

Automatic Execution Path Finding Tool

A Thesis Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Fan Yang

Fall 2010

© 2010

Fan Yang

ALL RIGHTS RESERVED

SAN JOSE STATE UNIVERSITY
AUTOMATIC EXECUTION PATH FINDING TOOL

by
Fan Yang

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp, Department of Computer Science DATE

Dr. Robert Chun, Department of Computer Science DATE

Dr. Xiuduan Fang, Software engineer in Google DATE

APPROVE FOR THE UNIVERSITY

Associate Dean, Office of Graduate Studies and Research DATE

ACKNOWLEDGEMENTS

I am indebted to my advisor, Dr. Mark Stamp, for helping me to complete this paper and giving me a lot of useful advice when I needed help. Dr. Mark Stamp always supported me in my masters project. And thanks to Dr. Robert Chun and Dr. Xiuduan Fang for serving on my defense committee

Thank you to my family and friends, Lin Huang, Leyan and Yue Chen for supporting my studies and my life.

ABSTRACT

AUTOMATIC SOFTWARE PATCHING TOOL

By FanYang

Today, there are many hackers trying to break software using reverse engineering techniques. To better protect software, we need to understand reverse engineering methods. This project presents a tool that automatically analyzes executable code in a manner similar to the way an attack might analyze code.

Using reverse engineering techniques, executables can be disassembled to yield an assembly code representation of the program. Assembly code is much more human readable than binary code. With assembly code, an attacker can attempt to modify the program to do something other than what was intended by the developer.

The tool that we have developed allows the user to specify any two points in the assembly code, and then attempts to find a valid execution path between the two specified points. That is, the goal of this project is to develop a tool that will automatically find a path between two given points in the assembly code, say, point A and point B, and then modify the executable so that if point A is reached, then point B will be reached later.

Keywords: Reverse Engineering, Security, Ollydbg, IDA Pro, Tree Data Structure

Contents

Contents	vi
1.0 Introduction	1
2.0 Reverse Engineering Background	2
2.1 Code Reverse Engineering	2
2.2 Data Reverse Engineering	3
2.3 Current Development	4
2.4 Future Trends in Program Analysis	5
2.5 Some Tools for Reverse Engineering	6
3.0 Finding the Execution Path	8
3.1 A Typical Case for Reverse Engineering	8
3.2 Analyze the Problem	10
3.3 Storing and Traversal Algorithm	11
4.0 Implementation	13
4.1 Developing the Tool	13
4.2 Principle for OllyDbg Plug-in	14
4.3 Implementation Details and Data Analysis	16
4.4 Statistics for Testing Cases	32
4.5 Optimization	34
5.0 Future works	35
5.1 Improve The Algorithm for Tree Traversal	35
5.2 Optimize Depth-First Search Algorithm	36
5.3 Analyze the Functionality	36
6.0 Conclusion	36
Reference	38

Table of Figure

Figure 1. Ollydbg main windows	7
Figure 2. Typical case for authentication.....	9
Figure 3. Tree data structure	12
Figure 4. Modules relation.....	17
Figure 5. Main flow for the program	18
Figure 6. Plug-in GUI	19
Figure 7. Screenshot before patching.....	22
Figure 8. The first solution.....	23
Figure 9. The second solution.....	24
Figure 10. Screenshot for visited instructions file	25
Figure 11. Screenshot for case one part two	26
Figure 12. Screenshot for case two	26
Figure 13. Path for case two	27
Figure 14. Screenshot for case three	28
Figure 15. Screenshot of the unsuccessful path searching result	29
Figure 16. Screenshot for case four.....	30
Figure 17. Snapshot for case four	31
Figure 18. Snapshot for case five.....	31
Figure 19. Result for case five	32
Figure 20. Testing cases statistics by Depth-First.....	32
Figure 21. Testing cases statistics by Breadth-First.....	33
Figure 22. Extreme case for binary tree.....	35

1.0 Introduction

Reverse Engineering was presented by Chikofsky and Cross in the 1990s [1]. The main idea is to discover the technological principles or implementation information of software by analyzing software structure or functions. Reverse engineering techniques are used to reproduce or reuse the structure extracted from a software system of which source code is not available or accessible. For example, scholars or developers can study the programming technology by analyzing others' software and then improve their own software development skills. Reverse engineering is currently successfully applied to some areas, such as recovering design patterns [2], reverse engineering binary code [3], or recovering architectures [4].

However, reverse engineering techniques have been extended to break software systems. Since developing new software has always been considered much more welcome than maintaining the existing systems, not much attention has been given by developers to doing the research and developing reverse engineering techniques. However, reverse engineering has recently become an attractive topic in computer science. More and more developers have come to participate in this field of computer science. Reverse engineering is ready to be taught in computer science and other related programs [5]. Reverse engineering techniques can also enhance software engineering [15]. Just like everything else, software reverse engineering can be used for good or malevolent purposes. Hackers can also make a Trojan version or insert virus into the software using reverse engineering technology. More realistically, users could patch software so that it could be used without registration or payment. For example, software does not normally function if the incorrect serial

number is input, but it can be used if the software has been patched, even if the serial number is incorrect.

2.0 Reverse Engineering Background

Reverse engineering is a very broad topic. It can be divided into two categories: Code Reverse Engineering and Data Reverse Engineering [1]. The form of reverse engineering covered by this paper is focused on the code level.

2.1 Code Reverse Engineering

Code reverse engineering is important in that the key business rules are buried in the code of the legacy systems. In nature, software is evolutionary, to update function, add new features, or improve performance and enhance quality. Additionally, the documentation for the software or system is very poor in most cases and the source code is probably the only reliable information for the system.

Over the past several years, a few techniques of code reverse engineering have been proposed to analyze code. In general, these techniques work well when handling the software at the syntactic level. But that may not be sufficient for researchers. For example, the architecture and engineering constraints are not available when analyzing the code, since it is in the software engineers' mind. Thus, we will miss the big picture behind the software system if we focus solely on these low levels of abstraction. The need for focusing on much higher levels of software architecture should be considered in future research.

Now we will take a look at an example for code reverse engineering. Due to different compile principles and language design, there are several types of code level reverse engineering design. One is to reverse the byte code generated by JAVA or C#, which is easy to do. We can simply download a de-compiler and input the byte code. Then the output is pure source code that is usually very close to the original version. Therefore, to make it hard to reverse engineer source code, avoid using Java or C#. Another case is to reverse executables produced from compiling C or C++. Developers can still find debuggers and de-compilers for such executables. However, the output is in assembly code rather than pure source code, making much harder for developers to read and get the structure of the original code.

2.2 Data Reverse Engineering

While code reverse engineering helps software engineers to understand how a function is executed, data reverse engineering is focused on what information a system has. Data reverse engineering is a new approach to deal with data disintegration problems combining structured data analysis techniques [6]. Data reverse engineering is not as popular as code reverse engineering since code reverse engineering is considered challenging and interesting. However, data reverse engineering has received more attention than it did a couple of years ago. The reasons are various. One is that researchers realize that data documentation recovered by these techniques can help migrating software systems. Another reason is that data mining techniques have become more popular to analyze data, driving the development of data reverse engineering.

In summary, data reverse engineering has drawn much attention in recent years. As a complement to code reverse engineering, data reverse engineering helps users to extract more information from an existing system.

2.3 Current Development

Reverse engineering is a helpful technique to analyze the algorithm or programs from an existing system. Additionally, it is useful to maintain or reduce the system. The importance of reverse engineering is recognized by scholars all over the world. Conferences such as WCRE (The Working Conference on Reverse Engineering), PASTE (The workshop on Program Analysis for Software Tools and Engineering), IWPC (International Workshop on Program Comprehension) are held annually [1]. The main development on reverse engineering is concentrated on code understanding. For example, reverse engineering outperformed on Millennium Bug making people realize the importance of reverse engineering. This is one reason why it is becoming more and more popular today. In particular, the main achievements of reverse engineering are concerned with program analysis, design recovery, and software visualization [7]. Many useful tools for reverse engineering have also been developed and will be discussed later in this paper.

Though reverse engineering development has been around for several decades, there are still some unsolved problems. For example, there is no unified concept, frame or terms. Moreover, most tools are not well integrated with other tools, restricting flexibility. Overall, reverse engineering is still in its early stages, and we can expect to see improvement in the future.

2.4 Future Trends in Program Analysis

Reverse engineering will aid software comprehension and server maintenance and reengineering. However it still has some challenges [12]. One of the key challenges for reverse engineering is the high dynamic feature for programs [7]. Many programs use powerful language features. For example, Java supports reflection, which allows a class to be loaded at run time. Under such circumstances, some reverse engineering techniques, such as static analysis, become useless. The reason is that it is impossible to find the reference for an object that will be loaded at run time. Therefore, studying analysis of static and dynamic dependence [14] and developing dynamic analysis for reverse engineering to complement the constraints of static analysis is necessary.

Another equally important challenge comes from the cross-language applications. The analysis tools must cover diverse languages and techniques for analyzing software. For example, developers can write some C code and integrate that with the java code using Java Native Interface, or including some SQL script in the program. Hence, simply analyzing the program in Java is not enough.

