Heterogeneous Pattern Matching for Intrusion Detection Systems and Digital Forensics

PhD Adviser:
Prof. Dr. Viorel Negru

Candidate:
Ciprian-Petrişor Pungilă

Faculty of Mathematics and Informatics
West University of Timişoara

A thesis submitted for the degree of
Doctor of Philosophy

2012

"Life is like a box of chocolates - you never know what you're gonna get."
(Forrest Gump)

Acknowledgments

I would like to express my most grateful thoughts to my parents, for their continued motivation into pursuing an academic career. Therefore, I dedicate this thesis to them, as it was their support and advice throughout the years that had made me become the person I am today.

Part of this thesis was supported by several international research grants, including:

- European Commission FP7-REGPOT-CT-2011-284595 (*HOST*)
- Romanian national grant PN-II-ID-PCE-2011-3-0260 (*AMICAS*)
- European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No. 224609 (*DEHEMS*)

Besides the motivational background involved in these projects, there are a number of people I would like to thank, which have guided my work at the university and encouraged my research, while also providing the means to pursue them both: first, I would like to express my gratitude to my supervisor, *Prof. Dr. Viorel Negru*, who inspired my academic career and whose help and support throughout the years has had a significant impact on the research I had performed. Second, I would like to thank *Prof. Dr. Dana Petcu*, for her support and patience throughout the development of the *DEHEMS* project. I wish to express my gratitude to the entire Computer Science department of the Faculty of Mathematics and Informatics at the West University of Timişoara, in particular to Prof. Dr. Daniela Zaharie (for her insight and advice related to the work I had carried out throughout the years), Assoc. Prof. Dr. Florin Fortiş (for trusting me and supporting my ideas in the different courses where we worked on together during the past few years) and Assoc. Prof. Dr. Mircea Drăgan (for his help and attention to detail in proof-reading, as well as his continued interest in the research I had performed). I also wish to express my courtesy to the members of the IeAT Research Institute for the work we had done together in the past. Special thanks go to Sean Baxter and Tim Murray at nVIDIA Corp., for their help and useful advice on the CUDA architecture throughout the development of the work for this thesis.

Abstract

This thesis focuses on the development, implementation and optimization of pattern-matching algorithms in two different, yet closely-related research fields: malicious code detection in intrusion detection systems and digital forensics (with a special focus on the data recovery process and the metadata collection stages it involves). The thesis introduces the motivational backgrounds for the development of the work, later on presents the related work and then continues with the main achievements obtained, while in the end a few conclusions and future research directions are discussed.

The main four chapters in this thesis, starting with Chapter 3 and up to chapter 6, show the main contributions of our work and address the following topics: Chapter 3 presents an efficient storage mechanism for hybrid CPU/GPU-based systems, and compares it with other known approaches to date. Chapter 4 proposes an innovative, highly parallel approach to the fast construction of very large Aho-Corasick and Commentz-Walter pattern matching automata on hybrid CPU/GPU-based systems, and compares it to existing sequential approaches. Chapter 5 proposes a new heuristics for profiling malicious behavior based on system-call analysis, using the Aho-Corasick algorithm, and also discusses a new hybrid compression mechanism for this automata based on dynamic programming, that reduces the storage space required for it. Finally, Chapter 6 proposes an efficient new method for collecting metadata and helping the human operator or automated tools used in the data recovery process as part of the computer forensic investigations.

The research and models obtained in this thesis extend the existing literature in the field of intrusion detection systems (malicious code detection in particular), by presenting: an innovative heuristics for behavioral analysis of code in executable files through system-call interception, a novel and highly efficient approach to efficiently storing pattern-matching automata in hybrid CPU/GPU-based systems, which serves as the base for an innovative model for the fast, GPU-accelerated construction of such very large automata (for both the Aho-Corasick and Commentz-Walter algorithms) and a new hybrid compression technique applied to the Aho-Corasick automata using a dynamic programming approach, that reduces storage space significantly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Introduction

Along with the invention of the transistor in the 1950s, a new era has emerged, dominated by increasingly faster machines, controlled through fine-tuned electronics. Starting with the first transistor built, the age of digital computing had risen to become more and more persistent, throughout the years, into everyday life, and its effects are immensely felt today - we are, essentially, a society based on the correct operational cycles of the machines we had built. It is therefore not surprising that the transistor was considered to be a part of the greatest breakthroughs of the 20th century[Pri04].

Increasing operational speeds and the significant progress in nano-technology in the past years have determined computers to become faster and more accurate. Smaller transistors translate into better performance and improved response times. Consumer PCs nowadays use 22nm technology on a daily basis, while 14nm-based technology is scheduled to appear soon, with a roadmap for 5nm being also released and advertised as being actively researched[int12]. While the immediate benefits of such progress have numerous applications in most scientific areas, they are unfortunately also being used with malicious intent. Probably the oldest examples of such intents is represented by computer viruses, whose history begins in the early ages of computing, more exactly in Von Neumann's research [vN66], where his theory of

"self-reproducing automata" describes the possibility to design a program that can reproduce itself. A computer virus is, in fact, an executable program which may cause damage to the computer or may interfere with its normal operation, according to [mic12].

Computer viruses have evolved significantly over time. Along with the benefits of improved technology and continuously improved operating systems, viruses have been following a massively increasing trend, starting with a single known virus in 1985 and up to over 1 million viruses existing in 2010, according to [SQ10]. Nowadays, there are over 17,7 million signatures [vir12]. Furthermore, complexity has increased significantly also, beginning with polymorphic viruses (which appeared first in 1990) and ending with the most complex, most difficult to detect and to remove specimen of them all: metamorphic viruses. A polymorphic virus is based on an engine which can mutate the code, while leaving the functionality the same. Mutation is achieved most of the time through encryption, but encryption alone is not sufficient - a decryption module also needs to exist in order to correctly interpret the mutated code for leaving the original semantics untouched. The decryption module is usually Achille's heel for polymorphic viruses, since antivirus programs can look for that particular module in order to try to identify the virus type or family. Metamorphic viruses are designed to rewrite their own code entirely from one infection to another. This makes them extremely resilient to antivirus detection and recent research performed into metamorphic code, such as Stuxnet [stu11] and Flame [fla12] (considered by the Kaspersky Labs researches to be "the most sophisticated cyber weapon" to have ever been in the wild [red12]), shows how tremendously difficult it is to decrypt, analyze and interpret such code.

1.2 Motivation

In parallel with virus development, antivirus research, as part of the intrusion detection systems field, has progressed significantly and while many antivirus competitors exist on the market, offering different protection solutions with different efficiencies, open-source alternatives, such as the ClamAV initiative [cla12], have been the primary focus of academic security researches. Antivirus programs can detect viruses essentially in two ways: by looking up virus signatures in the executable code, or by profiling the run-time behavior, usually through emulation. Virus signatures are data patterns which uniquely describe the identity of a virus or of a family of viruses. The problem of identifying a signature inside binary code is a classic problem of pattern matching. With the growing number of signatures each day, it is becoming increasingly difficult to implement efficient pattern matching techniques. Although multiple pattern matches algorithms have been proposed, such as the Aho-Corasick [AC75],

Wu-Manber [WM94] and Commentz-Walter [CW79] variants, real-time implementations still pose a challenge to reducing the scanning time of an executable. Parallel approaches to both single and multiple pattern matching have been researched, however given the low number of cores available on most CPUs usually, the speed increase obtained is still causing significant bottlenecks in real-time implementations.

Recent progress in the field of GPU technology, along with nVIDIA's CUDA architecture [cud12] have made possible the ability to build hybrid, CPU/GPU-based solutions, that could benefit from the high degree of parallelism offered by the GPU hardware. For intrusion detection systems, GPU-accelerated approaches are an active field of research. A lot of research has also been conducted for integrating GPU-acceleration into network intrusion detection systems (NIDS), and important speed-ups have also been obtained with GPU-accelerated variants of known algorithms applied to malicious code detection, particularly in antivirus-related research and implementations. Given the plural nature of pattern matching algorithms and the various fields of research where they find applications, an increasing interest in the past few years has been observed in the digital forensics area, where data recovery procedures and defragmentation approaches have been successfully used for tracking down pieces of information, putting together clues in forensic analysis and recovering lost or fragmented pieces of information from damaged hardware devices. The recent engineering breakthroughs in building more power-efficient GPUs, with better architectures and higher semiconductor densities has certainly opened up a wide range of possibilities for building complex hybrid systems in the near future, where the CPU and GPU interact, share information and perform computations as a single, unique and homogenous entity.

The main objective of this PhD thesis is to propose, present and evaluate the theoretical backgrounds for achieving pattern matching on two different levels of heterogeneity: in hybrid hardware environments (using both the CPU and the GPU for carrying out tasks as a single, hybrid-processing unit - where we concentrate on two important aspects of the implementation: low storage requirements and high-performance throughput) and hybrid implementations (such as hybrid-parallel approaches and implementations of various extensions to the existing research) for solving the pattern-matching problem more efficiently in both intrusion detection systems (with primary focus on malicious code detection) and digital forensics (where we are focusing on carving analysis as part of the digital forensics process).

1.3 Objectives of the Thesis

The objectives of this thesis are the following:

1. to analyze the current state-of-art for implementing pattern-matching in heterogeneous hardware environments, where the CPU and GPU work as a single entity designed to carry out the tasks given to it;

2. to propose a formalism for achieving efficient storage of very large pattern-matching automata in hybrid CPU/GPU memory, implement it and evaluate its performance compared to other existing known approaches;

3. to propose a formalism, architecture and computational model for the highly-parallel construction of very large pattern-matching automata in heterogeneous hardware environments;

4. to propose a model for implementing behavioral heuristics through heterogeneous implementations or extensions of pattern-matching algorithms for tracking down malicious code behavior;

5. to propose a model for achieving highly-compact pattern-matching automata by using different compression techniques, and apply the model to the Aho-Corasick automaton in the virus signature matching process;

6. to propose a heterogeneous approach to extending the regular-expression pattern-matching problem to the area of carving analysis in digital forensics, outlining the primary advantages and benefits of such an approach as part of cyber-crime investigations and data recovery.

1.4 Contribution of the Thesis

This thesis focuses on two aspects of research: the first relates to profiling malicious code behavior by optimizing and accelerating pattern matching algorithms, while the second discusses the applicability of our research to digital forensics and its possible future uses. Profiling malicious code behavior is a highly complex task, requiring several steps of computation, which is why in recent years numerous attempts to parallelize and optimize the known approaches have emerged. In this thesis, we are tackling with some of those approaches and propose innovative and highly efficient models for accelerating such algorithms, as well as optimizing them to support regular expression scanning and constraint-based matching.

We start by discussing an innovative and highly-efficient storage model in Chapter 3, presenting a new hybrid approach for achieving very efficient malware detection in hybrid CPU/GPU systems, without affecting throughput performance in any way. We discuss known GPU and hybrid storage mechanisms and propose a new model whose primary advantage is its low-storage requirements, as well as full-duplex memory transfers at the highest possible throughput between the host and the device, while

also requiring a single external pointer to the memory area holding the automaton. We present the final results by comparing storage models in modern antivirus implementations, starting with the ClamAV antivirus and outlining our approach's primary benefits in a hybrid-parallel implementation during the virus scanning process.

The storage model in Chapter 3 is used as a basis for the discussion in Chapter 4, where we present an innovative highly-parallel architecture for constructing fast pattern matching automata using a hybrid CPU/GPU approach, which improves the processing times by an important number of times and allows real-time updates to such automata, opening new research opportunities by offering the possibility to construct very fast, self-adjustable automata. This research represents the first parallel model (which is also capable of offering maximum throughput performance) for a very fast construction of pattern-matching automata, showing how the construction performance can be improved significantly when using virus signatures as part of the virus signature matching process, withholding highly-efficient storage both in host and device memory.

Chapter 5 proposes a new heuristic for profiling malicious code behavior through system call analysis by taking into account the moment in time when a call is made during program execution, with very little impact on performance. This represents the first model for time-based analysis of system calls, an important part of malicious code detection through system-call analysis heuristics in antivirus engines. The chapter also covers a new approach to achieving hybrid compression of the Aho-Corasick pattern matching automaton using dynamic programming aspects which significantly reduce the storage required for this automata, presenting the experimental results obtained when applying our methodology to the virus signature matching automaton in the virus detection process, a common component of all modern antivirus engines. Our hybrid approach shows that a combination of our dynamic programming implementation, along with several other improvements, can reduce the storage memory required by a few times compared to the naive or other existing approaches.

The last chapter, Chapter 6, discusses a new approach to improving the carving analysis process in digital forensics through employing efficient pattern-matching automata supporting regular-expressions, and shows how the classic problem of file-carving can be adapted to this model. The implementation covers a lexical analyzer for converting the XML-based signatures used in the structural analysis process into regular-expression-based signature formats used in automata creation. Our approach is based on header and structural analysis and allows different heuristics to be built on top of the detection engine for an accurate reconstruction of the original file, permitting therefore efficient, low-memory knowledge-based forensic analysis engines to be built.

1.5 Organization

The remaining chapters of this thesis are organized as follows:

Chapter 2 presents the latest existing algorithmic approaches to pattern matching, discussing both single and multiple pattern matching approaches (with special emphasize on the latter), and discusses their applicability to intrusion detection systems, malicious code behavior detection and digital forensics.

Chapter 3 discusses the challenges involved in performing efficient storage of the automata used in malicious code detection, discusses the GPU storage mechanisms and proposes a new, highly efficient approach to storing such automata in both RAM and GPU (device) memory.

Chapter 4 proposes an innovative and highly efficient hybrid CPU/GPU-based model for the fast construction of very large Aho-Corasick [AC75] and Commentz-Walter [CW79] automata, showing how such a model may be applied to malicious code detection in particular. We later on discuss the applicability of the model to other types of patterns and show that its applicability is not limited in the intrusion detection systems field of research, but can be used without any limitations to any other field involving multiple pattern matching through the two types of automata.

Chapter 5 discusses in-depth a few profiling heuristics for behavioral analysis commonly used in pattern matching automata using the Aho-Corasick algorithm [AC75], as well as a hybrid approach to compress the automata using a dynamic programming method. We present and discuss the challenges involved, along with the mathematical models for behavioral analysis, and show how they can be applied to malicious code detection in intrusion detection systems.

Chapter 6 proposes a new model (based on a pattern-matching automata heuristic) for collecting metadata, a vital component in the data recovery process commonly used in digital forensics, and shows how the model can significantly reduce human operator intervention by locating and reconstructing an improved partial map of the original un-damaged file, given the proper binary signatures for recovery are used.

At the end, Chapter 7 outlines the main contributions of the thesis and discusses future research opportunities and directions.

Chapter 2

Related Work

Contents

2.1 Pattern-Matching Algorithms

The pattern matching challenge has a long history in the computing era, dating back to the early stages in computing. The single pattern matching problem aims to find all occurrences of a given, non-empty keyword, into an input string, while later applications have extended the problem to finding multiple occurrences of a finite, non-empty set of keywords into an input string. Pattern matching applications are widely used in web search engines (where they are used to identify webpages of interest for the users), bioinformatics (in [Tum10], the authors discuss the ability to perform DNA analysis on GPU clusters, using a multiple pattern matching algorithm; in [Dud06], authors propose a heuristic that can reduce the number of comparisons by increasing the number of skips in the pattern matching problem) and intrusion detection systems (where they are for instance used in tracking down malicious network data packets, such as the approaches proposed in [MK06] and [CY06], or for enabling high-level

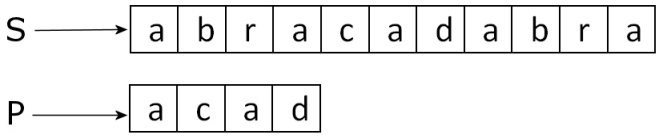


Figure 2.1: The input string and the pattern for upcoming examples.

architectures for pattern matching and implementing them on the GPU, as discussed in [LH12]; they are also widely used in antivirus engine implementations, such as the approach suggested in [Lee07] or using GPU-accelerated approaches for performing virus signature matching, as shown in [PN12]), with several other important applications in numerous fields of research.

Since pattern matching algorithms have a wide range of applications and are numerous, with several variants existing, we will present as follows only those algorithms finding the most common and relevant applications to the fields of intrusion detection systems and digital forensics. Cleophas et al in [CZ04] discusses a taxonomy of pattern matching algorithms that improves on the work of Watson et al from [WZ96], including the SPARE toolkit that Watson et al had developed and presented in [Wat04].

2.1.1 Single-Pattern Matching Algorithms

According to Watson in [Wat94], the single pattern-matching problem can be defined as follows: given an alphabet (a finite sequence of characters) Σ and a text string S = Γ[0], Γ[1], ..., Γ[N-1] of length N, find a string pattern P = Λ[0], Λ[1], ..., Λ[M-1] of length M, where $\Gamma[i] \in \Sigma$ and $\Lambda[j] \in \Sigma$, for each $1 \leq i \leq N$, $1 \leq j \leq M$ and M $\leq$ N. The algorithms existing nowadays propose several approaches to efficiently implementing and solving the pattern matching problem, aiming to improve the runtime performance while also decreasing storage space as much as possible.

We are going to present some of the most commonly used variants of single pattern matching algorithms, beginning with the brute-force approach and ending with approaches proposed by Knuth-Morris-Pratt ([Knu77]) and Boyer-Moore ([BM77]), which are the most commonly used in implementation nowadays given their very good performance. Most of the time, $| \Sigma | = 256$, to cover the entire ASCII alphabet, however some research, especially related to bioinformatics, uses smaller alphabet sizes. In the following, we denote by *P[i]* the character at position i in the string P.

