République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Batna 2 – Mostefa Ben Boulaïd
Faculté de Technologie
Département de Génie Industriel

Thèse

Préparée au sein du laboratoire d'Automatique & Productique

Présentée pour l'obtention du diplôme de :
Doctorat en Sciences en Génie Industriel
Option : Génie Industriel

Sous le Thème :

An Optimized Approach to Software Security via Malware Analysis

Présentée par :

OURLIS Lazhar

Devant le jury composé de :

M. ABDELHAMID Samir	MCA	Université de Batna 2	Président
M. BELLALA Djamel	MCA	Université de Batna 2	Rapporteur
M^{me}. BOUAME Souhila	MCA	Université de Batna 2	Examinateur
M.DJEFFAL Abdelhamid	MCA	Université de Biskra	Examinateur
M.KAHLOUL Laid	Prof	Université de Biskra	Examinateur
M. BENMOHAMMED Mohamed	Prof	Université de Constantine 2	Examinateur

Novembre 2020

To my parents and family

Contents

Chapter

List of Figures

List of Tables

Program Listings

Abbreviations

AC	Aho-Corasick
ACL	Access Control List
ASIC	Application-Specific Integrated Circuit
API	Application Programming Interface
ATM	Automated Teller Machine
AV	Anti-virus
AVX	Advanced Vector Extension
BM	Boyer-Moore
C&C	Command & Control
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CW	Commentz-Walter
DFA	Deterministic Finite Automaton
DOS	Denial Of Service
FAT	File Allocation Table
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Arrays
FTP	File Transfer Protocol
IA-32	Intel Architecture 32-bit
ICC	Intel® C++ compiler
IDS	Intrusion Detection System
IM	Instant Messenger
IPP	Integrated Performance Primitives
IPS	Intrusion Prevention System
IRC	Internet Relay Chat
KMP	Knuth-Morris-Pratt
KSN	Kaspersky Security Network
MBR	Master Boot Record
MKL	Math Kernel Library
MtE	Mutation Engine
OS	Operating System
P2P	Peer-to-Peer
PC	Personal Computer
PE	Portable Executable
POP	Post Office Protocol
RSA	Rivest–Shamir–Adleman
SIMD	Single Instruction Multiple Data
SMTP	Simple Mail Transfer Protocol
SSE	Streaming SIMD Extensions
URL	Uniform Resource Location
VB	Visual Basic
VBS	Visual Basic Script
WM	Wu-Manber

Publications in support of this dissertation

L. Ourlis and D. Bellala, "SIMD Implementation of the Aho-Corasick Algorithm using Intel AVX2", *Scalable Computing: Practice and Experience*, vol. 20, no. 3, pp. 563-576, 2019. Available: 10.12694/scpe.v20i3.1572.

Other publications

1. D. Bellala, L. Ourlis and H. Smadi, "Optimal Planning of Nurses' Redeployment at Patient Overflow", in *AASRI International Conference on Circuits and Systems (CAS 2015)*, Atlantis Press, 2015. Available: https://doi.org/10.2991/cas-15.2015.84

2. D. Bellala, L. Ourlis, H. Smadi, "Genetic Algorithm Optimization for the Average Position Analysis of a Welding Robot through a Metal Spar", in *MIC'2015 Metaheuristics International Conference*, Agadir, Morocco, 7-10 June 2015.

3. D. Bellala, H. Smadi, A. Medjghou and L. Ourlis, "Constructive and Enhanced Heuristics for a Welding Process Optimization of a Central Girder of a Semi-Trailer", in *6th International Conference on Metaheuristics and Nature Inspired computing*, Marrakech, Morocco, 27-31 October 2016. sciencesconf.org:meta 2016:108942.

4. Dj. Bellala, L. Ourlis and H. Smadi, "Optimized Positioning of a Robot Arm boring Regularly Distributed holes on a metallic sheet using Simulated Annealing Heuristic" in *6th International Conference on Metaheuristics and Nature Inspired computing*, Marrakech, Morocco, 27-31 October 2016. sciencesconf.org: meta2016: 108942.

5. Dj. Bellala, H. Kalla and L. Ourlis, "Optimization of an Underground Water Pipeline Building using Swarm Intelligence Algorithm PSO" in *7th International Conference on Metaheuristics and Nature Inspired computing*, Marrakech, Morocco. 27-31 October 2018. sciencesconf.org: meta2018.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor, Dr. Bellala Djamel, for his support and guidance during the course of my doctoral thesis.

Furthermore, I would like to thank my best friend Mr. Bezza Badredddine for his expert advice in programming Intel processors, and particularly in writing AVX code. Without his help, insightful comments and feedback, this dissertation would not be possible.

I want to thank my wife and my daughter. I spent many times working throughout my thesis, and they have always been supportive and understanding.

I want to thank my father, my mother, my brother and my sisters.

Finally, I want to thank my work colleagues at CNL Batna agency.

Abstract

Today, most anti-malware engines are heuristics. They classify objects, data streams and memory areas as benign or malicious based on their behavior. Most anti-virus manufacturers recognize that heuristic approach by itself allows to achieve up to 90% efficiency in terms of detection rate, but consumes more of system resources: Such anti-malware engines, available as open-source, are extremely inefficient in terms of system load since they extensively use machine learning algorithms. To reduce such system load, signature-based detection technology, which allows to filter most of all known malware samples, is strongly recommended to be used in conjunction with heuristics and cloud-based detection technologies. In this work, we present a fast signature scanner to detect malware, based on an improved Aho-Corasick (AC) algorithm for pattern searching, designed so that it can benefit from vectorization techniques, which add a form of data parallelism to the AC code. It is implemented with Intel© Advanced Vector Extensions (AVX2) packed instructions.

Résumé

De nos jours, la plupart des moteurs d'analyse de solutions anti-malware sont heuristiques. Ils classent les objets, les flux de données ainsi que les zones mémoires comme bénins ou malveillants en fonction de leur comportement. La plupart des fabricants d'antivirus reconnaissent que l'approche heuristique permet d'atteindre jusqu'à 90% d'efficacité en termes de taux de détection, mais consomme davantage de ressources systèmes : de tels moteurs anti-malware, disponibles en open-source, sont extrêmement inefficaces en termes d'utilisation de ressources système car ils font souvent appel à des algorithmes d'apprentissage automatique. Pour réduire cette charge système, il est fortement recommandé d'utiliser la détection par signature statique, qui permet de filtrer à elle seule la majorité des échantillons de programmes malveillants connus, en conjonction avec les technologies de détection heuristiques et celles basées sur le cloud. Dans ce travail, nous présentons un scanner de signatures rapide pour la détection de programmes malveillants, basé sur une version améliorée de l'algorithme Aho-Corasick pour la recherche de chaînes de caractères (ou motifs), conçue pour pouvoir bénéficier des techniques de vectorisation qui ajoutent une forme de parallélisme de données au code de l'algorithme. La solution proposée est implémentée en utilisant le jeu d'instructions d'Intel® Advanced Vector Extensions (AVX2).

ملخص

في الوقت الراهن، معظم محركات مكافحة برامج فيروسات الاعلام الالي هي محركات إحتمالية، أي أنها تصنف العناصر وتدفق البيانات ومناطق من الذاكرة على أنها ضارة أو غير ذلك (خالية من الفيرويسات) بناءً على سلوكها. الطرق الاحتمالية تسمح بتحقيق نسبة تصل إلى 90٪ من حيث معدل الكشف، ولكنها تستهلك نسبة معتبرة من موارد النظام. للتقليل من هذا العبء على النظام ، يوصى باستخدام تقنية الكشف القائمة على البصمة، والتي تسمح لوحدها بتصفية نسبة كبيرة من عينات البرامج الضارة المعروفة، بالترابط مع تقنيات الكشف القائمة على الطرق الاحتمالية والتقنيات القائمة على الكلاود. في هذه الدراسة، نقدم كاشفا لبصمات الفيروسات ذو مردودية عالية للكشف عن البرامج الضارة، صمم استنادًا إلى نسخة محسنة من خوارزمية Aho-Corasick (AC) المستعمل في البحث عن سلسلة الحروف، والمبرمجة بحيث يمكنها الاستفادة من تقنية "Vectorization"، التي تضيف شكلاً من التوازي في معالجة البيانات. الحل المقترح تم تجسيده باستعمال تعليمات Intel® Advanced Vector Extensions (AVX2).

Chapter 1

1 Introduction

Software is everywhere, it runs your car and it controls your cell phone. It is also how you access your bank financial services, receive electricity and natural gas, and fly from coast to coast [1]. Our numeric society depends highly on software to handle the sensitive and high-value data on which people's privacy, livelihood, health, and very lives depend. Its security is challenged on a daily basis by malware (malicious code such as viruses and worms).

The growth of the Internet connections, rapid multiplication of social networks and the advent of 4G that offers great opportunities for mobile networks led to an exponential increase in malware attacks. Every day a hundred million of samples with potentially harmful executable code are submitted to data security companies for analysis. From the threat landscape of 2015, Symantec Corporation reports that 430 million new unique pieces of malware were created during the same year, that is to say an increase of 36% compared to 2014 [2]. In their Malware Statistics Bulletin of 2016, Kaspersky Security Network (KSN), using data collected from computers running Kaspersky Lab products from 213 countries and territories worldwide, has detected a total of 4,071,588 unique malicious programs, recognized 261,774,932 unique malicious URLs and blocked attempts to launch malware capable of stealing money via online banking on 2,871,965 devices [3]. IT security experts believe that malware threats are their biggest worry for the upcoming years. They are also worried about the fact that the new malware is too sophisticated for being analyzed and then be out of their analysis and security capabilities.

Security applications such as anti-malware solutions, intrusion detection/prevention systems (IDS/IPS), and firewalls are very specialized applications that rely heavily on signature scanning and require pattern-matching capabilities at very high speeds with less memory usage. In fact, pattern-matching is a time and processor-intensive operation, and a bottleneck to the performance of these applications. In malware detection, pattern-matching

algorithm is at the heart of the anti-virus[1] (AV) scanner, it handles hundred thousand of patterns and contribute to more than 90% of the total running time of the system.

The basic form of malware detection is signature-based. When a new malware is identified, a new signature is created and added to the database. The signatures are typically byte-streams or small hashes that contain enough information specific to particular malware instances [4]. During the scanning process, the AV engine will compare the file under inspection to the database of all known malware signatures. If the file matches one signature, then the file is flagged as being infected and the AV engine knows exactly which procedure to apply in order to perform disinfection.

This approach is challenged by the increasing number of malware (large database to handle by the AV engine) and the shortened duration time for the effectiveness of signatures. It is effective only if the database contains the malware signatures (malware database must be frequently updated). Its major drawback is its inability to detect new or unknown malware and zero-day attacks[2]. The strength of the signature based-detection relies on the assumption that the behavior of one instance of a malicious software is representative of all instances [5]. This means, for example in case of a worm-type malware that usually propagate via network connections, the behavior of all compromised nodes can often be reliably predicted by just studying one node of the worm. Despite its drawback, this detection technique remains the most commonly used method of identifying malware and has many advantages which include its scanning speed that can be further increased by using other detection algorithms or changing the implementation of the existing ones. John Aycock wrote *"With hundreds of thousands of signatures to look for, searching for them one at a time is infeasible. The biggest technical challenge in scanning is finding algorithms which are able to look for multiple patterns efficiently, and which scale well"* [6, p. 55].

In the following thesis, we introduce efficient algorithmic designs for multiple pattern-matching that utilizes modern SIMD instructions introduced in 2013 with Intel processors: SIMD vectorization of the Aho-Corasick (AC), a widely used algorithm in today's malware scanners.

[1] However, anti-virus software has historically targeted a specific subset of malware, it is used interchangeably with anti-malware software. Both terms refer to software designed to detect, protect against, and remove malicious software.

[2] Zero-day attacks, or zero-day exploits, refer to attacks on software security weaknesses (vulnerabilities) that have not been patched or made public.

1.1 Motivation for the research

The motivation for the research undertaken in this PhD study is based on two important aspects for malware detection. First, signature-based detection is still a predominant technique to protect against malware because byte-stream (basic strings and regex patterns) signatures remain the only accurate way of detecting malicious code, which are not expected to disappear anytime soon according to AV compagnies. If signature-based detection does not guarantee perfection, it guarantees the protection from the most widespread threats. Second, the process of matching this type of signatures is usually the slowest, compared to matching other signature types such as MD5 hashes. Therefore, the focus is on byte-stream signatures since they are the most time-consuming operation of the scanning process (more than 90%) in order to reduce the overall scanning time of the AV software.

1.2 Thesis scope

This thesis is part of the optimization of malware detection. In particular, the studies and research conducted here aim to improve the performance of the AV engine when matching objects against malware byte-stream signatures stored in the program's malware definition database (improving the scanning process). To this end, we have analyzed the problem of malware detection using signature scanning and its reliance on the AC algorithm, a multi-pattern matching algorithm used in AV software like ClamAV and in many IDS like Snort. The proposed solution benefits from features implemented in modern processors that provide direct support for vectorization, a hardware optimization technique synonymous with early vector supercomputers, notably the various Cray platforms (like the Cray-1, 1975). Developers typically relay on the process of revising the scalar-oriented code loops, often by taking advantage of the compiler's auto vectorization, to generate vectorized objects utilizing the AVX/SIMD instructions.

1.3 Thesis outline

Chapter 2 presents an overview of the world of malware. It provides a brief historical of malware demonstrating how it evolved side-by-side with software technology and this co-evolution is expected to continue. It outlines malware propagation vectors, or methods to spread malware, and concealment strategies deployed by malware authors from a simple encryption to modern metamorphism. This chapter presents malware types or classification

according to John Aycock, Adelman, Microsoft and Kaspersky Labs. It also provides information on how malware is analyzed and detected.

Chapter 3 considers the pattern-matching problem and aims to give a comprehensive description on a subject and the application fields where it is applied. A complete review of some popular exact pattern-matching algorithms such as the Boyer-Moore and the AC algorithms are provided with their implementation.

Chapter 4 provides the bulk of our work; it provides a basic overview of the concepts behind parallel implementation of the AC algorithm using Intel® SIMD technology (AVX2), and gives a step-by-step guide to the vectorization process of the algorithm.

Chapter 5 presents a direct application of a vectorized AC algorithm: an implementation of a fast signature scanner that uses the optimized AC algorithm to detect malware.

Chapter 6 summarizes the thesis, and highlight directions for future work.

Chapter 2

2 Malware Overview

Most security threats start from the web, an effective place that provides great opportunities for fraudulent or abusive behavior and illegal access to business and sensitive client data. A malicious web page exploits a vulnerability in a program to gain arbitrary code execution, which allows to downloading and installing a malicious program, or malware, that infects the host.

2.1 A Brief History of Malware

A brief incursion in the history of malware shows that the idea of a computer virus, or a self-replicating string of code, extends back to 1949, when early computer scientist John Von Neumann wrote the *"Theory and Organization of Complicated Automata"*, a paper where he postulated the potential for a self-reproducing automata, or how a computer program could reproduce itself. In the 1950s, Bell Labs employees brought Von Neumann's idea to life when they created "Core Wars", a game where programs competed for control of the system [7].

The earliest viruses began to appear in the early 1970s. The first known computer virus was the "Creeper Worm" written for the Tenex operating system by Bob Thomas in 1971 at BBN Technologies. Thomas wanted to experiment a self-replicating program, he showed that it was possible for a computer program to move across a network but disappointed because Creeper didn't install instances of itself on several connected computer targets: Creeper gained access independently through a modem to the ARPANET[3] and copied itself to a remote system (Tenex terminal being infected), without being detected, where it displayed the message: "I'M THE CREEPER : CATCH ME IF YOU CAN", but then stopped, found another Tenex system, opened a new connection and removed itself from the previous one as it propagated. Shortly

[3] ARPANET or Advanced Research Projects Agency Network is a US military packet-switching network with distributed control and one of the first networks to implement the TCP/IP protocol suite. Both technologies became the technical foundation of the Internet.

after, Thomas' colleague, Ray Thomlinson – the inventor of e-mail, developed an enhanced version of Creeper that not only moved through the network, but had the ability to replicate itself. To complement his work, he created the Reaper program in 1973, which moved through the network and removed any copies of Creeper it found. Thomlinson did not realize, but he had created the very first piece of AV software. Although Creeper has not achieved all its goals and Reaper was a very rudimentary AV, both marked milestones in the history of computer virology [8].

When did the term "computer virus" arise? In November 1983, Fred Cohen, a graduate student from the University of Southern California demonstrated a computer virus during a security seminar at Lehigh University in Pennsylvania when inserting a floppy disk, containing his code written for a parasitic application that seized control of computer operations, into a VAX11/750 mainframe computer. The attendees noted how code hidden in a Unix program installed itself and took control of the system in a few minutes, bypassing all security mechanisms current at that time, replicating and spreading to other connected machines in the network. In 1984, Cohen published the results of his work in a paper "*Computer Viruses – Theory and Experiments*" [9] that defines a major computer security problem called a virus. It was the first paper to explicitly call a self-reproducing program a "virus", a term introduced by Cohen's academic adviser Prof. Leonard Adelman - the co-inventors of the RSA encryption [10].

In the early days of the personal computer (PC), before networks and internet connectivity became widespread, computer viruses spread on removable media and much often by infected floppy disks. Then a virus evolved to a computer worm, a standalone malware that spread through networks, and unlike a computer virus, it does not need a host program to propagate. Nowadays, malicious software comes in many forms including rootkits, Trojans, bots, ransomwares, etc., and can spread through email attachments, scam links on corrupted web pages, Internet file downloads and social engineering. Since malware inception, one thing has remained constant: it's continues to grow in frequency and sophistication, and still one of the most destructive menace affecting small and medium businesses (SMBs), large enterprises and government agencies. According to Cybersecurity Ventures, 91% of cyberattacks begin with spear phishing email, which are commonly used to infect businesses with ransomware. It predicts there will be ransomware attacks on businesses every 14 seconds with the global damage costs are to reach $11.5 billion annually by 2019 [11].

- **History of Malware — From an Academic Beginning to the Morris Worm, and the Viral Era (1970 – 1999)**

Early malware was simple file infector or boot sector, often spreading offline through the sharing of disks. However, the advent of new communication mediums gave rise to new opportunities for malware authors to cause trouble through sending their malicious creations by taking advantage of networks. In the following, a non-exhaustive list of the most significant early malware before the year 2000.

- **1971 – Creeper:** An experimental self-replicating program designed to test how a program might move across a network between connected computers.
- **1974 – Rabbit:** A self-replicating program written in Visual Basic Script (VBS) that made multiple copies of itself on a single computer until it crashes the system. It was named so for the speed at which it replicated [12].
- **1982 – Elk Cloner:** Written by Richard Skrenta, a 15-year-old student from Pittsburgh, Elk Cloner is one of the earliest widespread, self-replicating viruses designed to affect Apple computers through a floppy disk. The Elk Cloner virus, described by its author as "some dumb little practical joke", was the first virus that affected a machine during its booting process. Every 50th time that the machine would boot up, Elk Cloner displayed a friendly little poem on the infected system: "It will get on all your disks; It will infiltrate your chips; Yes, it's Cloner…". No serious damage was known to have been caused by the virus, but it is considered being responsible for the first large-scale computer virus outbreak in history [13].
- **1986 – Brain Virus:** Generally considered as the first IBM PC compatible boot-sector virus to infect MS-DOS computers, and one of the first stealth virus since it redirected the user to an early stored boot sector each time a program tried to access it. Its origin stems from two brothers, Basit Farooq Alvi and Amjad Farooq Alvi, from Lahore, Pakistan. They created the Brain virus to test flaws in their company's software: they just wanted to prevent their customers of making illegal copies of the software they had developed. The brothers claimed that their creation was intended to be harmless. They created the virus only for illegal copies of the software and put their contact details such as their names, phone numbers, and their shop's address in the virus code, so that users whose computers were infected (running a pirated copies of their software) could contact them for inoculation. Despite the authors' claims, many users around the world reported that the virus not simply renaming the computer floppy drive

and displaying a copyright message about the pirated software but it wiped data and caused their drives to become extremely slow and not responding. In the following the message displayed by the Brain virus on infected computers: “Welcome to the Dungeon © 1986 Brain & Amjads (pvt). BRAIN COMPUTER SERVICES 730 IZANAMI BLOCK ALLAMA IQBAL TOWN LAHORE-PAKISTAN PHONE: 430791,443248,280530. Beware of this VIRUS.... Contact us for vaccination...” [14].

- **1988 — Morris Worm:** Created by Cornell student Robert Tappan Morris aged 22 when trying to gauge the size of the Internet. To do so, he wrote a program designed to propagate across networks, infiltrating DEC VAX and Sun terminals running BSD UNIX connected to the Internet by exploiting three weaknesses:
    1. Vulnerability in the debug mode of Unix’s famous “Sendmail” program;
    2. A buffer overrun hole in the “Finger” daemon protocol, a software that helped to find logged on users in the system;
    3. REXEC/RSH (Unix programs that enable the user to run shell commands on a remote computer) network login set up without passwords.

  The worm corrupted thousands of Unix-based systems (more than 6000 University, military, and research center computers) and bring the network to its knees the first day released: the worm penetrated and crippled 10% of the Internet, the rest of the networks slowed down. Even though this was not the first worm, it was considered the first major attack, which was a wake-up call for Internet security engineers, to consider the risk of software bugs and start research and development for network security, and led to the creation of the Computer Emergency Response Team (CERT) for issues that might affect the Internet as a whole [15, 16].
- **1991 — Michelangelo Virus:** A variant of Stoned, a family of boot sector viruses dating from 1988 that move the original master boot record (MBR) of the hard drives and the boot sector of floppy disks. Named so because it was designed to infect DOS systems on March 6th, the birthday of the famed Renaissance artist Michelangelo. On that date, the virus would trigger its payload: it would overwrite the first one hundred sectors of the hard disk or any diskette present, with random characters. In addition, it does not check if the MBR has already been infected by the same or other boot viruses, and moves the previous virus to the original MBR location, thus making recovery of the MBR impossible, which resulted in loss of data. John McAfee, an information security pioneer, predicted that the virus could have infected more than 5 million machines, which was important that days. On March 6, 1992, the impact of the virus

differed from what the world's media was expected: only a few thousand machines were infected [17].

- **1999 — Melissa Virus:** The first mass-mailing virus also known as Kwyjibo (or Simpsons, the popular television series), written in Visual Basic (VB) by David L. Smith from New Jersey. The virus struck users of Microsoft word-processing programs Word 97 and Word 2000 by inserting a macro module named Melissa, then spreading as an email attachment using Microsoft Outlook, noted with the subject "*Important Message from [the name of someone]*" and body text that reads *"Here is that document you asked for...don't show anyone else*". Melissa virus started as an infected file *"list.doc"* posted on *"alt.sex"* newsgroup, claiming to be a list of usernames and passwords for 80 obscene websites that require memberships. Once the file opened, this causes the Melissa macro to be executed (when macros were enabled). The VB macro copies the virus-infected file into the "*normal.dot"* file used by Microsoft Word for custom settings. It inserts (*HKEY_CURRENT_USER Software Microsoft Office "Melissa?"="...by Kwyjibo")* entry in the Windows registry as a signature to avoid reinfection, and finally opens Microsoft Outlook and mails itself to the first 50 addresses in the Outlook address book. If Internet access or Outlook are not available, it would still infect other Word documents. Melissa was mostly innocuous and did not cause too much damage to infected user's PC. However, its mechanisms caused Denial of Service (DOS) attacks, which brought down the servers as they became overloaded with messages created by the virus (organizations' servers that relied on Microsoft Outlook as their email client) [18].

- **History of Malware — Use of Toolkits to Create Malware and Astonishing Rates of Infection (2000 – 2009)**

Between the years 2000 and 2009, Internet and email worms made headlines around the world. They grew significantly in number and in how fast they spread. At the same time, we witnessed the appearance of malware toolkits, crimeware kits targeting websites, and SQL injection attacks. Here's a summary of the most notable malware released during this period.

- **2000 – ILOVEYOU Worm:** Developed in VBS by two young programmers Reonel Ramones and Onel de Guzman from Philippines. The VBS worm, also known as Love Letter or Love Bug, is a destructive self-replicating computer worm that hit an estimated 50 million Windows PCs on May 2000. It started spreading via MS Outlook as an email message masqueraded in a love letter with the subject line *"ILOVE YOU"*

and the attachment *"LOVE-LETTER-FOR-YOU.txt.vbs"*. It was the first successful worm that propagate using social engineering. The *".vbs"*, as it is an extension for a file type that is known by the OS, was often hidden by default on Windows versions of the time. The worm spread worldwide, inflicting $15 billion in damages, corrupting different types of files with media extensions (such as images, documents and mp3 files), and shutting down computer systems of different corporations, government offices including the Pentagon and the CIA, Wall Street firms, etc. According to security expert Graham Cluley, there was nothing particularly smart about the worm' code that explained why it had spread around the world so quickly, the reason was that the worm responded to a universal need: the desired to be loved [19].

- **2001 – Anna Kournikova Worm:** On February 11 of the year 2001, Jan de Wit, a 20-year-old Dutch student who called himself "OnTheFly", unleashed the Anna Kournikova worm by disseminating it onto the newsgroup and it quickly began to spared. It only taken de Wit a few hours to create this malicious program using a Visual Basic Worm Generator toolkit developed by an 18 years old Argentinean programmer called [K] Alamar. The Anna Kournikova worm did not corrupted data on the infected machines but is very similar to the ILOVEYOU worm, in the way it is a VBS worm that propagated using social engineering -choosing a very attractive tennis player Anna Kournikova "the bait" as a name to the email attachment, and causing troubles in email servers around the world [20].
- **2003 – SQL Slammer Worm:** Sometimes called Sapphire, was the fastest computer spreading worms of all time, and surprisingly no worm has beat it since. It spread so widely and so quickly that human response was ineffective. As it began propagating throughout the Internet, the worm infected more than 90% of vulnerable hosts (about 75,000) within 10 minutes, slowing down Internet traffic, flooding networks worldwide, cancelling airline flights, interfering with election in Canada, and causing significant disruption to financial and government institutions. The worm exploited a buffer overflow vulnerability in Microsoft's SQL Server and Microsoft SQL Server Desktop Engine (MSDE) 2000. Although the patch had been released by Microsoft six months earlier, many organizations had not yet applied it. The worm spread as an in-memory process (only 376 bytes in size), and thus fitting inside a single packet, which allowed it to enjoy a rapid propagation from infected servers to different routers worldwide. These routers upon infected started sending the worm to random IP addresses, and sometimes crashed under the burden of extremely high traffic to handle.

This caused a denial of service (DOS) attacks condition on targets, which resulted in slowing down the networks including the Internet [21].

- **2004 – Cabir Worm:** Although this worm had no harmful effects, it is noteworthy because it is the first malware designed to infect mobile phones, and a wake-up call to demonstrate vulnerabilities in technology. The worm affected phones running the Symbian OS that support Bluetooth technology (Nokia's Series 60 smartphone platform). The worm arrived in a phone messaging inbox as a telephone game file "caribe.sis", and when the user clicked the file and chose to install it, the worm became activated (and each time the phone is turned on). When this happened, it would display "Caribe" on the phone screen, and started looking for new phones to infect over wireless Bluetooth signals [22].
- **2005 – Sony BMG Rootkit:** The Sony BMG scandal was unveiled on October 2005, when researcher Mark Russinovich posted on his blog detailed description of the rootkit software secretly installed by the company on millions of its music CDs, as part of its deceptive and illegal copy-protection measures. The rootkit, which often hides its existence and the existence of other malicious software from security applications (such as AV software), creates vulnerabilities on compromised machines, thus making them susceptible to infection by other worms and viruses. It also prevents other software from playing and/or burning copies of the CDs [23].
- **2007 – Zeus Trojan horse:** Often called Zbot, a most successful pieces of botnet designed for cyber thieves. First detected in July 2007 when it was used to steal online information from the U.S. Department of Transportation, and became widespread in March 2009. Originally, the malware only targeted machines running Microsoft Windows, but some newer versions of the Trojan have been found on some mobile devices running Symbian, BlackBerry or Android systems. While the Trojan can do a number of malicious and nasty things on infected machines, it mainly created a botnet, and then acted as a financial services Trojan designed to steal banking credentials by man-in-the-browser keystroke logging or form grabbing. Zeus was mainly spread through spam messages that often came in the form of email, and drive-by downloads via corrupted legitimate websites. The malware was used worldwide by cyber criminals. By 2010, it was estimated that the Trojan accounted for 44% of the banking malware infections, and caused damages worth US$100 million since its inception. The most famous case involved the arrest of more than 100 computer hackers by the

FBI for bank fraud in Eastern Europe after stealing $ 70 million using the Trojan [24, 25].

- **2008 – Koobface, Torpig & Conficker malware**

July 31 – Koobface: A multi-platform worm computer that targeted Microsoft Windows, Mac OS X, and Linux systems. It propagated primarily through social networking websites (like Bebo, Facebook, Skype, Netlog, Yahoo Messenger, MySpace, Twitte, etc), and uses compromised machines to build a peer-to-peer botnet. The worm much often sent malicious video links via social networking sites, which redirected users to false YouTube pages hosted on compromised web servers. Koobface allows hackers to track and record sensitive information about compromised users, such as logins and passwords entered on particular websites, and even find out sensitive financial data such as credit card numbers and banking information [26].

March 3 – Torpig: Torpig, also called Anserin or Sinowal, is a type of botnet that spread through Microsoft Windows systems infected by the Mebroot rootkit by a variety of Trojan horses. It exploited vulnerabilities found in older versions of web browsers and some popular software such as Java, Adobe Reader and Flash, for the purpose of stealing sensitive data including emails, Windows passwords, FTP credentials, POP and SMTP accounts. Once the machine is infected, it joins the network of zombies for the botnet, and thus allowing attackers full access to the system. One of the key points of interest about the Trojans used is that they can exist for so long (more than a year) and quietly collect information from the compromised machine without the victim's knowledge [27]. Trojans' creators periodically release new variants to ensure they stay undetectable and maintain their uninterrupted grip on compromised machines. The use of the Mebroot – a master boot record (MBR) rootkit, which is a new breed of stealthy malware, replaces the compromised system's MBR upon infection, and keeps the amount of system changes to a minimum. This makes it very difficult to detect from within the infected system by AV software or other protection means [28]. It is quite surprising that the Mebroot write to the MBR from within Windows OS then executes itself during the OS loading "starting process" without the need for any registry or file changes [28]. By November 2008, the Torpig has compromised more than 270,000 online bank accounts and 240,000 credit and debit cards from financial institutions worldwide [27] . In 2009, a team of researchers at UC Santa Barbara took control of the Torpig botnet for ten days. During that time, they observed more than 180 000 infected machines (number of actual bots), collected over 70 GB of stolen data and redirected

1.2 million IPs to their private Command & Control (C&C) server. Within just ten days, the Torpig was able to steal 8,310 login accounts at 410 different institutions, and 1,660 unique credit and debit card numbers worldwide, including cards from Visa (1,056), MasterCard (447), American Express (81), Maestro (36), and Discover (24) [29].

