UNIVERSITY OF OSLO
Department of Informatics

# Detection of Junk Instructions in Computer Viruses

Master thesis

Egil Aspevik Martinsen

June 2nd 2008

# Abstract

The techniques employed by viruses to avoid detection by antivirus scanners are becoming increasingly advanced. One technique commonly used by viruses to evade detection is *polymorphism*. The level of polymorphism in a virus indicates its ability to create different forms of itself. The use of *junk instructions* is a common technique to increase the level of polymorphism in a virus. Junk instructions are machine code instructions with no other function than to alter the appearance of a virus. Junk instructions do not contribute to the function of the virus, only the form.

This master thesis focuses on the problem of separating junk instructions from non-junk instructions in computer viruses. To assail the problem, a junk instruction detection (JID) framework has been developed, capable of detecting junk in viruses created for the Intel IA-32 Architecture® (x86). JID relies on the *static instruction information* produced by a disassembler. Static instruction information describes the static information found in an instruction: the number of input/output operands and their accesses. Because JID only depends on the static instruction information, JID can possible be ported to other processor architectures. As long as there exist a disassembler for the architecture capable of producing static instruction information, JID is portable.

The results of testing JID on polymorphic viruses are promising. Tests show that JID manages to detect junk instructions found in the polymorphic virus Zmist. It is believed that JID would work as a tool to detect and remove junk instructions from future polymorphic viruses, thus reducing the time spent analysing the virus. Additionally in this thesis, the virus Zmist is analysed as a case study, and a detection algorithm devised. The detection algorithm manages to detect 100% of the polymorphic version of Zmist, although shortcomings in the disassembler algorithm reduces this number to 94%.

**Keywords:** *Polymorphism, metamorphism, polymorphic, metamorphic, computer virus, malware, worm, junk instruction, garbage instruction, junk code, garbage code, Zmist*

# Preface

This report presents the results of my master thesis “Detection of Junk Instructions in Computer Viruses”. The report is a part of my Master’s Degree in Computer Science at the Department of Informatics, University of Oslo. The contents of this report has its origin from a problem presented by the Norwegian antivirus company Norman ASA.

First of all, I would like to thank my supervisor Leif Nilsen at UNIK. His supreme ability to give constructive, detailed and mathematical precise guidance, has helped me a lot during my thesis work. Without his advices, the quality of the thesis would have been significantly lower and I am truly grateful for his patience and support in the finishing stages of the thesis.

I would like to thank Norman ASA who has provided me with a foundation for creating this thesis. Specially, I would like to thank Snorre Fagerland, my main contact at Norman, for answering numerous questions, steering me away from common pitfalls and providing me with the tools necessary to perform virus analysing. I would also like to thank Gunnar A. Johansen and Steve for helping me with the testing environment at Norman ASA. My fellow students Arne Hanssen and Sverre Krohn Tennøe deserve a big thanks for providing excellent technical input and for reviewing my thesis. Also, their social input have been of great value to me.

I would also like to thank my family and my in-laws for interrupting their daily duties to babysit our precious Lotte, while I and my wife Åshild attended to our theses. They have also provided us with shelter, food and many laughs and I look forward to spending more time with them again.

The person who has helped me the most is my wife Åshild, and I owe her huge thanks. She has been my primary support when lacking motivation, and I could never have completed this without her. Åshild has taken care of Lotte, sacrificing her own work to give me time to devote myself to the thesis. She has also reviewed my thesis and provided me with many constructive suggestions.
Finally, I look forward spending more time with my daughter Lotte, who has had to put up with her fathers tedious work.

Oslo, June 2nd 2008
Egil Aspevik Martinsen

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The topic of this thesis is detection of junk instructions found in computer viruses. Junk instructions are one of many techniques used to increase the level of *polymorphism* in viruses. The expression polymorphism stems from Latin. Poly translates to "many" and morph translates to "form". The level of polymorphism in a virus indicates its ability to create different forms of itself. The higher the level of polymorphism, the greater variety between the forms. The traditional detection method is based on byte-signature scanning. The variety found in polymorphic viruses implies that each form requires an individual byte-signature. This leads to that the byte-signature approach in the context of polymorphic virus detection, is considered deprecated. Polymorphism is therefore a problem for the antivirus companies.

**Preliminary Notation**

The terms *metamorphic* and polymorphic computer viruses are often used promiscuously. Meta is Latin and translates to "change". Metamorphism is used to describe viruses *changing* themselves, instead of producing new forms. The two terms and their differences when used to describe computer viruses, are explained in section 2.4.5 on page 36. Until then, the term polymorphism is used to describe both types.

The initial case study done in the thesis was focused on a computer virus and the term computer virus has been used since. Where the correct term would perhaps be malware, the term computer virus or virus is therefore used. Only when the context require separate identification, the different terms are used.

## 1.1 Motivation

Virus writers have commonly been thought of as "computer nerds" seeking to increase their reputation in underground subcultures by writing viruses. From article "The Generic Virus Writer" [25] written in 1994, the motivation for writing viruses between the studied groups of virus writers varied. Writing viruses for economical benefits is not stated as a motivation. From "Hackers/Crackers" [51] written in 2006, the motivation for writing malicious programs or *malware* has changed. The quote is translated from Norwegian.

> *The annual report of SIS (Senter for Informasjonssikring) describes the current threat picture. The report shows that there is an increased development in the use of security holes to gain profits. It is not the "computer nerds" which are responsible for malware production anymore; organized crime is behind and large amounts of money is involved.*

Mikko Hypponen of antivirus company F-Secure stated the following in 2006 [22].

> *The most significant change has been the evolution of virus writing hobbyists into criminally operated gangs bent on financial gain.*

Symantec confirms this trend in [37]. Fortinet writes extensively about how the virus business is organised using mafia-like terms: the members are divided into *coders, kids, mobs and drops* [26]. Each group has dedicated tasks in the virus production cycle and there are large sums of money involved. Valerie McNiven, US Treasury advisor on cybercrime stated the following in 2005 [26].

> *Last year was the first year that proceeds from cybercrime were greater than proceeds from the sales of illegal drugs, and that was, I believe, over $105 billion.*

The concept of virus-writing has evolved from vandalism to organised crime. Following from this evolution, the amount of new complex viruses written are higher every year. Spinellis has devised a proof stating that general detection of evolving, bounded length viruses is NP-complete [34]. This is not necessarily true when a detection algorithm is allowed to produce *false positives*. False positives are clean files erroneously reported as containing a virus. Many articles on detection of both known and unknown polymorphic viruses are written and because of the increased cybercrime, there is an increased interest in the field. However, no unified method of detecting polymorphic viruses has been elected, and the different methods have their strengths and weaknesses.

## 1.2 Problem Statement

In October 2006, I contacted the Norwegian antivirus company Norman ASA with an enquiry about writing a master thesis on the subject of computer viruses. The selected field of interest was "Detection of polymorphic viruses". Because this is a rather broad field, two research goals were defined to aid the process of determining a final problem statement.

1. Learn about polymorphic viruses through a case study.
2. Examine if there is possible to infer any knowledge from the case study which could be used in general detection of polymorphic viruses.

The first research goal had the following implications. To analyse the case study, the PE file format, the x86 assembly language and diverse reverse engineering tools had to be mastered. The virus selected for the case study were *Zmist*, written in 2001. Because detecting Zmist would give a better foundation for the second research goal, the first goal were extended to include detection of Zmist.

After 8 months, the first research goal were reached and the focus was placed on the second research goal. When analysing Zmist, most of the time were spent *comparing* the different forms. The comparison foundation acquired lead to a working detection algorithm and conforming to the second goal, the following question was posed:
"Is it possible to automatise or semi-automatise this comparison process?"

The first step in such an automatised comparison, would be to remove the detectable elements that are <u>not</u> common patterns in the different forms of a polymorphic virus. Junk instructions are an example of such detectable elements, and it was decided that junk instructions should be removed as the first step in the process of creating a automatised comparison program.

### Problem Statement

As a result of the research conducted related to the second goal, the following problem statement was formulated.

> Is it possible to separate junk instructions from non-junk instructions in an executable file?

The contents of this thesis are therefore a product of the studies made, and the tools and techniques developed, to be able to produce an answer to the problem statement.

## 1.3 Thesis outline

The thesis is comprised of the following chapters.

- Chapter 1 (Introduction) is an introduction to the thesis, describing the motivation, research goals and problem statement.
- Chapter 2 (Background) contains background information necessary to know when analysing viruses. Further, computer virus history, polymorphic viruses and detection methods are explained in this chapter.

- Chapter 3 (Case Study) provides the results of the first research goal. In this chapter Zmist is discussed and analysed, and two detection attempts are described. Most of the Zmist analysis and both the detection attempts are results of my own findings. A disassembler termed *Gdasm* was specifically developed to aid the research conducted in this thesis and Gdasm is presented in this chapter.

- Chapter 4 (Analysis and Lessons Learnt) describes the analysis and outcome related to the second research goal.

- Chapter 5 (The JID Framework) contains the discussion and design of a junk instruction detection framework. This chapter tries to answer the problem statement and the results developed in this chapter are of my own findings.

- Chapter 6 (Testing) describes the different tests conducted and the results. The tests are related to the Zmist detection algorithm developed in chapter 3 and the JID framework developed in chapter 5.

- Chapter 7 (Conclusion and Future Work) contains a summary of the thesis, the contributions made and suggestions for future work.

# Chapter 2

# Background

Most computer viruses today infects one or more of the Microsoft Windows operating systems. To investigate viruses, it is crucial to have a basic understanding of Windows' executable file format (PE). The PE files contains executable binary machine code, created to run on a CPU.
The 32-bit x86 architecture has been the most common processor architecture used for development world wide and most computer viruses are written for x86. It is therefore necessary to understand the x86 assembler language which is the programming language for the x86 CPU architecture.
The binary machine code found in the PE files must be reversed (disassembled) into meaningful x86 instructions when analysing viruses. It is therefore necessary to learn about common disassembling techniques. Finally, computer viruses and detection methods are described.

## 2.1 PE File format

PE is an acronym for *Portable Executable* and is described in the PE file format specification from Microsoft [19]. A PE file is an executable file for the Microsoft Windows family of operating systems. PE files often have the file extension *.EXE* and a more popular term is "EXE-files", short for executable files. Because binary viruses can exist inside PE files, it is necessary to have a thoroughly understanding of the PE file format.

### 2.1.1 PE File Structure and Headers

A PE file contains many types of headers and several fields inside each header. Figure 2.1 on the following page illustrates the PE layout and is reproduced from [19, page 6]. MS-DOS doesn't support the PE file format, but for backwards-compatibility the MS-DOS header is placed first in a PE file. This header contains *stub* code compatible

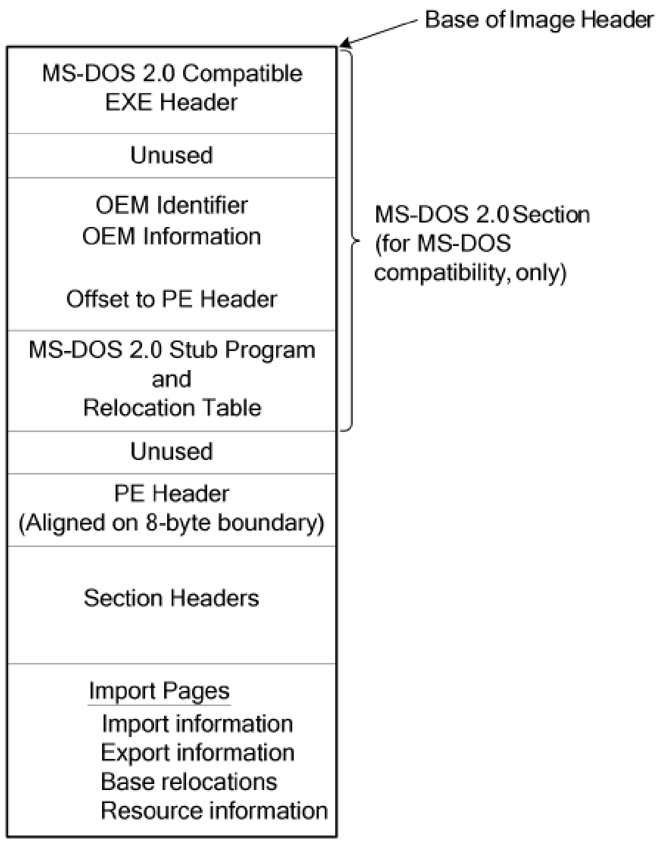


Figure 2.1: Figure illustrating the Microsoft PE Executable format

with MS-DOS, such that when the file is executed in MS-DOS the stub is executed. The stub typically displays “This program cannot be run in DOS mode” message, before exiting. Because valid MS-DOS executables start with the characters ‘MZ’, the MS-DOS header is often termed “MZ header”.

When an OS executing the PE file supports the PE file format, the PE loader of the OS locates the PE header using the “Offset to PE header” field found at position `0x3c` in the MZ header. It is within the PE header and the subsequent *sections* the code and data constituting the actual executable file is found. It it is therefore necessary to have a fundamental understanding of the PE header and the sections. The terms VA and RVA are used extensively in the PE file format and must be described prior to explaining the PE header and sections.

### VA and RVA

All the OSes capable of loading PE files also have support for virtual memory enabling each PE file to be loaded on the same address in memory. The address each PE file is loaded to is termed the *imagebase*, and is a field found in the PE header described next.

**VA** A *VA* is a *virtual address*. VA is the imagebase + the offset of the item in question loaded in memory (for example an instruction). The VA completely describes the location of one item in memory.

**RVA** An *RVA* is a *relative virtual address*. This is the VA - the imagebase, in short terms the offset of the item in question loaded in memory.

#### Important PE header Fields

There are a total of 53 header fields in the PE header. The following list enumerates the fields necessary to recognise in the thesis.

**Characteristics** Information about the attributes of the file. Examples are “the file is executable”, “relocation information has been removed from the file” and “the file is a system file, not a user program”.

**AddressofEntrypoint** When the OS executes a PE file, it needs to know where the code should begin its execution. The instruction at the *entry point* address is the first instruction executed by the CPU when the PE file is run. Usually the entry point is within the code section. The entry point address is stored in the PE header as an RVA.

**Imagebase** The *imagebase* is the memory address of where the first byte of the executable image is loaded. Default imagebase for Windows 95/98/Me/NT/2000/XP is `0x400000`.

### 2.1.2 Sections

A *section* is described in Microsoft’s PE specification as the *“basic unit of code or data within a PE file”*. Each section have a *section header*, and the header contains 10 fields. The 5 most characteristic fields are listed here.

**Name** The name of the section. The names themselves do not identify the sections and different compilers can produce different names. For instance, Borland uses the name “CODE” while GCC use the name “.text” to describe the executable section. The sections are identified by the characteristics.

**Raw size** The size of the section in the PE file.

**Virtual size** The size of the section when loaded in memory.

**Pointer to raw data** Pointer to the raw data in file (where does the section begin in the file).

**Characteristics** Information about the section. Examples are if it is executable, if it can be read from, if it can be written to, and more.

There can be many types of sections. Present in all PE files is a section containing executable code, often named `.text`. If the program has uninitialised data, this is often found in a section named `.bss`. Initialised data, such as global variables, are found in a section often named `.data`.

Having introduced a executable code section in a PE file, it will from now on be referred to as `.text`.

### 2.1.3 Relocation Section

Sometimes the OS cannot load the given PE file at the wanted imagebase. All memory addresses found in instructions in the PE file assumes the use of the default imagebase (`0x400000`). Unless the program is loaded at this default imagebase, the execution of the program would be erroneous. The relocation section, or relocation table, contains pointers to each instruction's RVA/VA. The pointers are called *fixups* and must be updated when executed at a non-default imagebase address. Each fixup points to the memory offset in the instruction, not the instruction itself. The PE loader can then loop through each fixup, adjusting the memory address to the new imagebase before the PE file is executed. The relocation section is often named `.reloc`.

### 2.1.4 Import Section

Practically all executable files uses one or more functions from shared libraries. The shared libraries on the Microsoft Windows platform often have the extension *DLL*. The *import section* contains references to the DLLs and the corresponding functions imported from each DLL. Only the functions used in the PE file are imported. The import section is split in two tables:

1. The *import table*. Each entry in this table contains the DLL reference (for instance `kernel32.dll`) and a pointer to the import lookup table.

2. The *import lookup table*. Each entry in this table contains a function reference within a given DLL (for instance the function `GetModuleHandleA` of `kernel32.dll`)

Since a PE file should execute on all systems it was compiled for (for instance Windows 2000 and Windows XP), the entries in the import lookup table cannot contain specific memory addresses for each imported function. This is because different OSes can load the DLL functions to different memory addresses. When a PE file is executed, the PE loader resolves the imported functions' memory addresses in the OS. Therefore, it doesn't matter if the function `GetModuleHandleA` resides in memory address `0x7C573DFC` or `0xBFF77716`. The PE loader resolves the correct memory address each time the PE file is run.

### 2.1.5 Summary

This chapter were started by pointing out that most binary viruses are written for the Microsoft Windows platform. The PE file format is used to structure the executable code designed to run on the Microsoft Windows platform and an introduction to the PE file format were therefore given in this section.

To understand how a virus works, it is necessary to master the language the executable code found in `.text` is expressed in. This is described in the next section.

## 2.2 x86 Assembler Language

All the information in this section is based on Intel's Architecture Manual Vol 1 [18], Vol 2 A-M [16] and Vol 2 N-Z [17].

The 32-bit Intel processor architecture, commonly known as "x86", has been the most common processor architecture used for software development worldwide. Because of the popularity of the x86 architecture, most viruses are written for x86. To be able to read virus code, the x86 assembler language must be mastered.

Assembler language is the low-level symbolic representation of the binary machine codes understandable by a CPU. It is more or less a one to one mapping between the machine codes and a symbolic representation. Because the machine codes are highly processor-dependent, a program written in assembler language is not portable between different processor architectures. The process of converting assembler language source code into binary machine code is called *assembling*. The process of reversing a block of binary machine code into assembler language representation is called *disassembling*. In most cases the virus source code is not available to a virus analyst. A disassembling of the binary file infected with the virus is therefore the best method of transforming the virus into human understandable operations.

### 2.2.1 Intel Syntax and Examples

This section describes the assembler language dialect used in this thesis. Further, this section gives two examples of assembler language code. These are provided as a motivation for further reading, and to illustrate the main features of the dialect used.

In this thesis, all assembler language listings use the dialect termed *Intel syntax*. A typical instruction in the Intel syntax has the following form:
`instruction destination, source`. Another common dialect is the *AT/T syntax*. This is not used or discussed any further in this thesis.

| # | Instruction |
|---|---|
| 1 | `mov EAX, 2` |
| 2 | `mov EBX, 3` |
| 3 | `add EAX, EBX` |

Table 2.1: x86 Assembler Language Example 1

| # | Instruction |
|---|---|
| 1 | `mov EAX,[input1]` |
| 2 | `add EAX, [input2]` |
| 3 | `mov [output+4], EAX` |

Table 2.2: x86 Assembler Language Example 2

Table 2.1 contains an example of Intel syntax assembler code. In instruction 1, the value 2 is copied into the register `EAX`. A register can be viewed as a temporary storage unit. In instruction 2, the value 3 is copied into the register `EBX`. In the third instruction, the value contained in `EBX` is added to the value in `EAX`. After instruction 3 has executed on the processor, `EAX` holds the value 5 and `EBX` the value 3.

Table 2.2 contains another example of Intel syntax assembler code. In instruction 1, the value found at memory address `input1` is copied into `EAX`. In instruction 2, the value found at memory address `input2` is added to `EAX`. Finally in instruction 3, the value in `EAX` is stored at memory address `output+4`. The square bracket: `[ptr]` found in x86 assembler language is equivalent to pointers in C: `*(ptr)`.

## 2.2.2 Instruction Format

An instruction consist of possible 6 parts: Prefix byte, Opcode bytes, ModR/M byte, SIB byte, Displacement bytes and Immediate bytes. The disassembler converts the six parts into a symbolic representation. To understand the material presented in this thesis it is not necessary to have a complete understanding of the different parts. When an instruction is represented symbolically, the instruction has the following format: `mnemonic operand1, operand2, operand3`.

A *mnemonic is a reserved name for a class of instruction opcodes which have the same function*. For instance the opcode mnemonic `MOV` has several different opcode byte encodings depending on what should be moved where, but the function is identical. From the symbolical representation, there can be from zero to three explicit *operands*. The instruction act on the operands, and operands can be viewed as arguments to the instruction. In addition to the explicit operands, there can also be several implicit operands. The implicit operands are typical "side-effects" of an instruction. For instance, the instruction `push EAX` pushes the content of the register `EAX` onto the stack. `EAX` is an explicit operand and the stack pointer register `esp` is an implicit operand. Each operand in an instruction is a source, a destination or both. A source operand can be any of the following listed here.

- an immediate operand or value encoded in the instruction itself
- a register
- a memory location
- an IO port (direct communication with hardware through a hardwired input/output port)

A destination operand can be any of the following listed here.

- a register
- a memory location
- an IO port

Immediate operands are values given as an operand to the instruction. In the instruction `sub EAX, 13`, the value 13 is an example of an immediate operand.

### 2.2.3 About the Basic Execution Environment

A *basic execution environment* is the resources available for executing instructions found in application- and OS-code. The components of the basic execution environment provides the foundation for executing instructions on a processor and they are listed here.

**Address space** The *address space* is the amount of memory the processor can address. There are possible $2^{32}$ addressable memory bytes in the x86 32-bit architecture.

**Basic program execution registers** Registers are the basic storage units in a processor. There are eight general-purpose registers, one control flag register and one program counter register.

**x87 FPU registers** The floating point unit (FPU) registers are used to do floating point operations.

**MMX registers** The multimedia extension (MMX) registers exist to perform "single-instruction multiple-data" (SIMD) instructions on 64-bit data and are often used in multimedia applications.

**XMM registers** The XMM (extension to MMX) registers are similar to the MMX registers, but are used on 128-bit data.

The registers constituting the *basic program execution registers* are used in all executable files and are explained on the next pages.

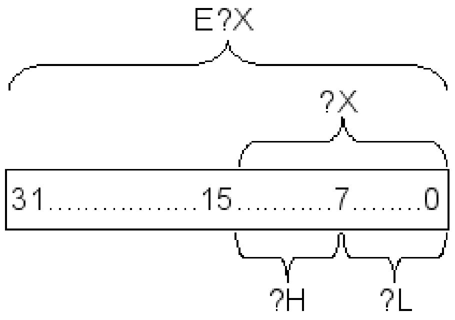


Figure 2.2: Substitute the question mark (?) with A,B,C or D.

### The General Purpose Registers

The general-purpose registers are provided for storing pointers and results from general operations. The general-purpose registers are the 32-bit registers: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EB` and `ESP`. Each of the 32-bit registers 16 least significant bits (lsb), can be addressed by the corresponding 16-bit registers: AX, BX, CX, DX, SI, DI, BP and SP. The 8 most significant bits (msb) of AX, BX, CX and DX, can be separately addressed by the following corresponding 8-bit registers: AH, BH, CH and DH. Also, the 8 lsb of the same 16-bit registers can be addressed by the following corresponding 8-bit registers: AL, BL, CL and DL. Figure 2.2 illustrates the addressing possibilities of EAX, EBX, ECX and EDX.

The Intel manual describes the following uses for each register [18].

**EAX** Accumulator for operands and results data.

**EBX** Pointer to data.

**ECX** Counter for string and loop operations.

**EDX** I/O pointer.

**ESI** Source pointer for string operations.

**EDI** Destination pointer for string operations.

**ESP** Stack pointer. This register contains the location in memory where the last element pushed onto the stack is found.

Note that this is the *intended* usage of each register. Each register can be used freely by the programmer in any type of operation.

#### EFLAGS register

The EFLAGS register contains several *flags*. Each flag is of size 1 bit, and has a fixed position in the EFLAGS register. A flag can be *set*, meaning that it receives the value 1, or *cleared*, meaning that it receives the value 0. Amongst all the flags found in the EFLAGS register, there are seven flags used in regular applications. The flags are listed here.

**Carry Flag** The CF flag is set/cleared depending on different results in arithmetic instructions.

**Parity Flag** The PF flag is set/cleared depending on parity of result.

**Adjust Flag** The AF flag is set/cleared depending on binary-coded decimal arithmetic.

**Zero Flag** The ZF flag is set if result is zero, cleared if not.

**Sign Flag** The SF flag is set if sign of an integer result is negative, cleared if positive.

**Overflow Flag** The OF flag is set if integer result overflowed, cleared if not.

**Direction Flag** The DF flag is used in string operations to tell if loop should process memory addresses in low-high (cleared) or high-low (set) order.

#### EIP register

The EIP register contains a memory pointer to the next instruction to be fetched and executed by the processor. EIP cannot be addressed explicitly, but is manipulated implicit through the following *control transfer instructions*: `JMP`, `CALL`, `RET`, `IRET` and the group of conditional jumps. Conditional jumps, or short *Jcc*s, are a collective term for the different opcodes which transfers the execution flow to the target given in the Jcc instruction when certain conditions are met. Examples are `JZ` (jump if zero flag is equal to 1), `JECXZ` (jump if ECX is equal to 0), `JNO` (jump if overflow flag is equal to 0). There are two possible outcomes when a Jcc is executed: The flow of execution is set to the *fall through branch* (jump not taken) or the *taken branch* (jump taken).

### 2.2.4 Instruction groups

There are a vast number of different instructions, but when grouped the number becomes manageable. The Intel manual [18] unify the instructions in 13 different groups. The following list describes each group.

**Data transfer** Move data between operands.

**Binary arithmetic** Basic binary integer computations on operands.

**Decimal arithmetic** Decimal adjusting.

**Logical** AND, OR, NOT, XOR operations on operands.

**Shift and rotate** Shift and rotate bits in operands.

**Bit and byte** Bit instructions test and modify individual bits in operands. Byte instructions set the value of a byte operand, based on flag status in the EFLAGS register.

**Control transfer** Provide jump, conditional jump, call and return to control the program flow.

**String** Move strings of bytes to and from memory.

**I/O** Move data between the processor's I/O ports and operands.

**Enter and leave** High level procedure entry and exit.

**Flag control** Instructions operating on the EFLAGS register.

**Segment register** Instructions loading pointers into segment registers.

**Miscellaneous** Instructions which cannot be placed in the other groups, such as `NOP` (no operation) and `CPUID` (identify processor).

Each group contains mnemonics performing various tasks, related to their grouping.

### 2.2.5 Summary

This section described the layout of the x86 processor architecture, and gave an introduction to the x86 assembler language. Mastering the x86 assembler language is essential when analysing viruses, because most viruses are written for the x86 architecture.

As previously described, the `.text` section contains the executable code of a program. Converting the bytes in the `.text` section into instructions is done by the disassembler, but the *sequence* the disassembler should process the bytes in, is not necessarily evident. The next section describes the two most common disassembler algorithms: linear sweep and recursive traversal.

## 2.3 Disassembling Algorithms

The process of converting a sequence of bytes into meaningful instructions is called disassembling. A *disassembling algorithm* describes the sequence the bytes should appear in, when given as input to the disassembler. There are two well-established disassembling algorithms: Linear sweep and recursive traversal [33].

## 2.3.1 Linear sweep

Linear sweep (LS) disassembles the byte stream sequentially. LS typically starts at the beginning of the code section and proceeds to the end. LS runs in $O(n)$. The following pseudocode illustrates LS.

