UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

DIPLOMA THESIS no. 60

Using optimization algorithms for malware deobfuscation

Branko Spasojević

Mentor: doc. dr. sc. Marin Golub

Zagreb, Croatia
June 2010.

To: Sanja, Ana, Megi

Abstract

Using optimization algorithms for malware deobfuscation

Analysis of malware binaries is constantly becoming more difficult with introduction of many different types of code obfuscators. One common theme in all obfuscators is transformation of code into a complex representation. This process can be viewed as inverse of compiler optimization techniques and as such can be partially removed using optimization algorithms. This paper presents common obfuscation techniques and a process of adapting optimization algorithms for removing obfuscations. Additionally, a plug-in for the IDA Pro disassembler is presented that demonstrates usability of the proposed optimization process as well as a set of techniques to speed up the process of analyzing obfuscated code.

Key words: deobfuscation, optimization, assembly, malware, binary, compiler

Sažetak

Pojednostavljenje izvršnog teksta zlonamjernih programa korištenjem optimizacijskih algoritama

Analiza izvršnog teksta zlonamjernih programa (virusi, crvi) je vrlo često namjerno otežana posebnim tehnikama skrivanja funkcionalnosti programa. Izvršni tekst se mijenja tako da ga je teško analizirati i razumjeti. Maskiranje stvarnog koda ubacivanjem instrukcija koje ne mijenjaju stanje programa, zamjena jednostavnih instrukcija kombinacijama složenijih instrukcija, povećanje broja skokova, itd. samo su neke od tehnika zaštite koda od analize. Jedan od načina kojim je moguće poboljšati izgled i povećati razumijevanje koda je korištenje optimizacijskih algoritama. Uklanjanje nepotrebnih instrukcija, smanjenje razgranatost i veličina programa samo su neke od posljedica primjene optimizacije. Svi primjeri rađeni su na Intel x86 platformi, ali su ideje primjenjive i na ostale tipove procesora.

Ključne riječi: optimizacija, asembler, zlonamjerni program, kompajler

Contents

Acknowledgements

First of all I would like to thank my mentor Marin Golub for allowing me to pursue my research interests and for providing guidance during the past several years. Special thanks go to everyone in INFIGO IS (alphabetical order): Bojan Ždrnja, Hrvoje Šegudović, Ivana Marijanović and Saša Jušić, for providin IDA Pro licence, enjoyable working environment and to Bojan for all morning RCE discussions. Next I would like to thank Matt Jonkman from *Emergingthreats.net* for sharing and uploading enormous quantities of malware samples for testing. Last but not least I would like to thank Domagoj Klasić for discussions and collaboration on exploring new malware analysis ideas.

1. Introduction

Motivated by profits and huge market potential, many software developers turn to the underground economy of the Internet. One of well known niches of illegal activity is developing and spreading of malware. Malware, short for malicious software, is term a used to refer to all kinds of hostile, intrusive and usually unwanted software. There are many categories of malware, and most of them are based on specific purposes of such malicious programs. Some of these categories include: viruses, worms, trojan horses, spyware, adware, rootkits and many more.

Malware exploits the black box property of today's computer software. For most computer users, a piece of software is just a black box they hope works as advertised. There aren't any methods that could prove that a piece of software we installed on our computer behaves as advertised and will not affect the state of operating system in an undesirable way (although there exists a discipline of formal verification to make it possible to prove or disprove the correctness of intended algorithms with respect to a certain formal specification). Malware authors exploit this property to build a variety of malicious software. To prevent execution of malware, anti-virus (AV) vendors use pattern matching and heuristics to identify malware samples and terminate their execution before any undesirable action takes place.

Most AV vendors protect customers by creating patterns based on examined malware samples. This approach is vulnerable to so called 0-day malware samples. 0-day is an expression used for samples that are new and previously unknown so there does not exist a pattern that could find (match) them. Microsoft reports detecting half a million unique malicious files every day [1] and although most of them are identical in terms of malicious activity it is still a huge number of files to analyze.

Most malware samples are easy to analyze and remove due to their simplicity, but there is emerging trend of complex and novel techniques used for spreading, infection and obfuscation of malware code. For that kind of malware it is not enough to detect and remove it from the host (infected) computer, but there also exists a need to understand its internal workings. Deeper understanding of malware enables researchers to efficiently stop spreading of infections and to optimize detection algorithms.

The most common mechanism used by malware authors to deter AV researchers is packing. Packers are first stage protectors. They are code wrappers that compress or encrypt original malware code inside themselves. During runtime they unpack the original malicious code and execute it. After unpacking, the researcher has the original code available for analysis. Because packers are just wrappers it is possible to automate unpacking process and save (dump) the original malware code in a new executable, effectively removing the wrapper from gift (malware). Many generic unpackers are available for unpacking majority of popular packers. Also, there are unpacking frameworks such as TitanEngine [3] that provide facilities for writing custom unpackers and thus simplifying the whole process.

After unpacking a malware sample, we are presented with the original malware code. Because unpacking is just first stage protection and the process of

unpacking can be automated, malware authors often add additional layers of protection. These layers consist of code transformations that obfuscate the original code. The idea behind this approach is that when AV researchers understand internal workings of malware it means game over for malware authors because AV software will be able to recognize malicious behavior either on static or dynamic malware traits.

Obfuscation process can be viewed as inverse of compiler optimization techniques and as such can be partially removed using optimization algorithms. This thesis presents common obfuscation techniques and a process of adapting optimization algorithms for removing obfuscations. Additionally, a plug-in for the IDA Pro [2] disassembler is presented that demonstrates usability of proposed optimization process thus allowing researchers to analyze malicious code faster and understand it better.

Although we concentrate on Intel x86 platform most of information applies to all processor platforms, such as CISC or RISC.

2. Malware Analysis

Methodology of malware analysis depends on researcher's goal. He is either interested in overall actions of a malware sample or in deep understanding of internal workings. There are many ways to reach the same goal, but some are more common and faster than others. Dynamic analysis gives information about execution and interaction of malware with its environment (operating system, file system, etc.). Static analysis involves reading code disassembly and understanding malware author's intentions. Each approach has its advantages and disadvantages but only when used interchangeably they provide the full picture of malware actions.

2.1. Dynamic Analysis

As the name suggests dynamic analysis is based on monitoring code while executing it. This way a researcher is able to observe malicious activity as it takes place. This kind of analysis is usually done inside virtualized environments such as VmWare [4], VirtualBox [5] or sandboxes like CWSandbox [6], Joebox [7].

Sandbox environments are simpler to use because they monitor and report analysis results collected from malware execution traces. They typically report all file system and registry changes, network activity, loaded DLL files and called system functions. This kind of report gives a researcher an overview of malware workings from which he can conclude malware type and categorize it.

Virtualized environments are used to separate malware environment from the working environment of a researcher. This separation limits infection exposure to the virtual operating system environment and enables the researcher to study infection process and malware workings. Running malware inside such an environment enables the researcher to use his preferred tools and to customize monitoring and logging capabilities.

The main tool for dynamic analysis of executables is a debugger. A debugger or debugging tool is a program that is used to test and debug other programs. The debugger provides capabilities to control execution of another executable file and examine its execution environment. The execution environment includes the executable image in memory, mapped memory address space, processor registers and additional loaded libraries. As the debugger is intermediary between malware code and physical processor, the researcher can examine code before it is actually executed thus enabling him to log all executed instructions and inspect malware behavior. Some popular debuggers for analyzing malware are: OllyDbg [8], ImmunityDbg [9] and WinDbg [10].

2.2. Static Analysis

Static analysis is a technique of code analysis where a researcher examines malware code without executing it. Usually this process involves disassembling executable code and reading the disassembled listing. Static analysis is immune to unintentional infection that can result from running a malware sample. Static analysis also assumes more engagement from the researcher to understand malware workings and greatly depends on implemented obfuscation measures.

Static analysis is more demanding because the researcher must assume what parts of code are executed and in what order as he does not have an execution trace like with dynamic analysis. Benefits of static analysis are that the researcher familiarizes himself with internal workings of the sample and is thus able to better understand potential dangers of infection and also write better detection extensions.

One of the most popular static analysis tools is IDA Pro. IDA has a disassembly engine that is capable of disassembling many processor's executables and bytecode files. Besides converting processor opcodes into meaningful assembly language it also provides an editor for easy navigation, labeling, annotation and graphing capabilities. Another reason for IDA's success is a rich programming API that exposes many internal functionalities of IDA and enables writing powerful extensions. IDA also provides debugging functionality by using external debuggers and emulators like: Gdb, WinDbg and Bochs.

3. Obfuscation

Obfuscation is the concealment of intended meaning of code, making it confusing, intentionally ambiguous and more difficult to interpret. Obfuscation can be done either on source code or executable code.

Below is a list of common obfuscation goals:

- Make code comprehension more difficult.
- Increase analysis time.
- Maximize code size.
- Maximize number of branches.
- Hide malicious purpose.

Source code obfuscation can be used to hide and protect private components of programs written in interpreted languages. Interpreted languages don't always ship with compilers that rewrite code to assembly thus making it more difficult to analyze, so source code obfuscation is the easiest way to enhance privacy of code. The International Obfuscated C Code Contest [11] (IOCCC) is a competition focused on exploring obfuscation possibilities of C language and it demonstrates effectiveness of obfuscation transformations for making source code reading and comprehension difficult. There is another type of source code obfuscation that aims at backdooring and inserting malicious behavior inside regular looking code. The idea is to make the code look as innocent as possible while implementing malicious behavior or adding backdoors. Similar to IOCCC there exist Underhanded C Contest [12] which is based on writing innocent-looking C code implementing malicious behavior inside a program solving a specific task.

Executable code obfuscation aims at transforming assembly code to make analysis using debuggers, disassemblers and decompilers difficult. All of these tools rely on some assumptions about code constructs and executable file structure to extract as much information as possible. Obfuscation techniques change properties of executable code and file structure in such ways that it makes assumptions made by analysis tools false and thus minimizes information that a researcher receives.

Ability to remove changes made by obfuscation process enables the researcher to efficiently use his preferred tools and continue with usual analysis methodology. Obfuscation techniques can be separated in several groups based on purpose of obfuscation and data they modify. Examples of classifying obfuscation techniques based on purpose include:

- Anti-debugging
- Anti-disassembly

Examples of classifying obfuscation techniques based on data they transform include:

- Control-flow obfuscations
- Data-flow obfuscations

Researchers are more interested in transformation techniques used to obfuscate code rather than their classification by purpose. That is because transformations can be more easily removed when grouped by data transformations that generated them.