November 21 – Conficker: Conficker worm, also known as Downadup or Kido, targeted Windows PCs worldwide. It could have originated from either Ukraine or China, and gained a great media attention expecting the worm to deliver massively destructing payload by April 1, 2009. While that never happened "false alarm", the worm quietly turned more than 10 million PCs into servers of e-mail spam (through the installation of another virus "Waledac"), and made them joining a network of zombies that received commands from a remote server (a botnet) [30].

- **History of Malware — State-sponsored Cyber Warfare, Increasingly Pervasive and Profitable (2010 – Present time)**

Between 2010 and the present time, malicious programs gained in sophistication and complexity with dramatic growth in ransomware and other malicious schemes. Now they have the ability to infiltrate factories, affect industrial control systems and become more profitable. In June 17, a state sponsors malware "Stuxnet", appeared. It was the first discovered malware targeting military systems. CryptoLocker, CryptoWall and CryptoDefense ransomwares, using cryptoviral extortion techniques, enter the scene by the end of 2013. Recently, other more elaborated malware created by well-funded development teams, upped the game dramatically such as the WannaCry ransomware and the Petya malware. These are some notable varieties of malware that have had a major impact between 2010 and today.

- **2010 – Stuxnet Worm:** A most complex and sophisticated malware ever created, designed specifically to target industrial controllers made by Siemens, used in Iran's uranium enrichment equipment. The worm exploited zero-day vulnerabilities[4], rootkits, AV evasion techniques, network infection routines, peer-to-peer updates, a C&C interface and included the ability to infect hardware "physical payload" as well as software. Kaspersky Lab's R. Schouwenberg estimated that it took two to three years for a team of ten developers to create the worm in its final form [31]. It is widely acknowledged that the worm was created by U.S. and Israeli intelligence agencies. The

[4] A zero-day vulnerability is a software issue with no known patches. Until a fix is available, cybercriminals could exploit it.

classified worm development program has been codenamed "Operation Olympic Games" [31]. Stuxnet targeted only programmable logic controllers (PLCs), used to control and automate machine processes for separating nuclear material, including the rotational speed of the centrifuges that powers nuclear weapons and reactors. First, it scans networks looking for computers running the Microsoft Windows OS, then seeking out Siemens Step7 software that is used to reprogram the PLC devices. Once finding a PLC computer, the malware updates its code over the Internet and begins its damage by sending instructions to the electro-mechanical equipment the computer controlled. At the same time, false feedbacks are sent to the main controller in a way that anyone monitoring the equipment would have had no indication of a problem until the equipment crashes [32]. The Stuxnet compromised over 200,000 machines mainly in Iran, and severely ruined its nuclear installations by destroying 10% (about 1000) centrifuges between November 2009 and January 2010 [32].

- **2013 – CryptoLocker:** Is a Trojan ransomware targeting Microsoft Windows. It encrypts files on compromised systems, using RSA encryption with the private key stored only on the malware's control servers, and then asks for a ransom to recover the files back. While the Trojan itself is not difficult to remove, the affected files remain encrypted until the victim makes payment. CryptoLocker uses the Gameover ZeuS botnet and malicious email attachments to spread across the Internet seeking out Windows PCs, and then encrypts files stored on local or mounted network drives. Between September 2013 and May 2014, CryptoLocker affected approximately 500,000 victims worldwide, and there were around $30 million in ransom payments [33].
- **2015 – BlackEnergy:** Towards the end of the year 2015, about half of the dwellings in the Ivano-Frankivsk region of Ukraine (about 1.4 million inhabitants) were left without electricity for few hours. According to the Ukrainian news media TSN, the cause of the blackout is a hacker attack, which was not an isolated case since other Ukrainian energy companies were also targeted by cybercriminals at the same time, using the BlackEnergy backdoor to plant a KillDisk module onto the compromised computers that would render them unbootable [34]. According to Kaspersky Lab researchers, the infection vector used in these attacks was Microsoft Word documents that spread as an attachment in a spear-phishing emails. Upon opening the document, a dialog box appears to inform the user that macros must be enabled to display its content. Activating the macros triggers the BlackEnergy malware infection [35]. The

Ukrainian security company published two screenshots of emails used to spoof the sender address to appear to be belonging to the Ukrainian parliament. This is a good example demonstrating the use of social engineering instead of exploiting software vulnerabilities. If victims are trapped, they end up infected with malware [34].

- **2017 – WannaCry & NotPetya ransomwares**

  May – WannaCry ransom: Also known as Wanna Decryptor, is a bundle of malicious software including a worm, a backdoor and ransomware. It targets Windows PCs by encrypting data and refusing access to the infected systems until receiving the ransom payments (300 to 600 Dollars in Bitcoin cryptocurrency). The malware propagated via "EternalBlue" exploit[5], which takes advantage of a vulnerability in the SMB protocol[6], first uncovered by the USA's National Security Agency (NSA), and was patched by Microsoft at the time to close the exploit, but older versions of Windows and those left unpatched were left open to attacks. The WannaCry holds down systems through email attachment or as a download masqueraded in something harmless that many users end up clicking on it. Within 24 hours, WannaCry had infected more than 230,000 computers, and brought down major computer systems over 150 countries. It affected hospitals, financial institutions, telecommunication companies, and many other industries. The U.S. Government, under Trump administration, officially attributes the WannaCry cyberattack to North Korea [36].

  June – NotPetya ransomware: NotPetya an evolution of the Petya malware, a family of encrypting ransomware discovered in 2016 that propagated via infected email attachments, targeting Microsoft Windows systems and infecting the master boot record (MBR) to encrypt a hard drive's file system, and therefore preventing the system from booting unless the victim makes a payment. In June 2017, NotPetya (the name given by Kaspersky Lab to this Petya variant) was used in global cyberattack, targeting Ukraine and affecting many critical infrastructure of the country. It propagated by taking advantage of the "EternalBlue" exploit, which was earlier used by the WannaCry ransomware. In fact, whether the ransom is paid or not, the system

[5] An exploit is a piece of software, a chunk of data, or a sequence of commands designed to take advantage of a flaw or vulnerability in a computer system, typically for malicious purposes such as installing malware, usually posted by hacker groups.

[6] Server Message Block (SMB) is the transport protocol used by Windows machines for a wide variety of purposes such as file sharing, printer sharing and access to remote Windows services, which operates over TCP ports 139 and 445.

will never be decrypted since this variant is modified so that it is unable to recover its own changes thus leaving the computer completely unusable [37].

- **History of Malware — Is This Just the Beginning?**

Today malware gained much of its initial footing, every device with a microprocessor is at risk. Cybercriminals have successfully deployed malware to infect more devices from computers to just about everything electronic such as smartphones, ATM & vending machines, security cameras, nuclear plants, etc. Recently, a new kind of threat is discovered: a "pre-installed malware" has been found on more than 7.4 million Android devices. Google security researcher Maddie Stone explained how pre-installed applications are exploited to execute malware without drawing the user's attention. This security vulnerability targeted Android's open-source operating system. And since these are pre-installed applications, they cannot be detected by AV software, or removed from these devices, they can only be deactivated. This type of malicious software is harder to spot and more difficult to get rid of. Always according to Stone *"If malware or security issues can make its way as a pre-installed application, then the damage it can do is greater, and that's why we need so much reviewing, auditing and analysis"* [38].

2.2 Malware definition

Malware, or malicious software, is a general term for various types of software designed to infiltrate and compromise a computing device functions, bypass access controls and steal sensitive data. The term "virus", introduced in Fred Cohen's work in 1984 as "*a program that can infect other programs by modifying them to include a possibly evolved copy of itself"* [9] and commonly used to designate a more general class of malicious programs, is misleading since other classes of malicious programs such as Trojans or bots have nothing to do with viruses [39]. In its special publication of July 2013, the National Institute of Standards and Technology (NIST) defines malware as "*a program that is covertly inserted into another program with the intent to destroy data, run destructive or intrusive programs, or otherwise compromise the confidentiality, integrity, or availability of the victim's data, applications, or operating system.*" [40].

In technical terms, a malware consists up to six main parts, although it is not necessary to have all these parts in place at the same time [41]:

1. *Insertion* or *infection mechanism* enables a malware to insert itself on the victim's target and mostly avoids multiple infections;
2. *Avoidance* deploys methods to evade detection;
3. *Eradication* uses methods, which allow a malware to remove itself after the payload has been executed;
4. *Replication* mechanism gives a malware the ability to create instances of itself and to propagate to other targets;
5. *Trigger* is a specified event that initiates the payload execution;
6. *Payload* is the portion of a malware' program that creates an imminent threat of damage to the target victim.

2.3 Malware propagation vectors

Malware authors use a variety of methods, often known as propagation vectors, to spread their malicious creations. Instant messaging, email attachments and drive-by downloads are the most used vectors for spreading malware through social engineering and scam techniques. Ad-based malware, software cracks, video codecs and Microsoft products activators (made available as a free download) are also used by hackers for this purpose. Malware no longer relies on a user action (or a negligent user's inaction) to spread. Some popular applications such as Java & Visual Basic scripts, ActiveX, and Java applets make it possible for it to propagate without the user interaction. Persistent Internet connections provide great opportunities for attackers and malware authors to exploit software vulnerabilities that are not yet patched (before the software vendor publishes a security update to address them). Any other support, medium or mechanism used for distributing or sharing malicious software is considered as a vector for malware propagation. In what follows, we give details of some of the most widely used propagation vectors.

- *Email attachments*: Email attacks exploit vulnerabilities in the email software or in its libraries (usually used to display images, PDF or Word files). When the email software downloads a message and displays it, an embedded manipulated object can exploit these vulnerabilities and causes the malicious code to be executed. Across different threat vectors, email attachment is the most frequently used delivery mechanism to spread malware, no other distribution vector comes close: not compromised websites or network file sharing technologies, nor malvertising campaigns. According to Symantec Internet Security Threat Report of 2017, 74% of compromised emails distributed their payload

via email attachments [42]. Hackers use sophisticated programs that are typically able to retrieve weak email passwords, allowing them to access thousands of email accounts. This means anyone could potentially receive compromised emails from friends, banks (claiming there is something wrong with the account), or big-name corporations (also known as phishing emails) with dangerous attachments and might not hesitate to open them because he trusts them. Fraudulent FedEx and UPS emails are very common [43].

- *Drive-by downloads*: The malware deliver its payload simply by opening a malicious email message, clicking on a deceptive pop-up window or visiting a compromised website without the knowledge of the user [44]. In the latter case, viewing is enough to cause the infection without the need to click anywhere on the malicious web pages of a compromised website. In drive-by download attack, malware authors exploit vulnerabilities within a web browser or its plugins[7] (now mostly deprecated), using scripting languages[8] (JavaScript) to inject their malicious code [45]. Outdated and obsolete software, such as old browsers or old operating systems, can significantly increase the risk of infection via drive-by downloads vector. By now, drive-by downloads may include more sophisticated attacks such as advanced ransomware packages that spread quickly over a network once downloaded [46]. A term *drive-by install*, which refers to installation rather than download, is sometimes used instead.

- *Phishing*: A form of social engineering[9] attacks that use email or malicious web sites and solicit personal data, often financial, from a person's computer to steal their money. The problem with phishing attacks is that humans trust email as a platform, and that's their first downfall [47]. The most known phishing scenario is as follows: a user is suddenly alerted by email from his bank, often suggesting that there is something wrong with his account, and asking him to confirm his login and password. When he clicks the link in the email, he is taken to a compromised webpage that resembles, more or less, to that of

[7] A plugin or addon is a software component that adds a specific feature or improves a specific functionality of an existing program. Two web plugins examples are the *Adobe Acrobat Reader* that enables the web browser to read .PDF files and the *Adobe Flash Player* for playing videos.

[8] A scripting language is a programming language for a special run-time environment used to automate the execution of certain tasks, which alternatively could be executed by a human operator. Scripting languages are also used to extract information from data sets. They are less code intensive compared to traditional programming languages.

[9] Social engineering is the term used for a broad range of malicious activities accomplished through human interactions (attacks that exploit weaknesses of humans rather than holes found in software). It uses psychological manipulation to trick users into making security mistakes or divulging confidential information.

the actual bank — but actually designed to steal his credentials. When a user responds with the requested information, after entering his login and password on the malicious webpage, attackers can use them to gain access to his account [48]. Of all attack vectors, phishing attacks remain the most commonly exploited, they accounted for more than 90% of all successful cyberattacks [49]. Phishing attempts targeting specific individuals or companies are known as *spear phishing*, whereby attackers tailor their messages based on the information gathered about their target to increase their odds of success [47]. The most illustrative example is the case of a Russian cyber espionage Threat Group-4127 targeting email accounts linked to Hillary Clinton's 2016 presidential campaign and the U.S. Democrat National Committee, reported under the name Fancy Bear. According to the report analysis, published in June 2016 by SecureWorks® Counter Threat Unit™ (CTU) researchers tracking the activities of the group, the attack used a spear phishing campaign targeting more than 1800 Google accounts. In particular, the group implemented the *accounts-google.com* domain, and used a Bitly URL-shortening service to hide the location of a spoofed Google login webpage. Users who clicked the link were directed to a compromised Google account login webpage. That's how the group could recover usernames and logins entered to access the contents of the associated Gmail account [50].

- *Ad-Based malware*[10]*:* Online advertisements, or ads, embedding malware owned by hackers and cybercriminals, usually distributed via popular trusted websites that make users potentially vulnerable. In 2016, millions of people could get infected just by visiting some large and established websites such as The New York Times, MSN.COM (owned by Microsoft) and some popular gossip sites [51]. Malicious ads can also be directly embedded into trusted applications. Malware is injected onto the user's machine by clicking or sometimes by just viewing a malicious ad. When this happened, victims are often redirected to an exploit kit[11] landing page on a compromised server. Once the victim's machine successfully connected to the server, the exploit kit executes to determine software vulnerabilities that could be used by the attacker to upload and execute malware (Experts call these attack vectors "drive-by attacks"), and could also

[10] Not to confuse with *Adware (advertising-supported software)*, which is a software that automatically displays or downloads advertising material such as banners or pop-ups when a user is online.

[11] An exploit kit is a type of malware that runs on web servers with the purpose of identifying software vulnerabilities in a client's machine. Angler kit surfaced in 2013 is an example of a cyber criminals' favorite exploit kit responsible for 80% of drive-by attacks that time.

perform additional malicious actions such as stealing data, locking or holding the machine ransom, or adding it to a botnet. All this happened behind the scenes, out of sight and without any interaction from the user [52]. However, malicious ads can be blocked by installing an appropriate program or a browser plugin against ads and self-executing scripts. But ad publishers don't like this trend. They always claimed that blocking ads is attacking the freedom of speech. They have a good alibi since ads bring them a lot of money [51].

2.4 Malware concealment strategies

Malware concealment is a real challenge for AV analysts. Malware authors use different strategies to hide their malicious creation and make it last as longer as possible. These strategies cannot fully prevent the process of detection of malware but prolonged it. The longer malware can hide itself from detection, the more time it has for spread and infection. In the following, we review these strategies and their evolution from a simple encryption to modern metamorphism.

2.4.1 Encryption

The earliest method used by malware authors to escape analysis was encryption. Cascade, appeared in 1987, was the first known encrypted malware [53]. An encrypted malware is made of the malware main body (infection, trigger, and payload) encrypted in some way and the decryption loop. When the malware is encrypted, it cannot execute until decrypted. The idea behind this form of obfuscation is that the decryptor loop is small enough compared to the malware body, and only provides a smaller profile for AV software to detect [6, p. 35]. Encryption can be achieved through 1:1 mapping transforming code "byte by byte" such as the substitution cipher, zero-operand instruction such as INC (increment) or NEG (negate) instructions, reversible instruction including ADD or XOR with random keys, encryption keys (used in cryptography), etc.

2.4.2 Stealth

Malware normally modifies data resources on the infected system. A stealth malware is a malware that conceal the infection itself, not just its body. Furthermore, it tries to hide from both the user and AV software [6, p. 37]. When other applications within an infected system request parts of the resources previously modified by the malware, stealth code provides the

unchanged data instead of the viral code [54]. An example of a stealth infection is the Brain virus released in 1986 (the first DOS virus) that infected the boot sector of disks formatted with the DOS File Allocation Table (FAT) file system. Brain redirected all attempts to read an infected sector, supposed to contain boot instructions, to other sectors on the disk where the original uninfected boot sector is located [55].

2.4.3 Packing

Packing or "executable compression" also known as "self-extracting archive" is a technique of compressing and/or encrypting an executable file[12]. Both the packed data with the unpacking routine code are combined into a single packed executable (an archived file) that can be restored to its original executable form when the packed file is loaded into memory [56]. Users wouldn't have to unpack it manually before it could be executed [57]. While packers can be used to make files smaller, to save storage space or reduce data transmission time, by software companies in distributing their products, they are also used by malware authors as a form of malware obfuscation. The packing adds an extra layer of code wrapped around a malware to conceal it. As a result, any archived file must be unpacked for analysis before a piece of malware can be detected. Usually malware authors pack their code multiple times using different packers to bypass detection and thwart analysis. But doing so, they create successive new variants of the same malware [59]. Unless an AV scanner understands which packers are used and how they are applied, it can't track malicious code through the packed file [59]. Among obfuscation techniques, packing is the most used due to the availability of several free and open-source packers. According to [60], among 20, 000 malware samples collected in 2008, more than 80% appear to be using packing techniques from 150 different packing families. Moreover, there is evidence that nearly half of new discovered malware were simply repacked versions of existing ones.

2.4.4 Oligomorphism

Malware authors realized that spotting encrypted malware remains simple for AV software since the decryptor is long enough and doesn't change, so they decided to implement mutated

[12] Based on their behaviors, packers can be classified into four categories: (i) ***Compressors*** that reduce file sizes through compression with little or no anti-unpacking tricks such as Ultimate Packer for Executables (UPX); (ii) ***Crypters*** that encrypt the original file contents without any compression such as Yoda's Crypter; (iii) ***Protectors*** combine features from both compressors and crypters such as Armadillo; (iv) ***Bundlers pack*** a software package of several executable and other data files into a single bundled executable file, which unpacks files within the package without extracting them to disk such as PEBundle [58].

decryptor loop's code that change with each infection. The simplest technique to do so is the use of several decryptors instead of a single one. An oligomorphic or semi-polymorphic malware is an encrypted malware which has a predefined set of different decryptors. The malware randomly picks a new decryptor for each new infection. For example, Whale had 30 different decryptor variants [61], and W95/Memorial 96 decryptors [62]. In terms of detection, oligomorphism only makes a malware slightly more difficult to catch. AV software will simply have to list all possible decryptors, and look for them all [6, p. 38].

2.4.5 Polymorphism

To overcome the limitation of oligomorphism, malware authors create polymorphic malware that would generate an infinite number of decryptor loop variations, thus making analysis more difficult. Examples include 1260 or V2PX virus developed by Mark Washburn in 1990 as part of a research project aiming to demonstrate the limitation of AV scanner at that time (considered as the first polymorphic malware) [63], Tremor virus that has almost six billion possible decryptors! [64], and the Virlock ransomware discovered in 2014 (the first instance of polymorphic ransomware) [65]. Clearly polymorphic malware can't be detected by enumerating all the possible decryptors, but how can the polymorphic malware detect previous infection? and how does change its decryptor for each new infection? When the malware morphs, it must be able to recognize infection by any of its mutating form. This couldn’t be possible, only if the infection detection mechanism is totally independent of the malware code. Many solutions for overcoming this hurdle exist, such as using a file timestamp[13], a file size[14], filesystem features[15], hiding a flag in unused areas of an executable file, or using external storage. Regarding the decryptors changing problem, it can be resolved by using the mutation engine, or MtE (written by the Bulgarian Dark Avenger in 1991). It turns out to be a toolkit, which can permute code in a large number of ways, and therefore converts an ordinary malware to a polymorphic one by supplying a new decryption routine with an encrypted malware body at each request [63].

[13] A malware could modify the timestamp of an infected file, so that the sum of its time and date represents a constant value K for all infections. Since many software programs only display the last two digits of the year, an infected file's year could be increased by 100 without drawing attention [6, p. 39].

[14] The size of an infected file can be increased to some meaningful size, such as a multiple of 1234 [6, p. 39].

[15] Some filesystems (such as NTFS) allow files to be tagged with arbitrary attributes that can be used by a malicious program to store code, data, or flags which indicate that a file has been infected [6, p. 39].

2.4.6 Metamorphism

A metamorphic malware is a malware that has the ability to change its internal structure without altering its functionality. Igor Muttik defined a metamorphic malware as a body-polymorphic [63]. It does not have a decryptor loop (because it isn't encrypted) or a constant malware body but is able to create a new generation of malware that will look completely different. In contrast to a polymorphic malware that doesn't change its MtE after each infection because it can't reside in the encrypted part (a malware body), a metamorphic malware completely re-writes its code including its MtE, so that each newly infection of itself no longer matches its previous iteration. [6, p. 46]. Notable examples of metamorphic malware include the Win32/Apparition, Win32/Evol, Win95/Zmist and {Win32, Linux}/Simile. Metamorphism is relatively simple to implement in malware that spread as source code, like macro viruses. Win32/Apparition virus, written in Pascal, carries its source code and drops it whenever a compiler is found on a system being infected, the virus' mutation engine undoes old obfuscations, applies new transformations, in a way that will look completely different, and recompiles itself.

2.5 Malware obfuscation techniques

The obfuscation is a programming technique that makes code intentionally obscured (to prevent reverse engineering or to hinder analysis using heuristics) and harder to understand (deliver unclear code to anyone who try to break it) [60]. It consists of converting code, by applying several transformations, to a new different version in appearance but functionally equal to the original one (before being obfuscated). Originally, this technique intended to protect the intellectual property of software developers, but since it is extensively used by malware authors in changing their code signature continually to evade detection by AV scanners, it is therefore important to analyze how it is applied to malware. In what follow, we give the most commonly obfuscation techniques used by polymorphic and metamorphic malware. They mutate the code syntactically but keep the semantics of code same.

- *Garbage code insertion:* Garbage code insertion is a rudimentary technique that injects some ineffective instructions over a program to change its binary sequence appearance while keeping its behavior [67]. This will make the program's control-flow more complex and more difficult to understand. Various types of garbage codes come in the form of instructions that are not executed such as NOP (no operation–do nothing) values, instructions that have no effect on program outcomes such as "ADD EAX, 0", or those

located in areas of the program that will never be executed (refer to as dead-code). The Win95/Zperm family of viruses are examples of malware that create new variants by removal and insertion of garbage instructions [62].

- *Register reassignment:* Register reassignment is another simple technique that exchange registers (for example, by switching EAX, EBX and EDX registers with EBX, EDX and EAX registers) or memory variables in different instances of a malware, which modifies the binary sequence of the code without altering its behavior [67]. The W95/Regswap virus is a good malware example that uses the same code but different registers in each new generation [62].

- *Subroutine reordering:* In this technique, the original code of the malware is obfuscated by changing the order of its subroutines in a randomized manner [67]. This reordering is irrelevant and does not affect the execution of the malware. Such technique can generate n! subroutine permutations, where n is the number of subroutines. For example, Win32/Ghost malware had 10 subroutines, leading to 10! or 3628800 different malware variants [62].

- *Instruction substitution:* In this technique, the original code evolves by replacing some instructions with other semantically equivalent ones [68]. For instance, MOV can be replaced with PUSH/POP. For example,

```
MOV EAX, 4
  Can be replaced by the equivalent instructions sequence:
PUSH 4
POP EAX
```

  This technique can greatly change the signature of the code and makes it harder to de-obfuscate especially when implemented via a library of equivalent instructions that made unavailable [69].

- *Code transposition:* Code transposition or code permutation reorders the sequence of the instructions of the original code while preserving its behavior [70]. There are two methods to deploy this technique. The first one shuffles the instructions in a random manner, and then recovers the original execution order by inserting the unconditional branches or jumps. The second method creates new variants by reordering the independent or free instructions that have no impact on one another. Since it is very hard to find this kind of instructions, this method is very difficult to implement, but can increase the cost of detection.

- *Code integration:* One of the most advanced technique to obfuscate code introduced by the Win95/Zmist malware, which binds itself to the code of the program to infect [68]. First, the Win95/Zmist decompile its target program into small manageable objects, seamlessly inserts itself between them, and then reassembles the integrated code (the original program and the malware) into a single executable.

2.6 Malware types

Malicious software can be classified different ways in order to distinguish the types of malware from each other. This allows for understanding their threat level, the way they infect and how to defeat them. They can be classified according to:

- The malware specific behavior or its mode of operation.
- The malware attack vectors or the method used to deliver its malicious payload.
- The type of vulnerability in an operating system (OS) or application that can leave it open to malware exploits.
- The malware specific objective. As an example, a Spyware aims to steal personal or organizational information.
- The targeted platform or device, such as a specific OS or attacks that target a mobile device.
- The malware stealth techniques or how malware hides itself. As an example, a Rootkit that hides its existence (and the existence of other malicious programs) by replacing legitimate OS components with malicious equivalent ones.

In 2006, Aycock classified malware into ten (10) different types according to their method of operation [6, p. 11-21] as shown in Fig.2.1. Three (03) characteristics are associated with each malware type: *(i) Self-replicating* or the way it propagates, *(ii) Population growth* that describes the change in the number of malware instances due to replication and *(iii) Parasitic* a technique whereby malware is inserted into existing files “executable files” on a system in order to exist. Also, according to Aycock a logic bomb is a type of malicious code that can be standalone or inserted into existing code, with a payload that is executed using a pre-defined date as a trigger. A logic bomb has the following characteristics: possibly parasitic, doesn’t replicate and has no population growth.

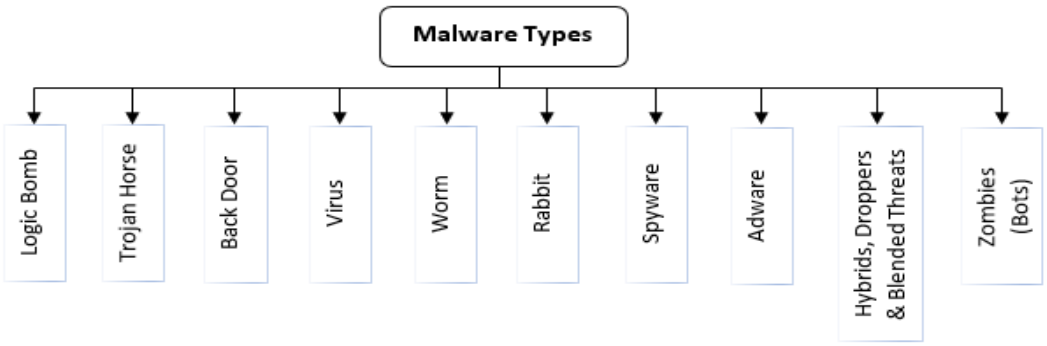


Figure 2.1: Aycock's classification of malware [6]

In 2010, Filiol in his research has taken up the Adleman's classification of malware in which malicious programs are classified into two (02) main classes "*Simple*" and "*Self-Reproducing*" based on their specific objective [39] as shown in Fig. 2.2. Also, a malware in Trojan's class is a malicious program masqueraded as harmless to lure a user to use it and inadvertently runs the Trojan. Malware of this class has generally as an objective the steal of sensitive data. Also, Spyware, password-grabbing login programs, keyloggers, etc., are particular instances of Trojans.

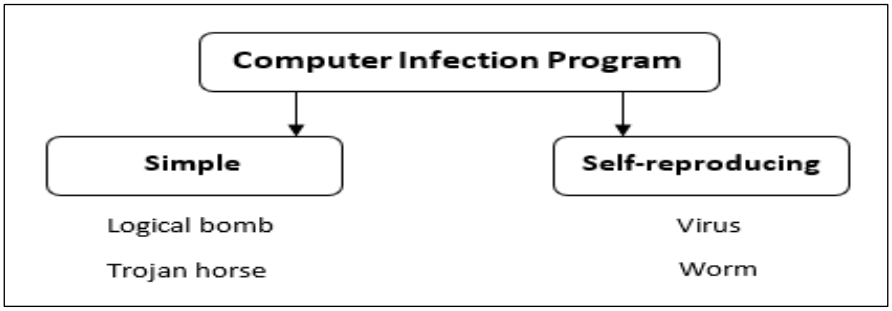


Figure 2.2: Adleman's classification of malware [39]

To better understand the malware threats landscape, Microsoft researchers have developed a taxonomy for classifying malicious software according to their methods of propagation as shown in Fig.2.3. This classification method was published in Microsoft Security Intelligence Report of 2011 "*Zeroing in on malware propagation methods*" [71]. To apply this taxonomy, they analyzed infections reported by the Microsoft Malicious Software Removal Tool (MSRT), running over 600 million Windows PCs worldwide, and specifically targeting malware families with relatively high risk for the user.

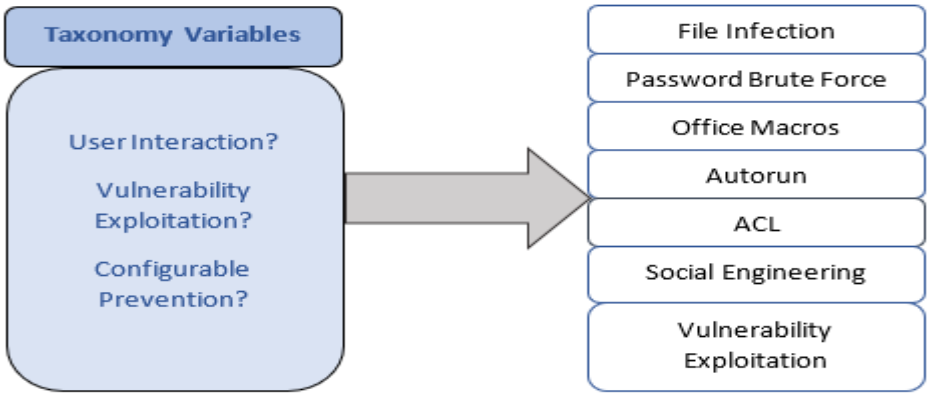


Figure 2.3: Microsoft malware classification [71]

Malware families use different mechanisms to spread. When a malware is discovered on a machine, it is very hard to determine its actual propagation mechanism used to get into the system. To overcome the difficulty in determining the exact propagation method used to spread the malware infection, an "equal buckets" approach was used in which the detections of these families were evenly distributed between each category in which they were known to spread. For example, Win32/Conficker worm propagates by either exploiting a CVE[16]-2008-4250 vulnerability, by taking advantage of AutoRun on mapped and removable drives, or by using a password dictionary. Using this approach, 100 Conficker infections will be translated into 25 vulnerability-related propagations, 25 into AutoRun USB, 25 into AutoRun network, and 25 into password brute force activity. Families that spread via exploits were classified according to the time elapsed since the release of the patch fixing the vulnerability by the software vendor (the age of the security update closing the exploit) at the time of analysis. We distinguish between:

- *Zero-day*. If the exploit is discovered at any time before the release of the MSRT version that detected it, it is considered a zero-day exploit.
- *Update Available*. The security update related to the vulnerability was first released less than a year before the recorded detection.
- *Update Long Available*. The security update related to the vulnerability was first released more than a year before the recorded detection.

