

**REVERSE ENGINEERING**

**Technical report submitted in partial fulfillment of the requirement**

**for the award of degree of**

**BACHELOR OF TECHNOLOGY**

**In**

**COMPUTER SCIENCE and ENGINEERING**

**By**

**U.M.K. DIKSHIT**

**(11S31A0560)**



**Under the esteemed guidance of**

**Asst. Prof Md.RAZIUDDIN**

**Department of Computer Science and Engineering**

**SLC’S INSTITUTE OF ENGINEERING AND TECHNOLOGY**

**Piglipur (V), Hayathnagar (M), RangaReddy – 501512**

**(Affiliated to Jawaharlal Nehru Technological University)**

**2011-2015**



**Survey No 17/2, Piglipur Village, Near Ramoji Film City, Hyderabad. 501 51**

**(Affiliated to Jawaharlal Nehru Technological University)**

**Website:** [**www.slc.edu.in**](http://www.slc.edu.in/) **email: enquiry@slciet.com**

**CERTIFICATE**

This is to certify that the seminar report entitled **“REVERSE ENGINEERING”.** That is being submitted by **U.M.K. DIKSHIT (11S31A0560)** a Technical report Submitted in fulfillment of the curriculum, Bachelor of Technology in Computer Science and Engineering affiliated to the JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY is a record of bonafide work carried out by him under my guidance and supervision.The results embodied in this synopsis have not been submitted to any other university or Institute for the award of any degree.

**Internal Guide Head of Department**

**Asst. Prof Md. RAZIUDDIN Prof.T.VENKATA RAMANAM.Tech(Ph.D)**

**Dept. of Computer Science Engg Dept. of Computer Science Engg**

**SLC'S Institute of Engg & Tech SLC'S Institute of Engg & Tech**

**ACKNOWLEDGEMENT**

I would like to express my gratitude to all the people behind the screen who have helped me transform an idea into a real time application.

I 1would like to express my heart-felt gratitude to my parents without whom I would not have been privileged to achieve and fulfill my dreams. A special thanks to my Secretary, **T. SATYANARAYANA GARU,** for having founded such an esteemed institution. I am also grateful to my Principal, Dr.**C.SRIRAM GARU**

I profoundly thank **Prof. T. VENKATA RAMANA**, Head of the Department of Computer Science and Engineering, who has been an excellent guide and also a great source of inspiration to my work.

The satisfaction and euphoria that accompany the successful completion of the task would be great, but incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts with success. In this context, I would like to thank all the other staff members, both teaching and non-teaching, who have extended their timely help and eased my task.

And last but not the least; I would like to express my sincere thanks to the entire “**REVERSE ENGINEERING”** for constantly having been a source of inspiration and knowledge.

**U.M.K. DIKSHIT**

**(11S31A0560)**

**ABSTRACT**

The term reverse engineering finds its origins in hardware technology and denotes the process of obtaining the specification of complex hardware systems. Now the meaning of this notion has shifted to software. As far as we know there is not (yet) a standard definition of what reverse engineering is but in we can read:

Reverse Engineering is the process of analyzing a subject system to

(i) Identify the system’s components and their inter-relationships and

(ii) Create representations of the system in another form or at a higher level of abstraction.

Reverse engineering of computer software has assumed greater importance in recent years because of the need to examine legacy code to remove the year 2000 bug. There are different types of reverse engineering based on the level of abstraction of the code to be reengineered; machine code, assembly code, source code or even CASE code. Provides for a comparative overview of the legal standing on reverse engineering at the international level. Propose challenges to the global electronic community in relation to existing and future legislation in the area of reverse engineering and protection of digital works.

Software reverse engineering is a broad term that encompasses an array of methods and tools to derive information and knowledge from existing software artifacts and leverage it into software engineering processes.

Reverse engineering is a process of examination rather than a process of change, as the core of reverse engineering is deriving, from available software artifacts, representations understandable by humans.

Software reverse engineering originated in software maintenance: the standard IEEE-12191

recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation. Since then, it has been successfully exploited to deal with numerous software engineering problems. A non-exhaustive list includes recovering architectures and design patterns, re-documenting programs and database, identifying reusable assets, building traceability between software artifacts, computing change impacts, re-modularizing existing systems, renewing user interfaces, migrating towards new architectures and platforms.