3.1. Disassembler

A disassembler is a program that translates machine instructions (opcodes) into assembly language. In essence it provides reverse operation of an assembler. Assembly is a textual representation of processor instructions suitable for programmers to read and write programs in it. The difference between assembler and other high-level languages is that a single processor instruction translates to a single assembly instruction, so there is a one-to-one mapping between them. On the other hand, a single high-level expression can evaluate into one or more assembly instructions.

Executable file formats consist of several segments like: .text, .data, .idata, .rdata and other. Each segment has one or more flags that specify its properties like read, write or execute. Although segments that have an executable flag set can contain code it does not mean that everything inside it is code. Except code, such segments can contain variable data, alignment space or encrypted/encoded data that has yet to become code. Being unable to distinguish code from data makes it difficult for a researcher to reason about existence of potentially hidden code segments or functions. One additional problem with analysis of malware on Intel x86 platform is that it uses CISC[1] instructions. This enables malware authors to exploit instruction complexity and complicate the disassembly process. As a result of that complexity there exist several disassembly algorithms that try to solve some of the above mentioned problems.

3.1.1. Linear Sweep Disassembly

The linear sweep algorithm is the simplest disassembly algorithm. It begins by disassembling code from the start address and sweeps addresses in linear order till it reaches the end. Pseudo code in Figure 3.1 illustrates this algorithm.

```
LinearSweep(StartAddr, EndAddr)
  addr = StartAddr
  instructions = List()
  While addr < EndAddr:
    instructions.Append( Decode(addr) )
    addr += Length( instructions.Last() )
  return instructions
```

Figure 3.1 - Linear sweep algorithm

3.1.2. Recursive Traversal Disassembly

Unlike the linear sweep algorithm recursive traversal takes control flow of disassembled code into account. This somewhat solves problems of mixing code and data in the same segment because when it comes across a control flow

[1] Complex instruction set computing

branch it will continue the disassembly following branches and effectively skipping any non instruction data. Pseudo code in Figure 3.2 illustrates this algorithm.

```
RecursiveTraversal(StartAddr)
  todo_list = List()
  done_list = List()
  todo_list.Append(StartAddr)
  instructions = List()
  While Length(todo_list) > 0:
    addr = todo_list.Pop()
    If addr in done_list:
      Continue
    instruction = Decode(addr)
    instructions.Append( instruction )
    If Type(instruction) == "branch":
      For successor in Successors(instruction):
        todo_list.Append(successor)
    Else:
     todo_list.Append( addr + Length(instruction) )
```

Figure 3.2 - Recursive traversal algorithm

3.2. Control-Flow Obfuscations

Control-flow obfuscation is a common name for all code transformations that affect execution flow of a program. This can be caused either by adding complexity trough inserting jumps or by indirectly affecting program flow using techniques such as exception handlers.

Next we will examine some standard control-flow obfuscations. In subsequent sections we will concentrate on removing these obfuscations.

3.2.1. Inserting unconditional jumps

Most disassembly and debugging tools show code as a list of sequential instructions inside a linear memory segment. Figure 3.3 shows a typical disassembly listing of a regular function.

```
.text:00401000 sub_401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 push    esi
.text:00401001                 lea     esi, [eax+eax*2]
.text:00401004                 shl     esi, 2
.text:00401007                 mov     ecx, dword_46AE44[esi]
.text:0040100D                 lea     eax, unk_46AE48[esi]
.text:00401013                 cmp     ecx, [eax]
.text:00401015                 push    edi
.text:00401016                 jl      short loc_401032
.text:00401018                 add     ecx, 20h
.text:0040101B                 push    8               ; int
.text:0040101D                 push    ecx             ; int
.text:0040101E                 lea     edi, unk_46AE40[esi]
.text:00401024                 push    dword ptr [edi] ; void *
.text:00401026                 mov     [eax], ecx
.text:00401028                 call    sub_409F34
```

```
.text:0040102D                  add     esp, 0Ch
.text:00401030                  mov     [edi], eax
```

Figure 3.3 - Linear disassembly listing

This type of listing is similar to regular source code editors and in most cases it is usable as is. Some disassembly tools like IDA Pro have capabilities to show code listing as a graph of basic blocks[2]. This type of visualization as shown in Figure 3.4, has an additional benefit of clustering relevant function code together without depending on address space. As a result this gives a nice overview of the function flow and logical segments.

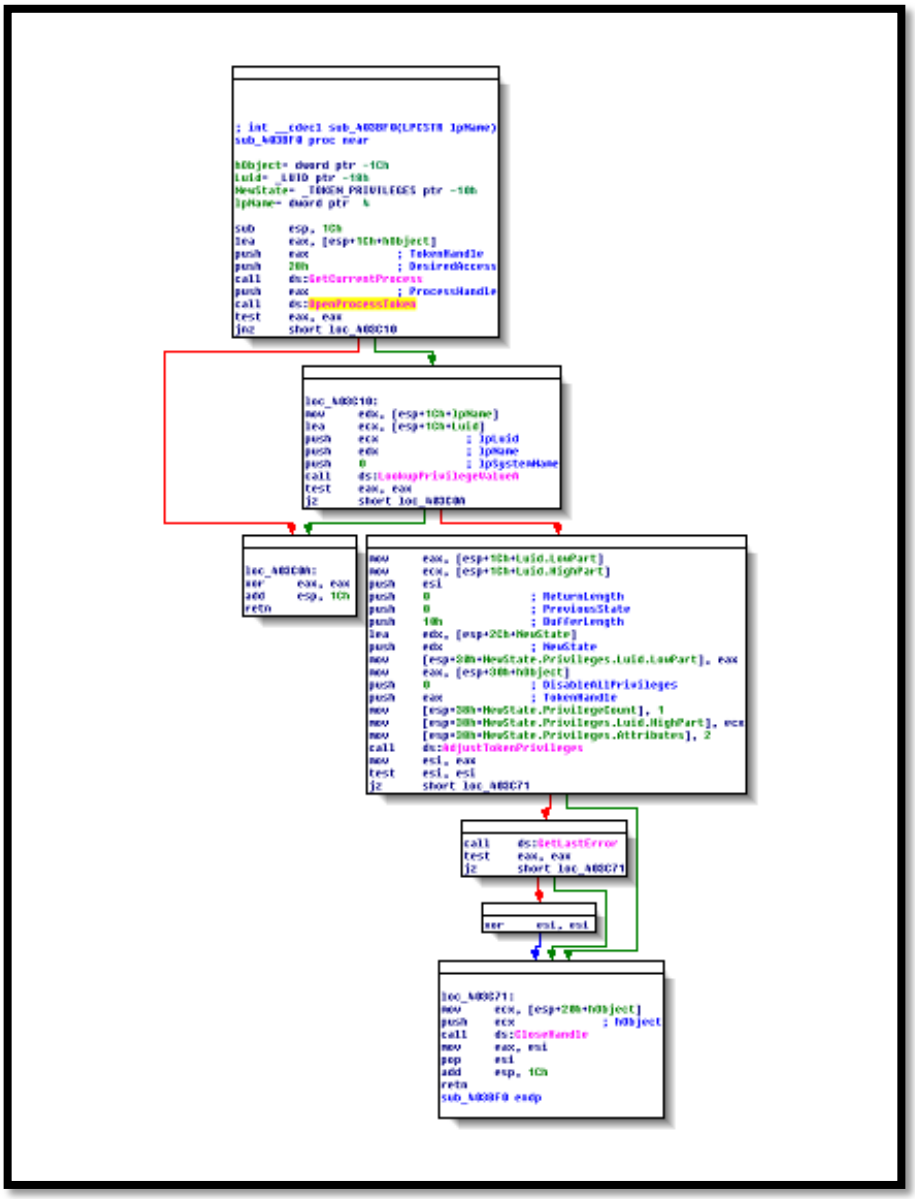

Figure 3.4 – Graph function view

[2] Basic block is code block that has single entry point, one exit point and no jump instructions within it

Insertion of unconditional jumps is an obfuscation method that aims at making linear disassembly listing unusable and graph visualization bloated. Since linear disassembly depends on location of instructions, spreading basic blocks all over the address space results in a code listing that is very difficult to follow. Figure 3.5 describes an idea used to obfuscate linear code display.

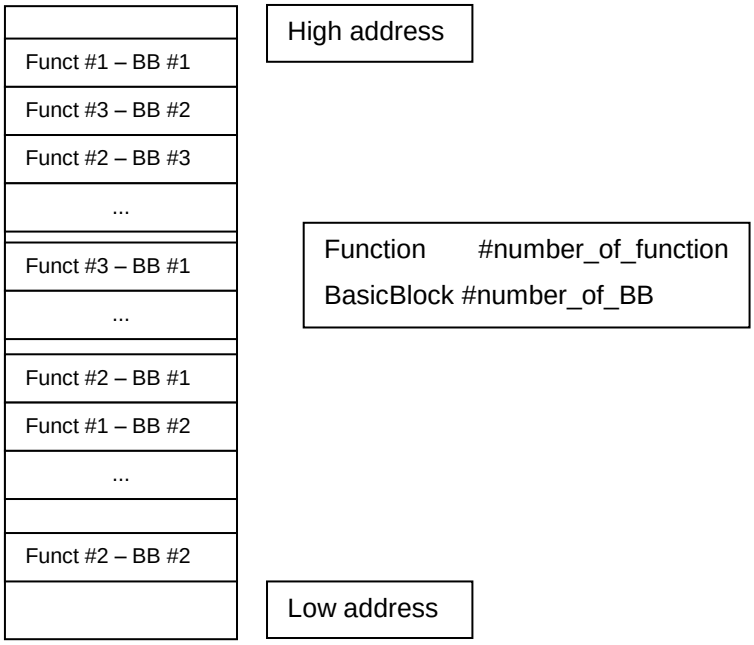

Figure 3.5 – Function separation

Figure 3.6 shows an obfuscated linear disassembly listing. Obfuscation can be noticed from usage of small basic blocks ending with a JMP instruction and using alignment padding between basic blocks.

```
.text:004073B5                  call    _main
.text:004073BA                  add     esp, 14h
.text:004073BD                  jmp     short loc_40738F
.text:004073BF
.text:004073BF loc_4073BF:
.text:004073BF                  mov     ecx, eax
.text:004073C1                  jmp     loc_412870
.text:004073C6                  align 4
.text:004073C8                  dd 5 dup(0)
.text:004073DC                  db 0
.text:004073DD
.text:004073DD loc_4073DD:
.text:004073DD                  jz      short loc_4073A5
.text:004073DF                  push    eax
.text:004073E0                  call    _main
.text:004073E5                  jmp     loc_412860
.text:004073EA                  align 10h
.text:004073F0
.text:004073F0 loc_4073F0:
.text:004073F0                  jz      short loc_407423
.text:004073F2                  push    dword ptr [ebp+8]
.text:004073F5                  call    _main
.text:004073FA                  jmp     loc_412851
```

Figure 3.6 - Obfuscated linear listing

Graph visualization can efficiently gather all basic blocks from memory space and visualize it in one place correctly disregarding obfuscation. To make this type of obfuscation impact graph visualization display it is necessary to increase the number of basic blocks. This can be accomplished by inserting unconditional jumps (the JMP instruction) anywhere within a basic block. This increases CFG complexity and makes it much more difficult to analyze functions. An example of this type of obfuscation is illustrated in Figure 3.7.