2.1.1.1 The Brute-Force Algorithm

The brute force algorithm is the simplest (and the slowest) of existing variants of multiple pattern matching. It's basic idea lies in comparing, starting with each character Γ*[j]* at position j ($j \in \{0, 1, ..., N - M - 1\}$) in the input string, up to the

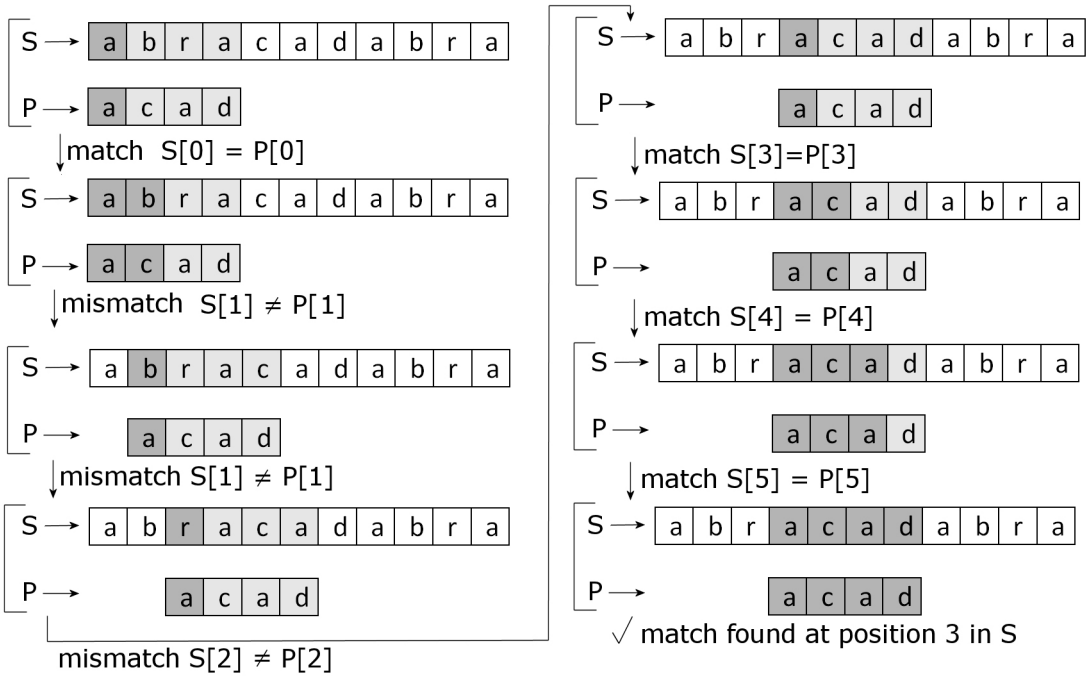


Figure 2.2: Comparisons performed by the brute-force simple pattern matching algorithm.

next M-1 characters, the characters in S with those in P. In case of a mismatch, the comparisons begin again, starting with the next position, at Γ*[j+1]*. The algorithm's worst-case performance is *O(M × (N-M-1))* (and matches *O(N × M)*, according to [Cor02]), in the situation where all comparisons match, excepting the last character in P, while the best-case performance is *O(N)* (assuming that the only comparison being performed is the one at the first character in P). According to Cormen in [Cor02], the expected number of comparisons, or the average case performance, is linear and at most *2 × (N-M+1)*, which equals *O(N)*.

Figure 2.2 shows the list of comparisons performed by the brute-force algorithm for the example shown in Figure 2.1. The algorithm for performing a brute-force search is presented as follows:

- *Input*
  1: S, the input string (of length N)
  2: P, the pattern to look for (of length M)

- *Brute-force simple pattern matching*
  1: **for** $i \leftarrow 0$ to $N-1$ **do**
  2: match ← true
  3: **for** $j \leftarrow 0$ to $M-1$ **do**
  4: **if** S[i+j] ≠ P[j] **then**

```
5:          match ← false
6:          break
7:        end if
8:      end for
9:      if match = true then
10:       report match found at position i
11:     end if
12: end for
```

2.1.1.2 The Karp-Rabin Algorithm

Karp and Rabin[KR87] proposed a string matching solution by relying on hash computation. Hashes are mathematical functions that assign to each string their numerical value. Starting from the observation that if the two hashes of two strings are equal, then the strings themselves are also equal, Karp and Rabin have built hashes for each of the successive substrings of the pattern. Given that the algorithm relies heavily on the efficient computation of the hash values, and that in certain situations, a certain hash function can produce the same output for two different string inputs (also called a *hash collision*, resulting in a possible mismatch which may be reported by the algorithm as a match), an efficient hash function is vital to the correct functional behavior of the approach. The algorithm uses Rabin's fingerprinting technique presented in [Rab81]. Broder, in [Bro93], discusses applications of Rabin's fingerprinting method, and shows that a fingerprint is in fact similar to the hash computation of a larger input (similar to the MD5 hashes presented in [Riv92]), and that there is a slight chance that *collisions* may appear whenever two hash values are identical, even though the fingerprints are different.

The algorithm works by computing all possible hash values for the N-M+1 substrings of length M found in the input string of length N, and making use of these at run-time, when comparing hash values ensure that we can avoid multiple comparisons, such as those performed by the brute-force algorithm for example in 2.1.1.1. The algorithm maintains a *sliding window* substring of length M over the input string, from left to right, that slides over the input and recomputes the hash value for the sliding window at each iteration. If the hash values of the keyword and of the sliding window are equal, we have a match, otherwise the sliding window slides one more character to the right. According to [Cor02], the worst-case run-time performance is *O((N-M+1) × M)* and matches that of the brute-force algorithm, while the expected run-time performance is *O(N+M)*.

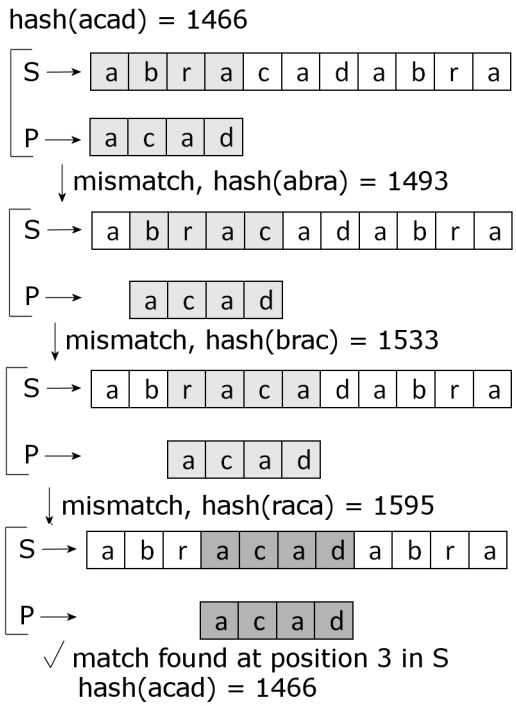


Figure 2.3: Comparisons performed by the Karp-Rabin pattern matching algorithm.

A possible hashing function for a keyword w of length M (where $\mid w[i] \mid$ represents the ASCII numerical code of the character at position i in w) is:

$$hash(w[0, .., M\text{-}1]) = (\sum_{i=0}^{M-1} \mid w[i] \mid \times 2^{M-1-i}) \mod q, \tag{2.1}$$

where q is a large number (and usually prime). The sliding window concept ensures that for the input string S, the hashing value *hash(S[i+1, ..., i+M])* is easily recomputed from *hash(S[i, ..., i+M-1]* and *hash(S[i+M])*, therefore we can express this as:

$$hash(S[i+1, ..., i+M]) = recomputeHash(S[i], S[i+M], hash(S[i, ..., i+M\text{-}1])) \tag{2.2}$$

which would determine:

$$recomputeHash(a, b, d) = ((d - a \times 2^{M-1}) \times 2 + b) \mod q \tag{2.3}$$

Whenever a hash collision occurs, a character-by-character comparison must be performed to ensure that there is a match in the input string. The algorithm for performing a Karp-Rabin search is shown below:

- *Input*

1: *S*, the input string
2: *P*, the pattern to look for
3: *hash(S[0, ..., M-1])*, the hash of the sliding window

- *The Karp-Rabin simple pattern matching algorithm*

1: $i \leftarrow 0$
2: **while** $i \leq N - M + 1$ **do**
3: **if** hash(P) = hash(S[i, ..., i+M-1]) **then**
4: report match found at position i
5: **end if**
6: hash(S[i+1, ..., i+M]) = recomputeHash(S[i], S[i+M], S[i, ..., i+M-1])
7: $i \leftarrow i + 1$
8: **end while**

An example of the run-time process on the pattern and input string from Figure 2.1 is shown in Figure 2.3.

2.1.1.3 The Knuth-Morris-Pratt Algorithm

Knuth et al have presented in [Knu77] an improved algorithm for simple pattern matching based on the approach built by Morris and Pratt in [MP70], that can avoid certain character comparisons by precomputing a model for the pattern P itself, and using that computed model for avoiding certain unnecessary comparisons. This came as a natural extension to the observation that certain patterns, in particular those containing repetitive substrings (e.g. *abracadabra, aabbaabbcc*, etc.), perform unnecessary repeated comparisons whenever some mismatches occur.

In order to present the algorithm we need to define, based on Cormen et al ([Cor02]) the notions of *prefix* and *suffix* of a pattern $P = \{P[0, 1, ..., M-1]\}$ of length M. The prefix $Pre_L(P)$ of length $L \geq 1$, and suffix $Suf_L(P)$ of length L, for pattern P are defined as follows:

$$Pre_L(P) = P[0, 1, ..., L-1] \tag{2.4}$$

$$Suf_L(P) = P[M-L, M-L+1, ..., M-1] \tag{2.5}$$

Basically, $Pre_L(P)$ is a prefix of P if P begins with $Pre_L(P)$, and $Suf_L(P)$ is a suffix of P if P ends in $Suf_L(P)$. We also define the border $Bor_L(P)$ of length L for pattern P as being a substring that is both a prefix and a suffix of P, meaning:

$$Bor_L(P) = Pre_L(P), Bor_L(P) = Suf_L(P) \tag{2.6}$$

The empty string ε is always a border of P, but ε itself has no border. The preprocessing stage of the algorithm computes an array v=$\{v[0], v[1], ..., v[M]\}$ of length $M+1$,

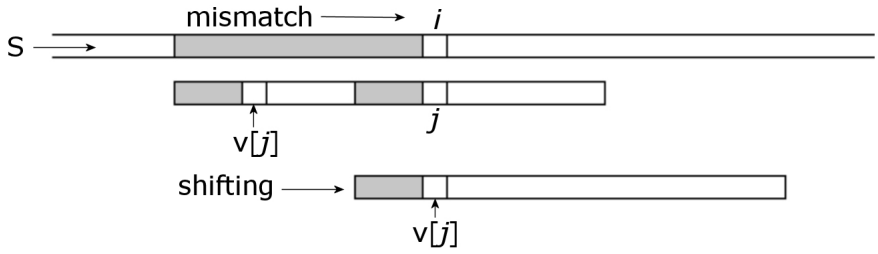


Figure 2.4: The Knuth-Morris-Pratt jump performed in case of a mismatch at position j.

so that $Bor_{v[i]}(Pre_i(P))$ is non-empty and *v[i]* is the highest value with this property (basically, *v[i]* represents the length of the widest border of the prefix of length i of P). As there is no border for the empty string, $v[0] = -1$. In the preprocessing stage, *v[i+1]* is determined by verifying if a border of the prefix $Pre_i(P)$ can be extended with the character *P[i]*, e.g. if *P[v[i+1]] = P[i]*. The list of borders that are to be examined, in decreasing order, is obtained from the values *v[i]*, *v[v[i]]*, etc. and the loop performing this task terminates when no border can be extended (that is, the value becomes -1):

- *Input*
  1: v, the array used in pre-computation
  2: S, the input string
  3: P, the pattern to look for

- *The Knuth-Morris-Pratt preprocessing stage*
  1: $i \leftarrow 0$
  2: $j \leftarrow -1$
  3: $v[0] \leftarrow -1$
  4: **while** $i < M$ **do**
  5: **while** $j \geq 0$ and $P[i] \neq P[j]$ **do**
  6: $j \leftarrow v[j]$
  7: **end while**
  8: $i \leftarrow i + 1$
  9: $j \leftarrow j + 1$
  10: **if** $P[i] = P[j]$ **then**
  11: $v[i] \leftarrow v[j]$
  12: **else**
  13: $v[i] \leftarrow j$
  14: **end if**
  15: **end while**

- *The Knuth-Morris-Pratt algorithm*

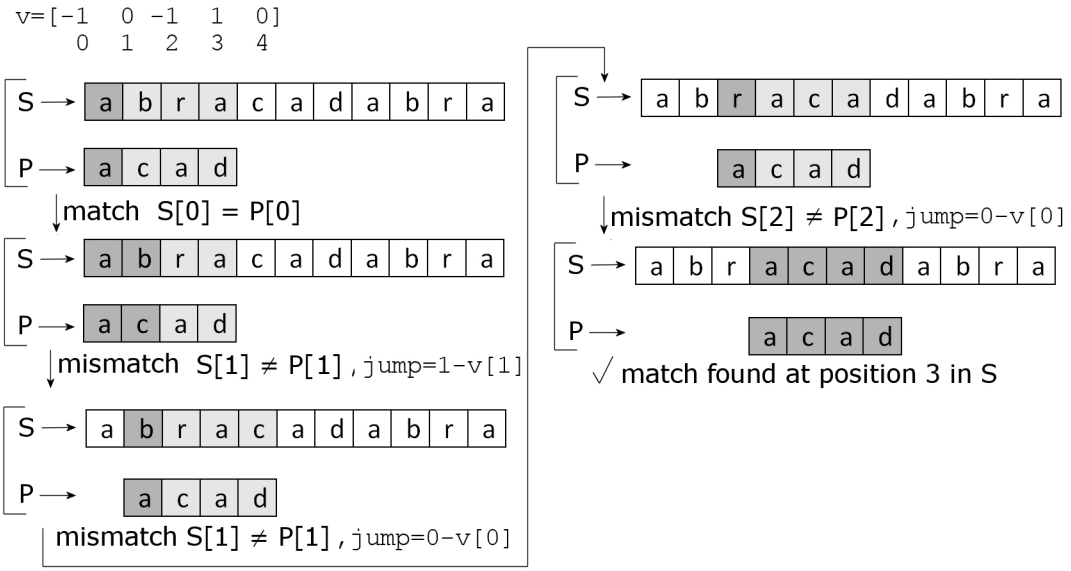


Figure 2.5: The Knuth-Morris-Pratt matching process for the example in Figure 2.1.

```
1: i ← 0
2: j ← 0
3: while i < N do
4:    while j ≥ 0 and S[i] ≠ P[j] do
5:       j ← v[j]
6:    end while
7:    i ← i + 1
8:    j ← j + 1
9:    if j ≥ M then
10:      match found at position i − j
11:      j ← v[j]
12:   end if
13: end while
```

A jumping example based on the precomputation table for the Knuth-Morris-Pratt algorithm is presented in Figure 2.4: in case of a mismatch at position *j*, the longest border $Bor_{max_j}(Pre_j(P))$ is considered. When resuming comparisons at position *v[j]*, a jump is performed and if a mismatch occurs once more, the next longest border is considered, and so on, until no border is left (*j = -1*) or the next character is a match. If all M characters are matched in the sliding window ($j = M$), a match is reported at position *i - j*, after which a jump is performed by the length of the widest border possible. As shown by Cormen et al in [Cor02], the searching algorithm has a complexity of *O(N)*, while the preprocessing requires *O(M)* steps. An example of the comparisons performed by the algorithm for our example is shown in Figure 2.5.

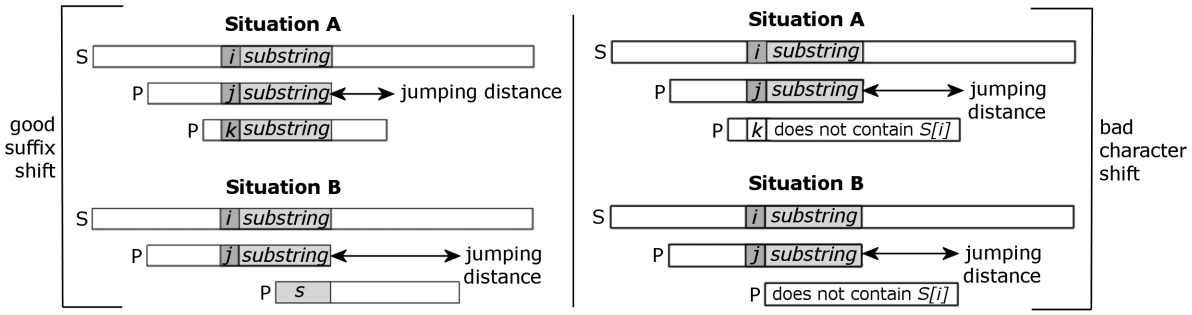


Figure 2.6: The Boyer-Moore jump distance computation for good-suffix (left) and bad-character (right) shifts.

2.1.1.4 The Boyer-Moore Algorithm

One of the fastest algorithms in practice was proposed by Boyer and Moore in[BM77] and bases its idea on performing comparisons from right to left, as opposed to usual left-to-right searches performed by most other algorithms, hoping that we can jump in this way on higher distances whenever a mismatch is found. By using two pre-computed jumping distances for the pattern (called *good-suffix shift* and *bad-character shift*), the algorithm can perform as fast as $O(\frac{N}{M})$ in the best case.

