields and method identifiers to make them meaningless. Software watermarking involves embedding a unique identifier within a piece of software, to discourage software theft.[4] Watermarking does not prevent theft but instead discourages software thieves by providing a means to identify the owner of a piece of software and/or the origin of the stolen software. It can then be extracted by an extractor or verified by a recognizer to prove ownership of software.

The former extracts the original watermark, while the latter merely confirms the presence of a watermark. Tamper-proof mechanism that is embedded in the dynamic data structures of a program. Tamper-proofing method is based on transforming numeric or non-numeric constant values in the text of the watermarked program into function calls whose value depends on the watermark data structure. Under reasonable assumptions about the knowledge and re-sources of an attacker, we argue that no attacker can be certain that they have altered our tamper proofed watermark unless they take a risk of acting program correctness in some way that may be difficult to detect [7] [8].

Software protection is a promising technology to cope with various malicious or illegal accesses to mission critical servers. Code obfuscation has been proposed as the solution to problems such as protection of transient secrets in programs, protection of algorithms, license management for software, protection of digital watermarks in programs, software based tamper resistance and protection of mobile agents [9][10].

The goal of software protection and obfuscation is to make the reverse engineering process more costly than developing the program separately. The developers create new protections which reverse engineers create new tools to break.

II. CODE OBFUSCATION

Obfuscation means “to make difficult to perceive or understand” or ‘making something less clear and harder to understand’. Code obfuscation in programming world means making code harder to understand or read, generally for privacy or security purposes. Code obfuscation makes it harder for a security analyst to understand the malicious payload of a program. In most cases an analyst needs to study the program at the machine code level, with little or no extra information available, apart from his experience.Security through obscurity has long been viewed with disregard in the security community. However, there are applications where obscurity can provide a higher level of protection to its source code.In computing, obfuscation is used to transform the code into a form that is functionally identical to the original code but is much more difficult to understand and reverse engineer using tools.

We are not assuming here that obfuscation will make the code impossible to reverse engineer. The aim is to increase the cost of reverse engineering the code, so that it becomes infeasible. There should be a significant difference between the time needed to obfuscate and the time needed to deobfuscate.

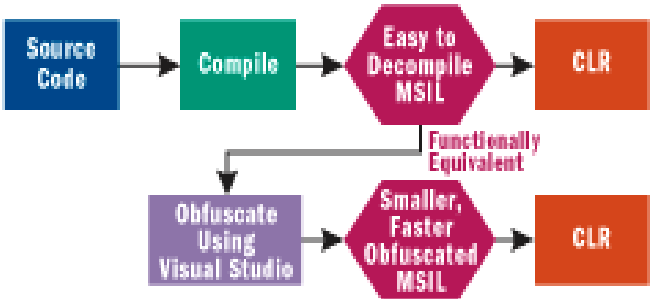


Figure 1. Obfuscation Process

Obfuscation is a process that is applied to compiled .NET assemblies, not source code. An obfuscator never reads or alters the source code. Figure 1. shows the flow of the obfuscation process. The output of the obfuscator is another set of assemblies, functionally equivalent to the input assemblies, yet transformed in ways that hinder reverse engineering.

A concise formal description of code obfuscation is described below [11] and figure represents the complete functional block diagram for code obfuscation.