Additionally, the obfuscation technique makes reverse engineering more difficult [13]. For instance, users can normally get Java source code by using reverse byte code. However, the readability for the code is greatly reduced if the original code has been obfuscated.

In summary, as programming techniques develop, reverse engineering techniques should also be updated; otherwise, they will become outdated and useless.

2.5 Some Tools for Reverse Engineering

There are numerous debugger tools on the market, either commercialized or for academic purposes. In this section, we will present the development of reverse engineering, as well as the main functionality for each of the useful tools.

There are many good tools for reverse engineering. Depending on the functionalities, these tools are described by different names, such as de-compilers, disassemblers, obfuscators, shrinkers, optimizers, preverifiers, and so on. All of them are very useful for reverse engineering. Developers can use these tools to enhance the security of their programs to avoid attacks. But hackers can also glean some useful information and find defects. Here are several tools for software reversing, security checking, and reuse:

CafeBabe: a Java bytecode editor. It works as an editor or disassembler for Java bytecode [8].

COBF: a source code obfuscator. It makes it difficult for programmers to understand the source code, but the obfuscated code still has the same functionality as the original source code [8].

Jad: a Java bytecode de-compiler. It attempts to produce Java source code from a given bytecode [8].

Reverse Engineering Compiler: a machine code de-compiler. It reads an executable file and attempts to produce a C-like representation of the code [8].

IDA Pro: an interactive debugger and disassembler for binary code [8].

Another very popular and useful tool in reverse engineering for machine code is OllyDbg. OllyDbg is an analyzing debugger for 32-bit assembler level [9]. It is shareware, but the users can use it for free. OllyDbg is a new dynamic behavior tracing tool, comparable with the idea of IDA Pro and SoftICE. Since OllyDbg is freely available and easy to use, it is currently the most popular debugger and disassembler. Furthermore, OllyDbg also provides an open framework for plug-in. Enthusiasts can develop different kinds of plug-ins to OllyDbg.

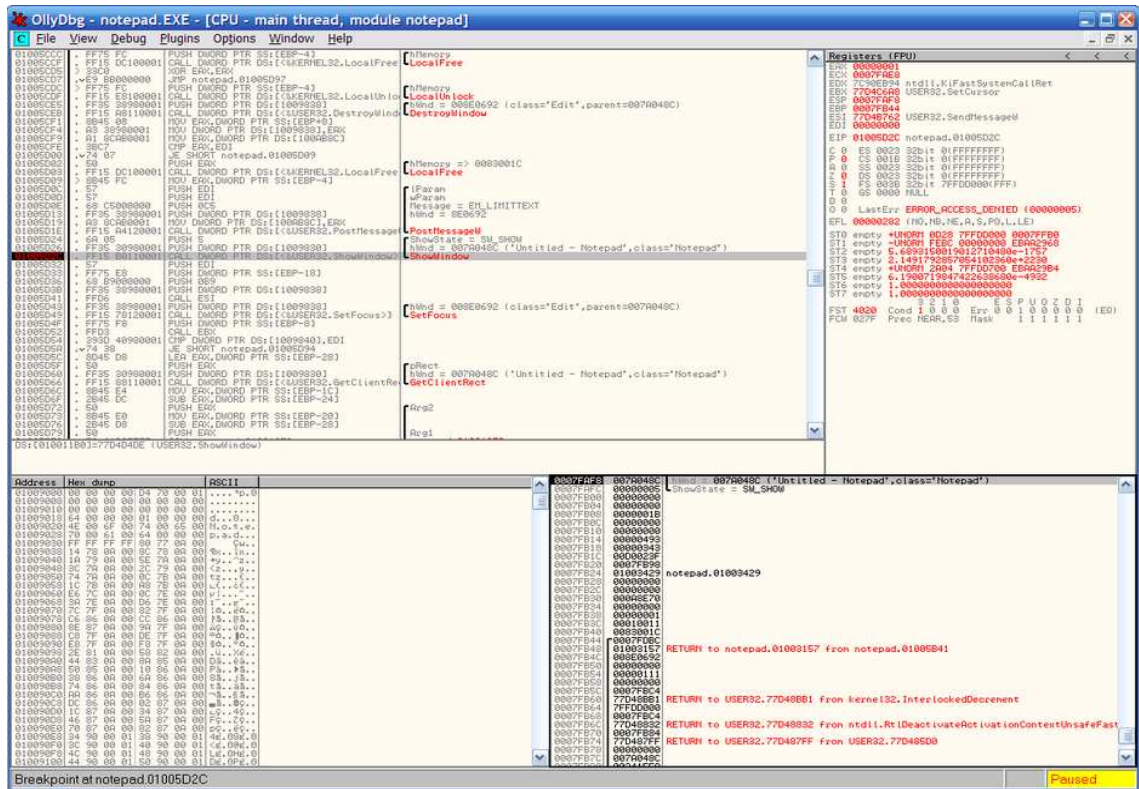


Figure 1. Ollydbg main windows

Figure 1 shows a screenshot of the main window of the OllyDbg software. The main user interface of OllyDbg is composed of several windows: CPU window, LOG window, executable modules window, thread window, and so on.

The de-compiler window is composed of four items: virtual address, hex dump, disassembly code, and comment. The language used to develop software is quite important. Users can use OllyDbg to decompile any executable files developed in C/C++, Pascal, however, not all the executable files can be decompiled correctly. For example, if the author has developed a program using C#, which is similar to Java, then no useful information will be obtained through OllyDbg. C# language has its own de-compiler that can convert the executable file directly to the source code.

3.0 Finding the Execution Path

3.1 A Typical Case for Reverse Engineering

As discussed above, users should have no problem decompiling byte code, since the output would be pure source code. But users have difficulty in disassembling executable files, since only the assembly code is available. Users probably will need to study the assembly code so that they can get some information from the application, such as the structure of some portion of the software, or the execution path of a certain function. For example, if we look at a serial verification function in assembly, we probably will find some interesting string, e.g. "Please input the serial number," and "Congratulations, the serial number is correct." These two messages are very sensitive in this code, since it provides the exact information that the function will show to users. In other words, these

two points should be like an entrance and an exit for the serial number authentication function with the starting point at “input the serial number” and ending point at “the serial number is correct” if the serial number is approved. A similar situation to that discussed above is shown in the Figure 2:

00401310	. E0 70050000	CALL serial.00401000	
00401315	. E8 16010000	CALL serial.00401430	
0040131A	. C70424 008040	MOV DWORD PTR SS:[ESP],serial.00408000	ASCII 0A,"Enter Seri"
00401321	. E8 54000000	CALL serial.0040137A	
00401326	. 8D45 D8	LEA EAX,DWORD PTR SS:[EBP-28]	
00401329	. 894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
0040132D	. C70424 168040	MOV DWORD PTR SS:[ESP],serial.00408016	ASCII "%s"
00401334	. E8 7F500000	CALL <JMP.&msvcrt scanf>	scanf
00401339	. 8D45 D8	LEA EAX,DWORD PTR SS:[EBP-28]	
0040133C	. C74424 08 0800	MOV DWORD PTR SS:[ESP+8],8	
00401344	. C74424 04 1980	MOV DWORD PTR SS:[ESP+4],serial.00408016	ASCII "DEADBEEF"
0040134C	. 890424	MOV DWORD PTR SS:[ESP],EAX	
0040134F	. E8 5C500000	CALL <JMP.&msvcrt strcmp>	strcmp
00401354	. 85C0	TEST EAX,EAX	
00401356	. 74 0E	JE SHORT serial.00401366	
00401358	. C70424 248040	MOV DWORD PTR SS:[ESP],serial.00408024	ASCII "Error! Incorrect serial num"
0040135F	. E8 16000000	CALL serial.0040137A	
00401364	. EB 0C	JMP SHORT serial.00401372	
00401366	.> C70424 508040	MOV DWORD PTR SS:[ESP],serial.00408050	ASCII "Serial number is correct.␣"
0040136D	. E8 08000000	CALL serial.0040137A	

Figure 2. Typical case for authentication

Can we force the program to execute to the instruction of showing the message of “the serial number is correct” even if we do not have the correct password? To perform such a task, we need to change the execution sequence defined by the executable. After examining the code carefully, we find that there is a conditional jump above the message. Depending on the value of the conditional jump, the execution path can go to one path if the condition is true, or go to another path if the condition is false. If we can change the condition for the jumps, for example, changing the condition from true to false, or if we can change the type of jump instruction, for example, changing the jump instruction to its opposite, then the executing sequence will be changed as we expect. Basically, there are two steps to achieve the goal. The first step is to find the path between the two points, another step is to change some condition and make the jump true or not true. Usually, these are done by loading the executable in a debugger and tracking the execution step by

step manually. If these two points are located only a few jumps away from each other, it is not hard to hack the underlying logic. But sometimes the path between two points is long, and users cannot easily find it. Furthermore, finding the path between the two points manually is always tedious and time consuming. A tool that allows users to input the two points and automatically patch the file would be very useful. This project has been established to fill this need. Specifically, the goal of this project is to develop a tool to find a path between two arbitrary points in an executable, and if the path is found, it will try to change the binary code to guarantee that once the beginning point is entered, the end point will be reached. The section bellow will describe how to design and implement such a tool.