[16] CVE, short for Common Vulnerabilities and Exposures, is a list of publicly disclosed computer security flaws. Each entry of the list contains an identification number, a description, and at least one public reference—for publicly known cybersecurity vulnerabilities. When someone refers to a CVE, they usually mean the CVE ID number assigned to a security flaw.

For example, the vulnerability exploited by Conficker worm, was issued by Microsoft (security bulletin MS08-067) in October 2008, so it will be listed in the "Update Long Available" category (when referring to the year 2011).

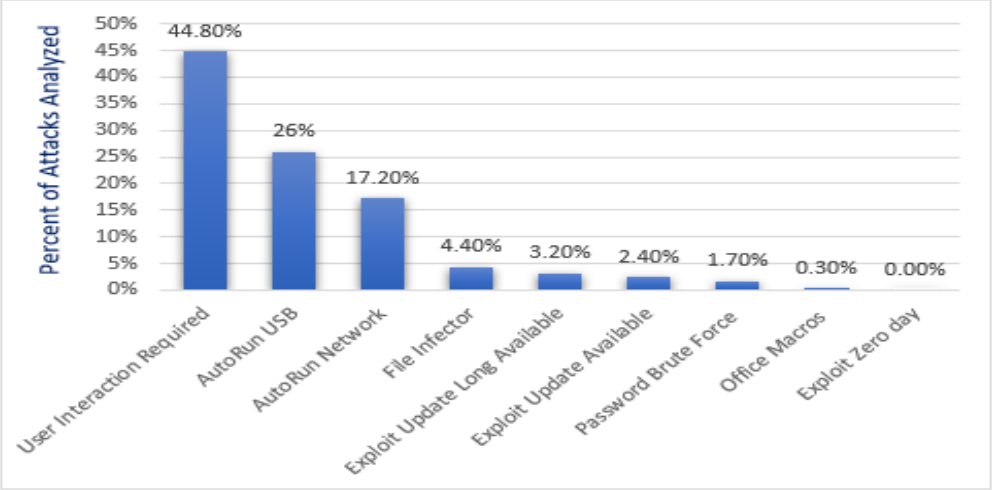


Figure 2.4: Malware detected by the MSRT by means of propagation ability [71]

Figure 2.4 shows the results of the analysis:

- Infections relying on user interaction to spread account for 45% of the total analyzed.
- Infections caused by malicious software that misused USB AutoRun feature accounted for 26 % and those misused network AutoRun accounted for 17%.
- 5.6% of the MSRT detections analyzed were caused by exploits (3.2% are classified as "Update Long Available", 2.4% are classified as "Update Available", and no infection was caused by exploit classified in "Zero-day" category).
- Viruses, or file infectors, accounted for 4 % of detections.
- The password brute force (accounted for 2%) and Office macro (0.3%) infections of the total, were each identified in just one of the malware families examined.

Alternatively, Kaspersky Lab classifies malware or unwanted objects according to their activity on the target victim. Each object is given a clear description and a specific location in the classification tree diagram. Objects whose behavior possessing the least threat are placed in the lower area of the diagram, and those having a greater threat behavior are placed in the upper part as shown in Fig.2.5.

Since individual malware items often have several malicious functions and propagation modes, additional classification rules are required to avoid any possible location confusion in the classification diagram. Also, a malware item that spread via an email attachment or as files

using P2P (peer-to-peer) networks and has the ability to harvest email addresses from a host victim, can be classified either as an Email-Worm, a P2P-Worm, or a Trojan-Mailfinder. To avoid any confusion about classifying this malware item, the following rules are applied: *(i) a threat level is assigned to each behavior*, and *(ii) risky behaviors outrank behaviors with a relatively low risk.* So, in a given example, the malware item will be classified as an Email-Worm, since the P2P-Worm or Trojan-Mailfinder behaviors are less risky than the Email-Worm behavior.

In case of a malware item that has multiple functions with equal threat levels such as Trojan-ArcBomb, Trojan-Notifier, Trojan-Proxy, Trojan-Clicker, Trojan-SMS, Trojan-Mailfinder, Trojan-Banker, Trojan-IM, Trojan-GameThief, Trojan-Ransom, Trojan-DDoS, Trojan-Spy, Trojan-PSW, Trojan-Dropper, or Trojan-Downloader – the malware item will be simply classified as a Trojan. In the same way, a malware item such as IM-Worm, P2P-Worm, or IRC-Worm – will be classified as a Worm [72].

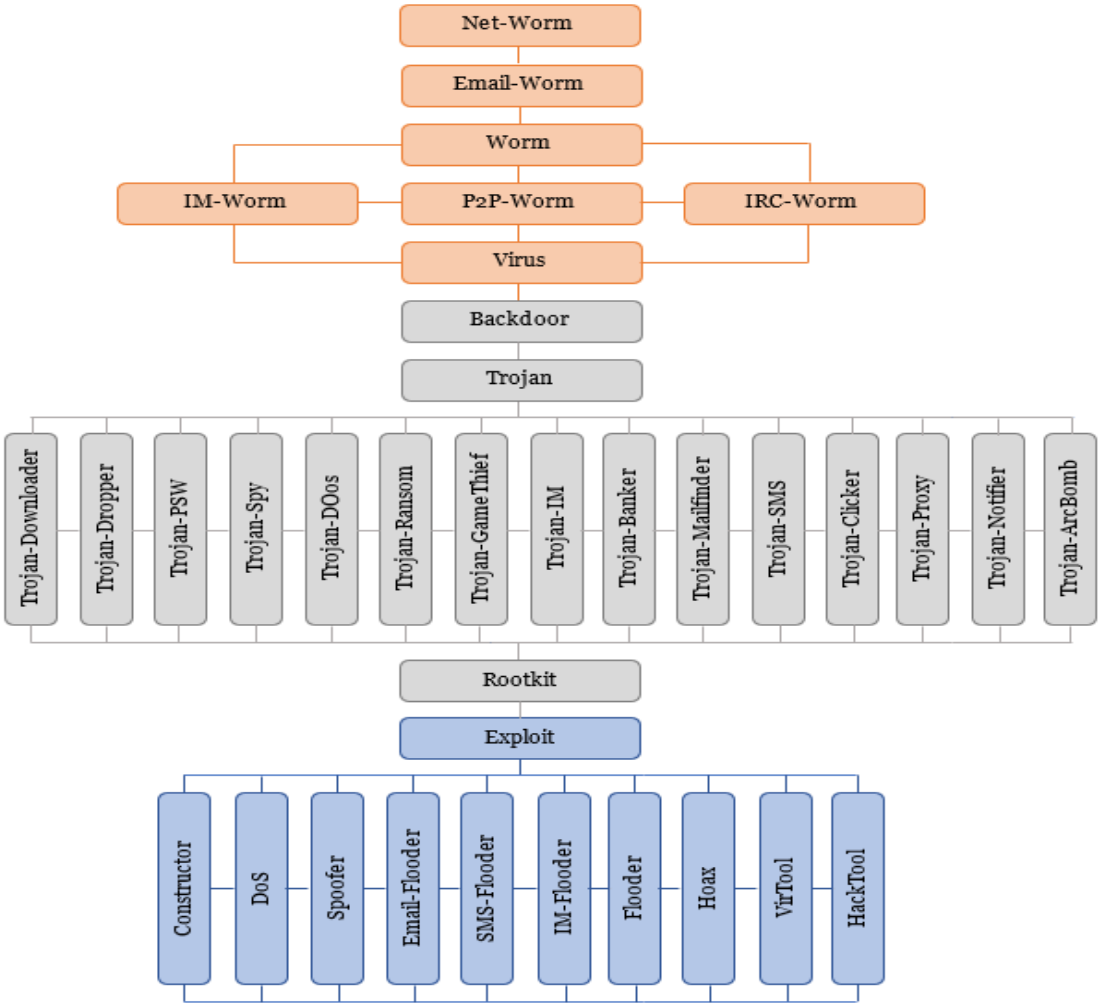


Figure 2.5: Kaspersky Lab malware classification diagram [72]

2.7 Malware analysis

Malware analysis is essentially concerned with the study of malware's behavior [73]. It refers to the process by which the purpose, functionality, origin, type and potential impact of a given malware are determined. The analysis' objective is to understand the working of malware and how can be detected and removed. It involves analyzing the infected file to determine identifiable patterns that can be used for the purpose of generating a signature of the malware, further utilized by security applications such as AV software to detect it. The method by which malware analysis is performed generally falls under one of the following three types:

2.7.1 Static analysis

This is the process of analyzing and dissecting the different resources of the malware binary file without executing it. All possible execution paths of the program are explored. The extracted information collected from analysis can reveal simple characteristics of the malware such as its file type, strings embedded within a file (referencing filenames, URLs or domain names controlled by the attacker) and the functions and libraries being called by the executable, or most complicated characteristics such as identification of the maliciousness based on unencrypted code or strings found in the malware (fingerprinting the malware) [74]. Static analysis is easy and not risky. It is easy since no special conditions are required to conduct analysis, the piece of malware is simply subjected to different static analysis tools. It is without risk of an infection occurring because the malware is not running during the analysis [74].

Dynamic analysis

Dynamic analysis, also known as behavioral analysis, is the process of executing the suspected binary file in an isolated environment and observing its behavior while it is actually running [73]. Dynamic malware analysis provides a more in-depth view into the malware's functions compared to static analysis. Usually, a sandbox environment[17] is used as an isolated test environment to prevent the malware from escaping and actually infecting the system, such environments are virtual systems that can be rolled back to their initial clean state after the analysis is complete [75]. Changes triggered by the malware's execution, such as

[17] A security mechanism for separating running programs that provides a fully implemented malware test environment with the appropriate dynamic analysis tools where the analyst can run and observe a malware's behavior.

modifications in the file system or configuration files, are generally monitored and recorded during analysis [74].

2.7.2 Memory analysis (memory forensics)

This is the technique of analyzing the memory image (analyzing volatile data[18] in a computer's memory dump[19]) taken from the running computer to extract artifacts relevant to the malicious program before the infected system is powered off [73]. It is typically a forensic technique to investigate and identify malware attacks that do not leave detectable tracks on hard drive data but very few signs of infection elsewhere such as fileless[20] malware [76]. Memory forensics is especially useful to track stealth and evasive capabilities of malicious software.

2.8 Malware detection

While malware analysis is about studying the behavior of the malicious code, malware detection is about detecting its presence in the system or distinguishing whether a specific program is malicious or not [77]. It is the process of scanning the computer and files to detect the malware that is present in the system [78]. It involves using signatures generated during the analysis process and focuses on methods that detect certain malware' classes. Both, malware analysis and malware detection are essential to protect the system.

Detecting malware programs by their behavior or appearance is proved that is undecidable [79]; there is no such algorithm that can detect all malware forms. However, by tuning existing algorithms or combining different scanning techniques, a good detection rate can be achieved. To detect malware, AV products use signature-based detection and heuristics.

Detection is the most important task that an AV does up in comparison to other tasks: *identification* (which form of malware is it?) and *disinfection* or *cleaning*. It can be performed using generic methods that deduce the presence of a malicious code from environmental anomalies (both known and unknown malware can be detected) or using malware-specific methods, which only work with known malware [6, p. 53].

---

[18] the data stored in temporary memory (RAM) on a computer while it is running. When the computer is turned off, volatile data is lost immediately.

[19] A memory dump or a core dump is a snapshot capture of computer memory data from a specific instant.

[20] Fileless malware, also known as memory-resident malware, is a type of malicious program that writes itself directly onto a computer's system memory.

To detect malware, AV products use *signature detection* for exact matches and *heuristics*[21] for extrapolated detection. Heuristics is a generic term for various techniques like emulation, file anomalies, API hooking, sandboxing and data mining, used to detect unknown or complex malware instances by their behavior, for which signatures are not yet available.

AV software is expected to detect all infected files on a system. Unfortunately, this is not the case. Detection isn't perfect and can result in false alarms (or false positives FPs). A false positive[22] is when an AV software examines a legitimate file or a website and reports a malware even though a malware isn't really there, which can damage the file by removing useful code when disinfecting, or even crash the machine when quarantining or deleting system files. This can occur due to a faulty signature or after improper disinfection, refers to as ghost positive, where a malware is detected that is no longer there by the same or other AV engines [80]. False positives are one of the most important measurement for AV quality.

2.8.1 Signature-based detection

Signature-based detection is a basic technique to protect against malware and still the most widely used today, and an important viable component of anti-malware defences that AV companies continue to improve. In this technique, AV engine (the scanner) compares input blocks of a file under inspection against the signatures (or patterns) of all known malware. These signatures are stored in the program's malware definition database, which is regularly updated. If the file matches one signature, then the scanner assumes that this file is infected. Otherwise, it is considered safe. Signature-based detection scanning is similar to finding the offending bytes in the file being inspected.

The signature for an executable malware typically is a series of machine code bytes, usually known as "byte-streams" or "strings". They are created by examining the disassembled code of a malware binary. Sequence of bytes are extracted from the suspicious section of the disassembled code, and used in building the signature of the malware. A typical signature looks like as shown in Fig.2.6. This kind of signatures can be efficiently matched using string processing algorithms presented in the next chapter. Other kind of signatures exist, they can be checksums (modified versions of the CRC32 algorithm are generally used), cryptographic

[21] The terms "heuristics-based" and "behavioral detection" are often used interchangeably.

[22] In contrast, a false negative is when the AV software examines an infected file and doesn't detect a malware that's present.

hashes (cryptographic hash functions such as MD5, SHA1, and SHA2), fuzzy hashing (fuzzy hash algorithms are applied) and graph-based hashes for executable files [4]. For the purpose of our study, we focus on byte-stream signatures since they require a significant time to be processed by the virus scanner despite their minority in today AV database signatures. Experiments on the ClamAv[23] databases [80, 81], show that 95% of the total scanning time was spent matching basic string and regex[24] signature types, which only account for less than 10% of the total signatures.

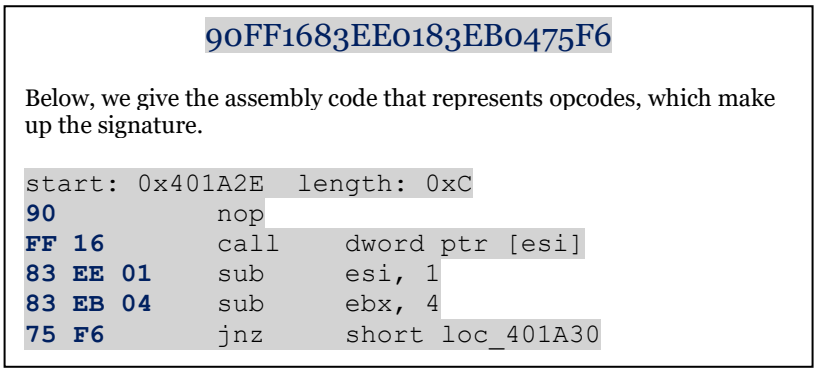


Figure 2.6: Hexadecimal representation of a typical malware signature and the corresponding x86 code snippet

In the early days of AV software, malware signatures numbered in the hundreds. Scanning a file was fairly quick. Now, there is an astonished number[25] of known viruses, worms, Trojans, and other malware types that AV software struggle to detect by keeping scanning performance within acceptable limits. To keep a handle on performance, AV engines classified signatures by the type of infection they represent- boot sector, executable file, macros, scripts, etc. When scanning a particular file, only the signatures that pertain to that file type are matched. For example, a script signature would not be used to scan an executable file. Next, certain rules are applied to avoid scanning the complete file. Depending on the type of the file being scanned– .exe, .doc, or .pdf, the scanning process is limited to areas (the beginning, the end, or some specific locations) in the file that are more likely to contain a malicious code [83].

[23] ClamAntiVirus (or ClamAV) is one of the most widely used open-source anti-malware toolkits available. Initially developed for UNIX systems but latter a Windows version was produced.

[24] Regex signature type is an extension of the basic string signature type with various wildcards, written in a custom language with regular expression-like syntax.

[25] The ClamAV malware database is updated at least every four hours and as of 10 February 2017 contained over 5,760,000 malware signatures (source: https://en.wikipedia.org/wiki/Clam_AntiVirus#Mac_OS_X).

While it is more effective to identify a specific malware, it can be quicker to detect a malware family through a generic signature, like the one shown in Fig.2.7. Many viruses, worms and Trojans start as a single infection, but grow into dozens of slightly different strains, through mutation or modifications by other malware writers. Instead of creating a separate signature for every malware variant, malware researchers find locations that all malware in a family share, and create a single generic signature. These signatures are created using wildcards[26] since they contain fragments of unique code from a number of different locations in the compromised file. Using wildcards allow the scanner to detect if a malicious code is padded with other junk code [83].

FF1890EF16C00F9EB04{-3}683EE0183EB04F6C9A00??7A

{-3} is a wildcard for 0 to 3 arbitrary bytes
? is a wildcard for one arbitrary byte

Figure 2.7: Hexadecimal representation of a generic malware signature

String scanning, wildcards and mismatches methods, as described in [63] and briefly explained below are the most popular methods used by AV software to detect malware based on their string signatures. For example, the ClamAV scanner defines a simple format for representing signatures including disjunctions (aa|bb), gaps {n-m}, wildcards'?' and character classes [84] as shown in Fig.2.8. In addition to generic detection other methods, which involve scanning for pre-defined sequences of bytes, such as hashing, bookmarks, top-and-tail scanning, entry-point and fixed-point scanning are also applied to speed up the scanning process.

(8a|86)0666(29|31)(c8|d0|d8|e8|f8)(86|88)0646

Below, we give the corresponding x86 code snipet .

```
8a 06      ; mov al, byte ptr [esi]
66 31 e8   ; mov ax, bp
88 06      ; mov byte ptr [esi], al
46         ; inc esi
```

Figure 2.8: ClamAV signature for the Virut malware [84]

- ***String scanning:*** The AV scanner searches for a sequence of bytes that is typical to a particular malware instance, extracted for each different malicious program and

[26] A wildcard is a character that may be substituted for zero or more characters in a string. Common wildcard characters include ***** - to match any number of characters (or bytes), **?** - to match a single character, and **%** - to match zero or more characters.

cataloged in an AV signatures database. The following example is a signature string of the W32/Beast virus in EXE files, published in [85]: 83EB 0274 1683 EBOE 740A 81EB 0301 0000.

- ***Wildcards:*** The scanner is allowed to skip one or more bytes and some regular expressions. Some early generation encrypted, polymorphic, and even metamorphic malware can be spotted using this method. In the following example the bytes represented by the character '?' are skipped. The wildcard %3 means that the scanner will try to match the next byte (03 in the example below) in any of the three positions that follow it 83EB 0274 ??83 EBO? 740A 81EB %3 0301 0000.
- ***Mismatches:*** Used to create generic signatures for a family of malware. They allow any given number of bytes in a string to be of arbitrary value, regardless of their position. For example, the 0F 02 03 04 0A 07 08 FF string with the mismatch value of 2 would match the following pattern 0F E8 03 04 0A 07 9A FF.

Signature detection based on string signatures is challenged by the increasing number of malicious programs since an AV engine uses a separate signature for each malware variant, and vulnerable to evasion techniques (such as obfuscating internal data, environmental awareness, timing-based evasion, confusing automated tools, etc.) since the signatures are derived from known malware. Its major flaw is its inability to detect new malware and zero-day attacks. Its strength relies on the assumption that the behavior of one instance of malicious software is quite similar of all instances[27] [86]. This detection technique still the most commonly used method of identifying malware and has many advantages which include:

- Does not require the malware sample to run to detect it;
- Less number of resources is required for malware detection;
- During the scanning process, false alarms are almost impossible because the signatures are mostly unique [81];
- It provides exact nature of infection, which other techniques like integrity checker[28] cannot provide [87];
- It is simple to update a signature database with new ones whenever new malware is discovered [82];

[27] For worms, this means that by studying one node of the worm, the behavior of all nodes that are compromised by the worm can be reliably predicted.

[28] A technique that looks up for signatures based on checksums and cryptographic hashes.

- Its scanning speed can further be increased by using other detection algorithms or optimizing the implementation of the existing ones (Tuning an algorithm's implementation). For example, by exploiting optimization capabilities offered by new microprocessors like vectorization.

2.8.2 Heuristic-based detection

Heuristics is a generic term for various techniques like emulation, sandboxing, data mining, API hooking and file anomalies, used to detect complex malware. Heuristic scanner inspects the code statically and dynamically to determine the *probabilities* of a file infection.

Static heuristic methods are based on the examination of a malicious file looking for sequences of instructions, which are generally indicative of suspicious code that differentiate the malware from safe files like junk code, decryption loops, use of undocumented API calls, etc. The heuristic data collected from this examination are then analyzed by weighting each heuristic's value. If the result obtained by summing these values exceeds a predefined threshold, then the file is considered to be *probably* infected.

Dynamic heuristics are almost the same as static heuristics with the difference that the data, about the code being analyzed, are collected from the emulator[29]. In this technique, AV engine inspects a running program's behavior looking for suspicious actions. It may look for the program trying to write directly to the disk, or a program eavesdropping over a network connection using TCP/IP ports, etc. These dynamic methods incur a run-time overhead for monitoring that is not imposed by static methods [6, p. 80].

Heuristics are generally used to spot new malware threats before signature updates become available.

[29] A safe virtual environment to monitor running code.

Chapter 3

3 Pattern-matching algorithms

In computer science, pattern-matching occurs as part of data processing (such as text data mining, web search, bibliographic search programs, or searching and substitution commands within text editors), digital forensics (digital signatures demonstrating the authenticity of a digital message or document), full text search (utility programs such as *grep*, *awk* and *sed* of Unix systems), speech and image recognition, information retrieval (biological sequence analysis and gene identification), plagiarism, and anti-malware solutions (intrusion detection systems IDS, virus scanners, web and spam filters, etc.). Figure 3.1 shows different application fields using a pattern-matching concept. All these application fields usually process a large quantity of textual data, and require pattern-matching capabilities at very high speeds with less memory usage.

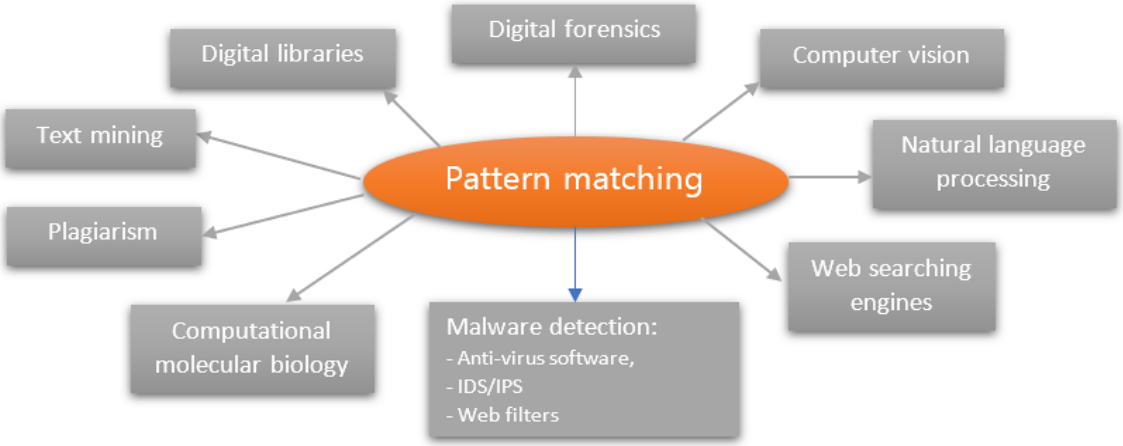


Figure 3.1: Application fields using pattern-matching

3.1 Pattern-matching problem

Pattern-matching, sometimes also referred to as string-matching, or string-searching is the concept of finding a sequence of characters, often called patterns, inside a provided text (or binary data). The matching process includes the location of these patterns inside the text. Formally, pattern-matching is the problem of locating all occurrences of a pattern P of length m in a text T of length n. Both the pattern and the searched text are vectors of elements of a

finite set called an alphabet Σ[30]. We say that pattern P occurs with valid shift s in text T if $0 <= s <= n-m$ and $T[s + 1..s + m] = P[1..m]$.The pattern-matching is therefore the problem of finding all valid shifts with which a pattern P occurs inside a text T [88].

Pattern-matching algorithmic complexity is usually analyzed by the running time of the algorithm and the memory space required by the computations. To this, we can add a setup time, or a substantial amount of computation before the algorithm can begin searching (time spent during the pre-processing phase), and the need for backtracking or not (since moving back and forth through the search text can entail some form of buffering if the search text doesn't exist in memory, but sent to the program as a stream of data) [89, p. 96-97]. Considering the running time feature, all pattern-matching algorithms perform on the order of $a + bn$, where a is the pre-processing time, b is a constant indicating the number of comparisons made for each character, and n is the number of characters in the text being searched. The effectiveness of the algorithms is measured according to their ability to lower the value of b. Effective ones perform in less than one comparison per character searched ($b < 1$), termed as sub-linear pattern-matching algorithms [89, p. 96-97].

Algorithmic complexity is concerned about how fast a particular algorithm performs. Specifically, we want to determine the amount of time taken by an algorithm to execute on some inputs without depending on its implementation. Of course, the algorithm will take different amounts of time on the same inputs that depend on the processor, memory or disk speed, the machine instruction set, the compiler, etc. In this context, the *computational complexity* is introduced. It refers to the *worst-case time* that an algorithm can take in order to solve a given problem [90] i.e., the execution instance that causes maximum number of operations to be executed. Both *average-case* and *best-case times* provide valuable information of the algorithmic performance related to specific cases but usually we are most interested in the *worst-case* in analyzing an algorithm because it defines clearly its running time upper bound which is good information.

Many algorithms to solve pattern-matching problem exist. They are classified either as *single* or *multiple* pattern algorithms based on the number of patterns to look for. For k patterns to be matched, the single-pattern matching algorithm must be repeated k times. In contrast, the multiple pattern-matching algorithm need only to be executed once. Applications that relay on

[30] The alphabet Σ may be a usual human alphabet (for example, the letters A through Z in the Latin alphabet). Other applications may use a binary alphabet (Σ = {0,1}) or a DNA alphabet (Σ = A,C,G,T}).

this class of algorithms may require *exact* (case of most applications) or *approximate* pattern-matching[31]. The performance of security applications heavily relies on signature scanning and require exact pattern-matching capabilities at very high speeds with less memory consumption.

3.2 Exact pattern-matching algorithms

The intuitive method of solving the exact pattern-matching problem is to compare the pattern to every position in the text and check for a match. This method was presented by the naïve, or brute force string search algorithm, and is considered to be the first algorithm [3.3]. Later, large number of algorithms have been developed to solve single exact string-matching problem including Karp-Rabin algorithm, the Knuth-Morris-Pratt (KMP) algorithm, and several variants of the Boyer-Moore (BM) algorithm like Turbo BM, Reverse Colussi, Boyer-Moore-Horspool (BMH), Quick search, etc. In real world applications such as bioinformatics, text mining, and malware detection there may be more patterns to deal with simultaneously. The multiple pattern-matching algorithms solve the problem of finding more patterns concurrently. Many solutions for exact string-matching of multiple patterns is known to exist. However, most of them have been designed for moderately sized pattern sets, and therefore they do not fit well with larger sets of patterns. The most commonly used solutions are Aho-Corasick, Wu-Manber and Commentz-Walter algorithms.

3.3 Brute force algorithm

In the naive, brute force algorithm we check for all positions, between 0 and n-m, in the text at which the pattern could match. The algorithm requires no pre-processing phase, and the scanning process can be performed in any order (but usually, it is carried out from left to right) one character at a time. Formally, the pattern P is compared against the text factor[32] $T[0..m]$, then against $T[1..1 + m]$, then against $T[2..2 + m]$ etc., until there are no more matching characters. If P is exhausted (in which case an occurrence of P is found) or a mismatch occurs,

[31] Approximate pattern-matching (or a fuzzy string searching) is the technique of finding strings that match a pattern approximately. In contrast to exact pattern-matching in which a complete pattern is compared with the selected text window, in approximation matching only a small portion of the pattern is considered. The most illustrative application of approximate matchers is spell checking.

[32] The text factor $T[j..j + m]$ currently being compared against the pattern P is called the text window.

the text window is shifted one character to the right, and the matching process is restarted. The corresponding C++ code is given in listing 3.1.

Listing 3.1: C++ implementation of the brute force algorithm

```
// Input: Text = T[0 ... n], Pattern = P[0 ... m]
// The Output_matching_pos function reports positions of all occurrence of P in T

void BF_search(unsigned char* T, unsigned char* P, int n, int m)
{
	int i, j;

	for (i = 0; i <= n - m; i++)
	{
		for (j = 0; j<m; j++)
		 if (T[i + j] != P[j]) break;
		 if (j == m) Output_matching_pos(i);    //pattern found
	}
}
```

The outer loop is called at most n-m+1 times and the inner loop is executed m times for every iteration of the outer loop, so the worst-case time complexity of the brute force algorithm is $m \times (n\text{-}m+1)$, and since m is very small compared to n, the total comparisons is $\sim m \times n$: the overall time complexity of the algorithm is $O(m \times n)$. This can happen, for example, when $P = a^{m-1} b$ and $T = a^n$.

The brute force algorithm is very easy to implement. However, its practical running time may be too slow, especially with larger texts and patterns compared to other algorithms. To reduce the number of character comparisons, different approaches exist. They all start by processing the pattern to gain information on its structure and the occurring characters it contained (pre-processing phase) before starting the process of matching the pattern against the text (searching phase).

3.4 The Boyer-Moore algorithm

In 1977 Robert S. Boyer and J Strother Moore published their paper "*A Fast String Searching Algorithm*" [91]. One of the most classic and efficient string-searching algorithms, which is considered as a benchmark for practical string-search literature. Their technique proposed a new approach to solve the pattern-matching problem without examining all the characters in the text. For each alignment of the pattern P with the text T, the algorithm scans for an occurrence of P by processing its characters from right to left rather than from left to right as in the brute force algorithm. In case of a mismatch, the pattern is shifted to the right of the text

based upon two functions: *the bad character shift* and *the good suffix shift*, pre-computed during the pre-processing phase (heuristic approach). Basically, whenever we have a mismatch, we try both functions. Each function will tell us the number of characters in text T that can be skipped, and simply we take the maximum of the two. Both of the above functions can be used independently to search a pattern in a text.