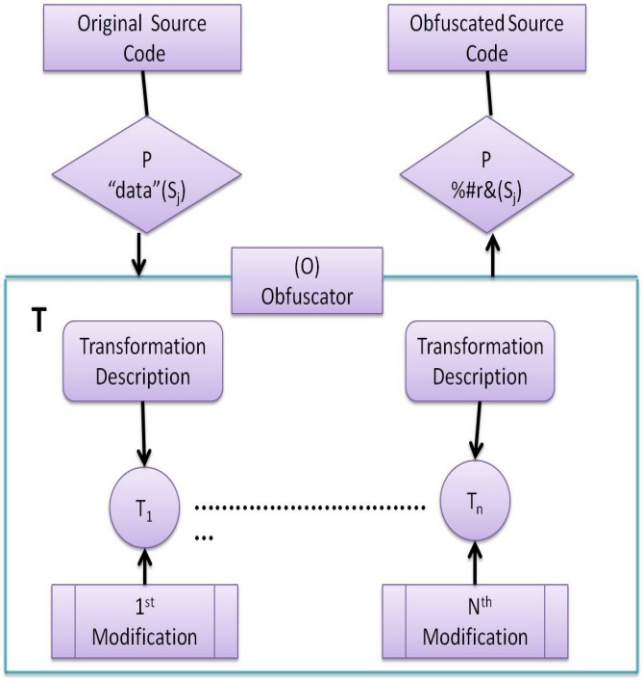


Figure 2. Block diagram of Code obfuscation

Given a program P and a set of obfuscation transformations T, generate a program P’ such that:[4]

a) P’ retains the functionality of P,
b) P’ is difficult to reverse engineer and
c) P’ performs comparably to P (reduced obfuscation cost)

An obfuscator O is an efficient, probabilistic compiler that transforms a program P into a new program P’ = O (P) such that:

CONFERENCE PAPER
II International Conference on
“Advance Computing and Creating Entrepreneurs (ACCE2013)”
On 19-20 Feb 2013
Organized by
2nd SIG-WNs, Div IV & Udaipur Chapter , CSI , IEEE Computer Society Chapter India Council ,
IEEE Student Chapter Geetanjali Institute of Technical Studies, Udaipur, Rajasthan, India

a. ***Functionality:*** The obfuscated program should have the same functionality (that is, input/output behavior) as the input program. For such programs, the functionality requirement is that the input program and the obfuscated program should compute the same program.

b. ***Efficiency:*** The obfuscated program shouldn't be much less efficient than the input program. We are willing to allow some overhead in the obfuscated program, but it shouldn't be the case that it runs in time, say, exponentially slower than the input program.

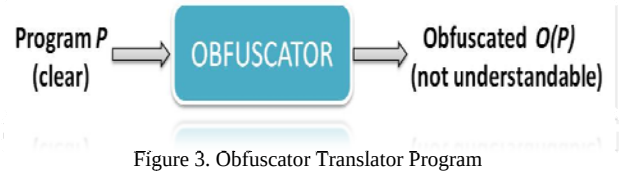


Figure 3. Obfuscator Translator Program

The obfuscator function has three propertied properties:

(a). Functionality preserving

(b). Increase of code size, time & space requirements are restricted (usually by constant factor)

(c). Obfuscated program is not understandable.

An obfuscator use transformations (T) for obfuscating the original source code. Obfuscation transformations need to be resilient. After applying transformation T_i to program segment $\mathbf{S_j}$ and generating an obfuscated statement $\mathbf{S_j}'$, it must be prohibitively hard to build an automated tool that can be generate $\mathbf{S_j}$ $\mathbf{S_j}'$.

III. ASSESSING QUALITY PARAMETERS

The quality of an obfuscation method is determined by the combination of its potency, resilience, stealth and cost.

a. ***Potency****:* Potency defines to what degree the transformed code is more obscure than the original. Software complexity metrics define various complexity measures for software, such as number of predicates it contains, depth of its inheritance tree, nesting levels, etc. While the goal of good software design is to minimize complexity based on these parameters, the goal of obfuscation is to maximize it.[4]

b. ***Resilience****:* Resilience defines how well the transformed code can resist automated deobfuscation attacks. It is a combination of the programmer effort to create a deobfuscator and the time and space required by the deobfuscator. The highest degree of resilience is a *one-way* transformation that cannot be undone by a deobfuscator. An example is when the obfuscation removes information such as source code formatting.

c. ***Stealth****:* Stealth defines how well the obfuscated code blends with the rest of the program. If the transformation introduces code that stands out from the rest of the program, it may be difficult for a deobfuscator to spot, but it can easily be spotted by a reverse engineer. Stealth is context-sensitive; what is stealthy in one program may not be in another.

