



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY



**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INTELLIGENT SYSTEMS

**NÁZEV PRÁCE**

THESIS TITLE

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**JMÉNO PŘÍJMENÍ**

**VEDOUCÍ PRÁCE**

SUPERVISOR

**Doc. RNDr. JMÉNO PŘÍJMENÍ, Ph.D.**

BRNO 2016

## **Abstrakt**

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém jazyce.

## **Abstract**

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

## **Klíčová slova**

Sem budou zapsána jednotlivá klíčová slova v českém jazyce, oddělená čárkami.

## **Keywords**

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

## **Citace**

Jméno Příjmení: Thesis title, diplomová práce, Brno, FIT VUT v Brně, 2016

# Thesis title

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jméno Příjmení

April 10, 2016

## Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

© Jméno Příjmení, 2016.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Malware Detection</b>	<b>4</b>
2.1	Obfuscation and Malware Taxonomy . . . . .	4
2.1.1	Packing, encryption and oligomorphism . . . . .	5
2.1.2	Polymorphism . . . . .	5
2.1.3	Metamorphism . . . . .	7
2.2	Syntactic Detection . . . . .	8
2.2.1	Algorithmic scanning . . . . .	9
2.2.2	Code emulation . . . . .	9
2.3	Behavioral Detection . . . . .	9
<b>3</b>	<b>Static Analysis</b>	<b>10</b>
3.1	Posets and Lattices . . . . .	10
3.2	Abstract Interpretation . . . . .	10
<b>4</b>	<b>Formal Models</b>	<b>11</b>
4.1	Tree Automata . . . . .	11
4.1.1	Inference . . . . .	11
<b>5</b>	<b>The LLVM Compiler Framework</b>	<b>12</b>
5.1	Structure and Design . . . . .	12
5.2	Intermediate Representation . . . . .	12
<b>6</b>	<b>Detector Design and Implementation</b>	<b>13</b>
6.1	Input generation . . . . .	13
6.2	Taint analysis . . . . .	13
6.3	Classifier Inference . . . . .	13
6.4	Testing and Results . . . . .	13
<b>7</b>	<b>Conclusion</b>	<b>14</b>
7.1	Discussion . . . . .	14
7.2	Future Work . . . . .	14
	<b>Literature</b>	<b>15</b>
	<b>Appendices</b>	<b>16</b>
	List of Appendices . . . . .	17

# Chapter 1

## Introduction

Information technologies and systems have found use in almost all aspects of human life. From personal communication and entertainment to industrial manufacturing, services and commerce. Because of this widespread adoption, safety and security of information systems and the data they hold has become a major concern. Throughout the years, development and administration of information systems has shown to be a complex task and errors are a common occurrence. These errors may pose a safety risk to anyone using the system. Some of these errors can be abused by malicious parties to achieve goals that are not in alignment with the wishes of a legitimate user. The answer to this problem is twofold. By advancing the processes and technologies that we use to develop information systems we lower the number of errors and weaknesses that the system has prior to its use. By carefully monitoring usage of the system after deployment, we lower the risk of abuse of errors that were not corrected during development.

*Malware* is a term used to describe any software tools primarily designed for malicious use against a host information system. Common uses include causing damage to the host system, denying usage of the host to legitimate users and theft of data and computing resources. Depending on its purpose and method of transfer, malware can be classified into several families. For example a *trojan* is malware that is introduced into a host, often by a legitimate user, disguising itself as harmless software. On the other hand a *worm* often abuses an error in the host system to gain access. Both of these are examples of malware that function as standalone pieces of software. *Viruses*, in contrast to trojans and worms, need a host file or software to propagate. The term *payload* is often used to describe the code that will carry out the intended use of the malware. A *keylogger* will log keystrokes to gather sensitive information about the systems users, while a *backdoor* will grant access to the host system to an illegitimate user.

As a means to protect information systems and user data, security systems were developed. Examples include *firewalls*, which come in the form of hardware and software or *intrusion detection* and *intrusion prevention* systems. But probably the most common security system is an *antivirus* software. The former can be described as passive measures, since they mitigate threats that are already being carried out by a human agent or malware. Antiviruses on the other hand were originally designed to protect against malware by actively scanning files present in the host system.