**CONTENTS**

**1. INTRODUCTION**

1.1 Reverse Engineering 1

1.2 Difference between Reverse Engineering & Forward Engineering 1

1.3 Decompilers 2

1.4 Phases of Decompiler 3

1.5 Grouping of Phases 5

**2.** **FRONT END**

2.1 Syntax Analysis 6

2.2 Semantic Analysis 6

2.3 Intermediate Code Generation 6

2.4 Control Flow Graph Generation 7

**3.** **DATA ANALYSIS** 8

3.1 Types of Optimizations 9

**4.** **CONTROL FLOW ANALYSIS**

4.1 Structuring Algorithms 11

4.2 Application Order 13

**5.** **BACK END**

5.1 Generating code for Basic Blocks 14

5.2 Generating code from Control Flow Graph 15

**6.** **CHALLENGES** 16

**7.** **CONCLUSION** 17

18

**BIBILOGRAPHY**

**1 INTRODUCTION**

**1.1 Reverse Engineering**

Reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. The purpose of reverse engineering is to understand a software system in order to facilitate enhancement, correction, documentation, redesign, or reprogramming in a different programming language.

**1.2 Difference between Reverse Engineering and Forward**

**Engineering.**

|  |  |
| --- | --- |
| **REVERSE ENGINEERING** | **FORWARD ENGINEERING** |
| Given an application, deduce tentative  requirements. | Given requirements, develop an application. |
| Less certain. An implementation can yield  different requirements, depending on the  reverse engineer’s interpretation. | More certain. The developer has requirements  and must deliver an application that implements them. |
| Adaptive. The reverse engineer must find out  what the developer actually did. | Prescriptive. Developers are told how to work. |
| Less mature. Skilled staff sparse. | More mature. Skilled staff readily available. |
| Can be performed 10 to 100 times faster than  forward engineering.(days to weeks of work). | Time consuming (months to years of work). |
| The model can be imperfect. Salvaging partial  information is still useful. | The model must be correct and complete or the  application will fail. |

**1.3 Decompilers**

Reverse engineering is done with the help of decompilers. A decompiler is a program that reads a program written in a machine language (the source language) and translates it into an equivalent program in a high level language (the target language) (see Figure 1.2). A decompiler, or reverse compiler, thus attempts to reverse the process of a compiler, which translates a high- level language program into a binary or executable program.

Basic decompiler techniques are used to decompile binary programs from a wide variety of machine languages to a diversity of high-level languages. The structure of decompilers is based on the structure of compilers; similar principles and techniques are used to perform the

analysis of programs.

Source Program Decompiler Target Program

(High Level Lang.) (Machine Lang.)

Figure 1.2 Decompilation Process

Machine code for any sizeable program will be unintelligible to any normal human. Using a program called a de-compiler, it is possible, to a limited extent, to convert machine or object code to a form resembling the original source code. Even if this is done the comments or explanatory notes will have been lost so that a decompiled program will usually be very difficult to understand.24

There are also a number of technical impediments that may restrict access to the object code itself, prior to attempting its decompilation. One is encryption of the machine code, as illustrated in the cases of *Mars UK v Teknowledge* and the recent US case of *Universal City Studios v Reimerdes* involving DVDs. The second technical difficulty is that the code may exist as the contents of a ROM chip.

**1.4 The Phases of a Decompiler**

Conceptually, a decompiler is structured in a similar way to a compiler, by a series of phases that transform the source machine program from one representation to another. The typical phases of a decompiler are shown in Figure 1.3. These phases represent the logical organization of a decompiler. No lexical analysis or scanning phase in the decompiler. This is due to the simplicity of machine languages; all tokens are represented by bytes or bits of a byte. Given a byte, it is not possible to determine whether that byte forms the start of a new token or not.

* Binary Program
* Syntax Analyzer
* Semantic Analyzer
* Intermediate Code Generator
* Data Flow Analyzer
* Control Flow Analyzer
* Code Generator
* HLL program

Binary Program

Syntax Analyzer

Semantic Analyzer

Intermediate Code

Generator

Data Flow Analyzer

Control Flow

Analyzer

Code Generator

HLL program

1.3 Phases of a Decompiler

**1.5 The Grouping of Phases**

The decompiler phases presented in Section 1.3 are normally grouped in the implementation of the decompiler. As shown in Figure 1.4, three different modules are distinguished: front-end, UDM, and back-end.

The front-end consists of those phases that are machine and machine language dependent. These phases include lexical, syntax and semantic analyses, and intermediate code and control flow graph generation. As a whole, these phases produce an intermediate, machine independent representation of the program.