The formal definition of the jump distances for the Boyer-Moore algorithm, based on the definition proposed by Lecroq et al in [LC04], is: *GoodSuffixShift[i+1] = min*$\{s$ *> 0: (for each j,* $i < j < M$*: k* $\geq j$ *or P[j-k] = P[j]) and (if* $j < i$ *then P[i-s]* $\neq$ *P[i])*$\}$ and *GoodSuffixShift[0]* is equal to the length of the period of P. The bad-character jump distance is stored in a table of size $|\Sigma|$ (size of the alphabet) and is defined as follows: *BadCharacterShift[c]* $= \{$*min(i:* $1 \leq i \leq$ *M-1 and P[M-1-i] = c)*$\}$ if c appears in P, otherwise the value of the function is M.

The jumping distances for the algorithm are calculated as follows (see Figure 2.6 for a graphical illustration):

- For the good-suffix shift:
  1. If a mismatch occurs between *P[i]* and *S[i+j]* at position j, the segment *S[i+j+1, ..., j+M-1] = P[i+1, ..., M-1]* will be aligned with the right-most occurrence in P that is followed by a character different than *P[i]* (situation A, *P[k]* $\neq$ *S[i]*).
  2. If there is no segment that verifies this property, the longest suffix $Suf_{max}(S[i+j+1, ..., j+M-1])$ will be aligned with a matching prefix of P (situation B).
- For the bad-character shift:

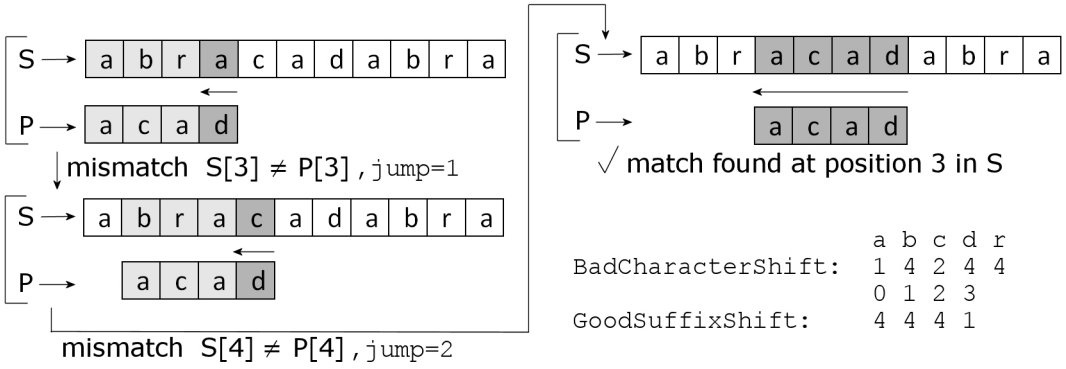


Figure 2.7: The Boyer-Moore pattern matching process for the example in Figure 2.1.

1. The character *S[i+j]* is aligned with the right-most occurrence in *P[0, ..., M-2]* (situation A, *P[k] = S[i]*).
2. If there is no occurrence of *S[i+j]* in the pattern P, the sliding window is aligned with *S[i+j+1]* (situation B).

For performing the shift, the maximum between the good-suffix shift and the bad-character shift values is used, since the bad-character jump distance can be negative. The Boyer-Moore algorithm is presented as follows:

- *Input*
  1: *GoodSuffixShift (GS)*, the pre-computed shifting distances
  2: *BadCharacterShift (BC)*, the pre-computed shifting distances
  3: S, the input string
  4: P, the pattern to look for

- *The Boyer-Moore algorithm*
  1: $i \leftarrow 0$
  2: **while** $i \leq N - M$ **do**
  3: $\quad i \leftarrow M - 1$
  4: $\quad$ **while** $i \geq 0$ and $P[i] = S[i + j]$ **do**
  5: $\quad\quad i \leftarrow i + 1$
  6: $\quad$ **end while**
  7: $\quad$ **if** $i < 0$ **then**
  8: $\quad\quad$ match found at position j
  9: $\quad\quad j \leftarrow j + GS[0]$
  10: $\quad$ **else**
  11: $\quad\quad j \leftarrow j + max(GS[i], BC[S[i + j]] - M + 1 + i)$
  12: $\quad$ **end if**
  13: **end while**

The run-time performance of the algorithm is still quadratic in the worst-case, *O(N × M)*, for a pattern containing repeated cycles. If the pattern is non-periodic however, Cole had shown in [Col91] that at most *3N* comparisons are performed.

2.1.2 Multiple-Pattern Matching Algorithms

The multiple pattern matching problem is a logical extension of the single pattern matching challenge and modifies the requirements so that the problem no longer focuses on a single pattern, but rather on multiple patterns to be searched for in the input string. While this problem could be easily (yet not necessarily efficiently) solved by applying repeated single pattern matching algorithms, there are better solutions that perform faster and have improved storage efficiency. We will focus in this section on these solutions and present some of the most widely spread approaches nowadays for solving this problem.

The multiple pattern matching problem can be formulated as follows, keeping the notations from 2.1.1: given the alphabet (a finite sequence of characters) Σ and the input string S = Γ[0], Γ[1], ..., Γ[M-1] of length N, find all occurrences of the K patterns P_i = Λ_i[0], Λ_i[1], ..., Λ_i[M_i-1] of length M_i in S, where S, $P_i \in \Sigma$, $i \in \{0, 1, ..., K\}$ and M $\leq$ N.

2.1.2.1 The Aho-Corasick Algorithm

One of the most widespread algorithms used nowadays to solve the multiple pattern matching problem is that proposed by Aho and Corasick in [AC75]. The algorithm is an extension of the Knuth-Morris-Pratt approach to multiple keywords, by constructing a finite automaton and building on top of it. The Aho-Corasick algorithm is also related, although not identical, to the notion of suffix (position) trees, a concept introduced by Weiner in [Wei73], another approach commonly used in pattern matching applications because of its ability to find and represent repeated substrings efficiently. Crochemore et al in [CR94] have described an algorithm called the Backward Directed Acyclic Word Graph, which simulates the bit parallelism of the suffix automaton (therefore, its efficiency can be observed only if the length of the pattern is smaller than the memory word length of the machine it is ran on); later on, Frederiksson in [Fre09] had provided an algorithm that represents a hybrid based on the suffix tree concept ([Wei73]) and the Backward-DAWG algorithm ([CR94]).

Cormen et al in [Cor02] have introduced the notion of finite automaton as being a 5-tuple pair. Based on this definition, we can introduce the definition of an Aho-Corasick finite state machine as being a 7-tuple *(S, s_0, S^F, Σ, goto, failure, output)*, where:

- S is a finite set of states

- $s_0 = root \in S$ is the initial state
- $S^F \subseteq S$ is the set of accepting states
- Σ is the input alphabet
- *goto* is a function defined as *goto:* $S \times \Sigma \rightarrow S$, also known as the transition function
- *failure* is defined as *failure:* $S \rightarrow S$ and represents the failure (mismatch) function
- *output:* $S \rightarrow S$ represents the output function

The basic behavior of the automaton can be expressed as follows:

1. if the automaton is in state s and reads input character c, the current state becomes *goto(s, c)* (if defined) or *failure(s)* if undefined
2. if the current state of the automaton is s and $s \in S^F$, s is an *accepting* or *final* state
3. the *output(s)* function returns true if $s \in S^F$

The algorithm is based on the trie (prefix) tree data structure, which is commonly used to store an array having strings as keys. The root of the trie tree is associated to the empty ϵ string, and for a given node v of depth d, all nodes of the trie v' with depth $d' > d$ have a common prefix of length d (Figure 2.8).

The algorithm is comprised of the following steps:

1. create a trie of all the keywords to be searched for in the input string
2. pre-process the trie by computing, for each node, the *failure* function defined earlier
   - each node v is located at depth *depth(v)* and has the word *word(v)* associated to it
   - *failure(v)* = v', where $Suf_{d'}(word(v)) = word(v')$, where $d' < depth(v')$ and d' is the biggest integer for which this property is verified (in other words, the *failure* function points to a node v', whose associated word is the maximal suffix of *word(v)*, the word of the current node)
   - if there exists no d' for which the property above is verified, *failure(v)=root*
3. perform the match process

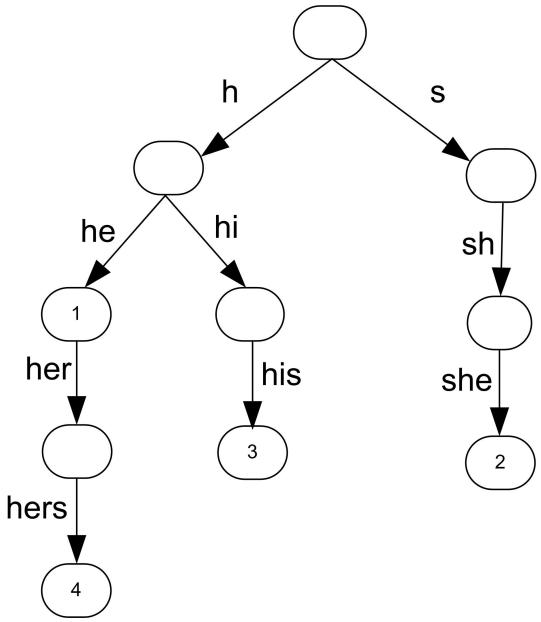


Figure 2.8: The trie tree created for the keyword set {he, she, his, hers}. Accepting states are numbered.

The algorithm is presented as follows:

- *Input*
  1: S, the input string
  2: P_i, the patterns to look for

- *The Aho-Corasick algorithm*
  1: $state \leftarrow root$
  2: **for** $i \leftarrow 1$ to N **do**
  3: **while** goto(state, S[i]) = undefined **do**
  4: $state \leftarrow failure(state)$
  5: **end while**
  6: $state \leftarrow goto(state, S[i])$
  7: **if** output(state) $\neq \emptyset$ **then**
  8: match found, $state \in S^F$
  9: **end if**
  10: **end for**

One great practical advantage of the algorithm is that it is optimal in the worst case, having a run-time performance of $O(N+\sum_{i=0}^{K} M_i+Z)$, where Z is the number of matches found. The automaton created for the trie in Figure 2.8 is shown in Figure 2.9.

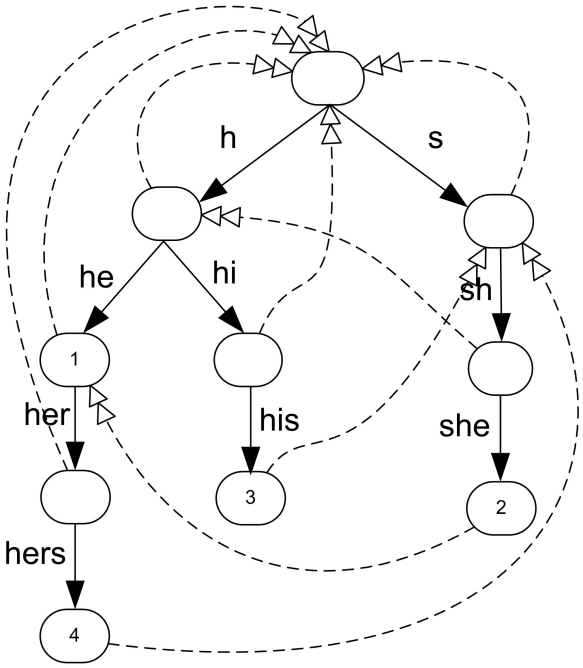


Figure 2.9: The Aho-Corasick created for the keyword set {he, she, his, hers}. Failure functions are dashed.

2.1.2.2 The Commentz-Walter Algorithm

Commentz and Walter have proposed in [CW79] an algorithm that is based on a similar idea as that proposed by Boyer and Moore in [BM77] but extended to the multiple pattern matching problem, resulting in a variant which attempts to perform larger jumps in the input string whenever a mismatch occur, if the situation allows it, based on a pre-computed model which is similar to the bad-character and good-suffix shifts used in the single pattern matching variant. In other words, the algorithm attempts to spot certain properties of the keywords and use that to compute an efficient jump table that can be used at run-time to decrease the number of comparisons and therefore achieve better running times.

The Boyer-Moore variant discussed in [BM77] had performed the search by comparing characters from right to left, as opposed to the normal direction of comparison from left to right, used by the brute-force or Knuth-Morris-Pratt variants. In a similar manner, the Commentz-Walter algorithm builds upon a trie tree constructed from the set of reversed patterns. Additionally, the algorithm introduces two new jump distances, called *shift1* and *shift2*. They are computed as follows:

- for the root of the trie, *shift1=1* and *shift2*=w_{min} (where w_{min} is the length of the shortest keyword in the trie);
- otherwise, *shift1*=min{w_{min}, {l; l=d(v')-d(v)}}, where d(v) is the depth of the current node v in the tree, and v' is an element of *set1(v)*={v'; word(v) is a

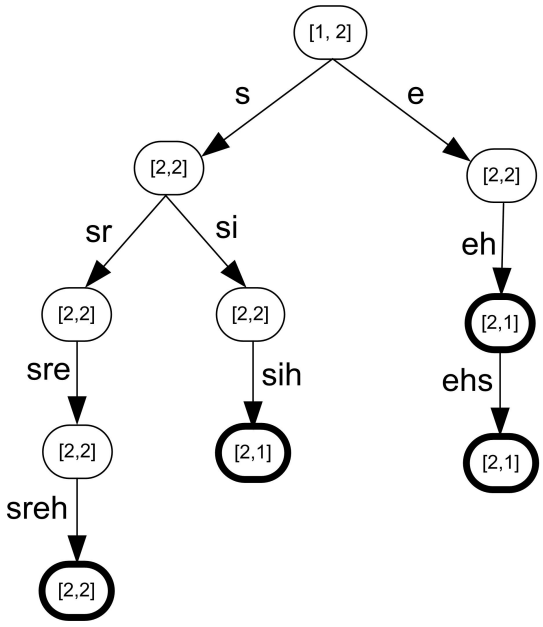


Figure 2.10: The Commentz-Walter tree for the keyword set {he, she, his, hers}. Shift distances are represented as *[shift1, shift2]* and leaf nodes are marked **bold**.

proper suffix of word(v') }.

- *shift2*=min{*shift2*(parent node of v), {l; l=d(v')-d(v)}}, where v' is an element of *set2(v)*={v'; v' is an element of *set1(v)* and v' is a leaf in the trie}.

Basically, the shift distances for a node are being computed as follows:

- the *shift1* distance is being computed as the minimum between w_{min} and the depth difference between the current node and a node of a higher depth for which the word at the current node is a suffix
- the *shift2* distance is being computed as the minimum between the *shift2* value of the parent node, and the minimum depth difference between the current node and a leaf of higher depth for which the word at the current node is a suffix

It begins scanning the input text and makes comparisons from right to left, shifting the text by a value of {min(*shift2*(v), max{*shift1*(v), *char*(*inputText*[*pos*-*depth*]-*depth*-1}} (*pos* is the current position in the file, *depth* is the current depth in the tree). The *char* function is defined as *char(c)* = min{w_{min}+*1*, $d(v)$}, where v is a node with label c. An example of how the Commentz-Walter tree looks like for the same set of patterns used in the Aho-Corasick automaton in Figure 2.9 is shown in Figure 2.10. The algorithm for performing the scanning is presented below:

- *Input*

1: S, the input string
2: P_i of length *min(w_{P_i})*, the patterns to look for

- *The Commentz-Walter algorithm*

1: *state* $\leftarrow$ *root*
2: $i \leftarrow$ *min(w_{P_i},* $i \in 0, 1, ..., K$*)*
3: $j \leftarrow 0$
4: **while** $i \leq N$ **do**
5: **while** v has child v' with label $S[i-j]$ **do**
6: $v \leftarrow v'$
7: $j \leftarrow j + 1$
8: **if** *output*$(v) \neq \emptyset$ **then**
9: match found at position $i - j$
10: **end if**
11: **end while**
12: $i \leftarrow$ *min(shift2(v), max(shift1(v), char(S[i-j])-j-1))*
13: $j \leftarrow 0$
14: **end while**

The algorithm is not optimal in the worst case and in fact, it has a worst-case behavior that is much worse than that of the Aho-Corasick algorithm, being *O($N \times w_{max}$)*, where w_{max} is the maximum length of the patterns in the input set. The variant presented is marked in the original paper [CW79] as *B*, with a different variant *B1* that uses the exact same trie as the algorithm we have discussed, but which reduces the worst-case scanning time to *O(N)* by using a buffer of size w_{max}, which holds the last few characters scanned from the input string. In practice though, as shown by Watson in [Wat95], the algorithm performs much better than its Aho-Corasick counterpart, since it can skip larger portions of text.

2.1.2.3 The Wu-Manber Algorithm

Wu and Manber in [WM94] have proposed a fast algorithm which is based on a hashing principle, similar to the Karp-Rabin approach in 2.1.1.2. The Wu-Manber multiple pattern matching algorithm focuses on the generic purpose of the matching, that is speeding up the search in the average case by implementing shift and hash tables, in a manner similar to the Boyer-Moore bad character shift approach [BM77].The algorithm uses three tables built during the pre-computation phase: a SHIFT table, a HASH table, and a PREFIX table. The SHIFT is similar to the Boyer-Moore bad character skip table [BM77].

