

A Developer's Guide to Enhance Software Security

Mahin Talukder

Abstract

To enhance software security a number of approaches have been introduced by computer security researchers. Code Obfuscation is one of the approaches widely used by developers to protect software code. Code Obfuscation increases the complexity of the code which makes the code harder to understand and cannot be processed without interpretation. Program slicing works by finding the parts of a program relevant to the value of a chosen set of variables at some chosen point in a program. A slice is constructed by deleting the parts of the program that are irrelevant to those values. Code obfuscation and program slicing are two different areas of research. Code obfuscation and program slicing provide different techniques for software engineering. These engineering approaches are welcomed in various business organization to protect intellectual property embedded in software algorithms. In this paper the author discusses the current state of art in the areas of program obfuscation and program slicing. In addition, a number of approaches are discussed in respect of software security that should be considered. It is argued that program slicing in obfuscated code would provide a better understanding and approach to software security in commercial business.

Keywords: *computer security, code obfuscation, computer programming, program slicing, software developments, software security*

Introduction

We are living in a technological era. As the technologies evolve, requirements from the users also increase. There are thousands of software programmes published commercially and non- commercially to maintain business standards, as well as making life easier for day-to-day living. Today, it is quite impossible to run a business without technologies and information. Software is used to create, modify and store information and to access that information when needed. Leakages of information and unauthorised access could result in business failure, as well as creating bad publicity for the organization. On the other hand, business growth leads to new developments within systems that requires additional man power for the business. Software developers and researchers have considered the consequences associated with inadequate security provision, and, on the basis of findings, various branches in the IT security industry have been developed: for example, software security, software reuse, software re-engineering and software reverse engineering. As a consequence of rising demand, universities, governments and business customers are developing human expertise within these areas.

Software security is an idea implemented to protect software against malicious attack and other hacker risks, so that the software continues to function correctly under such potential risks. Security is necessary to provide integrity, authentication and availability. The common adversary is an untrusted entry that attacks the system from outside. Security systems can be compromised in many ways. A successful attack can take control over the system, attack or modify software in way that the software vendor does not intend, or breach confidentiality rights by extracting intellectual property from the software, such as algorithms, designs, or implementations and making illegal copies of the software. There are number of approaches to prevent attack: e.g., Obfuscation, Tamper Proofing and Watermarking.

On the other hand, in most engineering disciplines, systems are designed by using existing components that have been used in other systems. Software engineering has been more focused on original development, but it is now recognised that to achieve better software, more quickly and at lower cost, it is necessary to adopt a design process that is based on systematic software, given the fact that size and complexity of program has become harder to maintain and test. Program slicing, introduced by Mark Weiser, is currently seen as a rapid development on software re-use and debugging. With the use of program slicing we can determine which pieces of code are responsible for a specific change or flow in the program statement and limit program functionality to what is needed.

Aim and objectives

Determining the aim and setting objectives in a project is crucial for success and provides the structure required to complete the project. The aim and objectives of this project are as follows:

The aim of this project is to determine the security strength of an obfuscated code using program slicing.

The main objective of this research is to identify an appropriate set of obfuscators based on program functionality that can be used to create software which is difficult to reverse engineer.

To achieve this objective, the following research questions will be addressed.

1. What is the current state of art in code obfuscation?
2. How to determine which obfuscation is better and why?
3. How effective is program slicing in de-obfuscating codes?
4. How to improve obfuscation to improve software security?

Literature

Obfuscation in computer engineering is a term used to protect the reverse engineering process. An obfuscated program preserves the same functionality of the original program and uses minimum overhead. Obfuscated codes are harder to analyse or comprehend, because

they create multiple versions of the same program. The goal of software protection through code obfuscation is to transform the source code of an application to the point that it becomes “harder to understand” — which can mean it becomes unintelligible to automate program comprehension tools and the results of program analyses become less useful to a human adversary (Davis, 1998). Despite various definitions of obfuscation, there are number of classifications that determine the properties of obfuscated codes. Lexical Transformation, Control flow transformation, Data transformation, Anti- disassembly, anti-debugging and encryption are the types of classification of obfuscation that are mostly used. These transformations are adopted according to the requirements of software protection.