In general, the algorithm runs faster when working on a large alphabet and its speed increases as the pattern length increases. On typical inputs, the Boyer-Moore algorithm makes ~n/m character comparisons to search for a pattern in the text: O(m/n) best performance [92]. However, it is not optimal to use if the pattern to match is very short or has a low probability to be found in the text. The time complexity of the algorithm is O(m×n) during the searching phase, and O(m+σ) in the pre-processing phase [93].

3.4.1 The bad character heuristic

The idea behind a bad character shift is simple. The character of the text T which doesn't match with the current character of the pattern P is called the bad character. Upon mismatch, we shift P until the mismatch becomes a match, or we move P past the mismatching character in T as shown in Fig 3.2.

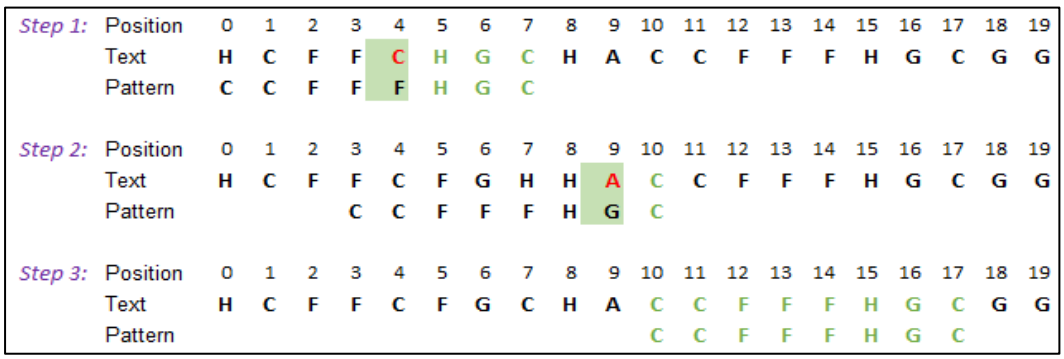


Figure 3.2: The bad character shift heuristic

In the example above (Step 1 - first alignment of the pattern with the text), comparison between "C" and "F" causes a mismatch at position 4 in the text. The bad character "C" occurs in the pattern at positions 0 and 1, the pattern P can be shifted over the text T by 03 positions (or skip 02 alignments) so that the rightmost "C" in the pattern is aligned to text character "C" (aligning in order for a mismatch to become a match). In step 2, the bad character "A" doesn't occur anyway to the left in the pattern. That means, we have to move P all the way past the mismatching character "A" in T.

To implement the bad character shift, we use a table (an array `BadCharacter[]`) that gives, for each character in the text *T*, the index of its rightmost occurrence in the pattern *P* (or -1 if the character does not exist in the pattern). This value tells us exactly how far to shift the pattern to the right (the number of characters to skip) if that character occurs in the text and causes a mismatch during the matching process, as show in in listing 3.2.

Listing 3.2: The BadCharacter table

```
void BC_Heuristic(char* pattern, int* BadCharacter, int M)
{
    int m = M;                          //pattern length

    for (int i = 0; i<MAXCHAR; i++)     //set position to -1 for all characters
       BadCharacter[i] = -1;

    for (int i = 0; i < M; i++)      //rightmost position for character in pattern
       BadCharacter[pattern[i]] = i; //-1 for characters not in pattern
}
```

With the BadCharacter table pre-computed, the implementation of the bad character shift heuristic is straightforward. While the amount of *shift* is less or equal *(n-m)*, the pattern is aligned with the text at position *shift+j,* and the matching process starts. We have an index *shift+j* moving from left to right through the text, and an index *j* moving from right to left through the pattern. For each alignment, we keep decreasing the value of *j* from *(m-1)* down to 0 as long as the pattern character matches the text character (`pat[j]` is equal to `txt[shift + j]`). If the value of *j* is reduced to -1 that means all the pattern characters matched the text characters for this alignment. Otherwise, there is a character mismatch at *shift+j* position. The amount of the *shift* value is updated in both cases accordingly and another alignment begins, as show in in listing 3.3.

Listing 3.3: C++ implementation of the bad character shift heuristic

```
void SearchPattern(char* txt, char* pat, int *array, int N, int M)

{       int n = N;        //txt lenght
        int m = M;        //pat lenght

        BC_Heuristic(pat, BadCharacter, m);       //fill bad character array
        int shift = 0;
        int j;

        while (shift <= (n - m))
        {
                j = m - 1;
                while (j >= 0 && pat[j] == txt[shift + j])
                {
```

```
                j--;     //decreasing j when the pattern character
            }            // matches the text character


            if (j < 0)
            {     //pattern found
                if ((shift + m) <= n)
                    shift += m - BadCharacter[txt[shift + m]];
            }
            else
            {     //pattern not found
                shift += max(1, j - BadCharacter[txt[shift + j]]);
            }
        }
}
```

The bad character heuristic can lead to a shift distance of m (length of the pattern), if the first comparison causes a mismatch and the mismatching text character does not occur anyway in the pattern. The time complexity for the bad character shift algorithm is $O(m/n)$ for best case and $O(m \times n)$ for the worst case.

3.4.2 The good suffix heuristic

The main idea of a good suffix heuristic is to shift the pattern more efficiently when a mismatch occurs by aligning the *overlapping* part of the text and the pattern together. Considering the illustration given in Fig 3.3. A substring of pattern P (the green part) which has been matched with a substring t of text T, is called good suffix. The question is where should the pattern be shifted to find the next possible location for checking when a mismatch occurs? two cases have to be considered.

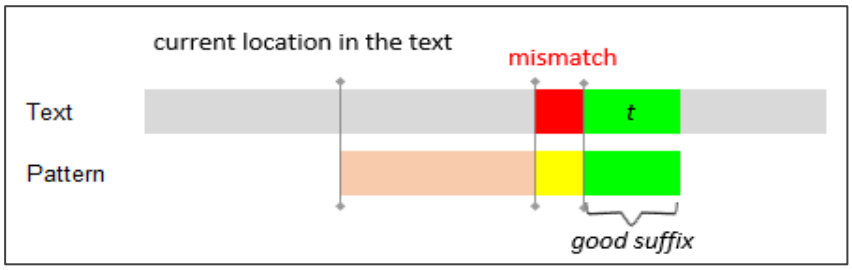


Figure 3.3: The good suffix

- ***Case 1:*** The matching suffix occurs somewhere else in the pattern, as shown in Fig 3.4. In this case, we shift the pattern to the right, over the text, so that the green parts are aligned.

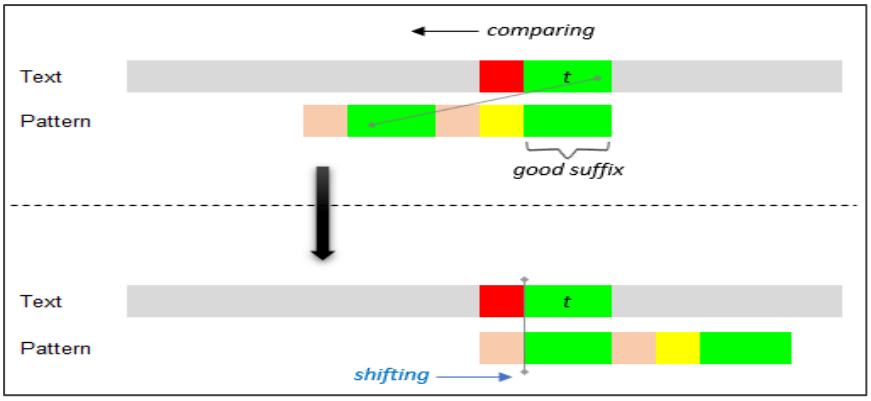


Figure 3.4: The matching suffix occurs somewhere else in the pattern

- ***Case 2:*** Only a partial of the good suffix, the dark green part, occurs at the beginning of the pattern, as shown in Fig 3.5. In this case, we shift the pattern to the right, over the text, so that the dark green parts are aligned.

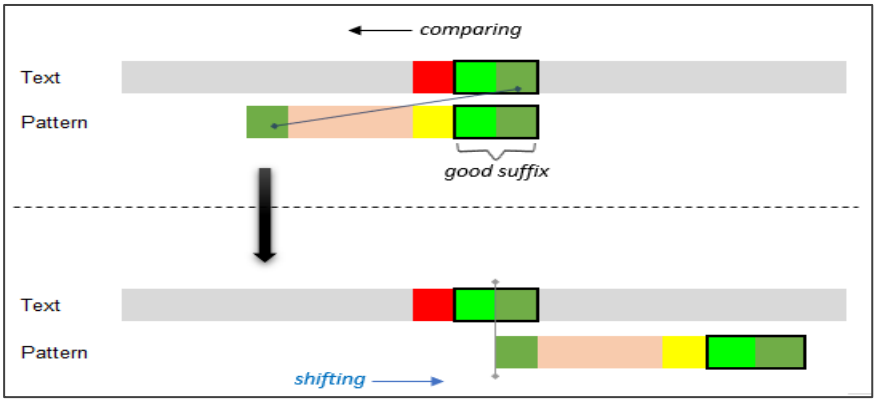


Figure 3.5: A partial of a good suffix occurs as a prefix of the pattern

To implement the good-suffix heuristics, a *shift* table is created during the pre-processing phase that is done separately for each case. Each entry of the table *shift*[*i*] contains the shift distance of the pattern if a mismatch occurs at position $(i - 1)$. That is, the suffix of the pattern starting at position i has matched, and a mismatch occurs at position $(i - 1)$ in the text. Before we go any further in our implementation, let us first discuss the idea of the border of a string.

Definition:

Let $x = x_0 \ldots x_{k-1}$, $k \in \mathbb{N}$ a string of length k over an alphabet Σ.

A prefix of x is a substring u (x starts with u) with

$u = x_0 \ldots x_{b-1}$ where $b \in \{0, \ldots, k\}$

A suffix of x is a substring u (x ends with u) with

$u = x_{k-b} \ldots x_{k-1}$ where $b \in \{0, \ldots, k\}$

A prefix (suffix) u of x is called a proper prefix (suffix), if $u \neq x$. That is, its length b is less than k.

A border of x is a substring r, which is both proper prefix and proper suffix of x, with

$r = x_0 \ldots x_{b-1}$ and $r = x_{k-b} \ldots x_{k-1}$ where $b \in \{0, \ldots, k-1\}$

Its length b is called the width of the border.

Example:

Let x = babab.

The proper prefixes of x are {ε, b, ba, bab, baba}.

The proper suffixes of x are {ε, b, ab, bab, abab}.

The borders of x are {ε, b, bab}, having widths 0, 1 and 3 respectively.

The empty string ε has no border, and is always a border of x, for all $x \in \Sigma^+$.

In the pre-processing phase, the borders of the suffix of the pattern have to be determined since the matching suffix is a border of a suffix of the pattern. Then, during the searching phase, the shift distance will be computed according to the prefix that has matched. Now let's go back to our implementation and consider each case independently, as explained in [94]:

- ***Case 1:*** To determine the borders of the pattern suffixes, we use a *bpos* (border position) table. However, the inverse mapping is needed between a given border and its corresponding shortest suffix in the pattern. Each entry *bpos*[*i*] contains the starting position of the wideset border of the pattern suffix starting at position *i*. The suffix ε at position *m* has no border, therefore *bpos*[*m*] is set to (*m*+1). Similar to the Knuth-Morris-Pratt algorithm [95]. The shift position is determined by the borders which cannot be extended to the left, as shown in the following example:

Positions	0	1	2	3	4	5	6	7
Pattern ***P***	**a**	**b**	**b**	**a**	**b**	**a**	**b**	
bpos	5	6	4	5	6	7	7	8
shift	0	0	0	0	2	0	4	1

  Considering the suffix *babab*, its widest border *bab* starting at position 2, starting at position 4. Therefore, *bpos*[2] = 4.

  The widest border of suffix *ab* starting at position 5 is *ε*, starting at position 7. Therefore, *bpos* [5] is set to (m+1) = 6+1= 7.

The suffix *babab* starting at position 2, has border *bab* starting at position 4 that cannot be extended to the left since ($P[1] = b$) ≠ ($P[3] = a$). Therefore, the shift distance *shift*[4] = the difference 4 – 2 = 2 if *bab* has matched and then a mismatch occurs.

In addition, the suffix *babab* starting at position 2 has another border *b*, starting at position 6. Following the same reasoning, the shift distance *shift*[6] = 6- 2 = 4 in case if *b* has matched and then a mismatch occurs.

Finally, considering the suffix *b* starting at position 6, which has border ε starting at position 7 that cannot be extended to the left. The shift distance *shift*[7] = 7 – 6 = 1 if nothing has matched. That is, a mismatch occurs in the first comparison.

Listing 3.4: Preprocessing for good suffix shift case1

```
// preprocessing for good suffix case1
void preprocess_gsuffix_case1(int *shift, int *bpos, char *pat, int m)
{
  // m is the length of pattern
  int i = m, j = m + 1;
  bpos[i] = j;

  while (i>0)
  {
      //if characters at position i-1 and j-1 are different,
      //then continue searching to right of the pattern for border
      while (j <= m && pat[i - 1] != pat[j - 1])
         {
         //if the character preceding the occurrence of a substring t in
               //pattern P is different than the mismatching character in P,
         //we stop skipping the occurrences and shift the pattern from i to j
               if (shift[j] == 0)
                      shift[j] = j - i;

               //Update the position of next border
               j = bpos[j];
         }
      // if the pattern characters at positions i-1 and j-1 matched ==>
            // border is found, and its starting position is stored in bpos table
          i--;
          j--;
          bpos[i] = j;
         }
}
```

- ***Case 2:*** In this case where only a part of the good suffix occurs at the beginning of the pattern, the latter has to be shifted as far as its widest matching border allows. Therefore, for each suffix the widest border of the pattern that is contained in that suffix has to be determined. The starting position of the widest border of the pattern at all is stored in *bpos*[*0*].

Positions	0	1	2	3	4	5	6	7
Pattern *P*	a	b	b	a	b	a	b	
bpos	5	6	4	5	6	7	7	8
shift	5	5	5	5	2	5	4	1

In the above example, the value of *bpos[0]* equal to 5 since the border *ab* starts at that position. Initially, this value is stored in all free entries of the table *shift*. But when the suffix of the pattern becomes shorter than this value, the algorithm continues with the next wider border of the pattern, as shown in listing 3.5.

Listing 3.5: Preprocessing for good suffix shift case 2

```
// preprocessing for good suffix case2
void preprocess_gsuffix_case2(int *shift, int *bpos, char *pat, int m)
{
int i, j;
j = bpos[0];
	for (i = 0; i <= m; i++)
	{
		// j = bpos[0] is stored in all free entries of the shift table
		if (shift[i] == 0)
			shift[i] = j;

		//if the suffix becomes shorter than bpos[0],
		//the value of j is set to the position of next widest border
		if (i == j)
			j = bpos[j];
	}
}
```

The good suffix heuristic can lead to a shift distance of *m* (length of the pattern), if only the first comparison was a match, and the matched character does not occur elsewhere in the pattern. Due to its difficulty in understanding and implementing, some versions of the Boyer-Moore algorithm, such as the Horspool algorithm [96] or the Sunday algorithm [97], are implemented without the good suffix heuristic. They claimed that the good suffix would not save many comparisons. However, this is not the case for small alphabets [94].

3.5 The Aho-Corasick algorithm

Suppose we need to search a large medical database for any and all references to chest pain, chronic bronchitis, or pneumonia at the same time. Of course, we could simply do the search once for each of these phrases (or keywords), but there is a more suitable solution that will search the database in a single pass for any occurrence of these keywords. The proposed approach was first presented by Alfred V.Aho and M. J. Corasick [98]. The technique they

developed revolves around constructing a finite automaton during the pre-processing phase from a set of keywords (or patterns). The matching involves the automaton scanning the text string, stepping through the input characters one at a time and changing the state of the automaton. At every state transition for every text character, the algorithm checks if there is a match by observing whether the current state is an output state (which indicates that a pattern has been found) or not. The time complexity of the Aho-Corasick (AC) algorithm is proportional to the total length of the patterns during the pre-processing phase and only proportional to the size of the text being processed during the searching phase [89, p.128]. The AC algorithm has the advantage of examining each text character only once, and locating all occurrences of patterns within a given text in one pass. Its major drawback is the space memory required to store the automaton states, which increases with their number. Other algorithms in respect to multiple pattern-matching were developed, the most common are Wu-Manber and Commentz-Walter algorithms.

The Wu-Manber (WM) algorithm [99] uses the bad-character shift (indicating the characters to skip during the scanning phase) from the Boyer-Moore algorithm, but looks at blocks of text instead of single characters to improve the matching performance: both the pattern and the text are treated as blocks. The algorithm also builds three tables during the pre-processing phase: the hashing or SHIFT table to determine the number of characters that can be skipped when scanning the text, the HASH and PREFIX tables to determine which pattern is a candidate for a match (and eventually to verify this match) when the shift value equals to 0. In practice, the algorithm is more suitable when dealing with patterns of similar length, and its running time does not increase in proportion to the size of the pattern set. But, in case of short patterns, the scanning time increases due to the decrease in characters shifting [100].

The Commentz-Walter (CW) algorithm [101] is a suffix-based, exact multiple string-matching solution that combines the concepts of Boyer-Moore and AC algorithms. In the pre-processing phase, the CW algorithm constructs a finite-automaton similar to that of the AC algorithm but from the reversed set of patterns to use the shifting method of the Boyer-Moore algorithm, which usually handles mismatches in a way to obtain a sub-linear time complexity. The matching involves the automaton scanning through the text string in a backward manner: characters of the patterns are scanned from right-to-left beginning with the rightmost one, exactly as the Boyer-Moore algorithm does. The length of the matching window is the minimum pattern length. In case of a mismatch or a complete match of the pattern, the CW algorithm uses a recomputed shift table to identify portions of the text to be skipped and shifts

the window to the right accordingly. With small numbers of patterns to look for, Boyer-Moore aspects of the CW algorithm can make it faster than the AC algorithm, but with larger numbers of patterns, the AC algorithm has a slight advantage [102, 103].

3.5.1 Review of the AC algorithm

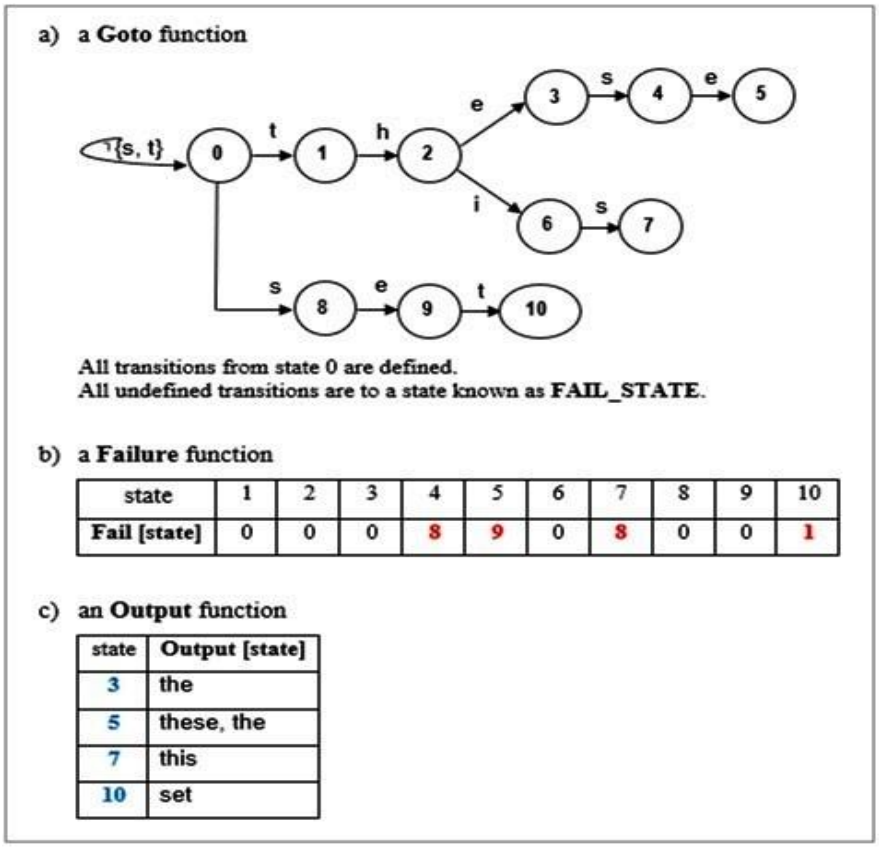
The AC algorithm, first described by Alfred V. Aho and Margeret J. Corasick at Bell laboratories in 1975 [98], is a direct extension of the Knuth-Morris-Pratt (KMP) algorithm [95], which uses automata approach to solve multiple pattern-matching problems. Initially designed to accelerate the library bibliographic search program. The performance gained using this algorithm was between 5x and 10x faster compared to the original program [98].

The AC algorithm builds a finite state machine (FSM) from the patterns, then uses the pattern-matching machine to locate all occurrences of these patterns (that may overlap) within a given text in one pass. Each state of the machine is identified by a number (an integer value). When a character of an input text is processed, one or more finite automaton state transitions are made. State transitions (or machine behavior) are dictated by three functions, namely: (i) the *goto* function *g* indicates state transitions by mapping a pair (current state, input character) into either a state or a fail state. In the latter case, the fail state is indicated by calling the failure function. Some of these states are designed as output states indicating that a set of patterns has been found, (ii) the *failure* function *f* defines which state transition to move to in case of a mismatched character, (iii) the *output* function indicates the patterns found and their locations in the text when the machine reaches an output state.

We can better describe how the algorithm works by considering an example. Suppose we want to search a text for the set of patterns {these, this, the, set}. An appropriate state machine for these four (04) patterns is described in the Goto, Failure and Output tables as shown in Fig 3.6. We start with an empty *goto* function (all characters →S0), and then we add all patterns, one by one. In the same time, we construct the Output table. In the second step, we compute the *failure* function and update the Output table as described next.

Once the automaton is built, the matching process is straightforward using the above-mentioned functions. The machine inspects all the characters from the beginning of the text successively, one character at a time, by changing the state of the automaton (*goto* function is operating) and occasionally emitting output (*output* function is operating). If there is no valid state transition for the character under inspection, then the machine detects a transition failure and tries to investigate a match from other states (*failure* function is operating). We start in

state 0, and then we examine each character of the text. For example, if a character *t* is seen in state 0, then we move to state 1. An *h* character would then take us to state 2. However, if the next character is not an *e* or *i*, we consult Fail [2] and change to state 0. Note that from state 2, Goto[2]['*e*'] = 3 and Goto[2]['*i*'] = 6, but Goto[2][anything else] = FAIL STATE. If we arrive at a state with a non-empty Output function, then a matching string is found. The corresponding C++ code is given in listing 3.6.



state	1	2	3	4	5	6	7	8	9	10
Fail [state]	0	0	0	8	9	0	8	0	0	1

state	Output [state]
3	the
5	these, the
7	this
10	set

Figure 3.6: Pattern-matching machine for the set of keywords {these, this, the, set}

Listing 3.6: C++ implementation of algorithm 1 - Pattern-matching maching according to[98]

```
//str is a pointer to the text to search
//N is the number of the characters to be examined (size of the text)
void AC_Search(unsigned char *str, int N)
{
	int state = 0; for (int i = 0; i<N; i++)
	{
		unsigned char c = str[i];
		while (Goto[state][c] == FAIL_STATE) {state = Fail[state];}
		state = Goto[state][c];
		//output
		if (Output[state] != NULL) {//Here we report a match}
	}
}
```

In the following, we construct the various tables of the AC algorithm. First, we construct the Goto and Output tables. We start with an empty Goto table and step by step we process each word from the pattern set by keeping track of the state transition made in the Goto table when examining each character. Where possible, we overlay patterns that begin with the same characters. We also construct the Output table at the same time: each pattern is identified within the Output table by using its output state as an index to this table. For example, to build the Goto table in Fig 3.6, we start with the pattern *these (S0 ─ t → S1─ h → S2 ─ e → S3 ─ s → S4 ─ e → S5)*. Next, we add the pattern *this (S2─ i → S6─ s → S7)*, and finally the pattern *set (S0─ s → S8─ e → S9─ t → S10)*. The pattern *the (S0─ t → S1─ h → S2─ e → S3)* overlays with the pattern *these*, which is already processed. We just consider its output state in the Output table. Then, we construct the Fail table by taking into account the overlapping search patterns, and the possibility that a given state might find multiple search patterns. We proceed by examining all states of the Goto table by defining "depth" to be the distance of state x from state 0. Then, states 1 and 8 are at depth 1, while states 2 and 9 are at depth 2. Intuitively, Fail[depth 1 states] should bring us to state 0. But Fail [depth 2 states] depends on both Fail and Goto functions of the depth 0 and 1 states. In general terms, Fail[depth n states] depends on the Fail and Goto functions of all states of lesser depth. Consider, for example, the computation of Fail [10] shown in Fig 3.6. The character that brought us from state 3 to state 4 was *s*. Now, we search all other transitions from state 0 to state 3 that make use of *s*. We find just one *s* that takes us from state 0 to state 8. Therefore, Fail [10] is set to 8. Note how the calculation of Fail has turned up the overlap of the patterns *these* and *the*. Output is updated at the same time. When an overlap is discovered, we need only to update (by concatenating) the content indexed by the two overlapping states.

Finally, we must deal with issues that may affect the performance of the algorithm such as its searching speed and the memory requirements. Indeed, we must define the number of possible states, stored as integers, that our machine might have (#define MAXSTATES xx). This number depends on the number and the length of the patterns to look for. For example, in case of searching an English text for a set of 10 short patterns (words) having length less or equal 18 characters, MAXSTATES is set to about 80. While searching for a set of 10 long patterns (sentences) having length between 19 and 70 characters, MAXSTATES will be set to about 600 (see the experience results reported in Tables 4.3 and 4.4). The next issue is how to store the state transitions of the *Goto* function. This would involve a jump table for each state, which depends on the type of the search being performed: if we are searching an English text

for some words (patterns are subset of 256 characters), then the table would contain a next state for each ASCII character (among 256 possibilities), and the Goto table will be defined as follows `(int Goto[MAXSTATES][256];)`. While searching DNA sequences for DNA keywords (patterns are subset of 4 characters A, C, G and T), the Goto table will be declared as follows `(int Goto[MAXSTATES][4];)`. Finally, when dealing with virus signatures based on byte stream or regular expressions (patterns are subset of hexadecimal characters and wildcards), defining the Goto table as `(int Goto[MAXSTATES][25];)` is more than enough. Notice that the memory requirements of the AC algorithm can be taken directly from the different tables used in constructing the automaton during the pre-processing phase since it is the only structure used in the matching process. Unfortunately, the space complexity can be quite large depending on the alphabet and the patterns set. In the worst case it would be O($m \times c$) where c is the size of the alphabet Σ and m is the total length of the patterns.

To speed up the matching process, the *AC_search()* function implemented in C++ (listing 3.6) can be rewritten in assembly as follows:

Listing 3.7: ASM implementation of algorithm 1 - Pattern-matching machine according to[98]

```
//str is a pointer to the text to search
//N is the number of the characters to be examined (size of the text)
declspec(naked) void __fastcall  AC_Search_asm(unsigned char *strp, int N)
{
	_asm push ebp		//To locate strp and N parameters in debug mode
	_asm mov ebp, esp

	_asm push esi
	_asm push ebx
	_asm push edi

	//fastcall ==>	edx == N	ecx == strp	eax	==	last param
	_asm	add edx, ecx	// edx == N+strp
	_asm	xor esi, esi	// esi == state	== 0

loopi :
	//unsigned char c = str[i];
	//while (Goto[state][c] == Fail_STATE) { state = Fail[state]; }
	//state = Goto[state][c];
	_asm jmp do_while_test

redo_this_while :
	_asm mov esi, dword ptr Fail[esi * 4]	//state = Fail[state];

	do_while_test :
	_asm movzx ebx, byte ptr[ecx]		//c=str[i]	//c==ebx
	_asm mov eax, esi
	_asm shl eax, 10
	_asm cmp dword ptr Goto[eax + ebx * 4], -1
	_asm jz redo_this_while
	_asm mov esi, dword ptr Goto[eax + ebx * 4] //state = Goto[state][c];
```

```
#ifdef Include_Output
		// OutCount[state] += OutPutf[state];
		_asm mov eax, OutPutf[esi * 4]	//OutPutf must be static
		_asm add OutCount[esi * 4], eax	//OutCount must be static
#endif
	_asm inc ecx
	_asm cmp ecx, edx
	_asm jle loopi

	_asm pop edi
	_asm pop ebx
	_asm pop esi

	_asm pop ebp
	_asm ret
}
```

We also need to measure the execution time, or a CPU time, of a given function. To this end, we include the following C++ module *"QueryPerformance.h"* to our program.

Listing 3.8: Query performance module

```
#ifndef Time_ms_unitH
#define Time_ms_unitH
#include<windows.h>

class Time_ms{
	LARGE_INTEGER frequency;
	LARGE_INTEGER start;
	LARGE_INTEGER end;

	double interval()
	{
		return static_cast<double>(end.QuadPart - start.QuadPart) /
frequency.QuadPart;
	}

public:
	Time_ms()
	{
		::QueryPerformanceFrequency(&frequency);
		Start();
	}

	void Start() //to be called just before calling the function for which we
			//wish to measure a CPU time
	{
		::QueryPerformanceCounter(&start);
	}

	double End() //to be called just after the function for which we
			//wish to measure a CPU time returns
	{
		::QueryPerformanceCounter(&end);
		return interval();
	}
};
#endif
```

3.5.2 Review of the AC algorithm using the *next-move* function

The failure function *f* as implemented in the previous AC algorithm is not optimal. Consider the pattern-matching machine of Fig 3.6. We see Goto[4] ['*e*'] = 5. If the machine is in state 4 and the current input character is not an *e*, then the machine would enter state Fail[4] = 8. Since the machine has already determined that an input character is not an *e*, it does not need to consider the value of the *goto* function of state 8 on *e*. In fact, if the pattern *set* was not present, then the machine could change directly from state 4 to state 0. Thus, skipping an intermediate transition to state 0. To eliminate unnecessary failure transitions in the previous version of the AC algorithm (the original version), the *next-move* function of a deterministic finite automaton (DFA) can be used in place of the *goto* and *failure* functions. Converting the algorithm to a DFA allows to indicate for each pair (given state, given character) the next state to move to, which is always a valid state.