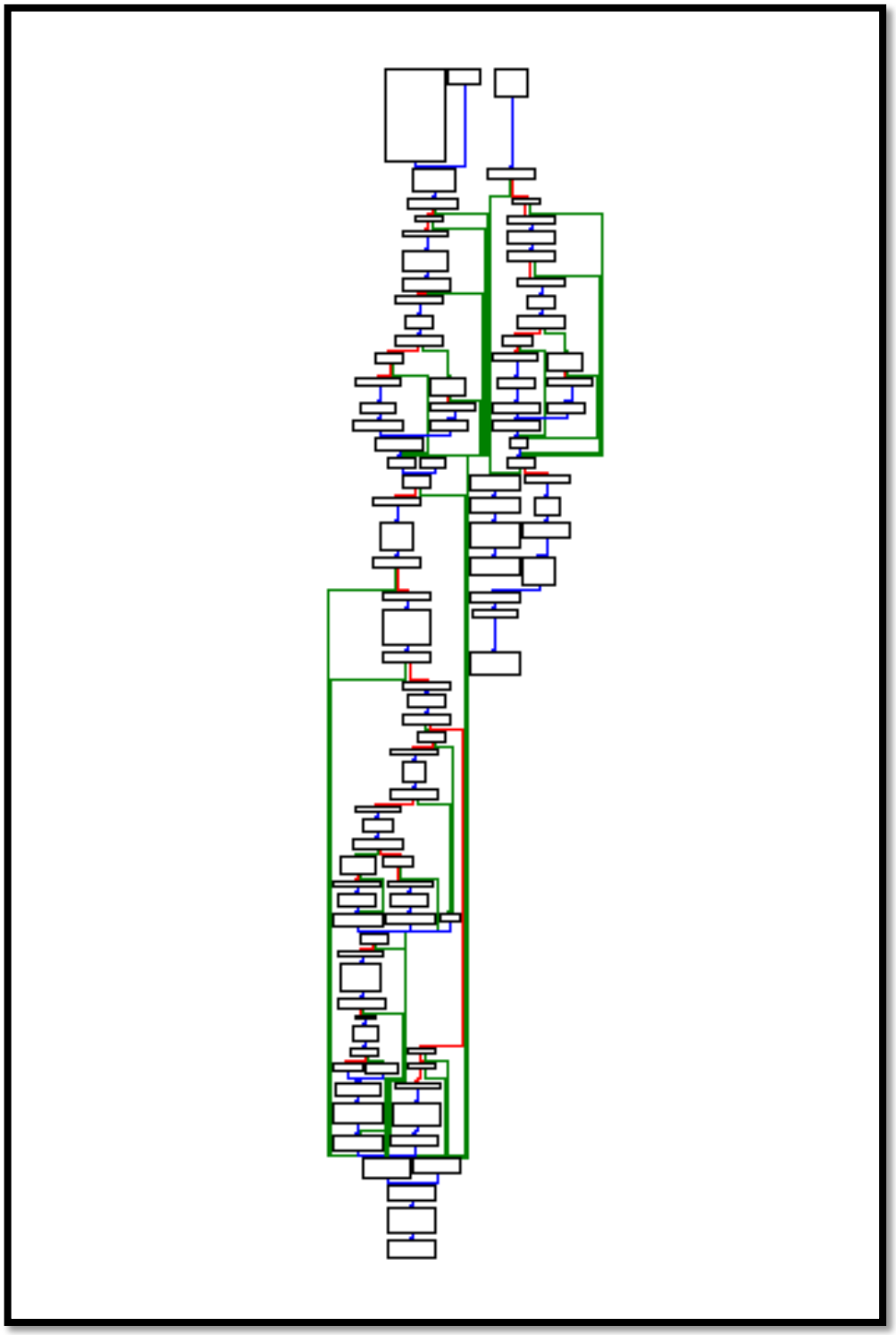

Figure 3.7 – Obfuscating graph view

3.2.2. Inserting conditional jumps

Conditional jumps insertion is enhanced unconditional jump obfuscation. Unconditional jumps obfuscation increases complexity of code but it does not affect program logic. Conditional jump insertion imposes another problem on the researcher, who has to decide which of the flow paths is taken. This enables malware authors to increase time needed to fully understand and analyze a malware sample.

Conditional jump destination is determined by checking appropriate EFLAGS register fields. The EFLAGS register contains fields that represent information about previously executed instructions. It contains the following flags:

- OF – overflow flag
- SF – sign flag
- ZF – zero flag
- CF – carry flag
- PF – parity flag

With insertion of conditional jumps the researcher is not able to deduce which execution path is taken by only performing a static analysis of a sample. Reason for that is this flags are updated depending on the instructions being executed. A malware author can artificially create suitable conditions such that needed flag is set and wanted code path is always taken. Several examples of such scenarios are illustrated in the following table.

Table 3.1 - Conditional jump obfuscation

`stc ;set carry flag`
`jb loc_804D729 ;jump if carry`
`;ELSE`
`pop ds`
`out 7Bh, eax`
	`xor eax, eax ;eax=0`
	`jz loc_804D729 ;jump if zero`
	`;ELSE`
	`pop ds`
	`out 7Bh, eax`

3.3. Data-Flow Obfuscation

Data-flow obfuscation is a transformation based on changing data and instruction representation. There are two main data-flow obfuscation types: inserting no operation (NOP) instructions and substituting instructions with complex representations.

3.3.1. Inserting no operation instructions

Insertion of no operation (NOP) instructions is a technique aimed at confusing researcher and increasing code complexity. Inserted instructions do not change semantics or control flow of a program but instead add to size of code to be analyzed thus increasing the total time needed for analysis. This type of inserted code is also called dead code because it does not affect state of a program.

There are a few things malware authors have to take care of while inserting NOP instructions. Not all instructions affect program state the same way, so those instructions that have some kind of external effect on the program like affecting the

stack or CPU flags have to be taken into special consideration. One way to implement this type of obfuscation is to separate instructions based on their effects. This way, it is possible to use only a subset of instructions that does not affect any processor flags and insert them anywhere in a basic block. Example of this type of obfuscation is shown in the following table.

Table 3.2 - NOP insertion obfuscation

Before Obfuscation	**After Obfuscation**

`mov eax, [ebp-24h]`
`jmp 0x12344321`
	`mov eax, [ebp-24h]`
	`push eax`
	`push ebx`
	`pop eax`
	`pop ebx`
	`xchg eax, ebx`
	`jmp 0x12344321`

Another way to take care of accidently modifying flags used for conditional jumps is to insert NOP instructions at the beginning of a basic block. This way the accidental modifications of flags will not affect conditional jump because instructions that affect the jump will be executed after the inserted NOPs and will overwrite all previous modifications of flags. Example of this type of obfuscation is shown in the following table.

Table 3.3 - NOP insertion in beginning of BB

Before Obfuscation	**After Obfuscation**

`test eax, eax`
`jz 0x12344321`
	`push eax`
	`mov eax, ebx`
	`sub eax, ecx`
	`add eax, 0x379`
	`sub eax, 0x29`
	`lea eax, [ebx + ecx]`
	`pop eax`
	`test eax, eax`
	`jz 0x12344321`

3.3.2. Substituting instructions with complex representation

CISC processor architectures provides a vast number of instructions so it is possible to solve common programming problems using different instruction subsets. Compilers play a significant role in finding best instruction subset for translating high level constructs, be it for speed or size performance. By analyzing commonly generated compiler code constructs it is possible to write obfuscation that translates optimal code into more complex representation.

One of most common language constructs is variable assignment. The processor provides the *MOV* instruction for implementation of assignment semantic. Because of frequency of its usage it is one of the best targets to obfuscate. The following table shows several examples of *MOV* obfuscations to illustrate commonly used ideas.

Table 3.4 - Instruction substitution idea

Before Obfuscation	**After Obfuscation**
`mov ebx, eax`	`push eax` `pop ebx`

`mov ebx, eax`	`xchg ebx, eax`
	`push ebx`
	`pop eax`

`mov ebx, 0xdeadbeef`	`push 0xdeadbeef` `pop ebx`

`mov ebx, 0xdeadbeef`	`sub esp, 0x4`
	`mov [esp], 0xdeadbeef`
	`pop ebx`

Using the same idea it is possible to construct obfuscations for most instructions, ultimately making code analysis hard and time consuming. The following table lists some popular instruction obfuscation themes.

Table 3.5 - Instruction substitution typical examples

Before Obfuscation	**After Obfuscation**

`xchg eax, ebx`	`xor eax, ebx`
	`xor ebx, eax`
	`xor eax, ebx`

`jmp 0x12344321`	`mov eax, 0x12344321` `jmp eax`
`jmp 0x12344321`	`jz 0x12344321` `jnz 0x12344321`

`jmp 0x12344321`	`push 0x12344320`
	`mov eax, [esp]`
	`add esp, 4`
	`inc eax`
	`jmp eax`

`call 0x12344321`	`push 0x12344321` `ret`
`ret`	`pop eax` `jmp eax`

4. Code Optimization

Code optimization is the process of modifying a program's code to make some aspect of it work more efficiently or satisfy other preferences like the program's size. Usual goal for code optimization is making the code execute faster. To achieve this, compilers remove unnecessary and transform suboptimal code sequences. Because obfuscations make suboptimal code transformations, and expressiveness of assembly language makes obfuscation possibilities unmanageable, a rule based deobfuscation process is unfeasible. By applying optimization algorithms one can undo obfuscation changes and simplify code representation without the need to enumerate all possible obfuscation types.

Next are presented several optimizations which can solve most common obfuscation techniques.

4.1. Control-Flow Optimization

Control-flow optimization is the process that aims at efficient memory organization of basic blocks (code blocks). Efficient basic block organization enables faster code execution by reducing the need for branching. Effects of this optimization are larger basic blocks, grouping of function blocks inside small memory regions and easier code navigation. This type of optimization can remove insertion of conditional and unconditional jumps.

Control-flow optimization process works on control-flow graphs (CFG). A CFG is program representation that uses graphs for modeling program's control flow. It is built using recursive traversal disassembly and uses basic blocks as node representation. Following is a list of graph properties:

- Each graph represents a single function.
- There is no code sharing between graphs (functions).
- Each node represents a basic block.
- Edges represent execution paths.