The algorithm considers blocks of text instead of single patterns. If M is the sum of the keyword lengths and c the size of the alphabet, then the block length B is optimized for running time and space when B=$\log_c 2M$ (in practice, B=2 or B=3

are recommended). The SHIFT table determines the shift based on the last B bytes rather than just one. A hash function maps blocks to an integer used as an index in the SHIFT, whereas the total space required here is $O(c^B)$ (for B=2, we have 2^{16}=65,536 elements, and for B=3 we have 2^{24} elements).

If S has the B bytes currently being scanned in the input data, the algorithm takes into account two different situations:

- if S does not appear as a substring in any pattern, SHIFT[*hash(S)*] = w_{min} - B + 1
- otherwise, find the rightmost occurrence *pos* of S in any of the keywords, then SHIFT[*hash(S)*] = w_{min} - q

The matching phase works as follows:

1. compute hash value based on current B bytes of data (e.g. *input[i-B+1] ... input[i]*)
2. if SHIFT[h] > 0, shift text by this value and return to step 1, otherwise go to next step
3. compute hash value of prefix of text (read W_{min} bytes to the left), store as *hashPrefix*
4. for each p where HASH[h] $\leq$ HASH[h+1], if PREFIX[p] = *hashPrefix* check actual keyword (*pattern[p]*) against the text directly, output matches (if any)

2.2 Intrusion Detection Systems

Scarfone et al in [SM07] define an intrusion detection system (or commonly known in literate under the acronym IDS) as being the process of monitoring all events which are occurring in a computer system or network, followed by the analysis of possible or imminent violations of security policies, with either of them being related to computer security, acceptable use or standard security practices. Intrusion prevention is the process of detecting violations of such policies and preventing them from occurring, while eradication refers to eliminating all violations of such sort and restoring the system to its original, pre-intrusion state.

Malware (a term used for ***mal****icious soft****ware***) is a particular type of program or software aimed at damaging or offering access to a victim's computer. The term is used nowadays to express a wide variety of types of dangerous, intrusive or harassing software. The degree of malicious activity that a software contains is based on the perceived intent of the creator and not on functionality of the software itself. The most common types of malware existing today are: computer viruses, spyware, adware,

trojan horses, worms, rootkits and others. In digital forensics and law, the term *computer contaminant* may be found as describing malware, with several U.S. states (for example, California and West Virginia) containing a description of such term in their legal code ([Malb, Mala]).

It has become a common practice during time for many malware programs to disguise themselves as being genuine products, while the actual purpose remained hidden. Such software can cause damages to the operating system it is running on (e.g. many viruses caused damages to the MS-DOS operating system back in the 90s and even changed or completely eradicated the bootloader, a critical component that loads the operating system in memory on computer startup), can disrupt normal activities during work (e.g. commonly known as adware and spyware programs), or can even go as far as damaging internal hardware devices (e.g. one of the first viruses to cause major damage was CIH, also known as Chernobyl or Spacefiller according to [Wikb], causing more than 60 million computers to be infected and an estimated of more than 1 billion dollars in damage).

Another important area of research for intrusion detection systems are network IDS. A network IDS is an intrusion detection system which aims to detect unauthorized access to a computer network through traffic analysis. Network IDS are important to maintaining a secure network and preventing attacks on it, such as those of type Denial-of-Service (DoS). Denial of service attacks (or distributed DoS attacks) are attempts to make the a network or computer resource unavailable to its users, either temporarily or indefinitely. The common victims of such attacks are usually represented by banks, domain nameservers, payment gateways, high-profile websites and so on. The term is not limited to the network IDS terminology, but also finds applicability in the management of CPU resources, as discussed by Fledel et al. in [FC10].

2.2.1 Classification

Early intrusion detection system classifications were proposed in the late 1980s by Lunt [Lun88], followed by similar classification work in the early 90s by McAuliffe et al. [MH90] and Esmaili et al. [Esm95], with more in-depth reviews by Jackson et al. in [Jac99].

According to Rahman in [Rah00], there are two large categories of intrusion detection systems:

- **Statistical anomaly based** systems, which evaluate the incoming data and attempt to quantify, using one or more algorithms, the anomalous activity which characterizes the data. If this classification takes the anomalous value associated with the data over a certain threshold, the data is classified as being anomalous and is probably therefore discarded.

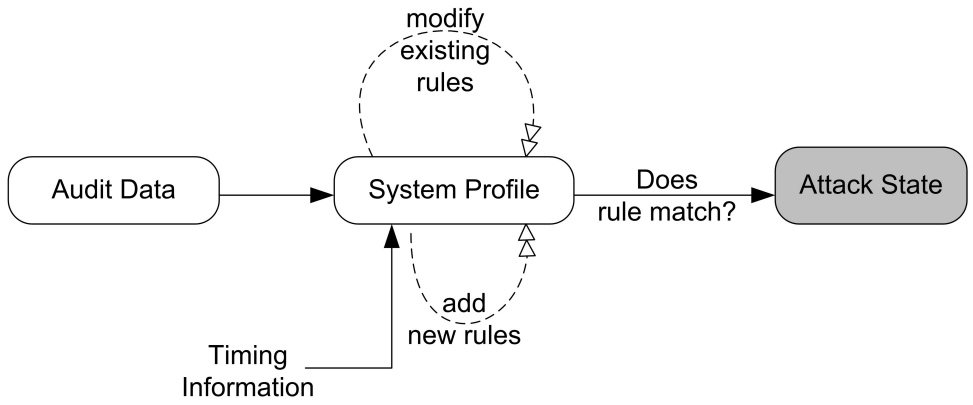


Figure 2.11: A typical misuse detection system (according to [Rah00]).

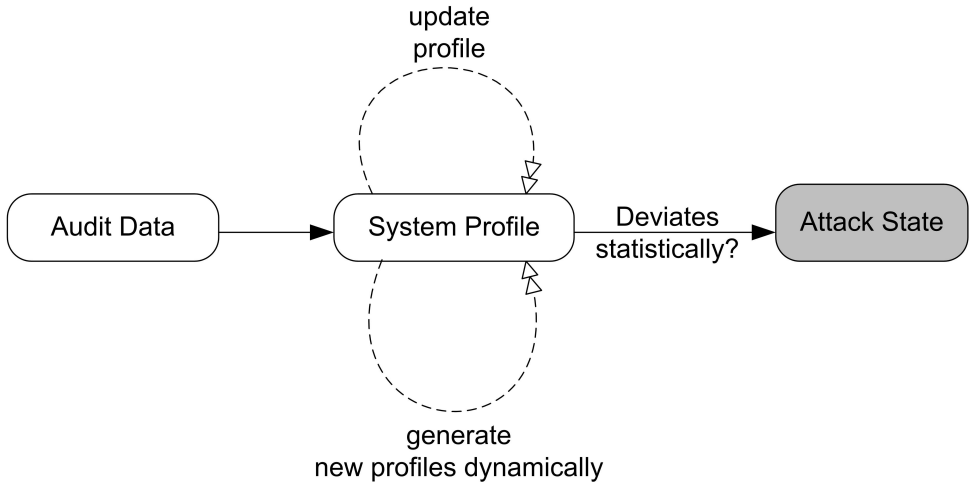


Figure 2.12: A typical anomaly detection system (according to [Rah00]).

- **Pattern-based** systems, or **signature-based**, which attempt to classify the incoming data by comparing it to a list of known malicious activities, described by the signatures or patterns mentioned. The system is, in this case, dependant on an existing pattern database which characterizes the known malicious behaviors.

A typical misuse detection system is shown in Figure 2.11, while an anomaly detection system is depicted in Figure 2.12. Debar et al in [DHW99] have proposed a taxonomy for classifying intrusion detection systems, presented in Figure 2.13. In the taxonomy proposed, they focus on two types of characteristics: functional and non-functional. The functional characteristics refer to three types of IDS, based on detection method, the behavior and the audit source location (an element mostly common to network IDS).

The taxonomy of Debar et al. from [DHW99] has been later on revised in [DH00] (by Debar et al. themselves) and by Axelsson in [Axe00]. The revised taxonomy of

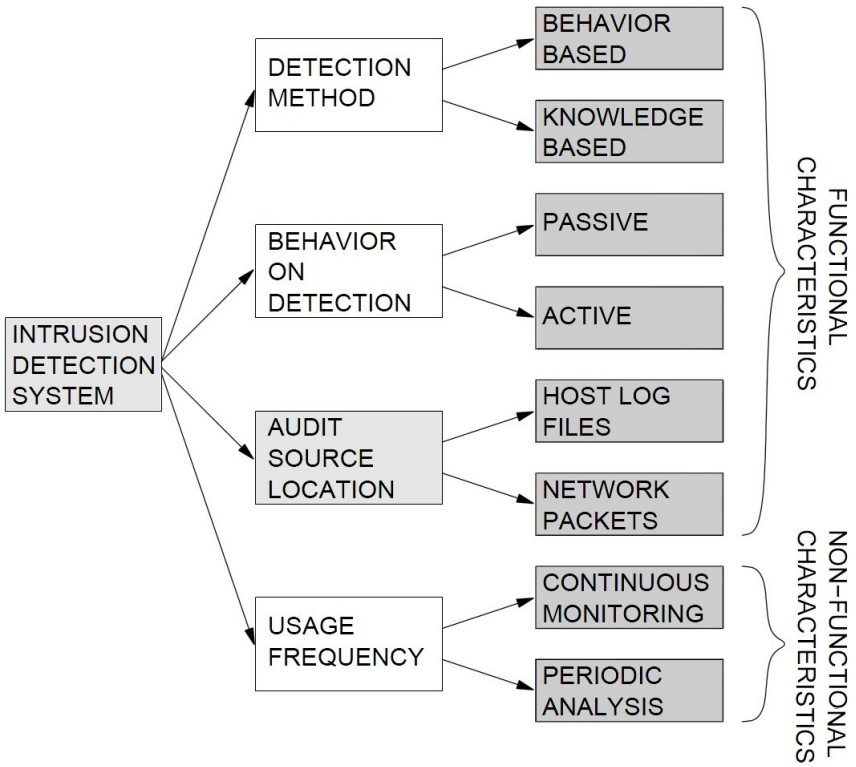


Figure 2.13: The taxonomy for IDS classification as proposed by Debar et al in [DHW99].

Debar et al. in [DH00] adds one new type of intrusion detection system, based on a *detection paradigm*, which is *state* (where the IDS attempts to classify the state as being a failure or an error state) or *transition* based (where the IDS is being monitored for all states that may be representing an intrusion), both supporting proactive and nonperturbing evaluation.

2.2.2 Detecting Malicious Code Behavior

The problem of detecting malicious code is a critical component of modern antivirus engines nowadays and focuses on detecting pieces of code belonging to an executable file that are dangerous to the system or represent an imminent threat to it. Such malicious code can be sometimes difficult to track down and eradicate, and is most commonly found in viruses, worms and trojan horses.

Detecting and removing malicious code can be a difficult task to accomplish, as pointed out by Tanase from Symantec Corp. in [Tan10]. The most important part in keeping the system safe is backing up permanently the important data, as later on there is very little room for error when tracking down intrusions and sometimes harmful compromises need to be made (e.g. files may need to be deleted as they become unrecoverable, or data is lost as a consequence of the intrusion). In order

to determine the malign behavior, a step-by-step analysis of the evidence needs be made, since most intruders leave traces of information behind which could be picked up by an in-depth forensic analysis, for instance logs, access file times, and so on.

As we had already outlined in 1.1, the most obvious example of malicious code is represented by computer viruses. Polymorphic viruses and metamorphic viruses pose many additional challenges, since polymorphic viruses partially encrypt their own code to hide its detection from signature-based antivirus engines, while metamorphic viruses shift their code and structure permanently, creating completely new copies of themselves which keep the malicious functionality intact, yet are completely different than the previous copy of the virus that had propagated. Determining whenever an intrusion occurs can be usually conducted as a survey comprised of several steps, as pointed out in [Tan10], paying extra attention to the details that might show information about a potential perpetrator. Here are some generic guidelines to how malicious behavior and intrusions can be detected:

- A first step would be the log analysis of the machine and, if possible, taking down the machine during the process to prevent further damage. Since some systems cannot be taken down too easily without disrupting certain types of services that they offer, the logs may have to be analyzed separately. And even in such a situation, there is always the possibility for an intruder to alter, delete or remove clues from the logs. Furthermore, certain logs are vast and the analysis may not be possible by human operator intervention, but using forensic tools or utilities. Any unusual or suspicious traces of activity must be captured with great attention to detail and later on analyzed for indications as to how and where the intrusion occurred.

- A next logical step is the behavior analysis of the system. Discovering anomalies can be easier if there are changes in the running configuration of the system, and such information is usually observed by investigating the running processes, the amount of resources (both in terms of CPU usage and memory usage) a process uses, how often a job is scheduled to execute and to whom it belongs (for instance, system-scheduled jobs that require high CPU priority and run at a pre-defined regular interval could mean a potential breach in the system, a risk commonly found in a trojan horse's behavior).

- Network traffic analysis and network resource utilization is also an important step for locating intrusions. Certain types of malware (worms, trojan horses, rootkits, etc.) open ports on a victim's computer and sometimes also establish permanent connections to a certain server from where the victim computer may be manipulated easily. Observing network data traffic, port usage and the running services on a machine can be therefore a vital component to tracking down abnormal behavior.

- Observing file system changes can offer clues of vital important to tracking down virus infections. Viruses replicate from one executable code to another, and keeping hashes of files can be a good practice in observing whenever file changes occur - an approach that had been used by older antivirus engines, such as the Norman Thunder-Byte AntiVirus (TBAV), and which is still found in certain modern antivirus engines today. Since viruses can easily change the access and modified times of a file, the only way to determine if an infection had occurred remains comparing the hash obtained by analyzing the file's contents at regular time intervals: if the hashes differ, the file contents has changed which may suggest an infection of some sort; if a pattern can be observed in the changes performed on the files (e.g. only executable files have changed from the last hash verification), then the risk of an infection can increase exponentially and may suggest that immediate action is required.

Shevchenko from Kaspersky Lab in [She08] discusses a model for detecting malicious code behavior, starting from the premise that any architecture of such type is comprised of two components: one that is analytical, and one technical, pointing out that in practice for certain infections such components may not be clearly separated, although in terms of functionality their differences are important. The technical component represents the set of program algorithms which select the data to be analyzed by the analytical component (the data itself varies and can be of several types, from program actions to file data contents), while the analytical component acts as a decision-making system, issuing after the analysis a verdict on the threat level of the data analyzed.

One of the simplest techniques for finding infections is looking for virus signatures inside an executable file's contents. In this scenario, the technical component discussed in [She08] collects information about the file and passes the data to the analytical component, which compares sequences of bytes in the files with the known signatures in order to determine whether an infection exists or not. A separation of the analytical and technical component is shown in Figure 2.14, where the HIPS acronym stands for Host Prevention Intrusion System.

The emulation module in Figure 2.14 is a component widely spread today and relies on breaking down the program into commands, analyzing these commands and issuing a threat level for their effects. This way, the program's behavior can be observed without posing any threat to the operating system or to the system itself. Emulating engines exist in most, if not all, modern antivirus engines, although sometimes they may be a part of a virtualization engine (also called a *sandbox*). The sandbox offers real-world access to the executing program, but poses many additional limitation from preventing any damage from happening, for instance denying access to vital operating system files or resources during run-time. The level of abstraction

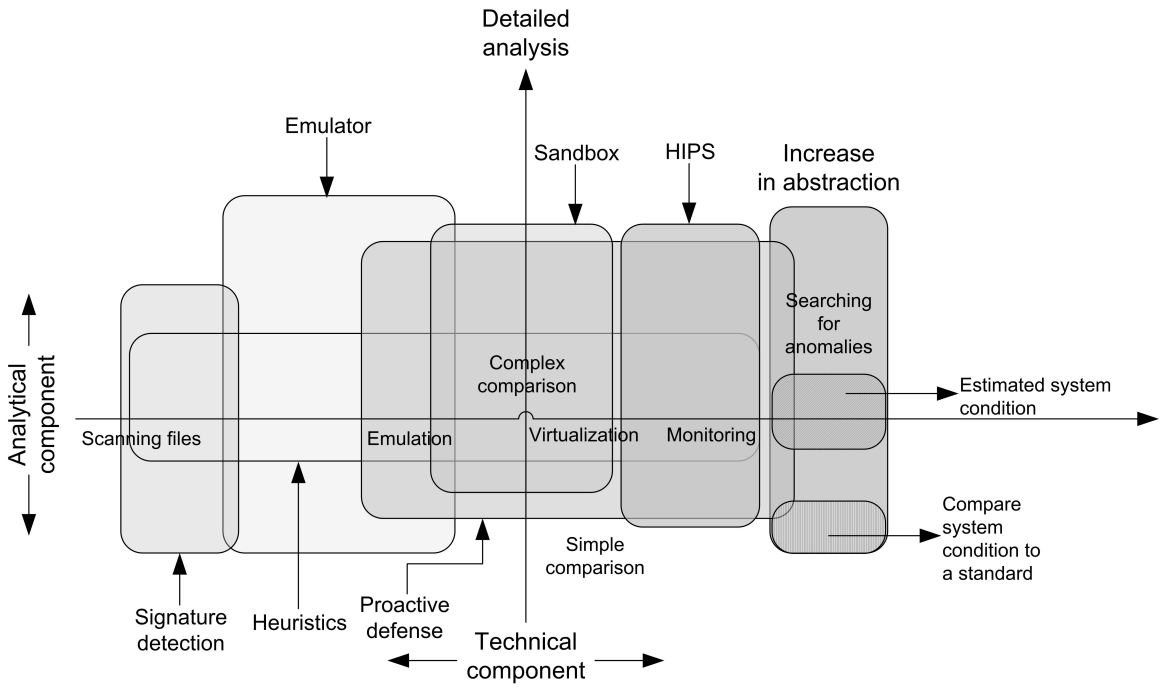


Figure 2.14: The analytical and technical components used in intrusion detection systems as proposed by Shevchenko [She08].

increases with the monitoring of system events and the final element of the architecture is represented by the system anomaly scanner, which determines if the operating system's state is healthy or if it has been compromised in any way as a result of an infection.