**Algorithm 2.3.1:** DISASSEMBLE(*Addr*)

```
while Addr is a valid section address
  do { i=DecodeInstr(Addr)
       output(i)
```

#### Advantages

Compared to recursive traversal, LS's main advantage is simplicity. There is no need to parse the instruction disassembled, and there is no need for an input buffer to store the instructions. Another advantage is that all code within an executable code section is disassembled, because all the bytes in the code section is processed by the disassembler.

#### Disadvantages

LS doesn't provide any logical structure in the instruction output. This is a drawback when an analysis method requires the disassembled instructions to appear in the same order as when executed on a CPU.

Misinterpretation is another disadvantage. If code and data are mixed together in a section, misinterpretation is likely to occur: All the bytes used for data stored in an executable code section, is misinterpreted as code. Without utilising any information about the placement of data and code, it is only possible to detect misinterpretation when an invalid opcode byte sequence, termed *bad instructions*, is encountered. As seen in the Intel instruction manuals, the x86 architecture has many byte patterns decoding into valid instructions and few byte sequences yield bad instructions.

## 2.3.2 Recursive traversal

Recursive traversal (RT) follows the designated flow of the program. RT starts the disassembling at a program's entry point, and follows the flow from there. If a control transfer instruction is encountered, the transfer is followed if possible. RT runs in $O(n)$.

#### Algorithm

The basic algorithm for recursive traversal given in [33] is reproduced here.

| # | Instruction |
|---|---|
| 1 | `mov EAX, 0x4000320` |
| 2 | `add EAX, 0x10` |
| 3 | `xor EAX, 0x14` |
| 4 | `call [EAX+6]` |

Table 2.3: Example illustrating an instruction containing an unresolvable control flow

**Algorithm 2.3.2:** DISASSEMBLE(*Addr*, *instrList*)

```
if (Addr.visited)
  then return
do
  instr = DecodeInstr(Addr)
  Addr.visited = true
  add instr to instrList
  output(instr)
  if (instr is a branch or function call)
    then for each possible target t of instr
           do Disassemble(t, instrList)
    else Addr += instr.length
while Addr is a valid instruction address;
```

## Advantages

RT's main advantage is correct disassembling of a code section with both code and data mixed together. Also, the decoded instructions appear in the same order as when the instructions are executed on a CPU.

## Disadvantages

The disadvantage of RT is when control transfers cannot be resolved. The x86 assembler language contains many addressing forms, and some are only resolvable when executing the instructions. Table 2.3 illustrates the concept of *unresolvable control flow* instructions. The instruction sequence illustrated is a valid x86 instruction sequence. Unless the sequence is emulated or executed on a CPU, it is not possible to determine the value of EAX in instruction 4. Instruction 4 is therefore an instruction containing an unresolvable control flow. Because the destination of the control transfer instruction cannot be resolved, the RT algorithm cannot disassemble the byte stream found at the destination. Schwarz et al. lists the two possible consequences when this situation occurs. *failure to disassemble some reachable code* (if control transfer instruction is not followed) or *erroneous disassembling of data* (if some algorithm makes an erroneous guess of the jump destination).

### 2.3.3 Summary

In this section, the two most common disassembling algorithms have been explained and their advantages and disadvantages have been discussed. The tools needed to understand polymorphic viruses have now been acquired, and the following section is devoted to computer viruses.

## 2.4 Computer Viruses

This section is about computer viruses. The layout is structured as follows. First, terms used in this and subsequent sections are defined. Second, a brief history illustrating the evolution of both virus technology and the motivation for writing the viruses is given. Finally, polymorphic viruses are introduced and techniques for achieving polymorphism are discussed.

### 2.4.1 Definitions

Peter Szor defines a computer virus in his book “The Art of Computer Virus Research and Defence” as follows [38].

> *A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself.*

The definition disagrees with the common belief that a program needs to do unwanted, explicit damage to be termed a virus. The definition contains <u>all</u> programs intentionally spreading themselves without any acknowledgement from the user.

There are many different types of malware. Aycock lists several types in his book “Computer Viruses and Malware” [13]. Amongst the types listed are: Viruses, worms, trojans, backdoors. Viruses and worms are two types of malware often confused with each other, and the two types are explained here.

#### Computer viruses

Computer viruses, or viruses, hide inside hosts the same way biological viruses hide inside cells. To replicate, viruses need new hosts. Viruses infect files on the host machine, and they spread when users exchange files, for instance through USB-pens or a shared directory in a P2P application. Note that mapped network drives, for instance an NFS mount or a mapped UNC path in Windows, might provide the virus an entrance through the network. Harley et al list in “Viruses Revealed” [24] three parts a computer virus consists of.

1. Infection mechanism - How the virus spreads.

2. Trigger - If the virus is to deliver the payload or not.
3. Payload - What the virus does except from spreading. Payloads can be intentional or accidental (bugs in virus code, damaged in network transport, etc).

**Computer worms**

Computer worms, or worms, are often confused with viruses. Worms are similar to viruses except for two main differences. The differences are listed here.

1. A worm is not parasitic, i.e. it doesn't require a host to infect a new target. It is a *stand-alone application* performing self-replication. This is often done by exploiting buffer overflows in different Internet applications.
2. Worms spread to new hosts through the network via different network protocols (for instance the TCP/IP protocol).

As stated in the introduction to chapter 1, the term "virus" is used for describing computer viruses, trojans, worms and other types of malware in the thesis. The specific terms are only used when the context requires separate identification of the terms.

### 2.4.2 History

The term "computer virus" as it is used today, is attributed Len Eidelmen who defined the term in 1983 [48]. However, as early as in 1949, John Von Neumann devised a theory of a program capable of replicating itself as long as a computer can hold information in "memory" [31].

The first acknowledged computer virus appearing on regular users computers (often termed "in the wild"), was "Elk Cloner" [44]. Elk Cloner was written for the Apple II computer and displayed a poem on every 50th boot. In 1986, "(c) Brain" appeared. Brain is considered the first virus compatible with the IBM PC. It was written by two brothers from Pakistan, and it spread over the whole world [42].

Later, more malicious viruses appeared. "CIH" also known as "Chernobyl" was one of the most direct damage-dealing viruses known [43]. The first version was seen in the wild in 1998. CIH fills the first megabyte of the hard drive with zeroes, and it also tries to write to Flash BIOS. If the latter is successful, the Flash BIOS needs to be replaced/rewritten for the computer to work.

Isolated, most other viruses do not cause an equal amount of damage to the computer as the CIH virus did. However, they can cause more damage world-wide because of their ability to spread. The damage can be anything. For instance, the amount of spam is something regarded as damage because spam contributes to slowing the Internet down. In 1999 the "Melissa" worm caused a damage estimated to $80 million dollars. In 2001, the "Code Red" worm caused a damage estimated to $2 billion dollars [31]. In

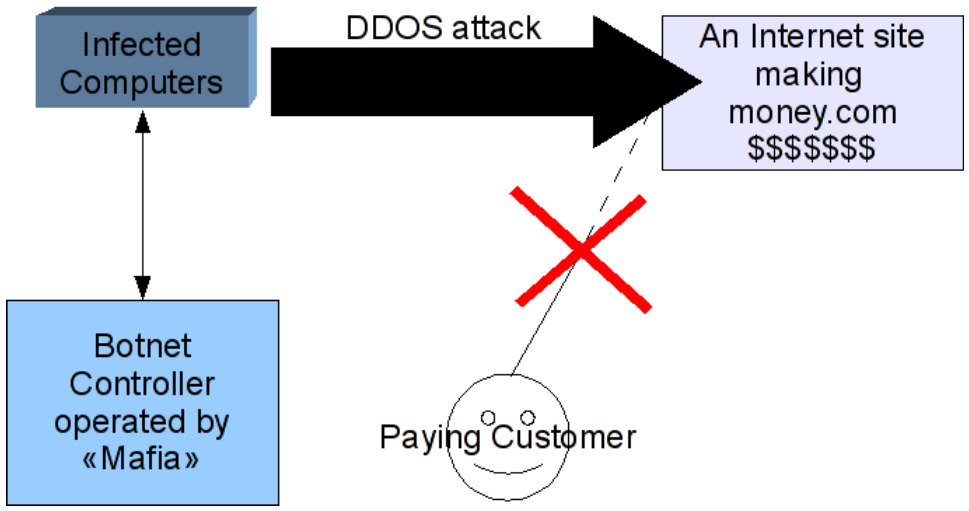

Infected Computers
DDOS attack
An Internet site making money.com $$$$$$$
Botnet Controller operated by «Mafia»
Paying Customer

Figure 2.3: Blackmailing scheme

2003, the estimated amount reaches an even higher level. A UK-based digital risk firm called "mi2g" estimates the worm MyDoom to have caused a damage equal to $38.5 billion dollars. However, the correctness of this estimate and others are discussed by Vmyths, "a site dedicated to erase the computer virus hysteria" [11].

## Change of Motivation

Regarding viruses, the situation of today has changed. There has been a massive explosion in the number of viruses created and released in the wild. The reason for this increase lies in the change the Internet has undergone for the last decade. The Internet has changed from being an information provider, to the world's biggest marketplace. As mentioned in the introduction, the virus business is organised in mafia-like structures. The reason for organised crime to enter the virus scene is simple: money. The scheme in figure 2.3 illustrates another reason to why the mafia-analogy has been introduced. Assume a "mafia" has control over a vast number of infected computers, termed a *botnet* (net of robots). First, a virus is written to gather bots to the botnet. When enough bots are under the mafia's control, the owners of the site making legitimate money are contacted by the mafia. The mafia threaten them to pay protection money. If they do not comply, a distributed denial of service (DDOS) attack is launched on the site. The reason for launching a DDOS attack on the site, is that paying customers and others will in the worst case be unable to use the site at all, because it is too busy serving the bots requests. If the DDOS attack lasts for days, the paying customers grow tired of not being able to use the service they are paying for, and head over to other sites providing similar services instead. The number of infected computers vary between the researchers estimating the sizes of botnets [40]. Some researchers estimate the size to be at maximum a few thousand bots, while others estimate a botnet to reach 350000 bots. It is however clear that botnets exist, and are used in organised crime.

The scheme illustrated in the figure is only one of many money-making schemes requiring viruses to be installed on computers. *Spamming* and *phishing* are other techniques used to make money. For instance, the *Bagle* worm opens TCP port 6777, providing remote access to the infected computer [21]. This can be used to install various programs on the infected computer and harvest sensitive information such as VISA card numbers. Another "interesting" extortion scheme is *ransomware* [45]. When the virus is executed, it encrypts files present on the computer, such as DOC and JPG files. The user receives a message stating that in exchange for money, the user will receive a decryption tool which recovers the previously encrypted files. "Gpcode" is an example of such a virus [41].

All the organised crime business models require viruses infecting computers without being detected. The virus writers are therefore constantly developing techniques for avoiding detection. One of the major techniques for making a virus "undetectable" is code changing.

### Code Changing viruses

The first polymorphic virus appeared in 1990, and was called "1260" [12]. It was created by Mark Washburn as a research virus to demonstrate for the antivirus companies that string-based pattern scanners fail, when such viruses are encountered. Each infected file increases with 1260 bytes (hence the name). The payload is encrypted with two sliding keys different in each infection. The decryptor contains do-nothing instructions inserted between the real decryptor instructions. The virus was never released in the wild.

The "Marburg" and "HPS" viruses were the first to use 32-bit polymorphic engines [23]. "Appariton" was the first metamorphic virus (See section 2.4.5 on page 36 for definition). In 2001, Simile and Zmist made their entrance. These are two of the most complex poly- and metamorphic viruses known. The metamorphic engine of Simile contains over 12000 lines of Intel x86 assembly code[13]. Zmist is chosen as the case study in this thesis and chapter 3 on page 47 contains a thorough analysis of Zmist.

### Server-side Polymorphism

Today, the amount of new file-infecting viruses is low compared to other types of malware such as worms and trojans. The polymorphic techniques established in the file-infecting polymorphic viruses are however still valid. Commtouch's "Malware outbreak trends" for the first quarter of 2007 [15] reports of an increase in "server-side polymorphic malware". This type of malware is created by a metamorphic engine residing on one or more servers. When enough worms have been created, they are launched simultaneously on the Internet. The following quote is taken from Commtouch's report.

> *Rapid simultaneous release of massive amounts of distinct variants allows each variant to do maximum damage before a suitable signature or heuristic can provide protection.*

The type of attack described in the quote is known as a "zero-hour" attack, because of the rapid spread before it is detected by antivirus software. Because the worms are pre-mutated, there is no need for mutation engines present in the worm-files. When the mutation engine is not present, the virus analysts cannot reliably analyse the algorithm which produced the mutated worms. The virus analyst can only acquire samples and construct a detection algorithm based on the samples. The "Storm Worm" is an example of malware effectively utilising server-side polymorphism. Between 18th January 2007 and 31th January 2007, there were observed at average 3824 distinct variants of the Storm Worm per day. On 28th January the maximum of 7893 distinct variants were observed [14]. The payload of the Storm Worm is to generate bots that can be used to send spam emails and participate in botnets.

Having illustrated the use of polymorphism, polymorphic viruses and techniques for achieving polymorphism will be described. Polymorphic viruses are built upon mechanisms developed in *encrypted viruses* and this type is therefore described prior to polymorphic viruses.

### 2.4.3 Encrypted Viruses

A host with an *encrypted virus* infection contains two parts: a *decryptor* and an encrypted *engine* [13]. When the host is executed, the decryptor receives control in some fashion, decrypts the encrypted engine and runs it. The decrypted engine is responsible for delivering the payload and infecting new hosts. Because the encryption key vary for each new infection, a static signature cannot be used to detect the encrypted engine. Except for the key used, the decryptor is static in each infection, revealing itself for traditional *signature-based scanning*. Signature-based scanning will be explained in 2.5.1 on page 41.

#### About the Decryptor

The decryptor's main task is to decrypt an encrypted engine, transfer control to the engine and possibly control back to the host code after the engine has infected new hosts. It is typically implemented as a loop inside the host with a control transfer instruction to the engine after the loop. The size of the decryptor can range from under ten to theoretically an infinitely number of instructions.

#### About the Engine

The engine is the main part of the virus. The engine is responsible for finding new hosts, encrypt itself, create a decryptor which successfully manage to decrypt the engine, and inserts the engine and decryptor into the hosts.

### 2.4.4 Polymorphic Viruses

A *polymorphic* virus contains the same two parts as an encrypted virus: a decryptor and an encrypted engine [13]. The difference between a regular encrypted virus and a polymorphic virus lies in the decryptor: The decryptor of a polymorphic virus is syntactically different for each infection. The engine is capable of constructing a seemingly unlimited amount of different decryptor versions. The decrypted engine is the same each time, i.e. the length and syntax of the engine code is constant in each infection. Because of the “unlimited” amount of decryptors, the traditional signature-based scanning cannot be used since that would require an “unlimited” amount of signatures.

Webster & Malcolm uses the term *allomorphic* in [27]. An *allomorph* is any two generations of the same code-changing virus that differ syntactically. Two allomorphic code segments are semantically equal, but not syntactically equal. Polymorphic viruses have allomorphic decryptors. There are many ways of creating allomorphic code and section 2.4.7 on the facing page explains common methods for achieving polymorphism.

### 2.4.5 Metamorphic Viruses

*Metamorphic* viruses are code changing viruses similar to polymorphic viruses. The difference is that there is *no encryption* of the virus engine, and hence no decryptor. The engine itself avoids detection by changing each time it replicates. Two engines of the same metamorphic virus is therefore allomorphic

#### About the Engine

The engine of a metamorphic virus is by Harleys definition similar to the polymorphic engine. The techniques needed to create new instances of itself, are the same as the techniques used in polymorphic viruses when creating new decryptors (see section 2.4.7 on the next page for a description of the most used methods). The difference is that the methods are applied on the whole engine, instead of constructing a new decryptor each time the virus infects a new host. This self-modifying code should be allomorphic for each time the virus infects, and a small coding error could cause an early death of the virus when the error evolves into new variants.

### 2.4.6 Notation usage for poly- and metamorphic viruses

Polymorphic and metamorphic viruses are similar considering detection methods. Both types change appearances for each new infection. It is therefore reasonable to continue using “polymorphic viruses” to describe both polymorphic and metamorphic viruses, unless the context requires separate identification of the two types.

### 2.4.7 Achieving Polymorphism

In this section, the most common techniques used to achieve polymorphism are explained. The essence in all the types listed, is to change the syntax and not the semantics, i.e. make allomorphic code segments. The techniques listed here, are a summary from techniques listed in Aycock [13].

**Instruction Equivalence** This technique is based on the fact that different sequences of instructions achieve the same. Consider the following code.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `sub EAX, EAX` |
| 2 | `add [mem], EAX` | `xor EAX, 4` |
| 3 | | `add [mem], EAX` |
| 4 | | `add [mem], EAX` |

Table 2.4: Instruction Equivalence

In sequence 1, the value 8 is added to the content of memory address `mem`. In sequence 2, the value 4 is added twice to the content of `mem`, achieving the same as sequence 1. The two code segments are therefore allomorphic.

**Instruction Reordering** If two or more instructions do not have any dependencies, their execution order do not impose on the overall program state. Consider the following code.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `mov EBX, 4` |
| 2 | `mov EBX, 4` | `mov EAX, 8` |
| 3 | `add EAX,EBX` | `add EAX, EBX` |

Table 2.5: Instruction Reordering

The instruction reordering must take into account the dependencies between the registers. For instance, the two sequences would not be allomorphic if the `add` instruction appeared before one of the `mov` instructions in only one of the sequences.

**Register Renaming** This technique changes the registers used in the instruction sequence. Consider the following code.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `mov EDX, 8` |
| 2 | `mov EBX, 4` | `mov ECX, 4` |
| 3 | `add EAX,EBX` | `add EDX, ECX` |

Table 2.6: Register Renaming

The two code sequences are allomorphic, but the instruction encoding is different because of the different registers used.

**Reordering Data** This technique changes the location of memory variables. Consider the following code.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `mov EAX, 8` |
| 2 | `add [0x402008], EAX` | `add [0x403004], EAX` |

Table 2.7: Reordering Data

The memory location is changed and the instruction encoding is different.

**Jump Islands** Jump islands are linked together with unconditional jumps (`JMP`). Consider the following example.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `mov EAX, 8` |
| 2 | `mov EBX, 4` | `jmp insn 3` |
| 3 | `add EAX,EBX` | `mov EBX, 4` |
| 4 | | `jmp insn 5` |
| 5 | | `add EAX, EBX` |

Table 2.8: Jump Islands

Because the instructions are scattered in sequence 2, the syntax is changed compared to sequence 1.

**Junk Code** Junk code is code inserted to obfuscate and confuse without altering the semantic of the program. The state of the original code is equal to the state from the original + junk code. Consider the following example.

| # | Instruction Seq 1 | Instruction Seq 2 |
|---|---|---|
| 1 | `mov EAX, 8` | `mov EAX, 8` |
| 2 | `mov EBX, 4` | `mov ECX, EAX` |
| 3 | `add EAX,EBX` | `mov EBX, 4` |
| 4 | `mov ECX, 2` | `xor ECX, EAX` |
| 5 | | `add EAX, EBX` |
| 6 | | `sub ECX, EBX` |
| 7 | | `mov ECX, 2` |

Table 2.9: Junk Code

In sequence 2, the `ECX` register is used for junk code. Note that in sequence 2, instructions 2,4 and 6 do not contribute to the output of the program. Sequence 1 and 2 are therefore allomorphic.

All the different techniques listed can be combined, increasing the level of polymorphism employed in a virus.

### 2.4.8 Summary

This section gave an introduction to computer viruses and their history from the beginning to the present situation. The main focus has been on polymorphic viruses and to describe the techniques for achieving polymorphism.

Polymorphic viruses are advanced and so are their detection methods. The next chapter is devoted to explaining the established detection methods.

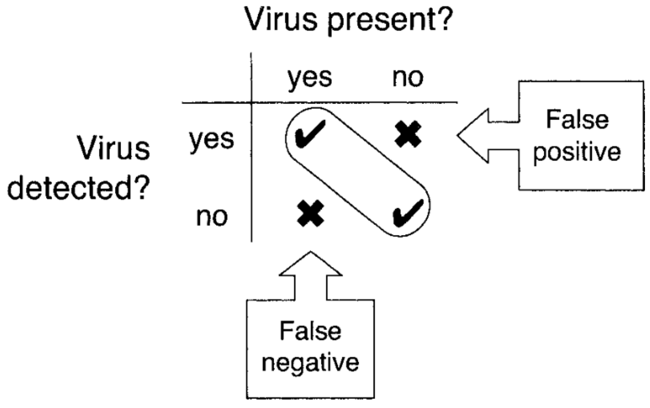


Figure 2.4: Figure illustrating FP, FN, TP and TN

## 2.5 Computer Virus Detection

Aycock [13] divides antivirus detection techniques into two parts: static and dynamic methods. The static methods attempt to detect the presence of a virus without executing any code. Vice-versa, the dynamic methods execute the code and attempt to detect the presence of a virus by properties only present when executing the virus. Szor [38] separates antivirus detection into algorithmic methods (static), emulation methods (dynamic) and heuristic methods (both static and dynamic).

### Outcomes in Virus Detection

Figure 2.4 (reproduced from Aycock [13, page 54]) illustrates the four possible results considering virus detection. The outcomes are independent of the method used for detection. In the figure, a perfect detection method would always return with results laying within the circle on the diagonal. The terms false positive, false negative, true positive and true negative are used to describe the different results.

The term *true positive* (TP) is used to describe a virus is present in a file which is detected. If a virus is absent in a file and the virus is not detected, the term is *true negatives* (TN). The term *false negatives* (FN) is used when a scanner reports a file as clean, but there is a virus present. To describe a file reported by a scanner as infected but the file is clean, the term *false positives* (FP) is used. All algorithms strive to reduce the number of false negatives and false positives, and to decrease the time spent verifying true negatives. If a file actually contains an infection (true positive), the time spent verifying the virus' presence is not critical to the user. It is the time spent on deciding whether a file is *clean* or not, that should be minimised. Unfortunately, there is a vast number of different viruses. Using elaborate detection methods on all files present on a computer, would result in unacceptable time delays. The use of filtering techniques such as *geometric properties* (GP), is a common method for sorting which viruses the antivirus product should scan for. The following paragraph describes GP.

GP can also be used as a static heuristic, reporting whether a file is "suspicious" and would require a more complex scan (often termed *deep scan*).

#### Geometric Properties

Geometric properties are the alterations a virus does to the file structure. Szor lists common geometric properties found in viruses in [38]. Execution of code starting in the last section in the PE file is an example of a geometric property. This would imply that a virus has prefixed its code to the PE file. This is detected by following the entry point in the PE header and examine if it points to an address found in the last section defined in the PE header. Another common geometric property is the use of not-normal names for code section. For instance, it is suspicious that a section named `.reloc` has the EXECUTABLE characteristic set. Values considered not-normal in the PE header is another geometric property used to indicate the presence of a virus. For instance, there is a "checksum" PE header field containing a checksum of the PE file. If this checksum is not correct (i.e. the virus doesn't update this checksum), this can be used to classify a file as suspicious.

#### Layout

It exists far to many detection approaches to describe all in this section. Selected detection methods from the different categories are provided instead, to illustrate the level of complexity and diversity employed by the different methods. The material is presented using Szor's categorisation. First, three algorithmic approaches are described. Then, detection using emulation is described. Last, one example of a static heuristic method and one example of a dynamic heuristic method are described.

### 2.5.1 Algorithmic Approaches

The Merriam-Webster's online dictionary [30] defines an algorithm as *a step-by-step procedure for solving a problem or accomplishing some end especially by a computer*. An algorithmic antivirus approach searches especially for a known virus with some knowledge of the virus' properties. How the search is conducted and what the properties of the virus are, are different with each algorithmic approach.

#### Traditional antivirus methods

The established method for detecting viruses in the past has been *signature based scanning*[13]. The scan itself has been employed *on-demand*. A *signature* consist of the sequence of bytes necessary to identify a virus. As the viruses became more complex, the signature based scanning was not suited to detect the most advanced viruses. Signatures with *wildcard* support and *on-access* scanners (scanning on the fly) was

```
mov [edi], IMM ⟶ push ECX
                 mov ECX, IMM
                 mov [edi], ECX
                 pop ECX
```

Figure 2.5: Example of term rewriting

developed to battle the new viruses. When polymorphic viruses started to appear, the antivirus companies were forced to find new techniques instead of upgrading the old ones. However, traditional signatures are still used and needed, for instance in emulation-based detection.

From now on in this thesis, the term *signature* is not necessarily related to byte-scanning, but to something that defines a virus' property relative to the method. An example is the reduced engine-signature together with the language normalising detection method. This detection method is described after the part about X-raying.

### X-raying

X-raying is a cryptographic detection method [38]. Viruses employing encryption and polymorphism contains a static encrypted engine. Many viruses containing an encrypted engine, encrypts the engine using a single instruction such as `ADD`, `SUB`, `XOR` and `ROL`/`ROR` (rotate left/right). A X-raying method decrypts the engine, and uses traditional byte-scans to verify the presence/absence of a virus. When the engine is encrypted using different keys, the X-raying method must enumerate all possible keys.

When the placement of the encrypted engine is different with each infection, X-raying is ineffective. X-raying starting from every byte sequence present in the file in question would yield an intolerable high scanning time.

### Language Normalising

Language normalising reduces the polymorphic code to a canonical form. Because polymorphic decryptors can be short, language normalisation is in general better suited for detection of metamorphic viruses. In [28], Rachit normalises code with *term rewriting*. Term rewriting is a way of replacing an opcode sequence with another semantically equal opcode sequence. An example given is in figure 2.5. To detect an instance of a metamorphic virus, the code fragment in question must be term-rewritten to normal form and compared to an a'priori analysed engine.

A more mathematical approach is described by Webster & Malcolm in [27]. In this article, OBJ is used to prove equivalences between different allomorphic code fragments. OBJ is described as *an algebraic specification formalism and theorem prover based on order-sorted equational logic*. When rules have been defined for each opcode type used in two code fragments, OBJ can reduce each fragment and prove/disprove if the reduced output is equivalent. One example given in the article is a proof that two allomorphic generations of Zmorph.A manipulate the stack equivalently in both generations. To detect an instance of a metamorphic virus, the code fragment in question must be reduced and compared to an a'priori reduced engine. The a'priori reduced engine would then act as a signature.

The major drawback with language normalising is a high computing cost. Reducing a block of instructions to normal form and then comparing these to previously analysed forms requires a CPU time most users would find intolerable if applied on every file a virus scanner scans.

### 2.5.2 Emulation

The material about emulation is taken from Szor [38]. An *emulator* is an application simulating a CPU with an execution environment. Detection of a virus with emulation requires an emulator combined with an algorithmic scanning method such as traditional byte-sequence scanning. A polymorphic virus can then be detected by examining the contents of the emulated memory when executing a file inside the emulator. By actively scanning the memory for known patterns of a polymorphic engine, the presence/absence of the virus in the given file is determined.

There are two major drawbacks considering emulation. The first is the time spent emulating files. Because emulating is slower than actually executing a program on the CPU, this requires the file in question to be processed slower than if it actually were run. The second is that only one path of execution is traversed for each emulation. Assume a program has three Jcc instructions, each branch taken target pointing to a block of code not reachable by any other of the Jccs. Each fall through branch points to the next Jcc. This results in four different paths of execution. To ensure that there are no virus present in the file, the emulator would need to traverse all four paths.