Most compilers treat call instructions as always returning and do not split basic blocks after a *CALL* instruction. While this makes sense in regular code, malware can exploit such assumptions and never return from a function call. This makes traditional basic block structure unfitting, so we will treat call instructions as conditional jumps and split the CFG after it. Following is a list of basic block properties:

- Has single entry point.
- Has single exit point.
- There are no code references to the body of basic block.
- Conditional, unconditional and call instructions mark the end of block.

When the function graph has been built, CF optimization traverses it and performs the following checks:

- If the block ends with an unconditional jump and its successor has only one parent then merge two blocks.

- If the block ends in a conditional jump and its false branch points to a conditional block which contains only conditional jump that is inverse of the previous one then substitute the conditional jump with unconditional and remove false branch.

The CF optimization algorithm repeatedly traverses a graph until there are no changes made to the CFG. The following table contains the CF optimization algorithm pseudocode.

```
CFGOptimization(root):
  changed = True
  addr_todo = List()
  addr_done = List()
  addr_todo.Append(root)
  while changed == True:
    changed = False
    addr = addr_todo.Pop()
    if addr in addr_done:
      continue
    addr_done.Append(addr)
    if SatisfyCFGOptimizationCondition(addr):
      MergeBlocks(addr)
      changed = True
    addr_todo.Append(Successors(addr))
```

Figure 4.1 - CFG optimization algorithm

4.2. Peephole Optimization

Peephole optimization is an optimization technique performed over a very small set of instructions in a code segment. This optimization enables usage of more aggressive optimization rules without the need for performing expensive data or control flow analysis. The size of an instruction set of peephole optimization is called a peephole or a window. The window size can vary, but usually is 3-8 instructions. The peephole technique can also be used to implement some less powerful variations of constant propagation and constant folding.

Peephole optimization is used for removing following obfuscation techniques:

- Complex instruction representation
- Conditional jump insertion
- Control flow changes

Optimization identification and transformation is rule based and can be subjected to user preferences. The following algorithm illustrates the working of peephole optimization.

```
PeepHole(root):
  WINDOW_SIZE = 5
  block = root
  while node:
    instruction_offset = 0
    While True:
      if rule = MatchRule(node, instruction_offset,
WINDOW_SIZE):
        ApplyRule(node, instruction_offset, rule)
```

```
      if instruction_offset != node.Size()-1:
        instruction_offset += 1
      else:
        break
    node = node.NextNode()
```

Figure 4.2 - PeepHole optimization algorithm

4.3. Constant Propagation

Constant propagation is a transformation that, given a constant assignment to a variable, replaces all subsequent usages of that variable, as an instruction argument, with a constant. The basic idea of constant propagation is given in the following table.

Table 4.1 - Constant propagation example

Before optimization	**After optimization**

`X = 10`
`Y = X`
`Z = X + Y`
	`X = 10`
	`Y = 10`
	`Z = 10 + 10`

Constant propagation optimization is a very interesting tool for simplifying arithmetic expressions, which are frequently used to obfuscate register values. When used together with constant folding it can also be used for replacing unknown jump and call arguments.

4.4. Constant Folding

Constant expression evaluation, or constant folding, is optimization used for substituting constant argument expressions with an evaluated result. Constant expression arguments can be integers, floating point numbers and truth values (logical operations). This kind of transformation has to be safe and produce code that is equivalent (semantically) to the original code. Safe transformations are those that generate code that will produce same results as the original ones. Integer and truth operations are usually safe because there are only a few cases that have to be handled with special care. The most well known integer problem is division by zero. Dividing or modulo any integer with 0 will result in the processor generating a "*integer division or modulo by zero*" exception. This follows from mathematical statement that division of any number by zero is not defined. Except special cases that are inherited from the number theory, there are also special cases that are reflecting number representation in memory. One less known example of such case is division of INT_MIN (-2147483648) and -1. Dividing these two numbers will result in generation of a floating point exception. On a two complement system |INT_MIN| is one greater than INT_MAX, so the result of the operation simply cannot fit in an integer and will raise an exception. But evaluating this expression in a language that uses different number representation will not generate the exception and will report an answer as |INT_MIN|. Examples of such cases are illustrated in following table.

Table 4.2 - Integer arithmetic exception

C, Assembly	**Python**
./eval 12 0 Floating point exception (core dumped)	In [2]: 12/0 ZeroDivisionError
./eval -2147483648 -1 Floating point exception (core dumped)	In [1]: -214783648 / -1 Out[1]: 214783648

Unlike integers, floating point numbers are more sensitive to constant folding so special care has to be taken when evaluating floating point expressions. Floating point arithmetic can be evaluated using code emulation on a designated CPU. While emulating the code, all expressions that generate exceptions should be left unmodified. Reason for this is that exceptions are one of the ways that malware can use for altering control flow, and for this reason all exceptions must occur even after code optimization.

Constant folding is most frequently used together with constant propagation to obtain the best optimization results. The following table illustrates how iterated combination of constant propagation and folding simplifies the code.

Table 4.3 - Example of combining constant propagation and folding

Original code	**Constant propagation**	**Constant folding**
X = 10 Y = X Z = X + Y Y = Z		
	X = 10 Y = 10 Z = 10 + 10 Y = Z	
		X = 10 Y = 10 Z = 20 Y = Z
	X = 10 Y = 10 Z = 20 Y = 20	

Some common obfuscation patterns and their optimized versions are presented in following table.

Table 4.4 - Real example of constant propagation and folding

Original code	**Constant propagation**	**Constant folding**

mov eax, 0x1234
shl eax, 0x10
add eax, 0x4321
jmp eax
	mov eax, 0x1234
	;eax=0x1234
	shl eax, 0x10
	;eax = 0x1234 << 0x10
	add eax, 0x4321
	jmp eax
		mov eax, 0x1234
		;eax=0x1234
		shl eax, 0x10
		;eax = 0x12340000
		add eax, 0x4321
		jmp eax

	mov eax, 0x1234 ;eax=0x1234	mov eax, 0x1234 ;eax=0x1234

	shl eax, 0x10 ;eax = 0x12340000 add eax, 0x4321 eax = 0x12340000 + 0x4321 jmp eax	shl eax, 0x10 ;eax = 0x12340000 add eax, 0x4321 eax = 0x12344320 jmp eax

	mov eax, 0x1234
	mov eax, 0x12340000
	mov eax, 0x12344321
	jmp 0x12344321	

4.5. Dead code removal

Dead code removal is an optimization technique that removes instructions that do not affect program state. Dead code in obfuscated programs can be a result of insertion of no operation instructions and removing it greatly simplifies the program's logic. Dead code can also be a result of several optimization techniques like constant propagation or folding.

Simple dead code removal algorithm checks if a register value is being overwritten before it is used. If that is the case then the first assignment to that register can be safely removed. One thing to note is that word *register* describes not only general purpose registers like (eax, esi, and the rest) but also the EFLAGS register.

Example of dead code removal is illustrated in following table.

Table 4.5 - Dead code removal example

Original code	**Dead code removal**

mov eax, 0x1234
mov eax, 0x12340000
mov eax, 0x12344321
jmp 0x12344321
	mov eax, 0x12344321
	jmp 0x12344321

4.6. Optimizations ordering

Optimizations can be performed in any desired order, but there are some typical optimization orderings. The following figure illustrates common optimization order used by compilers. Optimizations are grouped in blocks and block ordering is illustrated with arrows. Typically only a subset of optimizations from a specific block is performed.

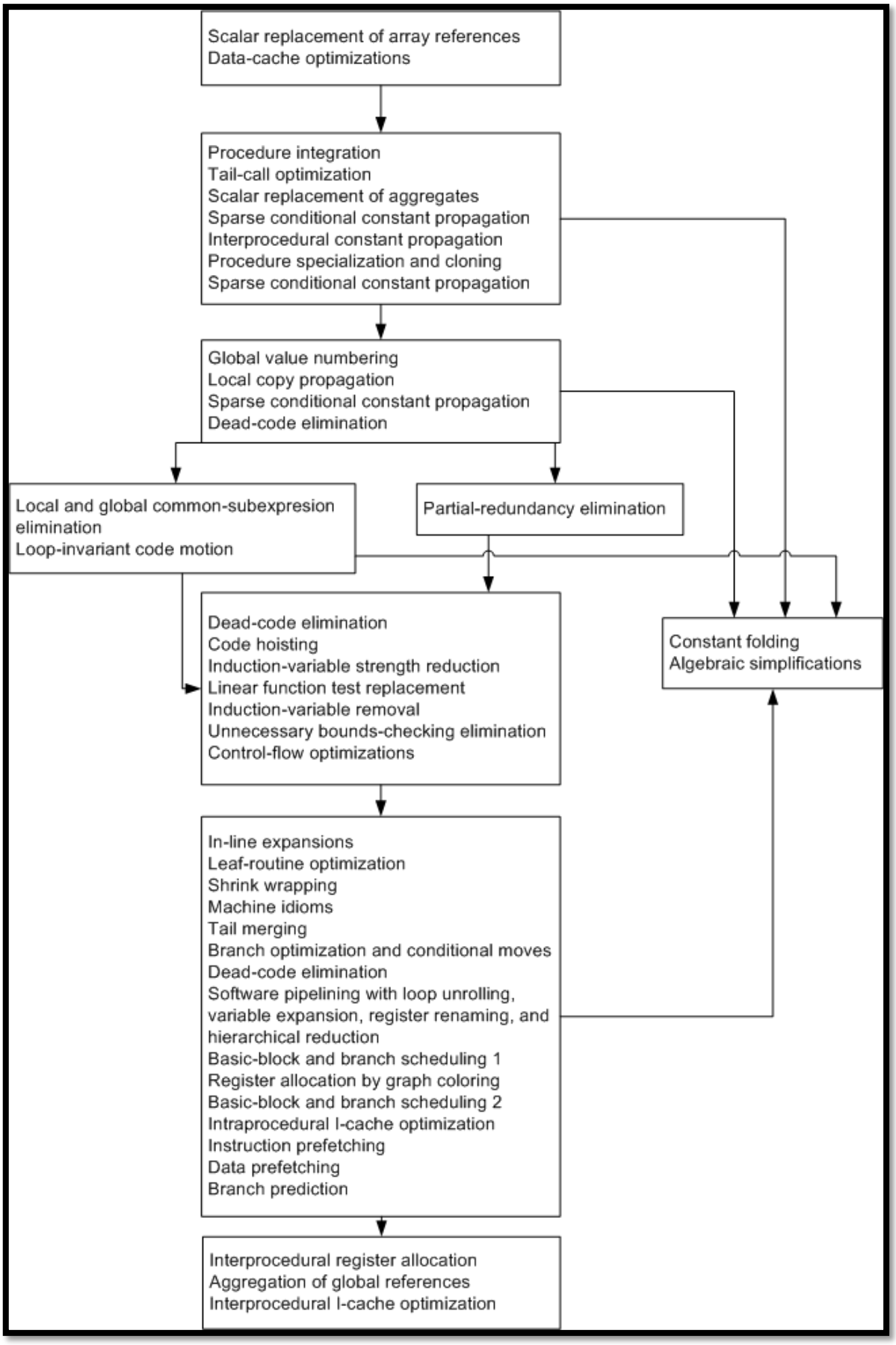

Figure 4.3 – Compiler optimization block diagram

5. Binary Optimization Framework

As a part of this thesis and as a testing ground for various optimizations, a binary optimization framework has been developed. It is meant to be a modular extension of the IDA Pro disassembler capable of rewriting binaries using optimization algorithms. Next important feature should be integration into current disassembly sessions and ability to browse optimized code inside the IDA disassembly window. Also, it should not modify original code, but enable synchronization between optimized and original code. To satisfy all of the above the following design was chosen:

- New code segment is created to accommodate optimized code.
- All optimizations are done in memory on low level code representation.
- After optimization, the assembler generates opcodes and writes them to the optimized code segment.
- All optimized functions contain address links to their originating functions.
- Functions should be reconstructed so that IDA recognizes them.
- Comment reconstructed code with useful information (eg. recognized protection schemes).
- Enable users to add custom heuristic optimization transformations.