One of the most spread open-source antivirus programs today is ClamAV [cla12]. One important advantage towards research in the field of computer security and malicious code detection in particular is the permanently updated database (usually one or more times per day), with a large number of malware signatures, which are used in many studies for comparison and improvement to the existing solutions.

2.3 Digital Forensics

Digital forensics, according to Reith et al. in [Rei02] and Carrier et al. in [Car02], is part of the *forensic science*, a branch of science focused on the investigation of material found in digital devices, along with its potential recovery, and is often related to crime investigations, also called *cyber-crime* [cyb]. The president of the United States, Barrack Obama, had declared in [Oba09] that the number of threats his country is dealing with "is one of the most serious economic and national security challenges we face as a nation". Verizon, the Dutch High Tech Crime Unit and the U.S. Secret

Service had produced a report in 2011 [ea11], where they had stated that the amount of intrusions (hacking and malware combined) was at the highest level ever at that time.

Palmer in [Pal01] had described the first steps to performing work in digital forensics as being *preservation, collection, validation and identification.* The full abstract digital forensics model proposed by Palmer in the same paper defines as key components the following though: identification, preparation, approach strategy, preservation, collection, examination, analysis, presentation and returning evidence. *Identification* refers to determining the type of an incident after successfully identifying it, *preparation* refers to preparing search warrants and the tools required, the *approach strategy* refers to defining a strategy which is capable of collecting untouched evidence without affecting the victim or interfering with it as little as possible, while *preservation* refers to isolating, securing and preserving the digital evidence. The *collection* of digital evidence refers to duplicating the evidence using standard forensic methods, so that the *examination* can be performed afterwards through an in-depth review of the collected evidence. The last two steps are the *presentation* of the evidence, which exposes a summary of the conclusions achieved as a result of all previous steps, and *returning the evidence* once the process has completed. Returning the evidence is not an explicit step in digital forensics and it is not used often in the investigation process.

The real-world impact of digital forensics is significant and we will discuss as follows a few applications of the field to different areas of interest for this thesis.

2.3.1 Cyber-Crime Investigations

One of the first major applications of digital forensics is cyber-crime investigations, where clues that may be vital to a trial can be found and prosecutors may be offered the proof they need to motivate the charges. As pointed out by Nelson et al. in [NC10], the evidence found as a result of the investigation can be *incriminatory* (also called *inculpatory*) or *exculpatory*, meaning the suspect is being cleared of the charges. While most of the time it is sufficient to use software forensic tools for looking up pieces of information useful in the investigation process, sometimes it may be required to use state-of-the-art technology to retrieve lost or deleted data, with the costs starting from $3,000 and going up to $20,000 or even more.

Some of the most common crimes involving computers include covert financial operations (e.g. money laundering from drug operations), human trafficking, electronic fraud, identity theft, etc. Whenever the evidence is deleted *a priori* by the suspect, data forensic technologies help with its recovery and provide essential clues to the prosecution.

Along the technical side of the cyber-crime investigations itself, which is the main subject of interest for this thesis, there is also a social and ethical side to the problem,

with direct impact in law, a topic thoroughly discussed by Harrington in [Har11]. The author discusses the challenges existing when co-working with a digital forensics expert and discusses the moral and ethical aspects that had appeared throughout time in several such cases.

2.3.2 Data Recovery

A second major application of digital forensics is in the data recovery process. While computer forensics and data recovery are separated domains, according to Nelson et al. [NC10], the techniques in computer forensics can be successfully be applied to help improve the data recovery process. Such a process might be necessary in order to recover important business-related data that had been accidentally or unintentionally lost, but may also be useful in the cyber-crime investigation process when attempting to recover files or photos erased on purpose by potential criminals in order to hide their traces.

Data loss is affecting business of all sizes, but the financial losses are usually reaching their peak when we are discussing medium or enterprise-sized businesses. While data loss has a number of potential causes (e.g. hardware defect, faulty human intervention, mechanical error, etc.), the consequences can be of great importance to how the business is further conducted and sometimes can mean bankruptcy if the matter is not treated seriously.

Although storage devices have evolved during time, with the classic hard-disk drives being replaced slowly by the non-mechanical solid state disk (SSD) devices, which also offer the high performance benefit at a much higher cost, both types of storage are prone to hardware or logical failures, which directly affect the reliability and persistence of the data in question [Pun12]. While sometimes data recovery is impossible given the amount of damage produced (in which case, the backup solution remains the only viable approach to securing a business's assets in time), most of the times partial data recovery is possible through the use of forensic investigation tools (for instance, when logical errors exist in the filesystem, or when the partition table has been damaged).

The financial aspects of data recovery are also worth mentioning, given that according to [IoM], a typical recovery scenario costs from $5,000 per day, and can vary with the degree of damage to the data. In 2009, a study [HA] discussing the average SSD data recovery costs had estimated the amount of $2,850 per damaged SSD, with a timeframe to completion of about three weeks.

A series of best approaches for data leakage protection is being discussed by Bunn et al. in [Ben11], where the authors discuss a series of steps required for protection, such as developing the incident response plan, establishing an emergency response team and testing the plan. They also discuss internal and external investigations and evaluate a series of products for performing the forensic analysis.

2.4 Summary

In this chapter we have discussed a series of the most common pattern matching algorithms used in malicious code detection and digital forensics in Section 2.1, discussing both single pattern matching (Section 2.1.1) and multiple pattern matching (Section 2.1.2) variants, we have presented a classification of intrusion detection systems (Section 2.2) and a discussion on how malicious code can be detected (Section 2.2.2), and have introduced digital forensics in Section 2.3, along with some of its potential applications and real-world impact. We have outlined the most important aspects of an IDS, discussing some of the most commonly known classifications (Section 2.2.1) based on the works of Rahman in [Rah00] and Debar et al in [DHW99], and discussed the basic steps in a forensic investigation as defined by Palmer in [Pal01] (Section 2.3). Lastly, we have presented two of the major applications of the digital forensics process, particularly in fighting cyber-crime investigations (Section 2.3.1) and in the data recovery process (Section 2.3.2).

Chapter 3

An Efficient Mechanism for Storing Very Large Automata in Hybrid CPU/GPU-Based Memory

Contents

The rapid progress of technology in the last years had determined the sudden growth in processing power of both CPUs but also GPUs, with performance nowadays in GPUs that can far exceed the bandwidth of a normal CPU. The immediate advantage is that the CPU can be offloaded with certain computationally intensive tasks, while the GPU can handle them without disrupting the normal user's work.

The two major players in the graphics card industry, nVIDIA and Advanced Micro Devices (AMD, former ATI), have both announced graphics cards that are programmable, nVIDIA when introducing the CUDA [cud12] programming framework for their graphics cards, followed by AMD with the AMD FireStream [AMD], a streaming multiprocessor to use parallel stream processing for heavy computations.

3.1 The CUDA Architecture

CUDA (Compute Unified Device Architecture) [cud12] was first proposed by nVIDIA in 2007, offering versions for both the Windows and Linux operating systems, while version 2.0 contained support for MacOS X also. CUDA was designed to be of use in both professional and casual graphics cards, having similarities with the OpenCL [Ope] (Khronos Group) framework and with Microsoft's DirectCompute [Dir] alternative. Figure 3.1 presents the overall CUDA architecture.

The latest GPUs in the nVIDIA line, starting with the G8X, can perform computations similar to a microprocessor using the CUDA programming model, however there is a particular feature to CUDA that makes it different than normal programming on a CPU: GPUs are designed to run multiple (but slower) execution threads, in parallel, compared to running one single (but much faster) thread as a normal CPU. Nonetheless, given the massively parallel architecture of GPUs today, the throughput of a GPU can greatly exceed (by a number of times) that of a CPU. The ability to perform CPU-alike operations on a GPU is often referred to as *general-purpose computing on graphics processing units*, or GPGPU [GPG].

3.1.1 The GF114 Streaming Multiprocessor

The GF114 streaming multiprocessor is commonly found in graphics card in the nVIDIA GTX 560 series. For the tests in this thesis, we have used a middle-range GeForce GTX 560 Ti graphics card, based on the Fermi [Fer] architecture, built in a 40nm fabrication process and having an estimated of 1.95 billion transistors in a die size of just 360mm^2. The theoretical peak memory bandwidth of the GF114 is 128.3 GB/s, while overclocked versions can reach up to 146.6 GB/s according to [Tec]. The chip's overall architecture is presented in Figure 3.2.

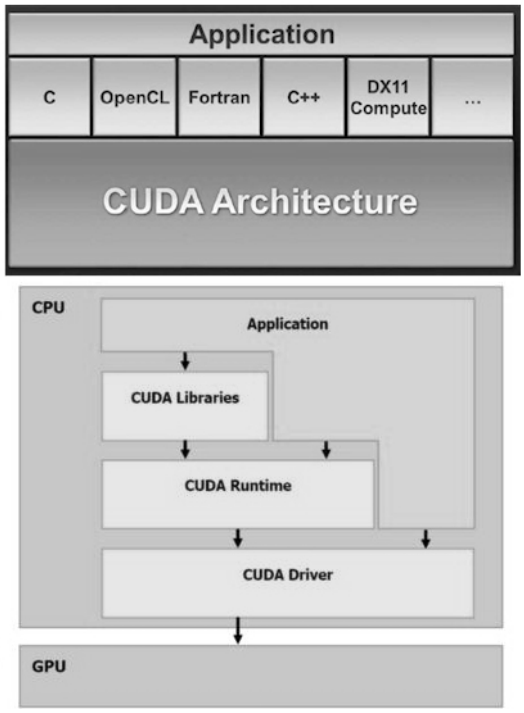


Figure 3.1: The CUDA architecture [cud12].

The GTX 560 Ti graphics card has 8 streaming multiprocessors (also called SMs), with 48 cores per each SM, as shown in Figure 3.3.

In GPGPU terminology [GPG], the term *host* refers to the CPU, while the term *device* refers to the GPU. Both have their own types of memory, with the host using the RAM memory, while the device uses the video memory, that is usually much more limited.

The GF114 chip has both *shared* and *global* memory. Global memory is accessible to all threads, but accessing it is much slower than when working with shared memory, which is shared by all threads in a block and has the lifespan of the block. While *registers* are the fastest form of memory on the multiprocessor, they are only accessible from a thread and have the lifetime of the thread - similar to these, *shared memory* (which can go up to 48KB per multiprocessor as of now) can be just as fast as a register in certain situations (reading from same memory location, etc.), it is accessible from any thread of the block from which it was created and has the lifespan of a block. On the other side, global memory is much slower (up to 150× to 200× slower, but it is accessible from both the host and the device and has the lifespan of the application itself. *Local memory* belongs to a thread, can be still as slow as global memory (since it resides in it), and is only accessible by the thread (having therefore the lifespan of the thread).

An optimization problem common to GPU programming using the CUDA architecture refers to uncoalesced reads and writes from and to the global memory, given

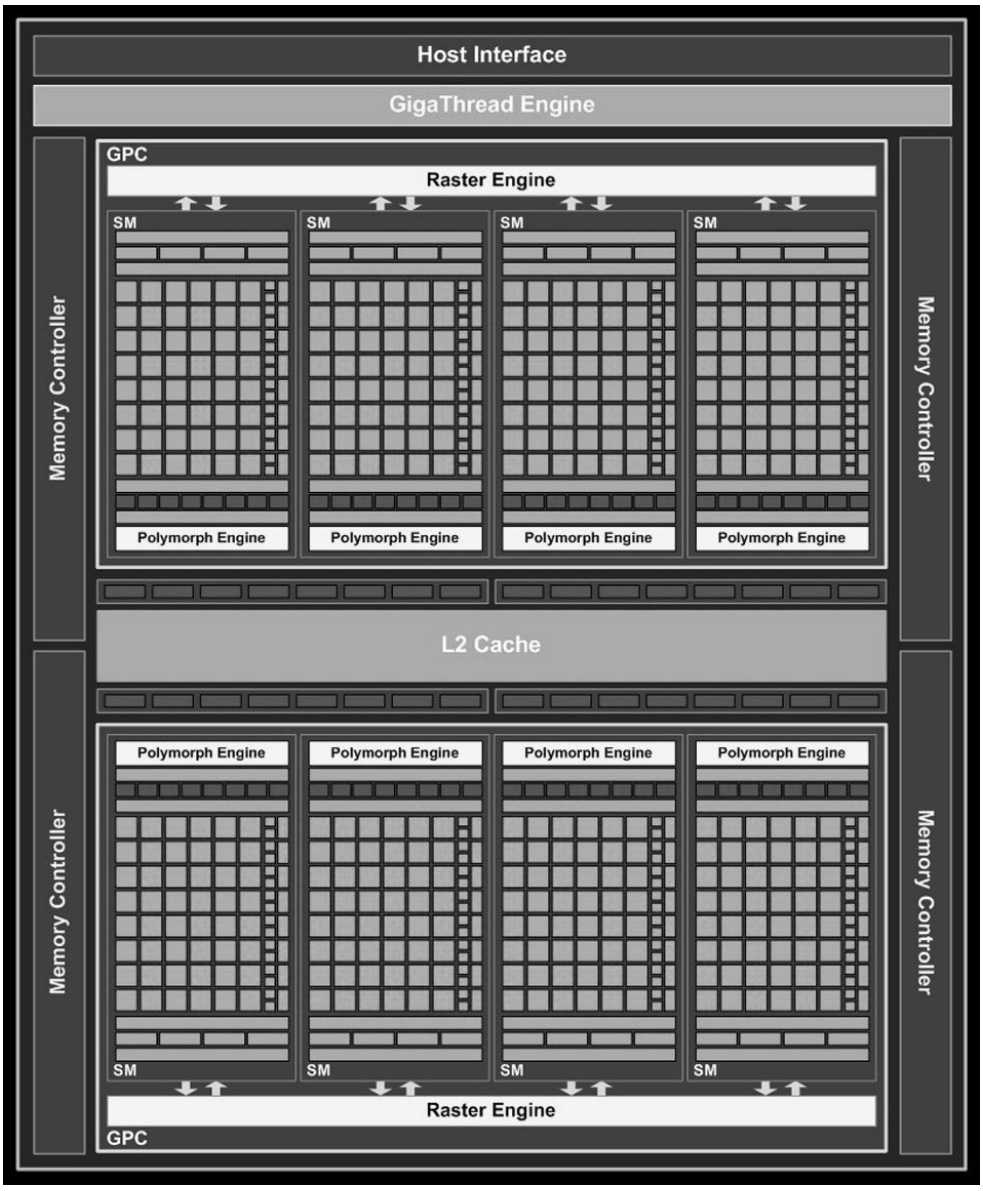


Figure 3.2: The GTX 560 Ti overall architecture [Tom].

their slow performance compared to the shared memory. In order to significantly optimize the performance and throughput of the algorithms, it is recommended therefore to use the shared memory resources (limited as they are) and avoid dispersed reads and writes if at all possible.

3.1.2 Experimental Testbed

For our implementation and experiments throughout this thesis, we have used an ASUS GeForce GTX 560 Ti DirectCU II Top graphics card, having 384 CUDA cores, equipped with 1 GB of DDR5 memory and having an engine clock of 900 MHz, with a memory clock set to 4,200 MHz and a shader clock of 1,800 MHz. To compare the results against a normal CPU, we have used an overclocked version of the Core i7 2600K CPU based on the Sandy Bridge architecture [San], clocked at 4.3 GHz.

3.1.3 The Programming Paradigm

The fundamental programming execution unit in the CUDA architecture is the *thread*. A *thread* is a piece of code being executed sequentially. A *block* in CUDA can contain from 64 to 512 threads, and the recommended occupancy according to the CUDA Occupancy Calculator [CUDa] is about 66.66% for the GF114. For our GPU, in which a block contains at most 192 threads, that results into 128 threads per block, which is the amount we have used in all our implementations and experiments. Data is being processed in *warps*, which are groups of 32 threads in GF114, the minimum size of the data processed in SIMD by a CUDA multiprocessor. Warps which are currently active are scheduled using a thread scheduler, which periodically switches between warps using a proprietary scheduling algorithm, in order to increase to make efficient use of the computational power offered by all multiprocessors.

Blocks are organized as linear, two-dimensional or three-dimensional grids (Figure 3.4), and the code runs in the GPU through *kernels*. A *kernel* is a function being executed by one thread in the GPU, but multiple threads are executing the same kernel at the same time. Since the GF114 chip has 8 streaming multiprocessors with 48 warps per SM and 32 threads per warp, it could in theory run up to $8 \times 48 \times 32 = 12{,}288$ threads in parallel.

3.1.4 Atomics

Atomic operations in parallel programming are operations which are performed without interference from other threads, in order to prevent *race conditions*, which are common problems in multi-threaded applications. According to [Rac], a *race condition* or *hazard* is a flaw in a process causing the output result of the process to be "unexpectedly and critically dependent on the sequence of timing of other events".