The UDM is the Universal Decompiling Machine; an intermediate module that is completely machine and language independent, and that performs the core of the Decompiling analysis. Two phases are included in this module, the data flow and the control flow analyzers.

Finally, the back-end consists of those phases that are high-level or target language dependent. This module is the code generator. The grouping of phases is a mechanism used by compiler writers to generate compilers for different machines and different languages. If the back-end of the compiler is rewritten for a different machine, a new compiler for that machine is constructed by using the original front-end. In a similar way, a new front-end for another high- level language definition can be written and used with the original back-end.

Binary Program

Front End

Machine Dependent

UDM Analysis

Back End

Language Dependent

HLL program

1.4 Grouping of Phases of Decompiler

**2 FRONT END**

The front-end is a machine dependent module, which takes as input the binary source program, parses the program, and produces as output the control flow graph and intermediate language representation of the program.

**2.1 Syntax Analysis**

The syntax analyzer is the first phase of the decompiler. Its role is to group a sequence of bytes into a phrase or sentence of the language. This sequence of bytes is checked for syntactic structure, that is, that the string belongs to the language. Valid strings are represented by a parse tree, which is input into the next phase, the semantic analyzer. The syntax analyzer is also known as the parser.

The syntax of a machine language can be precisely specified by a grammar. In machine languages, only instructions or statements are specified; there are no control structures as in high-level languages. In general, a grammar provides a precise notation for specifying any language.

**2.2 Semantic Analysis**

The semantic analysis phase determines the meaning of a group of machine instructions, collects information on the individual instructions of a subroutine, and propagates this information across the instructions of the subroutine. In this way, base data types such as integers and long integers are propagated across the subroutine.

**2.3 Intermediate Code Generation**

In a decompiler, the front-end translates the machine language source code into an intermediate representation, which is suitable for analysis by the Universal Decompiling Machine. The low-level intermediate representation is implemented in quadruples, which make explicit the operands use in the instruction, as shown below. The opcode field holds the low- level intermediate opcode; the destination field holds the destination operand (i.e. an identifier), and the src1 and src2 fields hold the source operands of the instruction. Some instructions do not

use two source operands, so only the src1 field is used.

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | dest | src1 | src2 |
| add  push  xor | bx  sp  ax | Bx  cx  ax | 3 ;add bx,3  X ;push cx  X ;xor ax |

**2.4 Control Flow Graph Generation**

The control flow graph generation phase constructs a call graph of the source program, and a control flow graph of basic blocks for each subroutine of the program. These graphs are used to analyze the program in the Universal Decompiling Machine (UDM) module.

**Basic Blocks:**

A basic block is a sequence of instructions that has one entry point and one exit point. If one instruction of the basic block is executed, all other instructions are executed as well.

The set of instructions of a program can be uniquely partitioned into a set of non-overlapping basic blocks, starting from the program's entry point.

**Control Flow Graphs:**

A control flow graph is a directed graph that represents the flow of control of a program, thus, it only represents the instructions of such a program. The nodes of this graph represent basic blocks of the program, and the edges represent the flow of control between nodes.

**Graph Optimizations:**

Flow-of-control optimization is the method by which redundant jumps are eliminated. For Decompilation, we are interested in eliminating all jumps to jumps, and conditional jumps to jumps, as the target jump holds the address of the target branch, and makes use of intermediate basic blocks that can be removed from the graph. The removal of jumps to conditional jumps is not desired.

**3 Data Flow Analysis**

The low-level intermediate code generated by the front-end is an assembler type representation that makes use of registers and condition codes. This representation can be transformed into a higher-level representation that does not make use of such low-level concepts, and that regenerates the high-level concept of expression. The transformation of low-level to high-level intermediate code is done by means of program transformations, traditionally referred to as optimizations.

The types of transformations that are required by the data flow analysis phase include, the elimination of useless instructions, the elimination of condition codes, the determination of register arguments and function return register(s), the elimination of registers and intermediate instructions by the regeneration of expressions, the determination of actual parameters, and the propagation of data type across subroutine calls. Most of these transformations are required to improve the quality of the low-level intermediate code, and to reconstruct some of the information lost during the compilation process. In the case of the elimination of useless instructions, this step is required even for optimizing compilers when there exist machine instructions that perform more than one function at a time.

Conventional data flow analysis is the process of collecting information about the way variables are used in a program, and summarizing it in the form of sets. The decompiler to transform and improve the quality of the intermediate code uses this information.