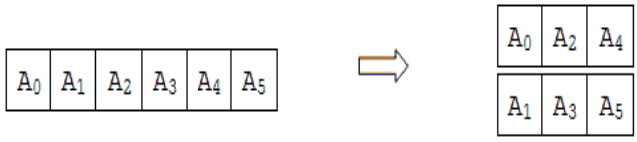
d. ***Cost****:* Cost is the execution time and space overhead in the obfuscated code compared to the original code. A transformation with no cost associated is free. Cost is also context-sensitive.[4]

IV. CODE OBFUSCATION TECHNIQUES

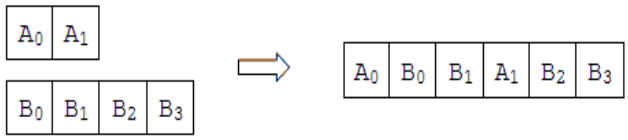
Obfuscation methods are classified depending on the information they target. Some simple transformations target the lexical structure of the program while others target the data structures or the control flow.

Let's look at the different ways to make arrays difficult to read and make sense of. The objective is to confuse a human being or a deobfuscator from understanding the true nature and purpose of the array.

First, an array could be **split** into multiple sub-arrays to confuse the reader about its structure. For example, consider an array which holds a list of names: if we split it into two arrays, it will be interpreted as two different lists of names.[4][5]



A related idea is to **merge** multiple arrays into one, like this:



We could even **fold** an array, i.e. increase the number of dimension.

Or **flatten** an array, i.e. reduce the number of dimensions.

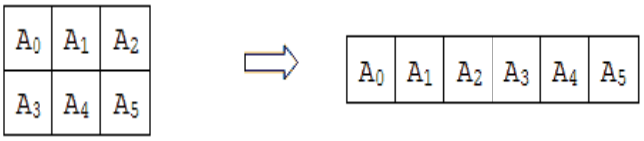


Figure 4. Array Structure Using Merging Technique

Each of the above makes the array more difficult to interpret. Observe that array splitting and folding increase the complexity of the code. They thus confuse the reverse engineer, i.e. increase the potency of the transform. On the other hand, array merging and flattening decrease the complexity. But they increase the obscurity of a program because they introduce bogus structure or remove structure from the original program.

Another method for obfuscating arrays is to reorder elements within the array. A mapping function *f* can be

CONFERENCE PAPER
II International Conference on
"Advance Computing and Creating Entrepreneurs (ACCE2013)"
On 19-20 Feb 2013
Organized by
2nd SIG-WNs, Div IV & Udaipur Chapter , CSI , IEEE Computer Society Chapter India Council ,
IEEE Student Chapter Geetanjali Institute of Technical Studies, Udaipur, Rajasthan, India

defined such that the i^{th} element in the original array is relocated to the j^{th} position where $j = f(i)$. This function can be used for the mapping of elements in the original and reordered array. Reverse engineers can figure out what's going on here, so this has low potency and resilience; but these are also free since there is no change in memory or execution time associated with shuffling the positions of array elements.[4][5]

We have looked at different methods to make code so complex with reverse engineering. Let's quickly recap how effective each of these techniques is in terms of Potency, Resilience and Cost.

Table I Various Transform Methods

Transform	Potency	Resilience	Cost
Split Array	Varies	Weak	Free
Merge Array	Varies	Weak	Free
Flatten Array	Varies	Weak	Free
Fold Array	Varies	Weak	Free
Reorder Array	Low	Weak	Cheap

These parameters were potency — the ability to confuse a human reverse engineer, resilience — the ability to fool a deobfuscator, and cost — the increase in execution time and memory required in the obfuscated code. Data obfuscation does not make your programs "fool-proof" against reverse engineering; but it adds a second level of defense.

Obfuscation methods are further classified based on the kind of operation they perform on the targeted information. Some methods manipulate the aggregation of control or data, while others affect the ordering. Code obfuscation can be achieved through one or more of the following methods, the different obfuscation methods are:

A. Binary Structure Obfuscation - A source code obfuscator accepts a program source file, and generates another functionally equivalent source file, which is much harder to understand or reverse-engineer. This is useful for technical protection of intellectual property when source code must be delivered for public execution purposes.