Layout Obfuscation

The three main categories identified by Collberg, namely, layout, control-flow and data obfuscation give a brief overview of some of the obfuscation techniques that have developed and are applicable to a wide variety of programming languages. (Collberg, Thomborson and Low, 1997) For example, layout obfuscation is responsible for the look of

a program rather than semantics changes. Applying layout obfuscation code commands can lead to bogus commands that have no meaning to others. Another layout obfuscation can change one formatting of a code to a different one. These can also include whitespaces and program indentation (*The International Obfuscated C Code Contest*, 1997). Popular obfuscation, for example, changes the name of variables, classes and methods to a different naming and making it complicated to reverser engineer. In general, layout obfuscations attempt to change the syntax of the program rather than the semantics of the program.

Control-flow Obfuscation

Control-flow obfuscation is responsible for semantic transformation. The set of semantic transformations are targeted at the control-flow; so these transformations try to change conditional statements, jumps and loops. Control-flow in a program can be changed by adding bogus conditional jumps or changing loops, etcetera. According to Majumdar et al (*The International Obfuscated C Code Contest*, 1997) two techniques highlighted are Opaque Predicates and Control-Flow flattening.

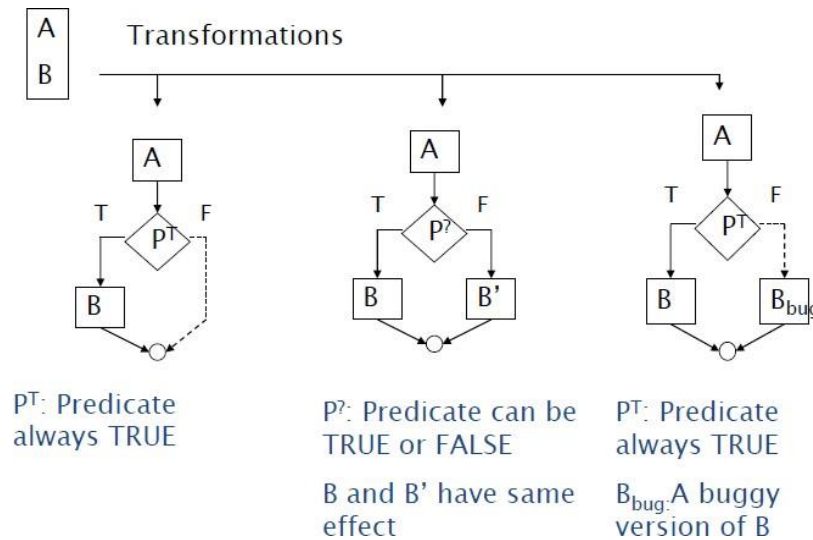


Figure 1 Control-flow Obfuscation (The International Obfuscated C Code Contest, 1997)

An opaque predicate is an obfuscated condition that, followed with a conditional operation, will make the analysis harder and in some cases impossible until the code is executed and that specific condition is evaluated. Opaque predicates set the condition either true or false for a variety of reasons until the outcome of the program is evaluated. The control flow flattening obfuscation completely obscures the links between basic blocks. Wang describes the process of chenxification, which puts the basic blocks of a program into a large switch-statement (called a dispatcher) that decides where to jump next, based on an opaque variable (Wang, Gross and Taylor, 2001).

Control flow flattening using a central dispatcher has also been described by Chow and similarly by Linn and Udupal. (Chow *et al.*, 2001); Udupa, Debray and Madou, 2005) The procedure uses a so-called branch function to obfuscate the targets of CALL instructions. All calls are forced to pass through the branch function, which directs the control flow to the actual target based on a call table. Wang has described a technique for flattening control-flow. The idea of flattening control-flow transformation is to remove control-flow constructs, that is to say loops, so that all basic blocks have the same predecessor and successor in the control-flow graph. (Wang, Gross and Taylor, 2001)

<pre> init; while (cond) { loop_body; } </pre>		<pre> var = 1; switch (var) { case 1 : init; var = 2; break; case 2 : if (cond) var = 3; else var = 4; break; case 3 : loop_body; var = 2; break; case 4 : var = 1; end; } </pre>
--	--	---

Figure 2 flattening control-flow transformation (Wang, Gross and Taylor, 2001)

Figure 2 flattening control-flow transformation is an example of flattening control-flow transformation where a simple while loop is converted into a switch statement and variable var acts as a dispatcher and effectively controls the execution of the block of codes.