3.2 Analyze the Problem

All the binary files can be translated into assembly code by the de-compilers. The route for the program is composed of assembly instructions. A good analogy is that we can treat the program as a map roadmap of the United States, and all the functions or sub functions in the program are treated as different sites in the map. The users are driving on the map. According to the roadmap, a driver can go to other places within the U.S, but there may be some sites that are forbidden for security reasons, so some sites cannot be reached, although they do exist. Similarly, a hacker always wants to approach an execution path. In order to get to the specific site, the hacker should create a route which can reach the place without any blockage. To find such an execution path, we can think of a question: Can go to Google from San Jose State University?

How can we go to Google from San Jose State University? Normally, the first step is to search the highway that is connected to San Jose State University, and we will find 280. But we cannot get to Google directly from 280, so we need to find which highway is connected to 280, and then we get 87, 680, and 101. The next step is to check whether one of them is connected to Google. We find we can go to Google via Highway 101. Finally, recording all the highways together, we can get the route of: 280-101. This is an ideal situation. We can find the route and there is no blockage between points. But what if one site requires a ticket to get in, and we do not have that ticket? In that case, we get a route, but not a throughway. The algorithm for that problem should be able to store all the connection sites on the map. According to this feature of the problem, the tree data structure can be used to store the map.

3.3 Storing and Traversal Algorithm

A tree is an important data structure developers can use to deal with the diagrams. By definition, a tree is a collection, and it maps the relationship for that collection. The elements of the collection are called nodes, and the relationship is called parents or children. A special node in the tree is the root node, which is on the top of the tree. For the route problem, the source site corresponds to the root node in the tree, and all other sites should be its children. Specifically, the task is to find a path between the root node and a specific node in the tree. Of course, due to the limitations of memory, the process cannot store all the nodes. They should be stored according to the distance to the root node. Figure 3 shows an example of data structure.

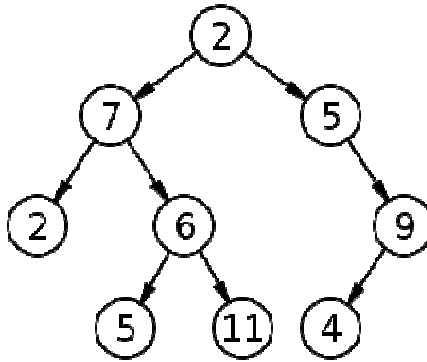


Figure 3. Tree data structure

Resource: http://en.wikipedia.org/wiki/File:Binary_tree.svg

It seems that the process of finding the route is to perform a tree traverse, but it is actually slightly different from traversing a tree, since there is no predefined tree to be traversed.

The nodes on a map are interconnected, and there are some loops among the nodes. In fact, we build the tree structure while we traverse the path. Generally, there are two types of traversal: Depth-First Traversal [10] and Breadth-First Traversal [11].

The main process of the Depth-First Traversal is presented below. Suppose that x is the currently visited node. After x is visited, find an edge that connects from x to another node y . If y has already been visited, then try to find another node that hasn't been visited, otherwise, reach the unvisited node y and change the flag to show it has been visited. Continue to search from node y until all the routes have been visited. Then go back to node x , and try to find another edge from x that has not been visited. If x is not the root, then trace back to the nodes that have been visited before, otherwise all the nodes on the tree map have been visited and the whole traversal is finished [8].

Breadth-First Traversal is a little different from Depth-First Traversal. If the currently visited node is x , then find all the nodes next to node x that have not been visited. Visit all these nodes. Repeat the steps above until all the nodes have been visited [9].

It is easy to find the shortest route between two nodes using Breadth-First Traversal, but the progress may not be very efficient, since a large amount of memory may be required to store the nodes.

On the other hand, Depth-First Traversal can do a better job of finding the route if the distance between the two points is long. However, if the destination is just under a conditional jump instruction, this algorithm will still jump to another place and then return back and find the destination that is just under it. This method is really not very efficient for finding the path. Since both the traversals have advantages and disadvantages, selection between them should be based on the specific requirements of the user. Specifically, Depth-First Traversal is better if the destination is in the branch that it will visit first, and Breadth-First Traversal is better if the destination is only a couple of jumps away from the beginning.

4.0 Implementation

4.1 Developing the Tool

One solution to deal with the assembly code is to develop a plug-in to a disassembler.

The disassembler could help us to convert object code into an assembly code, and therefore can save us a lot of work. We know that OllyDbg not only is an excellent debugger for assembler level analyzing, but also provides a lot of useful APIs combined with de-

tailed documents. OllyDbg is mature software. Anyone can download the software and the plug-in development kit, named PDK, from the Office web site. The programming language for OllyDbg plug-in development is C language. The optional compilers suggested by the author for developing the plug-in are Borland's C++5.5, Borland's C++ Builder 5, and Microsoft's Visual C++ 5.0. In this case, we have chosen Borland's C++ 5.5. There are some subtle differences among different compilers when building plug-ins. Fortunately, all the compile details have already been provided by the author. What the users need to do is to follow the instructions step by step and build their own plug-in for Ollydbg. This paper will discuss in detail how to develop a plug-in on OllyDbg in the following sections.

4.2 Principle for OllyDbg Plug-in

As mentioned above, OllyDbg is an excellent dynamic debugger tool. Not only does it have a very powerful de-compile and analysis ability, but it also has a very good framework that allows users to develop their own plug-in on it.

Before discussing how to develop a plug-in, let us first take a look at how the plug-in works in OllyDbg. The plug-in is provided as a dynamic link library (DLL) to OllyDbg. Then the plug-in will be visible under the menu. We can get details of how OllyDbg works from both the OllyDbg perspective and the plug-in perspective.

From the OllyDbg perspective, the first step for Ollydbg is to check whether it has the DLL file. If the file exists, it will perform the following tasks:

- (1) Load the DLL file, and find the entrance,

- (2) Call the callback functions to get the name and version information of the plug-in,
- (3) Call other callback functions to initialize the plug-in, including applying resources, recovering the global parameters, and so on.

If the DLL file cannot be executed for the three steps, OllyDbg will start unsuccessfully, send out an error message, and exit. Otherwise, it will maintain the queue of plug-in and send the message or call the functions in the plug-in when:

- (1) Users call the plug-in via menu or shortcut menu
- (2) The status of debugging is changed, for example, loading, running, pausing, and restarting
- (3) The system is shut down and up
- (4) The system cannot recognize the message, for example the key combination
- (5) The unreadable data in the configure file

Finally, OllyDbg will call the callback function to release all the resources, and store the parameters when it is shut down.

From the plug-in perspective, the plug-in will do some of the work below:

- (1) Collect and maintain the information for debug and provide it to the users
- (2) Add some additional information and make the debug much more convenient
- (3) Participate in the debugging process

(4) Load scripts

Generally speaking, not only does the plug-in need to get various kinds of information from OllyDbg, but also it needs to interact with OllyDbg. Basically, the plug-in calls the OllyDbg plug-in, API, to perform required functions.

One special function in API is the callback function. Normally, a plug-in will call the functions that are provided and implemented by the OllyDbg. However, the callback function will call the function implemented by the plug-ins. Note that the callback function is the method managed by OllyDbg.

4.3 Implementation Details and Data Analysis

4.3.1 Module Design

The program is used to analyze assembly code and get a possible solution for a user's requirements. So the project is mainly divided into four modules: De-compiler (DC), Code Analyzer (CA), Code Optimization (CO), and User Interface (UI). Figure 4 shows the relationship amongst the four modules.

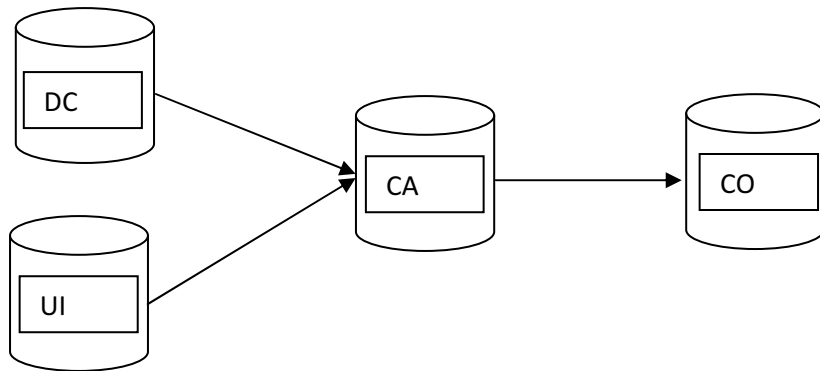


Figure 4. Modules relation

De-compiler is completed by Ollydbg which loads the object and disassembles to assembly code. User Interface allows users to input the instructions, which could be setting two points (beginning and ending points), starting optimization. Code Analyzer and Code Optimization are the key modules in the program. Code Analyzer finds a path between any two points in the assembly code. Code Optimization is then used to change the code to qualify the requirements for the path found during Code Analyzing period. Figure 5 shows the main flow of this application.