This provides the following analysis benefits:

- All binary modifications are done to the IDA database, no modifications are done to the original binary.
- It is possible to cross reference optimized and original code.
- Optimized code is located in a separate segment and does not overlap with original code.
- All IDA features like cross references, variable renaming etc. can be applied on the optimized code segment.
- Enables usage or provides better performance to other code analysis tools like HexRays decompiler.
- It is possible to execute or emulate optimized code.
- IDA recognizes functions and enables usage of graph disassembly view.
- Commented anti-debugging and anti-disassembly techniques as notifications for less experienced analysts.
- Custom tailored heuristic rules for analyzed sample enables better and faster optimization process.

The framework consists of three main modules:

- Assembler
  - Assembles transformed code constructs to opcodes.
- Stack emulator
  - Responsible for stack alias analysis and value tracking.

- Optimizer
    - Control and data flow optimization algorithms.

The following figure illustrates framework overview with internal organization and module interaction.

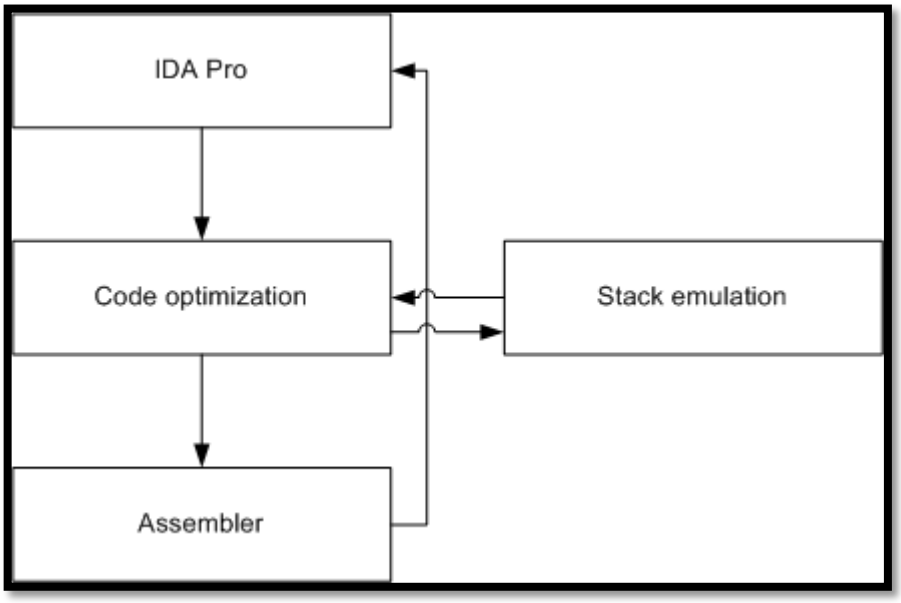

Figure 5.1 – Framework overview

A more detailed scheme of the whole optimization process is given in Figure 5.2. This figure illustrates steps taken to optimize arbitrary code.

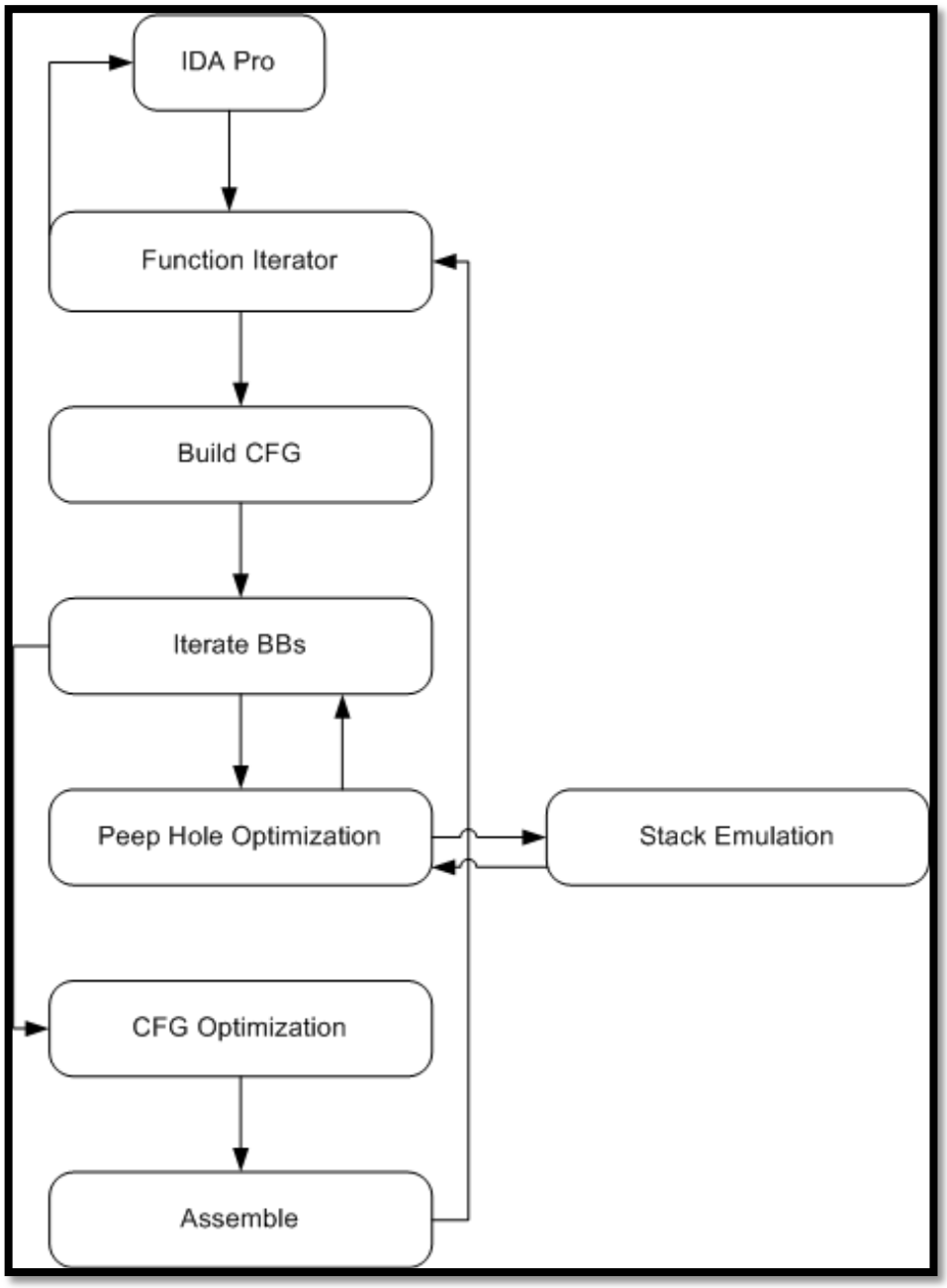

Figure 5.2 – Detailed optimization process

5.1. Optimizer

Optimization algorithms are grouped in two main categories: peephole and CFG optimizations.

Peephole incorporates the following optimizations:

- Heuristic transformation rules
  - Predefined rules
  - Researchers custom regex defined rules
- Constant propagation
- Constant folding

- Dead code elimination

Control flow optimization incorporates following transformations:

- Removal of unconditional jumps
- Removal of conditional jumps

5.2. Assembler

The assembler component is responsible for insertion of code into the IDA database and making it available for analysis.

The IDA Pro disassembler uses slightly different assembly language for instruction representation. This difference can cause some problems while trying to assemble IDA's specific representation into opcodes. To circumvent this problem it is necessary to modify assembly source or assembler compiler to recognize new representation. Because only a subset of language is different, it was chosen to modify assembler compiler. An assembler wrapper is used to generate opcodes for instructions that differ from standard, and provided assembler compiler API is used to assemble all regular instructions.

Additional benefit is that it generates working executable code that can be executed or emulated. This provides researchers with ability to investigate code execution or collect executed code results.

6. Results

In this section we examine optimization results on some obfuscated samples. The first example in Figure 6.1, is a program obfuscated with PE-Scrambler developed by Nick Harbour, and demonstrated at the DEFCON 16 conference in August 2008.