Several properties are required by code-improving transformations, including:

1. A transformation must preserve the meaning of programs.

2. A transformation must be worth the effort.

Techniques for Decompilation optimization of the intermediate code are presented here.

**3.1 Code Improving Optimizations:**

This section presents the code-improving transformations used by a decompiler. The aim of these optimizations is to eliminate the low-level language concepts of condition codes, registers, and intermediate instructions, and introduce the high-level concept of expressions of more than two operands. For this purpose, it is noted that push instructions are used in a variety of ways by today's compilers. Parameter passing is the most common use of this instruction, by pushing them before the subroutine call, in the order specified by the calling convention in use. Register spilling is used whenever the compiler runs out of registers to compute an expression. Push and pop are also used to preserve the contents of registers across procedure calls, and to copy values into registers.

###### Dead-Register Elimination

An identifier is dead at a point in a program if its value is not used following the definition of the variable. It is said that the instruction that defines a dead identifier is useless, and thus can be eliminated or removed from the code.

1 ax = tmp / di

2 dx = tmp % di

3 dx = 3

4 dx:ax = ax \* dx

5 si = ax

Instruction 1 defines register ax, instruction 2 defines register dx, and instruction 3 redefines register dx. There is no use of register dx between the definition at instruction 2 and instruction 3, thus, the definition of register dx at instruction 2 is dead, and this instruction becomes useless since it defines only register dx. The previous sequence of instructions is replaced by the following code:

1 ax = tmp / di

2 dx = 3

3 dx:ax = ax \* dx

4 si = ax

The definition of register dx at instruction 3 is used in the multiplication of instruction 4, where the register is redefined, as well as register ax. Instruction 5 uses register ax, and there are no further uses of register dx before redefinition of this register , thus, this last definition of dx is dead and must be eliminated. Since instruction 4 defines not only dx but also ax, and ax is not dead, the instruction is not useless as it still defines a live register; therefore, the instruction is modified to reflect the fact that only register ax is defined, as follows:

1 ax = tmp / di

2 dx = 3

3 ax = ax \* dx

4 si = ax

###### Dead-Condition Code Elimination

In a similar way to dead-register elimination, a condition code is dead at a point in a program if its value is not used before redefinition.

###### Condition Code Propagation

Condition codes are flags used by the machine to signal the occurrence of a condition. In general, several machine instructions set these flags, ranging from1 to 3 different flags being set by the one instruction, and fewer instructions make use of those flags, only using 1 or 2 flags. After dead-condition code elimination, the excess definitions of condition codes are

eliminated, thus, all remaining flags are used by subsequent instructions.

**4. Control Flow Analysis**

The control flow graph constructed by the front-end has no information on high-level language control structures, such as if-then-else and while () loops. Such a graph can be converted into a structured high-level language graph by means of a structuring algorithm. High-level control structures are detected in the graph, and sub graphs of control structures are tagged in the graph.

**4.1 Structuring Algorithms**

In Decompilation, the aim of a structuring algorithm is to determine the underlying control structures of an arbitrary graph, thus converting it into a functional and semantic equivalent graph. Arbitrary graph stands for any control flow graph, reducible or irreducible, from a structured or unstructured language. Since it is not known what language the initial program was written in, and what compiler was used (e.g. what optimizations were turned on), the use of goto jumps must be allowed in case the graph cannot be structured into a set of generic high-level structures.

**Structuring Loops**

In order to structure loops, a loop in terms of a graph representation needs to be defined. This representation must be able to not only determine the extent of a loop, but also provide a nesting order for the loops. The representation of a loop by means of cycles is too fine a representation since loops are not necessarily properly nested or disjoint. The use of strongly connected components as loops is too coarse a representation as there is no nest ing order. The use of strongly connected regions does not provide a unique cover of the graph, and does not cover the entire graph. Once a loop has been found, the type of loop (e.g. pre-tested, post - tested, endless) is determined.

**Finding the type of Loop**

The type of a loop is determined by the header and latching nodes of the loop. In a pre- tested loop, the 2-way header node determines whether the loop is executed or not, and the 1- way latching node transfers control back to the header node. A post-tested loop is characterized by a 2-way latching node that branches back to the header of the loop or out of the loop, and any type of header node. Finally, an endless loop has a 1-way latching node that transfers control back to the header node, and any type of header node.