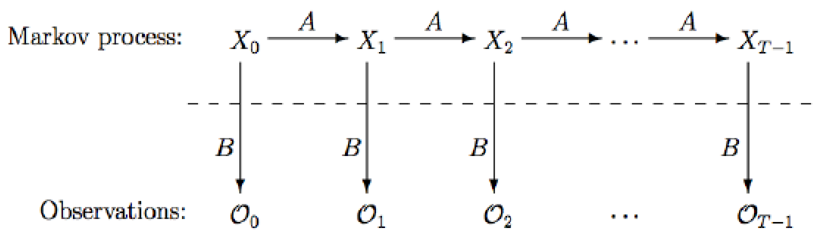
The running time scales badly with the number of Jccs and emulators are therefore seldom used to detect viruses alone. However, emulation is a powerful tool in combination with other scanning methods.

### 2.5.3 Heuristic Methods

There are two different types of heuristics: *static* and *dynamic* heuristics. Static heuristic analyse code fragments without running them, dynamic is vice versa. One common property is their ability to detect unknown viruses, because the heuristics looks for "virus-like" behaviour. The common problem with the heuristic methods is the possible large amount of false positives reported.

#### Machine Learning

An example of a static heuristic method is the methods falling under the *machine learning* group. *Hidden Markov Model* (HMM) is such a machine learning technique



$$\begin{tikzcd}
X_0 \arrow[r, "A"] \arrow[d, "B"'] & X_1 \arrow[r, "A"] \arrow[d, "B"'] & X_2 \arrow[r, "A"] \arrow[d, "B"'] & \cdots \arrow[r, "A"] & X_{T-1} \arrow[d, "B"'] \\
\mathcal{O}_0 & \mathcal{O}_1 & \mathcal{O}_2 & \cdots & \mathcal{O}_{T-1}
\end{tikzcd}$$

t = number of states, $X_t$ = states, A = state transition probability distribution, B = observation probability distribution, $O_t$ = observation sequence

Figure 2.6: Generic Hidden Markov Model figure

and is described in this section. Wong uses HMMs to detect metamorphic malware in [49]. HMMs are well suited for statistical pattern analysis and have been used in many fields, for instance speech recognition and biological sequence analysis.

A HMM is a *state machine* containing the assumption that there is a *Markov process* which cannot be observed directly. A process is a Markov process if the conditional probabilities of future states of the process depends *only* on the current state, and not past states. There are three main components of an HMM: the *states*, the *observations* and the *transitions*. The states represent some unknown or *hidden* features of the input data. Each state has a probability distribution for a set of observation symbols and the transitions between the states have fixed probabilities. In figure 2.5.3 (reproduced from Wong [49, page 28]), a generic HMM is illustrated.

HMMs can be trained to detect if two generations are related. First, the HMM model is set up with a fixed number of hidden states. Wong experimented with two to six states. The HMM is trained so that the probability for observing the average metamorphic engine is maximised. This is done by feeding the opcodes of several generations of metamorphic viral code to the HMM. The opcode sequences corresponds to the observations in the HMM. What the states represent when the HMM is trained is not directly observable. However, it is possible to obtain information on the Markov process *indirectly* by the means of observations and their probabilities. An example is English text which is fed to a two stated HMM, as done by Wong et al. in [35]. Examining the observations and their probabilities in the different states, they found that the two states corresponds to vowels and consonants. However, it is important to understand that to detect new generations of metamorphic code, the states need not to be *interpreted* like in the example.

To detect new generations of metamorphic code, the sequence in question is fed to the HMM. The HMM computes the probabilities for that the sequence is similar to a previously trained metamorphic engine. If the probabilities are over the trained threshold, the sequence is reported as an allomorphic version of the trained metamorphic engine.

There are several drawbacks with using machine learning in general, including HMM. First, as it is with most advanced methods, the computing cost is high and this results in a slow virus scanner. As mentioned in the conclusion of language normalising, this is something not tolerated by most users. Another drawback is that heuristic is prone to reporting false positives or false negatives, depending on the thresholds used.

### Behaviour Blockers

A *behaviour blocker* (BB) monitors a running program in real time. BB is a member of dynamic heuristic. A BB has a set of rules, defining what should be allowed and what should not be allowed. For instance, displaying question boxes on the screen should be allowed while a search for all EXE files on a hard-drive and opening them with read/write permissions should not be allowed. Signatures can be developed based on a virus' action pattern. A simple example would be when the files `c:\windows\system32\cmd_.EXE` and `c:\windows\000.EXE` are created, the BB can detect this and conclude that virus x is infecting the system.

As illustrated, the virus has to actually run on the computer to identify it using BB. It would be better if the virus could be detected *before* it was executed on the computer. Therefore, BB is often used in combination with emulators and other methods to execute the virus safely. This technique is often called *sandboxing*.

## 2.6 Summary

This chapter contains one of the two parts constituting the first research goal posed in the introduction, namely to "learn about polymorphic viruses through a case study". This chapter provides a foundation for studying polymorphic viruses in detail. Because most viruses are written for the Microsoft Windows platforms, the PE file format was explained in section 2.1 on page 19. The PE file format is the executable file format used in Microsoft Windows platforms. Inside the PE files, there are executable code (including the code of a possible virus). To understand this code, the x86 assembler language was necessary to master and an introduction was given in section 2.2 on page 23. The instructions found inside the code section of a PE file can be analysed in many different sequences. The disassembling algorithms described in section 2.3 on page 28 are two established methods of analysing the sequences. Having established a basic foundation for understanding computer viruses, section 2.4 on page 31 introduced viruses and explores polymorphism in detail. Finally, counter-measures were discussed in section 2.5 on page 40 which is about computer virus detection.

Having provided a foundation for examining polymorphic viruses, the next chapter is about the case studied in the thesis: the poly- and metamorphic virus Zmist.

# Chapter 3

# Case Study

The first research goal posed in section 1.2 on page 16 states the following: *Learn about polymorphic viruses through a case study*. Further, it was extended to include detection of the case study. In this chapter, analysis of the virus Zmist is presented as well as a developed detection method of Zmist. Szor & Ferrie has written a paper containing an analysis of Zmist, termed “Zmist opportunities” [39]. This paper is one of two sources used in this chapter. The second source of information is results from my own research on Zmist. Section 3.2 on page 51 is an introduction to Zmist and is primarily based on Szor & Ferrie’s findings. Sections 3.3 on page 56 and 3.4 on page 62 contains respectively an analysis, and a detection of the polymorphic decryptor. Both sections represents results of my own research.

A disassembler termed *Gdasm* was specifically developed to aid the research conducted in this thesis. Aside from having been a tool used to gain knowledge about Zmist, Gdasm provides the foundation for the Zmist detection algorithm developed.

The disassembling output of the files referred to in this section, is found on the CD described in appendix A on page 125.

## 3.1 Case Study Discussion

### 3.1.1 Choosing a Case

Choosing a virus for the case study was done in collaboration with Norman ASA. The following argument provides a background for the choice of the virus *Zmist* as a case study.

Zmist is superior to other polymorphic viruses when considering the level of complexity.

- Zmist is a polymorphic- *and* metamorphic virus. Zmist is one of few viruses which is both poly- and metamorphic, and utilises most of the techniques for achieving polymorphism described in section 2.4.7 on page 37.

- Zmist is the only virus known to utilise the entry point obscuring technique termed: *code integration*.

**Code Integration and EPO**

Aycock [13] describes the different techniques a virus use to insert itself into host files and *code integration* is described as the most advanced type of insertion method. The host code must be moved away to make space for the virus. Branch targets, memory references and data location references must be updated to point to the new location of the original host code which was moved away. Aycock states that this technique is rarely seen and the only known virus fully employing it is Zmist. Because there is no sign of where Zmist is placed inside the virus, code integration is often referred to as the perfect entry point obscuring (EPO) technique. EPO is a term used to describe that the entry point of the virus is more or less random, i.e. the entry point field in the PE header is not modified to point directly to the virus code. The starting point of the virus is obscured because it cannot be directly observed in the PE header and finding this starting point requires an increased scanner effort.

The polymorphism, metamorphism and the unique entry point obscuring technique, made Zmist the obvious choice of case study. Szor & Ferrie confirms the complexity of the virus:

> *Many of us will not have seen a virus approaching this complexity for a few years. We could easily call Zmist one of the most complex binary viruses ever written.*

In the virus world, Zmist is fairly old. Because of its high complexity it is still used as an indicator whether an AV product contains the technology needed to detect polymorphic viruses or not. AV comparatives is an Internet site dedicated to “Independent comparatives of Anti-Virus Software” [20]. Their *On-demand comparative* report from February 2008 shows that 7 of 14 manage to detect 100% of the samples of Zmist.B. 8 of 14 manage to detect 100% of the Zmist.D samples. Because many antivirus companies (including Norman ASA) does not detect Zmist today, an algorithm for detecting Zmist would therefore have real-world impact. This motivates further the choice of Zmist as a case study.

### 3.1.2 Methodology

To understand Zmist, the learning process was divided in two phases.

- First, read up on Zmist and polymorphic techniques in general.
- Second, investigate Zmist in an iterative trial by error fashion, using appropriate tools.

The results of the first phase is found both in the background chapter and in the introduction to Zmist found in section 3.2 on page 51. The results of the second phase is primarily found in section 3.3 on page 56.

### 3.1.3 Tools used in Case Study

The following tools were used during the case study:

- VMware Workstation, version 6.0.3.
- IDA Pro - The Interactive Disassembler, version 5.0.880.
- NAP - Norman SandBox Analyzer Pro, version 1.02d.
- Gdasm - General Disassembler (made specially for the thesis).

#### VMware Workstation

VMware Workstation [9] is a virtual computer capable of running multiple virtual machines on one host computer. A virtual machine is essentially a CPU emulator, capable of emulating everything that is designed to run on a CPU, inclusive an operating system. From the homepage VMware describes their products as follows.

> *The VMware approach to virtualization inserts a thin layer of software directly on the computer hardware or on a host operating system. This software layer creates virtual machines and contains a virtual machine monitor or "hypervisor" that allocates hardware resources dynamically and transparently so that multiple operating systems can run concurrently on a single physical computer without even knowing it.*

There are many benefits of using virtual machines in conjunction with analysis of viruses. A virtual machine can easily be cloned from a clean master copy. When a virus is executed inside a virtual machine, it is easy to reset the machine to a clean state. The infected machine can be deleted, and a clone is created from the clean master copy. This can be done in minutes, compared to the hours spent installing and configuring an operating system.

Because the cost of running a virus is very low in a virtual environment, it is easy to create new virus samples. This is beneficial when analysing polymorphic viruses, because many samples provides a better insight on how the polymorphism is employed in the virus. Many samples are also needed to increase the trustworthiness of an eventual detection algorithm.

**IDA Pro**

IDA Pro [2] is a highly sophisticated disassembler. It is regarded as one of the most advanced disassemblers available. The following quote is taken from the homepage of IDA Pro.

> *IDA Pro combines an interactive, programmable, multi-processor disassembler coupled to a local and remote debugger and augmented by a complete plugin programming environment.*

**Norman SandBox Analyzer Pro**

Norman SandBox Analyzer Pro (NAP) [6] is a tool developed specifically for analysing suspicious PE files. It is a controllable emulator/debugger, where the PE file in question is run in the SandBox OS. NAP provides a view of loaded libraries, threads, created sockets, registers, stack contents and tons of other information interesting to a virus analyst. It is possible to emulate instruction for instruction and set breakpoints based on various criteria. From the homepage of Norman, NAP is described as follows.

> *When working with Norman SandBox Analyzer Pro you can use an extensive list of parameters enabling you to analyse and manipulate the emulation, extend emulation cycles, etc. You will also be able to explore files and changes made to the simulated SandBox OS to get the full view of the impact executing the respective file would have had if it was run on a "real computer".*

### 3.1.4 Gdasm

The first research goal states that the case study should be detected. It was therefore necessary to have a disassembler supporting a plugin environment in which the detection algorithm could be written. An alternative to developing a disassembler from scratch, was to use the programmable plugin environment found in IDA Pro. However, writing a disassembler would increase the understanding of the PE file format and the x86 assembler language, and it was decided that the disassembler should be programmed from scratch.

*Gdasm* is an abbreviation for "general disassembler", and is written in C. Gdasm consists of a PE parser and the RT algorithm. Because of the weaknesses in the linear sweep algorithm described in 2.3.1 on page 29, the recursive traversal algorithm (RT) described in 2.3.2 on page 29 were chosen as the preferred disassembling algorithm. Note that the actual process of disassembling, i.e. converting bytes to instructions is not done by Gdasm, but by a library called *libdisasm* (version 0.23) [4].

Figure 3.1 on the facing page illustrates the components and the general work flow of Gdasm. The following list describes the states in the figure.

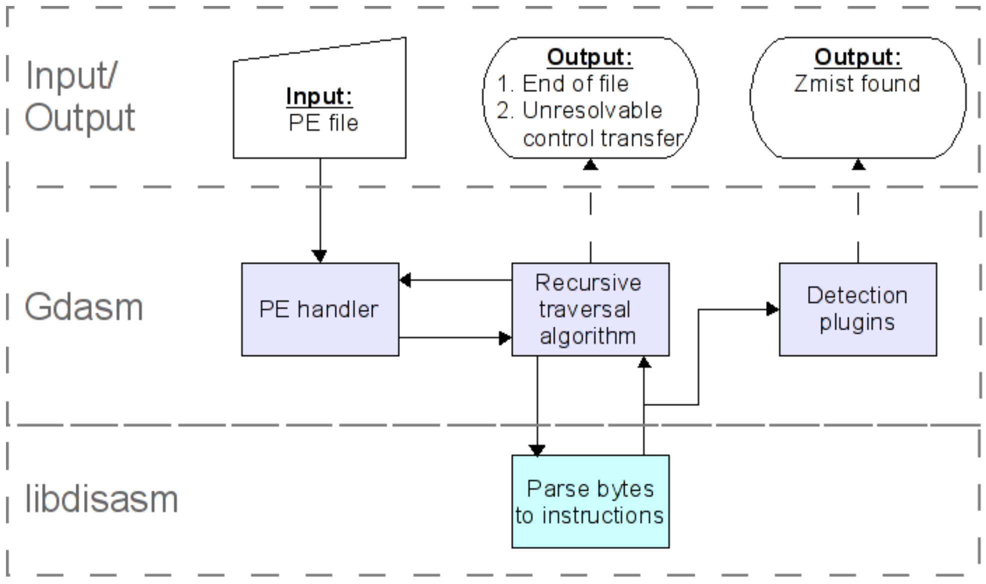

Input/ Output
Input: PE file
Output: 1. End of file 2. Unresolvable control transfer
Output: Zmist found
Gdasm
PE handler
Recursive traversal algorithm
Detection plugins
libdisasm
Parse bytes to instructions

Figure 3.1: Flow chart illustrating the Gdasm program

1. A PE file is given as input to Gdasm. The PE handler loads the section containing entry point into memory, and calls the RT algorithm function with entry point as argument.
2. The RT function passes the data pointer to libdisasm which returns an instruction to the RT function. The instruction is also passed on to a detection plugin containing code for detecting Zmist.
3. The RT function then sets the data pointer based on the instruction given corresponding to the RT algorithm. If an instruction points to a section not loaded in memory, the PE handler loads this section in memory from the PE file.
4. The cycle is repeated from stage 2, until the file is disassembled or an unresolvable instruction occur.

The source code of Gdasm is found on the CD described in appendix A on page 125.

## 3.2 Introduction to Zmist

Zmist is written by a Russian virus writer who calls her/himself “Zombie”. In January 2001, “Zombie” released “Total Zombification”, an e-zine containing several texts and source files related to creating viruses [10]. One of the source files contained *Mistfall*, which is the title of the main virus engine in Zmist[1]. The antivirus companies named

[1]One of the strings which can be found in memory when Zmist is run is: “with morning comes Mistfall”, which is the title of a science fiction story by George R. R. Martin written in 1973.

the virus from the first letter of the author's alias (Zombie) and the name of the engine (Mistfall).

The material is structured as follows. First, different versions of Zmist and their inequalities are illustrated in section 3.2.1. Second, the code integration technique used by Zmist is discussed in section 3.2.2. Third, the three different infection types Zmist choose from are discussed in section 3.2.3 on page 54. Fourth, suggested detection methods for two of the three infection types is found in section 3.2.4 on page 55 "Detection of JMP and Unencrypted Engine". Finally, analysis, code examples and developed detection algorithms for the last infection type is found in section 3.3 on page 56 and section 3.4 on page 62.

### 3.2.1 Zmist Versions

There are currently four different versions of Zmist: Zmist.A, Zmist.B, Zmist.C and Zmist.D. With each new version, "vulnerabilities" used by AV products to discover Zmist are removed by "Zombie". Table 3.1 illustrates the differences between the versions. The information provided in the table is taken from "Zombie's" own changelog, released in the 29A e-zine [50]. The writeup on Zmist done by Szor & Ferrie focuses on Zmist.A, but the detection methods developed discussed in section 3.4 on page 62 is valid for all Zmist versions.

| **Zmist version** | **Changes** |
|---|---|
| Zmist.A | Initial release |
| Zmist.B | - 'Z'-sign from MZ header removed, multi-infections can occur, max 2-4 times.<br>-UEP probability changed from 1/10 to 1/5<br>-Decryptor scheme changed |
| Zmist.C | -Randomly remove PE fixups<br>-More trash generated within permutated body of engine<br>-Encryption scheme improved |
| Zmist.D | -Random CALL's from decryptor to RETN's within host code<br>-Bugfixes |

Table 3.1: Differences between Zmist versions

### 3.2.2 Code Integration

Szor & Ferrie states the following on code integration: *"This* [code integration] *is something never seen before in previous viruses"*. Aycock [13] has seen this technique employed in Zmist only. It is therefore interesting to examine this unique method, starting with the consequences of code integration (CI) for Zmist.

### CI and consequences for Zmist

When Zmist infects a new host, it disassembles the host program and search for suitable instructions to insert itself after. This means that all code and data references to locations appearing possibly before and after the inserted virus code must be regenerated. Consider the following clean-code snippet:

```
0x401000 mov EAX,13h
0x401004 cmp ECX,1
0x401006 jz 0x401020
...
0x401020 mov [0x401040], EAX
```

If a 20 byte Zmist-code snippet should be inserted after the `jz 0x401020` instruction, the `mov [0x401040],EAX` instruction would not be executed as intended unless the address of the instruction is updated to `0x401060`. The same problem occurs with the `jz 0x401020`, where the address must be regenerated to `0x401040`. Because of this regeneration of locations, code integration requires the host to have fixups. Fixups were described in section 2.1.3 on page 22.

### CI and Zmist receiving control

Because of CI, Zmist is the "perfect" entry point obscuring (EPO) virus. There is no need for Zmist to replace any host-calls with its own code transfer, because Zmist insert the code between the host-code. There is no need for Zmist to add new sections to the PE file it infects or place its code in sections usually reserved for data. Because of this, there are no code transfers to unusual PE-sections which could be triggered by heuristics.

Because of CI, Zmist might never receive control. Consider the following code snippet:

```
Host code
..
Host conditional jump
ZMIST ENTRY POINT CODE
Host conditional jump destination
Host code
..
```

If the conditional jump is taken when the code is run, Zmist is not executed. From the Zmist point of view, this is not a problem because only one execution of the Zmist code results in an infection.

| **Name** | **Clean code→Zmist** | **Zmist→clean code** |
|---|---|---|
| Absolute indirect call and ret | `call [mem]` | `ret` |
| Relative call and ret | `call imm` | `ret` |
| Relative jump | `jmp imm` | `jmp imm` |
| Part of flow | UEP | UEP |

Table 3.2: Zmist receiving- and passing-control transfer instructions

Zmist receives control using one of four different instructions with a corresponding instruction returning the control to clean code. Table 3.2 illustrates the instructions with corresponding return instruction. The last element in the table is unique for Zmist. Because of CI, Zmist can be a part of the natural instruction flow of the clean code. "Zombie" calls this method for *UEP*. Szor & Ferrie suggest this might be an acronym for *Unknown Entry Point*, since there are no direct references to where the entry point of the virus is.

### 3.2.3 Infection Mechanisms

When analysing a virus, it is important to know how the virus appears in the host. Zmist choose one of three different infection types when infecting new hosts.

1. JMP after each host instruction.
2. Unencrypted Mistfall engine.
3. Encrypted Mistfall engine with polymorphic decryptor.

#### JMP after each host instruction

In this infection type, the Mistfall engine is not included in the infected host. The virus is therefore not able to replicate. The only difference between the clean host and the infected host, is a JMP instruction with some random bytes inserted between each original instruction. Consider the following code-snippet.

```
orig inst 1
JMP to orig inst 2
garbage bytes
orig inst 2
JMP to orig inst 3
garbage bytes
...
```

The clean host instructions are preserved, and the infected host runs as if it weren't infected. The infected host would theoretically execute slower because of the additional instructions added by Zmist. In Zmist.A, it is a 1/10 probability that a file is infected using this technique.

#### Unencrypted Mistfall Engine

The "Unencrypted Mistfall Engine" infection type inserts an unencrypted engine in the infected host. The engine receives control using one of the four types described in table 3.2 on the facing page. In Zmist.A, it is also a 1/10 probability that a file is infected using this technique. From the changelog in table 3.1 on page 52, in Zmist.B, C and D the probability is increased to 2/10.

#### Encrypted Mistfall Engine

In the "Encrypted Mistfall Engine" infection type, the Mistfall engine is encrypted in the infected host. Because the engine is encrypted, a decryptor is also present. The decryptor is polymorphic and receives control using one of the four types described in table 3.2 on the facing page. It is a 8/10 probability that a file is infected using this technique in Zmist.A. This infection type is from now on referred to as infection type *pd*, short for "polymorphic decryptor infection type".

If the host PE-file does not contain a writable section with initialised data (`.data`) within the three first sections, this infection type is not chosen. If `.data` is present, the virtual size is increased with 32KB, making room for the variables used in the decryption routine and the decrypted engine.

The encrypted engine is placed in the `.text` segment.

### 3.2.4 Detection of JMP and Unencrypted Engine

Detecting the JMP infection type is possible using instruction statistics. Approximately 50% of the instructions disassembled using the RT algorithm would be JMP instructions. It is considered unusual of a compiler to produce such code. It is therefore assumed that detection of this type is trivial, and is therefore not investigated further.

To detect the presence of the unencrypted Mistfall engine, one proposal was to use a conventional byte signature with wildcard support. This was suggested as a possible approach, because even though the engine is metamorphic, there are fixed parts appearing in each instance of the virus, i.e. the level of metamorphism is not high enough to avoid detection by byte-signatures. This proposition was not investigated any further, because of two reasons. First, searching for byte-signatures has been done by the antivirus industry for decades. The competence on finding accurate byte-signatures is therefore high and it was natural that this task should be left to the virus analysts at Norman ASA. Second, the detection of the polymorphic decryptor infection type was still unsolved.

Therefore, the rest of this chapter focuses on the polymorphic decryptor infection type.

## 3.3 About the Polymorphic Decryptor Infection type

As stated in section 3.2.3 on the previous page, 80% of the Zmist.A infected files contains the pd-infection type. The analysis effort was therefore initially directed at the files containing this infection type. The material described in this section is valid for all four Zmist versions (A,B,C and D). The material is a result of my own research.

### 3.3.1 Analysis of the Polymorphic Decryptor

Figure 3.2 on the facing page is a flow chart illustrating the flow:
clean code → decryptor → engine → decryptor → clean code.

#### The "Initialising code"-state

This is the first Zmist-state in the flow chart illustrated in figure 3.2 on the next page.

There are two keys involved in the decryption process. One permanent *seed key* and one dynamic *round key* changing for each iteration of the decryption. Both keys are stored in the `.data` section. The "Initialising code"-state performs the following.

1. Save the registers which is to be used in the decryptor on the stack (`push reg`), by a move onto the `.data` section (`mov [.data-address], reg`), or by a push/pop pair (`push reg ; pop [.data-address]`).

2. Initialise the round key variable with a static value.

3. Initialise accounting variables used by the decryption routine.

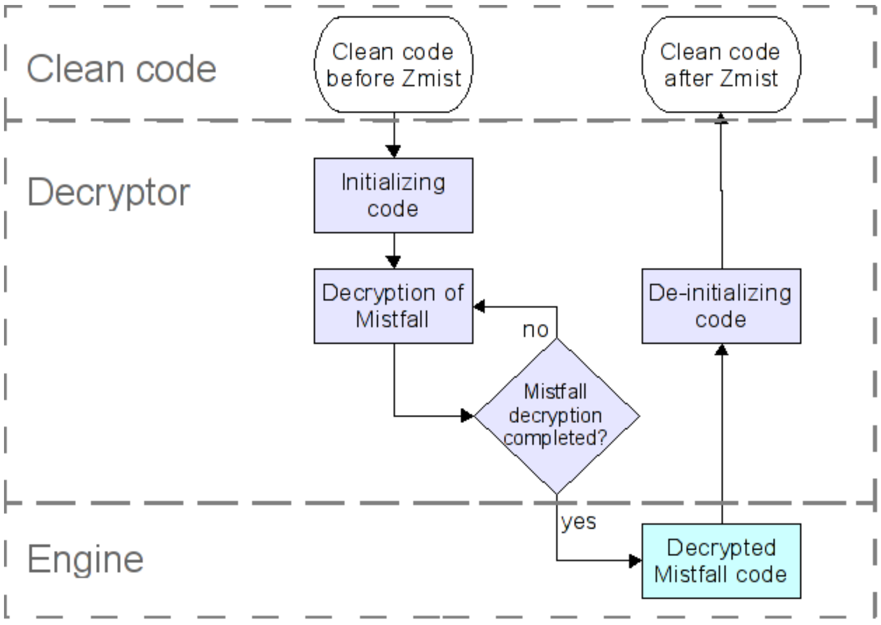


Figure 3.2: pd-infection type flow chart

### The “Decryption of Mistfall”-state

This is the second Zmist-state in the flow chart illustrated in figure 3.2.

In the “Decryption of Mistfall”-state, the actual decryption of the engine is done. The state can be summarised as follows.

1. The instruction used for decryption is one of the following: `add`, `xor` or `sub`. For each iteration, 4 bytes of the encrypted engine is fetched and decrypted using the round key. After the decryption, the 4 decrypted bytes are written to the `.data` section. Both the fetch and store is done sequentially.
2. A new round key is generated by performing an `add`, `xor` or `sub` instruction between the seed key and the existing round key.

The two steps are repeated until the whole engine is decrypted. The reason for generating a new round key for each iteration is probably to make x-raying more difficult. X-raying were described in section 2.5.1 on page 41. Note that the seed key, the round key, the decryption instruction and the round key instruction is different each time Mistfall generates a new polymorphic decryptor. The placement of the keys are also different.

### The “Mistfall decryption completed?”-question box

This is the question box in the flow chart illustrated in figure 3.2.

Zmist use two different methods to loop back to another round of decryption. With both methods, a compare (`cmp`) is done between two arbitrary registers. One register holds the total size of the encrypted engine, the other holds the number of bytes decrypted. The difference between the two methods is how they jump back to the start of the decryptor.

1. The first method is a common way of looping code. After the compare, a Jcc instruction transfers the control back to the decryptor loop. The type of Jcc instruction vary amongst many variants, such as: `jl,jz,jg,jnz`. The list is not complete. The fall through branch is code responsible for transferring control to the decrypted engine.