Data obfuscation

Data obfuscation in code obfuscation technique is adopted to modify data in a form where stored data in a program can be hidden from direct analysis. Data obfuscation also requires the program code to be modified, so that original data representation can be recognized during the runtime of a program or in execution time. Collberg and Nagra describe many data obfuscation techniques (Collberg and Nagra, 2010). Data obfuscation converts a variable or constant into multiple

computational results of one or several variables and constants. As a result, it makes it harder for an analyst to identify variables and constants. Two separate functions create a mapping between actual variables and the split representation. One of the functions is executed in obfuscation time and another one is reconstructed with the original value of the obfuscated variables and constants.

String splitting and keyword substitution are the data obfuscations that are mostly applied to string objects. String splitting converts a single string into multiple substrings and assigns a random variable for those substrings. Keyword substitution is normally used to assign a specific variable to a different variable name (Schrittwieser *et al.*, 2015).

<pre>document.write("Hello world");</pre> (a) <pre>var mystring=document; mystring.write ("Hello world");</pre> (c)	<pre>var xs = "ite("; var lw = "lo w"; var ao = "doc"; var tc = "nt.wr"; var zg = "orl"; var mv = "\"Hel"; var m = "d\""; var dg = "ume"; eval(ao + dg + tc+ xs + mv + lw + zg + m);</pre> (b)
---	--

Figure 3 Data Obfuscation (Schrittwieser et al., 2015)

Figure 3 (a) represents a standard string. The string represented as (b) when string splitting obfuscation is applied and thirdly (c) is obtained from (a) when keyword substitution obfuscation is applied. Along with these data a vast area of obfuscation exists in specifying data, variable encoding, array transformation and exception handling.

There are number of obfuscators introduced by researchers that are used for commercial and non-commercial purposes. The obfuscation takes place either in binary-to- binary, source-to-binary or source-to-source arrangements. As the project's main aim is to determine the most effective and complex code generated by various obfuscators, source-to-source code arrangements are discussed. Companies like Semantic Designs which provides automated tools for software engineering supports obfuscation for number of programming languages. Many conventional

obfuscation tools use ad hoc string processing methods to carry out the obfuscation process. String processing can sometimes work, but it often fails when there are multiple statements per line or nested comments are encountered. This is because programming languages have complex formation and name resolution rules; therefore, processing them reliably usually requires a complete language transformations. Failure to process the program correctly could produce a weakened version of obfuscation.

Semantic Designs

Semantic Designs sort the source language lexical and syntax rules into compiler-like data structures, carries out the obfuscation, and then unparses back to source text. This approach ensures that the syntax structures found match those of the language. Thus, Semantic Designs use built-in extensions for

language that are dependent on source code transformation.

Semantic Designs obfuscators are presently available on Windows 2003, XP, or later operating systems and available obfuscators are Ada, ECMAScript (JavaScript), including script embedded in HTML, XML, ASP or PHP __, C in many dialects, C++ in many dialects, C# v5 and v6, Java v7 and v8, PHP version 5 __, PLSQL, SystemC, SystemVerilog, Visual Basic 6 (Beta), VBScript, including Active Server Pages (Beta), Verilog 2012, VHDL 2002 and 2008 (*Semantic Designs: Source Code Obfuscators*, 2018)

Irdeto

Irdeto is a technological company that provides support for software developments, software engineering and software security. Their services include security on banking systems, cloud store security and automobile security. Irdeto Cloakware for Automobiles, for example, is a multi-layered, multi-faceted product that resolves issues relating to obfuscation, integrity verification, anti- debugging, white-box cryptography, diversification and renewability. It offers a full suite of patented technologies and cybercrime services that not only protects the vehicle environment, but also protects itself from being reverse engineered and modified by hackers(*Semantic Designs: Source Code Obfuscators*, 2018).