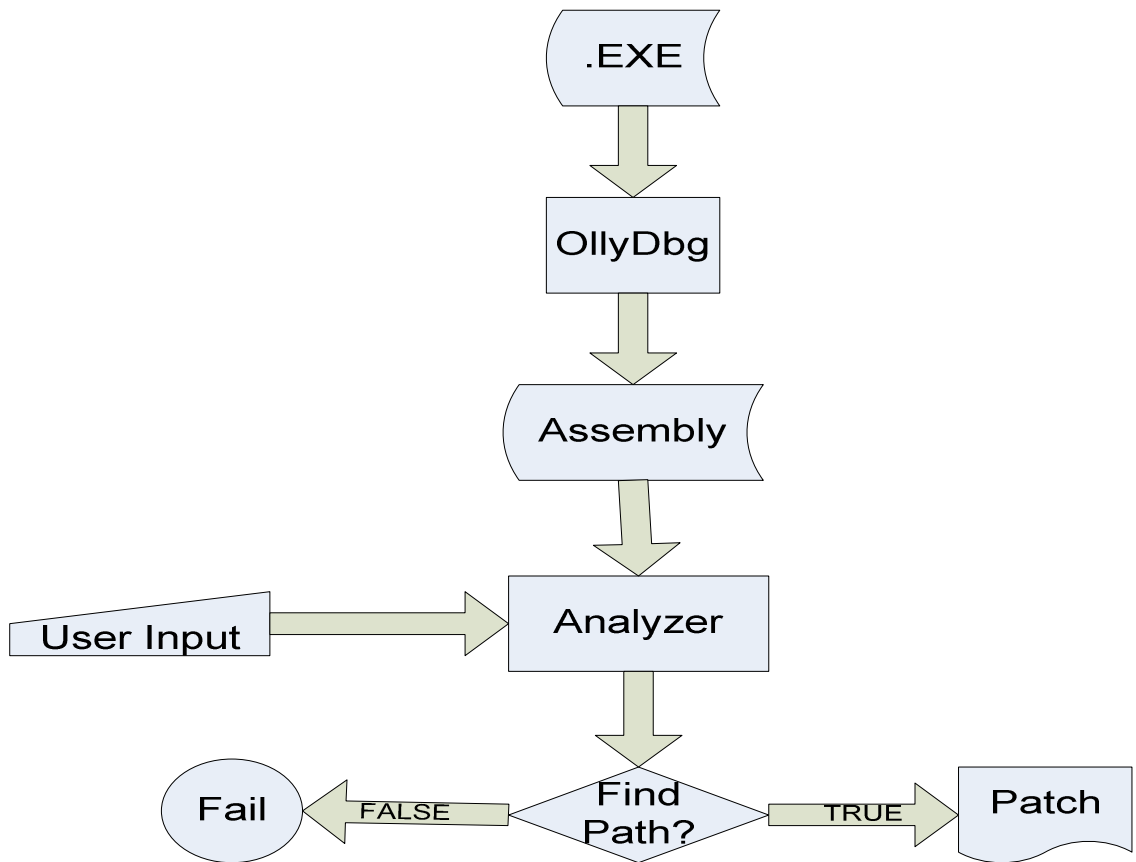


Figure 5. Main flow for the program

For this application, the binary file, which is shown as .exe above, will be loaded into OllyDbg and de-compiled to assembly code. At the same time, the user should provide the start and exit points, with the selected algorithm, Depth-First or Breadth-First. The program will then read the assembly automatically and try to find a possible path between these two points. If a path is found, it will call another function to patch the application, otherwise, the mission will fail and send out a message to show that no path is found. The Figure 6 shows a screenshot of the plug-in.

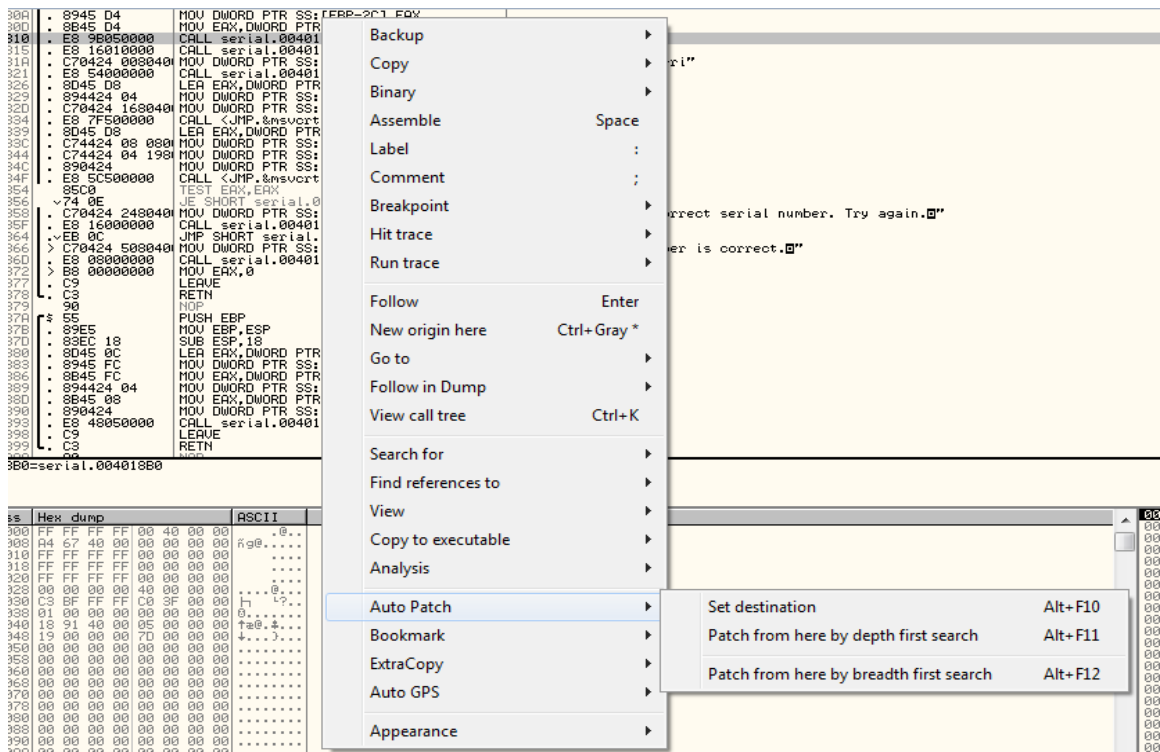


Figure 6. Plug-in GUI

Right click the button on the address that we want to set the destination or patch it from, and a pop-up window will show the menu. In Auto Patch, the following options are provided by the menu.

Set destination: set the ending point of the search

Patch from here by depth first search: use the Depth-First algorithm to find the path between the start point and the end point and then patch them.

Patch from here by breadth first search: use the Breadth-First algorithm to find the path between the start point and the end point then patch them.

Users have to set the destination point first and then go to the start point to do the search.

4.3.2 Basic Analysis for Assembly Code

There are four different types of jump instructions in the assembly code. The first one is a jump instruction which will jump the code to a specific location with no condition required. The second is a call instruction that is similar to the first one, but the difference is that the call function can call a system function to input or output some data, for example, input user name and password. The third one is a return instruction, which will make the code return to the location where the function is called. These three types of instructions are not affected by other instructions, and thus cannot be changed. However, the last type of jump instruction is conditional, for example, JE, JNE. Since the jump action is dependent on the previous calculation or comparison, the conditional jump provides users the opportunity to change the condition for the jump. The main task in Code Analyzing and Code Optimization is focused on conditional jump instruction in assembly code.

4.3.3 Depth-First Implementation

Basically, the idea for the Depth-First algorithm is that the assembly code must jump if it is a jump instruction, no matter if it is a conditional jump instruction or a non-conditional jump instruction. The program will run back if nothing is found. For the implementation, one stack and one array are used to deal with the jump instruction.

There is a slightly different usage between the two types of data structure. For a new jump point, the new node should be pushed to the stack and stored in the array, but the node will be popped off if no solution is found in the path, but not for array. The array is used to store all the possible jump instructions in the code and check whether they have already been visited when a new instruction is coming. The stack is used to store the cur-

rent jumped instructions. It will pop a part of them off if there is no path in the current sub path and push a new instruction for new path until reaching a conclusion.

4.3.4 Breadth-First Implementation

For the Breadth-First algorithm, the schema is quite different from the Depth-First algorithm. For this implementation, a queue is used to store all the possible jump instructions, and the sequence for visiting is from beginning to end. The new instructions will enqueue, and the next executing jump instructions will dequeue from the queue, making the jumps in a particular order.

4.3.5 Difficulty in Developing

One of the most challenging aspects of plug-in development is debugging. The source code has to be compiled into a dll file before being loaded by OllyDbg. Since OllyDbg is software, not a debugger, the user cannot debug to trace the interval value. Without a debugger, the only useful, though inconvenient, solution is to output all the interval values to a file.