With the introduction of antiviruses, malware authors had to come up with ways to hide the presence of their malware in a host system. One method of achieving this is *obfuscation*. The goal of obfuscation is to make malware hard to distinguish from legitimate software by changing the malware in a way that preserves its original functionality or purpose. This also

makes detection a per system task if the obfuscations are applied in a randomized manner, since the same malware can differ in some way from system to system. Obfuscations can change the malware on several levels. Obfuscations that primarily change how the malware appears as a file, are called *syntactic*, while those which change how the malware functions while preserving it's intended purpose are referred to as *semantic*. On the other end, antivirus software needs to implement methods that can detect malware despite being obfuscated. As with malware, detectors can be classified as syntactic or *signature-based* and semantic or in other words *behavioral*.

The goal of this work is to research applications of formal analysis and verification to behavioral malware detection and present a proof of concept malware detector that applies these methods. The work begins by giving a brief introduction to the challenges of detecting obfuscated malware in chapter 2. Static analyses based on formal methods are introduced in chapter 3. In order to detect and classify malicious behavior descriptions obtained by static analysis, suitable models and methods are proposed in chapter 4. The LLVM compiler development framework is introduced in chapter 5 as a means to implement the mentioned static analyses. The design and implementation of a proof-of-concept detector prototype is described in chapter 6. Finally conclusion with a discussion about choices that were made with possible alternatives and potential for future work is given in chapter 7.

## Chapter 2

# Malware Detection

Early research in the field of computer virology shows that perfectly reliable malware detection is theoretically impossible[?] and more recent research shows that practical malware detection can be made computationally infeasible[?]. Despite these negative results, antivirus software is commercially successful and research in the field of malware detection is meaningful.

Since any method of malware detection is bound to be imperfect, there are several metrics that are used in order to describe the performance of a malware detection method. The number of *false positive* and *false negative* results over a test set of legitimate software and malware respectively gives insight into the detection capabilities, while simple time and memory consumption metrics give insight into the cost and potential scalability of the detection method. Another useful distinction to make is whether the detection method is capable of detecting malware that it has not seen before, but has seen similar malware in terms of syntax or semantics, depending on the type of detector. This capability is referred to as *forward detection*[?] or *generalization*[?].

This chapter aims to introduce topics that impact these metrics. From the point of view of the malware itself, various types of obfuscation will be presented and a taxonomy of malware based on obfuscation techniques will be presented. This part of the chapter is based on [?], [?] and [?]. From the detection point of view, the advantages, drawbacks and overall principles behind syntactic and semantic detection will be introduced, taking from [?] and again [?].

## 2.1 Obfuscation and Malware Taxonomy

A crucial part of any malware today is the ability to hide itself from detection in a host system. The most commonly used technique to achieve this is obfuscation, which in the context of computer programs can be defined as a transformation aimed to hide functionality and hinder analysis. As a transformation it can be applied on any form the program takes, from the original source code, down to the binary file. Figure 2.1 shows the typical compilation chain of a C-like language with all the representations a program takes. In the case of malware the most common obfuscations are applied to the binary file and on the level of assembly code. It is interesting to note that obfuscations also find use in the field of intellectual property protection in which they protect against unwanted reverse-engineering. In this case the obfuscations are applied on the source code or the intermediate representation used by the compiler or interpreter.

# TODO

Figure 2.1: Compilation chain of a C-like language

A prominent feature when using any kind of obfuscation is that, when applied in a randomized manner, one can generate a number of different versions of the final malware. These versions are referred to as *mutations* and the quality of an obfuscation technique is often measured by the number of possible mutations it allows.

## 2.1.1 Packing, encryption and oligomorphism

Simple, yet effective methods of obfuscation go no further than changing the way a malware binary file looks. A common example is the use of compression and simple encryption algorithms.

*Packing* is a technique which uses compression algorithms to scramble the content of the binary file[?]. A decompression procedure is then added to the compressed malware binary so that when the executable is loaded into memory, it will first decompress the malware code and then proceed to execute it. Packing also reduces file size which helps propagation via size-limited channels i.e email attachments.

Another common way of obfuscating binary files is through encryption. Similarly to packing, the main malware body is encrypted using a simple cipher and a decryption procedure is added to the result. The encryption key is often carried with the decryption procedure or can be easily computed from data available to the decryption procedure. Most commonly used ciphers include XOR ciphers, RC4 or TEA.

By using multiple packers or encryption procedures and keys one can obtain a sizeable number of possible mutations. Malware that generates its mutations in this manner is *oligomorphic*. Although the number of possible mutations is high, the decompression and decryption procedures themselves are not mutated. This produces an opportunity for malware analysts to create signatures that target these procedures and detect oligomorphic malware based on them.