Stunnix

Stunnix provides solutions for source code obfuscation for commercial uses. Stunnix uses C/C++ Obfuscator, Perl-Obfus, JavaScript Obfuscator, and VBScript Obfuscator as tools to make source

codes unreadable and prevent illegal reuse, IP theft and tampering f source codes (*Semantic Designs: Source Code Obfuscators*, 2018).

Tigress

Tigress is a diversifying virtualiser/obfuscator for the C language that supports many novel defences against both static and dynamic reverse engineering and de-virtualisation attacks. In particular, Tigress protects against static de-virtualisation by generating virtual instruction sets of arbitrary complexity and diversity, by producing interpreters with multiple types of instruction dispatches and by inserting codes for anti-alias analysis. Tigress protects against dynamic de-virtualisation by merging the real code with bogus functions, by inserting implicit flow, and by creating slowly- executing re-entrant interpreters. It implements its own version of code packing through the use of runtime code generation. Finally, Tigress' dynamic transformation provides a generalised form of continuous runtime code modification (Tigress, 2017). Tigress source code-to-source code transformation builds on CIL (C Intermediate Language) and MyJIT which is a small library that allows compilation and execution of intermediate codes into machine code. CIF is a high-level representation which, along with JIT, permits easy analysis and source code –to- source

Code transformation. Tigress supports all of the c99 language, including gcc extensions and source-to-source transformations that facilitate the examination of transformed codes.

Tigress takes C files, seeds as per user provided, and several transformations at the same time. Multiple C files are combined into one C file, and transformations are accommodated within the function of C file. There are three major transformations supported by Tigress. They are –

- Virtualization that transforms a function into an interpreter.
- Jitting that transforms a function into one that generates its machine code at runtime.
- JitDynamic that transforms a function into one that continuously modifies its machine code at runtime.

In addition to these three major transformations, Tigress has Control Flow Flattening, Function Splitting, Merging, Argument Randomization, Control Flow Splitting with Opaque Predicates, encoding of Literals, Data, and Arithmetic Transformation (*Tigress Transformations*, 2017).

Tigress was originally designed as a remote device for testing the veracity of backend systems. Now Tigress is an open-source software that is consistently in use in respect of finding the strength of obfuscated code, measuring software obfuscation reliance against automated attacks and testing code analysis for cytometric algorithms.

Program Slicing

Program Slicing is a decomposition technique that extracts parts of a programme by applying special computations. The first and most commonly accepted definition was provided by Mark Weiser in 1979. According to Weise, A slice S of a program P on a slicing criterion $C = (i, V)$ is any executable program with following two properties.

1. S can be obtained from P by deleting zero or more statements from P .
2. Whenever P halts on input I from I with state trajectory T' , then S also halts on input I with state trajectory T , and $PROJ_c(T) = PROJ_c(T')$, where $PROJ_c$ is the projection function associated with criterion C . He claimed a slice to be the mental abstraction that people make when they are debugging a program. A slice consists of all the statements of a program that may affect the values of some variables in a set 'V' at some point of interest 'p'. The pair (p, V) is referred to as a slicing criterion (Katti and Terdal, 2010).

Variants of program slicing:

A slice of a program is computed either by using static information known as a static slice or by using dynamic information known as dynamic slice. Static slicing seeks to find the executable subset of a program's statement that exhibits the same behaviour of a specified variable at a specific location as the original

program for all possible inputs. In contrast, dynamic slicing preserves the behaviour of slicing criterion only with respect to specific input (Katti and Terdal, 2010). Conditioned slicing is a technique to identify statements and predicates which contribute to the computation of a selected set of variables when some chosen condition is satisfied (Katti and Terdal, 2010). The technique has previously been used in program comprehension and re-engineering (Katti and Terdal, 2010). Observation-based slicing has been recently introduced by researchers as an alternative to dependent-based program slicing which has been derived from dynamic slicing. Observation-based slicing uses a delete-execute-observe approach wherein the observer can determine if the trajectory of the slicing criterion has been changed. (Binkley, et al., 2014).