Another problem found during the testing period is that it is hard to test code that is not in the executing sequence. The only thing we can do is to do static analysis.

4.3.6 Sample Cases

All the cases in this section 4.3.6 Sample Cases are tested on the serial.exe executable from Dr. Stamp's Reverse engineering course except test case three is from serial2.exe.

Case One:

In this case, we will demonstrate patching an executable using Depth-First algorithm and Breadth-First Algorithm. Even though both approaches yield the same results, we will show the difference during the process of finding the path.

```

00401300 | . 8B45 04      MOV EAX,DWORD PTR SS:[EBP-2C]
00401310 | . E8 9B050000  CALL serial.004018B0
00401315 | . E8 16010000  CALL serial.00401430
0040131A | . C70424 000040 MOV DWORD PTR SS:[ESP],serial.00400800 ASCII 0A,"Enter Seri"
00401321 | . E8 54000000  CALL serial.0040137A
00401326 | . 8D45 08      LEA EAX,DWORD PTR SS:[EBP-28]
00401329 | . 894424 04     MOV DWORD PTR SS:[ESP+4],EAX
0040132D | . C70424 160040 MOV DWORD PTR SS:[ESP],serial.00400816 ASCII "%s"
00401334 | . E8 7F500000  CALL <JMP.&msvcrt.scanf> scanf
00401339 | . 8D45 08      LEA EAX,DWORD PTR SS:[EBP-28]
0040133C | . C74424 08 0801 MOV DWORD PTR SS:[ESP+8],8
00401344 | . C74424 04 1981 MOV DWORD PTR SS:[ESP+4],serial.00400801 ASCII "DEADBEEF"
0040134C | . 890424      MOV DWORD PTR SS:[ESP],EAX
0040134F | . E8 5C500000  CALL <JMP.&msvcrt.strncmp> strncmp
00401354 | . 85C0        TEST EAX,EAX
00401356 | . 74 0E       IF SHORT serial.00401366
00401358 | . C70424 240040 MOV DWORD PTR SS:[ESP],serial.00400824 ASCII "Error! Incorrect serial number. Try again."
0040135F | . E8 16000000  CALL serial.0040137A
00401364 | . EB 0C       JMP SHORT serial.00401372
00401366 | . C70424 500040 MOV DWORD PTR SS:[ESP],serial.00400850 ASCII "Serial number is correct."
0040136D | . E8 08000000  CALL serial.0040137A
00401372 | . B8 00000000  MOV EAX,0
00401377 | . C9         LEAVE
00401378 | . C3         RETN

```

Figure 7. Screenshot before patching

Figure 7 above shows the assembly code after binary code, which is a simple serial number authentication function, is loaded on OllyDbg. It is not hard to find the three interesting messages in the code: “Enter seri” at 0040131A, “Error! Incorrect serial number. Try again” at 00401358, and “Serial number is correct” at 00401366. These messages are probably the most important information in this authentication for users who want to change the execution logic of the authentication function.

As we know, the computer processor executes instructions in sequence. So normally, the executing sequence of this piece of code will start at the beginning of the code and go down one by one. But how could this code execute to the address “Serial number is correct” (at 00401366) instead of execute to the address of “Error! Incorrect serial number.

Try again”(at 00401358)? Instructions that can change the execution path may give us a hint. At the address of 00401356, there is an condition jump instruction “JE SHORT 00401366”. An condition jump can choose one of the two different execution paths depending on the condition it exams. If the condition is no satisfied, the next instruction will show the message “Error, Incorrect serial number. Try again.” ,where we don’t want to get. However, it can jump to the address users desire to if it is qualified to jump. At this point, the previous instruction “TEST EAX,EAX” in this code plays an important role for the qualification. TEST instruction in the assembly code is to AND each bit for the parameters without saving the result, but the most importantly it could influence the flag bits, or the conditions that a condition jump will exam. One way to change the execution path is to change the instruction “TEST EAX,EAX” to “XOR EAX,EAX”, since the instruction of “XOR EAX, EAX” will set an opposite flag from what the “TEST EAX,EAX” does. Then the instruction will jump to the opposite destination on the same execution settings. The Figure 8 below shows how the code is modified.

00401310	. E8 70050000	CALL serial.00401000	
00401315	. E8 16010000	CALL serial.00401430	
0040131A	. C70424 008040	MOV DWORD PTR SS:[ESP],serial.00408000	ASCII 0A,"Enter Seri"
00401321	. E8 54000000	CALL serial.0040137A	
00401326	. 8D45 D8	LEA EAX,DWORD PTR SS:[EBP-28]	
00401329	. 894424 04	MOV DWORD PTR SS:[ESP+4],EAX	
0040132D	. C70424 168040	MOV DWORD PTR SS:[ESP],serial.00408016	ASCII "%s"
00401334	. E8 7F500000	CALL <JMP.&msvort.scanf>	scanf
00401339	. 8D45 D8	LEA EAX,DWORD PTR SS:[EBP-28]	
0040133C	. C74424 08 080	MOV DWORD PTR SS:[ESP+8],8	
00401344	. C74424 04 198	MOV DWORD PTR SS:[ESP+4],serial.0040801	ASCII "DEADBEEF"
0040134C	. 890424	MOV DWORD PTR SS:[ESP],EAX	
0040134F	. E8 5C500000	CALL <JMP.&msvort.strncmp>	strncmp
00401354	. 33C0	XOR EAX,EAX	
00401356	. 74 0E	JE SHORT serial.00401366	
00401358	. C70424 248040	MOV DWORD PTR SS:[ESP],serial.00408024	ASCII "Error! Incorrect serial number. Try again."
0040135F	. E8 16000000	CALL serial.0040137A	
00401364	. 74 0C	JMP SHORT serial.00401372	
00401366	. C70424 508040	MOV DWORD PTR SS:[ESP],serial.00408050	ASCII "Serial number is correct."
0040136D	. E8 08000000	CALL serial.0040137A	
00401372	. B8 00000000	MOV EAX,0	
00401377	. C9	LEAVE	

Figure 8. The first solution

Another solution is to replace the conditional jump with unconditional jump instruction “JMP”. No matter what condition they have, the instruction will jump to the address of 00401366. For the tool, this solution is selected and implemented in Code Optimization Module. We can see how the code is modified in Figure 9:

```

00401315 | . E8 16010000 | CALL serial.00401430
0040131A | . C70424 008040 | MOV DWORD PTR SS:[ESP],serial.00408000 | ASCII 0A,"Enter Seri"
00401321 | . E8 54000000 | CALL serial.0040137A
00401326 | . 8D45 D8 | LEA EAX,DWORD PTR SS:[EBP-28]
00401329 | . 894424 04 | MOV DWORD PTR SS:[ESP+4],EAX
0040132D | . C70424 168040 | MOV DWORD PTR SS:[ESP],serial.00408016 | ASCII "%s"
00401334 | . E8 7F500000 | CALL <JMP.&msvrt scanf> | scanf
00401339 | . 8D45 D8 | LEA EAX,DWORD PTR SS:[EBP-28]
0040133C | . C74424 08 0800 | MOV DWORD PTR SS:[ESP+8],8
00401344 | . C74424 04 1980 | MOV DWORD PTR SS:[ESP+4],serial.0040801F | ASCII "DEADBEEF"
0040134C | . 890424 | MOV DWORD PTR SS:[ESP],EAX
0040134F | . E8 5C500000 | CALL <JMP.&msvrt strcmp> | strcmp
00401354 | . 85C0 | TEST EBX,EBX
00401356 | . EB 0E | JMP SHORT serial.00401366
00401358 | . C70424 248040 | MOV DWORD PTR SS:[ESP],serial.00408024 | ASCII "Error! Incorrect serial number"
0040135F | . E8 16000000 | CALL serial.0040137A
00401364 | . EB 0C | JMP SHORT serial.00401372
00401366 | . C70424 508040 | MOV DWORD PTR SS:[ESP],serial.00408050 | ASCII "Serial number is correct."
0040136D | . E8 08000000 | CALL serial.0040137A

```

Figure 9. The second solution

Two different search algorithms are used to find the path: Depth-First and Breadth-First. For this plug-in, the result shows the expected executing sequence if the path between two points is available. In this case, the two algorithms provide the same path for the solution, which is the same as described above. However, the process of finding the path is quite different. Table 1 shows a comparison of two algorithms in terms of code line examined in assembly code, or more formally, the number of instructions examined. Both of them start searching from the same address 0040131A and end at the same address 00401366. In addition, they provide the same path from the start point to the end point, and that the length of the executing sequence is 14 lines of instructions in assembly. However, Depth-First algorithm has to visit 365 lines of instruction before it can find the path, in contrast to 80 lines of instructions for Breadth-First algorithm. In Figure 10, the

vertical scroll bar in the screen shots shows the difference visually. This case is an example of those cases in which the paths can be found.