## 2.1.2 Polymorphism

The solution to the shortcomings of oligomorphic malware is to mutate their decompression and decryption procedures. *Polymorphic* malware achieves this by applying obfuscations to the code of those procedures. Depending on the obfuscations applied the number of possible mutations rises dramatically. The result is malware whose mutations share almost no resemblance to each other as binary files and thus cannot be generally detected by matching against byte signatures of binary files. A list of commonly used obfuscations follows.

**Dead code insertion** Code with no effect is inserted into the original code. Common examples include inserting NOP instructions between original assembly instructions or performing XOR operations over two identical operands. While very simple to implement, it



is also very simple to reverse the transformation and recover the original code. Combined with other obfuscations however, the inserted code may be further transformed and made very hard to recognize as dead.



Figure 2.2: Example of dead code insertion

**Register reassignment** Operands of assembly code instructions are often stored in registers. The obfuscation changes the registers in which the operands of instructions are stored. The number of possible mutations this obfuscations allows can be large depending on the instructions used and number of registers available, however it can be easily circumvented via using inexact byte signatures. This technique was prominently used in the Win95/Regswap virus.



Figure 2.3: Example of register reassignment

**Subroutine reordering** Or code permutation is a transformation where standalone code segments such as subroutines or basic blocks can be randomly reordered to produce up to  $N!$  mutations, where  $N$  is the number of such segments. W95/Zperm is the common example of a virus that uses this obfuscation.

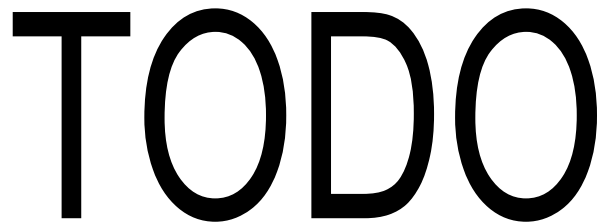


Figure 2.4: Example of subroutine reordering

**Instruction substitution** The effect of a single instruction in code can be often emulated via a sequence of instructions. Typical examples are simple arithmetic and boolean operations which can be emulated in a number of ways each. Other substitutions include moves between registers being replaced by PUSH and POP instructions on the assembly level.

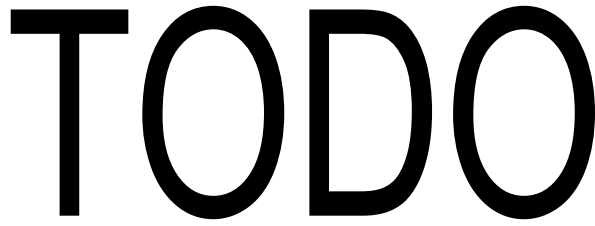


Figure 2.5: Example of instruction substitution

**Code transposition** Similar to subroutine reordering, this obfuscation reorders instructions to change the binary file. If the reordered instructions are dependent, original execution order is restored via unconditional jumps. Depending on the code representation, determining whether two instructions are dependent may require a non-trivial analysis.

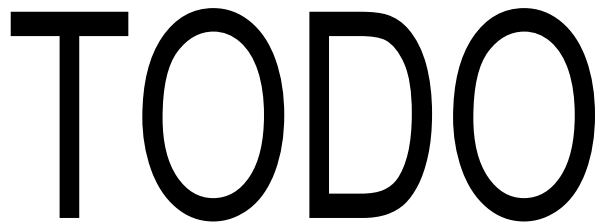


Figure 2.6: Example of code transposition

**Code integration** A technique that uses another executable binary file for obfuscation. The malware executable targets a host executable, disassembles it, inserts its own code into it and reassembles the host, preserving its original functionality. This way the malware can generate as many mutations as there are executable binaries in the host system, while providing other advantages like obfuscating the malware's entry point, since the execution of malware code is interleaved with the execution of the host binary. This obfuscation was introduced in the famous W95/Zmist virus.

### 2.1.3 Metamorphism

The main weakness of polymorphic malware was the fact that even though the unpacking and decryption procedures were mutated, the main malicious code was not. By allowing the malware to run in a controlled environment, the malicious code would eventually appear in memory and could be matched against a byte signature.

*Metamorphic* malware answers this weakness by applying obfuscating code transformations to the whole code of the malware, including the malicious parts. This way the malware changes significantly from one mutation to another and leaves little space for simple byte signature detection. The downside to this malware is the complexity of a mutation engine capable of such code transformations. Figure 2.7 shows the process of generating a new mutation from an old one, once a metamorphic malware is executed. The best example of an implementation is the virus Simile and its MetaPHOR mutation engine.