6.1. Function optimization

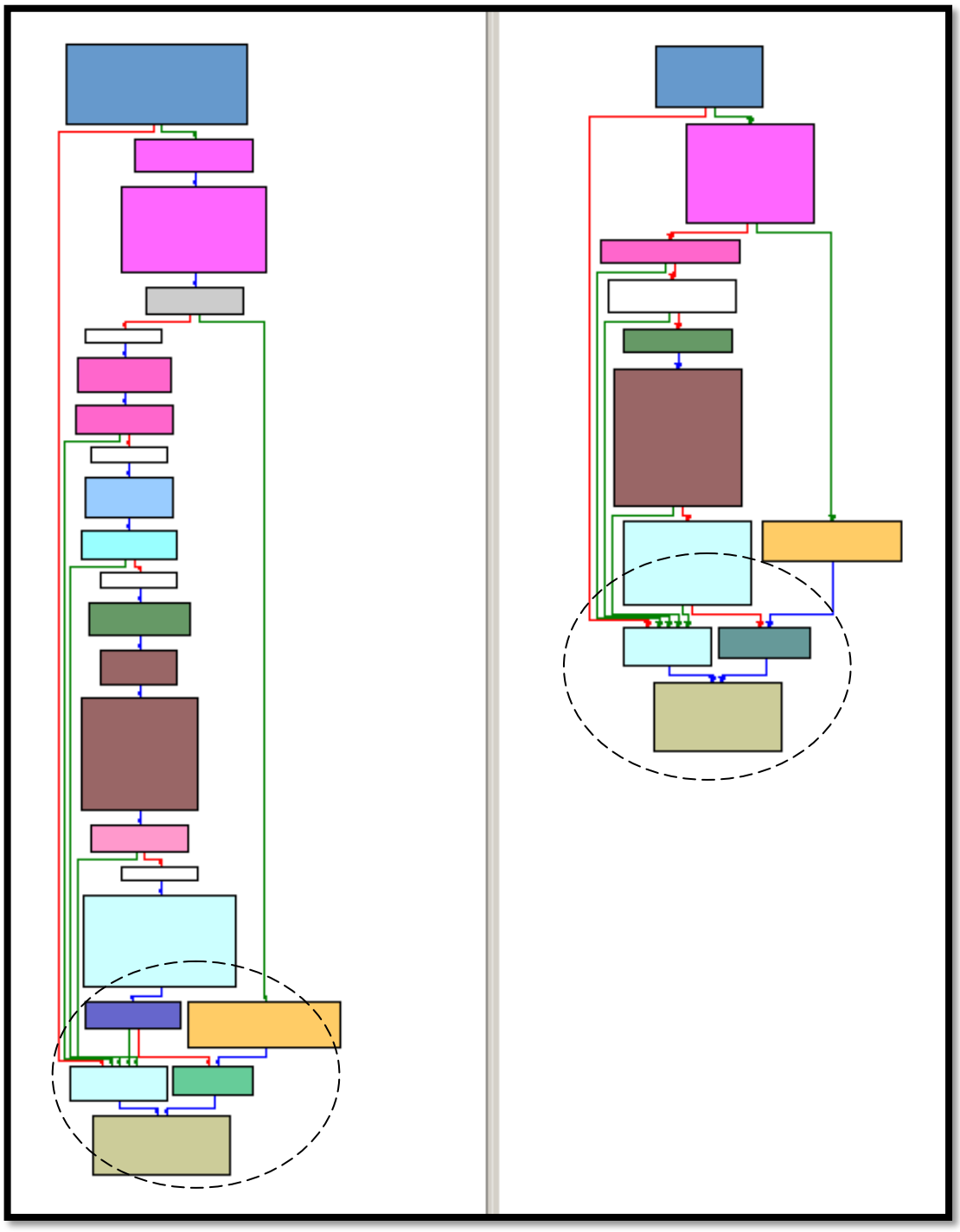

Figure 6.1 – PE-Scrambler, Left – Obfuscated, Right – Optimized

Figure 6.1 shows comparison of original and optimized CFG. The basic blocks are colored to represent which blocks are optimized and still exist in the optimized function. Smaller number of basic blocks results from merging unconditional jumps and smaller block size is a result of peephole optimizations.

Figure 6.2 comes from zooming in the circled code in Figure 6.1 and shows how IDA was unable to show part of the original function, because stack analysis had failed (left frame), and how the optimized graph contains full code listing (right frame).

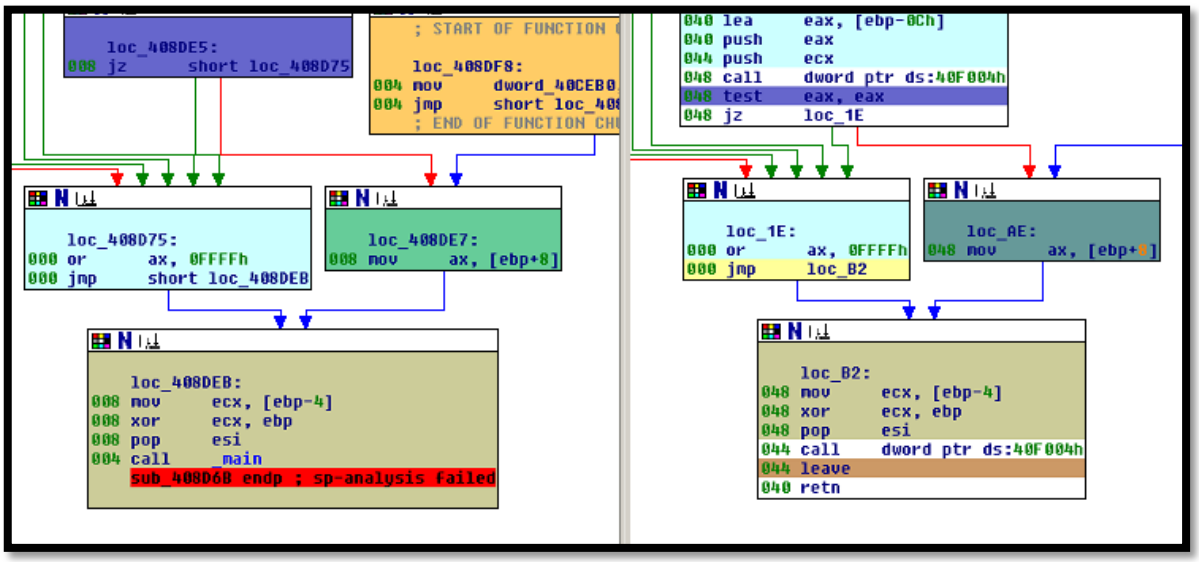

Figure 6.2 – Full code listing in optimized function, Left – Obfuscated, Right – Optimized

IDA has its own heuristic engine for detecting code constructs and uses it for overaproximating function size. Figure 6.3 shows how IDA adds basic blocks containing conditional jumps to the functions graph view. Since references from jumps split basic blocks, the code looks needlessly more complicated and fragmented. This is an example of an obfuscation that exploits workings of a disassembler to increase complexity of the CFG and slow down analysis.

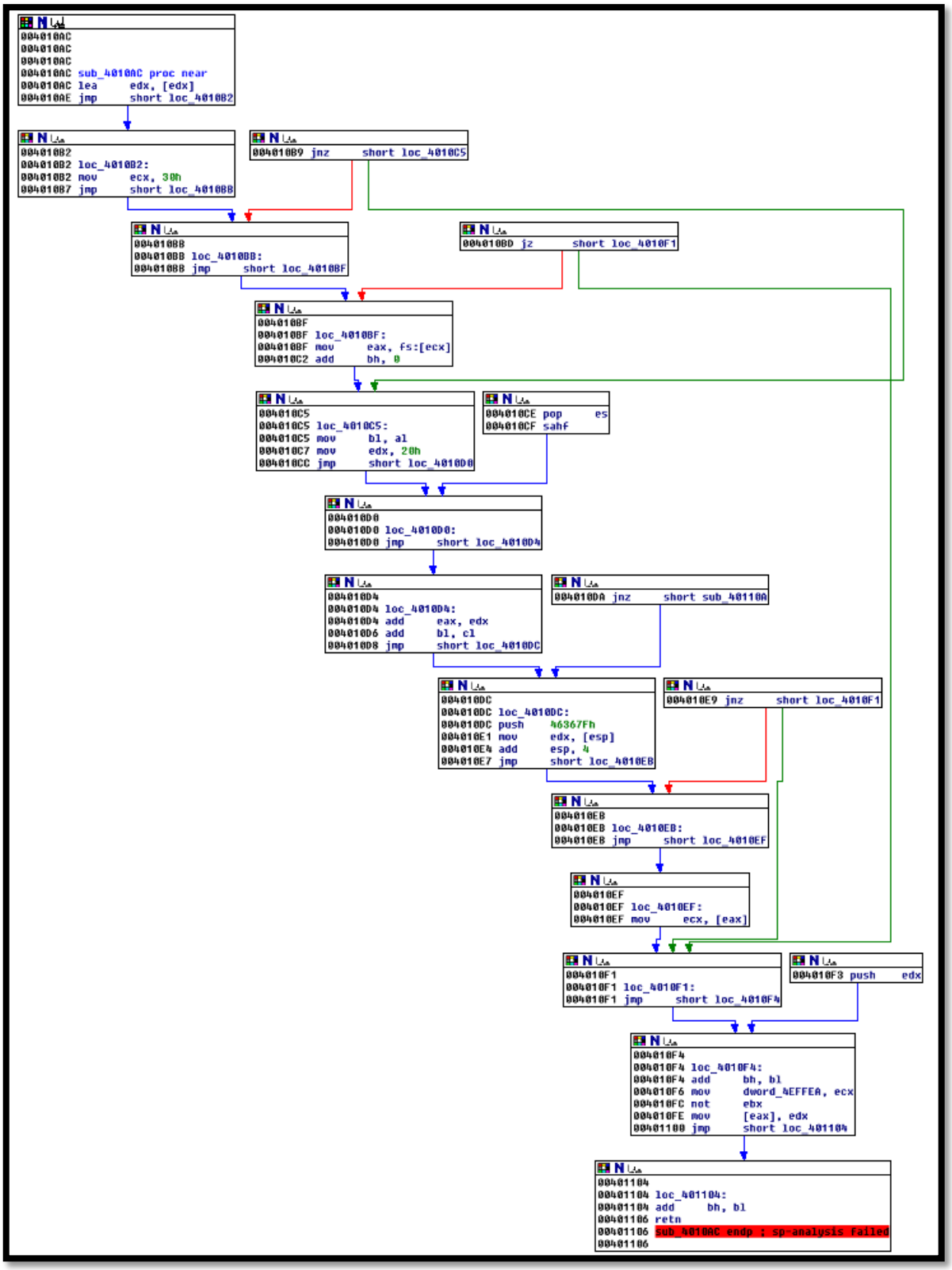

Figure 6.3 – CFG obfuscation, Left – Obfuscated, Right – Optimized

Figure 6.4 shows the optimized version of function in Figure 6.3 that consists of only a single block and reflects obfuscation and optimization complexity ratio.