2. The second method is an unusual variant. After the compare, one of the `SETcc` instructions (Set Byte on Condition, [17]) is used with a 8 bit register as operand. After the `SETcc` instruction is executed, the register holds a value of 0 or 1. The code then jumps using this register as an index to a jump table with two entries. One of the entries is a pointer to the start of the decryptor loop. The other entry is a pointer to code responsible for transferring control to the decrypted engine. The instruction responsible for the jump is on the following form: `jmp [reg*4 + imm]`.

Because the second method is relatively specific for Zmist, a real-world example is given in table 3.3.

| # | **Instructions** |
|---|---|
| 1 | `cmp EDX, edi` |
| 2 | `setc dl` |
| 3 | `movzx EDX, dl` |
| 4 | `jmp [EDX*4+0x40C438]` |

Table 3.3: Unusual conditional control transfer

The instructions are reproduced from a disassembling of the file *REGEDIT-4.VXE* using the Gdasm tool. In instruction 2, `dl` is 0 or 1. In instruction 3, `dl` is moved into `EDX` by `movzx` which zero-extends the value [16]. The JMP destination would then be `[40C438]` or `[40C43C]` depending on the value in `EDX`.

When the decryptor has decrypted the engine, control is transferred to a part of the decryptor dedicated to transferring control to the Mistfall engine. One of the two following instructions is used to transfer control to Mistfall:

1. `call displacement`

2. `call [mem]`

The engine returns to the decryptor with a `ret` instruction.

#### The “De-initialising code”-state

This is the fourth Zmist-state illustrated in the flow chart in figure 3.2 on page 57.

In this state, the registers saved in the “Initialising code”-state is restored and control is transferred back to the host code. The transfer back is done by the corresponding instruction in table 3.2 on page 54.

### 3.3.2 Code Examples of Decryptor

To illustrate the level of polymorphism Zmist incorporates in the decryptors, disassembling excerpts of three infected files are given in table 3.4 on the following page. The full disassembling outputs are found on the CD described in appendix A on page 125.

For each decryptor, the 45 first instructions are reproduced. D1 is an excerpt from the file *SNDREC32-5.VXE*, D2 from the file *HWINFO-5.VXE* and D3 from the file *PMTS-5.VXE*. Colours are added to illustrate different features within the code. The features are explained in section 3.3.3.

The three examples are of the Zmist.A type. There are no significant differences between the decryptors found in Zmist.A, B, C or D and these examples represent a true picture of a Zmist decryptor independent of version. The examples are produced with the Gdasm tool described in section 3.1.4 on page 50. The three examples illustrated are referred to as D1, D2 and D3. An individual instruction in the table is referred to as a coordinate in the table matrix: (column, row). For instance (D1,3) is the third instruction in D1: `bsf ECX, EAX`.

### 3.3.3 Polymorphic Techniques Employed

This section describes the polymorphic techniques employed in the Zmist code examples in table 3.4 on the following page. Descriptions of common polymorphic techniques were given in section 2.4.7 on page 37.

#### EPO techniques

The EPO techniques employed in Zmist was previously described in table 3.2 on page 54. In D1, the code receives control via an *absolute indirect call*. D2 employs the *UEP* technique while D3 receives control via a *relative call*.

Table 3.4: Code example of 3 Zmist decryptors

| # | Decryptor 1 (D1) | Decryptor 2 (D2) | Decryptor 3 (D3) |
|---|---|---|---|
| 1 | call [0x4082FE] | mov [0x421634], edi | call 0x00003090 |
| 2 | mov [0x41B000], ECX | mov [0x41FBFC], EBX | push EAX |
| 3 | bsf ECX, EAX | mov edi, 0x00421D90 | pop [0x415E64] |
| 4 | and ECX, 0x9ACEDD96 | push EAX | mov al, 0x4B |
| 5 | mov [0x41BE98], EBX | pop [edi] | xadd EAX |
| 6 | mov ECX, esi | mov [0x41FBF0], 0x00000000 | adc EAX, 0x01D257DA |
| 7 | btr ECX, ECX | mov edi, 0x004143E2 | push esi |
| 8 | mov ECX, 0x5543994C | push edi | repnz neg EAX |
| 9 | mov EBX, 0x00426758 | pop EAX | imul esi, ECX, 0xCD32F179 |
| 10 | imul ECX, EDX, 0xA461EDD1 | mov EBX, [EAX] | jmp 0x000030AF |
| 11 | movsx ECX, ax | mov [0x4211B8], EBX | repz mov EAX, 0x004167FC |
| 12 | mov [EBX], esi | mov edi, [0x4035E5] | not esi |
| 13 | test al, 0xBA | mov EBX, edi | lea esi, [-1587682296] |
| 14 | test edi, esi | mov [0x41FBF8], EBX | mov esi, esi |
| 15 | repnz imul ECX, esi | mov edi, 0x0041FBF0 | bsr esi, ECX |
| 16 | mov esi, 0x00000000 | push [edi] | push ECX |
| 17 | bswap EBX | pop EAX | test esi, EAX |
| 18 | mov [0x424C70], esi | mov EBX, EAX | pop [EAX] |
| 19 | mov EBX, edi | push EBX | mov esi, 0x60C25CE0 |
| 20 | add ch, bl | pop EAX | bswap esi |
| 21 | mov esi, 0x0040A7E1 | mov edi, 0x00421630 | mov esi, EBX |
| 22 | mov ECX, [esi] | push EAX | movsx esi, bp |
| 23 | mov EBX, ECX | pop [edi] | xadd esi |
| 24 | shld esi, ECX, st(1) | mov EAX, 0x00421630 | adc esi, 0x52ED71CC |
| 25 | mov esi, 0x00423E40 | mov EBX, [EAX] | jmp 0x000030E0 |
| 26 | dec cl | add EBX, [0x4062A8] | test EBX, ECX |
| 27 | inc ECX | mov [0x421630], EBX | mov ah, 0xF9 |
| 28 | mov ECX, esi | mov edi, [0x421630] | mov esi, 0xFCA091F5 |
| 29 | push EBX | mov EBX, [edi] | imul EAX, esi, 0x45285F96 |
| 30 | mov esi, EBP | mov EAX, EBX | mov ECX, edi |
| 31 | xchg esi, esi | mov [0x41FBF4], EAX | test ah, bh |
| 32 | pop [ECX] | push [0x41FBF4] | bsf EAX, EDX |
| 33 | sal esi, 0x99 | pop edi | adc EAX, 0x86DDDCAD |
| 34 | mov esi, 0x49FEF735 | mov EBX, 0x004211B8 | sub esi, 0x2154A1C0 |
| 35 | mov esi, 0x00416723 | push [EBX] | mov EAX, EBP |
| 36 | mov EBX, [esi] | pop EAX | rcl EAX, 0xA7 |
| 37 | push EBX | xor edi, EAX | shrd ECX, EBP, 0x44 |
| 38 | bts esi, 0x7D | mov EBX, 0x0041FBF4 | sub esi, 0xACEEC496 |
| 39 | sub esi, edi | mov EAX, EBX | bts ECX, 0x44 |
| 40 | pop ECX | mov [EAX], edi | shl EAX, 0x00000001 |
| 41 | repnz push ECX | push [0x41FBF8] | xor esi, 0x2E5D2B9F |
| 42 | inc EBX | pop EBX | xadd ECX |
| 43 | mov bl, cl | mov edi, EBX | lea ECX, [0x5D46D419] |
| 44 | pop [0x424C6C] | mov EAX, 0x004211B8 | sal cl, 0x76 |
| 45 | shrd EBX, esi, 0xE9 | mov EBX, EAX | mov [0x417CF0], esi |

### Junk Instructions

There are several examples of *junk instructions* found in the decryptors. Examples of junk instructions is marked in blue colour in D1. Note that there are lots of other junk instructions in D1 and D3 not marked blue. D2 does not contain any junk instructions at all.

It is possible when (D1,6) and (D1,13) has been evaluated to decide that the information produced in (D1,3) and (D1,4) is overwritten and therefore junk. (D1,6) overwrites the `ECX` register and (D1,13) overwrites the flags set by (D1,4).

(D1,13) is a junk instruction itself, because the flags set by (D1,13) is overwritten in (D1,14).

### Instruction Equivalence

In D2, there are no junk instructions as seen in D1 or D3. D2 illustrates the existence of "ineffective" ways of propagating values in Zmist. The ineffective code is marked in green and the example consists of instructions in the range (D2,15) to (D2,24).

No sane compiler would have generated the sequence of instructions marked in green, but the sequence does not classify as junk. This is because the sequence affects the decryption process. The following example illustrates why the sequence is considered ineffective. The transport of the value at memory address `0x0041fbf0` uses 9 instructions and 3 different registers before reaching the destination memory address `0x00421630`. The following two instructions performs the same action using 2 instructions and 1 register:

```
mov EAX, [0x0041fbf0]
mov [0x00421630], EAX
```

Another example of "ineffective" code is illustrated in D3 with the following 5 instructions: (D3,28), (D3,34), (D3,38), (D3,41) and (D3,45). The instructions are marked in red. Also note the instructions (D1,16),(D1,18) and (D2,6), all marked in red. In D1 and D2, a 32-bit memory address is overwritten with the value 0. In D3, there are no obvious instructions accomplishing the equivalent. However, a closer examination of the 5 red instructions in D3 provides the equal operation: $(0xFCA091F5 - 0x2154A1C0 - 0xACEEC496) \oplus 0x2E5D2B9F = 0$. A memory address is therefore zeroed in D3 as well.

### JMP islands

Because of the code integration technique employed, the code flow can be altered in practically unlimited arbitrary ways. An arbitrary number of JMPs linking scattered code islands together can be found in the decryptors. In D1 and D2 there are no

JMP insertions while in D3 there are 2, coloured yellow. Counting the JMPs in the decryptors listed on the CD described in appendix A on page 125, there are 6 JMPs in D1, 0 JMPs in D2 and 10 JMPs in D3.

## 3.4 Detection of the Polymorphic Decryptor

From phase two described in the methodology section ( 3.1.2 on page 48), Zmist should be investigated by using an "iterative trial and error" approach. Many iterations of trial and error lead to two major detection attempts. These two are described in this section.

The following two problems arises when considering Zmist decryptor detection. The first problem would be solved if a solution of problem 2 is found, but not vice-versa.

**Problem 1** Detect the presence of a Zmist decryptor reliably.

**Problem 2** Locate a Zmist decryptor reliably.

### 3.4.1 Geometric Properties

A Zmist-infected file has geometric properties which can indicate the presence of Zmist. The properties described are a summary from [39]. Geometric properties were described in section 2.5 on page 41.

One geometric property is the 'Z'-sign in the Zmist.A version. For each file Zmist.A infects, the character 'Z' is placed at offset 0x1C in the MZ header of the infected file. This offset is not used in general by Windows applications. The 'Z'-sign is used by Zmist to avoid additional infections of an already infected file. From the changelog illustrated in table 3.1 on page 52, the 'Z'-signing of infected files is removed in Zmist.B.

Another geometric property of Zmist is the update of fixups. When Zmist infects a host, the fixups of the infected file is updated by Zmist. Because the engine is placed in .text, and the size of the encrypted engine is approximately 20KB, it would be possible to detect such a 20KB gap between two fixup entries and use this to indicate a Zmist infection. As it was with the 'Z'-sign, fixups are removed in a later version of Zmist, namely Zmist.C.

As the two examples illustrate, relying on geometric properties *only* is not reliable. First of all, geometric properties does not solve problem 1. Other viruses can for instance write a 'Z' at position 0x1C and the detection of Zmist would not be reliable. In addition, the virus author can easily remove or improve some of the geometric properties and detection will fail. In both cases illustrated here, "Zombie" removed the properties revealing the presence of Zmist, in later versions of the virus. The use of geometric properties is however necessary to *indicate* the presence of a virus. Validating/invalidating geometric properties in a file is fast compared to scanning the whole file and is often used in antivirus products to weed out the files to scan properly. In this thesis, the geometric properties are not used to confirm/deny the presence of Zmist in a file. It is the Zmist decryptor that is difficult to detect. A detection algorithm must therefore be found independent of geometric properties.

### 3.4.2 Detection by Instruction Statistic

After analysing a number of Zmist decryptors, it became possible to detect the decryptor manually quickly. This was possible because Zmist uses instructions seldom found in clean code. By a manual search for `xchg` or `bswap`, the presence of a decryptor was confirmed. This lead to the first serious attempt on detecting the Zmist decryptor. The assumption was that if the instruction statistic on Zmist was significant different than the instruction statistic on clean code, instruction statistic would provide a solution to problem 1.

The detection attempt by using instruction statistic did not lead to a working detection algorithm. However, the results found, lead to another more detection attempt which was more successful. The testing and results of the instruction statistic attempt is described here instead of the testing chapter, because they provide a foundation for the attempt described in section 3.4.3 on page 66.

#### Setup

The following files were used to conduct the instruction statistic experiment.

- 152 clean files from a Windows 98 installation.
- 46 Zmist.A infected files, manually confirmed infected with pd inf.type.

The statistic foundation were gathered and processed in three phases. First, a plugin was written to Gdasm. This plugin extracts the instruction mnemonic and inserts this into a MySQL database. Second, Gdasm was run on the files described in the setup and the mnemonics extracted from each file were stored into the MySQL database. Finally, OpenOffice was used to manipulate the mnemonics found in the MySQL base.

This procedure was applied to both clean and Zmist infected files.

#### Discussion

When analysing the results, the following observation were made: The *number of instructions* (NOI) in the decryptor is independent of the NOI in the host program. The location of the Zmist decryptor inside an arbitrary host is unknown. The whole host must therefore be disassembled and processed. As a consequence of the NOI-independence and the unknown location, the statistic produced is not precise enough.

| **NOI** | **WINHLP32-4.VXE** | **INTERNAT-4.VXE** |
|---|---|---|
| NOI in decryptor | 342 | 227 |
| NOI in whole host | 40663 | 384 |

Table 3.5: Number of instruction (NOI) in two Zmist.A samples

Consider table 3.5 on the following page. In WINHLP32-4.VXE, the Zmist instructions contribute with $342/40633 \approx 0,8\%$ of the total instruction statistic. In INTERNAT-4.VXE, the percentage is increased to $227/384 \approx 59,1\%$. Unless the instructions used in the Zmist decryptor are instructions seldom or never used in clean code, instruction statistics is not a reliable detection method. If the instructions used in the Zmist decryptor is different from clean code, instruction statistic could decide if a given file is infected or not.

Figure 3.4.2 on the next page plots the instruction statistic of the file HWINFO-5.VXE together with the clean code statistic. The figure shows that for each instruction found in HWINFO-5.VXE, the infected file follows the clean code statistic. HWINFO-5.VXE is a proof of the existence of Zmist decryptors that use the same instruction mnemonics as clean code. It is therefore concluded that instruction statistic will not be a successful approach to Zmist detection.

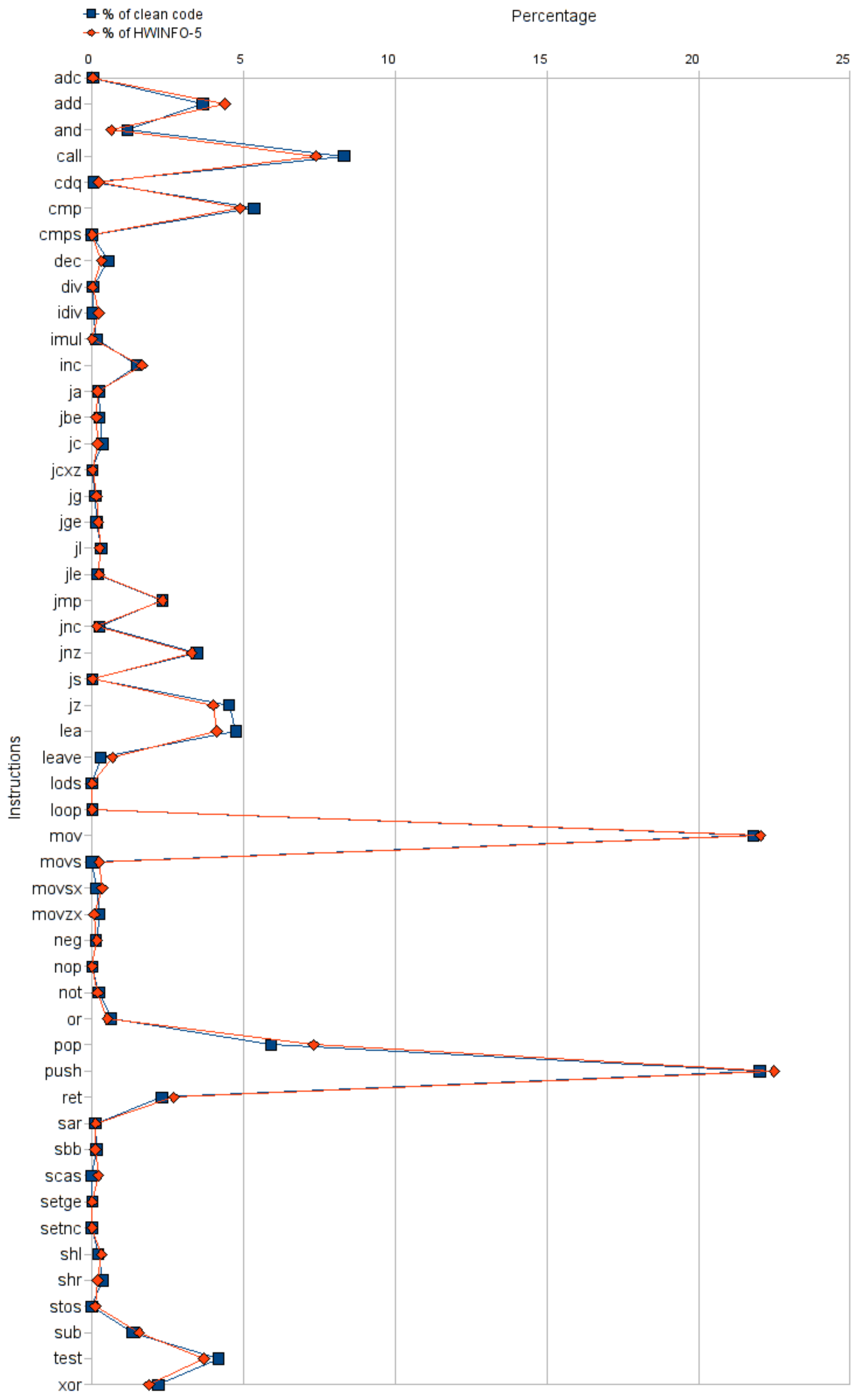


Figure 3.3: Percentage of instructions found in HWINFO-5.VXE, plotted together clean code percentage

### 3.4.3 Detection by Decryptor Properties

Szor & Ferrie states the following in their Zmist paper.

> *Due to its extreme camouflage Zmist is clearly the perfect anti-heuristics virus.*

The statistical approach previously described confirms this quote. Another weakness with the statistical approach is that the decryptor is not located precisely i.e. it doesn't solve problem 2. With this in mind, a new detection approach was constructed. This approach uses the *properties* of the decryptor. This is a more tailored attempt at detecting the decryptor, compared to the instruction statistic.

#### Common properties

After analysing several decryptors, a pattern of common properties became evident. The properties are listed here.

- Register saves. This is described in the next section.
- The ESP register is not referenced explicitly in any of the observed decryptors.
- No other conditional jumps than the loop construct (described in section 3.3.1 on page 57) is present in any of the observed decryptors. The least observed distance measured in instructions from start to loop construct, was 84 instructions (HWINFO-5.VXE).
- There are no `call` or `ret` instructions in any of the observed decryptors.
- Besides the loop construct `jmp [reg*4 + offset]`, there are no occurrences of the addressing type `[reg+offset]`.

The hypothesis is that when combining the common properties listed, they are not frequently found in clean code, and can therefore be used as "ingredients" in a detection algorithm.

| # | **Not valid decryptor** |
|---|---|
| 1 | `push EAX` |
| 2 | `pop [mem]` |
| 3 | `mov [0x400000], EBX` |
| 4 | `mov ECX, EAX` |

Both `EAX` and `EBX` is saved correctly in instruction 1,2 and 3. In instruction 4, `ECX` is overwritten before being saved. This is something not observed in the Zmist decryptors.

Table 3.6: Constructed example of an invalid decryptor considering register saves

### Register saves

All decryptors observed save the registers used in the decryptor. This is done randomly for each register using one of the three methods.

- `push reg`
- `push reg; pop [mem/reg_2]`
- `mov [mem/reg2], reg`

In the listing, reg2 is an already saved register by the decryptor. For the first register save, an absolute memory address is used unless the stack-method is used. The registers saved are always 32-bit registers and the least amount of utilised registers observed in a decryptor was 3.

This implies that in a valid decryptor, there are no register overwrites unless the register has been saved first. Table 3.6 illustrates a constructed example of an *invalid* decryptor considering register saves.

Assume all Zmist decryptors store the registers in the way described. Because register saves is the first operation a Zmist decryptor performs, the start of a Zmist decryptor can be located precisely (corresponding to problem 2).

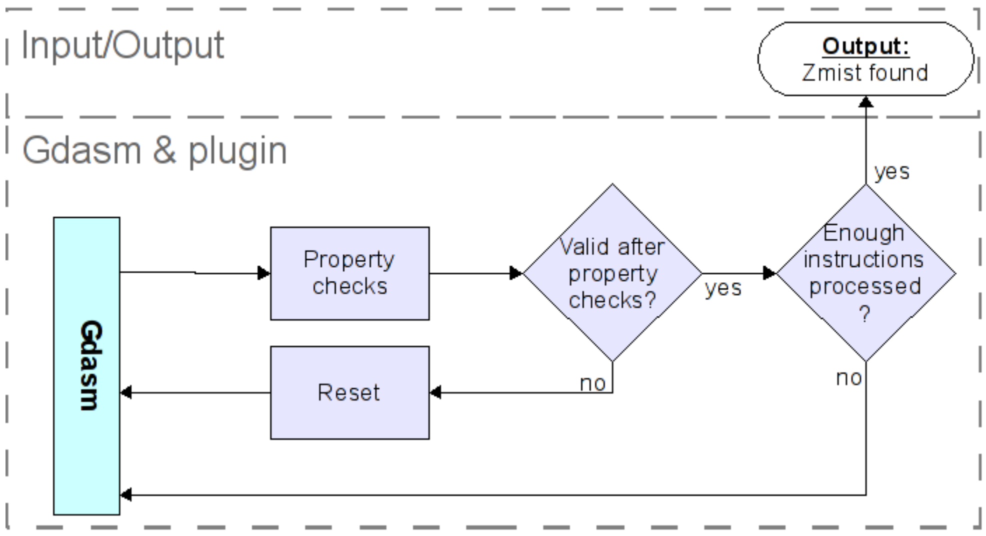


Figure 3.4: This figure is the flow chart of a plugin written for Gdasm. The Gdasm flow chart was previously illustrated in figure 3.1 on page 51

### Construction of the Detection Algorithm

The properties listed constitute the “ingredients” to a detection algorithm. A plugin was written to Gdasm containing the detection algorithm. Figure 3.4 illustrates the flow in the plugin.

1. The “property check”-state receives an instruction from the “Gdasm”-state. The properties checked are the properties previously listed in 3.4.3 on page 66. The regsave property discussed must be satisfied initially for the other properties to be checked.

2. Violations of the properties result in a reset of the plugin. If there are no violations, the control reaches the “Enough instructions processed?”-question box. This evaluates to “yes” if enough instructions since first regsave has passed through the property check.

Testing of the decryptor detection is done in chapter 6 and the results are promising. Of the decryptors tested, 94% were detected using the decryptor properties approach described. The remaining 6% left undetected, are attributed to the choice of disassembling algorithm (RT).

The source code of the Zmist Decryptor detection is found on the CD described in appendix A on page 125.

## 3.5 Summary

This chapter fulfils the first research goal posed in the introduction. The goal was to "learn about polymorphic viruses through a case study", and was extended to include detection of the case study. In this chapter, the poly- and metamorphic virus Zmist has been analysed and two detection approaches have been formulated.

As stated in the second research goal, from the knowledge acquired in the case study it should be examined if it is possible to infer anything which could be used in detection of polymorphic viruses. The next chapter assails the second research goal.

# Chapter 4

# Analysis and Lessons Learnt

## 4.1 Case Study Aftermath

It took approximately 8 months from the start of the Zmist analysis, to a detection algorithm was found. Assume having an undetectable botnet-communicating virus executing on a number of computers. Further assume a detection algorithm is developed after 8 months of the virus initial release. The infected computers could contribute to illegal activities such as illustrated in the blackmailing scheme for 8 months, before the virus could be detected due the lack of a detection algorithm. This illustrates a desire to reduce the analysis period, thus reducing the lifetime of a virus.

The following question was therefore posed: How can the time spent analysing a polymorphic virus be reduced? The common patterns tying the Zmist decryptors together were found manually in 8 months. The majority of the time were spent comparing a vast number of different decryptors, trying to find suitable common patterns which could be used in a detection algorithm. Following from this acknowledgement, a new question was posed: Is it possible to automatise or semi-automatise this comparison process?

### 4.1.1 Comparison Program

As an answer to the last question, an idea of a *comparison program* was born. Such a program would aid the virus analyst in finding the common patterns between different instances of a polymorphic virus. The program would take as input a number of infected files. The program would then apply a *comparison algorithm* on the input, and finally output the common patterns.

#### Accurate location of virus code

The first problem with designing such a comparison program was: how to locate the virus accurately in the infected files passed as input to the comparison program? If the virus is not located accurately, the foundation of the comparison is not valid and erroneous results would be produced. Most virus analysts have environments for safely analysing and running viruses, such as emulators and virtual machines (see the tools section 3.1.3 on page 49). In such environments, it is easy to discover the set of files which have been infected when a virus has been executed. The following paragraph explains how to discover the infected files in a virtualised environment.

Assume there is a virus an analyst wish to analyse inside a virtual machine. Assume further there exist a copy of the clean files found in the OS installed in the virtual machine *outside* of the virtual machine, not reachable by the virus. The virus is executed inside the virtual machine and files are infected with the virus. Which files have been infected, can be discovered by a hash-comparison of the files inside the virtual machine, to the set of clean files on the outside. This process can be repeated, thus obtaining several infected files. To locate the virus in each infected file, consider the following. Let $x_c$ denote a clean file and $x_v^1, x_v^2, \ldots, x_v^n$ the virus-infected versions of $x_c$. Using the recursive traversal disassembling algorithm, by comparing clean-file-instructions to infected-file-instructions in $x_c$ to $x_v^i, 1 \leq i \leq n$, the entry point of the virus can be found in $x_v^i$.

The comparison program would then take as input the different infections and compare them to each other: $x_v^i$ compared to $x_v^r$, $\forall i \neq r$.

### Comparison Algorithm

Having solved the problem of locating the virus, the next problem became apparent: how should a comparison algorithm be designed?

The first step in a comparison algorithm would be to remove the detectable elements from each of the infected files that are <u>not</u> common patterns. This would provide a more homogeneous comparison foundation. Consider junk instructions, described in section 2.4.7 on page 37 and illustrated in the Zmist decryptor examples 3.4 on page 60. The junk instructions are not necessarily common patterns between the decryptors. Because junk instructions don't contribute to the semantics of the decryptors, they can be inserted arbitrarily by the virus and vary syntactically in numerous ways. Junk instructions should therefore be removed in each infected file prior to comparing them to each other.