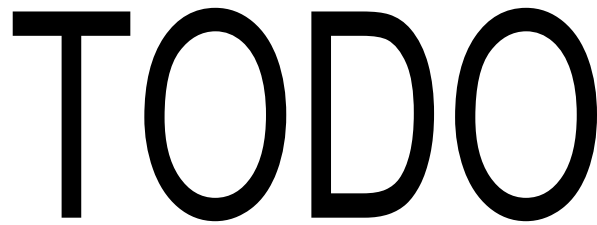


Figure 2.7: Mutation stages of a metamorphic malware

## 2.2 Syntactic Detection

Historically malware detection techniques were centered around syntactic properties of a binary file under inspection. These methods often look for specific byte sequences in files and any file that contains a byte sequence that is deemed malicious is marked as malware. The detection rates of these methods were heavily dependent on the length of the byte sequence which was looked for and the structure of the malware code. The advantages of this approach, to this day, are speed and scalability. The main disadvantage is that they are easy to bypass using any kind of obfuscation. To circumvent this, several modifications were introduced.

**Wildcards** The modification adds symbols with special semantics into the byte signature. Consider figure 2.8. The upper sequence of bytes is a part of the W32/Beast virus and the lower sequence is the wildcard byte signature used to detect it. The ? symbol marks an optional occurrence of a half-byte, while %2 says that the next byte may occur twice in the following two bytes.



Figure 2.8: An example of a wildcard byte signature

**Mismatches** Allow an inexact matching of a byte signature. The following example illustrates an algorithm that allows a mismatch of three bytes. After encountering a mismatch, the algorithm notes the mismatch and the next byte of the signature is used for matching. Figure 2.9 shows a malware byte sequence and three byte signatures that match it.

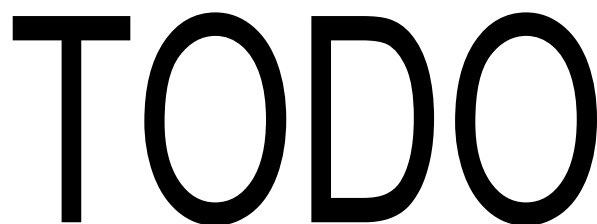


Figure 2.9: An example of mismatch byte signatures

### 2.2.1 Algorithmic scanning

As obfuscated malware grew more commonplace, detection moved to methods that used malware-specific heuristics for detection. An example of such a heuristic is *smart scanning* which was able to beat simple obfuscations by ignoring NOP instructions or by performing matching only in the scope of basic blocks. *X-Ray scanning* on the other hand targeted malware encrypted with weak ciphers. The method used a brute-force attack to uncover the encryption key and once uncovered performed a byte signature matching. *Filtering* restricted detection to only those files that could contain malicious code.

Finally a generalized version of heuristic detection was dubbed *algorithmic scanning* and introduced specialized malware description languages that aided the creation of detection procedures. An algorithmic detector was then essentially an interpreter of such a language coupled with a database of detection procedures.

### 2.2.2 Code emulation

Emulation-based detection methods were developed as an answer to strong polymorphic malware. In an emulation-based detector, the suspicious executable is allowed to run in a controlled environment. In case of a polymorphic malware, the packed or encrypted malicious code will eventually be exposed in the memory of the environment, where it can be again detected by a byte signature.

The success of this method is however dependent on how precisely the controlled environment simulates a real host. If the malware detects that it's running in a controlled environment it can halt any malicious activity and perform benign computations. Or the malware can exhibit its malicious behavior only under certain conditions, such as time or a certain host system language localization.

The main disadvantage however is that an antivirus must not hinder a legitimate user in his use of a system and its files. Therefore the detection method must be fast, which in the context of an emulation-based detector means that it has only limited time to spend on a single file. So the simplest way for a malware to avoid detection is to perform benign computation until the detector runs out of time.

## 2.3 Behavioral Detection

## Chapter 3

# Static Analysis

### 3.1 Posets and Lattices

### 3.2 Abstract Interpretation

## Chapter 4

# Formal Models

### 4.1 Tree Automata

#### 4.1.1 Inference

## Chapter 5

# The LLVM Compiler Framework

### 5.1 Structure and Design

### 5.2 Intermediate Representation

## Chapter 6

# Detector Design and Implementation

- 6.1 Input generation
- 6.2 Taint analysis
- 6.3 Classifier Inference
- 6.4 Testing and Results



## Chapter 7

# Conclusion

### 7.1 Discussion

### 7.2 Future Work

# Bibliography

# Appendices

## List of Appendices