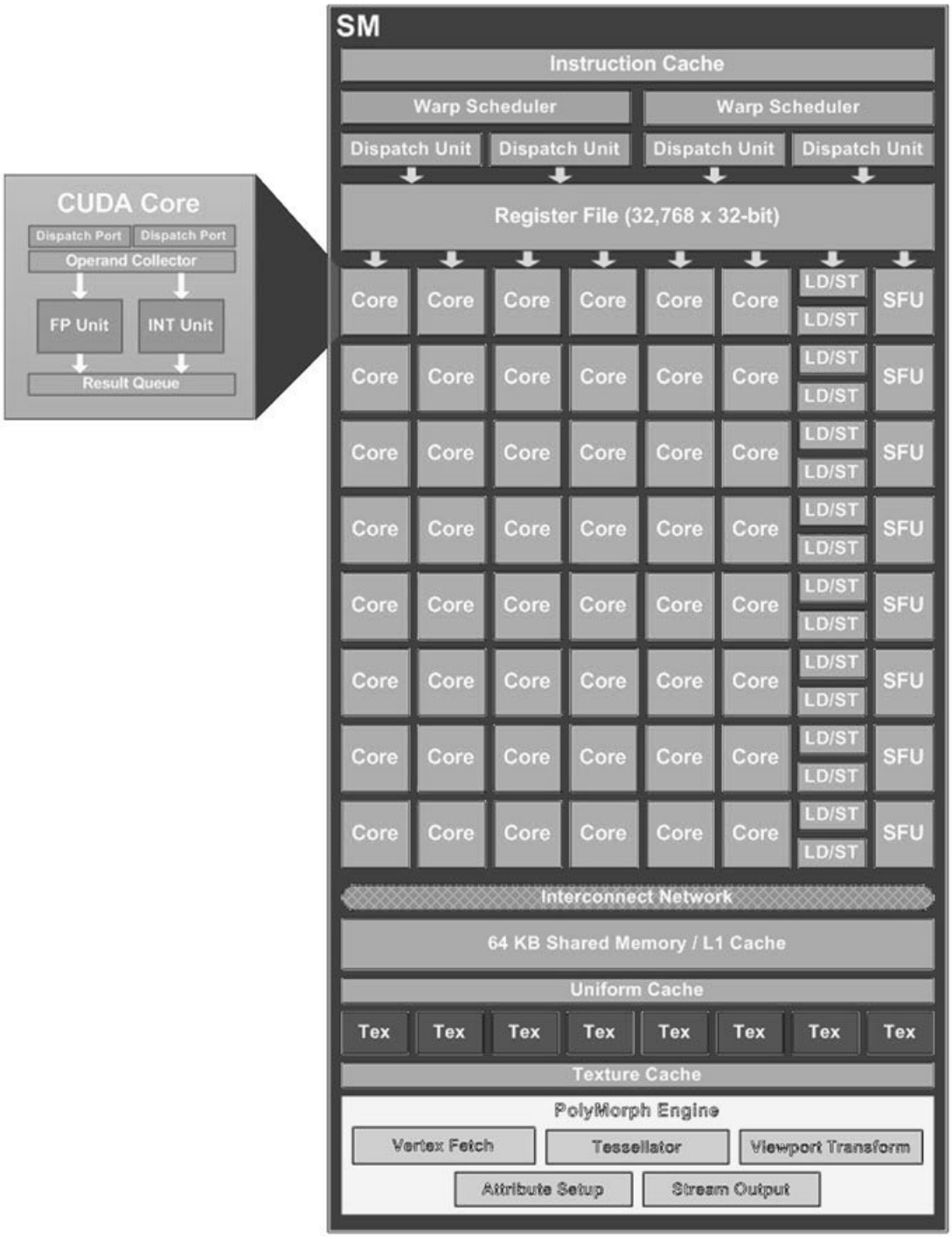


Figure 3.3: The GF114 streaming multiprocessor architecture [Tom].

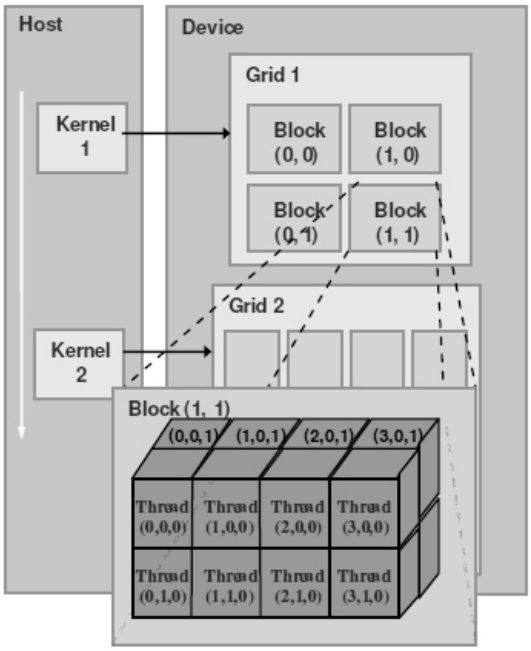


Figure 3.4: The CUDA thread model [CUDb].

As a simple explanation, whenever a thread attempts a increment a location at a memory location, it issues a few simple logical steps in accomplish that: it reads the value at that memory location (step 1), it increments it (step 2) and then writes it back (step 3). If no race condition would exist, and the thread would be stopped by another executing thread immediately after finishing step 2, but not yet reaching step 3, and the other executing thread increments the same variable at the same memory location, then in the end when the original thread resumes, it will write back the original value incremented only once, instead of it being incremented twice (by both threads therefore).

The CUDA architecture supports several atomic operations that are capable of reading, writing, modifying and writing a value back to a memory address without the execution of such an operation being interrupted by another thread, which therefore guarantees that a race condition does not occur. Atomic operations applied to global memory addresses ensure that any two threads, even from different blocks, can access that memory location with the race condition safely avoided.

We will exemplify with the *atomicMax()* function from the CUDA C Programming Guide [CUDc]. The prototypes of this function are the following:

```
1 int atomicMax(int* address, int val);
2 unsigned int atomicMax(unsigned int* address, unsigned int val);
```

The function reads the 32-bit word **old** from the memory address indicated by **address** (which can be located in both shared or global memory), computes the

maximum between **old** and **val**, and stores back at the same memory address the maximum of the two, returning the **old** value. A similar behavior is common to many other atomic functions in the CUDA SDK as well.

3.2 The ClamAV Signatures

ClamAV [cla12] is an open-source initiative with a large database of malware samples. Security researchers throughout the world are using many of these samples as a testbed when testing experimental algorithms or when proposing new solutions for malicious code detection.

The ClamAV signature database is comprised of two different types of signatures: a daily updated database, which is smaller in size (for easier updates), called *daily.cvd*, and a primary signature database which is larger in size, called *main.cvd*. Using the *sigtool* that ClamAV comes with, one can extract these two files from the database and inspect their contents, therefore obtaining access to the actual signatures. There are also other types of signature files that ClamAV uses, which we will discuss in Section 3.2.1.

3.2.1 The Signature Formats

According to ClamAV's own internal documents [Cre], the signatures formats that ClamAV understands are split into several types:

- For **.hdb** files:
  - Format: **MD5:Size:MalwareName**
- For **.mdb** files:
  - Format: **PESectionSize:MD5:MalwareName**
  - Example: *237568:ce914ca1bbea795a702185443166362 3:Trojan.Bagle-328*
- For **.ndb** files:
  - Format: **MalwareName:TargetType:Offset:HexSignature**
  - *TargetType* can be any of the following: 0 (any file), 1 (PE), 2 (OLE2 component, e.g. VBA script), 3 (HTML, normalized), 4 (mail file), 5 (graphics), 6 (ELF, UNIX format), 7 (ASCII text file, normalized)
  - Example: *Trojan.Exchanger:1:EP+229:e81c000000e8e 6ffffff81c34e8dbffffffe 846ffffffe2e4*
- For local signatures (ClamAV 0.94 and onwards), **.ldb** files:

- Format: **SignatureName; TargetDescriptionBlock; LogicalExpression; Subsig0; Subsig1; Subsig2; ...**
- Example: *Worm.Godog;Target:0;((0—1—2—3)& (4));(0);(1);(2);(3);(4); ... (signatures following) ...*

For **.mdb** files, the *pe-sig* Ruby script [pe-] is commonly used to produce signatures for each section inside a Portable Executable (PE) file.

For antivirus programs, the signatures posing the most interest are those based on MD5 hashes [Riv92] and the ones containing opcodes that need to be matched inside a program executable's contents. In Section 3.2.2 we discuss how ClamAV stores such signatures.

The hash-based signatures ensure a fast and accurate match for most trojans and worms, but cannot detect the majority of viral infections, since the hash is being calculated for the entire file in question, while polymorphic and other viruses may shift their code and therefore trigger the hash to change from one infection to another. The opcode-based signatures can be significantly complex and may also contain several regular expressions in them, which allow for greater accuracy when performing a match. The regular expressions used are the following:

- **?**, used to match a single half-byte
- **??**, used to match en entire byte
- *****, used to match any number of bytes
- **(aa‖bb)**, used to match either expression **aa** or expression **bb**
- **{n}**, used to match exactly **n** bytes
- **{n-}**, used to match at least **n** bytes
- **{n-m}**, used to match between **n** and **m** bytes
- **{-m}**, used to match at most **m** bytes

3.2.2 The Storage Methodology

Miretskiy et al. in [MDWZ04] have proposed an on-access antivirus file system, describing also the ClamAV storage format. ClamAV stores its virus signatures in memory using a modified Aho-Corasick [AC75] tree (Figure 3.5), which actually depends on the length of the smallest signature.

Since the alphabet size for the ClamAV signatures is the size of the entire ASCII set, ClamAV needs to hold 256 pointers at each node in the tree to all possible matches. Each node occupies therefore (assuming 32-bit pointers) at least 4 bytes

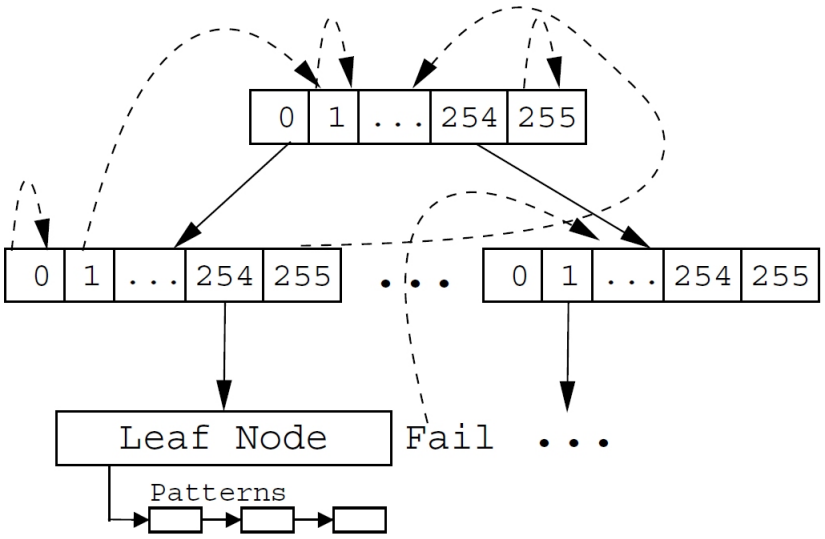


Figure 3.5: The ClamAV modified Aho-Corasick tree [MDWZ04]. Dashed lines show failure transitions.

per pointer × 256 pointers = 1,024 bytes (in their paper at [MDWZ04], the authors state that it occupied 1,049 bytes). There is a direct dependence of the size of memory required to store such a large automaton and the number of nodes in the automaton, as well as of the depth of the tree. The Aho-Corasick automaton builds a tree of depth equal to the length of the longest pattern in the input set, making such a storage solution very demanding in terms of space. Furthermore, ClamAV limits the tree in depth to the length of the smallest pattern, which is currently only 2 bytes - therefore, the tree's depth is just 2. From this point onwards, all patterns are being stored as linked lists. The primary performance bottleneck here appears whenever a large number of patterns share the same prefix.

ClamAV's regular expression engine has been improved over time [Lin06] and it now uses two matching engines: the first is using the Aho-Corasick algorithm described in Section 2.1.2.1, and the second is using the Wu-Manber approach described in Section 2.1.2.3. The slower Aho-Corasick is only used when scanning for regular expressions containing any type of wildcards, while Wu-Manber is used for the remaining patterns.

3.3 CPU and GPU-Accelerated Approaches to Virus Scanning

Virus scanning implies determining whether a known virus signature resides in an executable file or not, which is in fact an application of the pattern matching problem discussed in Section 2.1. Given the high number of viruses and families of viruses in

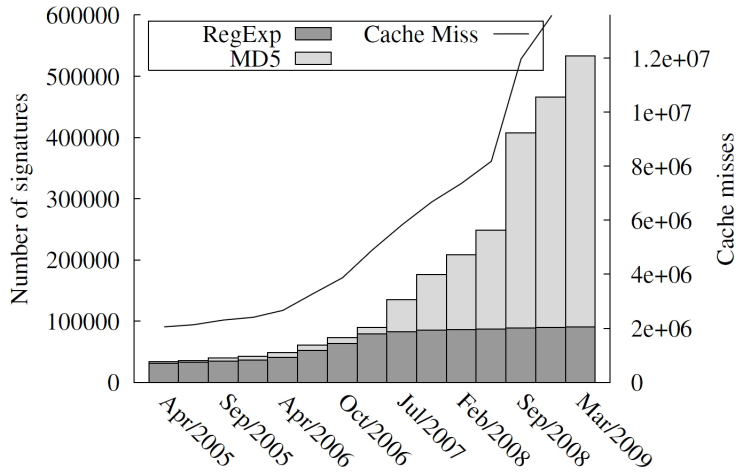


Figure 3.6: The ClamAV signature distribution over time, according to Cha et al. [CMJ+10].

the spread, resulting in a very high number of signatures (at the time of this writing, the two **.cvd** files from ClamAV contained over 60,000 virus signatures alone, not counting other signature files that it uses), and that such algorithms have different run-time performances (outlined in Sections 2.1.1 and 2.1.2), the run-time performance can determine very slow operativity during work. Even the most powerful antivirus engines consume quite an amount of resources that could be spared if the CPU could be offloaded with such computational work, which is why the GPU-based virus scanning had been a successful and welcome research topic for the past few years.

This section represents an outline of the research performed by the author in [PN] and outlines some of the most common CPU/GPU-based architectures for achieving fast virus signature matching.

3.3.1 Split-Screen

Cha et al. in [CMJ+10] have proposed an efficient anti-malware architecture called Split-Screen that performs an additional screening step prior to the signature matching process which can eliminate, according to its authors, over 90% of the non-infected files, leaving only suspicious files to be scanned and therefore reducing the scanning time significantly. Their approach is based on bloom filters, a concept introduced by Bloom in [Blo70]. A Bloom filter is currently a very compact structure to store a dictionary, and can determine if a pattern belongs to a dictionary or not. In the construction stage, fragments of fixed length are extracted from each signature and inserted into the filter. Later on, during scanning, a sliding window of the same size as the fragments is parsing the input data and the content at every byte is tested against the filter. The architecture used by SplitScreen is presented in Figure 3.7.

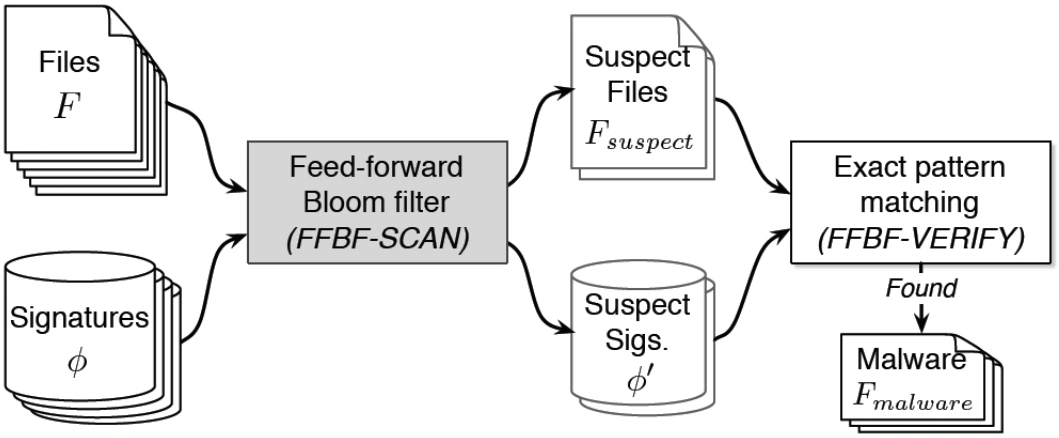


Figure 3.7: The SplitScreen architecture as proposed by Cha et al. in [CMJ+10].

One important observation from [CMJ+10] is that ClamAV has spent, in the authors' tests, over 95% of the time scanning and matching regular expressions, although these represent only a small fraction (about 16%) from the total number of signatures. The authors also propose a distributed architecture for their approach, showing that by allowing a small number of false positives in the detection process (which can be later on re-scanned using the classic algorithms implemented in ClamAV for greater accuracy), the throughput of ClamAV can be speed-up by 2 times, using twice less memory.