The decision of detecting junk instructions led to the problem statement of the thesis: *Is it possible to separate junk instructions from non-junk instructions in an executable file?*

## 4.2 Motivation for Junk Instruction Detection

From the previous discussion, the motivation for removing junk is to improve the results of a comparison algorithm. A process charting out the possibilities of detecting junk instructions was initiated. In the process, there were three main questions.

### 4.2.1 Independence

The first question posed was: Could a method for junk instruction detection exist as a separate stand-alone function, as opposed to an implemented function in a comparison program?

The answer to this question is yes. If junk instruction detection is developed as a standalone application, it will work as an independent tool detecting junk instructions in arbitrary programs. When junk instructions are detected and removed from a virus, it will be easier for a virus analyst to examine the virus with or without a comparison program. The junk instructions make the non-junk instructions more difficult to spot, hence confusing the picture of the virus' true function. The following quote is taken from an analysis of the *Bagle* virus, done by Konstantin Rozinov [32].

> *In fact, while reverse engineering, you can spend up to 80% of your time reading the values in registers and deducing what the code will do or is doing as a result of these values.*

If a junk instruction detection method can contribute to removing some of the instructions a virus analyst must read, the time spent analysing decreases. This conforms to the initial goal of this chapter: How can analysis time be reduced?

### 4.2.2 Other Uses

The second question posed was: Are there any other uses for a junk instruction detection method? The presence of junk instructions in an executable file can be an indicator of an unknown virus. The junk instruction detection method can be used to detect the amount of junk instructions present and can therefore act as a heuristic. Junk instruction detection would therefore have other uses.

### 4.2.3 Similar work

The third question posed was: Is there done anything similar? The terms "junk code" [13] , "garbage instructions"[23] and "garbage code" [5] are also popular terms for junk instructions. The following databases were searched for the different terms.

- IEEE [3]
- ACM [1]
- Springerlink [8]

IEEE returned no results, but ACM and Springerlink returned results. Some articles contained an explanation of what junk instructions are. Others described virus detection approaches using some form for emulation or language normalising technique. These techniques in conjunction with junk instruction detection are described next.

#### Emulation

Emulation as a virus detection method were described in section 2.5.2 on page 43. Because emulation runs the virus code in a specific environment, there can exist code segments which does not receive control when emulated. Emulation demands a high amount of CPU cycles. Because of the possible use of junk instruction detection in heuristics, emulation is not suitable for junk instruction detection.

#### Language Normalising

Language normalising were described in section 2.5.1 on page 42. This method requires a mapping of the x86 assembler language, to the internal language used in the normalising technique. This is a tedious task because of the vast amount of instructions found in the x86 assembler language. Language normalising also requires a high amount of CPU time. Because all the virus code are processed, it is better suited for junk instruction detection than emulation. The heuristic argument used in emulation is however valid. Language normalising is a powerful technique, but it is believed that junk instruction detection can be done in using simpler methods.

### 4.2.4 Conclusion

The three questions posed had answers pointing in the direction of developing a junk instruction detection (JID) framework: First, developing a JID framework would have a separate function even if the comparison program would not be developed. This is important, because developing a JID framework could demand the rest of the disposable time for the thesis. Second, a JID framework can be used in heuristics to indicate the presence of an unknown virus. Third, there were no (published) similar work addressing junk instruction detection directly. The language normalising technique could be used, but it is believed that junk instruction detection can be done without utilising the full powers of language normalisation.

The main subject for the rest of this thesis is therefore detection of junk instructions. The next chapter is devoted to the design and implementation of the JID framework.

# Chapter 5

# The JID Framework

This chapter contains the specification and design of the Junk Instruction Detection (JID) framework and provides the foundation for an implementation of JID.

## 5.1 Methodology

The discussion in this section should work as a guideline, or a methodology further in the development of the JID framework.

### 5.1.1 General Limitations

Initially, it is important to develop a framework capable of detecting the typical junk code techniques used in the viruses of today. Rather than creating a framework capable of detecting all types of theoretical possible junk code techniques, the framework should be simple and expandable. If the framework is developed such that it is possible to do iterative expansions, then the initial design should focus on detecting the most basic types of junk code insertions.

The limitation proposed is reasonable. Insertion of advanced types of junk code, such as junk code propagating over several conditional jumps, is difficult for the virus writer to achieve properly in the general case. It is therefore assumed that viruses utilising the more advanced types of junk code appear less frequently, than viruses utilising the more basic types.

### 5.1.2 Assumptions

The number of possible instruction sequences are enormous. If assumptions or rules considering aspects of JID are made without having a sound foundation, there can exist instruction sequences counter-proving the rules created. Any assumption or rule must therefore be justified, to ensure that JID performs as intended, on all forms of instruction sequences.

### 5.1.3 Method for Junk Instruction Detection

#### Emulation and Language Normalisation

From the discussion in the previous chapter, emulation is not usable as a method for JID. The language normalisation technique would perform badly when it comes to heuristics. However, unless a simpler method is found, language normalisation would be the preferred technique for junk instruction detection.

#### Static Instruction Information

In the search for a simpler method, the following question were posed: Is it possible to detect the junk instructions without having an understanding of the instructions' semantics? For instance, can junk instructions be detected without understanding that the function of the instruction `add EAX, EBX` is to add `EBX` to `EAX`? Can junk instructions be detected, by only recognising that `EBX` is first read and then written to `EAX`, i.e. there has been a flow of information from `EBX` to `EAX`? By only utilising this type of *static instruction information*, it is not necessary to have an understanding of what each of the instructions found in the x86 instruction set does. This has three implications.

First, a disassembler reporting the reads and writes between operands is the only tool needed. Second, junk instructions present in programs written for other platforms can be detected using the same method for JID, as long there exist a disassembler producing static instruction information for the executables running on the platform. Examples of other platforms are the 64 bit Intel Architecture (successor of the 32-bit Intel Architecture) and ARM (often used in embedded devices such as mobile telephones). Third, if it is possible to detect junk instructions by the static instruction information produced in a recursive traversal (section 2.3.2 on page 29) or a linear sweep (section 2.3.1 on page 29), then junk detection would run in $O(n)$. $O(n)$ is the lowest achievable complexity when traversing all instructions found in a virus. Because all instructions must be traversed in a heuristic scan to assure the absence of malicious code, $O(n)$ is the lowest complexity possible for a heuristic scan analysing code of an arbitrary file. JID developed as described could therefore be included in a heuristic scanner without notable performance faults.

### 5.1.4 Conclusion

The three implications previously discussed, motivates an attempt to detect junk instructions using a disassembler together with the static instruction information. The

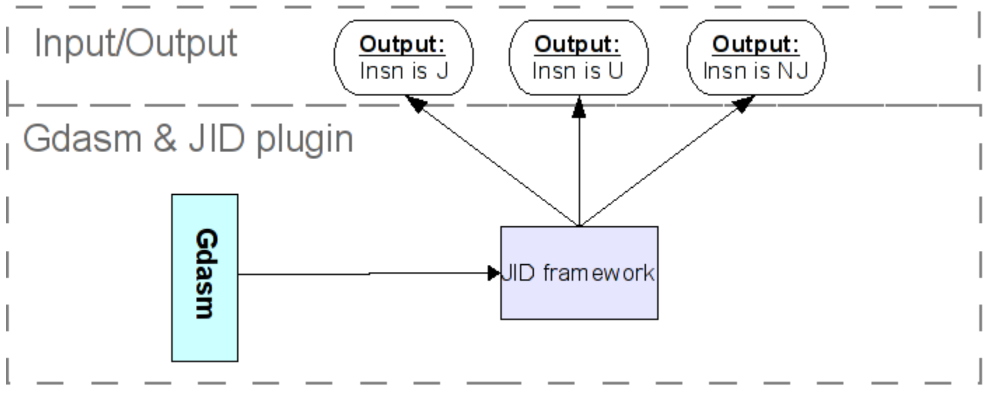


Figure 5.1: Illustrating the general flow of the junk detecting framework

remaining material in this chapter is devoted to the design/research and implementation of JID with the aid of a disassembler and static instruction information.

## 5.2 Introduction to JID

The following describes the layout of this section. First, a general overview of the framework and how it is intended to work is given in section 5.2.1. Second, a discussion on how instructions can be classified junk (J) and non-junk (NJ) using static instruction information is provided in section 5.2.2. Third, the fundamental elements of the framework is described in section 5.2.3. Finally, a summary listing the most important requirements found in this introduction is given in section 5.2.4. Section 5.3 on page 81 provides the detailed design of the framework, based on the discussion in this introduction.

### 5.2.1 Framework Overview

The goal of the framework is the following: For each instruction presented to the framework by the disassembler, the framework should classify the instruction into one of the categories: junk (J), non-junk (NJ) or if it cannot obtain a classification: unclassifiable (U). The unclassifiable category is equivalent to NJ except for the name. The term unclassifiable suggests manual inspection by the virus analyst and must therefore have a different name than NJ.

#### JID Developed as Plugin to Gdasm

Because junk code often are a sequence of several instructions, it is necessary to analyse the sequence as whole, not each individual instruction. Recursive traversal should therefore be used, because the decoded instructions appear in the same order as if they were executed on a CPU. The tool Gdasm 3.1.4 on page 50 is already written to use the recursive traversal, so JID would be developed as a plugin to Gdasm. Figure 5.1 on the preceding page illustrates the flow of the framework.

The theory developed around JID would fit well in an object-oriented environment. This implies that a object-oriented programming language should be chosen. C++ is an object-oriented language with more or less the equivalent performance and control as C. Because C is a subset of C++, and it is easy to integrate a C++ plugin to a C application, C++ was the chosen language for implementing JID.

### 5.2.2 Classification

This section discusses aspects around junk (J) and non-junk (NJ) classification of instructions. J and NJ classification are described first, followed by a discussion on misclassifications.

#### J Classification

How can instructions be classified as J? Junk instructions are featured heavily in the Zmist decryptors found in section 3.4 on page 60. One feature making it possible to identify the instructions as junk, is register overwrites. Data stored in a register alone does not change the semantic of a program. Only using the registers in operations affecting the program, justifies having data stored in registers. By detecting situations where registers are overwritten without utilising the data in reasonable operations, junk instructions can be identified. What is classified as "reasonable operations", is described in the next section on NJ Classification. The same argument can be used on flag setting. Assume an instruction sets the flags (for instance one of the compare instructions: `cmp` or `test`) and the same flags are overwritten by another compare instruction. The first compare instruction can then be classified as junk, because the information it created is overwritten without being used.

There can exist other types of junk instructions not utilising registers or flags. An example is junk instructions using memory. Resolving all memory addresses would need emulation in the general case, because of several different addressing mode used in the x86 assembler language. Consider the example in table 5.1 on the facing page. Here, both `value1` and `value2` are written to the memory address `0x403204`, and instructions 1,2,3,4 and 5 is J. In the example, it is not possible to resolve the memory location in instruction 5, by using static instruction information only. Therefore, J propagating through memory is not detectable by the framework.

| # | Instruction | Explanation |
|---|---|---|
| 1 | `mov EAX,0x1000` | 0x1000 is moved into EAX |
| 2 | `xor EAX,0x5002` | EAX is XOR'ed with 0x5000 |
| 3 | `shl EAX, 8` | Value in EAX shifted left 8 bits |
| 4 | `or EAX, 0x3004` | EAX is OR'ed with 0x3200 |
| 5 | `mov [EAX], value1` | value1 is moved to [EAX] |
| 6 | `mov [0x403204], value2` | value2 is moved to [0x403204] |

Table 5.1: Value in EAX after instructions 1,2,3,4 has executed is: $(((0x1000 \oplus 0x5002) << 8)|0x3004) = 0x403204$

### NJ Classification

How can instructions be classified as NJ? The opposite of J would be: if the register/flag *r* has passed its information on to another register/flag *o* before being overwritten, the instructions operating on *r* can be classified NJ. However, *o* can be overwritten as well. In that situation, the previous instructions operating on *r* should not automatically be classified as NJ.

This situation illustrates a key point: Something which is predefined as NJ should be found - a "safe haven" classifying everything that flows into it as NJ. In light of the discussion on memory in the previous section 5.2.2, it is natural that when junk instructions not utilising memory is not supported, memory should be predefined as NJ. All the instructions in the example in table 5.1 would then be classified as NJ.

### Misclassifications

It is necessary to examine the consequence of an erroneous classification of a sequence of instructions. There are two cases of classification errors:

**False negatives** A sequence of instructions is classified as NJ, but it is known a'priori to the classification to be J.

**False positives** Vice versa, a sequence of instructions is classified as J, but it is known a'priori to the classification to be NJ.

When J code is classified as NJ, a heuristic scan would report the junk-code as non-junk, resulting in false negatives. The time used by a virus analyst to understand/detect the virus is increased, since the instructions reported as NJ would confuse the analysis.

When NJ code is classified as J, a heuristic scan would report the normal code as junk resulting in false positives. If the virus analyst trust the framework, the instructions reported as J would be ignored/discarded in the further inspection of the virus. When this type of misclassification occurs, the picture of the virus' true function is corrupted, because NJ-instructions are missing and the semantic of the program is incorrect.

False negatives are worse than false positives for heuristic scanning, but better for manual analysis, and vice versa. It is more important to improve the manual analysis of a virus than to improve a heuristic scan. Therefore, false positives must not occur and the amount of false negatives should be minimised.

### 5.2.3 Elements of the Framework

As illustrated in figure 5.1 on the preceding page, the framework receives instructions as input and outputs the classification of each instruction. From 2.2.2 on page 24, an instruction is made from one opcode and zero or more operands. The two components and their implications on the framework are discussed here.

#### Operands

From the initial discussion on static instruction information, the flows between the operands should be used, to correctly determine the classification of the instruction containing the operands. The *information flows* between all the explicit and implicit operands found in a sequence of instructions, must be analysed to investigate what information is overwritten and what information is stored in a predefined NJ location. The framework would need to recognise the operands and their accesses: For instance, the framework must identify EAX from EBX to detect that there has been an information flow from EBX to EAX in the instruction `sub EAX,EBX`. In the sub instruction, EAX and EBX is read, then the result of the operation EAX-EBX is stored (write) in EAX. A *flow* from EBX to EAX has therefore occurred.

The discussion on NJ revealed the need for predefined NJ locations, from now on referenced as *NJ-operands*. What operand types should be defined NJ is discussed further in the thesis.

#### Opcodes

Because of the initial goal to use only static instruction information, the opcode information should not in general be used in the classification process. The result of the previous subtraction, i.e. the outcome of the operation EAX-EBX is not necessary to calculate to recognise that information in EBX has flowed into EAX. It is the *presence/absence* of a flow between operands that is important to record and analyse, not the exact value of each operand and operation.

### 5.2.4 Summary

The requirements and key points discussed are summarised in this section. The following summary is used as a reference point in further discussions in the thesis.

**J** When an operand is overwritten, and the information found in the operand has not been transferred to another operand or used in any other operation characterised as non-junk, the instructions referring the operand can be classified as J.

**NJ** When the information found in an operand is transferred to a "NJ-operand", or the information is utilised in an operation which can be characterised as non-junk, the instructions referring the operand can be classified as NJ.

**False positives** NJ classified as J must be avoided at all cost, because it will corrupt the correct picture of the virus true function. This is termed *the primary requirement*.

**False negatives** J classified as NJ should be avoided if possible. This requirement is termed *the secondary requirement*.

**Linearity** Unless there are situations which violate the primary requirement, or the secondary requirement heavily, the framework must follow the recursive traversal algorithm which runs in linear time, $O(n)$. This requirement is termed *the linear requirement*.

This section discussed the preliminaries of a JID framework. The following section contains the building blocks and developed logic of the JID framework, making it possible to adapt JID to the x86 assembler language.

## 5.3 Definitions, Rules and properties of the Framework

This section contains the theoretical foundation for creating the JID framework. The definitions and rules in this section are used both in the adaption of JID to the x86 assembler language in section 5.4 on page 90, and in the development of the JID plugin previously discussed.

The section is organised as follows. First, basic terminology is introduced. This is terminology used in the more complex rules and definitions. Second, *information flows* are defined. Third, *instruction chains* are defined. An instruction chain is a structure for ordering multiple instructions based on certain properties. Fourth, the term *entities* is defined. An entity is a structure uniting instruction chains and information flows. Last, how to classify entities as J, NJ or U is discussed.

### 5.3.1 Basic Terminology

#### Classification

The term classification is used frequently, justifying the following shorthand definition: A classification of a classifiable element X is written as $CL(X) = Y$, meaning X is classified as Y. For instance, instead of writing *"... then the classification of instruction i is NJ... "*, it will be written *"... then CL(i)=NJ... "*.

#### Instruction Index

Because each disassembled instruction is traversed only once by the recursive traversal algorithm, each instruction has a unique position in the output of the algorithm. The position is denoted *the instruction index* and uniquely identifies an instruction in the output of the recursive traversal algorithm. Figure 5.2 on the next page explaining instruction chains, also illustrates the concept of instruction indexes.

#### Current Instruction and Current Instruction Index

The *current instruction* is the instruction presented to the framework for classification by the disassembler. The *current instruction index* is the instruction index of the current instruction.

### 5.3.2 Information Flows

From the discussion in section 5.1.3 on page 76, information flows between operands must be recorded and analysed, in order to classify instructions correctly. The following definition contains the concept of information flows between the different operands in a single instruction.

**Definition 5.3.1** (Instruction Flows)**.** An instruction $i$ containing the set of operands $R$, contains zero or more *information flows*.
Each $r \in R$ with READ and/or EXECUTE access, flows to all $w \in R$ with WRITE access. This is written as: $r \rightarrow w$.
If $w$ has OVERWRITE access, it is written as $r \Rightarrow w$.

Because JID should be developed as a plugin to Gdasm, the disassembling library used is libdisasm (section 3.1.4 on page 50). For each operand in a disassembled instruction, libdisasm provides the access of the operand through the construct `enum x86_op_access`. It is stated in `libdis.h` of libdisasm [4]: *... by definition, Execute Access implies Read Access and Not Write Access.* Hence the EXECUTE-part in the definition. The OVERWRITE access is not defined as a separate access in libdisasm, but from the discussion in section 5.2.2 on page 78, it is necessary to identify the OVERWRITE access. OVERWRITE is defined as a WRITE access together with a set of special opcodes described in section 5.4 on page 90 and subsections. All definitions, rules, statements and proofs are established using $\rightarrow$ (WRITE), because $\Rightarrow$ (OVERWRITE) implies $\rightarrow$ (WRITE).

### 5.3.3 Instruction Chains and Current Instruction

Most instructions do not contain an information flow classifying the instruction *itself*. For instance, there is no J- or NJ-classification operations in the instruction

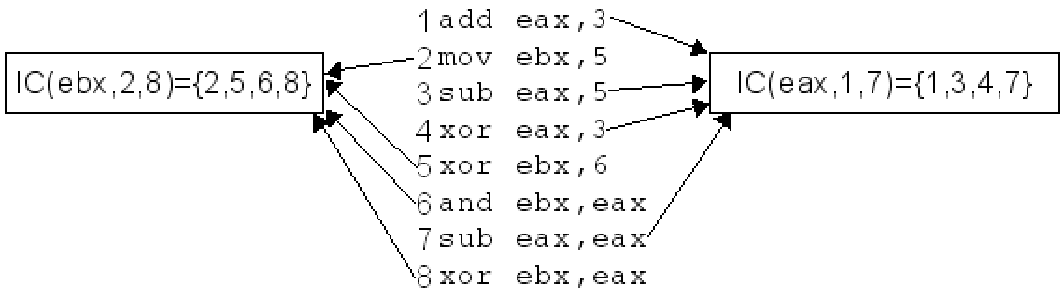


Figure 5.2: Each number to the left of the instruction is its index.

`sub EAX, EBX`. As seen in the Zmist decryptor D1 in section 3.4 on page 60, the instructions marked blue don't contain an operation that could trigger a J classification. For instance, (D1,3) and (D1,4) operate on the ECX register and the flags. Because both (D1,3) and (D1,4) apply operations on ECX and flags which is overwritten in respectively (D1,6) and (D1,13), both (D1,3) and (D1,4) is classified J. This motivates the definition of an instruction chain with respect to an operand.

**Definition 5.3.2** (IC). An *instruction chain (IC)* is a list of instruction indexes. Common for all the instructions referred to in an IC, is the occurrence of an operand $r$. The notation $IC(r, start, stop)$, defines the indexes to instructions which refers $r$, starting with the instruction at index $start$ and ending with the instruction referring $r$ at index $stop$. The inequality $start \leq stop$ is valid, because of the unique, auto-incrementing indexing of each instruction in the recursive traversal algorithm, .

An example of an IC is illustrated in figure 5.2. If the start/stop instructions are implied by the context or not important then $IC(r, start, stop)$ can be referred to as $IC(r)$. Rules considering when an IC should start and stop is defined later in section 5.3.6 on page 89 about "Entity Lists". For now, assume the start position of an IC is equal to the index of the first instruction included in IC. Further assume the stop position of an IC is equal to the index of the last instruction included in the IC when the RT algorithm exits.

## IC Classification

One of the main ideas behind defining IC's, is that all instruction indexes in an IC(r) should receive the same classification. This is because classification is done based on the "fate" of an operand in an instruction. The following rule establishes the classification of an IC.

**Rule 5.1.** An $IC(r)$ can be classified as X: $CL(IC(r)) = X$. This is equivalent to the following logic: $\forall i \in IC(r), CL(i) = X$

## Instruction inclusion in IC

The following rule defines how the current instruction index is included in an IC.

**Rule 5.2.** When the current instruction $i$ contains the flow $v \rightarrow r$ for operands $r$, $v$, then the index of $i$ should be added to $IC(r)$.

If there doesn't exist any flow $v \rightarrow r$ in $i$, then $i$ doesn't contribute to altering any operand and $CL(i) = NJ$. $i$ can be J itself, for instance the `NOP` instruction (no operation) can be inserted randomly, *acting* as J. Note that `NOP` is also used in clean code and is not per definition a junk instruction. `NOP` can be detected as a separate junk instruction by searching for the instruction in the output of the recursive traversal algorithm. This can easily be done without the use of the framework established in the thesis, and is not considered further.

**Prioritisation of Classifications in an IC**

A problem could occur if the current instruction $i$ contains operands $v_1, v_2, r_1, r_2$ with flows such that $r_1 \rightarrow v_1$ and $r_2 \rightarrow v_2$. By the rule defined, the index of $i$ would be added to both $IC(v_1)$ and $IC(v_2)$. If $CL(IC(v_1)) \neq CL(IC(v_2))$, the classification would be ambiguous, unless the different classifications are prioritised. With background in the primary and secondary requirements and the discussion on U in section 5.2.1 on page 77, the following rule establishes the prioritisation of the different classifications.

**Rule 5.3.**
1. If initially $CL(IC(r)) = NJ$, then the classification of $IC(r)$ cannot be changed to another classification.
2. If initially $CL(IC(r)) = U$, then the classification of $IC(r)$ can be changed to NJ.
3. If initially $CL(IC(r)) = J$, then the classification of $IC(r)$ can be changed to NJ or U.

To illustrate the implications of this rule: Given an instruction $i$, $CL(i) = J$ if and only if $\forall$ operands $v$ contained in $i$, $CL(IC(v)) = J$.

## 5.3.4 Flow set, FL

When classifying, it is not a particular instruction itself that is interesting, but the presence/absence of a flow in an interval of instruction indexes. This motivates the definition of a flow set, *FL*, and its uses becomes apparent when considering the J/NJ classification rules.

**Definition 5.3.3** (FL)**.** Associated with $IC(r, start, stop)$, there is a set FL(r). FL(r) is the set of all information flows referencing $r$, found in the instructions pointed to by the indexes in the interval $[start, stop]$.

In an interval of instruction indexes [start,stop], FL(r) contains all flows found in the instructions pointed to by the index interval. The flows are of the type $x \rightarrow r$ and

$r \rightarrow x$, where x is an arbitrary operand. Note that in FL(r), both flow directions are present, but in IC(r) instruction only containing flows of the type $v \rightarrow r$ are indexed. The discussion on J/NJ classification in section 5.3.6 on the following page, reveals why both flow directions must be present in FL(r).

### 5.3.5 Entities

The two elements IC and FL are bound together. Most of the time it is necessary to reference flows in FL(r), but the context is lost unless the start/stop positions of IC(r) is referenced together with FL(r) as well. An *entity* unites the IC and the FL together in one element. The primary motivation for defining an entity is to simplify notation. Another important motivation is that the definition of entities can be used directly in the implementation of JID, namely a C++ class containing an IC and the associated FL. Therefore, when entities are used throughout the thesis, it is easier to convert the theory/design into a working implementation.

#### Definition and Properties

**Definition 5.3.4** (Entity)**.** Given an operand $r$, the entity $r_e$ associated with $r$ is defined as an ordered pair $r_e = (IC, FL)$, where IC is a instruction chain for the operand $r$, $IC(r, start, stop)$. FL is the set of all information flows referencing $r$, found in the instructions pointed to by the indexes in the interval $[start, stop]$.

The separate elements of an entity can be accessed in the following object-oriented manner:

- The $IC(r)$ of entity $r_e$ can be accessed as $r_e.IC$
- The $FL(r)$ of entity $r_e$ can be accessed as $r_e.FL$

How instructions are added to $r_e.IC$ was previously described in rule 5.2 on the preceding page. Note that in the definition of FL, not only the flows contained in the instructions pointed to by the indexes in $r_e.IC$ is included in $r_e.FL$, but $\underline{\text{all}}$ flows referencing the operand $r$ in the *interval* given by $r_e.IC$.

#### Entity Terminology

- $CL(r_e)$ is defined as: $CL(r_e) = CL(r_e.IC)$. In words, $r_e$ can be classified as NJ, J or U, meaning that it is $r_e.IC$ which is classified as NJ, J or U.
- Because of the frequent use of the term information flow, the following short-form is defined: $r_e$ can be said to flow to another entity $x_e$, denoted as $r_e \rightarrow x_e$. This is the equivalent to the following logic:
  $\exists(r \rightarrow x) \in r_e.FL \cap x_e.FL$

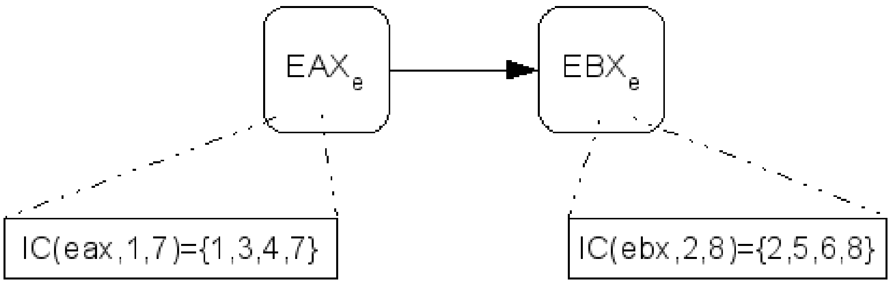


Figure 5.3: Illustrating corresponding entities to the IC created in figure 5.2 on page 83

| # | **Instruction** | **Flow equivalent** |
|---|---|---|
| 1 | `mov EAX, 4` | $4 \Rightarrow EAX$ |
| 2 | `mov EBX, 3` | $3 \Rightarrow EBX$ |
| 3 | `add EAX, EBX` | $EBX \rightarrow EAX$ |
| 4 | `mov [mem], EAX` | $EAX \Rightarrow NJ-operand$ |

Table 5.2: Example illustrating NJ Classification and NJ recursion rule

To illustrate the use of entities, figure 5.3 illustrates the entities created from the instructions used in figure 5.2 on page 83.