Program Analysis

Programme analysis is the process of automatically analysing the behaviour of a program. Program analysis offers techniques to predict the codes of a programme and observes behaviour dynamically during the run-time. Researchers have proposed different frameworks to describe data flow analysis within a program.

Control flow analysis

The aim of control flow analysis is to discover the hierarchical flow of control within a programme. Some special statements of a programme are known as control flow statements. They are used conditionally to execute specific statements. For example, 'if' statements are used to execute specific statements, and 'while' statements are used to execute one or more statements and the other changes interrupting normal sequential flow through structured jumps like 'return, break and continue'. The process is illustrated in Figure 4, overleaf.

```

1  read(n);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while (i <= n) {
6      sum := sum + i;
7      product := product * i;
8      i := i + 1;
9  }
9  print(sum);
10 print(product);

```

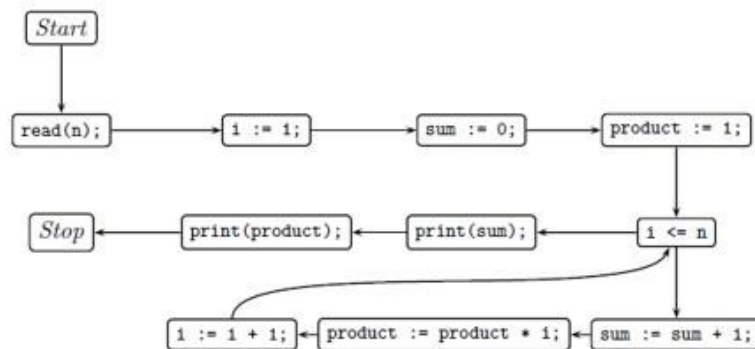


Figure 4 Control flow graph (Kim and Joao, 2006)

The most common representation of control flow analysis of a program is a control flow graph. Nodes of a control flow graph represent statements, and the control flow makes

Data flow analysis:

Data flow analysis provides global information on data manipulation within a program. Dataflow analysis is usually performed on the program's control-flow graph (CFG). The goal is to associate with each program component (each node of the CFG) information that is guaranteed to hold at that point on all executions. Thus, a dataflow problem includes

- A CFG,
- A domain D of "dataflow facts",

predictions. Thus, program control flow is examined to measure the number of independent paths in the source code.

- A dataflow fact "init" (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- An operator $[]$ (used to combine incoming information from multiple predecessors),
- For each CFG node n , a dataflow function $f_n : D \rightarrow D$ (that defines the effect of the execution (Späth, Ali and Bodden, 2017)).

There are two types of data flow problem constant propagation: (1) forward problems

and (2) live-variable analysis known as backward problems. In constant propagation information at a node n summarises what can happen on paths from "enter" to n . On the other hand, in live-variable analysis information at a node " n " summarizes what can happen on paths from " n " to "exit". This aims to propagate any constant assigned to a variable through the flow graph and to substitute it as the use of a related variable for optimisation purposes. A value of a variable is considered active or inactive based on the uses of that variable in the near future.

ORBS as Slicing algorithms

Observation Based Slicing was introduced by (Binkley *et al.*, 2014) with a view to slicing heterogeneous programs consisting of components written in different programming languages and performance slicing that includes binary components or libraries. Although ORBS has been derived from dynamic slicing, some critical differences exist. Observation-based slices are based on observed dependencies rather than statically determined. Observation-based slices actually delete statements in a file of interest and execute the program to observe if the

projected trajectory has changed. For the slices taken with respect to a given slicing criterion, ORBS must capture the trajectory for the slicing criterion, delete statements from the components of interest and may leave intact other components, including binary components and other source files.