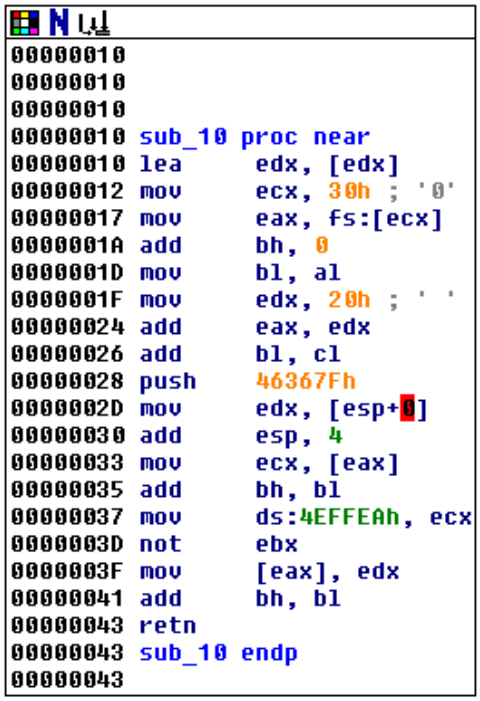

Figure 6.4 – Optimized CFG

Figure 6.5 show side by side disassembly view of the obfuscated and optimized function code. It is evident that the obfuscated code is very fragmented and inside every basic block is only one or two instructions of interest.

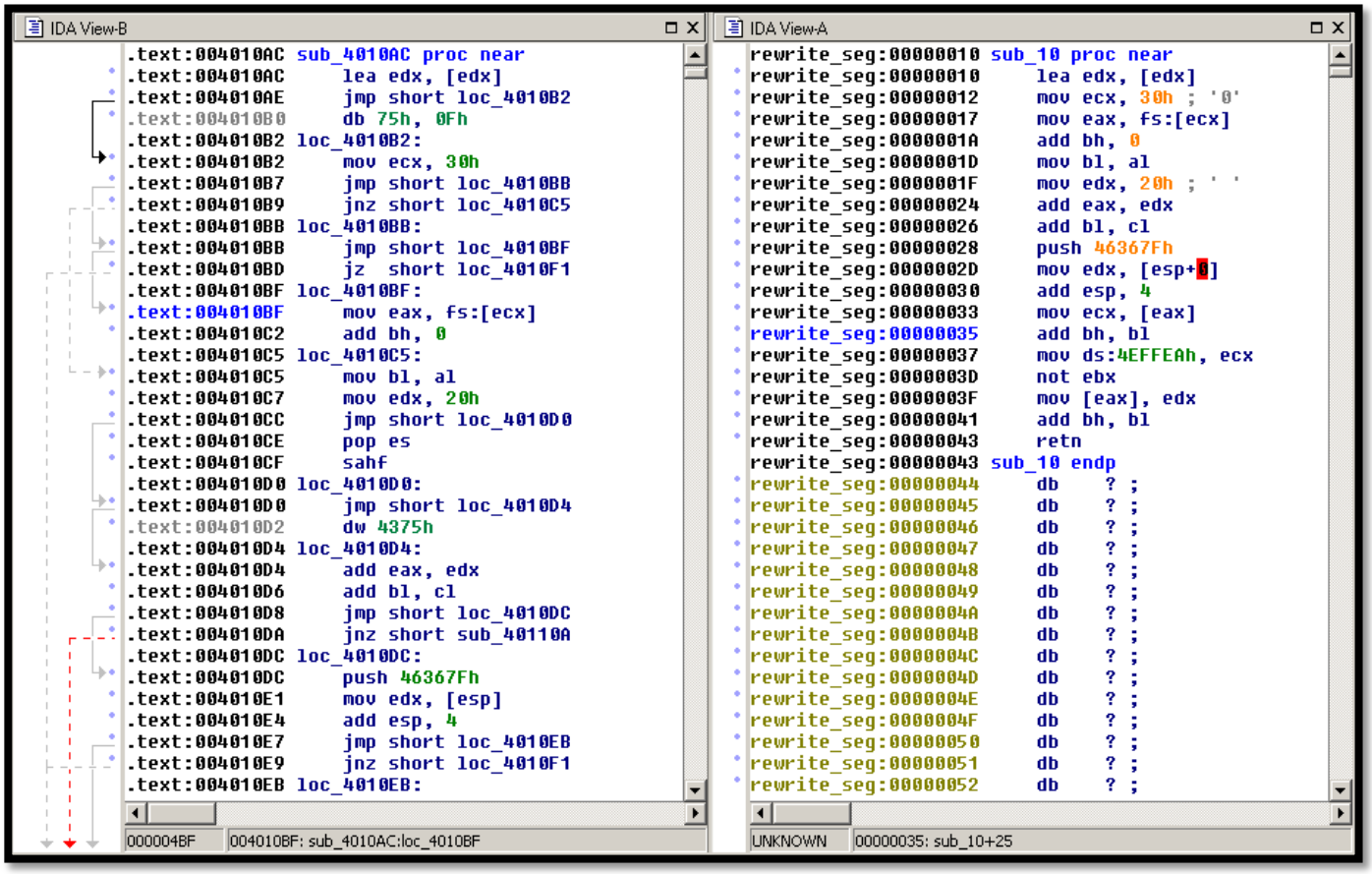

Figure 6.5 – Disassembly view, Left – Obfuscated, Right – Optimized

Figure 6.6 illustrates optimization results on a more complex CFG. The following example illustrates also fake block references that act as dead code, but are present in a functions graph view.

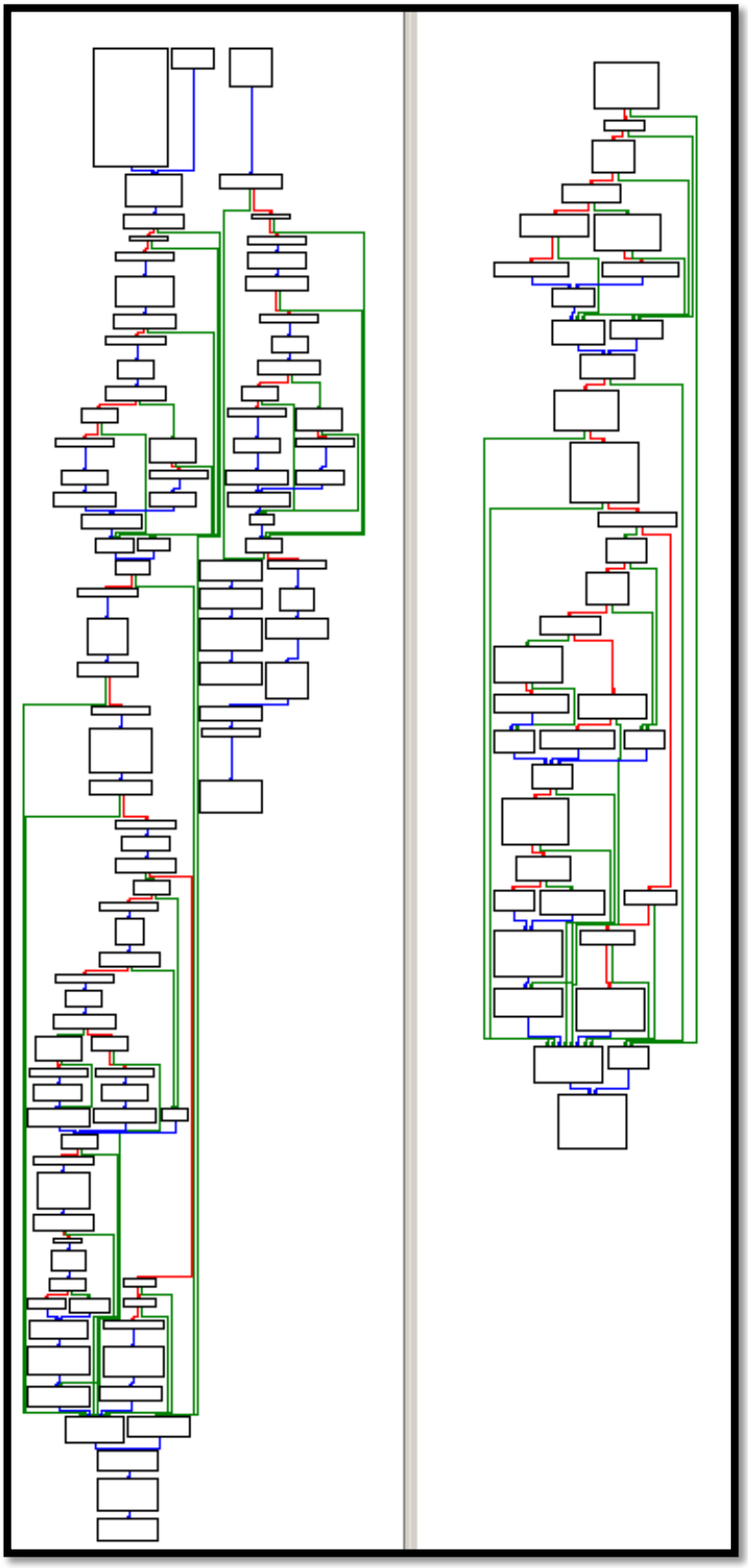

Figure 6.6 – Optimizing complex CFG, Left – Obfuscated, Right – Optimized

6.2. Function reconstruction

Figure 6.7 illustrates custom jump insertion and CFG obfuscation that bypasses IDA function detection and results in wrong function definition. This kind of obfuscation is effective at destroying IDA's function detection algorithms and as such prevents usage of graph view. Because basic blocks are scattered across address space, the linear disassembly view is very hard to follow. The left frame shows obfuscated disassembly view and the right one reconstructed functions code.