Listing 3.9: C++ implementation of the AC algorithm using the next-move function

```
//len_movef:table indicating the number of characters to process for each state
//movef_char: table indicating the character being processed for a given state
//movef_nextstate: table indicating the next state if a match occurs
void AC_Nextmove_search(unsigned char *str, int N)
{
	int state = 0;
	for (int j = 0; j < N; j++)
	{
		unsigned char c = str[j];
		for (int i = 0; i < len_movef[state]; i++)
		{
			if (c == movef_char[state][i])
			{
				state = movef_nextstate[state][i];
				goto nextOK;
			}
		}
		state = 0;

nextOK:
	//Statement used in [98] and can be skipped if we want to do output later
	if (Output[state] != NULL) { //Here we report a match }

	//Here we use an array to do output after processing the whole text
	OutCount[state] += OutPutf[state]
		}
	}
}
```

The *next-move* function is computed from the *goto* and the *failure* functions using algorithm 4 in [98]. The purpose of this function is to build the three (03) following tables: len_movef, movef_char and movef_nextstate (indicating respectively for each state the

number of characters to process, the character being processed and the next state in case of a match) used by the algorithm. The *next-move* function is coded as follows. For example, in state 0, we have a transition on *s* to state 8, a transition on *t* to state 1, and all other transitions on any other characters are to state 0 (not shown in Fig 3.6). The C++ code for the *next-move* function according to [98] is given in listing 3.9, and its corresponding ASM version in listing 3.10.

Listing 3.10: ASM implementation of the AC algorithm using the next-move function

```
---------------------------------------------------------------------------------------------------------------------
declspec(naked) void __fastcall  movef_search_asm(unsigned char *strp, int N)
{
       //fastcall ==>     edx == N       ecx == strp     eax == last param
       _asm push ebp
       _asm mov ebp, esp

       _asm push esi
       _asm push ebx
       _asm push edi

       _asm  add edx, ecx    // edx == N+strp
       _asm  mov ebp, ecx
       _asm  xor esi, esi    // esi == state  == 0

loopj :
       _asm mov al, [ebp]

loopi :
       _asm mov ecx, len_movef[esi * 4]
       _asm shl esi, 8   // state  =  state*256
                         // 256 = size of a line of movef_nextstate table
       _asm mov ebx, ecx
       _asm lea edi, movef_char[esi]
       _asm repne scasb
       _asm jne notfound
       _asm sub ebx, ecx
       _asm dec ebx    // when char is at 0 position, ecx is decremented by 1 via
                       // repne scasb instruction

       _asm add esi, ebx // state_X_256  = state_X_256 + i

       // state = movef_nextstate[state][i];
       // 4 = table movef_nextstate element size

       _asm mov esi, movef_nextstate[esi * 4]
       goto nextOK;

notfound:
       _asm xor esi, esi

nextOK :
       // OutCount[state] += OutPutf[state];
       _asm mov eax, OutPutf[esi * 4]          //OutPutf must be static
       _asm add OutCount[esi * 4], eax         //OutCount must be static
       _asm inc ebp
       _asm cmp ebp, edx
       _asm jle loopj
```

```
        _asm pop edi
        _asm pop ebx
        _asm pop esi

        _asm pop ebp
        _asm ret
}
```

Theoretically, the DFA can reduce up to 50% of state transitions, which would reduce extra comparisons and accelerate significantly the matching process. In practice such accelerations cannot be reached since the pattern-matching machine spends a lot of its time in state 0, from which there are no failure transitions [98] as shown in our experience results reported in Tables 4.2, 4.3 and 4.4. From the above pseudo-code and algorithms 1, 2 and 3 given in [98], it appears clearly that building the different tables is the difficult task of the AC algorithm. The searching process is easy once these tables are available. This method of coding state transitions is more economical than storing them as a two-dimensional array. However, it requires extra memory space larger than the corresponding representation for the *goto* function. Using the *next-move* function with the set of patterns {these, this, the, set} would make the sequence of state transitions shown in Fig 3.7.

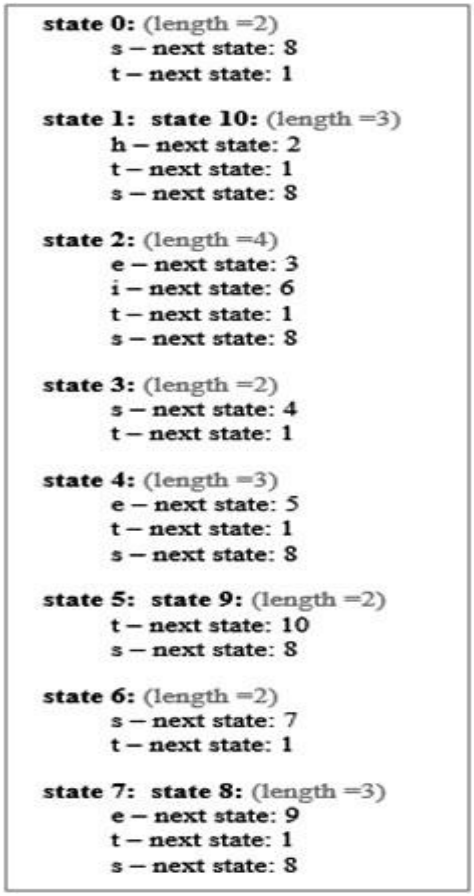


Figure 3.7: State transitions using the *next-move* function

Chapter 4

4 SIMD Implementation of the Aho-Corasick Algorithm using Intel® AVX2

Over the years, computer designers have always improved the performance of the machines. The limitations of the speed of electronic components, which has achieved extremely high packaging densities of circuits and almost reached its limits has led to a search for other accelerating mechanisms at architectural level with the aim to *(i) decrease latency*, or the time from start to completion of an operation, by using multiple register sets, placing high speed memory -caches between the processor and main memory, hardware implementation of frequently-executed operations, etc. and *(ii) increase bandwidth*, or the width and rate of operations, through one of the following two forms of parallelism: pipelining and replication [104].

Pipelining is the process of breaking down the execution of an operation into several stages, which enables simultaneous execution of different stages for a stream of operations by different modules able to operate concurrently (the pipeline). This process allows each subsequent operation to enter the pipeline as soon as the previous operation has exited the first stage. Once all stages of the pipeline are filled, a new result of an operation execution becomes available after every unit of time it takes to complete the slowest stage. Replication, on the other hand, is the process of duplicating executing units (replicating hardware resources) to enable simultaneous execution of different operations [104].

Common parallel architectures that use pipelining and/or replication can be classified as shown in Figure 4.1. This classification clearly situates the multimedia extensions, which borrow concepts from both the instruction-level parallel and the data parallel architectures, and when incorporated in process-level parallel architectures, they enable exploiting multilevel parallelism.

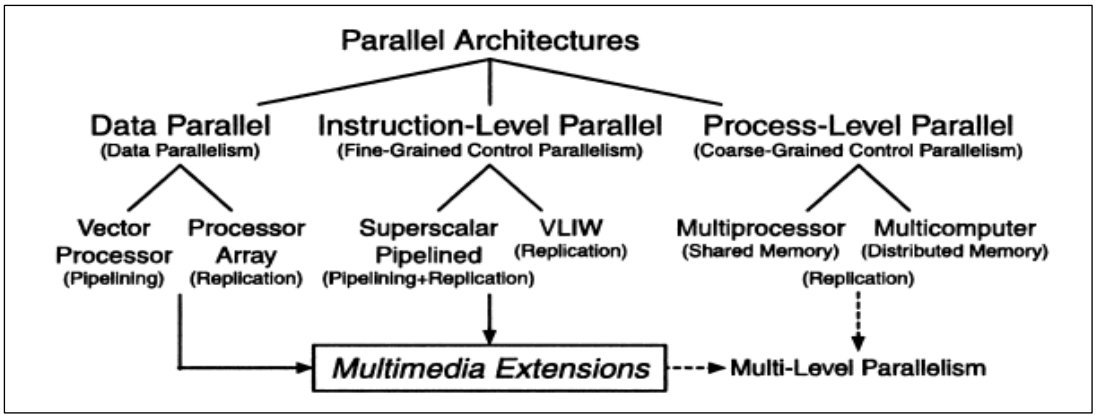


Figure 4.1: Classification of parallel architectures [105]

Multimedia extensions follow the Single Instruction Multiple Data (SIMD[33]) paradigm, where a single control unit dispatches instruction to a set of simple processing items that apply each of these instructions synchronously to different data, thus providing a new way to utilize data parallelism at instruction level [104]. Different processors, as shown in Table 4.1, support different multimedia instruction sets each with its own characteristics, but the underlying concepts are quite the same.

Processor	Vector Extension Name
Sun UltraSPARC	VIS (Visual Instruction Set)
Hewlett-Packard PA-RISC	MAX (Multimedia Acceleration eXtensions)
Intel Pentium	MMX
Intel Pentium	SSE (Streaming SIMD extensions)
Intel Core i7	SSE4 (Streaming SIMD extensions)
Intel Sandy Bridge processor	AVX (Advanced Vector Extensions)
Silicon Graphics	MDMX (MIPS Digital Media eXtentions)
Digital Alpha	MVI (Motion Video Instructions)
PowerPC	AltiVec
AMD K6-2	3Dnow!
AMD	XOP, FMA4, CVTI16
ARM	NEON
MIPS	DSP ASE

Table 4.1: First-generation short-vector processors [106]

In the following, we focus on multimedia extensions to the Intel® architecture, and more specifically on Advanced Vector Extensions (AVX).

[33] From Flynn's taxonomy 1966 (a specific classification of parallel computer architectures): SISD, SIMD, MISD and MIMD

4.1 Intel® SIMD extensions

The SIMD extensions were originally designed to speed up processing applications such as image processing, 3D rendering, video and audio processing, speech recognition, and data communication algorithms. They can also be used for other data-intensive scientific computations such as computational biology.

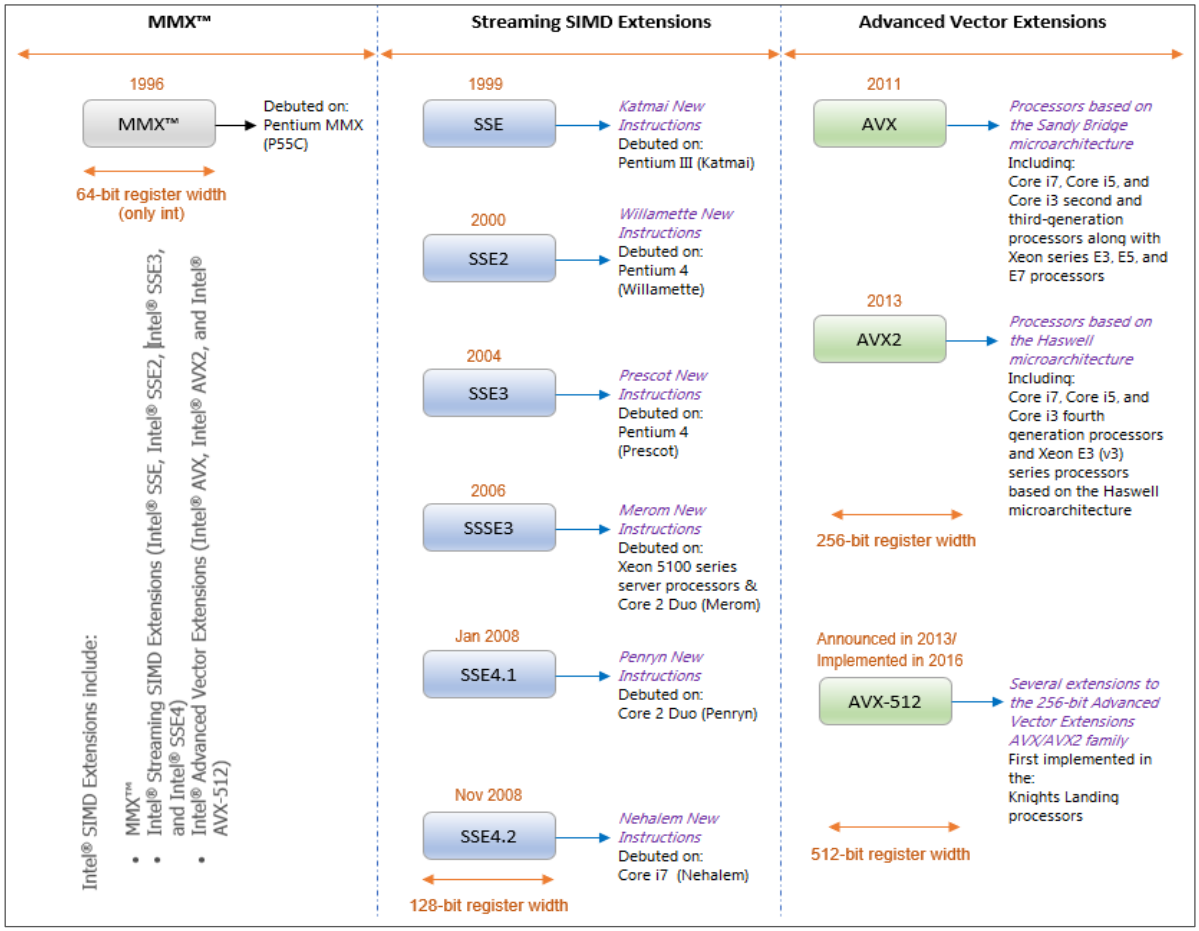


Figure 4.2: Intel® SIMD extensions

For Intel® processors, the vector instructions have been gradually introduced in different processor generations. Started with the MMX™ in 1996 (instructions in this extension operate only on packed integer values and rarely used in modern processors), followed by several Streaming SIMD Extensions (SSE) versions from 1999 to 2008 (available in almost any today processor), and continued to this day with AVX (supported in new processors). Figure 4.2 shows different Intel SIMD extensions to the IA-32 architecture[34].

[34] IA-32 short for "Intel Architecture 32-bit", is the 32-bit version of the x86 instruction set architecture, which is initially developed by Intel and implemented in its 8086 microprocessor (in 1978) ant its 8088 variant. The x86 family includes microprocessors compatible with the Intel 8086 instruction set.

4.1.1 MMX™ Technology

In 1996, Intel announced the first widely deployed desktop SIMD with a multimedia extension technology, called MMX[35], to the IA-32 architecture incorporated into the Pentium® processor P55C. The 64-bit MMX™ technology is a major improvement in Intel's architecture to accelerate multimedia applications. Microsoft incorporated support for these extensions in Direct3D, interactive 3D graphics, a set of API services for real-time, and their Visual C++ compiler [109]. The MMX™ technology consists of the following extensions to the Intel architecture:

- Single Instruction Multiple Data (SIMD) techniques for logic and arithmetic operations on packed data from/to 64-bit registers.
- Eight 64-bit wide MMX registers accessed directly using the register names MM0 to MM7, shared with Floating-Point (FP) registers.
- Four new 64-bit integer data types:
  - 8 packed bytes (8 x 8-bit)
  - 4 packed words (4 x 16-bit)
  - 2 packed doublewords (2 x 32-bit)
  - 1 quadword (l x 64-bit)
- 57 new basic general-purpose integer instructions that operate on the 64-bit data types to take advantage of the MMX technology.

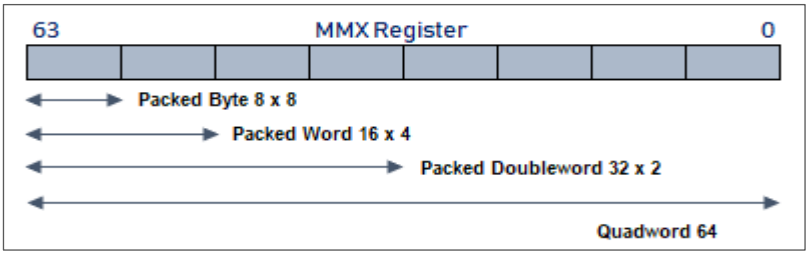


Figure 4.3: MMX data types [109]

The MMX technology supports basic arithmetic operations, logical operations, comparison, conversion instructions to convert between the new data types - pack data together, and unpack from small to larger data types, shift instructions that operate on the 64-bit integer data types, and data movement (data transfer for MMX registers or 32-bit and 64-bit load/store to memory) [109].

[35] MMX is actually not an abbreviation but a meaningless initialism trademarked by Intel [107]. The MMX initials have been explained as standing for *MultiMedia eXtension* [108], *Multiple Math eXtension*, or *Matrix Math eXtension*.

4.1.2 Streaming SIMD Extensions (SSE)

The MMX™ extensions introduced SIMD capability into the IA-32 architecture, with the 64-bit MMX registers, 64-bit packed integer data types, and instructions to perform SIMD operations on packed integers. SSE extensions expand the SIMD execution model to handle packed and scalar single-precision floating-point values contained in 128-bit registers.

In 1999, Intel introduced the Pentium III processor where they incorporated the SSE extensions (an update to MMX™), also called Katmai New Instructions (KNI) since they were originally included on the Katmai processor, which was the code name for the Pentium III core revision. Others processors based on the Pentium III core such as the Celeron 533A and faster Celeron processors also support SSE instructions [110].

The SSE instructions are especially useful with MPEG2[36] decoding. Processors supporting SSE can perform MPEG2 decoding in software at high-speed without the need to have an additional hardware decoder card. They are also much faster than previous processors in the field of speech synthesis and recognition. The main advantage of SSE over MMX™ is the support for single-precision floating-point SIMD operations, which was a bottleneck in the 3D graphics processing. SSE extensions are fully compatible with all software written for IA-32 processors before, and SSE floating-point instructions can be mixed with MMX™ instructions with no performance penalties. SSE adds 70 new instructions for graphics and sound processing over what MMX™ provided, control data cache ability, and data prefetching [110].

SSE expanded over the generations of Intel® processors to include SSE2, SSE3/SSSE3, and SSE4. These technologies add the following extensions to the IA-32 architecture, while maintaining full compatibility with previous IA-32 processors, applications and operating systems:

- Eight 128-bit registers: XMM0 - XMM7
- Two 128-bit floating-point and five 128-bit integer data types that can be used as:
  - 4 packed single-precision floating-point numbers (4 x 32-bit)
  - 2 packed double-precision floating-point numbers (2 x 64-bit)
  - 16 packed bytes (16 x 8-bit)
  - 8 packed words (8 x 16-bit)
  - 4 packed doublewords (4 x 32-bit)
  - 2 packed quadwords (2 x 64-bit)
  - 1 double quadword (l x 128-bit)

[36] MPEG2 is an audio and video codec standard published by the Moving Pictures Expert Group. It is the codec used to compress digital broadcast video and DVDs.

- Instructions that operate on 128-bit packed single-precision FP and additional 64-bit SIMD integer data types.

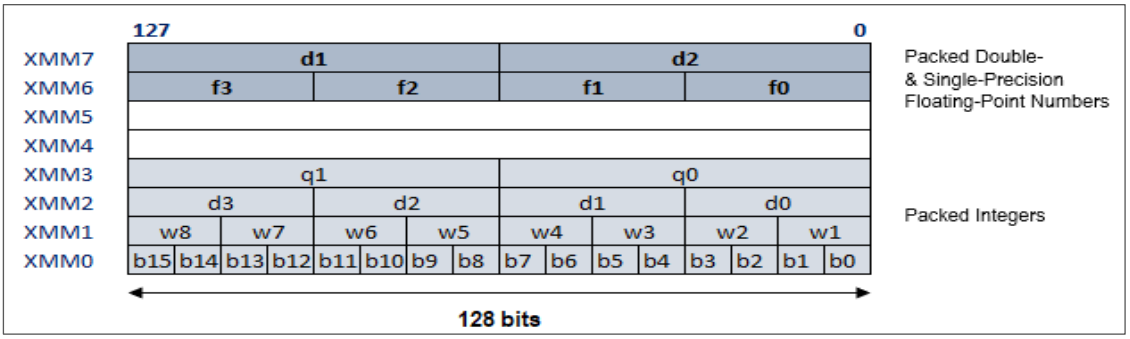


Figure 4.4: The 128-bit Streaming SIMD Extensions [114, p. 182]

Each SSE version adds new instructions to perform on multiple data objects and increases performance:

- ✓ **SSE2**, also called Willamette New Instructions (WNI), was introduced in November 2000 with the Pentium 4 processor, and added 144 additional SIMD instructions. It brought two major features: double-precision (64-bit) floating point for all SSE operations, and extended SIMD integer instructions introduced with MMX™ technology from 64 to 128 bits (XMM registers). SSE2 included all the previous MMX™ and SSE instructions.
- ✓ **SSE3**, called Prescott New Instructions (PNI), is an upgrade to SSE2 introduced in February 2004, with the Pentium® 4 Prescot processor. It included 13 new SIMD instructions that are primarily designed to improve thread synchronization, complex math, and specific application areas such as media and gaming. It also allowed to add or multiply two numbers that are stored in the same register, which wasn't possible in the earlier versions. SSE3 included all the previous MMX™, SSE, and SSE2 instructions.
- ✓ **SSSE3** (Supplemental SSE3), Merom New Instructions (MNI), was first introduced in June 2006 in Intel® processors based on the Core microarchitecture with the Xeon 5100 series server processors, and later in the Core 2 processors. SSSE3 adds 32 new SIMD instructions to SSE3 to accelerate computations on packed integers.
- ✓ **SSE4**, added 54 new instructions in Intel® 64 processors. A subset of 47 of these instructions were made available in January 2008 on the successor of Intel® Core™ microarchitecture (code named Penryn) -versions of the Intel® Core 2 processors. This

subset, referred to as SSE4.1 or Penryn New Instructions (PNI), was designed to accelerate the workloads of media, imaging, and 3D processing. It also improved compiler vectorization and support for DWORD computations. In November 2008, SSE4.1 was updated with the second subset of 7 instructions supported by Intel® processors based on the Nehalem microarchitecture -Core i7 processors, referred to as SSE4.2. These instructions improved string and text processing, and enhanced the capability of the 128-bit integer SIMD capability in SSE4.1. SSE4 supports all previous SSE version instructions [111].

4.1.3 Advanced Vector Extensions (AVX)

AVX is the next generation of the SIMD instruction sets, which increases the CPU registers size, provides an instruction format which allows three input arguments, and extends vector processing capability. This results in higher performance and more efficient data management for image and audio/video processing, 3D modeling, financial analytics, scientific simulations, and more. These extensions include Intel® AVX, Intel® AVX2, and Intel® AVX-512.

- ✓ **Intel® AVX,** is a 256-bit SIMD operations extension of Intel® SSE, designed to deal with applications which make a more intense use of floating-point operations such as visual processing, cryptography and data mining. In this extension, the support for integer operands is lacking. It was released in the early 2011s as part of the second-generation Intel® Core™ processor family (supported first by the Intel® Sandy Bridge processor released in Q1, 2011). AVX extends the previous SIMD instructions by:
  - Expanding all 16 XMM registers (128-bit SIMD registers) to 256 bits: YMM0 – YMM15, while SSE instructions operate on the lower half of the YMM registers.
  - Adds a three-operand non-destructive operation where the destination register is different from the two source operands.
  - Introduces a new prefix coding scheme, referred to as VEX[37], in instruction encoding format [112].
  - Introducing new operations to enhance vectorization such as:
    - Broadcasts;
    - Masked load & store.

[37] The VEX (from "vector extensions") prefix encoding allows the definition of new instructions and the extension or modification of previously existing instruction codes. It applies to SIMD instructions operating on YMM and XMM registers, but it is not supported for instructions operating on MMX™.

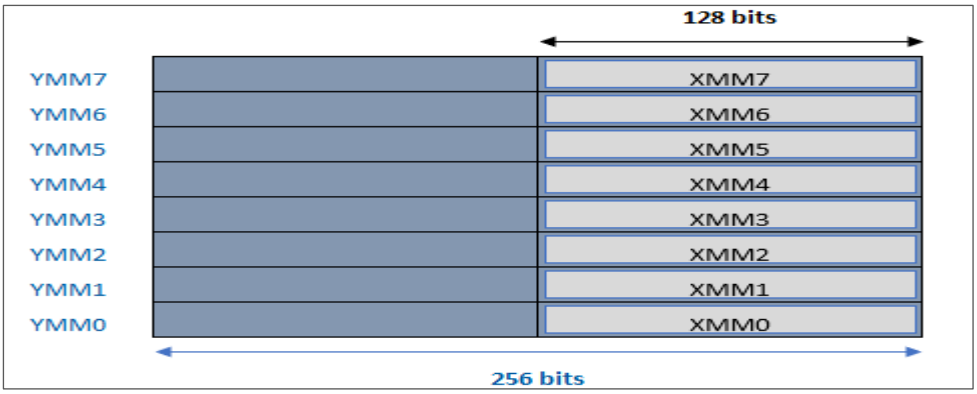


Figure 4.5: Intel SIMD registers (AVX) [114, p. 329]

Processors based on the Sandy Bridge microarchitecture and supporting AVX include Core i7, Core i5, and Core i3 second and third-generation processors along with Xeon series E3, E5, and E7 processors [113].

✓ **Intel® AVX2,** also known as Haswell New Instructions, was released in 2013 with the fourth generation Intel® Core™ processor family (supported first by the Intel® Haswell processor released in Q2, 2013). Intel® AVX2 follows the same programming model as the Intel® Advanced Vector Extensions (Intel® AVX), and extends it with:

- Support for integer computation demanding algorithms by extending SSE and AVX with 256-bit integer instructions.
- AVX2 includes the Fused Multiply-Add (FMA) extension which allows numbers to be multiplied and added in one operation: three-operand fused multiply-accumulate support (FMA3)
- Enhances vectorization with *Gather* operation that loads vector items from non-contiguous memory locations, and introduces vector *Shift* operations.
- It provides enhanced functionalities for *Broadcast/Permute* operations on data elements, and instructions to fetch non-contiguous data elements from memory.

The specification of the AVX2 instruction set and information needed to create applications using this extension are available under *the Intel® Architecture Instruction Set Extensions Programming Reference* [114]. Processors based on the Haswell microarchitecture and supporting AVX2 include Core i7, Core i5, and Core i3 fourth generation processors and Xeon E3 (v3) series processors based on the Haswell microarchitecture [113].

- ✓ **Intel® AVX-512** consist of several extensions to the 256-bit Advanced Vector Extensions AVX/AVX2 family of SIMD instructions announced by Intel® in July of 2013 and implemented later in Intel® Knights Landing processors (2016). These extensions not all meant to be supported by all processors implementing them[38]. AVX-512 extensions use a new EVEX prefix encoding with support for 512-bit vector registers, double the number of registers (32 registers ZMM0 – ZMM31), double the width of data registers (512-bit/register) and double the width of Fused-Multiply Add (FMA) units, compared to Intel® AVX2. Intel® AVX-512 instructions offer higher performance capabilities for the most demanding computational tasks [115].

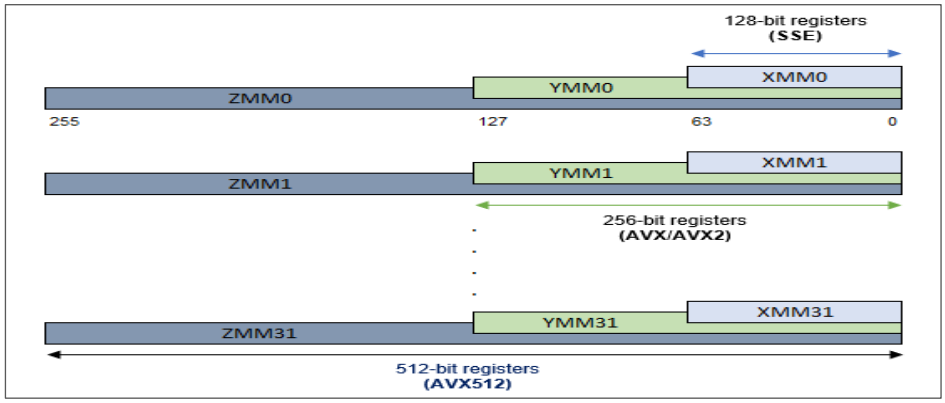


Figure 4.6: Intel SIMD registers (AVX-512)

In the vectorization process of the AC algorithm proposed in section 4.5, we use AVX2 extension to benefit from SIMD instructions operating on integer data type since states generated in the pattern-matching machine by running the AC algorithm are coded as integer values. But prior to using AVX2 instructions, we must first make sure that hardware and operating system support this extension as explained next.

4.1.4 Test for AVX2 support

In order for an application software to correctly use the AVX (or AVX2) instructions and avoid runtime crashes, it must properly detect hardware support for these instructions using CPUID instruction[39], and never assumes that by simply knowing the processor's model,

[38] For example, the Intel's Knights Landing processor (2016) supports the following AVX-512 extensions: AVX-512 Foundation, AVX-512 Conflict Detection Instructions (CD), AVX-512 Exponential and Reciprocal Instructions (ER) and AVX-512 Prefetch Instructions (PF).

[39] The CPUID (CPU IDentification) instruction is a processor supplementary instruction introduced by Intel in 1993 with the Pentium and SL-enhanced 486 processors allowing software to discover details of the processor features.

family, or its brand name. It is important to understand that AVX (or AVX2) instruction set is supported on a particular processor only if the corresponding CPUID feature flag is set [116]. In assembly language, the CPUID instruction takes no parameters but instead it implicitly uses the EAX register to determine the processor feature information (values returned in EAX, EBX, ECX and EDX registers, according to the input value entered initially in EAX register).

1- ***Detection of AVX instructions***

Application detection of AVX instruction set operating on the YMM state follows the general procedural flow as shown in Fig 4.7.

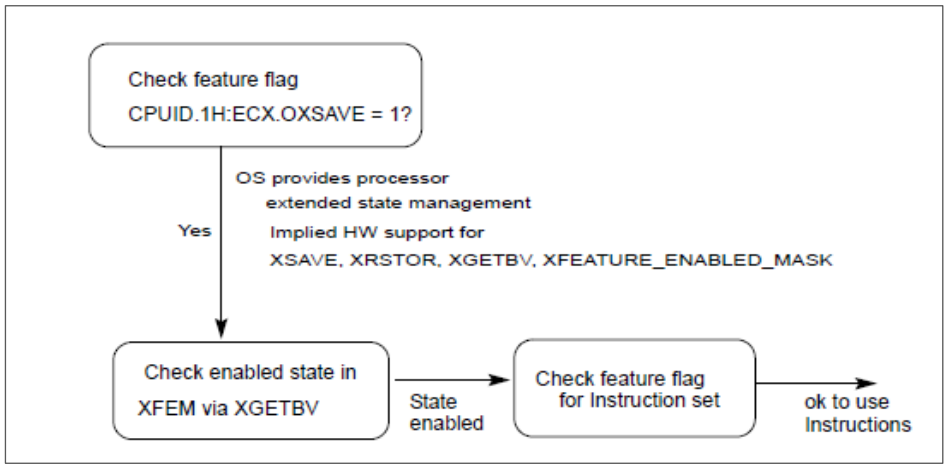


Figure 4.7: General procedural flow of application detection of AVX [114]

According to *Intel® Architecture Instruction Set Extensions Programming Reference* [114], the application software must identify that the operating system supports the XGETBV instruction, the YMM register state, in addition to processor's support for YMM state management using XSAVE/XRSTOR and AVX instructions as illustrated by the following pseudocode.