As outlined in [PN12], the majority of the signatures (more exactly, 96.6% from the set of a little over 62,000 used in [PN12]) use only the last four types of constraints discussed in Section 3.2.1. It is also possible to transform the remaining signatures internally into a format similar to the four types mentioned earlier, for instance *??* can be replaced with {*1-1*}, * with {*0,-*} and regular expressions of type *a(b||c)d* can be transformed into two separate expressions, *abd* and *acd*. In the testbed used in [PN12], from 62,302 signatures extracted from the ClamAV database, only 6,831 had contained regular expressions (a fraction of 11%, less than the 16% mentioned in [CMJ+10], which could show that there is a decreasing trend in adding signatures containing regular expressions in the ClamAV database over the past few years).

3.3.2 GPU Gems 3

One of the first studies on how GPU acceleration can be used to achieve fast parallel virus signature matching was presented by Seamans et al. in [SA07]. Their approach has a small resemblance (when it comes to the storage methodology) to the one proposed in [MDWZ04], which is discussed in Section 3.2.2, and is similar in terms of parsing the input data with the original PFAC algorithm discussed in Section 3.3.5. Here, the GPU maintains a list of virus signatures by first matching 2 bytes from

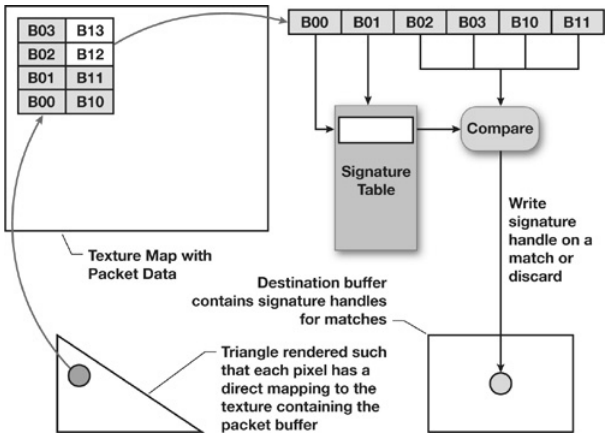


Figure 3.8: The algorithm for achieving virus signature matching as proposed by Seamans et al. in [SA07].

the signature itself and using them as an index into a 64,000-entry array, with each index being mapped at most to one signature. The next 4 bytes of the signature are stored in the array. Each thread running on the GPU will read the two bytes from the input string, which will be used as an index in the array, it will read the next 4 bytes preceding the initial two assuring that this way every consecutive 2 bytes are read and compared to the database (Figure 3.8).

The speedup achieved when using a GeForce 7800GTX GPu versus a Pentium 4 3GHz CPU averaged from 27× for the situation when no signatures match down to 11× as more matches are detected. While there are no discussions on the memory usage of the algorithm, the authors point out the lack of scalability of their approach (the algorithm supported at most 64,000 signatures, the size of the signature array in the GPU; furthermore, each signature had to have a unique sequence of 6 characters in it, including the two characters used as an index). A potential solution to the scalability problem is given with the mapping of the GPU array entries to multiple signatures, at the expense of additional processing at run-time.

3.3.3 GrAVity

Vasiliadis et al. in [VI10] have proposed a GPU-based architecture for virus scanning called GrAVity, that benefits fully from GPU acceleration and operates some modifications on the Aho-Corasick tree commonly used when performing scanning. Their architecture is shown in Figure 3.9 and bases its grounds on an idea similar to that proposed in the AVFS antivirus file-system in [MDWZ04]: instead of storing the entire pattern in the automaton, only patterns with common prefixes are stored (the length of the prefix varies in their tests) and leaf nodes store the remaining of the patterns.

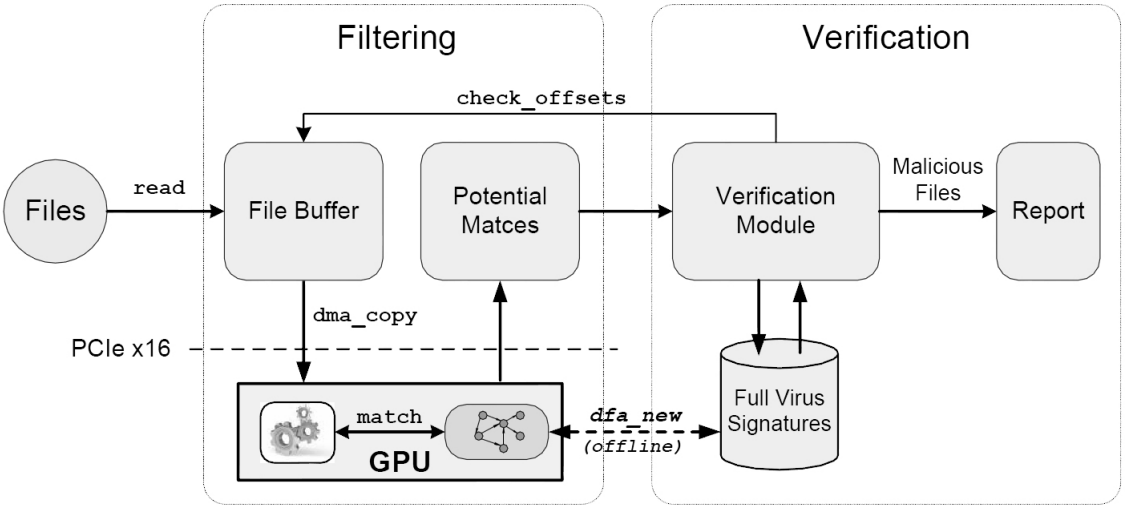


Figure 3.9: The GrAVity architecture as proposed by Vasiliadis et al. in [VI10].

Using this approach, certain false positives may appear in the matching process of the automaton, since the partial signatures may match certain files which are not threats or malware. In the experiments they performed, authors of GrAVity have observed that using a prefix of length 8 for the patterns generates less than 0.0001% false positives in a set of ordinary files in a computer system.

One downside to the GrAVity approach is the high memory storage required for the automaton, since each node uses 1KB in their implementation, leading to almost 400 MB of memory required for storing the automaton comprised of 400,000 nodes (obtained for a prefix length of 14). Their experiments achieved a throughput in ClamAV that was 100 times higher on the GTX 295 used versus a Xeon E5520 CPU, and up to 550 times higher when using cached files.

3.3.4 MIDeA

MIDeA is a network intrusion detection architecture using GPU acceleration proposed by Vasiliadis et al in [VPI11] that can benefit from network card, CPU and GPU parallelization, for increased scalability and performance. In their approach, there are no components needing to be serialized, meaning no component waits for another to execute or finish executing. The overall architecture is presented in Figure 3.10.

The set of patterns used in MIDeA come from Snort [sno], a popular open-source network intrusion detection system. MIDeA is using a memory representation based on compact states tables. With such tables, each state only contains the elements between the lower and the upper limits. The authors call this method the *AC-Compact* algorithm, while *AC-Full* is used to describe the classic Aho-Corasick algorithm. In their case, the transition function pointing to the next state is represented by *T[state,*

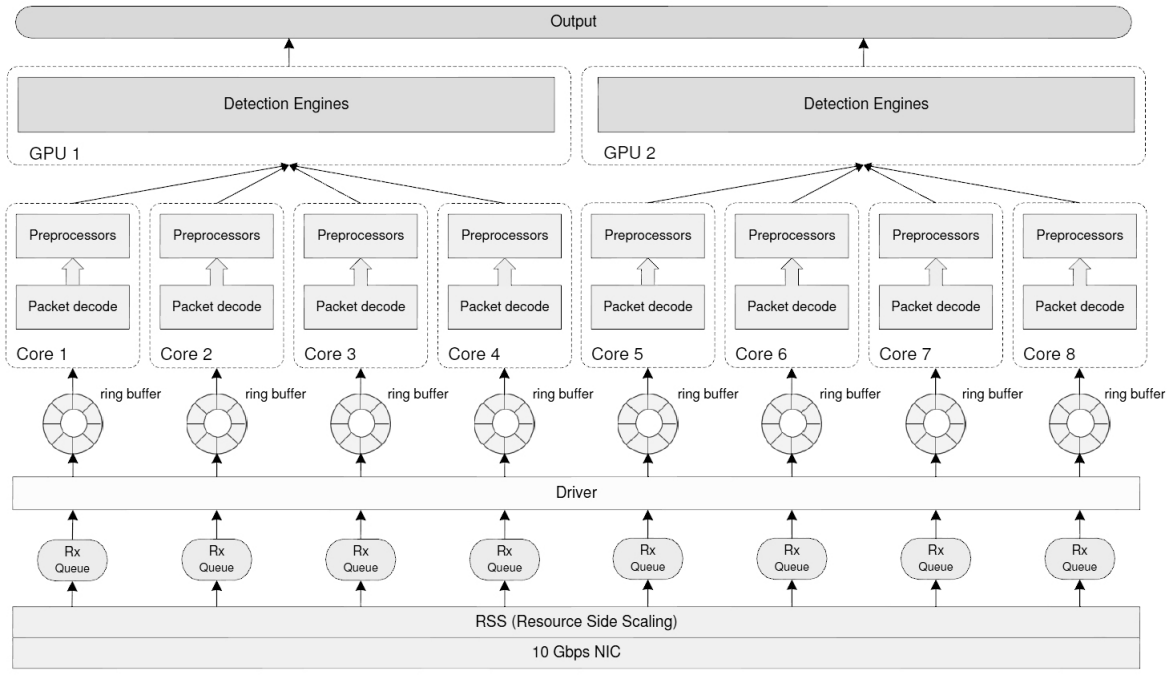


Figure 3.10: The MIDeA architecture as proposed by Vasiliadis et al. in [VPI11].

ch], where *state* is the current state and *ch* is the input character. The next state is therefore computed using the model shown in Figure 3.11, which reduces the throughput of the algorithm at the benefit of lower memory consumption (*AC-Compact* has about 77.7% of the throughput of the *AC-Full* approach in their experiments).

According to their experiments, for 8,192 rules and over 193,000 patterns, totalling a little over 1,7 million states in the automaton, the *AC-Full* algorithm used 890.46 MB of memory, while *AC-Compact* used only 24.18 MB. It is worth noticing however that if the *bandwidth* of each row in the table (the distance between the first non-zero value and the last non-zero value in the table) is sufficiently high, with a high dispersion of the elements throughout it, the memory consumption would be significantly higher, in the worst-case reaching that of the *AC-Full* algorithm, in which case it would also behave slower.

The original author of the *banded-row format storage* used in MIDeA is Marc Norton, a member of the Snort team, who in [Nor04] described the algorithm. Later on, Lee at al. in [LH08] have suggested an improved format for banded-row storage. First, they improved the original format by removing un-necessary failure conditions existing in the state table. For instance, assuming the tree in Figure 3.12 and the alphabet $\{a, b, c, d, e, f, g, h\}$, the transition vector for state 8 would be (11, _, _, 9, _, _, _, _) (where _ represents a failure), but this could also be represented as (11, _, _, 9) since all the following state transitions are failures. However, given that b and c fall in the bandwidth, they are both replaced with the transition from the failure

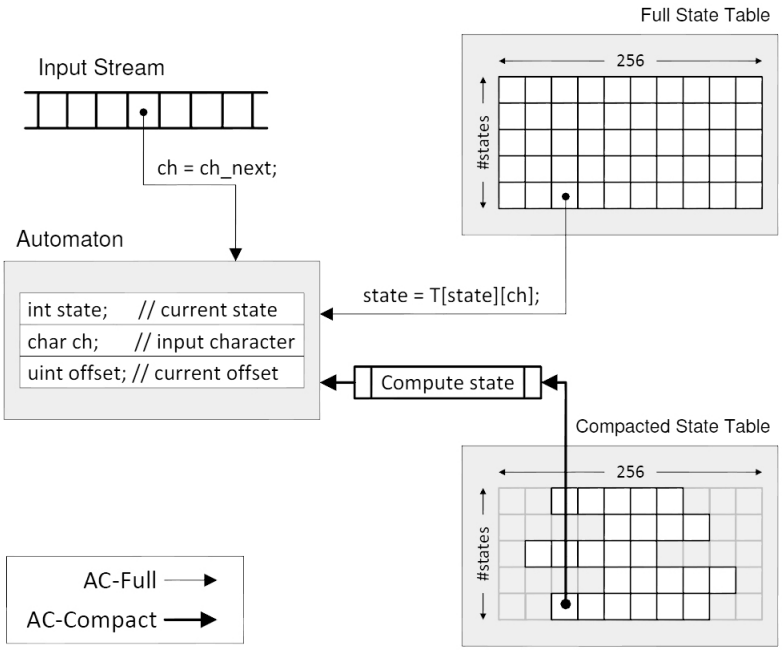


Figure 3.11: The MIDeA storage format used by Vasiliadis et al. in [VPI11].

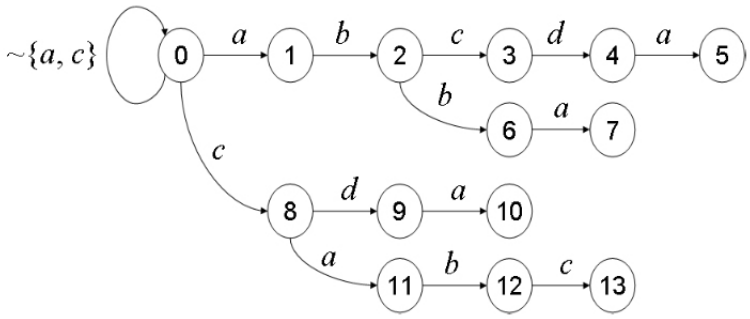


Figure 3.12: An example tree proposed by Leet et al. in [LH08].

function obtained by applying the character (e.g. for c, state 8's failure points to the root, so to state 0, and when applying c to state 0, as what would happen normally in case of a failure at state 8, we would once more reach state 8; applying the same logic to state d shows that the failure for this particular character points to state 0 again, so the new, compressed vector would become (4 0 11 0 8 9), where the first element, 4, represents the bandwidth length and the second element, 0, represents the starting index for this state).

The improved approach that Lee at al. are presenting in [LH08] is continuing the previous idea of the improved banded row format and focuses on compressing consecutive single-childed states in the automaton which are not final, reducing therefore storage requirements even further. The data structures required for this operation are significant and the pre-processing time for the automaton increases due to the com-

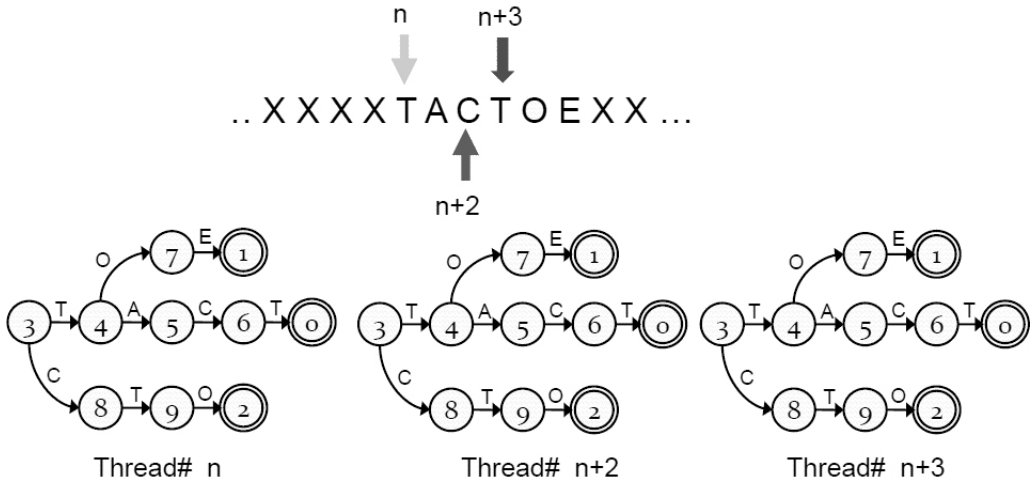


Figure 3.13: The Parallel Failureless Aho-Corasick algorithm proposed by Lin et al. in [LTL+10].

plexity of their construction: they use vectors for branched states, for single-childed states (storing the pattern number, the position in the pattern and the distance to the end of the pattern for each state), a failure function for all states and an output function for final states. Nevertheless, their approach is using much less memory than that of the classic Aho-Corasick algorithm, with the memory requirements being under 2% of the original algorithm in their experiments, and has about 40% better efficiency than the improved banded row format presented earlier. While ClamAV was still using less memory given the tree of depth 2 used, it lacks the performance, since whenever a match occurs, all patterns get checked sequentially for further accurate matches, an operation which can be very time-consuming.

3.3.5 PFAC

Lin et al. have presented in [Liu12] a library called PFAC for performing GPU-based string matching, which works in a similar way (in terms of input data parsing) to the approach presented in Section 3.3.2. PFAC comes from *Parallel Failureless Aho-Corasick* and was first introduced by Lin et al. in [LTL+10]. Its basic idea is to remove the failure function from the Aho-Corasick state machine and allocate each byte, at each position in the input string, to an individual thread in the GPU and traverse the state machine beginning with that location. Whenever a full match is found, each thread will know the exact position when a match had occurred. If at a given state during the traversal, no match is found, the thread terminates reporting a mismatch at that position. An example of how the input string is parsed and how thread allocation works is shown in Figure 3.13.