## 5.3.6 J and NJ Classifications of an Entity

The following rule follows naturally from the discussion on NJ and the rule of instruction inclusion (rule 5.2 on page 84). Because of rule 5.2, the current instruction index is not added to any IC and must be classified separately in the following rule.

**Rule 5.4** (NJ classification of entities)**.** Let $r$ be an operand, $v$ a NJ-operand and $r_e$ the entity associated with $r$. If the current instruction at index $i$ contains the flow $r \rightarrow v$, then $CL(r_e) = NJ$ and $CL(i) = NJ$.

The *NJ Classification Recursion* rule takes into account classification of other "helping" instructions when an entity has been classified as NJ. Table 5.2 is an example motivating this rule. In the example, instruction 1,3 and 4 is classified NJ by rule 5.4. Because EBX flows into something classified as NJ in instruction 3, it is natural that instruction 2 is classified as NJ as well.

**Rule 5.5** (NJ Classification Recursion)**.** If an entity $r_e$ is classified as NJ, then $\forall w_e$ such that $w_e \rightarrow r_e$, set $CL(w_e) = NJ$.

In conjunction with the discussion on FL(r) and flow inclusion in section 5.3.4 on page 84, the NJ Classification Recursion rule illustrates why FL(r) always has to include flows referencing $r$ of the type $x \rightarrow r$. Assume flows of the type $x \rightarrow r$ is <u>not</u>

included in FL(r), but $r \rightarrow x$ are. To classify IC(r) as NJ by the classification rule defined, a search for the flow $x \rightarrow r$ must be performed on all FL(x), where x refers all operands recognised by the framework. This does not scale well, for instance if detection of junk utilising memory should be implemented in the future. It is therefore necessary to include flows of the type $x \rightarrow r$ in FL(r).

### U Classification

Because U is the classification category for special cases, how an entity is classified as U is discussed further in the text. However, if an entity $r_e$ is classified as U, then all entities providing $r_e$ with information must also be classified U. This is equal to the NJ recursion rule 5.5 on the facing page, except that entities matching the condition is classified as U instead of NJ.

### J Classification

The following rule follows naturally from the discussion on J, and the rule of instruction inclusion 5.2 on page 84.

**Rule 5.6** (J Classification of Entities)**.** Let $r$, $v$ be operands and $r_e$ the entity associated with $r$. If the current instruction contains flow $v \Rightarrow r$ and $\nexists\, x_e$ such that $(r_e \rightarrow x_e)$, then $CL(r_e) = J$.

The example in table 5.3 illustrates why there cannot exist a flow from $r_e$ to $x_e$ for $r_e$ to be classified J.

| # | Instruction | Flow equivalent |
|---|---|---|
| 1 | `mov EAX, 4` | $4 \Rightarrow EAX$ |
| 2 | `add EBX, EAX` | $EAX \rightarrow EBX$ |
| 3 | `mov EAX, 5` | $5 \Rightarrow EAX$ |

Table 5.3: Example illustrating J rule

In instruction 2, EAX flows to EBX. Instruction 1 cannot be classified as J in instruction 3, because the information is still “alive” in EBX, due to instruction 2.

In conjunction with the discussion on FL(r) and flow inclusion in section 5.3.4 on page 84, the J rule illustrates why FL(r) has to include flows referencing $r$ of the type $r \rightarrow x$. The argument is equal to the previous argument on why flows of the type $x \rightarrow r$ must be included in FL(r), and is not repeated here.

### J Classification Resolving

Consider the example in table 5.4 on the following page. This example is an expanded version of the example given in table 5.3.

| # | Instruction | Flow equivalent |
|---|---|---|
| 1 | `mov EAX, 4` | $4 \Rightarrow EAX$ |
| 2 | `add EBX, EAX` | $EAX \rightarrow EBX$ |
| 3 | `mov EAX, 5` | $5 \Rightarrow EAX$ |
| 4 | `mov [mem], EAX` | $EAX \Rightarrow NJ - operand$ |
| 5 | `mov EBX, 6` | $6 \Rightarrow EBX$ |

Table 5.4: Expanded example illustrating J rule

By examining the instructions intuitively, that is without using any classification rules, it is obvious that instruction 1 and 2 is J, because instruction 3 and 5 overwrites the information produced in 1 and 2. Instructions 3 and 4 is NJ. When applying the J/NJ rules defined, the following outcome is observed.

Before instruction 3 is evaluated, the following IC's exist: $EAX_e.IC = \{1\}$, $EBX_e.IC = \{2\}$. When instruction 3 is evaluated, the J classification rule cannot classify $EAX_e$ as J (as discussed in the previous example with table 5.3 on the preceding page). Instruction 3 is therefore included in $EAX_e.IC$: $EAX_e.IC = \{1,3\}$. At instruction 4, by NJ classification rule, $EAX_e$ and instruction 4 is NJ. This is erroneous, and by the classification prioritisation (rule 5.3), the classification is irreversible.

The example illustrates two key points. First, multiple "versions" of an entity $r_e$ must exist, i.e. $r_e^1, r_e^2$ and so on. Second, an additional rule for J classification similar to the NJ Classification Recursion must be defined. Because this rule would rely on the multiple entity versions, the definition of *entity arrays* is given before the rule is defined.

**Entity Arrays**

**Definition 5.3.5** (Entity array)**.** An *entity array* $EL_r$ is an ordered array of zero or more $r_e$ entities: $EL_r = \{r_e^1, r_e^2, \ldots, r_e^n\}$. The array is ordered such that for $r_e^i.IC(r^i, start^i, stop^i)$, where $1 < i \leq n$, the boundary $stop^{i-1} < start^i$ holds. $stop^{i-1}$ is the stop index of $r_e^{i-1}.IC(r^{i-1})$.

Each entity created by the framework is inserted into one entity array and one only. A *current entity* is the last element added in $EL_r$. There is one current entity for each entity array. The $IC(r, start, ..)$ component of a current entity, does not receive any stop position until a new entity is added to the array.

**When new entities are inserted in entity arrays**

Because multiple IC's of an operand $r$ exist, a rule defining start/stop positions of the different IC's must be created.

**Rule 5.7.** A new entity $r_e^{n+1}$ is created, inserted into $EL_r$ and made the current entity of $EL_r$, when the current instruction $i$ with index $pos_i$ contains operands $r$ and $v$ with the two properties listed under. In both properties, the current instruction index $pos_i$ is added to $r_e^{n+1}.IC$ and the start position of $r_e^{n+1}.IC$ is $pos_i$. The stop position of $r_e^n$ is the index number, of the last instruction added to $r_e^n$.

1. $v \Rightarrow r$. $r$ is overwritten and $r_e^n$ must be preserved to include the possibility of delaying the classification. Because the information in $r_e^n$ is overwritten, the flow $r_e^n \rightarrow r_e^{n+1}$ should not be inserted in the respective entities.

2. $v \rightarrow r$ and the flow $r_e^n \rightarrow x_e$ for another entity $x_e$ exists. Because the information in $r_e^n$ is modified and not overwritten, the flow $r_e^n \rightarrow r_e^{n+1}$ must be inserted in the respective entities.

By including the pt. 1 in the rule, classification of $r_e^n$ can be delayed after i is processed. This is beneficial in the implementation, because classification can be performed after the current instruction is processed.

### 5.3.7 J Classification Resolving

Having established entity arrays, the following rule can be defined.

**Rule 5.8** (J Classification Resolving)**.** If an entity $r_e$ is classified as J, then
$\forall x_e$ such that $x_e \rightarrow r_e$, $x_e$ isn't the current entity of $EL_x$ and $\nexists\, y_e(x_e \rightarrow y_e$ where $y_e$ is not classified), set $CL(x_e) = J$

Because of the ordering in $EL_x$ (def 5.3.5), if $x_e$ isn't the current entity, the information contained in $x_e$ has been passed on or is overwritten (rule 5.7), hence the information can be viewed as "obsolete". If all the $y_e$ which $x_e$ passed information to is classified, then $x_e$ can be classified as J. Because of the classification prioritisation (rule 5.3), if $CL(x_e)$ is already classified and $CL(x_e) \neq J$, then $x_e$ cannot be classified J.

Revisit example 5.4 on page 88 and consider the following. As previously, before instruction 3 is evaluated, the following IC's exist: $EAX_e^1.IC = \{1\}$ and $EBX_e^1.IC = \{2\}$. When instruction 3 is evaluated, because $5 \Rightarrow EAX$, a new entity $EAX_e^2$ is created and made current entity in the entity list $EL_{EAX}$ and $EAX_e^2.IC = \{3\}$. When instruction 4 is evaluated, by NJ Classification rule 5.4 on page 86, $EAX_e^2$ and instruction 4 is NJ. When instruction 5 is evaluated, by the J Classification rule 5.6 on page 87, $EBX_e^1$ is J. By the J Classification Resolving rule 5.8 on the preceding page, $EAX_e^1$ is also classified J. All elements in example 5.4 on page 88 have now been classified correctly using the rules defined.

### Classifying Instruction

From the definitions of the J Classification rule5.6 and the NJ Classification rule 5.4, a single instruction is responsible for classifying one entity, which again by the NJ Recursion rule 5.5 and the J Resolving rule 5.8 can classify several entities. All U classification cases are also triggered by one single instruction as seen further in the text.
It is therefore reasonable to use the term *classifying instruction* for an instruction which triggers classification of one or many entities.

## 5.4 x86 Assembler Language and JID

In this section, details of the x86 assembler language concerning JID is enumerated and discussed. As discussed in section 5.2.3 on page 80, the framework should recognise the operands of an instruction. The list of operands is produced here [18].

- Immediate operands
- Memory operands
- Register operands

Immediate operands are discussed in section 5.4.1. Memory operands are discussed in section 5.4.2. Register operands are split into several different types of registers and are discussed in sections 5.4.3, 5.4.4, 5.4.5, 5.4.6 and 5.5.

### 5.4.1 Immediate Operands

When the current instruction $i$ contains an immediate value as operand, another operand $r$ with associated entity $r_e$, $r \neq imm$, is often present. The information flow contained in $i$ is often such that $imm \rightarrow r$. $r \rightarrow imm$ is obviously not possible, and therefore it is not necessary to maintain an entity associated with $imm$. The instruction referring $imm$ is by rule 5.2 on page 84, added to $r_e.IC$.

If there are instructions in which the immediate operand does not affect any other operand, the instructions should by the discussion following rule 5.2 on page 84 be classified NJ.

### 5.4.2 Memory Operands

Memory operands constitute of immediate values, segment registers and general-purpose registers [18]. In the instruction `sub [EAX+0x8],0x32`, [EAX+0x8] is an example of a memory operand. As discussed in 5.2.2 on page 78, the memory operand should be defined as a NJ-operand.

#### Memory Operands containing Registers

The framework is not concerned about the *content* of a memory address (section 5.2.2 on page 78). Because immediate values are of no value to the framework, the only information interesting is whether a memory operand contains registers or not. The following rule establishes how registers contained in memory operands should be treated.

**Rule 5.9.** Let $v$ be an operand with associated entity $v_e$, $r$ a register with associated entity $r_e$, $m$ a memory operand *containing* $r$. When the current instruction contains flows $v \rightarrow m$ or $m \rightarrow v$, the flow $r \rightarrow v$ should be added to $r_e.FL$ and $v_e.FL$, independent of the flow direction between $m$ and $v$.

The following discussion illustrates the reasoning behind the rule. Let $v$, $r$ and $m$ be defined as in the rule. Assume $v \rightarrow m$. It is not correct to presume $r \rightarrow m$, since this implies that information flows *from r to m*. Similar, assume $m \rightarrow v$. It is not correct to presume $m \rightarrow r$, since this implies that information flows *from m* or *to r*. If $v \rightarrow m$, *r aids v* pass data to a correct destination. If $m \rightarrow v$, *r aids v* retrieve data from a correct source. Regardless of flow direction, $r$ provides $v$ with information and $r \rightarrow v$ can be added to $r_e.FL$ and $v_e.FL$.

**Statement 5.4.1.** *Rule 5.9 doesn't violate the primary or secondary classification requirement*

| # | Instruction | Flow equivalent |
|---|---|---|
| 1 | `mov AX, value1` | $value1 \Rightarrow EAX$ |
| 2 | `rol EAX, 16` | value1 is rotated 16 bits left in EAX ($16 \rightarrow EAX$) |
| 3 | `mov AX, value2` | $value2 \Rightarrow AX$ |
| 4 | `mov NJ-op, EAX` | $EAX \Rightarrow$NJ-operand |

Table 5.5: After instruction 3 is evaluated, value1 resides in the 16 msb of EAX while value2 resides in the 16 lsb of EAX, see fig 5.4.

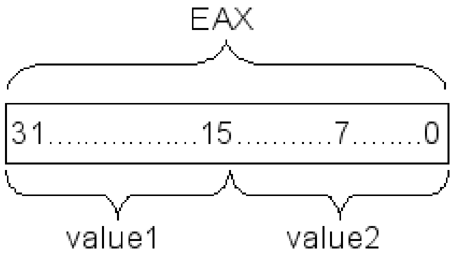


Figure 5.4: The layout of EAX from the example in table 5.5

*Proof.* Let $v$, $r$ and $m$ be defined as in the rule. There are two possible information flows between $v$ and m:

$v \rightarrow m$ $v_e$ is by definition NJ. $r_e$ would be classified as NJ by the NJ recursion rule 5.5.

$m \rightarrow v$ $v_e$ can be classified as J or NJ. If $v_e$ is classified as NJ, then $r_e$ would also be NJ. If $v_e$ is classified as J, then $r_e$ is not automatically classified as J, except if $r_e$ satisfies the conditions of the J resolving rule 5.8, and then $r_e$ would properly be J.

□

### 5.4.3 32-, 16- and 8-bit General-Purpose Registers

The general-purpose registers were described in section 2.2.3 on page 25. ESP and SP are discussed separately in section 5.4.5 on page 94, and are ignored in this section.

Summed together, a total of 22 different registers with associated entities must be identified by the framework. The following section discusses a solution which allows the framework to recognise the 32-bit registers only.

#### The 8- and 16-bit Register problem

**Definition 5.4.2.** A *subregister* is a x-bit register $r1$ which addresses a part of an y-bit register $r2$, $x < y$. When a register is not a subregister, it is termed *superregister* when

| # | **Instruction** | **Flow equivalent** |
|---|---|---|
| 1 | `add EAX, value1` | $value1 \rightarrow EAX$ |
| 2 | `mov AL, value2` | $value2 \Rightarrow EAX$ |
| 3 | `mov AL, value3` | $value3 \Rightarrow EAX$ |
| 4 | `mov NJ-op, EAX` | $EAX \Rightarrow$NJ-operand |

Table 5.6: The "Flow equivalent" column contains the flow equivalent if AL is treated as EAX. At instruction 4, the 24 msb of value1 and value3 is moved into a NJ entity.

the context requires separate identification of the two terms.

To illustrate the use of the term subregister: AL is a subregister of both AX and EAX, and AX is a subregister of EAX. Of the three registers, only EAX is a superregister. The example in table 5.5 on the preceding page illustrates the problem with subregisters. When the instructions are run, both value1 and value2 are stored in the NJ-operand. To detect that value1 has been rotated into the upper 16 msb of EAX, `rol` must be emulated or logic handling subregisters must be defined. As previously discussed, emulation should be avoided, leaving the option of defining a logic handling subregisters.

### The 8- and 16-bit Register solution

The least complex rule would be to treat each subregister as the corresponding superregister. Assume this rule is applied and consider the example in table 5.6. Instruction 2 leads to an overwrite of EAX and instruction 1 would be classified as J. Assume the 24 msb of value1 is NJ and value3 is NJ. Then NJ is erroneously classified as J in instruction 1 and this violates the primary classification requirement.

To avoid the misclassifications illustrated, the rule must be extended: treat each subregister as the corresponding superregister in all situations except for subregister overwrites. In the case of subregister overwrites, the overwrite must be treated as a regular write. Assuming this extended rule is applied, example 5.6 doesn't violate the primary classification requirement. Instead, the example violates the second requirement. This is not optimal, but as discussed in section 5.2.2 on page 79, better than having false positives. The following rule states what was found in the discussion.

**Rule 5.10.** Let $r^x$ be a subregister of a register. When the current instruction $i$ refers $r^x$, then $r^x$ should be substituted in the framework with the superregister of $r^x$. There is exactly one exception: If $i$ contains an operand $v$ such that $v \Rightarrow r^x$, this flow should be treated as $v \longrightarrow r^x$.

**Statement 5.4.3.** *Rule 5.10 does not violate the primary classification requirement.*

*Proof.* It must be proved that NJ cannot be classified as J. Let $r^x$ be a subregister. Let $r^f$ be the superregister of $r^x$. From rule 5.10, the register $r^x$ has associated entity $r^f_e$.

| # | Instruction | Flow equivalent |
|---|---|---|
| 1 | `add ESP, value1` | $value1 \rightarrow ESP$ |
| 2 | `xor EAX, EBX` | $EBX \rightarrow EAX$ |
| 3 | `sub ESP, value1` | $value1 \rightarrow ESP$ |
| 4 | `push EAX` | $EAX \Rightarrow [ESP]$ |

Table 5.7: Instructions 1 and 3 is J

Further assume the classification of $r_e^f$ is a'priori known as NJ. From rule 5.10, the only operation recognised by the framework that overwrites $r^x$, is the overwrite of $r^f$ and such an overwrite will possibly classify $r_e^f$ as J. Because $r^f$ is the superregister of $r^x$, an overwrite of $r^f$ will also overwrite $r^x$ when the instructions are run on a processor. This contradicts the assumption that $r_e^f$ was NJ and therefore $r_e^f$ cannot be classified J if it is NJ.

□

### 5.4.4 Segment-, FPU-, MMX-, XMM-, Control-, Debug- and MSR-registers

Segment-, FPU-, MMX-, XMM-, control-, debug- and MSR-registers are for the sake of simplicity ignored. The framework will not be able to recognise junk-instructions utilising these registers. It is possible to expand the framework with support for the registers without breaking the framework's logic. The following rule defines what action must be taken when one of the unsupported registers is encountered.

**Rule 5.11.** Assume the current instruction $i$ contains operands $r$ and f. Let $r$ be an operand recognised by the framework with associated entity $r_e$. Let f be an operand not recognised by the framework. If $r \rightarrow f$, then $CL(r_e) = U$.

In the rule, the correct classification of $r_e$ cannot be determined because the information flows to f, and cannot be tracked further. Therefore $r_e$ must be classified U. If $f \rightarrow r$, then f can be viewed as an immediate operand and no special considerations apply.

### 5.4.5 ESP Register (Stack)

The stack can be viewed as a memory operand containing the ESP register. Each instruction that pushes an operand $r$ on the stack, contains the flow $r \Rightarrow [ESP]$. Each instruction that pops an element from the stack into an operand $r$, contains the flow $[ESP] \Rightarrow r$. The `pop` instruction must be recognised specifically by the framework, because it contributes to defining the OVERWRITE access (as discussed in 5.3.2 on page 82).

| # | Instruction | Flow equivalent |
|---|---|---|
| 1 | `add EAX, 1` | $imm \rightarrow EAX$ |
| 2 | `cmp EAX, 2` | |
| 3 | `mov EAX, 3` | $imm \Rightarrow EAX$ |
| 4 | `jz location` | |

Table 5.8: Flag usage

It is difficult for a virus programmer to both utilise ESP as a junk-register, and using the stack simultaneously. There exist examples, and one constructed example is illustrated in table 5.7 on the preceding page. Instructions 1 and 3 can be classified correctly by the framework only if ESP is overwritten before the push instruction (4) occurs. It is assumed that instructions overwriting ESP appears seldom because the ESP controls the stack which is utilised frequently, and ESP is therefore defined as a NJ-operand.

The register SP is a subregister of the superregister ESP and would be a NJ-operand as well.

### 5.4.6 EFLAGS Register

The EFLAGS register contains several flags. The seven flags listed in section 2.2.3 on page 25 must be recognised by the framework because nearly all instructions set, clear or test one or more of these flags. Table 5.8 gives an example of a situation where flags not recognised by the framework result in a violation of the primary requirement. In the example, instruction 4 is a control transfer instruction changing the flow of execution to “location” if the zero flag is set (jump if zero). The compare in instruction 2 set, amongst others, the zero flag if the subtraction EAX-2 is equivalent to 0. Unless flags are recognised by the framework, instruction 1 and 2 would be classified as J when instruction 3 is evaluated.

#### Instructions Setting/Testing Separate Flags

Flags can be viewed as implicit operands to instructions which are read/written. How each instruction utilise the flags are hard-wired in the instruction and reported by libdisasm. For instance, the opcode `ADD` set/clear the OF, SF, ZF, AF, CF and PF flag according to the result of the computation [16].

**Rule 5.12.** Let F be the set of the seven flags described and $\forall f \in F$ there is an associated entity $f_e$. Let $i$ be an instruction which contains operand set $r$ and $\forall$r$\in R$ there is an associated entity $r_e$.

**Flag set/clear** $\forall f \in F$ which $i$ SET/CLEAR, and $\forall$r$\in R$ where $r$ has READ or EXECUTE access, the following flow is added to $r_e.FL$ and $f_e.FL$: $r \Rightarrow f$.

**Flag testing** $\forall f \in F$ which $i$ TEST, and $\forall$r$\in R$ where $r$ has WRITE access, the following flow is added to $r_e.FL$ and $f_e.FL$: $f \rightarrow r$.

When the rule defined is taken into account, the example in table 5.8 on the preceding page yields the following result. Instruction 2 contains the following flow: $EAX \Rightarrow$CMP-flags. In instruction 3, the J classification rule can therefore not classify $EAX_e$ as J. Instruction 4 contains the following flow: CMP-flags $\rightarrow$JZ. Assuming JZ is defined as NJ (discussed in the following section), by the NJ recursion rule 5.5 $EAX_e$ is NJ.

## 5.5 EIP Register and Control Transfer Instructions

Control transfer instructions have implications on the classification process, and each control transfer instruction is therefore discussed separately. Control transfer instructions were explained in section 2.2.3 on page 27.

If the current instruction $i$ contains an operand $r$, and $i$ contains the information flow $r \rightarrow EIP$, then $r$ alters the execution of the program flow and should be classified as NJ. EIP should therefore be defined as a NJ-operand. Because all control transfer instructions modifies/writes to EIP, they are by the NJ Classification rule classified as NJ.

### 5.5.1 Framework Limitations

#### Unresolvable Control Flows

Instructions containing an unresolvable control flow were described in section 2.3.2 on page 29. When an unresolvable control flow occur, unclassified entities are affected. Assume $r_e$ is an unclassified entity when the current instruction contains an unresolvable control flow. The *classifying instruction i* which would classify $r_e$ if the control flow *had* been resolved, cannot be deterministically reached when the destination of the control transfer instruction is unresolvable. The classification of $r_e$ would then be unknown and this leads to the following rule:

**Rule 5.13.** Let $r$ be the set of all operands recognised by the framework, $EL_r$ an entity list containing $r_e$ entities. When the current instruction $i$ is an instruction containing an unresolvable control flow, the following action must take place: $\forall\ r \in r\ r_e \in EL_r$ where $r_e$ is an unclassified current entity, set $CL(r_e) = U$.

Because U-classification is recursive (see section 5.3.6 on page 87), all entities flowing into $r_e$ is also classified U.

**Control transfer instructions pointing to visited positions in the RT algorithm**

Assume the current instruction $i$ is a control transfer instruction pointing to an instruction at $pos_j$, previously disassembled and processed by the framework. The linearity requirement would be violated if $pos_j$ was to be visited again and the framework would need to communicate to the disassembler that it must process instructions from $pos_j$ and further once more. The framework logic would be more complex and this feature should be left out for now. Unclassified entities should therefore follow rule 5.13 on the facing page when control transfer instructions pointing to visited positions in the disassembling occur.

### 5.5.2 JMP, CALL, RET, IRET and End Of Disassembling

#### JMP

The classification process is not affected by instructions of the type `jmp imm`, because the flow of the code is uniquely determined. If the instruction pointed to by imm has previously been disassembled and processed, rule 5.13 on the preceding page covers the classification of entities. Instructions like `jmp [reg]` and `jmp reg` are unresolvable and rule 5.13 on the facing page covers the classification of unclassified entities when such instructions appear. Note that if `jmp [r]` or `jmp r` is encountered, then the associated entity of $r$, $r_e$, is classified NJ because $r \rightarrow EIP$.

#### CALL

Except for the return address pushed on the stack, the call instruction works the same way `jmp` instructions do. `call imm` is in this context equivalent with `jmp imm` discussed in the previous section. When `call reg` and `call [reg]` is encountered, rule 5.13 on the preceding page covers the classification of unclassified entities.

#### RET/IRET

When a `ret` is encountered, the topmost element on the stack is popped into the EIP register. In the framework, the stack is viewed as memory. It is therefore impossible to determine the destination of the control transfer and rule 5.13 on the facing page covers the classification of unclassified entities when a `ret` appears. A `iret` instruction returns from interrupts and has additional tasks compared to a regular `ret`. However, when considering the *control transfer* aspect of `iret`, it is equal to `ret` and should follow the same procedure as `ret`.

| # | Instruction |
|---|---|
| 1 | `mov EAX, 1` |
| 2 | `cmp EAX, 1` |
| 3 | `je location` |

Table 5.9: Conditional jump in instruction 3 is always taken.

#### End Of Disassembling

When there are no more bytes left to disassemble, there can be unclassified entities. However, end of disassembling is not equivalent to end of program execution. Because of unresolvable control flows, an end of disassembling does not mean that the instructions executed on a processor would end at the same point the RT algorithm does. Therefore, rule 5.13 on page 96 covers the classification of the remaining unclassified entities when the RT algorithm has no more bytes to disassemble.

### 5.5.3 Conditional Jumps

Jccs are frequently used in program code and must be examined to investigate the implications on the framework. The immediate operand is the only operand to all the different Jcc instructions, and all Jccs are therefore resolvable.

Jccs are *deterministic* if it can be determined that the fall through branch or taken branch is reached for each time the Jcc is executed. *Non-deterministic* Jccs are the opposite of deterministic Jccs. Table 5.9 illustrates an example of a deterministic Jcc. In this example, EAX is always 1 when the compare at instruction 2 is evaluated. The result of the comparison is always true, and the branch in instruction 3 is always taken (je = jump if equal).

#### Loop Instructions

The `LOOP imm` instruction compares the ECX register to 0 and jumps to imm if the compare is false. Some versions of `LOOP` also compares the ZF flag in addition to the ECX register [16]. The `LOOP` instruction does the same as a Jcc instruction: it falls through or takes the branch, based on a comparison. LOOP is therefore not discussed any further.

#### The Problem with Jccs and Classification

If the entity $r_e$ is unclassified when a `Jcc imm` instruction occurs, both the fall through branch and the branch taken must be evaluated to classify $r_e$. $r_e$ can only be classified as J if the evaluation of both branches would result in $r_e$ being classified as J. This breaks the linearity requirement. Because Jcc instructions appear frequently in a program, the subject should however be discussed thoroughly to examine if it is possible to resolve the junk *without* breaking the linearity.