Table1: Result for Case One

	Start	End	Instructions in Path	Visited instructions
Depth-First	0040131A	00401366	14 lines	365 lines
Breadth-First	0040131A	00401366	14 lines	80 lines

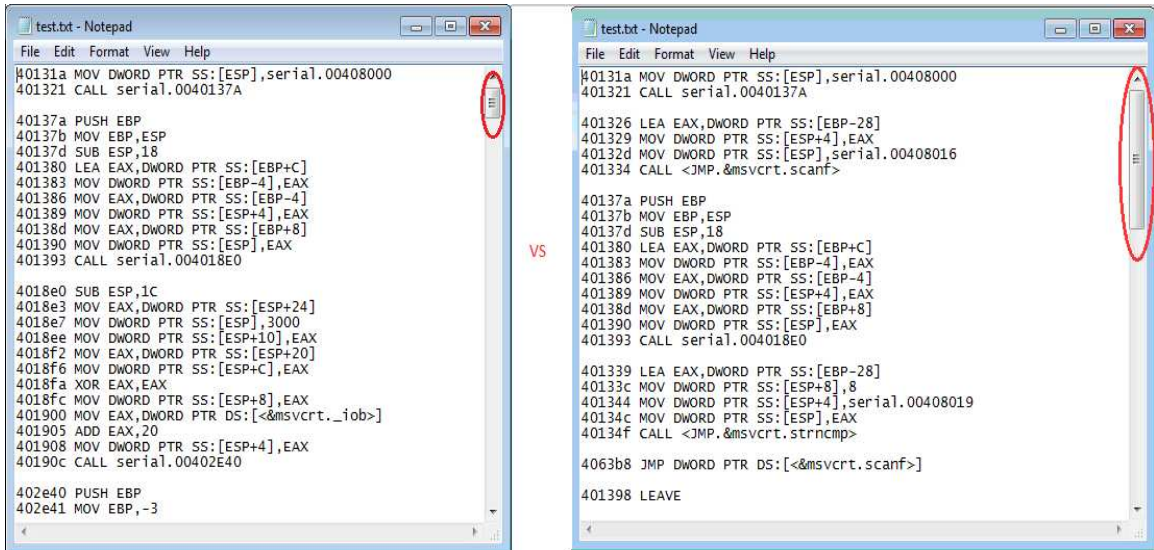


Figure 10. Screenshot for visited instructions file

How about setting the destination to 00401358 so that, no matter what serial number is input, it always jumps to the address showing a message of “serial number is incorrect,” even if the serial number is correct? As with the previous problem, we have two solu-

tions. One is to change the condition which will influence flag bit, another is to change the jump instruction to unconditional. Since the plug-in is implemented in the second one, all the cases below show the solution for the implementation. As the Figure 11 shows, the instruction must execute to 00401358.

```

00401310 . E8 9B050000 CALL serial.004018B0
00401315 . E8 16010000 CALL serial.00401430
0040131A > C70424 008040 MOV DWORD PTR SS:[ESP],serial.00408000 ASCII 0A,"Enter Seri"
00401321 . E8 54000000 CALL serial.0040137A
00401326 . 8D45 08 LEA EAX,DWORD PTR SS:[EBP-28]
00401329 . 894424 04 MOV DWORD PTR SS:[ESP+4],EAX
0040132D > C70424 168040 MOV DWORD PTR SS:[ESP],serial.00408016 ASCII "%s"
00401334 . E8 7F500000 CALL <JMP.&msvcrt.scanf> scanf
00401339 . 8D45 08 LEA EAX,DWORD PTR SS:[EBP-28]
0040133C . C74424 08 0801 MOV DWORD PTR SS:[ESP+8],8
00401344 . C74424 04 1981 MOV DWORD PTR SS:[ESP+4],serial.0040801 ASCII "DEADBEEF"
0040134C . 890424 MOV DWORD PTR SS:[ESP],EAX
0040134F . E8 5C500000 CALL <JMP.&msvcrt.strncmp> strncmp
00401354 . 85C0 TEST EAX,EAX
00401356 > EB 00 JMP SHORT serial.00401358
00401358 > C70424 248040 MOV DWORD PTR SS:[ESP],serial.00408024 ASCII "Error! Incorrect serial numb
0040135F . E8 16000000 CALL serial.0040137A
00401364 > EB 0C JMP SHORT serial.00401372
00401366 > C70424 508040 MOV DWORD PTR SS:[ESP],serial.00408050 ASCII "Serial number is correct.
0040136D . E8 08000000 CALL serial.0040137A

```

Figure 11. Screenshot for case one part two

Case Two:

This test case is similar to the previous one, and will be tested using the two algorithms. However, the efficiency of the two algorithms is quite different from the previous example. The Figure 12 shows the start point and the end point.

```

004046E2 . 898424 900000 MOV DWORD PTR SS:[ESP+90],EAX
004046E9 > 8B7424 74 MOV ESI,DWORD PTR SS:[ESP+74]
004046FD . 0000 ADD BYTE PTR DS:[EAX],0
004046FF > 7E 18 JLE SHORT serial.00404709
004046F1 . 8B4C24 74 MOV ECX,DWORD PTR SS:[ESP+74]
004046F5 . 8B4424 30 MOV EAX,DWORD PTR SS:[ESP+30]
004046F9 . 894C24 04 MOV DWORD PTR SS:[ESP+4],ECX
004046FD . 890424 MOV DWORD PTR SS:[ESP],EAX
00404700 . E8 0B160000 CALL serial.00405D10
00404705 . 894424 30 MOV DWORD PTR SS:[ESP+30],EAX
00404709 > 8B4C24 38 MOV ECX,DWORD PTR SS:[ESP+38]
0040470D . 0000 ADD BYTE PTR DS:[EAX],AL
0040470F > 7E 18 JLE SHORT serial.00404729
00404711 . 8B5424 38 MOV EDX,DWORD PTR SS:[ESP+38]
00404715 . 8B4C24 24 MOV ECX,DWORD PTR SS:[ESP+24]
00404719 . 895424 04 MOV DWORD PTR SS:[ESP+4],EDX
0040471D . 890C24 MOV DWORD PTR SS:[ESP],ECX
00404720 . E8 EB150000 CALL serial.00405D10
00404725 . 894424 24 MOV DWORD PTR SS:[ESP+24],EAX
00404729 > 8B5424 48 MOV EDX,DWORD PTR SS:[ESP+48]
0040472D . 85D2 TEST EDX,EDX
0040472F > 0F85 CEB50000 JNZ serial.00404D03

```

Figure 12. Screenshot for case two

For this case, the start point is at 004046ED and the end point is at 0040472D. The path between two points could be found very quickly by both algorithms, the actual path is 004046ED->004046EF->00404709->0040470D->0040470F->00404729->0040472D, which can be seen in Figure 13.

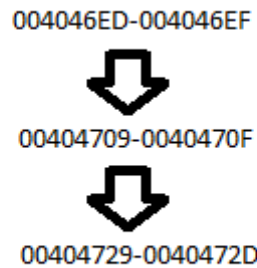


Figure 13. Path for case two

However, the visited number of lines for the finding process is quite different from the previous case.

Table2: Result for case two

	Start	End	Instruction in path	Visited instructions
Depth-First	004046ED	0040472D	6 lines	9 lines
Breadth-First	004046ED	0040472D	6 lines	20 lines

It's a little abnormal. Why is the visited length for Depth-First algorithm so much longer than Breadth-First algorithm in the first case while the Depth-First algorithm has a much

shorter visited length than the Breadth-First algorithm in this case? After closely examination, we found that the jumps occur ahead of the destination, never beyond it. This is why the path between the two points is getting shorter. However, for the last case, the jumps go outside the function to call another sub function, which is a waste of time, since the destination is in the main function in this case. So, normally, if the jumps don't exceed the address of the destination and don't call other sub functions, Depth-First algorithm could have more efficiency than Breadth-First algorithm. But on the average, Breadth-First algorithm is more efficient than Depth-First.

Case Three:

Unlike the previous to cases, we are going to demonstrate an example in which the path could not be found. Figure 14 shows an example of that:

00401590	. 55	PUSH EBP	
00401591	. 89E5	MOV EBP,ESP	
00401593	. 83EC 08	SUB ESP,8	
00401596	. A1 08704000	MOV EAX,DWORD PTR DS:[407008]	
00401598	. 8338 00	CMP DWORD PTR DS:[EAX],0	
0040159E	. 74 17	JE SHORT serial2.004015B7	
004015A0	> FF10	CALL DWORD PTR DS:[EAX]	
004015A2	. 8B15 08704000	MOV EDX,DWORD PTR DS:[407008]	
004015A8	. 8D42 04	LEA EAX,DWORD PTR DS:[EDX+4]	
004015AB	. 8B52 04	MOV EDX,DWORD PTR DS:[EDX+4]	
004015AE	. A3 08704000	MOV DWORD PTR DS:[407008],EAX	
004015B3	. 85D2	TEST EDX,EDX	
004015B5	. 75 E9	JNZ SHORT serial2.004015A0	
004015B7	> C9	LEAVE	
004015B8	. C3	RETN	
004015B9	. 8DB426 000000	LEA ESI,DWORD PTR DS:[ESI]	serial2.0040698C
004015C0	> 55	PUSH EBP	
004015C1	. 89E5	MOV EBP,ESP	

Figure 14. Screenshot for case three

In this code, the start point is selected at address of 00401590 and the end point is chosen at address of 004015B9. The messages box of the unsuccessful path searching result shows up in the OllyDbg as shown in Figure 15.