Listing 4.1: AVX detection process pseudocode [114]

```
INT supports_AVX()
{
    // Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 ==> XGETBV is
    // enabled for application use.
    // Detect CPUID.1:ECX.AVX[bit 28] = 1 ==> AVX instructions supported.
    // Results returned in EAX
       mov eax, 1
       cpuid
       and ecx, 018000000H
       cmp ecx, 018000000H // check both OSXSAVE and AVX feature flags
       jne not_supported
```

```
    // processor supports AVX instructions and XGETBV is enabled by OS

     mov ecx, 0    // specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV        // result in EDX:EAX
     and eax, 06H
     cmp eax, 06H  // check OS has enabled both XMM and YMM state support
     jne not_supported
     mov eax, 1
     jmp done

NOT_SUPPORTED:
mov eax, 0
done
}
```

2- ***Detection of AVX2 instructions***

According to [114], hardware support for AVX2 is indicated by the CPU identification instruction "CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1". But prior to this, application software must first identify that hardware supports AVX as explained before. AVX2 detection process pseudocode is given below.

Listing 4.2: AVX2 detection process pseudocode [114]

```
INT supports_avx2()
{
    // result in EAX
       mov eax, 1
       cpuid
       and ecx, 018000000H
       cmp ecx, 018000000H // check both OSXSAVE and AVX feature flags
       jne not_supported

     // processor supports AVX instructions and XGETBV is enabled by OS
       mov eax, 7
       mov ecx, 0
       cpuid
       and ebx, 20H
       cmp ebx, 20H   // check AVX2 feature flags
       jne not_supported

       mov ecx, 0    // specify 0 for XFEATURE_ENABLED_MASK register
       XGETBV        // result in EDX:EAX
       and eax, 06H
       cmp eax, 06H  // check OS has enabled both XMM and YMM state support
       jne not_supported

       mov eax, 1
       jmp done

NOT_SUPPORTED :
mov eax, 0
done :
}
```

4.2 Vectorization: Data Parallelism

Vectorization, or SIMD processing, is the process in which an algorithm is converted from a sequential or scalar implementation, that performs an operation one pair of operands at a time, to a vector process where multiple data operands are simultaneously performed in only one instruction (data-level parallelism), provided that this scalar algorithm is suitable for being parallelized using vectorization techniques [117]. In SIMD processing, operands[40] are treated not as individual integers or float-point numbers but rather as vectors. Moreover, not only SIMD instructions like addition, logical OR, data movement, conversion or comparison operate on vectors but also reading from and writing to memory (load/store instructions) operate on vectors too (all operations in SIMD are vector operations). In the example of the figure below, only one SIMD instruction is needed to process vector items, whereas in sequential processing, without SIMD, multiple instructions would be used instead.

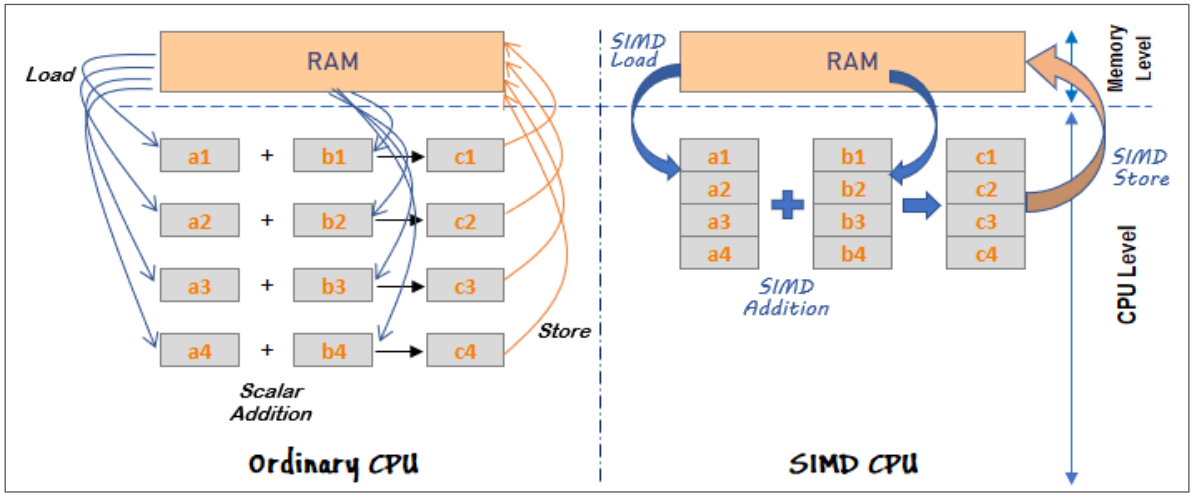


Figure 4.8: Scalar vs. SIMD vector addition

Figure 4.8 shows a comparison between scalar and SIMD vector addition. In case of scalar addition of four elements of vector *A(a1, a2, a3, a4)* with four elements of vector *B(b1, b2, b3, b4)*, each element of A and B is first read from memory before being added together, and then written back to the memory associated with vector *C(c1, c2, c3,c4)*. The CPU reads separately *a1* and *b1* form memory to the CPU registers (02 *load* instructions), adds *a1* with *b1* (01 *add* instruction) and saves the result back to the memory (01 *store* instruction). Instructions are issued in the same manner for other vector elements, giving rise to a total of 16 separate instructions: 4 × (02 *load*, 01 *add*, 01 *store*) instructions. In case of

[40] Adjacent data items of the same type and size refer to as vectors or packed data.

SIMD addition, the same task is performed, but this time only (01 *load* instruction) is used to read four elements of vector A from memory, similarly (01 *load* instruction) is used to read elements of B, then (01 *SIMD add* instruction) sums the corresponding elements of A and B, with a final (01 *store* instruction) used to write back four elements of C to memory, resulting in 04 separated SIMD instructions. In theory, the speed can be multiplied by 4: 16 separate instructions are reduced to 04 to produce the same result.

4.3 Vectorization approaches (SIMD programming methods)

A key to increase application performance is to take advantage of vectorization, which occurs when code makes use of SIMD instructions. Different approaches to achieving vectorization exist. The Intel® C++ compiler (ICC) provides several levels of vectorization supports such as inline assembly, intrinsics, C++ vector libraries and auto-vectorization (see Fig 4.9).

1. The first form of vectorization is to explicitly call the processors SIMD instructions using directly low-level assembly code (or inline assembly such as C-ASM in C/C++ code) and available registers. This way of utilizing vector instructions offers more control over the program and yields the maximum performance of the system but cross-platform porting is quite difficult (changes must be made at the source code level to execute the same program on another processor architecture);
2. Use of the compiler intrinsic functions (assembly-coded C style functions that provide access to many Intel® instructions, including SIMD instructions, without the need to explicitly write assembly code). In this approach, high level languages such as C/C++ or FORTRAN can be used to write the code. It provides almost the same benefits as using inline assembly and saves us to deal with register allocations, instructions scheduling and stack calls;
3. Use of Intel® Cilk™ Plus array notation[41] and elemental function[42] syntax. These notations can be used separately from each other to help the compiler perform parallelization. More details to using these language extensions can be found in [118].

[41] Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. In array notation **A[0:n] = 5;** sets the first n elements of array A to 5. Its equivalent scalar C/C+ Code: **for (i = 0; i < n; i++) A[i] = 5;**

[42] An elemental function is a regular function, which can be invoked either on scalar arguments or on array elements in parallel.

4. Use of high-level mathematics libraries, such as Intel® MKL (Math Kernel library), which implement vector instructions of common operations for linear algebra, signal analysis, statistics, etc [119]. Intel® MKL functions are fully thread-safe, so that multiple calls implying different threads will not conflict with one another [117]. Intel® Integrated Performance Primitives, or IPP library for C/C++ developers, is another cross-platform software library (available for Linux, macOS, Windows and Android operating systems.) that provides vectorized functions, which can be used for multimedia and data processing applications [117].

5. Compiler auto-vectorization: The easiest form of vectorization where no changes to the source code are required. As a result, the portability of the code is preserved. When compiling from high-level languages such as C/C++ or Fortran, automatic vectorization is enabled at the compiler default optimization level -O2. However, to gain the most from automatic-vectorization, there are several ways in the compiler can be provided with additional information that enable it to better vectorize -does the transformation and produces efficient SIMD code based on a handful of directives (compiler hints) in the code like #pragma SIMD statements to automatically guide the compiler to vectorize more loops, #restrict keywords indicating to the compiler that a given array corresponds to a unique region of memory (to eliminates the need for run-time checks for overlapping arrays), etc. This approach is referred to as auto-vectorization hints, or semi auto-vectorization [119];

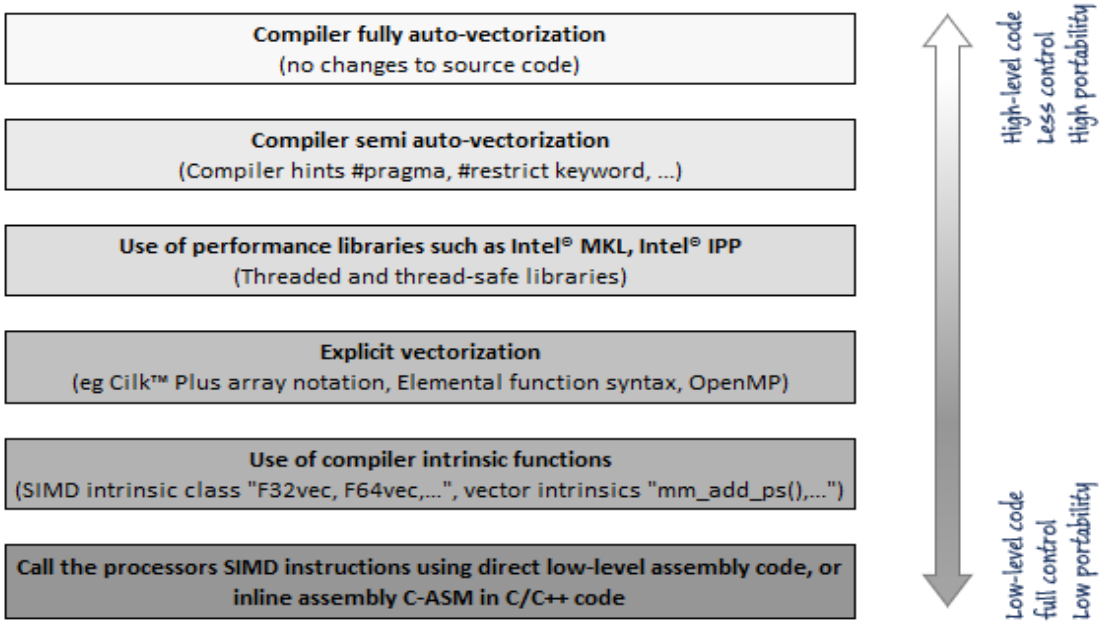


Figure 4.9: SIMD programming methods

Compiler auto-vectorization is the easiest form of vectorization that is present in many modern compilers (C/C++/Fortran), where specific portions of the sequential code are converted into equivalent parallel ones intended for use on vector processors. This is the most convenient way to do vectorization because cross-platform porting is preserved (performed by the compiler), in contrast to embedding SIMD assembly code into a source program, which is rather complicated because it requires careful handling of data transfers between available vector registers and memory. This latter method delivers better performance than the compiler auto-vectorization, but cross-platform porting is not guaranteed [119]. It is chosen here because it seems to be the most effective one for speeding up the AC algorithm.

4.4 Characteristics of SIMD operations

In SIMD processing, the same operation is applied to each element of the source operands. However, this is a constraint because programs often do not map well onto pure vector operations and even if it is the case, they might not reach the best resource utilization [120]. Programs can benefit from SIMD processing if they have highly repetitive loops, and use intensively instructions that operate on independent values in parallel. The practical speed-up with vectorization comes from the efficient data movement and the identification of vectorization possibilities in the program itself [121]. Vectorization fails if the structure of the program is not suitable for parallelization and its data types are either mixed, dependent or not aligned.

In the following we outline two main obstacles to vectorization that either prevent it or cause the compiler to decide that vectorization would not be worthwhile:

- ***Non-contiguous memory accesses:*** Considering 04 consecutive integers or floats that can be loaded directly from memory in a single AVX instruction. If the 04 integers are not aligned[43], they must be loaded separately, which is less effective since it implies using multiple instructions. In listing 4.3, we give two illustrative examples of non-contiguous memory access: loops with non-unit stride, and loops with indirect addressing.

Listing 4.3: Examples of non-contiguous memory access

```
// arrays accessed with stride 2
```

[43] Alignment is the adjustment of any data object in relation with other objects.

```
for (int i = 0; i<SIZE; i += 2) b[i] += a[i] * x[i];

// indirect addressing of x using index array
for (int i = 0; i<SIZE; i += 2) b[i] += a[i] * x[index[i]];
```

- ***Data Dependencies:*** Vectorization involves changes in the order of operations within a loop, since each SIMD instruction operates on several data items simultaneously. Vectorization is made possible only if this change of order does not impact the results of the calculation. All data dependencies must be respected to prevent incorrect results. The example of listing 4.4 shows a case of data dependencies where data items that are written to memory may appear in other iteration of the individual loop.

Listing 4.4: Examples of data dependencies

```
-----------------------------------------------------------------------------------------
// Example of read-after-write dependency (also known as flow dependency)
A[0] = 0;
for (j = 1; j<MAX; j++) A[j] = A[j - 1] + 1;
// this is equivalent to:
A[1] = A[0] + 1; A[2] = A[1] + 1; A[3] = A[2] + 1; A[4] = A[3] + 1;

// Example of write-after-read dependency (an anti-dependency)
for (j = 1; j<MAX; j++) A[j - 1] = A[j] + 1;
// this is equivalent to:
A[0] = A[1] + 1; A[1] = A[2] + 1; A[2] = A[3] + 1; A[3] = A[4] + 1;
```

In this example, the first loop cannot be vectorized because if the first two iterations are executed concurrently by a SIMD instruction, the value of A[1] may be used by the second iteration before it has been calculated by the first iteration, which could yield erroneous results. The second loop is also not safe for parallel execution, since A[1] may be used by the second iteration (iteration with the write) before it has been red by the first iteration. Nevertheless, it should be noted that in the second loop, no iteration with a higher value of j can complete before an iteration with a lower value of j, and so vectorization is safe; yields the same result as non-vectorized code.

For helping the compiler to better vectorize the code, we should avoid complex loop termination conditions and dependencies between loop iterations. We should also avoid branches such as *switch*, *goto* or *return* statement, and prefer the use of array notation to pointers because C programs impose very few restrictions on their use. Since effective vectorization comes from efficient data movement, we have to choose an appropriate data

layout that, in combination with code restructuring[44], will result in aligned memory accesses throughout the program: If our compilation targets Intel® AVX2 instructions, we try to align the data of the code on a 32- byte boundary since AVX data movement instructions, as other AVX instructions, are rather sensitive to alignment [122].

4.5 SIMD Implementation of the Aho-Corasick Algorithm using Intel® AVX2

One effective way to benefit from different levels of parallelism included in modern processors is the use of vectorization. The program to optimize has to be written using available vector registers (XMM, YMM and ZMM in case of Intel® processors) and SIMD instructions (MMX™, Streaming SIMD Extensions SSE and AVX instructions in case of Intel® processors). In this section, we try to identify vectorization opportunities from the structure of the AC program (its code structure), we particularly look to parallelize some data processing.

4.5.1 Vectorization of the Aho-Corasick Algorithm using Intel® AVX2

The second version of the AC algorithm, using the *next-move* function, seems vectorizable. Considering the sequence of state transitions in Fig 3.7. Each time the machine enters a new state, a text character being inspected, has to be matched against all characters of that state transitions one at a time. One way to speed up the searching process is by comparing simultaneously the text character to all characters of that state transitions. For example, if the machine enters state S2 then the character being examined is concurrently compared to *e*, *i*, *t* and *s* characters (data-level parallelism). This can be achieved using vector processing techniques. To implement the vectorized algorithm, we use VPBROADCASTB and VPCMPEQB Intel® AVX2 instructions. The VPBROADCASTB instruction loads a byte integer (a text character being processed) from memory via ECX and EDX registers and broadcasts it to thirty-two (32) locations in YMM1 register (data are aligned to 32-byte boundaries in order to use AVX extension). Next, we load all characters of state transitions of a current state in YMM2 register using VMOVUPS instruction. Finally, a SIMD comparison is performed between these two registers YMM1 and YMM2 using VPCMPEQB instruction, and the result is given in YMM3 register. The next state is indicated by using the value of YMM3 register as an index to another table, namely movef_nextstate table. This parallel

[44] like loop peeling, which consists in removing a number of loop iterations from the loop, and moving them in front of or after the loop, as appropriate. Other types of loop optimization exist such as loop interchange (changes the ordering of nested loops to minimize memory stride), loop unrolling (reduce iterations), etc.

matching process, as shown in Fig 4.10, is repeated until all characters of an input text are processed.

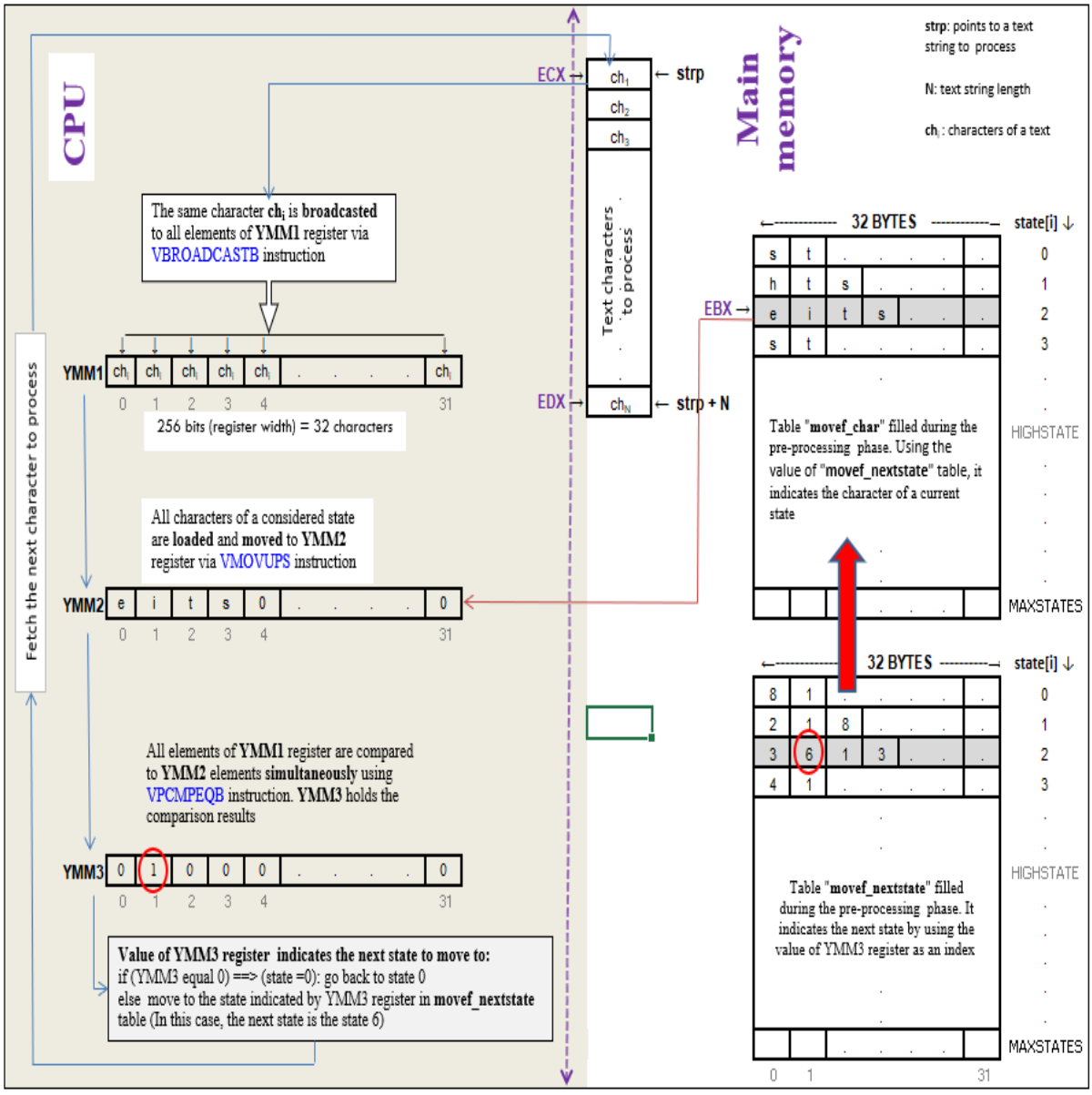


Figure 4.10: Vectorization process of the AC algorithm using AVX2 instructions

Note that when the number of patterns of a given state (max = 32 patterns/state) is not important, only few bytes of YMM2 register will be used. In this case, as shown in the example of Fig 4.10, we can write code to avoid comparing YMM1 register bytes ranging from byte 4 to byte 31 (having "chi" value) to their counterparts in YMM2 register (having zero value), but this is useless since the speed of comparison will not be affected (remains the same) because we are doing SIMD comparison. The vectorized version of the AC program using the *next-move* function is given in listing 4.5.

Listing 4.5: SIMD implementation of the AC algorithm

```
// Here we convert the scalar AC algorithm using the next-move function
// given in listing 3.9 to a vector process

_declspec(naked) void __fastcall AC_NextMove_search_avx2(unsigned char *strp, int N)

{
        // fastcall ==> edx == N     ecx == strp     eax == last param

        _asm push ebp
        _asm mov ebp, esp
        _asm push esi
        _asm push ebx
        _asm push edi

        _asm add edx, ecx   // edx == N + strp
        _asm xor esi, esi   // esi == state == 0
loopj :
        _asm VPBROADCASTB ymm1, [ecx] // ymm1 size = 32 bytes(characters)

        // first we start by coding the loopi code fragment in listing 3.9:
        // (1) for (int i = 0; i < len_movef[state]; i++) { ......}
        // (2) if (c == movef_char[state][i]) { ..(3)....}
        _asm lea ebx, movef_char
        _asm mov eax, esi
        _asm shl eax, 8  //edx == state*256 (256 bytes -line size in "movef_char" array)
        _asm vmovups ymm2, [eax + ebx]

                     //comparison result in ymm3   0xff 0x00 0x00 0xff ..... 0xff 0x00
        _asm vpcmpeqb ymm3, ymm2, ymm1
                     //comparison result in eax       1    0    0    1 ..... 1    0
        _asm vpmovmskb eax, ymm3
                      // i   ==   eax  ==                                    2
        _asm bsf eax, eax

        //for a given state, the number of state transitions <=32 (len_movef[state]<=32)
        _asm jz nonzero  // bsf instruction (if flagzero == 1) then jump

             // (3) {state = movef_nextstate[state][i]; goto nextOK;}
             // state = state*256, 256 bytes - size of a line in "movef_nextstate" array
        _asm shl esi, 8
        _asm add esi, eax              //state = state + i
        _asm lea edi, movef_nextstate  //state = movef_nextstate[state][i];
        _asm mov esi, [edi + esi*4]    //4 bytes -"movef_nextstate" array element size
        goto nextOK;

nonzero :
        _asm xor esi, esi

nextOK : // OutCount[state] += OutPutf[state];
        _asm mov eax, OutPutf[ esi * 4]    //OutPutf must be static
        _asm add OutCount[esi * 4], eax    //OutCount must be static

        _asm inc ecx
        _asm cmp ecx, edx
        _asm jle loopj

        _asm pop edi
        _asm pop ebx
        _asm pop esi
        _asm pop ebp
        _asm ret
}
```

4.5.2 Experimental results

In this section we evaluate the performance of the vectorized AC algorithm by comparing it to the AC algorithm using the *next-move* function as proposed by the authors in [98]. In particular, we compare the running time of the AC_NextMove_search_AVX2 program to the AC_NextMove_ search_ ASM program (the AC algorithm using the *next-move* function, coded in assembly before vectorizing the code as given in listing 3.10). Comparison is done by including the time of the pre-processing phase. The programs have been compiled with Microsoft Visual Studio Ultimate 2013 (version 12.0.21005.1 REL) and experiments were executed on LENOVO laptop, a 1.9 GHz CPU Intel I3-4030U with 4 GB RAM running Microsoft Windows 8.1 Pro 64 bits.

For the evaluation purpose, we first use different files of plain English text, collected from the Gutenberg website (http://www.gutenberg.org), their total size is ranging from 1 MB to 1 GB, and different pattern sizes categorized as follows: (i) *sp* short patterns designing English words of length less or equal 18 bytes and, (ii) *lp* long patterns designing English phrases of length greater than 18 bytes. Patterns were arbitrarily chosen, some of them may exist or not in text files to be processed (especially when matching an important number of patterns).

Each input file is processed by running respectively AC_NextMove_search_ASM and AC_NextMove_search_AVX2 programs. During the matching phase, the text characters of each file is matched against either *sp* and *lp* patterns of different size. All functions of the pre-processing phase are coded in C++ language. Only the search function, which behaves as a finite state string pattern-matching machine is coded in assembly (in case of AC_NextMove_search_ASM program) and codded using AVX2 instructions (in AC_NextMove_search_AVX2 program). The pre-processing phase consists of the following functions: the *goto*, the *failure* and the *next-move* functions that implement respectively algorithms 2, 3 and 4 in [98].

Table 4.2 summarizes the running times of the algorithms. Here we would point out that the vectorized AC algorithm, as implemented here, use only one YMM register (32 bytes length), which limits the number of state transitions to 32[45] (that can be reached once more than 100 patterns are used especially in case of long ones).

[45] 32 is the maximum number of state transitions that YMM register can hold: each state is coded using 1 byte.

File size (KB)	Nb patterns (short pattern sp, long pattern lp)	Total patterns length (Bytes)	T1 (seconds) AC_NextMove_ASM	T2 (seconds) AC_NextMove_AVX2	Speedup (%) [AVX2 /ASM]	Time spent in state 0 (%)
1125 (1 MB)	1 sp	8	0.005340	0.004020	32.84	99.43
	10 sp	53	0.014807	0.005675	160.92	96.90
	100 sp	632	0.043139	0.016071	168.43	76.96
	1 lp	55	0.004349	0.002532	71.76	99.80
	10 lp	578	0.013965	0.004723	195.68	98.95
	100 lp	3914	0.043059	0.018160	137.11	74.27
9690 (10 MB)	1 sp	8	0.036198	0.022953	57.70	99.46
	10 sp	53	0.130331	0.054303	140.01	97.01
	100 sp	632	0.360853	0.142437	153.34	76.94
	1 lp	55	0.041455	0.024482	69.33	99.77
	10 lp	578	0.125042	0.044071	183.73	98.76
	100 lp	3914	0.303146	0.151035	100.71	76.07
102240 (100 MB)	1 sp	8	0.479668	0.368961	30.01	99.49
	10 sp	53	1.444595	0.592323	143.89	97.25
	100 sp	632	4.546556	1.694503	168.31	77.31
	1 lp	55	0.385994	0.193441	99.54	99.75
	10 lp	578	1.399632	0.509368	174.78	98.99
	100 lp	3914	3.659735	1.775760	106.09	76.03
511955 (500 MB)	1 sp	8	2.262958	1.691504	33.78	99.49
	10 sp	53	5.409291	2.541107	112.87	97.26
	100 sp	632	22.958459	8.336736	175.39	77.35
	1 lp	55	2.090559	0.968195	115.92	99.77
	10 lp	578	7.656948	2.818840	171.63	99.05
	100 lp	3914	15.797668	8.034843	96.61	76.22
1048906 (1GB)	1 sp	8	4.163488	2.823843	47.44	99.49
	10 sp	53	12.267914	5.024411	144.17	97.25
	100 sp	632	40.054606	15.947976	151.16	77.34
	1 lp	55	3.991538	1.955866	104.08	99.76
	10 lp	578	12.254003	4.495377	172.59	99.05
	100 lp	3914	32.940271	16.499932	99.64	76.22

Table 4.2: Running times generated while searching different text file size for sets of different patterns

To deal with an important number of patterns, the AC_NextMove_AVX2 program has to be adjusted in a way that it uses a variable to store YMM value when the number of next states (state transitions) exceeds 32, or uses more than one YMM register instead. Another alternative is the use of ZMM registers (AVX- 512) where the width of the register is increased to 512 bits allowing to have up to 64 state transitions/register.

The experimental results reported in Table 4.2 show that the AC_NextMove_AVX2 program performs better than its counterpart AC_NextMove_ASM in all cases. The speed up gained depends on the pattern length and the number of patterns to search for. It is important to highlight the fact that in practice the AC algorithm, as stated by the authors in [98] and confirmed by our experiments, spends most of its time in state 0 (more than 90% of its processing time in most cases). From Table 4.2, we can also notice that the speed-up gained is independent on the file size.

In the second phase of our experiment, we use 10 different patterns (*sp* first then *lp* next having different lengths) to search for, through different file sizes, and we measure the speed-up gained in terms of time processing (in %) when executing the vectorized AC algorithm (see Tables 4.3 and 4.4).