**Finding the Loop Follow Node**

The loop follow node is the first node that is reached after the loop is terminated. In a pre-tested loop, the follow node is the successor of the loop header that does not belong to the loop. In a similar way, the follow node of a post-tested loop is the successor of the loop- latching node that does not belong to the loop. Initially, no follow nodes are taken for endless loops as neither the header nor the latching node jump out of the loop. But since an endless loop can have a jump out of the loop in the middle of the loop (e.g. a break in C), it can too have a follow node. Since the follow node is the first node that is reached after the loop is ended, it is desirable to find the closest node that is reached from the loop after an exit is performed. The closest node is the one with the smallest reverse post order numbering; i.e. the one that is closest to the loop (in numbering order). Any other node that is also reached from the loop can be reached from the closest node (because it must have a greater reverse post order numbering), thus, the closest node is considered the follow node of an endless loop.

**Structuring 2-way Conditionals**

Both a single branch conditional (i.e. if-then) and a conditional (i.e. if-then-else) sub graph have a common end node, from here onwards referred to as the follow node, that has the property of being immediately dominated by the 2-way header node. Whenever these sub graphs are nested, they can have different follow nodes or share the same common follow node.

A generalization of this example provides the algorithm to structure conditionals. The idea of the algorithm is to determine which nodes are header nodes of conditionals, and

which nodes are the follow of such conditionals. The type of the conditional can be determined after finding the follow node by checking whether one of the branches of the header node is equivalent to the follow node. Inner conditionals are traversed first, then outer ones, so a descending reverse post order traversal is performed (i.e. from greater to smaller node number). A set of unresolved conditional follow nodes is kept throughout the process. This set holds all 2-way header nodes for which a follow has not been found. For each 2-way node that is not part of the header or latching node of a loop, the follow node is calculated as the node that takes it as an immediate dominator and has two or more in-edges (since it must be reached by at least two different paths from the header). If there is more than one such node, the one that encloses the maximum number of nodes is selected (i.e. the one with the largest number). If such a node is not found, the 2-way header node is placed on the unresolved set. Whenever a follow node is found, all nodes that belong to the set of unresolved nodes are set to have the same follow node as the one just found (i.e. they are nested conditionals or unstructured conditionals that reach this node).

**Structuring n-way Conditionals**

N-way conditionals are structured in a similar way to 2-way conditionals. Nodes are traversed from bottom to top of the graph in order to find nested n-way conditionals first, followed by the outer ones. For each n-way node, a follow node is determined. This node will optimally have n in-edges coming from the n successor nodes of the n-way header node, and be immediately dominated by such header node.

**4.2 Application Order**

The structuring algorithms presented in the previous three sections determine the entry and exit (i.e. header and follow) nodes of sub graph that represent high-level loops, n-way, and 2-way structures. While implementing the algorithms that an ordering is to be followed, and it is: structure n-way conditionals, followed by loop structuring, and 2-way conditional structuring last. Loops are structured first than 2-way conditionals to ensure the Boolean condition that form part of pre-tested or post-tested loops is part of the loop, rather than the header of a 2-way conditional sub graph. Once a 2-way conditional has been marked as being in the header or latching node of a loop, it is not considered for further structuring.

**5 The Back-end**

The high-level intermediate code generated by the data flow analyzer, and the structured control flow graph generated by the control flow analyzer, are the input to the back-end.

This module is composed in its entirety by the code generator, which generates code for the target high-level language.

The code generator generates code for a predefined target high-level language. The following examples make use of the C language as target language, and the examples are based on the sample control flow graph after structuring information has been summarized on the graph.

**5.1 Generating Code for a Basic Block**

After data flow analysis, the intermediate instructions in a basic block are all high-level instructions; pseudo high-level instructions must have been eliminated from the code before this point. For each basic block, the instructions in the basic block are mapped to an equivalent instruction of the target language. Transfer of control instructions (i.e. jcond and jmp instructions) is dependent on the structure of the graph (i.e. they belong to a loop or a conditional jump (2 and n ways), or be equivalent to a goto), and hence, code is generated for them according to the control flow information. This section illustrates how code is generated for all other instructions of a basic block.

###### Generating Code for asgn Instructions

The asgn instruction assigns to an identifier an arithmetic expression or another identifier.The decompiler in abstract syntax trees stores expressions; therefore, a tree walker is used to generate code for them. Consider the instruction “asgn loc3, 5”.The left hand side is the local identifier loc3 and the right hand side is the constant identifier 5.