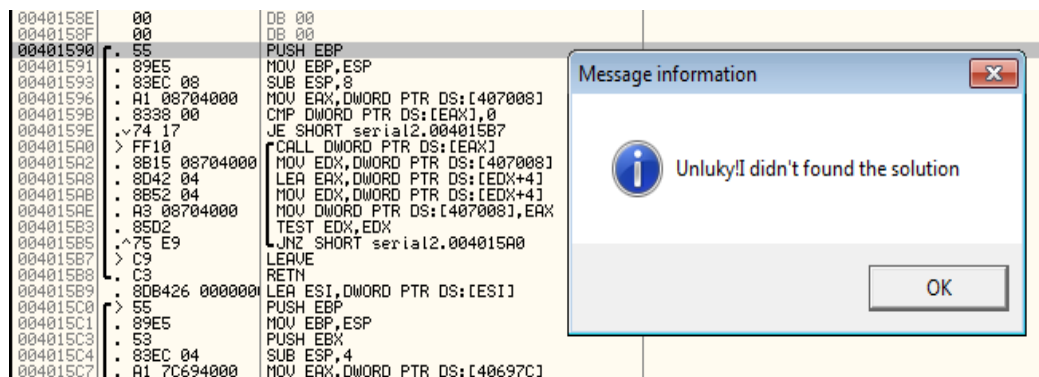


Figure 15. Screenshot of the unsuccessful path searching result

The execution starts to execute at 00401590 PUSH EBP, and goes down until it encounters conditional jump instruction JE 004015B7 at 0040159E. There are two optional sub paths for the execution. One is jump to 004015B7 and another is to execute the fellow instruction. For the Depth-First algorithm, it will jump to 004015B7 then meet the return instruction at 004015B8. The return instruction indicates the end of the searching path, since we do not know where the instruction will return to. Then the algorithm will go back to the jump point 0040159E to search the path from the following instructions. In this case, the next instruction is CALL EAX, in which the jump address is stored in the register and the value is available only at run time. We cannot predict the jump address by static analysis. But we can expect it will return back to the original place after call instruction. And then go down to the follow instructions. It jumps back to 004015A0 when it hit the JNZ 004015A0 instruction. It's a loop for this code section. We have to treat this situation as a dead end since it is running in an endless loop. Since two possible paths in this case have both met dead ends, we conclude that there is no solution for this case. This situation will occur in the Breadth-First algorithm as well.

Generally, there are two types of endings for the searching path. One is in the form of the RETURN instruction, the executing will definitely return to the calling function and then go down to the next instruction. Another one is because the parameter for the jump instruction is not available during static analysis. Just like the example in this case, jump to an address which is stored in a register. The reason why it cannot find a path is a limitation of the software (cannot read register data). It may find a path when the program is running.

Case Four:

The destination point is not always below the start point, or has a larger address than the start point. This plug-in can find a path whose start address is larger than its end address as long as it has a path. Figure 16 shows an example of that situation.

```

004017B8 . BA 1F000000 MOV EDX,1F
004017BD . 8943 4C     MOV DWORD PTR DS:[EBX+4C],EAX
004017C0 > 89D8       MOV EAX,EBX
004017C2 . 21C8       AND EAX,ECX
004017C4 . 89F8 01     CMP EAX,1
004017C7 . 19C0       SBB EAX,EAX
004017C9 . 24 20       AND AL,20
004017CB . 01C9       ADD ECX,ECX
004017CD . 04 41       ADD AL,41
004017CF . 88842A 48FFFF MOV BYTE PTR DS:[EDX+EBP-88],AL
004017D5 . 4A         DEC EDX
004017D7 . ^79 E7     JNS SHORT serial.004017C0
004017D9 . A1 6C804000 MOV EAX,DWORD PTR DS:[40806C]
004017DE . 8985 68FFFFFF MOV DWORD PTR SS:[EBP-98],EAX

```

Figure 16. Screenshot for case four

In this piece of assembly code, users set the start point at 004017CF and the end point at 004017C2. The destination has a smaller address than the start point. The instruction will run from 004017CF and go down until the jump instruction JNS 004017C0 at 00401707. At that time it has two options, one is go down and another is jump to 004017C0. After

tracing the two sub paths, the plug-in can find the destination is in the jump path. So the plug-in will patch the file and get the result, as shown in Figure 17.

```

004017B0 | . 8943 4C | MOV DWORD PTR DS:[EBX+4C],EAX
004017C0 | > 89D8 | MOV EAX,EBX
004017C2 | . 21C8 | AND EAX,ECX
004017C4 | . 83F8 01 | CMP EAX,1
004017C7 | . 19C0 | SBB EAX,EAX
004017C9 | . 24 20 | AND AL,20
004017CB | . 01C9 | ADD ECX,ECX
004017CD | . 04 41 | ADD AL,41
004017CF | . 88842A 48FFFF | MOV BYTE PTR DS:[ECX+EBP-88],AL
004017D6 | . 4A | DEC EDI
004017D7 | . ^EB E7 | JMP SHORT serial.004017C0
004017D9 | . A1 6C804000 | MOV EAX,DWORD PTR DS:[40806C]

```

Figure 17. Snapshot for case four

At this moment, the execution will jump to 004017C0 when it hits the JMP instruction, and go down and reach the destination address of 004017C2.

Case Five:

This case shows an example of changing multiple conditional jumps to unconditional jump to patch the executable. Figure 18 shows a snapshot of the code.

```

00401926 | . 8B52 04 | MOV EDI,DWORD PTR DS:[EDI+4]
00401929 | . 89C1 | MOV ECX,EAX
0040192B | . F6C6 20 | TEST DH,20
0040192E | . ^v75 08 | JNZ SHORT serial.00401938
00401930 | . 8B43 18 | MOV EAX,DWORD PTR DS:[EBX+18]
00401933 | . 3943 1C | CMP DWORD PTR DS:[EBX+1C],EAX
00401936 | . ^v7E 10 | JLE SHORT serial.00401948
00401938 | > F6C6 10 | TEST DH,10
0040193B | . ^v75 14 | JNZ SHORT serial.00401951
0040193D | . 8B13 | MOV EDI,DWORD PTR DS:[EBX]
0040193F | . 8B43 18 | MOV EAX,DWORD PTR DS:[EBX+18]
00401942 | . 8B0C02 | MOV BYTE PTR DS:[EDI+EAX],CL
00401945 | > 8B43 18 | MOV EAX,DWORD PTR DS:[EBX+18]
00401948 | > 40 | INC EAX
00401949 | . 8943 18 | MOV DWORD PTR DS:[EBX+18],EAX
0040194C | . 83C4 08 | ADD ESP,8
0040194F | . 5B | POP EBX
00401950 | . C3 | RETN
00401951 | > 890C24 | MOV DWORD PTR SS:[ESP],ECX
00401954 | . 8B03 | MOV EAX,DWORD PTR DS:[EBX]
00401956 | . 894424 04 | MOV DWORD PTR SS:[ESP+4],EAX

```

Figure 18. Snapshot for case five

The code will start at 00401929 and the destination is 00401954. There are more than one jump instructions between the two points. The program will get the path by changing two

conditional jumps at 0040192E and 00401938. Figure 19 shows the snapshot of the patched code in the picture below.

```

00401920 .: 8B52 04      MOV EAX,DWORD PTR DS:[EDX+4]
00401929 .: 89C1        MOV ECX,EAX
0040192B .: F6C6 20     TEST DL,20
0040192E .: EB 08       JMP SHORT serial.00401938
00401930 .: 8B43 18     MOV EAX,DWORD PTR DS:[EBX+18]
00401933 .: 3943 1C     CMP DWORD PTR DS:[EBX+1C],EAX
00401936 .: 7E 10       JLE SHORT serial.00401948
00401938 .: F6C6 10     TEST DL,10
0040193B .: EB 14       JMP SHORT serial.00401951
0040193D .: 8B13        MOV EAX,DWORD PTR DS:[EBX]
0040193F .: 8B43 18     MOV EAX,DWORD PTR DS:[EBX+18]
00401942 .: 880C02     MOV BYTE PTR DS:[EDX+EAX],CL
00401945 .: 8B43 18     MOV EAX,DWORD PTR DS:[EBX+18]
00401948 .: 40         INC EAX
00401949 .: 8943 18     MOV DWORD PTR DS:[EBX+18],EAX
0040194C .: 83C4 08     ADD ESP,8
0040194F .: 5B         POP EBX
00401950 .: C3         RETN
00401951 .: 890C24     MOV DWORD PTR SS:[ESP],ECX
00401954 .: 8B03        MOV EAX,DWORD PTR DS:[EBX]
00401956 .: 894424 04   MOV DWORD PTR SS:[ESP+4],EAX