B. Data Obfuscation - This is aimed at obscuring data and data structures. Techniques used in this method range from splitting variables, promoting scalars to objects, converting static data to procedure, change the encoding, changing the variable lifetime etc. It includes the following transformations:

a. Storage and Encoding Transformations – Obfuscating storage transformations attempt to choose unnatural storage classes for dynamic as well as static data. Similarly, encoding transformations attempt to choose unnatural encodings for common data types.

b. Splitting and Folding Variables– Boolean variables and other variables of restricted range can be split into two or more variables. To allow a variable V of type T to be split into two variables p and q of type U require us to provide three pieces of information [12]:

(a). A function f (p, q) that maps the value of p and q into the corresponding value of V,

(b). A function g (V) that maps the value of V into the corresponding values of p and q, and

(c). New operations (corresponding to the primitive operations on value of type T) cast in terms of operations on p and q of type U.

c. Promote Variables – There are a number of simple transformations that promote variables from a specialized storage class to a more general class. Their potency and resilience are generally low, but used in conjunction with other transformations that can be quite effective. For example, in java, an integer variable can be promoted to an integer object. The same is true for the other scalar types which all have corresponding "packaged" classes. Since java supports garbage collection, the objects will be automatically removed when they are no longer referenced.

d. Change Encoding – The easiest method to remove ASCII strings from a binary is to encrypt or encode them. A simple character by character XOR can obfuscate the strings to make them unreadable.

e. Ordering Transformations – The ordering transformations can also play a important role in data obfuscation, it is used to randomizing the order in which computations are performed. Randomize the order of methods and instance variables within classes and formal parameters within methods. The potency of these transformations is low and the resilience is one-way.

C. Control Flow Obfuscation - This aims at changing the control hierarchy with logic preservation. Here false conditional statements and other misleading constructs are introduced to confuse decompilers, but the logic of the code remains intact. Control flow obfuscation introduces false conditional statements and other misleading constructs in order to confuse and break decompilers. Instead of adding code constructs, Dotfuscator works by destroying the code patterns that decompilers use to recreate source code. [7] The end result is code that is semantically equivalent to the original but contains no clues as to how the code was originally written. Even if highly advanced decompilers come to pass, their output will be guesswork.

Control flow obfuscating transformations are applied, such as opaque predicates, insert dead or irrelevant code, extend loop conditions, add redundant operands, parallelize code, remove library calls and programming idioms, table interpretation, loop transformations, clone methods etc. in which some of are on the base of aggregation, computation and ordering transformation.

A predicate is opaque if its value is known a priori to the obfuscation, but this value is difficult for the deobfuscator to deduce. Given such opaque predicates, it is possible to

construct transformations that break up the flow-of-control of the given program by inserting dead or buggy code in branching guided by opaque predicates. Figure 5. Illustrate an example of how the insertion of an opaque predicate PT can be used in order to confuse the control flow of a given program P.

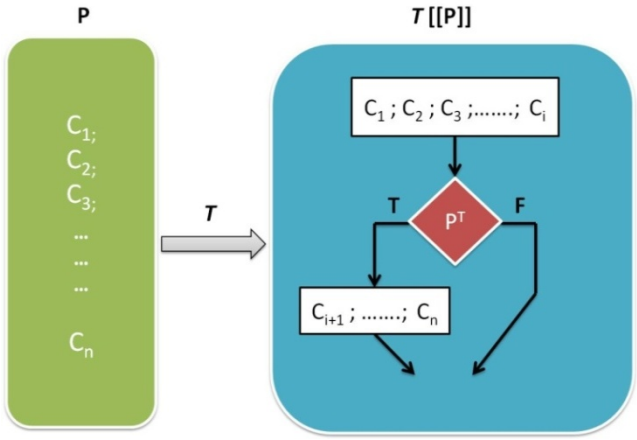


Figure 5. Control Flow Obfuscation

a. ***Aggregation obfuscation:*** Alters how statements are grouped together. An example is inlining, which means replacing a function call by the body of the function.

b. ***Ordering obfuscation:*** Alters the order in which statements are executed. An example is reversing a loop so that it iterates backwards or change the structure of the arrays.