Since both expressions are identifiers, the code is trivially translated to:

loc3 = 5

The following instruction uses an expression in its right hand

asgn loc3, (loc3 + loc4) - 10

The abstract syntax tree represents this instruction; only the right hand side of the instruction is stored in the abstract syntax tree format (field arg of the intermediate triplet).

From the tree, the right hand side is equivalent to the expression

(loc3 + loc4) - 10,

and the C code for this instruction is:

loc3 = (loc3 + loc4) - 10;

Generating code from an abstract syntax tree is solved in a recursive way according to the type of operator; binary or unary. For binary operators, the left branch of the tree is traversed, followed by the operator, and the traversal of the right branch. For unary operators, the operator is first displayed, followed by its sub tree expression. In both cases, the recursion ends when an identifier is met (i.e. the leaves of the tree).

###### Generating Code for call Instructions

The call instruction invokes a procedure with the list of actual arguments. This list is stored in the arg field and is a sequential list of expressions (i.e. arithmetic expressions and/or identifiers). The name of the procedure is displayed followed by the actual arguments, which are displayed using the tree walker algorithm.

###### Generating Code for ret Instructions

The ret instruction returns an expression/identifier in a function. If the return instruction does not take any arguments, the procedure is finished at that statement. The return of an expression is optional.

**5.2 Generating Code from Control Flow Graphs**

The information collected during control flow analysis of the graph is used in code generation to determine the order in which code should be generated for the graph. The generation of code from a graph can be viewed as the problem of generating code for the root node, recurring on the successor nodes that belong the structure rooted at the root node (if any), and continue code generation with the follow node of the structure.

**6 Challenges**

1**.** The main difficulty of a decompiler parser is the separation of code from data, that is, the determination of which bytes in memory represent code and which ones represent data.

2. Less mature. Skilled staff sparse.

3. The model can be imperfect. Salvaging partial information is still useful.

4. Less certain. An implementation can yield different requirements, depending on the

reverse engineer’s interpretation.

5. The determination of data types such as arrays, records, and pointers.

6. Sometimes the programs are more difficult to extract the algorithm.

7. Now-a-days the Programmers are obfuscating the main code so that it becomes more

complex to dissemble the source code by using above tools and procedures.

**7 Conclusion**

This seminar report has presented techniques for the reverse compilation or Decompilation of binary programs. Decompilers use similar principles and techniques used in compilers. The data flow analysis phase analyzes the low-level intermediate code and converts it into a high-level intermediate representation available in any high-level language. Finally, the code generation phase generates high-level code based on the high-level intermediate representation and the structured graph of each subroutine. In this way, a decompiler for a different machine can be built by writing a new front-end for that machine, and a decompiler for a different target high-level language can be built by writing a new back-end for the target language.

Further work on Decompilation can be done in two areas: the separation of code and data, and the determination of data types such as arrays, records, and pointers. The former area needs a robust method of determining n-way branch statements (i.e. indexed jumps) and indirect subroutine calls. The latter area needs heuristic methods to identify different types of

compound data types and propagate their values.

All software is made up of machine-readable code. In fact, code is what makes every program function the way it does. The code defines the software and the decisions it will make. Reverse engineering, as applied to software, is the process of looking for patterns in this code. By identifying certain code patterns, an attacker can locate potential software vulnerabilities.

Incentives to reverse engineer computer programs will continue to be an important method of understanding principles and functionality in existing software products.

**Bibliography**

* A. V. Aho, R. Sethi, and J. D. Ullman. Principles of Compiler Design. Narosa
  + - Publication House, 1985.
* Michael Blaha and James Rumbaugh. Object-Oriented Modeling and Design with
  + - UML. Second Edition, Pearson Education 2008.
* Cristina Cifuentes. Reverse Compilation Techniques, PhD Thesis. Queensland
  + - University of Technology, 1994.

**References**

* <http://www.cc.gatech.edu/reverse>.
* <http://opensecuritytraining.info/IntroductionToReverseEngineering.html>
* <https://blog.udemy.com/reverse-engineering-tutorial/>
* <http://althing.cs.dartmouth.edu/local/www.acm.uiuc.edu/sigmil/RevEng/>
* <https://www.eff.org/issues/coders/reverse-engineering-faq>
* <https://ethics.csc.ncsu.edu/intellectual/reverse/study.php>
* <http://www.austlii.edu.au/au/journals/JlLawInfoSci/2003/2.html#Heading38>