```

Figure 19. Result for case five

4.4 Statistics for Testing Cases

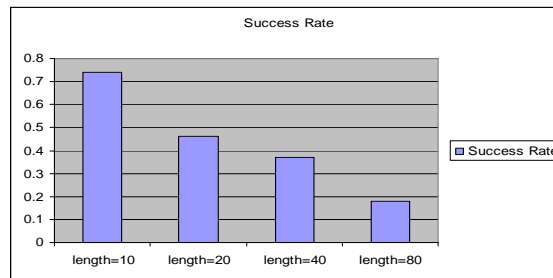


Figure 20. Testing cases statistics by Depth-First

All the test cases in this part are testing on the application of Firefox 3.6 which can be downloaded from here [http://www.mozilla.com/en-](http://www.mozilla.com/en-US/products/download.html?product=firefox-3.6.13&os=win&lang=en-US)

US/products/download.html?product=firefox-3.6.13&os=win&lang=en-US. The statistics are calculated by depth-first algorithm according to the distances between the start and end points which are randomly selected from Firefox. We have conducted the tests

with distances of 10, 20, 40 and 80. For each distance length, we conducted 100 test cases to generate the statistics of success rates. Figure 20 shows our experiment results. For the length=10, the rate of finding a path successfully is 0.74, but it drops to 0.46 for length=20, 0.37 for length=40, and 0.18 for length=80 respectively. It is obviously that the success rate decreases when the length between two points is getting longer. This is because the relevance between the two functions to which the start and end points belong decreases when the distance between these two points increases. For example, when these two points are close, there is a good chance that they are within a same function, and there is a path connects them. However, the distance between these two points is long, it is more likely that they belong to two unrelated functions, and therefore no path can be found.

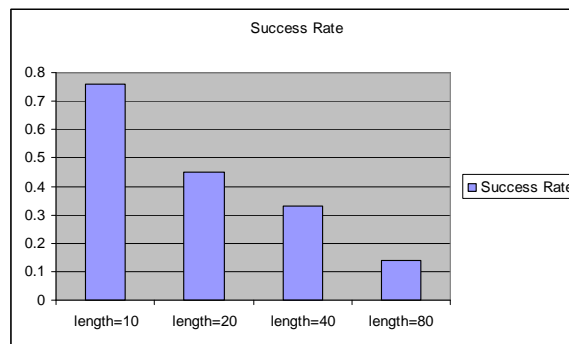


Figure 21. Testing cases statistics by Breadth-First

The figure 21 shows the testing cases statistics for Breadth-First algorithm. As the same process as the first one, we conduct the statistics by calculating the success rate of finding path between two points which are randomly selected from Firefox. The tests are conducted with distances of 10, 20, 40 and 80. Finally, we get the success rate 0.75 for length

=10, 0.45 for length = 20, 0.34 for length = 40, 0.15 for length=80. The conclusion is the same as the first statistics that the success rate decreases when the distance increases for the two points.

Additionally, we manually find 10 test cases which have path between two points for each different length of test cases (length=10, 20, 40, 80). Then we use this tool to find the path for these test cases automatically. From the experimental results, we found that all the paths that have been found manually are found by this tool. These results indicate that the cases for which the tool fails (see Figure 20 and 21) are due to the fact that no path exists, rather than a failure of the tool to find a possible path.

4.5 Optimization

There are two solutions to patch the code. One is to change the condition to true and make sure the conditional jump will execute if it needs to jump. The other is to change the conditional jump to unconditional jump. For the first one, patching code can be divided into several steps. First, get the conditional jump instruction from the assembly code and figure out whether it needs to jump. Second, determine which operation will change this condition to true. Finally, patch the condition and make it jump, if possible. For the second one, we only need to change the conditional jump to unconditional jump, JMP. It could jump to the original destination if the jump is required, otherwise just jump to the next instruction. In the plug-in implementation, the second solution is used.

5.0 Future works

5.1 Improve The Algorithm for Tree Traversal

In the Breadth-First algorithm, I used an array to store the tree structure. It is not very efficient to implement it by array, and a lot of memory space will be wasted if it is not a balanced tree. An extreme example is shown in the picture below.

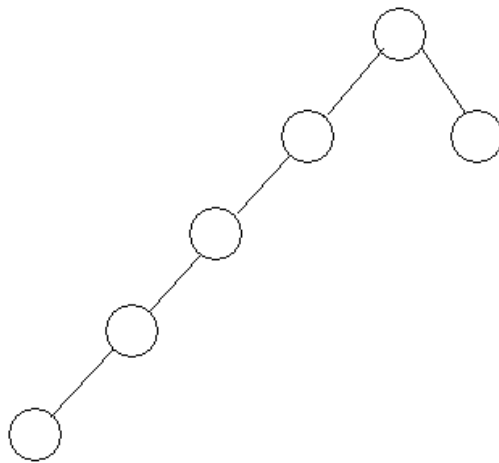


Figure 22. Extreme case for binary tree

For this case, $2^5 - 1 = 31$ memory space is needed to store the whole tree even though there are only six nodes in it. It is a tremendous waste of memory. One possible solution is to use linked list instead of array, which will save all the memory space that is not used but is occupied in an array. For linked list, there are three more fields that are used to point to the parent and child nodes. Combined with the data field, the linked list has four fields for each node. It should use $4 * 6 = 24$ memory space in total. We should find that much space is saved if the case has more layers.

5.2 Optimize Depth-First Search Algorithm

The Depth-First algorithm will jump if the jump instruction is met. Normally, the main function will call a sub function and repeat call the other sub function, but in most cases, the destination is in the main function. A lot of unnecessary work is done to find the path. We should find a way to judge whether the destination is in the main function or in the sub function, and then the algorithm can be optimized. We should not jump to the sub function if the destination is in the main function, otherwise, go to the sub function.

5.3 Analyze the Functionality

In the implementation, we find a path between two points and change the jump instructions without considering the skipped functions. In some cases this is okay, but in others, this can be a problem, since the skipped function will influence the following function. For example, a division function will first check if the divisor is equal to zero. If not, then continue calculating and get the result. But the function will cause a fault if we skip this check function and directly jump to the calculation function. So the algorithm for finding the path should be improved. A possible solution is to find all the paths for the two points and figure out which one requires less change and can therefore be closest to the real path.

6.0 Conclusion

The field of reverse engineering is wide, and is a topic of great interest in Computer Science. With the increasing requirements the industry, Reverse Engineering will become much more mature in the future. This project explores a small piece of this field, but it shows that developers can change a feature of the function even when the source code is

not available. The Depth-First algorithm and Breadth-First algorithm have advantages and disadvantages. Each will achieve better path coverage than the other in some cases. In general, in this project, the Breadth-First algorithm has shown better performance than the Depth-First algorithm. Developing a plug-in is a good idea to implement a function, since developers can use the existing functions and simply add new features to them. In this case, we still need to decompile the binary code to assembly code if the OllyDbg is not available.

Compared to creating a new program, developing a plug-in will take a lot of time to set up the developing environment and get familiar with its principles. It is a very good experience for developers, and we can learn a lot from the exercise, including how to provide the API and documentation.

Reference

1. Hausi A.Muller (2000). “Reverse engineering: a road map”. *Proceedings of the conference on The Future of Software Engineering*.
2. G. Antoniol (2001.) “Object-orient design patterns recovery”. *Journal of Systems and Software*, 59(2):181-196, 2001
3. C. Cifuentes and K.J. Gough. “Decompilation of binary programs”. *Softw., Pract.Exper.*, 25(7):811-829, 1995
4. R. Koschke. “Atomic Architectural Component Recovery for Program Understanding and Evolution”.. *PhD thesis, Univ. of Stuttgart, Germany, 2000*
5. Muhammad R.(2005). “Why Teach Reverse Engineering”. *Proceedings Proceedings of the 2005 ACM*.
6. P.H.Aiken. “Reverse engineering of data”. *IMB Corp*.
7. Eleni Stroulia (2002). “Dynamic analysis for reverse engineering and program understanding”. *SIGAPP Applied Computing Review*.
8. Reverse engineering tools descriptions http://reversingproject.info/?page_id=153
9. OllyDbg office website <http://www.ollydbg.de/>
10. Depth-First Algorithm http://en.wikipedia.org/wiki/Depth-first_search
11. Breadth-First Algorithm http://en.wikipedia.org/wiki/Breadth-first_search

12. Gerardo CanforaHarman (2007). "New Frontiers of Reverse Engineering". 2007 *Future of Software Engineering*.
13. Eldad Eilam (2005). "Reversing: secrets of Reverse Engineering". *Published by Wiley*.
14. T.Systa (1999). "On the relationship between static and dynamic models in reverse engineering java software." *In Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE-99), Atlanta, Georgia, USA, pages 304-313. IEEE Computer Society Press, October 1999.*
15. Ahmad K. Ghafarian. "Reverse engineering technique to enhance software engineering education." *ACM*