From the definition of ORBS, it is stated that "An observation-based slice S of a program P on a slicing criterion $C = (v; l; I)$ composed of variable, line l , and set of inputs I , is any executable program with the following properties:

1. The execution of P for every input I in I halts and produces a sequence of values $V(P; I; v; I)$ for variable v at line l .
2. S can be obtained from P by deleting zero or more statements from P .
3. The execution of S for every input I in I halts and produces a sequence of values $V(S; I; v; I)$ for variable v at line l .
4. $\forall I \in IV(P; I; v; I) = V(S; I; v; I).$ "

The algorithm is illustrated in figure 5, overleaf.

Algorithm 1: ORBS

ORBSlice($P, v, l, \bar{I}, \delta$)

Input: Source program, $P = \{p_1, \dots, p_n\}$, slicing criterion, $(v, l, \bar{I})$, and maximum deletion window size, δ

Output: A slice, S , of P for $(v, l, \bar{I})$

- (1) $O \leftarrow \text{SETUP}(P, v, l)$
- (2) $V \leftarrow \text{EXECUTE}(\text{BUILD}(O), \bar{I})$
- (3) $S \leftarrow \text{REVERSE}(O)$
- (4) **repeat**
- (5) $deleted \leftarrow \text{False}$
- (6) $i \leftarrow 1$
- (7) **while** $i \leq \text{length}(S)$
- (8) $builds \leftarrow \text{False}$
- (9) **for** $j = 1$ to δ
- (10) $S' \leftarrow S - \{s_i, \dots, s_{\min(\text{length}(S), i+j-1)}\}$
- (11) $B' \leftarrow \text{BUILD}(\text{REVERSE}(S'))$
- (12) **if** B' built successfully
- (13) $builds \leftarrow \text{True}$
- (14) **break**
- (15) **if** $builds$
- (16) $V' \leftarrow \text{EXECUTE}(B', \bar{I})$
- (17) **if** $V = V'$
- (18) $S \leftarrow S'$
- (19) $deleted \leftarrow \text{True}$
- (20) **else**
- (21) $i \leftarrow i + 1$
- (22) **until** $\neg deleted$
- (23) **return** $\text{REVERSE}(S)$

Figure 5 ORBS Algorithms (Binkley et al., 2014)

ORBS works by iteratively deleting sequences of lines until no more lines can be deleted. In each iteration, ORBS deletes from the source code and validates the deletion by compiling, executing and comparing the trajectory with the original program trajectory. If the original program trajectory is the same as the deleted program trajectory then the deletion is accepted. If the program trajectory does not match, then the deletion is discarded.

Methodology

Software engineering and software development process are continuously engaged in the process of designing, developing, testing and debugging. Development processes change according to user requirements. With the changes of time, new software security options are adopted. Although future developments cannot be predicted in advance, it is wise to take up-to-date security measures before implementing software. In today's world, many methods,

tools, equipment and approaches are available for software development, testing, debugging and security. Davida considered the following activities associated with software development to be particularly important.

- Critically analyse the current system
- Analyse different modelling techniques
- Specify system requirements including stability, scalability and security
- Design the system
- Implement and test the system

(Davida *et al.*, 2001)

The researcher intends to use number of approaches to complete the project. Primary research will focus upon identifying the nature of obfuscations that are used within software development and engineering. This includes small, medium and large business organisations, independent of programming language vendors. Primary research will also be conducted on the types of obfuscation that are commonly used in software development for security purposes. Obfuscated source code can be found on various online source code repository sites. The weight of obfuscation will be determined either by adopting techniques provided by program obfuscators or by obfuscating source codes of the author's choice from open source programming. Tigress is a C program obfuscator that provides techniques for obfuscation as well as performing obfuscation functions for non-commercial usages. Various program matrices can be used to determine how complex the program needs to be to reverse engineer. In addition, primary research will be conducted by inserting slicing criteria into the program to determine optimal

solutions and find the effectiveness of software reuse. To achieve the goal in program slicing, existing algorithms and approaches, evolutionary algorithms such as. genetic programming will provide the foundations of secondary research for this project.