c. ***Computation obfuscation:*** Alters the control flow in a program by hiding the true control-flow behind irrelevant statements that do not contribute to the actual computations, for example, by inserting object level code that has no source code equivalent, or by inserting new redundant code or code that will never be executed (dead code and null operations). [9]

D. Preventive Obfuscation– Preventive transformations are quite different from control and data transformations. The main goal of this method is not to obscure the program code but to make it more difficult to break for the deobfuscators. Instead of this design code to make known automatic deobfuscation techniques are difficult (inherent preventive transformations), and to explore known problem in current deobfuscators or decompilers (targeted preventive transformations) [12]

a. **Targeted preventive transformations:** It is focus on protection against decompilers and reverse engineering methods. Renaming metadata to gibberish or less obvious identifiers is one such technique and it tries to make automatic deobfuscation techniques more difficult. In a targeted preventive transformation, consider the Hose Mocha program. It was designed specifically to explore a weakness in the Mocha decompiler. Hose Mocha inserts extra instructions after every return statement in every method in the source program. This transformation has no effect on the behavior of the application, but it is enough to make Mocha crash.

b. ***Inherent preventive transformations:*** Tries to exploit known weaknesses in deobfuscator. It has low potency, high resilience and an ability to boost the resilience of other transformations. For example, assume that a for-loop has been reordered to run backwards, naturally, there is nothing stopping a deobfuscator from performing the same analysis and then returning the loop to forward execution. To prevent this, add a bogus data dependency to the reversed loop. The resilience this inherent preventive transformation adds to the loop reordering transformation depends on the complexity of the bogus dependency.

There are many commercial tools and some open source tools available in the market for achieving code obfuscation. For example, Oracle provides a way for shipping PL/SQL code, using the wrap utility that ships with the database. It will encrypt the source code into a format that cannot be reverse-engineered or edited. Code obfuscation introduces greater overhead. Unless the transform is optimized, obfuscated code runs slower in general than normal source code and wrapped package can be larger in size too. These however may be the price to be paid for enhanced protection of the source code.

V. CONCLUSION

Obfuscated code is source code or intermediate language that is very hard to read and understand, often intentionally. However, we need to compromise the software engineering principles. The code obfuscation can also use to increase the level of software secrecy with the integration of other technology. Programs known as obfuscators operate on source code, object code, or both mainly for the purpose of deterring reverse engineering, disassembly, or decompilation. Obfuscated code to prevent reverse engineering is typically done to manage risks that stem from unauthorized access to source code.

VI. REFERENCES

[1] L. Ertual, S.Venkatesh, “Novel on obfuscation algorithmfor software security”, International Conference on Software Engineering Research and Practice,2005,pp.209-215

[2] Taha H.A., Optimizing Techniques, II edition.

[3] Stephen Drape,”An obfuscation for Binary Trees”, IEEE, 2006.

[4] Ramakrishnan Srikant “Protecting Software Secrets using reverse engineering”.

[5] P.Nixon,”Securing Java based mobile Agents through Byte Code Obfuscation”, IEEE, 2006, pp 305-308.

[6] Benjamin Lynn,”Positive results and Techniques for Obfuscation” IEEE 2004, pp 123-127.

[7] Harn L. (1993). Self Protecting mobile agents, Proceeding of Workshop, Chung Cheng Institute of technology, ROC, p.p. 61 – 73.

[8] Syed Wahar Shah, Group oriented,"Static Analysis on Reverse Engineering", IEEE, Proc – computer digit tech - 141(5), p.p. 307-313.

[9] Hellman M. E. (1979). The method of software security, Scientific American -241, p.p. 130-139.

[10] D.Low, "A Taxonomy of Obfuscating transformations", American Mathematical Monthly – 36,p.p.15-30.

[11] Hwang M., Lin I. and Lie E.J. (2000). "A secure nonrepudiable tool for software protection research Scheme".with Known Signers, International Journal of Informatica - 11(2), p.p.1- 8.

[12] Chen C.C. (2001). Copyright Protection of J2EE web applications, IEEE Taipei, p.p. 26 – 28.