File size (KB)	Total patterns length (Bytes)	Number of generated states (MAXSTATES)	Number of next_states from state 0	Time spent in state 0 (%)	T1 (seconds) AC_NextMove_ ASM	T2 (seconds) AC_NextMove_ AVX2	Speedup (%) [AVX2 / AC_NextMove _ASM]	Speedup (%) [AVX2 / AC_NextMove_ ASM] TheoV
9690 (10 MB)	48	39	5	98.23	0.073203	0.044321	65.17	484,49
	50	38	3	98.42	0.046869	0.029719	57.71	302,42
	53	45	4	98.33	0.082621	0.048733	69.54	410,67
	53	50	6	97.02	0.098311	0.047394	107.43	570,69
	65	51	5	96.44	0.081480	0.048452	68.17	472,95
	82	73	5	93.83	0.118258	0.085747	37.92	454,41
102240 (100 MB)	48	39	5	98.29	0.085187	0.046276	84.08	484,79
	50	38	3	98.55	0.631662	0.412058	53.29	302,62
	53	45	4	98.46	0.836997	0.475502	76.02	411,45
	53	50	6	97.25	1.362945	0.580876	134.64	572,15
	65	51	5	96.41	0.961730	0.558828	72.10	472,60
	82	73	5	93.44	1.574822	1.024999	53.64	452,15
1048906 (1GB)	48	39	5	98.30	11.628367	4.678585	148.54	484,93
	50	38	3	98.56	5.675574	3.606079	57.39	302,65
	53	45	4	98.46	9.870739	5.026897	96.36	411,53
	53	50	6	97.26	12.450564	5.123501	143.01	572,14
	65	51	5	96.47	11.870859	5.670317	109.35	472,86
	82	73	5	93.43	14.372971	9.443510	52.20	452,22

Table 4.3: Running times generated while searching different text file size for sets of 10 different short patterns

File size (KB)	Total patterns length (Bytes)	Number of generated states (MAXSTATES)	Number of next_states from state 0	Time spent in state 0 (%)	T1 (seconds) AC_NextMove_ ASM	T2 (seconds) AC_NextMove_ AVX2	Speedup (%) [AVX2 / AC_NextMove _ASM]	Speedup (%) [AVX2 / AC_NextMove_ ASM] TheoV
9690 (10 MB)	560	558	9	98.72	0.204947	0.061213	234.81	884,82
	466	465	9	98.90	0.186187	0.048387	284.79	886,51
	409	402	8	95.22	0.134060	0.052976	153.06	767,76
	565	557	7	99.32	0.160171	0.028548	461.06	693,44
	373	362	6	94.65	0.078206	0.044593	75.38	577,11
	327	324	8	91.73	0.197331	0.099463	98.40	694,54
102240 (100 MB)	560	558	9	98.97	1.965827	0.623113	215.48	887,10
	466	465	9	99.20	2.178835	0.612167	255.92	889,77
	409	402	8	94.77	2.002379	0.649050	208.51	765,40
	565	557	7	99.54	1.389077	0.344041	303.75	695,20
	373	362	6	94.24	1.107682	0.562100	97.06	575,62
	327	324	8	91.17	1.847791	1.027445	79.84	689,78
1048906 (1GB)	560	558	9	99.02	18.291795	5.667745	222.73	887,74
	466	465	9	99.26	20.000147	5.505017	263.31	890,33
	409	402	8	94.79	17.471766	6.047175	188.92	765,57
	565	557	7	99.59	14.135224	2.825973	400.19	695,48
	373	362	6	94.25	13.016356	5.959448	118.42	575,69
	327	324	8	91.26	16.198667	9.344094	73.36	690,75

Table 4.4: Running times generated while searching different text file size for sets of 10 different long patterns

Here, we give a theoretical value (TheoV) of the speed-up when comparing the running time of the AC_NextMove_AVX2 algorithm against the AC_NextMove_ASM algorithm. This value is calculated by incrementing a variable inside an inner loop in AC_NextMove search function and dividing its value by the number of the processed characters at the end of the outer loop (which is the size of the input file). When considering these values, it is clearly noticeable from Tables 4.3 and 4.4 that the speed-up gained is independent on the file size. In order to get a clear idea about the automaton built, the number of states generated and the number of the states that can be reached from state 0 are given for each execution.

Experimental results clearly show that the vectorized version of the AC algorithm, using the *next-move* function, yields better performance in terms of speed whose running time ranged from two (×2) up to eight (×8) of its original counterpart (not vectorized). The speed-up gained depends on the number of the patterns to search for and their length but not on the file size.

The use of vectorization results on a higher return in performance and efficiency that depend on the code structure. But, once again, the programmer must examine at first the structure of the code to vectorize in depth to identify parts (code fragments) that offer vectorization opportunities, and make the data structures in the code nearly fit the structure built into the hardware.

Chapter 5

5 Improving the Signature Scanner using an Optimized Aho-Corasick algorithm

A significant part of the AV software execution time is dedicated to the scanning process: objects are scanned against a set of malware signatures in order to determine if they are safe or malicious. As it is important that the scanning process be as fast as possible, efforts must be made to minimize the time spent during matching these signatures. Considering AV databases, and particularly those of ClamAv software that have been used as experimental signature set in many researches related to malware scanning solutions [123, 124, 125, 126]. Most of the signatures contained in these databases are MD5[46] hashes of files, and the rest are basic string, regular expression (regex) patterns, and other type of signatures related to archives and spam e-mails. In their work on memory-based multi-pattern signature scanning for ClamAv antivirus [126], Dien et al, analyzed ClamAV database (database of September 2013 - ClamAv version 0.98.4) and found that 93.96% of the signatures are MD5 hashes but only account for around 8% of scanning time, and 4.7% are basic string and regex patterns (both types are byte-stream signatures) that account for 91.80% of scanning time, as shown in table 5.1.

Signature types	Basic string	Regex patterns	MD5 hashes	Other	Total
No. Signatures	88,885	9,670	2,277,804	47,866	2,424,225
No. Sig. ratio (%)	3.67%	0.40%	93.96%	1.97%	100%
Scan time ratio (%)	63.86%	27.94%	8.20%		100%

Table 5.1: Analysis on ClamAv database [126]

As we can see from Table 5.1, the largest portion of malware signatures are MD5 hashes. However, scanning against this type of signatures does not take time, and simply realized by

[46] The Message-Digest Algorithm 5 (MD5) is the most commonly used hash function for malware analysis, though the Secure Hash Algorithm 1 (SHA-1) and SHA-256 are also popular. MD5 hash is 128-bit string or 32 hexadecimal digits long. The idea behind the algorithm is to convert any size or length data (text or binary) to fixed size "hash value" output.

applying an MD5 hash function to the file (usually a specific section in a PE[47] file) being scanned, and matched the function result against all ClamAV database hashes of malware signatures. To implement an MD5 malware scanner, signature hashes can be collected from https://virusshare.com/hashes.4n6 web site, and get their corresponding names from the VirusTotal web site (Owned by Chronicle Security, a part of Google cloud that provides free, multi-scanner malware insights, and automatically share types of malware with the security community) at: https://www.virustotal.com/gui/file//detection. The GitHub repository is also a good choice since it provides both malware names and their MD5 hash signatures https://github.com/ytisf/theZoo/blob/master/malwares/Binaries.

Malware Name	Signature (MD5 Hash)	Aliases (Detected by AV engines as)	File Type
Duqu2.zip	e64d31ef596e86997ca0fffcfb3d1ce8	Trojan.Duqu.E (Arcabit), W64/Duqu.AC!tr (Fortinet), Malicious "high Confidence" (Endgame)	Zip
conficker	566119e4e5f4bda545b3b8af33c23698	Win.Worm.Conficker-246 (ClamAv), W32/Confick-A (Sophos), Worm.Win32.Kido.Ia75 (AegisLab), Win32:Kido-D [Wrm] (AVG & Avast)	Win32 DLL
Ransomware.Petya.exe	e8fb95ebb7e0db4c68a32947a74b5ff9	Trojan.Ransom.AUC (Arcabit), W32/Petya.EOB!tr.ransom (Fortinet), Trojan.GenericKD.33821777 (Ad-Aware/ eScan/ BitDefender/ FireEye/ GData)	Zip
Email-Worm.Win32.Bagle.aav	26ad30c1bb65a193a5f60f7e36c7f004	Win32/PSW.Agent.NJE (ESET-NOD32), Win32:Trojan-gen (Avast), Email-Worm.Win32.Bagle.aav (Kaspersky/ ZoneAlarm by Check Point)	Win32 DLL
Ransomware.Wannacry_Plus	30fe2f9a048d7a734c8d9233f64810ba	Ransom:Win32/CVE-2017-0147.A (Microsoft), Ransom.WannaCrypt (Malwarebytes), Ransom_WCRY.SMALYM (TrendMicro), Exploit.CVE.Win32.1766 (Zillya)	Win32 DLL
unpacked.mem	108756f41d114eb93e136ba2feb838d0	Trojan/Win32.Satana.C1492908 (AhnLab-V3), Trojan-FKOP!108756F41D11 (McAfee), W32.Ransomware.Satana (Webroot), Trojan-Downloader/W32.Upatre.73728.E (TACHYON)	Win32 EXE
index.html	8834639bd8664aca00b5599aaab833ea	HTML/Infected.WebPage.Gen2 (AntiVir), Trojan.Exploit.Iframe.AP (Bitdefender), IFrame.gen (F-Prot), Exploit.HTML.IFrame-6 (ClamAv)	HTML
Trojan.Sinowal.zip	9538a123b2e1489b39c5f86be4c11010	Trojan.Zusy.DEEB7 (Arcabit), W32/Sinowal.UJW!tr.bdr (Fortinet), Trojan.Win32.MaosBoot.dovomw (NANO-Antivirus)	Zip

Table 5.2: Malware MD5 signatures

[47] The Portable Executable (PE) format is a file format for executables, DLLs, FON Font files, etc., used in both x86 and x64 versions of Windows operating systems.

Table 5.2 gives some examples of malware MD5 signatures. However, it should be noted that the same malware can be identified differently when submitted to different AV engines (aliases). For example, the "Ransomware Wannacry_Plus" can be identified as *Ransom:Win32/CVE-2017-0147.A* (Microsoft), *Ransom.WannaCrypt* (Malwarebytes), or *Ransom_WCRY.SMALYM* (TrendMicro) to cite just a few. VirusTotal can provide a complete list of different aliases attributed to the same malware. So why different AV engines find the same malware as different? Because of the different ways that AV vendors seek out and classify malware. Each AV vendor uses different conventions to name every malware sample it discovers. The majority of malware samples are named based on the information extracted from the samples, such as their functionality (Downloader, Banker), the author's name (Morris worm), or they are given a completely generic name (Agent, Threat or Malware). In some cases, the malware authors name their creations themselves (Petya ransomware named by its creator "Janus Cybercrime Solutions" or shortly Janus).

As noted earlier from Table 5.1, MD5 signatures consume around 8% of scanning time despite their majority in AV database. Therefore, we ignore this kind of signatures and focus on byte-stream signatures since they consume more than 90% of the scanning time in order to reduce the overall scanning time of the AV software.

5.1 Scanning for byte-stream signatures

In contrast to scanning for MD5 hashes that does not take time, scanning for byte-streams (basic strings and regular expressions) takes a significant amount of time since the scanning process is carried out for *inside the file*. Therefore, the scanning time that can be optimized is the one related to scanning against this type of signatures.

The basic string signature, a simplest form of a malware signature, is essentially byte sequences (one or more constant sequence of bytes) common to a particular malware file that normally should not appear in other safe files. Table 5.3 gives few examples of such signatures. Each entry of the table consists of the malware name, followed by a short description of the malware, and its signature in hexadecimal format to detect its presence.

The basic string signatures, used in the experience, are especially collected from the German University of Freiburg (https://ad-teaching.informatik.uni-freiburg.de/cplusplus-ss2012/virus-signatures.txt) and the online United Kingdom magazine "Virus Bulletin" (https://www.virusbulletin.com/uploads/pdf/magazine/) websites.

Unlike MD5 signatures that are of fixed length (32 hexadecimal digits), basic string patterns can be of any length as shown in Table 5.3.

Malware name	Signature (Basic string pattern)	Signature size (hex digits)	Malware description
Paraguay.2858	B440B92A0BBA350BCD2126C74515000026C745170000B440	48	A polymorphic, stealth, 2858-byte appender with the texts 'VIRUS', 'PARAGUAY', 'Ver. 3.0', 'Programmed by Int13h, in Paraguay, South America.ANTI-VIR.DAT', 'CHKLIST.CPS', 'AVP.CRC' and 'C:\WINDOWS' 'CHKLIST.MS'. Infected files have their time-'C:\COMMAND.COM', stamps set to 60 seconds. The malware infects COM & EXE files and is memory-resident after infection.
Trojan.Downloader.Agent.AP.4	ffedaf4578706c6f10725c7b35383744424632442d393134356bfd7ffb2d3463396509 3243322d314639353344413717	96	The Trojan.Downloader.Agent.AP can damage the whole Windows System and make other install program unresponsive. The Trojan threat is used to hamper more and to steal useful resource from the infected Windows System. On every web browser there is option to save password, this is done for the user convenience. The hacker will attack on those web browser, mainly on the default one to collect those saved password.
Flooder.Win32.FloodBots.21	730d0a00466c6f6f64626f747320466c6f6f646572205620322e3120506f7274656420746f2057696e393520627920447242617264 6f	108	A type of bot malware that allows a hacker to take control of the affected computer. Once this happen, it can perform a variety of automated tasks that can display strange messages, slow down the system, or even crash it. With Flooder.Win32.FloodBots.21, a hacker can steal confidential information by sending spam, refusing certain services on the Internet and even committing "click fraud".
JS.Trojan.WindowBomb.A	77696e646f77222b77696e646f776e756d6265722c2277696474683d312c6865696768743d312c726573697a61626c653d6e6f22290d0a2020202020202020207769 6e646f776e756d6265722b2b3b0d0a2020202020207d0d0a7d0d0a	184	JS.WindowBomb is a Trojan that opens a very large number of Web browser windows. In some cases, the mouse stops working and the time on the system clock keeps changing.
TrojanDownloader.Win32.G-Spot.15	696320736179732044f2100000000fffff fff070000004d7367426f783b00558bec51 538945fc8b45fce8e4d5ffff33c055683d6 1400064ff306489206a006a00a1e09c4000 e8d8d5ffff508b45fce8cfd5ffff506a00e 8b3defffff85c0740433dbeb02b30133c05a 595964891068	222	A backdoor Trojan written in Delphi affecting Microsoft Windows operating systems. It uses a client/server relationship, where the server component is installed in the victim's system and the remote attacker has control of the client. The server attempts to open a port, typically TCP port 52978, to allow the client system to connect. Gspot could allow a remote attacker to gain unauthorized access to the system.

Table 5.3: Malware basic string signatures

A regular expression, or regex signature is an extension of the basic signature with various wildcards. For detecting complex file infectors (polymorphic malware samples) that change their physical forms (binary signatures) after every infection, regex signature support is necessary. Table 5.4 shows some regex features allowed in the ClamAV pattern set. For example, the wildcard "*" in ClamAv means matching any number of bytes: the pattern *aac*bbc* means that there can be 0 or more bytes between the symbols *aac* and *bbc*. Formally, "*" virtually separate a signature into two parts, e.g. *aac*bbc* is treated as two sub-signatures *aac* and *bbc* with any number of bytes between them. Each sub-signature must include a block of two static characters somewhere in its body [127].

Symbol	Meaning
??	match any byte, i.e. wildcard byte
a?	match a high nibble (the four high bits)
?a	match a low nibble (the four low bits)
(aa|bb)	match aa or bb
!(aa|bb)	match any byte except aa or bb
*	match any number of bytes
{n}	match n bytes
{-n}	match n or less bytes
{n-}	match n or more bytes
{n-m}	match between n and m bytes (m > n)

Table 5.4: Some regular expression features allowed in ClamAV malware signatures

The signature for *Cascade.1704.E* virus "*fa8bece800005b81eb????{1-8}8db7????bc????31343124464c75f8*" is defined with some wildcards *({n-m},??)*. Each symbol *"??"* can be replaced with any 1-byte data and the symbol *"{n-m}"* can be replaced with within in range from *n* to *m* number of bytes. When searching for this regex signature, the matching engine must be able to match separate patterns *"fa8bece800005b81eb", "8db7", "bc*", *"31343124464c75f8",* and verify the distant constraints between these patterns. More regex pattern examples are given in Table 5.5.

For instance, the AC algorithm in its original or vectorized forms can only handle basic string patterns, while certain malware patterns such as worm signatures could be specified by regular expressions. Of course, it can be advanced to systematically construct a finite state pattern-matching machine, which can indicate the ending position in a finite input string for malware patterns that are specified by regular expressions. Tsern-Huei Lee, proposed a generalized AC algorithm for signature based anti-virus applications, to handle patterns based on regex expressions. They addressed the problem of constructing a finite state pattern-matching machine for a set of strings *W* together with *n* simple regular expressions RE_1, RE_2, *…, and* RE_n. Each RE_k expression must at least include one symbol among: "*" (match any number of symbols), "?" (match any symbol), or "{min, max}" (match minimum of min, maximum of max symbols) used in regular expression definition. Like the AC algorithm, its construction procedure yields a pattern-matching machine dictated by the *goto*, *failure*, and *output* functions. The difference is that its constructed pattern-matching machine consists of multiple separated goto graphs connected by failure functions [128].

Malware name	Signature (Regex pattern)	Malware description
Trojan.Downloader.Small-382	0bc074328bd868????????68000400006a006a0068????????50e8????????0bc0740d50e8????????5850e8????????53e8????????c9c20400c8000000837d0802751550546a006a0068691100106a006a00e8????????58c9c20800	A trojan downloader that download and execute arbitrary malware from a site that is predefined by the trojan author.
Worm.Bagle.BB	eb1064646464646464646464646464646464eb??0d0a6d6b6b6e6b6867??????????6768*67686a67666866646a????????????????686a	A mass-mailing worm that targets computers running certain versions of Microsoft Windows.The worm sends itself as an attachment to e-mail addresses that it finds on the infected computer. It also spreads through file-sharing networks. The worm opens a backdoor and monitors a TCP port that allows remote attackers to execute files on the infected computer.
VBS.Pleo.A	457865637574652822{-256}3d417363284d696428{-256}20586f7220{-3}29222b766263726c662b226e6578743a	It carries out damaging actions on the affected computer. It captures certain information entered or saved by the user, with the corresponding threat to privacy. It causes the loss of information stored on the computer, and affects the network to which it's connected and other remote sites. Affected platforms: Windows XP/2000/NT/ME/98/95
Exloit.PDF.CVE_2008_2992-5	25504446*7574696c2e7072696e7466{-3}28222532(37|38|39)(30|31|32|33|34|35|36|37|38|39)66	Basically, the malicious PDF exploited CVE-2008-2992, a vulnerability, or a stack-based buffer overflow in Adobe Acrobat and Reader 8.1.2 and earlier that allows remote attackers to execute arbitrary code via a PDF file that calls the "util.printf()" JavaScript function with a crafted (invalid) format string argument. Successful exploitation of the vulnerability allows attackers to gain access to vulnerable systems.
Cascade.1704.E	fa8bece800005b81eb????{1-8}8db7????bc????31343124464c75f8	The Cascade virus (also known as *Herbstlaub* in Germany) is a prominent computer virus written in assembly language. It infected .COM files and had the effect of making text on the screen fall (or cascade) down and form a heap at the bottom of the screen. It was notable for using an encryption algorithm to avoid being detected.
Trojan.ZBot-12528	80f???67e30?f1[0-14]60	A Trojan Horse that belongs to the Zeus family of malware, targets Microsoft Windows and is often used to steal financial data (online banking credentials login details). It mainly spreads via email but can also install with a drive-by infection when the user visits a compromised webpage.

Table 5.5: Malware regex signatures

5.2 The signature scanner tool

The scanner tool consists of a malware signature database and a malware detection engine.

- *The signature database* also known as the *dat* file contains the uniquely identifiable signatures of all collected malware. To build the database, we have collected more than 1000 malware basic string signatures from the web that are in HEX format without commas or blank spaces. In our implementation, each malware record has two fields: a name of the malware and its signature as shown in Table 5.6.

Malware name	Malware signature in hexa-decimal format
Exe2Win-182	9c00007517ba9e00b8013dcd2193b440ba0001b9b600cd21b43ecd21b44fcd2173dac3
Perl.RemoteLog.A	7072696e7420223c484541443e3c5449544c453e437275656c2d496e74656e74696f6e7a20436769205365727665722046c6f67676572207632 3c2f5469746c653e5c6e223b
VirTool.Bat.BTG.03	a20000004a01002c02b11b00d400424154434820545 24f4a414e2047454e455241544f5220302e30330d4300f4002400b2200d00b02b202b202b202b202b
Trojan.PSW.KeyLogger.10	436f6e737472756374000000c0000004e006f007400690063006500000000001400000020006900730020004f006e006c006900 6e00650000000000000000006000000066007200 6f006d003d004b00650079004c006f006700670065007200260066
W97M.Melissa.AB	427265616b556d4f66664153 6c6963652e5375626a656374203d2022416c65786 1204c6f7665727320222026204170706c69636174696f6e2e557365724e616d65
Win32.Evyl.b	2740004b75f78d9d3c2940005756803b2e0f84e50200008a034388074784c075f6f6851029400010741066c747ff5c00e898ffffffe9c20200008b47fb0d002020203d2e6578650f85af0200008d854e2a40006a006a006a036a006a0068000000c050ff95dc2640008985b827
Worm.P2P.SpyBot.3	505249564d5347202573203a2573000a0025693a2025730 04b657973206c6f6767696e6720746f2025735c25730000 00462f4156204b696c6c6572005072

Table 5.6: Malware records

- *The malware detection engine* is implemented using the vectorized AC algorithm that enables scanning huge amounts of files in a very short period of time. It scans files of all types for malware excluding boot sector and partition table, which explains the absence of a malware type field from a malware records table. The scan is done by searching the whole files for entire signatures.

In practice, several techniques are used to reduce the performance cost of signature scanning. These techniques are made available by expanding the signatures to include additional information such as the type of an object to which is applied, the areas being examined withing the object, and the size of the areas being scanned [129] as shown in Fig.5.1.

To keep scan times down, two main techniques are usually deployed:

- First, malware signatures are classified by the type of infection they represent. This may include, boot sector, partition table, executable file or data file types. When a particular object is scanned, this is usually done through a process of elimination. Only the signatures that pertain to that object type will be used. For example, boot sector

signatures cannot be applied to executable files. The more signatures the AV engine has to match, the longer the scan is going to last. Limiting scanning by object type can significantly reduce the search space without impacting the effectiveness of a scanner.

- Second, when a particular file is scanned, a flag is used to denote if the engine only searches (i) at some specific file locations that are most likely to contain the signature and particularly at the entry point (the location in the file where program execution begins), (ii) for nibbles of signatures or (iii) searches the whole file if nothing is found. In case of a nibble match, further matching with the respective entire signature is curried out. Scanning an entire file for a sequence of bytes is not feasible, but searching for nibbles of signatures at certain specific file locations such as the entry point of an executable program or its header section accelerates the matching process and reduces the amount of a scan [129].

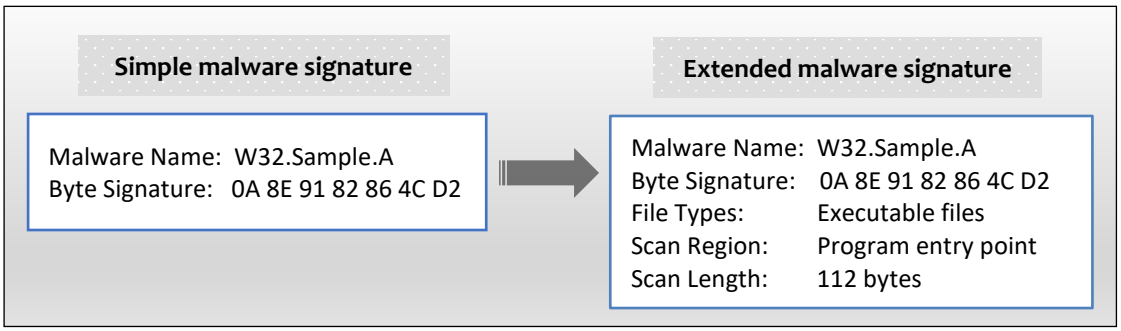


Figure 5.1: Expanding malware signature [129]

5.3 Performance analysis

Here, we keep the same searching functions, which behave as a finite state string pattern-matching machine, used in chapter 4 for the evaluation of SIMD implementation of the AC algorithm. The only thing that changes is that the files to be analyzed can be of various types: executable or data files, opened in binary mode, and matched against all malware patterns in the database.

To evaluate the performance of the scanner tool engine implementing the AC_NextMove_search_AVX2 function, we test it against the one implementing the AC_NextMove_search_ASM function for various malware signatures by varying the malware database size as a first step. Then we test the two programs by varying the malware signature size.

The programs have been compiled with Microsoft Visual Studio Ultimate 2013 (version 12.0.21005.1 REL) and experiments were conducted on DELL Inspiron 15 (3000 series) laptop, a 7th Generation Intel® Core™ i5-7200U Processor (3MB Cache, base clock speed 2.5 GHz and TurboBoost up to 3.1 GHz) with 6 GB RAM running Microsoft Windows 10 pro 64 bits.

5.3.1 Performance according to the malware database

In this test, we start the two programs to scan different files occupying 1 GB disk space by varying the malware signature database size. We keep the previous malware signatures and we add new ones to the database before running the next experience. Table 5.7 and Fig. 5.2 show the performance of the scanners.

The results reported in Table 5.7 and Fig.5.2 show that the scanner using the vectorized version of the AC algorithm performs better than its counterpart using the scalar version of the AC algorithm when varying the malware database size.

Database size (Number of malware signatures)	Size of malware signatures (Bytes)	T1 (seconds) AC_NextMove _ASM	T2 (seconds) AC_NextMove _AVX2
10	960	5.56971	3.15060
20	2036	7.30916	3.11254
50	5306	7.07618	3.02233
100	14094	10.32911	4.24513
500	75618	16.61124	6.18173
1000	161972	24.03371	8.59318

Table 5.7: Execution times generated when varying the malware database size (Amount of scan = 1GB)

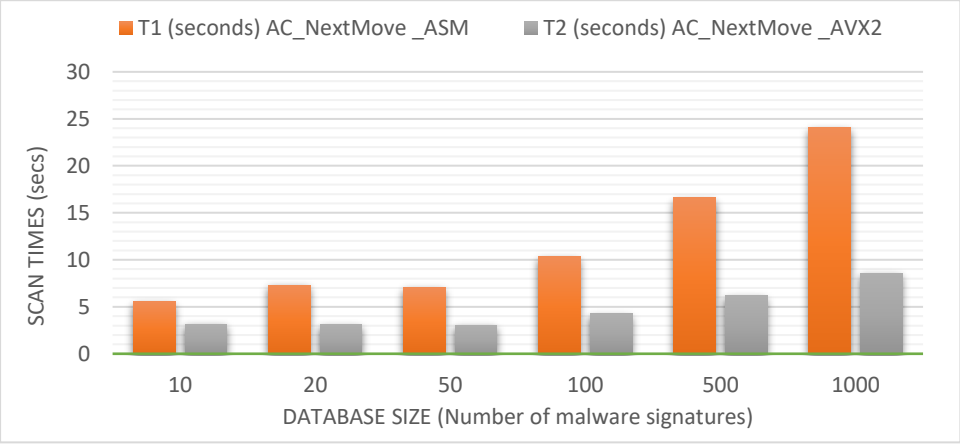


Figure 5.2: Performance chart according to the database size

In the second phase of the experience, we repeat the same experience by varying the amount of scan (5 GB and 10 GB). Table 5.8 summarizes the running times of the programs including those of the previous experience (1 GB) and gives the increase in the scanning speed (speedup) when comparing these programs: the scanner using the vectorized version of the AC algorithm and the one using the scalar version of the AC algorithm.

The results of this experience, as reported in Table 5.8, show that the vectorized scanner outperforms its counterpart using the scalar version of the AC algorithm when varying the amount of scan.

Total size of files to scan (GB)	Database size (Number of malware signatures)	Size of malware signatures (Bytes)	T1 (seconds) AC_NextMove _ASM	T2 (seconds) AC_NextMove _AVX2	Speedup (%) (AC_AVX2/ AC_ASM)
1			5.56971	3.15060	76.78
5	10	960	27.74999	15.62186	77.64
10			56.60074	31.98025	76.99
1			7.30916	3.11254	134.83
5	20	2036	36.61182	15.52961	135.75
10			74.27199	31.49397	135.83
1			7.07618	3.02233	134.13
5	50	5306	35.31963	15.10217	133.87
10			71.80279	30.64606	134.30
1			10.32911	4.24513	143.31
5	100	14094	46.00147	18.93131	142.99
10			97.01154	39.88751	143.21
1			16.61124	6.18173	168.71
5	500	75618	94.02869	34.97731	168.82
10			193.09861	72.02133	168.11
1			24.03371	8.59318	179.68
5	1000	161972	109.84837	39.19973	180.23
10			226.83957	81.07411	179.79

Table 5.8: Execution times generated when varying the malware database size (Amount of scan = 1GB, 5GB & 10GB)

Always considering the results reported in Table 5.8, the execution times generated by the third experience (database size = 50) when running the scanner using the scalar AC algorithm are shorter than the execution times generated by the second experience despite the number of malware signatures, which is much lower (database size = 20). This is due to the pattern-matching machine behavior (the automaton built from the data of the signatures), and precisely depends directly on the content of state S0, where the automaton spends most of its

time [98]. On the other side, when running the scanner using the vectorized AC algorithm, it can be noted that the execution times generated always by the third experience are the shortest times of all experiences. This is also due to the pattern-matching machine behavior and the contents of YMM2 register during the execution: the number of hexadecimal symbols in YMM2 register (the number of states) to be matched simultaneously to the hexadecimal symbol in YMM1 register (the symbol being processed that comes from the file under inspection) as shown in Fig 4.10.

Finally, from Table 5.8, we notice that the speedup, or the performance gained, is independent on the file size.

5.3.2 Performance according to the malware signature size

In this test, we run the two programs to scan 1 GB of different files by varying the malware signature size (only one signature is in the malware database at a time). We increase the signature size before running the next experience (by choosing another signature). Table 5.9 shows the performance of the scanner programs.

Size of a malware signature (Bytes)	T1 (seconds) AC_NextMove _ASM	T2 (seconds) AC_NextMove _AVX2	Speedup (%) (AC_AVX2/ AC_ASM)
32	2.35828	1.82128	29.48
62	2.36226	1.82119	29.71
90	2.36146	1.81974	29.77
146	2.35684	1.82178	29.37
216	2.35879	1.82600	29.18
270	2.35608	1.81693	29.67
334	2.38097	1.81963	30.85

Table 5.9:Execution times generated when varying the malware signature size (Amount of scan = 1 GB)

The results reported in Table 5.9 show that the scanner using the vectorized AC algorithm outperforms (speedup ≈30%) the scanner using the scalar version of the algorithm when scanning files looking for just one malware signature even when varying its size (increasing the malware signature size). The gain in speed processing supplied by vectorization (three-operand AVX instructions) is more important compared to the one provided by executing two-operand assembly instructions in spite of:

- Accessing memory using AVX instructions is more time consuming (32 bytes are used to read from/ or write to memory in order to process one hexadecimal symbol) compared to the scalar AC algorithm (1 byte is used to perform the same operation);

- Instruction execution times in the previous extensions of the x86 instruction set are usually faster than those of the AVX extensions (AVX instructions usually have a long latency[48]) [130];

In the study of malware signatures, this case is of no importance since an average AV database stores approximately hundreds of thousands of basic string signatures.

[48] Latency is the number of processor clocks it takes for an instruction to have its data available for use by another instruction.