### Problem Discussion

How frequently does junk instructions capable of utilising Jccs appear? It is difficult for a virus to produce junk propagating through one or more non-deterministic Jccs. The accounting of the registers that can be used for junk and those that are used for non-junk operations, increases when the Jcc is non-deterministic. It is therefore assumed that situations where J instructions propagate through non-deterministic Jccs seldom appears, because creating such an engine would be extremely difficult.

Deterministic Jccs as illustrated in table 5.9 on the preceding page, can be utilised easier in a virus than non-deterministic Jccs because in the context of the virus engine, they are equal to `jmp` instructions. This suggest that the amount of undetectable J-instructions propagating through Jccs could be high if the deterministic Jccs of the type illustrated in table 5.9 on the facing page appear frequently in the virus.

### Problem Conclusion

From the discussion, if it is possible to separate the deterministic Jccs from the non-deterministic, resolving J over Jccs would not be a problem. This is so because if there is a deterministic Jcc it is reduced to a JMP instruction. If the Jcc is non-deterministic, it is assumed that no J instructions propagate over the Jcc. In the general case, determining if a Jcc is always taken or not is a variant of the Halting Problem. Consider the following:

```
start
..
compare
Jcc start
halt
```

Determining if the fall through branch is always taken in the Jcc, is equivalent to determining if the program will halt or not, i.e. the Halting Problem. It is therefore not possible to separate the deterministic Jccs from the non-deterministic Jccs in the general case.

What is left is the option of detecting junk propagating over Jccs, whether it is deterministic or non-deterministic. The framework should for now not include the possibility to detect J propagating over Jccs, because it would violate the linearity requirement and make the framework logic more complex. Rule 5.13 on page 96 covers the classification of unclassified entities when a Jcc occurs.

## 5.6 Instruction Groups

Section 2.2.4 on page 27 lists the different instruction groups of the x86 assembly language. This section enumerates this list and examines if each group is handled in the framework.

**Data Transfer Instructions** The data transfer instructions move data between registers, and between registers and memory. The instruction `mov` is an example of a data transfer instruction. The data transfer instructions overwrites their destinations and should therefore be recognised as OVERWRITE instructions (as discussed in section 5.3.2 on page 82 on information flows).

**Binary-, Decimal-, Shift/Rotate-, Flag Control- and Bit/Byte-Instructions** Instructions falling under one of these groups are attended to by the framework. They provide different operations on operands and are not special in any way. Instructions falling under one of these groups *might* overwrite an operand. Because in general this would require emulation, the overwrites would be ignored in the JID framework. This *could* lead to a violation of the secondary requirement, but not the primary.

**Control Transfer Instructions** This group was covered in section 5.5 on page 96.

**String Instructions** The string instructions operate on memory using the registers as pointers. This group requires no special considerations in the framework, because the instructions only operate on memory and registers.

**Enter and Leave Instructions** The instructions `enter` and `leave` perform function prologue and epilogue. In view of the framework, this is something accomplished with a sequence of data transfer instructions and binary arithmetic instructions and this group requires no special considerations in the framework.

**I/O Instructions** An I/O port should be defined as a NJ-operand. This is reasonable, because it is not likely for J instructions to write to machine-dependant I/O ports. When I/O ports are defined as NJ-operands, the I/O instruction group works like any other group reading/writing to NJ-operands in the framework.

**Segment Register Instructions** All instructions found in the segment register instruction group loads a value from memory into a segment register and a general-purpose (gp) register. The gp registers will be overwritten, and all instructions found in this group should be recognised as OVERWRITE instructions (as discussed in section 5.3.2 on page 82 on information flows).

**Miscellaneous Instructions** The `NOP` instruction was previously discussed in section 5.3.3 on page 83. `CPUID` and `LEA` should be defined as OVERWRITE instructions, because they overwrite the registers given as implicit/explicit operands in the instructions. `XLAT` reads/writes AL and needs no special consideration by the framework.

### 5.6.1 Design of JID summary

With this discussion, the design of JID is finished. Operands and instructions found in the x86 assembler language have been discussed, and this chapter provides the foundation for the implementation of JID. The source code of JID is found on the CD described in appendix A on page 125. Testing of JID is described in the next chapter. Before discussing the tests and results, similar work discovered late in the thesis progress is discussed in the following section.

## 5.7 Static Single Assignment

After completing the design of JID, a friend who has knowledge on the field of compiler design and optimisation, mentioned the equivalences between JID and techniques in compiler optimisation. In compiler optimisation, there are techniques for *dead code elimination*. Dead code is code that could be described as junk, but it is not placed intentionally in the source code. One of the techniques which can be used to detect such dead code is called *static single assignment* (SSA), and was developed by 5 IBM researchers in the 1980s. This section explains SSA, dead code removal using SSA (SSA/DCR) and the similarities and differences to JID. The material on SSA is taken from Wikipedia [46] and a lecture note in Compiler Construction at the University of Colorado [36].

SSA/DCR is also used to detect dead code propagating over conditional branches (Jccs) by inserting *dominance frontiers*. The best algorithm for computing and inserting the dominance frontiers runs in $O(n^2)$, so linearity is not possible with SSA/DCR when considering dead code propagating over Jccs. Because detection of dead code propagating over Jccs is a feature not present in JID (section 5.5.3 on page 98 discusses why not), the theory for detecting dead code propagating over Jccs with SSA/DCR is not discussed.

### 5.7.1 SSA/DCR

In SSA, each variable is subscripted. When a statement defining a variable is encountered, the variable subscript is increased. An example is given in table 5.10 on the next page. In the example, each *assignment/definition* of a variable increases its subscript. Note that *usage* of a variable does not increase its subscript. Each subscripted variable is treated as a separate variable in SSA.

| Stm. # | Regular code | After SSA |
|---|---|---|
| s(1) | $y = 4$ | $y_1 = 4$ |
| s(2) | $x = 2$ | $x_1 = 2$ |
| s(3) | $x = y * 2$ | $x_2 = y_1 * 2$ |
| s(4) | $x = x + 2$ | $x_3 = x_2 + 2$ |
| s(5) | $x = y + 1$ | $x_4 = y_1 + 1$ |
| s(6) | $y = y + x$ | $y_2 = y_1 + x_4$ |

Table 5.10: Each assignment of variable x and y increases the variable's subscript. The leftmost column contains the statement numbers.

### UD-chains and DU-chains

To detect dead code, it is necessary to explain the terms *use-definition chains* (UD-chains) and *definition-use chains* (DU-chains).

For each statement in SSA, there is a UD-chain. A UD-chain contains the statements previously *defining* the variables used in the statement owning the UD-chain. Because in SSA, all definitions/assignments of a variable increases its subscript, there is at most n elements in a UD-chain of a statement $s(x)$, where n is the number of different variables used in $s(x)$.

Also, for each statement in SSA, there is a DU-chain. The DU-chain consists of a all the future statements *using* the variables defined in the statement. When all variables defined in a statement has no uses, the DU-chain of the statement is also empty.

The UD- and DU-chains for the example in table 5.10 are listed in table 5.11.

| Stm. # | UD-Chain | DU-chain |
|---|---|---|
| s(1) | $\emptyset$ | $s(3), s(5), s(6)$ |
| s(2) | $\emptyset$ | $\emptyset$ |
| s(3) | $s(1)$ | $s(4)$ |
| s(4) | $s(3)$ | $\emptyset$ |
| s(5) | $s(1)$ | $s(6)$ |
| s(6) | $s(1), s(5)$ | $\emptyset$ |

Table 5.11: UD- and DU-chains for example 5.10

## 5.7.2 Algorithm for Dead Code Elimination

The following illustrates an algorithm for eliminating dead code with the use of SSA, DU- and UD-chains. The algorithm is taken from [36]. As input, the algorithm requires a code-snippet (cs) containing the statements to evaluate, as well as the DU- and UD-chains for each statement.

**Algorithm 5.7.1:** DEAD CODE ELIMINATION USING SSA($cs$)

```
while ∃ a variable v in cs with no uses whose def has no side effects
  do { Delete the statement s that defines v
       for each of s's UD-chains
         do Delete corresponding DU-chain that points to s
```

The algorithm runs as long there are variables in cs with no uses. Applying the algorithm on the previous example, the following iterations occurs: In iteration 1, the statement $s(2)$ has no uses by its DU chain, and can be removed. Because $s(2)_{UD}$ is empty, the algorithm continues. In iteration 2, the statement $s(4)$ has no uses by its DU chain, and can be removed. $s(4)_{UD}$ contains $s(3)$ and $s(4)$ is removed from $s(3)_{DU}$. Now, $s(3)_{DU} = \{\emptyset\}$. In iteration 3, the statement $s(3)$ has no uses by its DU chain (because of the previous iteration), and can be removed. $s(3)$ is removed from $s(1)_{DU}$. In iteration 4, the only statement left with no uses is $s(6)$ but because it is still alive at the end of the cs, it should not be removed (corresponding to the "side effect" property in the algorithm).

In total, statements $s(2)$, $s(3)$, and $s(4)$ was removed by the algorithm.

### 5.7.3 Use of SSA/DCR as a method for JID

The major difference between SSA/DCR and JID, is that JID was designed for detecting junk in malicious code, while SSA/DCR was designed for optimisation in high level languages. That set aside, they are very similar. The variable subscripting found in SSA is somewhat equivalent to the multiple versions of entities found in entity lists in JID. The DU/UD-chains in SSA/DCR corresponds to flows between entities in JID. The algorithm for detecting dead code enumerates the DU/UD-chains, JID enumerates the flows searching for entities satisfying the same conditions as the DU/UD-chains in dead code. The differences are minimal.

Admittedly, SSA/DCR would work perfectly as a method for detecting junk instructions. The output of the previous algorithm is equal to the output of the JID framework, with the exception that the SSA/DCR algorithm removes the statements whereas JID would classify them as J. The rules and definitions in "Definitions, Rules and properties of Framework" (section 5.3 on page 81) performs similar to SSA/DCR and they perform both in $O(n)$. Because SSA/DCR is designed to optimise high level languages (HLLs), the discussion in "X86 Assembler Language and JID" (section 5.4 on page 90) is still valid, because certain properties found in the x86 assembler language is different from HLLs. The final conclusion is that something similar to JID has been developed before, although it has not been used in the context of malicious code (at least to my knowledge).

# Chapter 6

# Testing

This chapter is devoted to testing the Zmist detection developed in section 3.4.3 on page 66, and the JID framework developed in chapter 5. Before conducting the tests, necessary discussions and motivations are given in the following section.

## 6.1 Test Foundation

### 6.1.1 The Reason for Testing JID

The reason for testing JID is to achieve a confidence that JID manages to separate J instructions from NJ instructions. There is an a'priori known difference between clean files and Zmist infected files considering J instructions: In clean files there are no Zmist infections, while in infected files, Zmist is present with an arbitrary amount of J instructions. If JID manages to detect this difference, one can assume that JID manages to separate J instructions from NJ instructions.

### 6.1.2 How to measure J instructions

To be able to determine a method for measuring the amount of J instructions reported, expected properties concerning J will be discussed in clean and infected files.

#### Clean Files

Clean files are assumed to contain a minimal amount of junk instructions. This is because after compiler optimisations (such as SSA), the junk/dead code is removed from the code. However, programs not compiled with any optimisation can contain junk instructions. SSA and other methods for detecting dead code are applied on the source code and not the assembler code. Because of this, there can still be junk instructions in the assembler code corresponding to the source code, even if there is no dead code present in the source code. It is therefore not expected that clean files are junk free, but it is expected that the amount of J instructions is lower than the amount in virus code. From this, it is reasonable to assume that the amount of J instructions scale linearly with the total amount of instructions.

**Infected Files**

It is expected that the amount of J would be higher in Zmist infected files than the amount of J found in clean files. There are two situations possibly corrupting the percentage of J instructions found in Zmist. First, not all Zmist infections contain junk instructions. The type "JMP after each host instruction" described in 3.2.3 on page 54, contains no junk instructions inserted by Zmist. Also, there are infections of the type "polymorphic decryptor" not containing junk instructions. The decryptor D2 illustrated in section 3.3.2 on page 59 is such an example. Second, a problem similar to the problem with instruction statistic in section 3.4.2 on page 63 is present. With clean code, it was assumed that the amount of J instructions scale with the total size of the program. This is not a valid assumption when considering J code intentionally inserted. Because the size of Zmist is relatively constant compared to the size of the infected host, smaller programs will contribute more to the total J percentage than larger programs.

A better indicator than the average amount of J code present in a Zmist infected file, is the average *spread* or *distance* of the J code. Because the recursive traversal returns the instructions sequence as executed on a CPU, the Zmist decryptor will be traversed independent of location in the code. This yields a possibility to measure the distance between each J instruction. Because Zmist potentially contains many J instructions, it is expected that the average distance is small compared to clean files. It is therefore assumed that in a significance test, the distance measure would yield a stronger significant difference from clean code, than a measure using percentage of J code. When measuring the average distance, median should be used instead of calculating a mean distance. This is because the high probability of outliers corrupting an average. For instance, assume between position 8 and 200, there are 48 J instructions with the average distance of 4. If the next J instruction starting from position 200 is found at position 3000, the average distance would be: $(4 * 48 + 3000 - 200)/49 \approx 61$. In this example, the median would be 4.

There are however some problems with the use of median as well. Assume the average Zmist decryptor contains 200 J instructions. Further assume the total amount of J instructions in a program is 3000. This implies that 2800 of the J instructions are J instructions naturally found in clean code, and the median distance would then be amongst the distances found in the J instructions of the clean code. How this turns out in the statistics, depends on the amount of J found in clean code, similar to percentage. However, median distance scales better than using percentage of J code and is chosen in favour of percentage.

## 6.2 Testing JID

From the discussion, a clean file is assumed to contain a relative fixed amount of J code relative to its size. The same assumption cannot be applied to infected files. Therefore, the *median distance* of J instructions are used as a measurement in the clean and infected files when testing.

Before describing the results, the setup and tools used to gain the statistical information are described.

### 6.2.1 Setup

#### Clean files

The testing involving clean files were performed at Norman ASA. They have a large testbed containing several terabytes of clean files. In the testing, approximately 330GB of clean files were processed in Gdasm. The clean sample set constitute of 39638 PE files, yielding the average file size of 8,5MB. In total, there were 263556432 disassembled instructions distributed over the 39638 files. Of the nearly 264 million instructions, 156464 instructions were classified J. This yields a percentage of $156464/263556432 \approx 0,000594 \approx 0.06\%$ J.

The PE files found in the sample set originate from all types of software and different operating systems, and are collected from different sources by Norman ASA. The sample set is therefore assumed to provide a random distribution.

#### Infected files

In the Zmist sample set, there are 74 Zmist.A files, 4 Zmist.B files, 3 Zmist.C files and 144 Zmist.D files, in total 225 Zmist infected files. There were 4471267 disassembled instructions distributed over the 225 files. Of the nearly 4,5 million instructions, 132085 instructions were classified J. This yields a percentage of $132085/4471267 \approx 0,02954 \approx 3\%$ J.

Zmist.D contains all properties found in B and C. It is therefore not necessary to include several B and C samples in the sampleset. The B and C infections does not contain any property not detectable in D.

### 6.2.2 Gdasm and R

Both the Zmist Decryptor Detection (ZD) and the JID framework were developed as plugins to Gdasm. Gdasm was described in section 3.1.3 on page 49. The data output from Gdasm in the different experiments were used as input to R to manipulate the data there. R is a statistical tool described at the homepage as follows[7]: *R is a language and environment for statistical computing and graphics.*

For each file disassembled, Gdasm outputs the following 5 comma-separated values (CSV).

```
No. J instructions; No. NJ instructions;
No. instructions; Median J distance; No. Zmist decryptors found.
```

For each file in each sample set disassembled, the CSV values were written to a datafile. The two datafiles were then imported in R and manipulated there.

The CSV files are found on the CD described in appendix A on page 125

### 6.2.3 Results

This section provides the distributions found when applying Gdasm on the two sample sets.

#### Clean files

Calculating distance requires at least 2 instructions. Therefore, clean files having less than 2 instructions were discarded. This reduced the sample set from 39638 files to 16412 files, i.e. 23226 files had less than 2 J instructions.

Importing the 16412 median distances into R, the calculated mean of the distances were $\mu_c \approx 1299$. The median distribution is illustrated in figure 6.1 on the next page. The number of classes (bins) are set to 100.

#### Infected files

Similar to the clean files, calculating the distance requires at least 2 instructions. Removing the infected files having less than 2 instructions resulted in a sample size of 194, i.e. 31 files had less than 2 J instructions.

Importing the 194 median distances into R, the calculated mean of the distances were $\mu_i \approx 20$. The median distribution is illustrated in figure 6.2 on page 110. The number of classes (bins) are set to 100.

Having provided the numeric foundation, a statistical method for comparing the two sample sets should be discussed.

### 6.2.4 Choosing a Statistical Method

There are several different statistical hypothesis testing methods to choose from [47]. Summarised, they differ on whether the sample set is assumed to be normal or not, and if the values in the different sample sets are independent or dependent. Prior to choosing a statistical method, these two parameters must be discussed.

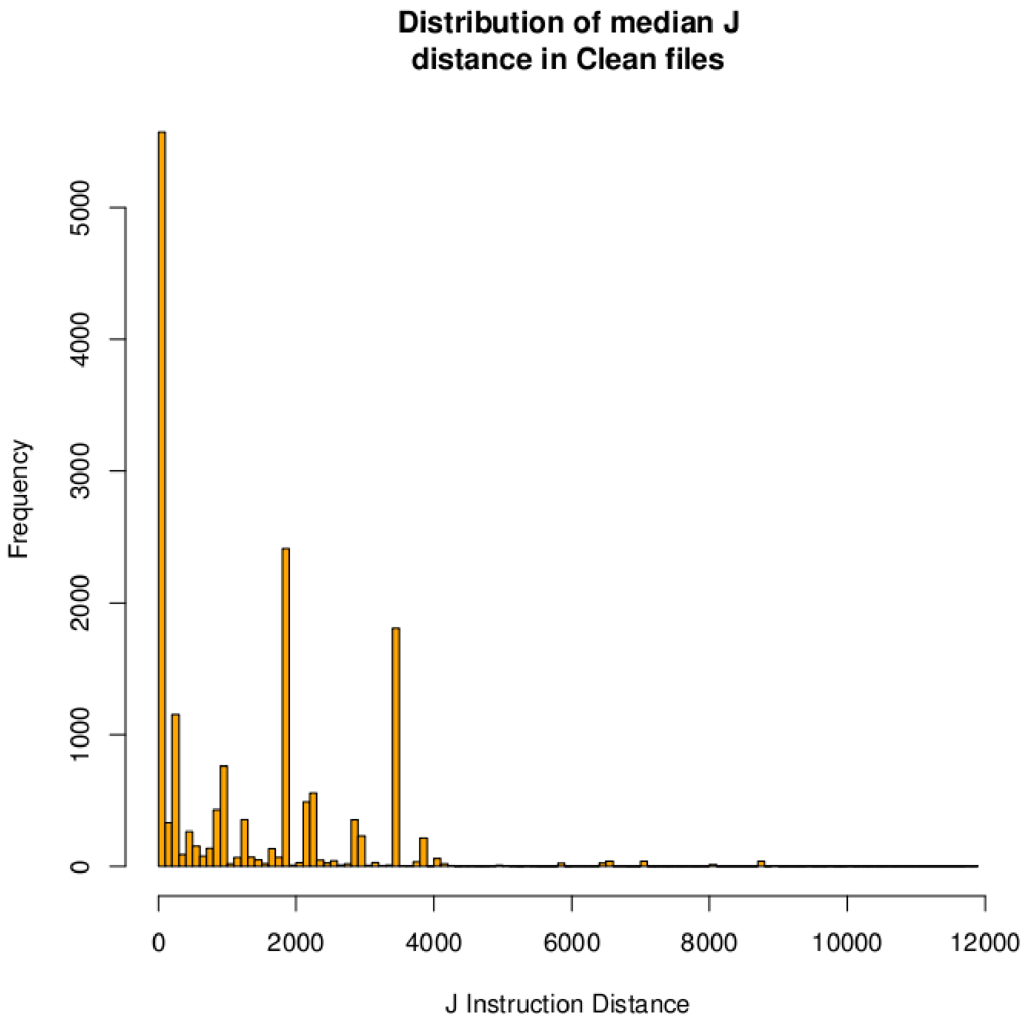


Figure 6.1: J median distribution in Clean files

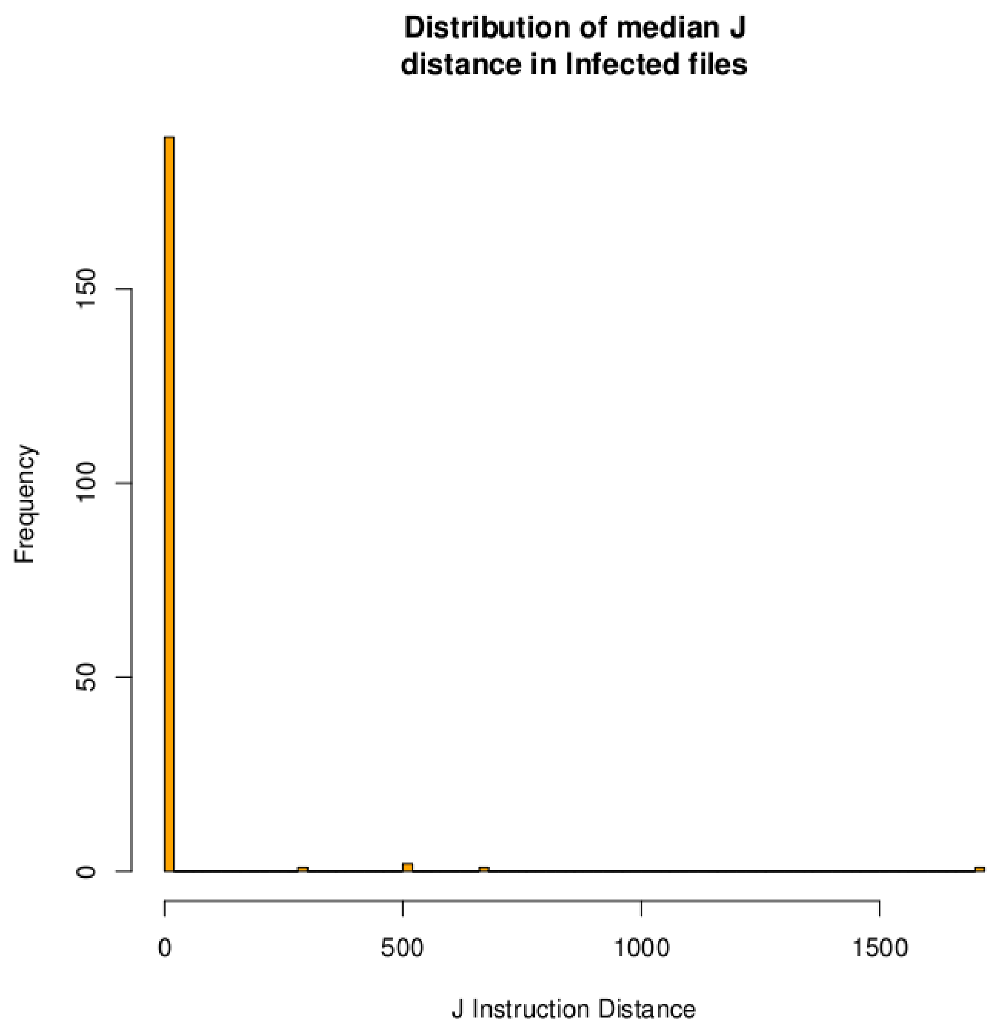


Figure 6.2: J median distribution in Infected files

### Independent Variables

The variables of the clean sample set, is not correlated to the variables of the infected sample set. They are selected from different sources and the clean samples are not altered to obtain the infected sample set or vice-versa in any way. They are therefore independent of each other.

### Sample sets normality

As seen in the distributions of clean files (figure 6.1 on page 109) and infected files (figure 6.2 on the facing page), both distributions are strongly skewed to the right. The clean and the infected sample sets are not normally distributed. There is no need to verify this using a normality test, because all the possible hypothesis testing methods possible to use with non-normal distributions can also be applied to normal distributions.

### Welch t-test

From the previous arguments, statistical hypothesis test capable of handling non-normal, independent sample sets should be found. From [29, 47], the suggested test in that case is the *Welch t-test*. There are two requirements when applying a Welch t-test on two sample sets. The first is that the sample sets are normally distributed or the number of samples summed together are larger than 40. The second is that the observations (samples) are independent from each other. The two test sets comply to both these requirements.

The Welch t-test gives a probability $p$, assuming the null hypothesis is true, that the that values of the test statistic will be at least as extreme as what was observed. Setting the null hypothesis to $\mu_i = \mu_c$ is reasonable, because this would imply that JID doesn't manage to divide the infected files from the clean files when considering J instruction distances. Because there are differences in the true population, a rejection of the null hypothesis would provide confidence that JID actually manages to separate J from NJ in the sample sets. More direct, the $p$ value can be interpreted as the probability for observing the value 1299 ($\mu_c$) and the value 20 ($\mu_i$) or even more extreme values, given that $\mu_i = \mu_c$.

## 6.2.5 Applying the Welch t-test

R supports the Welch t-test, thus making it easy to apply the test on the sample sets. The following is R output of the test applied on the clean and infected sample sets.

```
        Welch Two Sample t-test

data:  clean and inf
t = 83.6742, df = 974.788, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  1248.788     1308.771
sample estimates:
  mean of x    mean of y
  1298.68157   19.90206
```

The *p* value of the test shows that the null hypothesis can be rejected at significance level of $2.2e-16 = 2.2*10^{-16}$, and it is concluded that JID manages to classify J and NJ instructions correctly.

However, this test does not show if JID *never* violates the primary requirement (false positives) or secondary requirement (false negatives). The requirements were described in section 5.2.4 on page 80. The following two sections contains manual analysis of disassembler output, to investigate if there are violations of the requirements.

## 6.3 Testing for NJ as J, and J as NJ

As described, the significance testing between clean and infected files showed that JID manages to classify J and NJ instructions correctly. The testing does not show that the primary requirement is *never* violated. Because this was a strong requirement when designing JID, it is inspected manually in this section. The secondary requirement is not inspected in the files disassembled, because of the assumption that clean code contains a relatively small amount of J instructions.

Inspecting the 39638 clean files manually is not achievable. Instead, the programs containing significantly more J than others will be inspected manually. The number of programs are determined when examining the J percentage distribution.

### 6.3.1 Junk Percentage Distribution

Of the 39638 programs disassembled, 32637 had 0% J instructions. The remaining 7001 programs had the junk percentage distribution illustrated in figure 6.3 on the facing page. Rounding were applied up or down to the nearest thousandth. This implies that a program containing 0.05% junk is rounded up to 0.1% and a program containing 0.049% junk is rounded down to 0%. Each class (bin) in the figure has a width of 0.1%, resulting in 200 classes in the histogram. From the figure, 4355 programs contain 0.1% J, and 2646 programs contains a J percentage $> 0.1\%$. There were 53 programs containing more than 5% junk. 15 of these 53 programs contained 20% junk.