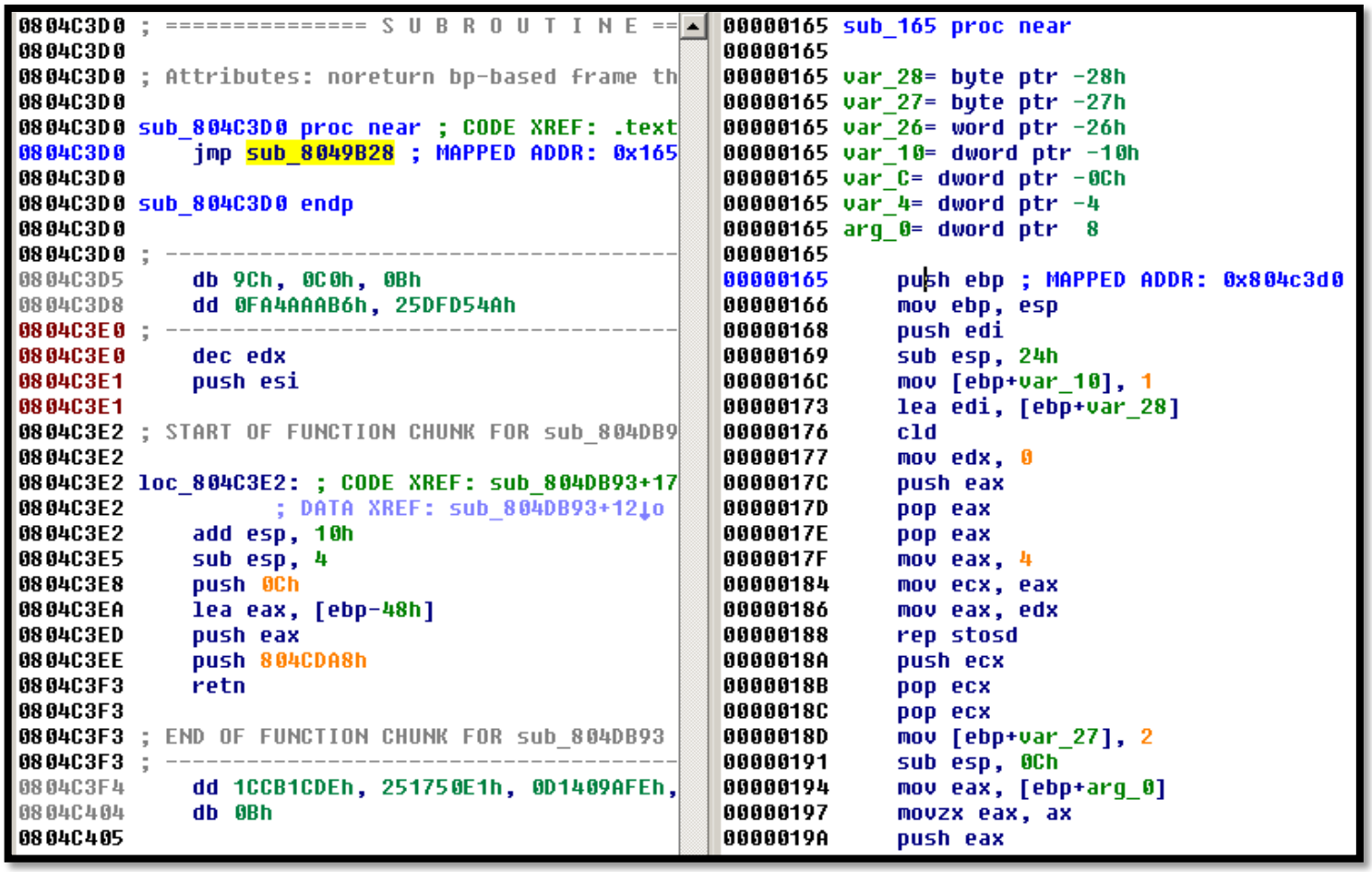

Figure 6.7 – Function reconstruction, Left – Obfuscated, Right – Optimized

Figure 6.8 shows graph view of reconstructed function code. It is evident that graph view is much easier to navigate and provides insight to function logic, but graph view is unavailable to researcher inspecting original code because code is obfuscated and IDA is unable to generate it.

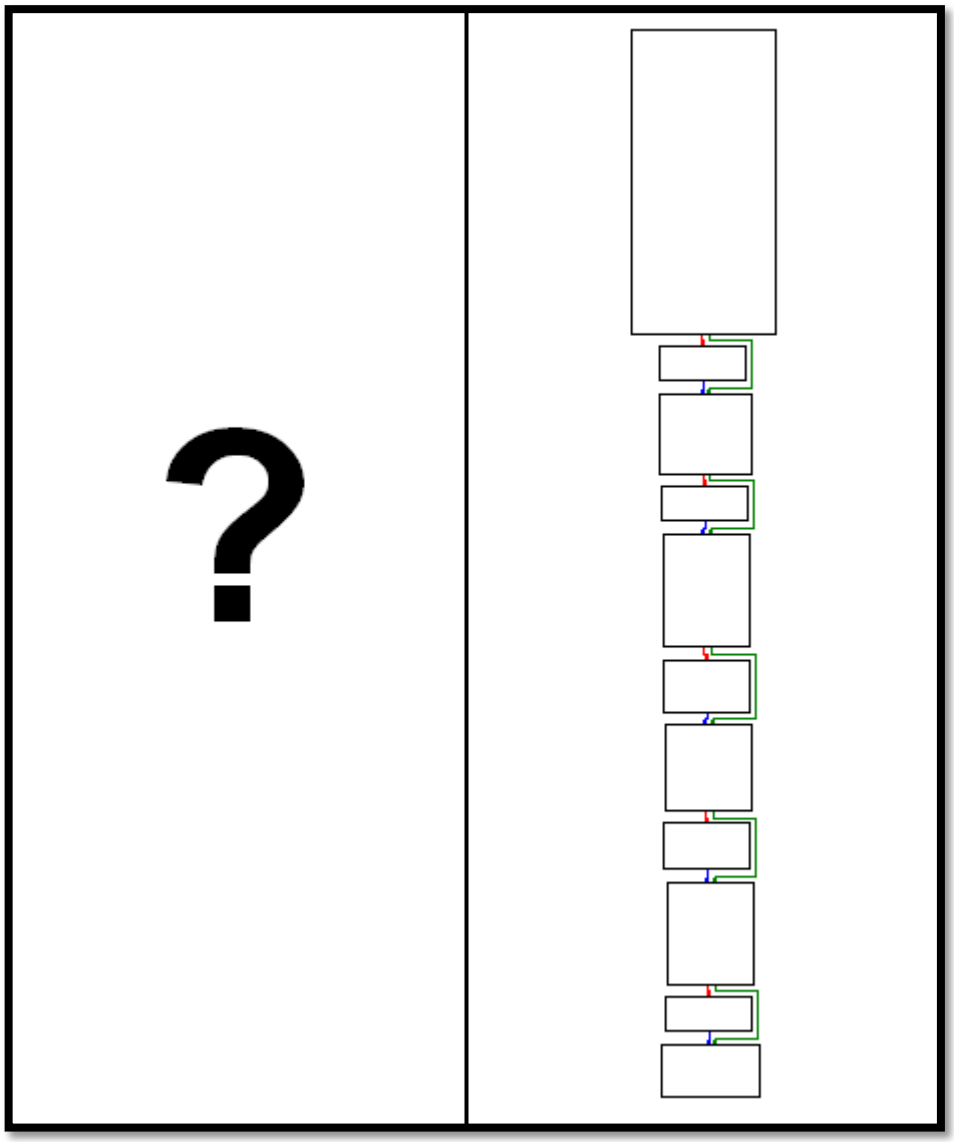

Figure 6.8 – Function graph reconstruction, Left – Obfuscated, Right – Optimized

7. Summary

Analysis speed is one of the biggest problems in the anti-virus industry. With the number of unique malware samples reaching millions in a single day, correct and fast code analysis is crucial. Optimization algorithms present an effective way for removing most obfuscations that are used today. Much of the compiler theory can be applied in removing obfuscations and building fast and reliable deobfuscation systems.

Optimization algorithms are especially successful in following:

- Removal of no operation instructions
- Simplifying complex instructions
- Removal of unconditional jumps
- Removal of conditional jumps
- Simplifying control-flow graph

While the optimization framework introduced in this thesis allows removing of many popular obfuscations, there is much space for improvement. One of the major problems of the presented framework is its dependence on the Intel x86 architecture. By introduction intermediate representation it is possible to create an architecture independent optimization framework that could also incorporate the SSA representation. The SSA representation is interesting due to its speed with many optimization algorithms.

By understanding traditional optimization problems and techniques it is possible to develop and customize compiler optimization algorithms for usage in malware deobfuscation/analysis. This in turn could provide better foundation for many other kinds of malware analysis techniques and provide better facilities to the AV industry.

8. Bibliography

1. Protecting our customers from half a million new unique malicious files every day, http://blogs.technet.com/mmpc/archive/2009/04/30/protecting-our-customers-from-half-a-million-new-unique-malicious-files-every-day.aspx
2. IDA Pro disassembler, http://www.hex-rays.com/idapro/
3. Titan Engine, http://reversinglabs.com/products/TitanEngine.php
4. VmWare, http://www.vmware.com/
5. VirtualBox, http://www.virtualbox.org/
6. MwAnalysis, http://www.mwanalysis.org/
7. JoeBox, http://www.joebox.org/
8. Olly Debugger, http://www.ollydbg.de/
9. Immunity Debugger, http://www.immunityinc.com/products-immdbg.shtml
10. WinDBG, http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx
11. The International Obfuscated C Code Contest, http://www.ioccc.org/
12. The Underhanded C Contest, http://underhanded.xcott.com/
13. C. Kruegel, W. Robertson, F. Valeur, G. Vigna, Static Disassembly of Obfuscated Binaries", Proceedings of the 13th USENIX Security Symposium, 2004.
14. G. Wroblewski, General Method of Program Code Obfuscation. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
15. S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In G. Davida and Y. Frankel, Eeditors, Information Security, ISC 2001, volume 2200 of Lectures Notes in Computer Science (LNCS):144– 155 Springer–Verlag, 2001. 68, 2001.
16. J. Raber, E. Laspe, An Automated Approach to the Identification and Removal of Code Obfuscation, http://www.rri-usa.org/Deobfuscator.pdf
17. C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Technical Report 148, The University of Auckland, New Zealand, 1997.
18. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997
19. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual, Intel Press, 2003
20. E. Eilam, Reversing: Secrets of Reverse Engineering, Wiley, 2005
21. N. Harbour, Advanced Software Armoring and Polymorphic Kung-Fu, http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-harbour.pdf
22. C. Linn, S. K. Debray, Obfuscation of executable code to improve resistance to static disassembly. Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003), Oct 2003.
23. S. Udupah, S. Debray, M. Madou, Deobfuscation: Reverse engineering obfuscated code, May 2005.

24. C. Cifuentes, K. Gough, Decompilation of Binary Programs. Software Practice & Experience, 25(7):811–829, July 1995.
25. W. Amme, P. Braun, E. Zehendner, F. Thomasset, Data dependence analysis of assembly code. Int. J. Parallel Proc., 2000.
26. D. Brumley, J. Newsome, Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University, School of Computer Science, December 2006.
27. C. Cifuentes, A. Fraboulet, Interprocedural data flow recovery of high-level language code from assembly. Technical Report 421, Univ. Queensland, 1997.
28. C. Cifuentes, D. Simon, A. Fraboulet, Assembly to high-level language translation. In Proc. Int. Conf. on Software Maintenance (ICSM), pages 228–237, 1998.
29. R. E. Rolles: Compiler 1, X86 Virtualizer 0, April 4th, 2008. http://www.openrce.org/blog/view/1110/
30. _g_, Fighting Oreans' VM (code virtualizer flavour), August 19th, 2008. http://www.woodmann.com/forum/showthread.php?t=12015
31. R. Rolles, Unpacking virtualization obfuscators, USENIX Workshop on Offensive Technologies, 2009.
32. C. Eagle, The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, No Starch Press, 1st ed, August 12, 2008.
33. L. Boehne, Pandora's Bochs: Automated Unpacking of Malware, Diploma thesis, January 2008.
34. G. Balakrishnan, WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison, August 2007.