Chapter 6

6 Conclusion

Although anti-malware technologies have evolved over the years, they still include classic signature matching technique as one of their main layers of defense against malware. The anti-malware software would monitor all the data coming from outside and entering the system. It would analyze and scan the contents to check if the source code or hashes in the files or packets match with any of the patterns of known threats in the repository. This fast-matching technique based on byte patterns, simple to implement and update for security vendors, complements more sophisticated detection methods and more advanced security threats analysis.

When malware arrives in AV company, it is first analyzed by automated analysis systems or by security experts when necessary. If the analysis revealed that it is actually a malware, a proper signature of the sample is extracted and added to the AV database malware signatures. The signature patterns are series of machine code bytes, usually known as “byte-streams” or “strings”. They can be basic (simple strings), or fairly complex including wildcard characters. They are consumed by simple pattern-matching techniques (for example, a specific sequence of bytes), cyclic redundancy codes CRCs (checksums), or cryptic hashes (MD5/SH-1/ SHA-256).

When a particular object has to be scanned, the AV engine matches its content against all appropriate entries in the AV database signatures. If it matches one signature, then the engine knows exactly which malware it is and which procedure has to be performed in order to contain the malicious object and do disinfection.

Signature-based detection relies on the definition files that contain signatures for viruses, worms and other types of malware that have been encountered in the wild. It cannot defend against threats unless their signatures are already added to the database. For this, AV software maintained its repository of known and identified threats; it automatically downloads new, updated definition files once a day or even more often. The effectiveness of AV products is measured in terms of the AV vendors having the maximum malware signatures, and their ability to include new signatures and make them available to their client systems. In the same

time, AV researchers constantly trim unnecessary signatures to cope with the database size that goes on increasing day by day, and to keep the product's scan time down. Even though, signature-based detection does not provide complete defense, it guarantees protection against millions of known and active threats.

Typically, AV software spends a bulk of its time scanning data streams against a large number of known signatures in the database. Its performance relies on the pattern-matching algorithm that it implements. Therefore, the used algorithm must be fast enough and scalable (able to maintain its efficiency when dealing with large databases). Techniques like partial or approximate matches may also be used to reduce the scanning time. In our work, we have analyzed the problem of malware detection based on signature scanning technique and showed how it can be implemented in practice. We have proposed a vectorized AC algorithm, which builds a fast finite state machine for matching occurrences of malware signature patterns simultaneously in linear time. The matching process is done with SIMD acceleration, which adds a form of data parallelism to the AC code (where one instruction or operation can be applied to a vector, or series of adjacent data). Others approaches to achieve matching performance exist, such as those based on specialized hardware, like ASICs and FPGAs, but are usually tied to a specific implementation, and very hard to code or modify.

The vectorized AC algorithm is implemented with Intel® AVX2 instruction set (Haswell New Instructions), an extensions to the x86 instruction set architecture released in 2013 with the fourth generation Intel® Core™ processor family and later supported by AMD processors (Excavator in 2015). This choice is well justified because AVX2 provides support for integer computation, thing that was lacking in the legacy SSE and AVX (introduced with Intel® Sandy Bridge processor in 2011). This AVX2 property allows to apply SIMD instructions to the machine states generated during the pre-processing phase of the AC algorithm, which can only be encoded as integer values.

In Chapter 4, we gave in-depth details of the vectorization process of the AC algorithm, in which the AC code is converted from its scalar implementation, which does an operation one pair of operands at a time, to a vector process, where a single instruction (SIMD operation) can refer to a vector. The important thing to consider before converting the scalar-oriented code is to identify vectorization opportunities from its code fragments, especially determining the loops that can be parallelized. Still, not all loops can or will be parallelized. This is exactly why we have considered the version of the AC algorithm using the *next-move* function, whose source code seems vectorizable. When vectorization is applied on computing systems that

support such actions, it results in more efficient data processing and a high application performance.

Vectorization now is more important than ever. Generating vectorized object code utilizing the AVX/SIMD instructions is a great way to accelerate software applications. For optimal performance boost, this hardware optimization technique can be combined with multithreading, where multiple activities can proceed concurrently in the same program for maximum utilization of the CPU. Both techniques are important when optimizing an application. Benchmarks have shown that combining theme can result in a higher performance boost (sometimes more than ×100) over just using vectorization or multithreading alone [131]. So, the next step toward speeding up the matching process of the AV scanner is multithreaded parallel implementation of the vectorized AC algorithm.

Bibliography

[1] G. McGraw, *Software Security: Building Security In.* Addison-Wesley Professional, 2006, pp 25-26.

[2] Symantec Corporation, "Internet Security Threat Report, Vol. 21", Symantec Corporation World Headquarters, Mountain View, CA 94043 USA, 2016. [Online]. Available: https://docs.broadcom.com/doc/istr-16-april-volume-21-en.

[3] M. Garnaeva, F. Sinitsyn, Y. Namestnikov, D. Makrushin and A. Liskin, "Kaspersky Security Bulletin: OVERALL STATISTICS FOR 2016", Kaspersky Lab, 2016. [online]. Available at: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07182326/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf.

[4] J. Koret and E. Bachaalany, *The Antivirus Hacker's Handbook.* Indianapolis: John Wiley & Sons, 2015, pp. 77-86.

[5] A. Edward Danso, J.B. Hayfron-Acquah, D. Dominic, A. Michael and A. Brighter, "Camouflage Worm Detection and Counter-Measures", *International Journal of Inventive Engineering and Sciences (IJIES)*, vol. 2, no. 9, 2014.

[6] J. Aycock, *Computer Viruses and Malware*. Boston, MA: Springer Science+Business Media, LLC, 2006.

[7] J. Love, "A Brief History of Malware—Its Evolution and Impact", *Lastline*. [Online]. Available: https://www.lastline.com/blog/history-of-malware-its-evolution-and-impact/. [Accessed: 11- Jul- 2018].

[8] "The History of Malware, from Pranks to Nuclear Sabotage", *Digital Trends*. [Online]. Available: https://www.digitaltrends.com/computing/history-of-malware/. [Accessed: 22- Jul- 2018].

[9] F. Cohen, "Fred Cohen & Associates", *All.net*. [Online]. Available: http://all.net/books/virus/ [Accessed: 22- Jul- 2018].

[10] "The History of Computer Viruses", *OpenMind*. [Online]. Available: https://www.bbvaopenmind.com/en/technology/digital-world/the-history-of-computer-viruses/. [Accessed: 22- Jul- 2018].

[11] "Global Ransomware Damage Costs Predicted To Hit $11.5 Billion By 2019", *Cybercrime Magazine*. [Online]. Available: https://cybersecurityventures.com/ransomware-damage-report-2017-part-2/. [Accessed: 24- Jul- 2018].

[12] "The very first viruses: Creeper, Wabbit and Brain.", *Info Carnivore*, 2010. [Online]. Available: https://infocarnivore.com/the-very-first-viruses-creeper-wabbit-and-brain/. [Accessed: 30- Jul- 2018].

[13] "The first computer virus was designed for an Apple computer, by a 15 year old", *Quick Heal Blog | Latest computer security news, tips, and advice*. [Online]. Available: https://blogs.quickheal.com/the-first-pc-virus-was-designed-for-an-apple-computer-by-a-15-year-old/. [Accessed: 30- Jul- 2018].

[14] "Brain -The first computer virus was created by two brothers from Pakistan. They just wanted to prevent their customers of making illegal software copies", *The Vintage News*, 2016. [Online]. Available: https://www.thevintagenews.com/2016/09/08/priority-brain-first-computer-virus-created-two-brothers-pakistan-just-wanted-prevent-customers-making-illegal-software-copies/. [Accessed: 03 - Aug- 2018].

[15] "The History of Cyber Security — Everything You Ever Wanted to Know ", *SentinelOne*, 2019. [Online]. Available: https://www.sentinelone.com/blog/history-of-cyber-security/. [Accessed: 07- Dec- 2019].

[16] "Morris Worm Turns 30 by Dinesh Abeywickrama & Inoshi Ratnasekera", *Siyalla News*, 2019. [Online]. Available: https://www.siyalla.info/2019/03/03/the-morris-worm-turns-30/. [Accessed: 07- Dec- 2019].

[17] "The Michelangelo Virus, 25 Years Later", *Trend Micro - Cybercrime & Digital Threats*, 2017. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/the-michelangelo-virus-25-years-later. [Accessed: 06 - Aug- 2018].

[18] "What is Melissa virus? - Definition from WhatIs.com", *SearchSecurity*. [Online]. Available: https://searchsecurity.techtarget.com/definition/Melissa-virus. [Accessed: 07 - Aug- 2018].

[19] "Love Bug: The Virus That Hit 50 Million People Turns 15", *Vice.com*. [Online]. Available: https://www.vice.com/en_us/article/d73jnk/love-bug-the-virus-that-hit-50-million-people-turns-15. [Accessed: 07 - Aug- 2018].

[20] G. Cluley, "Memories of the Anna Kournikova worm", *Naked Security*. [Online]. Available: https://nakedsecurity.sophos.com/2011/02/11/memories-anna-kournikova-worm/. [Accessed: 09 - Aug- 2018].

[21] "The Slammer Worm", *Spamlaws.com*. [Online]. Available: https://www.spamlaws.com/slammer-worm.html. [Accessed: 09 - Aug- 2018].

[22] "Cabir: World's First Wireless Worm", *Technewsworld.com*. [Online]. Available: https://www.technewsworld.com/story/34542.html. [Accessed: 13- Aug- 2018].

[23] "Sony BMG Copy Protection", *Malware Wiki*. [Online]. Available: https://malware.wikia.org/wiki/Sony_BMG_Copy_Protection. [Accessed: 13- Aug- 2018].

[24] "Zeus Virus", *usa.kaspersky.com*. [Online]. Available: https://usa.kaspersky.com/resource-center/threats/zeus-virus. [Accessed: 13- Aug- 2018].

[25] Unisys Stealth Solution Team, "*Zeus Malware: Threat Banking Industry*", Unisys Corporation, USA, 2010. [Online]. Available: https://botnetlegalnotice.com/citadel/files/Guerrino_Decl_Ex1.pdf. [Accessed: 13- Aug- 2018].

[26] "Koobface", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Koobface. [Accessed: 16- Aug- 2018].

[27] "BBC NEWS | Technology | Trojan virus steals banking info", *News.bbc.co.uk*. [Online]. Available: http://news.bbc.co.uk/2/hi/technology/7701227.stm. [Accessed: 16- Aug- 2018].

[28] "News from the Lab Archive: January 2004 to September 2015", *Archive.f-secure.com*. [Online]. Available: https://archive.f-secure.com/weblog/archives/00001393.html. [Accessed: 16- Aug- 2018].

[29] R. Naraine, "Botnet hijack: Inside the Torpig malware operation", *ZDNet*, 2009. [Online]. Available: https://www.zdnet.com/article/botnet-hijack-inside-the-torpig-malware-operation/. [Accessed: 16- Aug- 2018].

[30] "Conficker", *Malware Wiki*. [Online]. Available: https://malware.wikia.org/wiki/Conficker. [Accessed: 16- Aug- 2018].

[31] J. Fruhlinger, "What is Stuxnet, who created it and how does it work?", *CSO Online*. [Online]. Available: https://www.csoonline.com/article/3218104/what-is-stuxnet-who-created-it-and-how-does-it-work.html. [Accessed: 19- Aug- 2018].

[32] "Stuxnet", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Stuxnet. [Accessed: 19- Aug- 2018].

[33] "What is CryptoLocker Ransomware and How to Remove it", *What is CryptoLocker Ransomware and How to Remove it*. [Online]. Available: https://www.avast.com/c-cryptolocker. [Accessed: 21- Aug- 2018].

[34] R. Lipovsky, A. Cherepanov, R. Lipovsky and A. Cherepanov, "BlackEnergy trojan strikes again: Attacks Ukrainian electric power industry", *WeLiveSecurity*. [Online]. Available: https://www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacks-ukrainian-electric-power-industry/. [Accessed: 23- Aug- 2018].

[35] "BlackEnergy APT Attacks in Ukraine", *www.kaspersky.com*. [Online]. Available: https://www.kaspersky.com/resource-center/threats/blackenergy. [Accessed: 23- Aug- 2018].

[36] "Yahoo is now a part of Verizon Media", *Finance.yahoo.com*. [Online]. Available: https://finance.yahoo.com/news/trump-administration-blames-north-korea-wannacry-ransomware-065507252.html. [Accessed: 04- Nov- 2018].

[37] "Emerging Threats to Industrial Control Systems", *INCIBE-CERT*, 2018. [Online]. Available: https://www.incibe-cert.es/en/blog/emerging-threats-industrial-control-systems. [Accessed: 04- Nov - 2018].

[38] A. Ng, "Android malware that comes preinstalled is a massive threat", *CNET*. [Online]. Available: https://www.cnet.com/news/android-malware-that-comes-preinstalled-are-a-massive-threat/. [Accessed: 03- Feb- 2019].

[39] E. Filiol, "Viruses and Malware", *Handbook of Information and Communication Security*, Heidelberg: Springer, 2010, pp. 747-769, 2010.

[40] S. Murugiah and S. Karen, *Guide to Malware Incident Prevention and Handling for Desktops and Laptops*, Special publication SP800-83. U.S. Department of Commerce - National Institute of Standards and Technology NIST, 2013, p. 2. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-83r1.pdf. [Accessed: 10- Mar- 2018].

[41] S. Harris and F. Maymi, *CISSP® All-in-One Exam Guide*, 7th ed. New York: McGraw-Hill Education, 2016, pp. 15-64.

[42] N. Ben, *Internet Security Threat Report - Email Threats 2017*. Symantec, 2017, p. 9. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/ istr-email-threats-2017-en.pdf. [Accessed: 15- Mar- 2018].

[43] "Why You Should Download All Email Attachments to Your Desktop Before Opening - Howard Tech Advisors", *Howard Tech Advisors*. [Online]. Available: https://www.howardtechadvisors.com/blog/security/download-email-attachments-desktop-opening/. [Accessed: 15- Mar- 2018].

[44] S. Olsen, "Web surfers brace for pop-up downloads", *CNET*, 2002. [Online]. Available: https://www.cnet.com/news/web-surfers-brace-for-pop-up-downloads/. [Accessed: 16- Mar- 2018].

[45] L. Van Lam, W. Ian, G. Xiaoying and K. Peter, "Anatomy of Drive-by Download Attack", in *Proceedings of the Eleventh Australasian Information Security Conference - Volume 138. AISC '13.*, Darlinghurst, Australia, 2013, pp. 49–58.

[46] "Malware Attack Vectors: What to Expect in 2018", *Lastline*. [Online]. Available: https://www.lastline.com/blog/malware-attack-vectors/. [Accessed: 16- Mar- 2018].

[47] "Spear Phishing Remains Preferred Point of Entry in Targeted, Persistent Attacks", *Threatpost.com*. [Online]. Available: https://threatpost.com/spear-phishing-remains-preferred-point-entry-targeted-persistent-attacks-113012/77267/. [Accessed: 18- Mar- 2018].

[48] "What is Phishing and how does it affect email users", *usa.kaspersky.com*. [Online]. Available: https://usa.kaspersky.com/resource-center/preemptive-safety/what-is-phishings-impact-on-email. [Accessed: 18- Mar- 2018].

[49] "New Email Security Report from IRONSCALES Identifies Email Phishing Attack Detection, Mitigation and Remediation as Biggest Challenge for Security Teams", *PRWeb*. [Online]. Available: http://www.prweb.com/releases/2017/09/prweb14742215.htm. [Accessed: 18- Mar- 2018].

[50] "Threat Group-4127 Targets Google Accounts", *Secureworks.com*. [Online]. Available: https://www.secureworks.com/research/threat-group-4127-targets-google-accounts. [Accessed: 18- Mar- 2018].

[51] M. Prague and M. Brno, "Malware in ads causes trouble for millions of people. But ad publishers still fight against ad blocking - Blog MasterDC.com", *Master Internet*. [Online]. Available: https://www.masterdc.com/blog/malware-in-ads-threatens-millions-drive-by-attacks-ad-agencies-against-blocking/. [Accessed: 21- Mar- 2018].

[52] "Malvertising", *CIS*. [Online]. Available: https://www.cisecurity.org/blog/malvertising/. [Accessed: 21- Mar- 2018].

[53] F. Skulason, "Virus Encryption Techniques," *Virus Bulletin*, November 1990, pp. 13-16.

[54] B. Rad, M. Maslin and I. Suhaimi, "Camouflage in malware: from encryption to metamorphism", *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74-83, 2012.

[55] "Stealth Virus", *Spamlaws.com*. [Online]. Available: https://www.spamlaws.com/stealth-virus.html. [Accessed: 24- Mar- 2018].

.

[56] G. Vigna, "When Malware is Packing Heat", *Lastline*. [Online]. Available: https://www.lastline.com/labsblog/malware-packing/. [Accessed: 27- Mar- 2018].

[57] P. Arntz and P. Arntz, "Explained: Packer, Crypter, and Protector", *Malwarebytes Labs*. [Online]. Available: https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector/. [Accessed: 27- Mar- 2018].

[58] W. Yan, Z. Zheng and A. Nirwan, "Revealing packed malware", *IEEE Security & Privacy*, vol. 6, no. 5, pp. 65-69, 2008. Available: https://pdfs.semanticscholar.org/785a/8d4e207f3c5a6f587f2ce7278316ca3462e7.pdf

[59] "Packer", *Encyclopedia.kaspersky.com*. [Online]. Available: https://encyclopedia.kaspersky.com/glossary/packer/. [Accessed: 27- Mar- 2018].

[60] M. Morgenstern and A.Marx, "Runtime packer testing experiences", in *2nd International CARO Workshop*, 2008.

[61] Kaspersky Lab. Virus.DOS.Whale, 2000. *Whale appeared c*. 1990.

[62] P.Szor. "Hunting for metamorphic". In *Virus Bulletin Conference*, 2001.

[63] P. Szor, *The art of computer virus research and defense*. Upper Saddle River, N.J.: Addison-Wesley, 2005.

[64] C. Fischer. TREMOR analysis (PC). *VIRUS-L Digest*, 6(88), 1993.

[65] "What is polymorphic virus? - Definition from WhatIs.com", *SearchSecurity*. [Online]. Available: https://searchsecurity.techtarget.com/definition/polymorphic-malware. [Accessed: 27- Mar- 2018].

[66] "What is Obfuscation? - Definition from Techopedia", *Techopedia.com*. [Online]. Available: https://www.techopedia.com/definition/16375/obfuscation. [Accessed: 30- Mar- 2018].

[67] W. Wong and M. Stamp, "Hunting for metamorphic engines", *Journal in Computer Virology*, vol. 2, no. 3, pp. 211-229, 2006.

[68] E. Konstantinou, "Metamorphic Virus: Analysis and Detection", Royal Holloway, University of London, 2008. [Online]. Available: http://www.rhul.ac.uk/mathematics/techreports. [Accessed: 30- Mar- 2018].

[69] A. Balakrishnan and S. Chloe, *Code obfuscation literature survey| CS701 Construction of Compilers*. University of Wisconsin, Madison: Computer Sciences Department, 2005. [Online]. Available: http://pages.cs.wisc.edu/~arinib/writeup.pdf

[70] M. Christodorescu and J. Somesh, "Static analysis of executables to detect malicious patterns", in *12th conference on USENIX Security Symposium*, 2003, pp. 169-186.

[71] "Microsoft Security Intelligence Report volume 11 (January - June 2011) | An in-depth perspective on software vulnerabilities and exploits, malicious code threats, and potentially unwanted software in the first half of 2011 ZEROING IN ON MALWARE PROPAGATION METHODS", Microsoft.com, 2011. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=27605. [Accessed: 03- Apr- 2020].

[72] "Types of Malware", *www.kaspersky.com.au*. [Online]. Available: https://www.kaspersky.com.au/resource-center/threats/malware-classifications. [Accessed: 03- Apr- 2018].

[73] M. K A, *Learning malware analysis*. Birmingham, UK: Packt Publishing, 2018, pp. 8-9.

[74] C. Elisan, *Advanced Malware Analysis*. McGraw-Hill Education, 2015, pp. 5-7.

[75] "Malware analysis", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Malware_analysis. [Accessed: 04- Apr- 2018].

[76] "What Are Memory Forensics? A Definition of Memory Forensics", *Digital Guardian*. [Online]. Available: https://digitalguardian.com/blog/what-are-memory-forensics-definition-memory-forensics. [Accessed: 04- Apr- 2018].

[77] S. Katzenbeisser, J. Kinder and H. Veith, "Malware Detection", *Encyclopedia of Cryptography and Security*, pp. 752-755, 2011.

[78] "What is Malware Detection? | Importance of Malware Tool", *Comodo Enterprise*. [Online]. Available: https://enterprise.comodo.com/what-is-malware-detection.php. [Accessed: 06- Apr- 2018].

[79] F. Cohen, "Computer viruses: Theory and experiments", *Computers & Security*, vol. 6, no. 1, pp. 22-35, 1987. Available: https://www.profsandhu.com/cs5323_s18/cohen-1987.pdf.

[80] I.Muttik, "STRIPPING DOWN AN AV ENGINE", in *VIRUS BULLETIN CONFERENCE*, Oxfordshire, England, 2000, pp. 59-68.

[81] "SANS Institute: InfoSec Reading Room - Malicious Code", *Sans.org*. [Online]. Available: https://www.sans.org/reading-room/whitepapers/malicious/bypassing-malware-defenses-33378. [Accessed: 08- Apr- 2018].

[82] K. Sandeep and H. Eugene, *A GENERIC VIRUS SCANNER IN C++ |The COAST Project - Technical Report CSD-TR-92-Q62*. West Lafayette, IN 47907-1398: Department of Computer Sciences Purdue University, 1992.

[83] "Antivirus Research and Detection Techniques", *ExtremeTech*. [Online]. Available: https://www.extremetech.com/computing/51498-antivirus-research-and-detection-techniques. [Accessed: 08- Apr- 2018].

[84] W. Christian, F. Kevin, Y. Fabian and R. Konrad, "From Malware Signatures to Anti-Virus Assisted Attacks - Computer Science Report No. 2016-03", Technische Universität Braunschweig of Germany. Institute of System Security, 2016, p. 6. Available: https://arxiv.org/pdf/1610.06022.pdf.

[85] P. Szor, "Virus analysis 1: Beast regards", *Virus Bulletin*, 1999.

[86] J. Nazario, *Defense and detection strategies against Internet worms*. Boston, MA: Artech House, 2004, p. 176.

[87] S. Kanaujiya, D. Tripathi and N. Sharma, "Improving Speed of the Signature Scanner using BMH Algorithm", *International Journal of Computer Applications*, vol. 11, no. 4, pp. 27-31, 2010.

[88] T. Cormen, C. Leiserson, L. Ronald and R. Stein, *Introduction to algorithms*, 3rd ed. Mass: Massachusetts Institute of Technology, 2009, pp. 906-907.

[89] A. Binstock and J. Rex, *Practical algorithms for programmers*. Reading, Mass: Addison-Wesley, 1995.

[90] F. Lance and H. Steve, *A Short History of Computational Complexity*. 2002. [Online]. Available: https://people.cs.uchicago.edu/~fortnow/papers/history.pdf

[91] R. Boyer and J. Moore, "A fast string searching algorithm", *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977. Available: 10.1145/359842.359859.

[92] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2011, p. 770-773.

[93] C. Charras and T. Lecroq, *Handbook of exact string-matching algorithms*. London: Kings College Publ, 2004, pp. 91.

[94] "Boyer-Moore algorithm", *Inf.hs-flensburg.de*, 2018. [Online]. Available: https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/bmen.htm. [Accessed: 07- Feb- 2019].

[95] D. Knuth, J. Morris, Jr. and V. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323-350, 1977. Available: 10.1137/0206024.

[96] R. Horspool, "Practical fast searching in strings", *Software: Practice and Experience*, vol. 10, no. 6, pp. 501-506, 1980. Available: 10.1002/spe.4380100608.

[97] D. Sunday, "A very fast substring search algorithm", *Communications of the ACM*, vol. 33, no. 8, pp. 132-142, 1990. Available: 10.1145/79173.79184.

[98] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search", *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975. Available: 10.1145/360825.360855.

[99] S. Wu and M. Udi, "A fast algorithm for multi-pattern searching", University of Arizona. Department of Computer Science, 1994, pp. 1-11.

[100] L. Salmela, J. Tarhio and J. Kytöjoki, "Multipattern string matching withq-grams", *Journal of Experimental Algorithmics*, vol. 11, pp. 1.1, 2007. Available: 10.1145/1187436.1187438.

[101] B. Commentz-Walter, "A string matching algorithm fast on the average", In Proc. *Springer 6th International Colloquium on Automata, Languages, and Programming*, Berlin, Heidelberg, 1979, pp. 118-132.

[102] A. Aho, "Algorithms for finding patterns in strings" in *Handbook of theoretical computer science, vol A*. J. Leeuwen, Amsterdam: Elsevier, 1990, pp. 257-297.

[103] A. Jony, "Analysis of multiple string pattern matching algorithms", *International Journal of Advanced Computer Science and Information Technology (IJACSIT)*, vol. 3, no. 4, pp. 344-353, 2014.

[104] A. Bik, *The software vectorization handbook*. Hillsboro, Or.: Intel Press, 2006, pp. 1-3.

[105] D. Sima, K. Peter, F. Terence, *Advanced computer architectures - a design space approach*. International computer science series, Harlow: Addison-Wesley, 1997.

[106] A. Clements, Computer organization and architecture, 1st ed. Stamford, CT: Cengage Learning, 2014, pp. 316.

[107] "Intel | Data Center Solutions, IoT, and PC Innovation", *Intel*. [Online]. Available: http://www.intel.com/content/www/us/en/trademarks/mmx.html. [Accessed: 11- Feb- 2019].

[108] "Makers Unveil PCs With Intel's MMX Chip", *Archive.nytimes.com*. [Online]. Available: https://archive.nytimes.com/www.nytimes.com/library/cyber/week/010997intel.html. [Accessed: 11- Feb- 2019].

[109] *Intel MMX™ Technology Overview*. - Intel Corporation, 1996. [Online]. Available: https://www.ee.ryerson.ca/~courses/ele818/mmx.pdf

[110] S. Mueller, *Upgrading and repairing PCs*, 19th ed. Indianapolis, Ind.: Que Pub., 2010, pp. 71-73.

[111] *Intel® SSE4 Programming Reference*, Intel Corporation, 2007. [Online]. Available: http://www.info.univ-angers.fr/pub/richer/ens/l3info/ao/intel_sse4.pdf. [Accessed: 15- Jan- 2016].

[112] *Instruction Set Architecture Extensions*, Intel Corporation. [Online]. Available: https://software.intel.com/en-us/isa-extensions/ intel-avx. [Accessed: 07- Jan- 2016].

[113] D. Kusswurm, *Modern X86 Assembly Language Programming*. Berkeley, CA: Apress, 2014, pp. 01-03.

[114] *Intel® Architecture Instruction Set Extensions Programming Reference.* Intel Corporation, 2016. [Online]. Available: https://www.naic.edu/~phil/ software/intel/319433-014.pdf.

[115] "Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Overview", *Intel*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. [Accessed: 14- Feb- 2019].

[116] "How to detect New Instruction support in the 4th generation Intel®...", *Intel*. [Online]. Available: https://software.intel.com/en-us/articles/how-to-detect-new-instruction-support-in-the-4th-generation-intel-core-processor-family. [Accessed: 14- Feb- 2019].

[117] *An introduction to Vectorization with the Intel C++ Compiler*. Intel Corporation, 2016. [Online]. Available: https://software.intel.com/sites/default/ files/An\ Introduction\ to\ Vectorization\ with\ Intel\ C++\ Compiler\ 021712.pdf.

[118] *AVX Vectorisation and Cilk Plus*. Oxford e-Research Centre, 2016. [Online]. Available: http://www.oerc.ox.ac.uk/projects/asearch/ software/avx.

[119] A. Vladimirov, R. Asai and V. Karpusenko, *Parallel programming and optimization with Intel Xeon Phi coprocessors*, 2nd ed. Sunnyvale, CA 94085, USA: Colfax International, 2015, pp. 157-158.

[120] D. Page, *Practical Introduction to Computer Architecture*. London: Springer London, 2009, pp. 389-390.

[121] J. Jeffers, J. Reinders and A. Sodani, *Intel Xeon Phi coprocessor high-performance programming Knights Landing Edition.* Cambridge, MA: Morgan Kaufmann, 2016, pp. 11 - 12.

[122] *A Guide to Vectorization with Intel® C++ Compilers*. Intel Corporation, 2012. [Online]. Available: https://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf

[123] X. Zhou, X. Bo, Q. Yaxuan and L. Jun, "MRSI: A fast pattern matching algorithm for anti-virus applications", In Proc. *IEEE Seventh International Conference on Networking*, icn, 2008, pp. 256–261.

[124] J. T. L. Ho and G. G. F. Lemieux, "PERG: A scalable FPGA-based pattern-matching engine with consolidated Bloomier filters," In Proc. *IEEE International Conference on Field-Programmable Technology*, Taipei, 2008, pp. 73-80, doi: 10.1109/FPT.2008.4762368.

[125] V. Alexandre Nuno, "Detecting Computer Viruses using GPUs", *Pdfs.semanticscholar.org*. [Online]. Available: https://pdfs.semanticscholar.org/a878/94e14761d2afd0b51cfbbaa2b2819f8f31a1.pdf. [Accessed: 15- Feb- 2019].

[126] N. Dien, T. Tran and N. Tran, "Memory-based multi-pattern signature scanning for clamav antivirus", In Proc. *Springer International Conference on Future Data and Security Engineering*, Cham, '11, 2014, pp. 58-70.

[127] *Creating signatures for ClamAV*, ClamAV. [Online]. Available: https://aws.huihoo.com/clamav/signatures.pdf. [Accessed: 02- Jan- 2018].

[128] L. Tsern-Huei, "Generalized aho-corasick algorithm for signature based anti-virus applications", In Proc. *IEEE 16th International Conference on Computer Communications and Networks*, 2007, pp. 792-797.

[129] H. Bidgoli, *Handbook of information security - Threats, Vulnerabilities, Prevention, Detection, and Management*. Hoboken, N.J.: John Wiley & Sons, 2006, pp. 450 - 458.

[130] "Software optimization resources. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs ", *Agner.org*. [Online]. Available: https://www.agner.org/optimize//instruction_tables.pdf. [Accessed: 27- Feb- 2018].

[131] R. Friedman, "More Than Ever, Vectorization and Multithreading are Essential for Performance - insideHPC", *insideHPC*, 2017. [Online]. Available: https://insidehpc.com/2017/08/essential-performance/. [Accessed: 27- Feb- 2019].