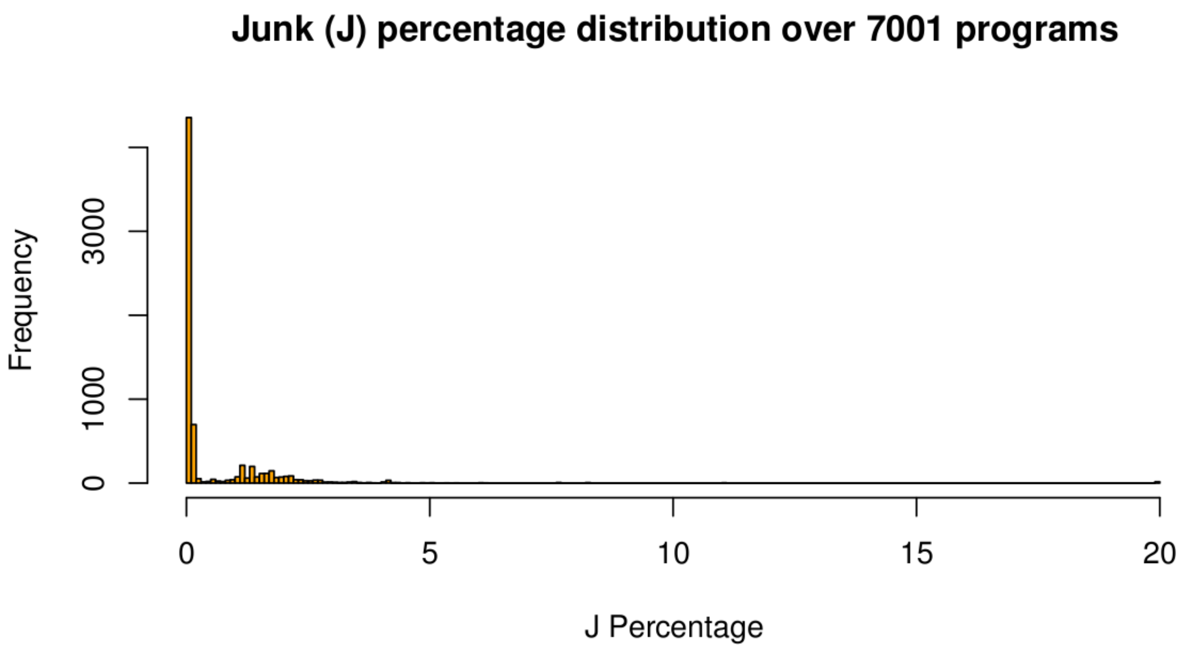


Figure 6.3: Figure illustrating junk percentage distribution

| # | Sequence 1 | Sequence 2 |
|---|---|---|
| 1 | `mov EAX, [EBP-0x320]` | `test [ECX], EAX` |
| 2 | `mov [imm],0x10001` | `cmp EAX, 0x1000` |
| 3 | `mov EAX, [imm]` | |

Table 6.1: Common sequences containing J

The J distribution shows that the amount of programs containing junk are declining rapidly when the J percentage increases. This suggest that the distribution is relatively constant and the value of the constant is approximately equal to the percentage found. The large amount of programs containing 0% J and the rapid declination in the distribution suggest that there are a few programs containing much J code. These programs contain perhaps hand-written code or are compiled without optimisation. The 53 programs containing more than 5% J were manually disassembled using Gdasm to examine if there are any violations of the primary requirement. The results of this manual disassembling is described in the following section.

### 6.3.2 Manual Disassembling

All the 53 programs disassembled contained both correct J classifications and false positives, i.e. violations of the primary requirement. The files containing J classifications which were correct, often had the same instruction sequences containing J. The sequences listed in table 6.1 are two examples of sequences often encountered.

For instance, 24 of the 53 clean files contained at least one occurrence of sequence 1. Because the sequences were present in many different files, this suggest that there is a common property, for instance the files were compiled with the same compiler or a statically linked library containing the sequence have been used in many of the files. There were also many instructions of the type `mov reg, reg`, especially `mov edi, edi`. There were a total of 8448 J instructions of the type `mov edi, edi` in the 53 clean files. This suggest that there could be some problems with the compiler used.

**Violations of Primary Requirement due to libdisasm**

Every false positive reported were due to the disassembler library used in Gdasm (libdisasm [4]). When the library disassembles an instruction, JID depends on the disassembler to produce correct information, especially considering the accesses of the operands. The following table illustrates a misclassification due to libdisasm.

| # | Sequence 1 | JID Classification | Correct Classification |
|---|---|---|---|
| 1 | `bt [esp],0x0` | J | NJ |
| 2 | `cmc` | NJ | NJ |
| 3 | `ja imm` | NJ | NJ |

Table 6.2: libdisasm error

JID classified instruction 1 as J. From the Intel manual, the instruction `bt` does the following:

> *Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag.*

When libdisasm disassembled the instruction `cmc`, libdisasm returned that the instruction *set* the carry flag (CF). From Intel's manual, the instruction `cmc` *complements* CF. This implies that CF must be *tested* before set, i.e. read before written. Because a flag write implies overwrite in JID, instruction 1 was classified J. After patching and recompiling libdisasm, the classification of instruction 1 was correct (NJ). There were numerous other examples which revealed the lack of correct operand accesses in libdisasm.The discovery of libdisasm errors suggest that the J percentage would be lower if the 39638 programs had been disassembled with Gdasm utilising the patched version of libdisasm.

Having gained empirical confidence in that no violations of the primary requirement occurs in clean files, violations of the primary requirement in infected files are investigated in the next section.

### 6.3.3 Infected Files

It is not necessary to analyse the J percentage found in the Zmist files to determine which files to analyse manually, because almost every infected file contains J instructions produced by Zmist. To limit the scope of the testing, the number of Zmist files to be manually disassembled and inspected were set to 30. The 30 Zmist files were divided as follows: 12 files of the type Zmist.A, 3 of the type Zmist.B, 3 of the type Zmist.C and 12 of the type Zmist.D. For each infection, JID was applied and the output was manually analysed.

### 6.3.4 Evaluation

In most of the analysed files, there were J instructions classified as U because of unresolvable control flows. This was expected, because of the U rules defined in chapter 5. There were no violations of the primary requirement or the secondary requirement in the files analysed.

The decryptors D1, D2 and D3 were illustrated in section 3.3.2 on page 59 to illustrate the Zmist decryptor and the level of polymorphism employed. In table 6.3 on the next page, D1, D2 and D3 are listed after JID has been applied and the J instructions removed. This is provided to illustrate that the level of polymorphism in each decryptor has decreased when the J instructions are removed.

## 6.4 Testing of Zmist Decryptor Detection on Zmist Samples

This section describes the testing of the Zmist Decryptor Detection (ZD) plugin to Gdasm (Gdasm/ZD). The ZD plugin is based on the algorithm described in section 3.4.3 on page 66. The amount of true positives vs false negatives, should be measured to investigate the detection rate of Gdasm/ZD.

### 6.4.1 Setup

Of the 225 Zmist files, there were 178 decryptor samples manually confirmed using IDA Pro and Norman Sandbox Analyzer Pro. These two tools were described in section 3.1.3 on page 49. The decryptor samples had the following distribution:

- 52 were infected with Zmist.A.
- 3 were infected with Zmist.B.
- 3 were infected with Zmist.C.
- 120 were infected with Zmist.D.

| # | Decryptor 1 (D1) | Decryptor 2 (D2) | Decryptor 3 (D3) |
|---|---|---|---|
| 1 | call [0x4082FE] | mov [0x421634], edi | call 0x00003090 |
| 2 | mov [0x41B000], ECX | mov [0x41FBFC], EBX | push EAX |
| 3 | mov [0x41BE98], EBX | mov edi, 0x00421D90 | pop [0x415E64] |
| 4 | mov EBX, 0x00426758 | push EAX | push esi |
| 5 | mov [EBX], esi | pop [edi] | jmp 0x000030AF |
| 6 | mov esi, 0x00000000 | mov [0x41FBF0], 0x00000000 | mov EAX, 0x004167FC |
| 7 | mov [0x424C70], esi | mov edi, 0x004143E2 | push ECX |
| 8 | mov esi, 0x0040A7E1 | push edi | pop [EAX] |
| 9 | mov ECX, [esi] | pop EAX | jmp 0x000030E0 |
| 10 | mov EBX, ECX | mov EBX, [EAX] | mov esi, 0xFCA091F5 |
| 11 | mov esi, 0x00423E40 | mov [0x4211B8], EBX | sub esi, 0x2154A1C0 |
| 12 | mov ECX, esi | mov edi, [0x4035E5] | sub esi, 0xACEEC496 |
| 13 | push EBX | mov EBX, edi | xor esi, 0x2E5D2B9F |
| 14 | pop [ECX] | mov [0x41FBF8], EBX | mov [0x417CF0], esi |
| 15 | mov esi, 0x00416723 | mov edi, 0x0041FBF0 | push [0x402A32] |
| 16 | mov EBX, [esi] | push [edi] | pop esi |
| 17 | push EBX | pop EAX | mov ECX, esi |
| 18 | pop ECX | mov EBX, EAX | push ECX |
| 19 | push ECX | push EBX | pop [0x415E6C] |
| 20 | pop [0x424C6C] | pop EAX | jmp 0x00003174 |
| 21 | mov esi, 0x00424C70 | mov edi, 0x00421630 | push [0x40A9AB] |
| 22 | mov EBX, [esi] | push EAX | jmp 0x00003180 |
| 23 | mov ECX, EBX | pop [edi] | pop EAX |
| 24 | mov esi, 0x0041B004 | mov EAX, 0x00421630 | push EAX |
| 25 | push ECX | mov EBX, [EAX] | pop [0x415E60] |
| 26 | pop [esi] | add EBX, [0x4062A8] | mov EAX, 0x00417CF0 |
| 27 | push [0x41B004] | mov [0x421630], EBX | mov ECX, [EAX] |
| 28 | pop EBX | mov edi, [0x421630] | push ECX |
| 29 | mov esi, 0x00416559 | mov EBX, [edi] | pop esi |
| 30 | push [esi] | mov EAX, EBX | push esi |
| 31 | pop ECX | mov [0x41FBF4], EAX | pop EAX |
| 32 | mov esi, ECX | push [0x41FBF4] | mov esi, 0x00417CF4 |
| 33 | add EBX, esi | pop edi | jmp 0x000031F1 |
| 34 | mov esi, 0x0041B004 | mov EBX, 0x004211B8 | push EAX |
| 35 | mov [esi], EBX | push [EBX] | jmp 0x000031F5 |
| 36 | jmp 0x00015968 | pop EAX | pop [esi] |
| 37 | push [0x41B004] | xor edi, EAX | mov EAX, 0x004040E6 |
| 38 | pop EBX | mov EBX, 0x0041FBF4 | push [EAX] |
| 39 | mov esi, EBX | mov EAX, EBX | pop esi |
| 40 | push [esi] | mov [EAX], edi | mov ECX, 0x00417CF4 |
| 41 | pop ECX | push [0x41FBF8] | add [ECX], esi |
| 42 | push ECX | pop EBX | jmp 0x0000203E |
| 43 | pop [0x424DCC] | mov edi, EBX | mov esi, 0x00417CF4 |
| 44 | mov EBX, 0x00424DCC | mov EAX, 0x004211B8 | mov ECX, [esi] |
| 45 | mov esi, [EBX] | mov EBX, EAX | mov EAX, [ECX] |

Table 6.3: Code example of 3 Zmist decryptors after applying JID

### 6.4.2 Results

Of the 178 decryptors, Gdasm/ZD managed to detect a decryptor in 167 of the files. This yields a detection percentage of $167/178 \approx 0.94 \approx 94\%$.

Of 11 files not detected, 2 were Zmist.A and 9 were Zmist.D.

### 6.4.3 Evaluation

A 94% detection rate is not good enough, but it indicates that Gdasm/ZD is a right step on the way to 100% detection. A manual inspection of the 11 files revealed that the recursive traversal was not able to reach the Zmist decryptors. Unresolvable addresses were the most common reason for RT not reaching the decryptor. In one file, Zmist had inserted itself in a function not used by the program.

To investigate if Gdasm/ZD would detect the decryptors if RT had found the decryptors, the start addresses of the decryptors were found manually by inspecting the decryptors in IDA Pro. For each of the 11 files, the start address were given as input to the RT algorithm used in Gdasm/ZD. When explicitly using the start addresses as input, Gdasm/ZD managed to detect the Zmist decryptors in all the 11 files.

The problem with false negatives is therefore attributed to the RT algorithm. To improve the detection rate, the RT algorithm must be extended with possibilities for resolving more unresolvable control flows, or another disassembler algorithm must be found. An option could be using LS and RT in a *hybrid* combination. A hybrid disassembler algorithm using LS and RT is sketched by Schwarz et al in “Disassembly of Executable Code Revisited” [33].

# Chapter 7

# Conclusion and Future Work

## 7.1 Summary

The topic of this thesis was to explore if junk instructions can be detected in executable files. There were two research goals defined in section 1.2 on page 16.

1. Learn about polymorphic viruses through a case study and detect the virus used as a case study.
2. Examine if there is possible to infer any knowledge from the case study which could be used in general detection of polymorphic viruses

The research conducted to achieve the two goals, led to the thesis' problem statement:

> Is it possible to separate junk instructions from non-junk instructions in an executable file?

To be able to achieve the two goals stated and to answer the problem statement, several topics had to be studied. Background knowledge related to the study of polymorphic computer viruses were acquired. Conforming to the first research goal, the virus Zmist was studied and a detection algorithm was devised. From the experiences made in the case study came the idea of a comparison program, in which the first step would be to detect junk instructions. Detection and removal of junk instructions would aid the virus analyst in speeding up the analysis process of a virus. As a foundation for answering the problem statement, the JID framework was designed and developed. To achieve confidence in the detection algorithm and the JID framework, several tests were conducted.

## 7.2 Results

The results presented in this section are related to the two research goals and the problem statement given in the introduction chapter.

### 7.2.1 Research Goal 1

This goal can be viewed as a knowledge acquirement goal. When evaluating the research conducted in the master thesis, I find that this goal has been reached. The terminology used in the virus/antivirus business has been learnt, mainly through reading literature. Extensive knowledge of the PE file format and the x86 assembler language were gained through studying Zmist and writing Gdasm. Equivalently important is the acquirement of a *methodology foundation*, which can be used in similar work in the future. The information channels discovered, the knowledge and usage of tools, and a more intuitive understanding of the art of reverse engineering, are examples found in the methodology foundation.

The knowledge acquired, was used to design and develop a relatively successful Zmist Decryptor detection algorithm. The algorithm managed to detect 100% of the 178 samples tested, although shortcomings in the chosen disassembling algorithm reduced this amount to 94%. Detection of Zmist is still a problem for many antivirus companies today and I consider the detection algorithm created a successful achievement. The conclusion is therefore that the first research goal has been reached.

### 7.2.2 Research Goal 2

This goal was defined to initiate a process which could lead to a problem statement. The knowledge acquired by achieving the first research goal was used in the discussion in chapter 4. Chapter 4 can be viewed as the response or consequence to the second research goal posed. When evaluating, I find that the second goal has been achieved, because a reasonable problem statement has been given. The problem statement had its origin in an idea of an automatised or semi-automatised comparison program. Others are also examining this idea. For instance, at the Virus Bulletin conference in October 2008, Ismael Vilar of the antivirus company Panda is going to talk about "Graph, entropy and grid computing: automatic comparison of malware". This shows that comparison is an ongoing topic.

The problem statement itself is detection of junk instructions. It is important to detect junk instructions prior to an automatic comparison, because removing junk instructions provides a more homogeneous comparison foundation. The conclusion is therefore that the second research goal has been reached, because a sound problem statement has been posed.

### 7.2.3 Problem Statement

Evaluating the question asked in the problem statement, the direct answer is that it is possible to separate junk instructions from non-junk instructions in an executable file. The theoretical foundation laid in chapter 5 resulted in an implementation managing to separate J- from NJ-instructions.

The tests of the JID framework showed that by using the median distance between J instructions, JID is capable of dividing clean files from infected files. The significance test showed that the null hypothesis *JID cannot separate infected files from clean*, can be rejected at a significance level of $2.2e - 16 = 2.2 * 10^{-16}$. This is a very strong result and supports a positive answer to the problem statement.

Because the result doesn't tell whether violations of the primary requirements and secondary requirements defined in chapter 5 *never* occur, further tests were conducted. Although purely empirical, these tests showed that JID itself doesn't violate the primary requirement. Based on the results of the statistics previously described and the empirical results, it is reasonable to conclude that JID works as a tool for separating junk instructions from non-junk instructions.

### 7.2.4 Similar Work

The approach of finding methods relating to junk detection could have been performed more thoroughly. If the SSA/DCR method described in chapter 5 had been discovered earlier in the thesis progress, the theory developed would have been unnecessary to create. However, because SSA/DCR focuses on dead code removal in high level languages, the topics concerning the x86 assembler language vs junk/dead code removal would still have had to be discussed.

I find that, although the theory I developed was new to me at the time it was created, it is not as innovative as initially thought and SSA/DCR would also work well as a method for junk instruction detection. However, the final conclusion is that the JID framework developed provides a satisfiable answer to the question posed in the problem statement.

## 7.3 Other Results

This section describes a contribution made, not directly related to the topics of the thesis.

### 7.3.1 Libdisasm Patches

When testing the JID framework, several misclassifications occurred. The errors were related to the disassembler library used, namely *libdisasm*. There were several cases of missing operand accesses defined for certain opcodes in the opcode tables (`ia32_opcode_tables.c` and `ia32_implicit.c`). These were patched according to the definitions of the different instructions found in the Intel manuals, and a patch were submitted to the developers of libdisasm. The patch is currently under review and it is expected that the patches are included in the next release of libdisasm.

## 7.4 Future Work

The last topic in this chapter is suggestions for future work.

### 7.4.1 Detection of Zmist

There are several topics left unsolved before 100% detection of Zmist can be achieved.

#### Disassembler Algorithm

When testing the Zmist Decryptor detection, the choice of disassembler algorithm proved unsatisfiable in 6% of the tested samples. As suggested in section 6.4.3 on page 117, a hybrid algorithm could perhaps resolve the addresses not found by the recursive traversal algorithm.

#### Other Infection Types

The Zmist Decryptor detection developed only detects the polymorphic decryptor infection type (see section 3.3 on page 56). As discussed, this infection type is present in about 80% of the Zmist infections. The "unencrypted metamorphic engine" and "JMP after each host instruction" types found in 20% of the Zmist infections, are left undetected. From the discussion in section 3.2.4 on page 55, a proposal was to detect the metamorphic engine using byte-signatures. The instruction statistics could aid in the detection of the JMP infection type. Detection of these two infection types were outside of the scope of the thesis and are regarded as future work.

#### Geometric Properties

Before integrating the detection algorithm in a production system, the geometric properties of Zmist should be found. As discussed in 3.4.1 on page 62, the geometric properties act as filter, separating the files given as input to a scanner in two groups: suspicious and not-suspicious. The files in the suspicious category should be scanned using a proper detection algorithm, and the not-suspicious files should not. This would increase the performance of the scanner, because the geometric properties of a file are very fast to compute compared to applying a detection algorithm on the file.

Szor lists some of the geometric properties in Zmist, but as discussed in section 3.4.1 on page 62, they are removed in later versions of Zmist. A search for the geometric alterations done by Zmist.D, should therefore be conducted before implementing a Zmist detection algorithm in a commerical virus scanner.

### 7.4.2 Applying JID on Other Viruses

This thesis focused on Zmist primarily, although the methods developed in JID detects J instructions in general. Concerning other viruses, there are two suggestions for future work.

1. Does JID manage to detect J in other viruses?
2. To what extent are J instructions used as a polymorphic technique in other viruses?

Preliminary tests using JID on the virus *Bagle*, done as a last minute investigation, shows that some versions of the Bagle are using J instructions to achieve polymorphism.

### 7.4.3 Comparison Program

JID is one of many building blocks used to construct an automated comparison program. There are however many blocks left before an automated comparison program is developed, and the following topics are suggested as future work.

#### Related Work

Related work is important to investigate prior to developing a comparison program. In my opinion, the following list would be appropriate starting points.

- As previously mentioned, there is a presentation to be held at the Virus Bulletin conference in October 2008 termed “Graph, entropy and grid computing: automatic comparison of malware”.
- Automatic comparison is applied in many fields. Examples are: automatic classification of books and images and sorting spam emails from non-spam emails. It should be investigated if these methods are applicable on the field of automatic comparison of computer viruses.

# Appendix A

# Source Code, Zmist Listings and Files

Instead of reproducing the source code created and Zmist disassemblies in the appendix, this is collected on the appended CD. The file `README` on the root of the CD explains the directory structure. The following are included on the CD.

- The source code of Gdasm.
- The source code of the patched version of libdisasm.
- The full disassembler listing of REGEDIT-4.VXE, SNDREC-5.VXE, HWINFO-5.VXE, PMTS-5.VXE, WINHLP32-4.VXE and INTERNAT-4.VXE.
- The source code of the Zmist Decryptor detection.
- The source code of JID.
- The CSV files containing the numeric material which was the foundation of the testing done in chapter 6.
- A PDF version of the thesis.

# Bibliography

[1] Acm digital library search, visited 06.10.07. http://portal.acm.org/portal.cfm.

[2] Ida pro homepage, visited 03.05.08. http://www.hex-rays.com/idapro/.

[3] Ieee explore article search, visited 05.10.07. http://ieeexplore.ieee.org/Xplore/dynhome.jsp.

[4] libdisasm homepage, visited 03.05.08. http://bastard.sourceforge.net/libdisasm.html.

[5] Malware evolution 2006, visited 14.01.08. http://www.kaspersky.com/malware_evolution_2006_summary.

[6] Norman sandbox analyzer pro homepage, visited 03.05.08. http://www.norman.com/microsites/malwareanalyzer/Products/analyzer-pro.

[7] The r project for statistical computing homepage, visited 29.05.08. http://www.r-project.org/.

[8] Springerlink article search, visited 05.10.07. http://www.springerlink.com/home/main.mpx.

[9] Vmware homepage, visited 03.05.08. http://www.vmware.com.

[10] Vx heavens, visited 25.02.07. http://vx.netlux.org.

[11] The Age. Mydoom damage estimate termed absurd, visited 12.08.07. http://www.theage.com.au/articles/2004/02/06/1075854035648.html.

[12] Articleworld. 1260, visited 14.08.07. http://www.articleworld.org/index.php/1260_(computer_virus).

[13] John Aycock. *Computer viruses and malware*. Springer, 1st edition, 2006.

[14] Commtouch. Malware outbreak trend report: Storm-worm, visited 16.02.2008. http://www.commtouch.com/downloads/storm-worm_motr.pdf.

[15] Commtouch. Malware outbreak trends, visited 16.02.2008. `http://www.commtouch.com/documents/Commtouch_2007_Q1_Malware_Trends.pdf`.

[16] Intel Corp. Intel architecture software developers manual vol 2 (a-m), visited 05.05.08. `http://www.intel.com/design/processor/manuals/253666.pdf`, 2007.

[17] Intel Corp. Intel architecture software developers manual vol 2 (n-z), visited 05.05.08. `http://www.intel.com/design/processor/manuals/253667.pdf`, 2007.

[18] Intel Corp. Intel architecture software developers manual volume 1: Basic architecture, visited 05.05.08. `http://www.intel.com/design/processor/manuals/253665.pdf`, 2008.

[19] Microsoft Corporation. Visual studio, microsoft portable executable and common object file format specification, visited 21.05.07. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[20] Andreas Clementi et al. Independent comparatives of anti-virus software, visited 10.03.08. `http://www.av-comparatives.org`.

[21] F-Secure. F-secure virus descriptions: Bagle, visited 26.04.08. `http://www.f-secure.com/v-descs/bagle.shtml`.

[22] F-Secure. The pc virus celebrates its 20th anniversary, visited 14.02.07. `http://www.f-secure.com/news/items/news_2006011900.shtml`.

[23] Peter Szor; Peter Ferrie. Hunting for metamorphics. Virus Bulletin Conference, 1999.

[24] D. Harley; R. Slade; U. E. Gattiker. *Viruses Revealed*. Osborne/McGraw-Hill, 2001.

[25] Sarah Gordon. *The generic virus writer*. Fourth International Virus Bulletin Conference. Jersey, 1994.

[26] Fortinet Guillaume Lovet. *Dirty money on the wires: the business models of cyber criminals*. Virus Bulletin Conference, 2006.

[27] Matt Webster; Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. In *Journal in Computer Virology*, pages 149–161. Springer Paris, August 2006.

[28] Rachit Mathur. Normalizing metamorphic malware using term rewriting. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 75–84. IEEE Computer Society, September 2006.

[29] David S. Moore; George P. McCabe. *Introduction to the Practice of Statistics*. W.H. Freeman and Company, 4th edition, 2003.

[30] Merriam-Webster. Online dictionary. `http://www.m-w.com`.

[31] Washington Post. A short history on computer viruses attacks, visited 18.02.07. `http://www.washingtonpost.com/wp-dyn/articles/A50636-2002Jun26.html`.

[32] Konstantin Rozinov. Reverse engineering: An in-depth analysis of the bagle virus, visited 20.04.08. `http://rozinov.sfs.poly.edu/papers/bagle_analysis_v.1.0.pdf`.

[33] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, USA, 2002. IEEE Computer Society.

[34] D. Spinellis. Reliable identification of bounded-length viruses is np-complete. volume 49. IEEE Computer Society, 2003.

[35] Wing Wong; Mark Stamp. Hunting for metamorphic engines. In *Journal in Computer Virology*, pages 211–229. Springer Paris, December 2006.

[36] Michelle Mills Strout. Lecture notes on static single assignment form (cs380c), visited 21.05.08. `http://www.cs.colostate.edu/~mstrout/CS553Fall06/slides/lecture18-SSAusage.pdf`.

[37] Symantec. Increase in cybercrime activity, visited 21.04.07. `http://www.symantec.com/about/news/release/article.jsp?prid=20060307_01`.

[38] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 3. feb 2005.

[39] Peter Ferrie; Peter Szor. Zmist opportunities. pages 45–54. Virus Bulletin, 29. Oct - 1. Nov 2001.

[40] Moheeb Abu Rajab; Jay Zarfoss; Fabian Monrose; Andreas Terzis. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[41] Viruslist. Virus.win32.gpcode.ai, visited 15.05.08. `http://www.viruslist.com/en/viruses/encyclopedia?virusid=164339`.

[42] Wikipedia. Brain. `http://en.wikipedia.org/w/index.php?title=%28c%29Brain&oldid=121990905`.

[43] Wikipedia. Cih. http://en.wikipedia.org/w/index.php?title=CIH_%28computer_virus%29&direction=prev&oldid=126585691.

[44] Wikipedia. Elk cloner. http://en.wikipedia.org/w/index.php?title=Elk_Cloner&oldid=116779037.

[45] Wikipedia. Ransomware*_malware*. http://en.wikipedia.org/w/index.php?title=Ransomware_%28malware%29&oldid=173954990.

[46] Wikipedia. Static single assignment form. http://en.wikipedia.org/w/index.php?title=Static_single_assignment_form&oldid=203948839.

[47] Wikipedia. Statistical hypothesis testing. http://en.wikipedia.org/w/index.php?title=Statistical_hypothesis_testing&oldid=215918978.

[48] Wikipedia. Timeline of notable computer viruses and worms. http://en.wikipedia.org/w/index.php?title=Timeline_of_notable_computer_viruses_and_worms&oldid=207972502.

[49] Wing Wong. Analysis and detection of metamorphic computer viruses. Master's thesis, Dept. of CS, SJSU, 2006.

[50] Zombie. *MistFall.Z0MBiE-10.d.* 29A#6 e-zine, 6. March 2002.

[51] Ørjan Nordvik. Hackere/crackere. Master's thesis, The University of Oslo, 2006.