Several software development approaches have been provided by computer science researchers. Hard and soft approaches, a unified approach, object-oriented development, a waterfall approach, the spiral model are most common development approaches in software engineering. Each of these development processes is suitable for particular types of problems (Lyu and Kanoun, 1996). For this project, an iterative and incremental development approach provides a basic structure about software development. After adding every component to the system, requirements are revised from the beginning. Knowledge and experience gathered in each cycle is carried out to the next level of iteration. Program obfuscation and program slicing will be evaluated in every iteration of the research cycle.

Conclusion

Program obfuscation is a good technique for implementing security measures in software applications. Obfuscated codes within a program make them harder to understand by humans. With the development of software, security hackers have started to break these codes for their benefits. Hackers have invented various strategies to analyse obfuscated codes and to identify program functionalities.

Applying a number of obfuscation codes would make programs harder to analyse. Program slicing would determine which piece of code is responsible for a specific functionality. A prime objective is to identify how effective program slicing is to obfuscated codes.

References

- Binkley, D. et al. (2014) 'ORBS: language-independent program slicing', in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, pp. 109–120. doi: 10.1145/2635868.2635893.
- Chow, S. et al. (2001) 'An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs', in: Springer, Berlin, Heidelberg, pp. 144–155. doi: 10.1007/3-540-45439-X_10.
- Collberg, C. and Nagra, J. (2010) *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Addison-Wesley.
- Collberg, C., Thomborson, C. and Low, D. (1997) *A Taxonomy of Obfuscating Transformations*. Available at: <https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf> (Accessed: 17 September 2018).
- Davida, G. et al. (2001) *Information security: 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001: proceedings, Proceedings of the 4th International Conference on Information Security*. Springer. Available at: <https://dl.acm.org/citation.cfm?id=744507> (Accessed: 17 September 2018).
- Davis, J. C. (1998) 'Protecting intellectual property in cyberspace', *IEEE Technology and Society Magazine*, 17(2), pp. 12–25. doi: 10.1109/44.682891.
- Katti, A. and Terdal, S. (2010) *Program Slicing for Refactoring: Static Slicer using Dynamic Analyser, International Journal of Computer Applications*. Available at: <https://www.ijcaonline.org/volume9/number6/pxc3871869.pdf> (Accessed: 17 September 2018).
- Kim, H. and Joao, J. A. (2006) 'Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths'. doi: 10.1109/MICRO.2006.20.
- Lyu, M. R. and Kanoun, K. (1996) *Handbook of software reliability engineering, Handbook of software reliability engineering*. IEEE Computer Society Press. Available at: <https://dl.acm.org/citation.cfm?id=239433> (Accessed: 19 September 2018).
- Schrittwieser, S. et al. (2015) '4 Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?', *ACM Comput. Surv.*, 49(4). doi: 10.1145/0000000.0000000.
- Semantic Designs: Source Code Obfuscators* (2018) *Semantic Designs*. Available at: <http://www.semdesigns.com/Products/Obfuscators/index.html?Home=Main> (Accessed: 21 February 2018).
- Späth, J., Ali, K. and Bodden, E. (2017) 'IDEal: Efficient and Precise Alias-Aware Dataflow Analysis', *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), pp. 1–27. doi: 10.1145/3133923.
- The International Obfuscated C Code Contest* (1997). Available at: <https://www.ioccc.org/index.html#news> (Accessed: 17 September 2018).
- Tigress Transformations* (2017). Available at: <http://tigress.cs.arizona.edu/transformPage/index.html#other> (Accessed: 8 March 2018).
- Udupa, S. K., Debray, S. K. and Madou, M. (2005) 'Deobfuscation: Reverse Engineering Obfuscated Code', in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, pp. 45–54. doi: 10.1109/WCRE.2005.13.
- Wang, Y., Gross, M. L. and Taylor, J.-S. (2001) 'Use of a Combined Enzymatic Digestion/ESI Mass Spectrometry Assay To Study the Effect of TATA-Binding Protein on Photoproduct Formation in a TATA Box[†]', *Biochemistry*, 40(39), pp. 11785–11793. doi: 10.1021/bi0111552.