The time complexity of the initial PFAC algorithm is $O(N \times M)$, with a space complexity of $O(256 \times N)$ (N is the number of nodes in the automaton, while M is the

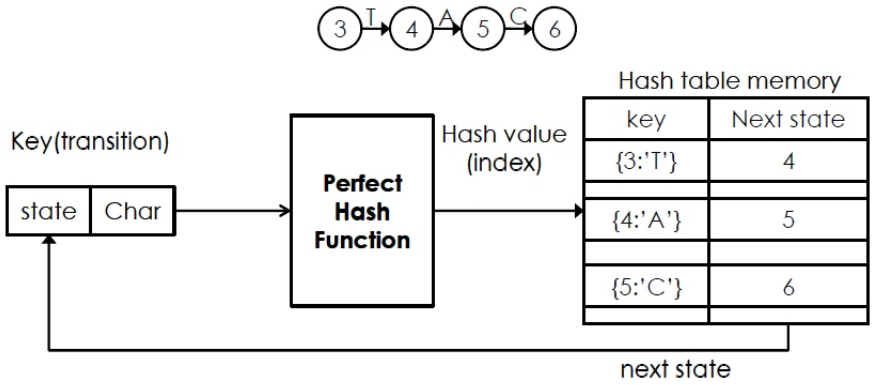


Figure 3.14: The Parallel Failureless Aho-Corasick perfect hashing algorithm proposed by Lin et al. in [Liu12].

longest length from all patterns). Some possible optimizations are suggested, such as using shared memory for reducing uncoalesced reads, or caching the state transition table into texture memory. In their experiments, the system throughput had increased when using PFAC, however one of its main drawbacks in real-world implementations is the lack of regular expression support in its native form. Additional synchronization routines and race conditions would have to be implemented in order to implement regular expression support, even if using a naive implementation such as trimming all regular expressions into their non-regex components and searching for each individually, with a final step verifying all partial matches for exact full matches.

In order to efficiently store the automaton in GPU (device) memory, the authors use a perfect hashing architecture (Figure 3.14) which allowed them to store only valid transitions of the automaton in a hashing table. They used the *Slide-Left-then-Right First-Fit (SRFF)* algorithm for creating a two-dimensional array of width w where each key k gets placed at location *(row, column)*=($\frac{k}{w}$, $k \mod w$), prioritizing rows by the number of keys in them so that, according to their priority: a) slide each row to the left so that the first key is aligned at the first column and b) slide each row to the right until each column only has a single key, and store that offset in an array *RT*.

The architecture of PFAC as proposed in [Liu12] may be found in Figure 3.16. To improve the perfect-hashing procedure, in the same paper the authors propose a *two-level perfect hashing* approach, where at level 2 the same hashing function is applied but with different parameters. The computational drawbacks here were related to the mod operation (which is slow on GPUs) and also to the fact that the initial perfect hashing approach does not consider the tree structure of the automaton. To solve the problem, given that each state has only a few valid transitions, they also propose a *modulo-free perfect hashing* approach and prove a theorem discussing the space storage requirements of their approach, stating that for the prime p=257 applied to

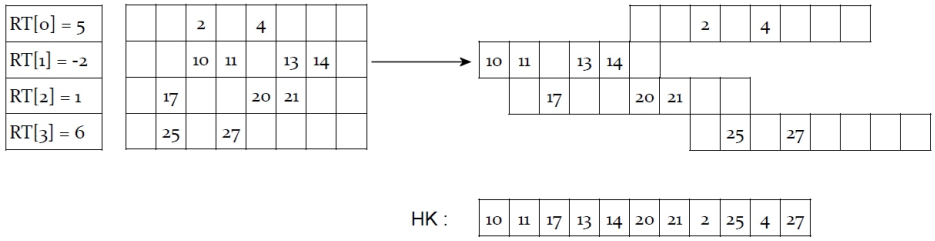


Figure 3.15: An example for the perfect hashing algorithm proposed by Lin et al. in [Liu12].

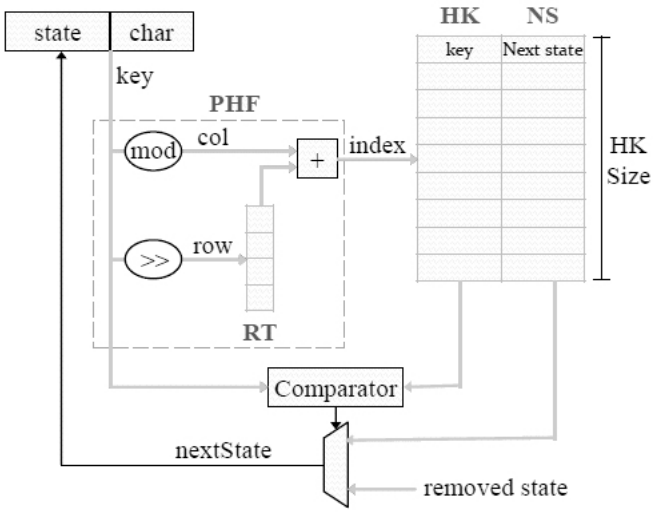


Figure 3.16: The architecture for the perfect hashing algorithm proposed by Lin et al. in [Liu12].

their hash function:

$$S = \sum_{i=0}^{N} s_i \leq min(21.4, 1 + 71 \times \frac{L-1}{N-1}) \times R, \tag{3.1}$$

where S is the total storage required, s_i is the number of memory locations for storing valid transitions of state i, N is the number of states and R the number of transitions.

The run-time performance of the improved PFAC using perfect hashing was about the same as that of the initial PFAC algorithm, but using about 4 times less memory than that used by an optimal implementation of PFAC (620 KB for 10,076 rules obtained from Snort [sno], totalling 126,776 states).

3.4 A Highly-Efficient Storage Model for Pattern Matching Automata

In [PN12] we have proposed a highly efficient memory storage architecture for pattern-matching automata, supporting both the Aho-Corasick and Commentz-Walter approaches. We are going to present the model as follows, outlining its primary benefits and comparing it to other known approaches up-to-date.

3.4.1 Motivation

The motivation for using exact signature matching when performing virus scanning lies in the history of malware and antivirus evolution over time: it is still feasible to detect and accurately match exact virus signatures, rather than going through more complicated steps (such as emulation or sandboxing, as discussed in Section 2.2.2) which may produce false positives. Furthermore, it is still the only accurate way of detecting most viral infections, excepting metamorphic viruses which are completely resilient to signature matching and where different heuristics need to be employed (in Chapter 5 we are discussing a series of such heuristics). The downside of this approach, compared to behavioral analysis, is that exact signature matching is much slower than employing heuristics. As a consequence, GPU-based virus scanners have been long scheduled to appear by both Kaspersky [Kas] and nVIDIA [nVI].

The operating system support for GPU-enabled processing had increased over the past few years. Microsoft had introduced the Windows Driver Model (WDM) in [wdm], which specifies that if the device is non-responsive by default for more than 3 seconds, as a result of a driver bug or other type of problem, the driver will reset itself. This has direct implications over how CUDA-enabled processing is being handled, since blocking the GPU with computations for more than 3 seconds would render the device driver to reset itself and re-initialize, causing all computations to be forcefully terminated and therefore the results completely lost. Currently, when performing development on CUDA-enabled devices, this feature can be disabled, however for end-users this is not an option as it increases the stability of the system. Furthermore, real-time architectures for virus scanning which work (or will work) with GPU acceleration will need to consider this limitation implicitly and implement a *watchdog* mechanism that would cease control of the GPU to the operating system at a time interval of less than 3 seconds, so that it can later on resume the processing and finish the tasks it was assigned.

3.4.2 Towards a Hybrid CPU/GPU Efficient Storage Architecture

The computational processing power of a GPU can be easily adapted to a large variety of algorithms, in many different fields of research. The computations on the GPU however can only be performed using the GPU's memory. In this case, data from the host (CPU) memory must be transferred to device (GPU) memory upfront, and the transfer needs to be completed before the computations begin, to ensure that the results are accurate and correct. Therefore, the basic operations that need to be performed in most hybrid architectures are the following:

1. (Host-side) Prepare the data to be processed by the device in host memory
2. (Host-side) Transfer the data from host memory to device memory
3. (Host-side) Initiate device computations
4. (Device-side) Perform computations using the transferred data
5. (Host-side) Wait for the results to be available in device memory
6. (Host-side) Transfer results from device memory to host memory
7. (Host-side) Interpret results and provide output to user

While the computational steps themselves have variable run-times, the memory transfers are usually the bottleneck of such architectures. In fact, in several architectures and models existing today, such memory transfers are not given the proper attention or focus. Let's exemplify by referring to the memory allocation process, both in the host and device memory: whenever employing *malloc()* type of calls in the CPU or GPU memory, the amount of memory requested is allocated and a pointer address is returned. However, repeating the memory allocation procedure a few times will generate different pointer addresses, at diverse (apparently sparse) memory locations (Figure 3.17) - this is caused by the internal memory management routines, which allocate memory so that future memory allocations or re-allocations can be successfully accomplished in the free spaces left behind, or memory areas expanded as needed by the program. While this has immediate benefits when using with memory-prone algorithms, which need to constantly resize their memory areas dynamically, it has severe drawbacks when using large data structures which require a large amount of memory, since those memory requirements can sometimes be a few times bigger than the actual memory being used by the program.

Another problem related to memory allocation is step 2 above: transferring memory from host memory to device memory would require going through each memory location from host memory, reading it, allocating memory in the device and copying

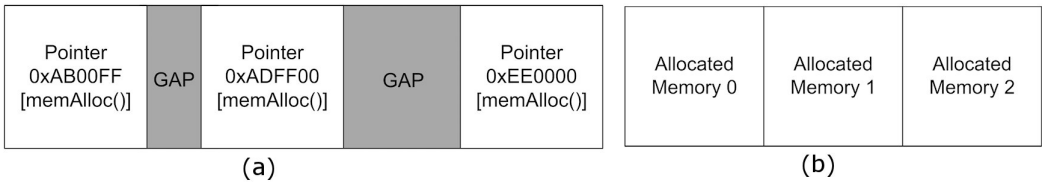


Figure 3.17: a) Sparse, consecutive memory allocations and gaps that appear in GPU memory; b) A single, sequentially allocated memory block in the GPU device.

it there. For a pattern matching automaton with a very high number of nodes (e.g. in the order of millions or more), this would require many consecutive memory allocation calls in device memory, which would again produce different pointers at sparse memory locations, as pointed out in Figure 3.17. Given the low amount of memory resources of a GPU compared to a CPU (a high-end GPU is still running memory in the order of only a few GBs, while consumer CPUs are commonly working with more than 8 GBs and up to 32 GBs on a daily basis), this would result in immediately filling the memory with unnecessary data, leaving little room for additional memory allocations and eventually causing the program to forcefully terminate on such an event.

The PCI-Express standard [pci] ensures top transfer speeds of 8 gigatransfers per second (8 GT/s), which offers sufficient bandwidth performance for transferring large chunks of data from device to host memory. When dealing with pattern matching automata however, developers aim for low memory requirements (low-size requirements for each node structure of the automaton) given the high number of nodes, and the bottleneck becomes obvious when attempting to transfer millions of nodes (of only a few bytes or kilo-bytes in size) as in step 2 above. In one of our simple tests performed on a number of 350,000 nodes only, each occupying a little over 40 bytes, copying all the nodes from the host to the device took almost 3 hours.

The model we are presenting does not suffer from these drawbacks: instead, it is able to represent the entire automaton as a consecutive memory area, using a series of compression and processing techniques, which allow it to be transferred all at once from the CPU to GPU memory, fully benefitting from the high-throughput offered by the PCI-Express interface in contemporary computers. Furthermore, the storage space required for it is always constant in S, the total number of states (nodes) in the automaton.

3.4.3 The Storage Model

In order to implement our model, we used a testing bed of 62,302 virus signatures from the March, 2012 version of the ClamAV [cla12] database. In order to achieve a sequential representation of our pattern matching automaton, we have used a stack-based layout, replacing the pointers inside the code (usually pointing to the widely

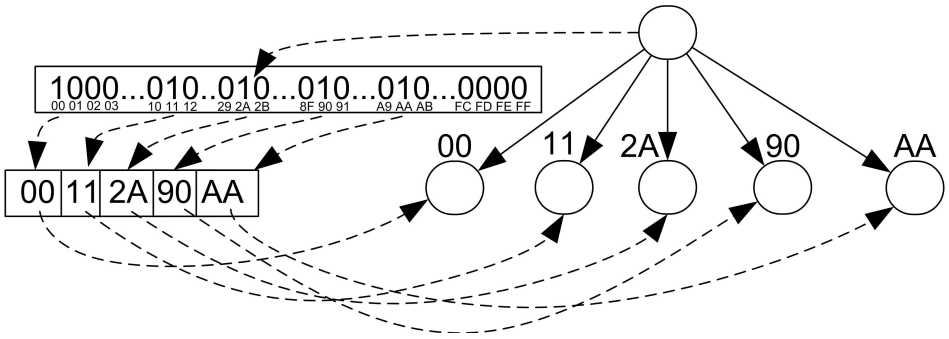


Figure 3.18: The bitmapped node implementation of a tree-based structure.

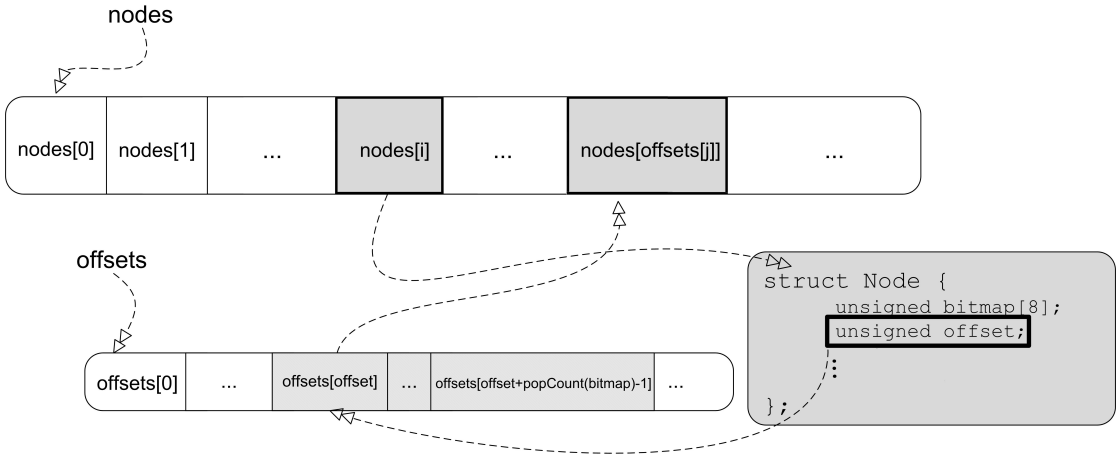


Figure 3.19: The stack-based memory model (with the offsets vector) used for efficient storage of the pattern-matching automaton in the GPU.

spread memory locations) with offsets in the stack. There are two variants we have proposed: the first uses a separate vector of offsets (Figure 3.19), while the second discards the vector entirely. Our approach represents a serialized representation of the tree, obtained by *flattening* the tree so that all children of a node are kept in consecutive offsets. By ensure that all children of a node are stored in a consecutive area, we are ensuring that we only need the starting offset of the first child in the parent node.

In order to reduce the size of the node structure, we used a bitmapped representation as opposed to the classic state table approach. In a bitmap representation (Figure 3.18), $|\ \Sigma\ |$ bits are used for representing possible transitions for each characters in the Σ alphabet. Additionally, a pointer to a linked list of locations is stored in the node, and the linked list only stores the valid transitions, in lexicographical order. In order to determine the location of child at position i in the alphabet, we compute the *popcount* [Knu09] for the first i positions in the bitmap, which will give us the index in the linked list for finding the desired location. The *popcount* value represents the number of 1s in a binary string.

The algorithm for the construction of the stack (along with the *offsets* array) from the host's memory representation is below:

- *Initialization*
  1: $topOfNodeStack \leftarrow 1$ (top of node stack)
  2: $currentPosition \leftarrow 0$ (position in the stack)
  3: $topOfOffsetStack \leftarrow 0$ (top of the stack of offsets)
  4: *node* (the currently processed node)

- *function addNode(node, currentPosition)*
  1: $nodeStack[currentPosition] \leftarrow node$
  2: $nodeStack[currentPosition].offset \leftarrow topOfOffsetStack$
  3: add to hash ($key \leftarrow node$, $value \leftarrow currentPosition$)
  4: $pc \leftarrow popCount(node.bitmap)$
  5: **for** $i \leftarrow 0$ to $pc - 1$ **do**
  6: $\quad offsetsStack[topOfOffsetStack] \leftarrow topOfNodeStack + i$
  7: $\quad topOfOffsetStack \leftarrow topOfOffsetStack + 1$
  8: **end for**
  9: $old \leftarrow topOfNodeStack$
  10: $topOfNodeStack \leftarrow topOfNodeStack + pc$
  11: **for** $i \leftarrow 0$ to $pc - 1$ **do**
  12: $\quad$ addNode($node.child[i]$, $old + i$)
  13: **end for**

The *offsets* array can however be removed and the offset pointer can be stored inside the node structure itself (Figure 3.20). As a consequence, the child at index i in a node n is accessible through the reference *nodes[offsets[n->offset+i]]*. The algorithm for performing a construction using this approach is presented below:

- *Initialization*
  1: $topOfNodeStack \leftarrow 1$ (top of node stack)
  2: $currentPosition \leftarrow 0$ (position in the stack)
  3: *node* (the currently processed node)

- *function addNode(node, currentPosition)*
  1: $nodeStack[currentPosition] \leftarrow node$
  2: $nodeStack[currentPosition].offset \leftarrow topOfNodeStack$
  3: add in hash ($key \leftarrow node$, $value \leftarrow currentPosition$)
  4: $pc \leftarrow popCount(node.bitmap)$
  5: $old \leftarrow currentPosition$
  6: $topOfNodeStack \leftarrow topOfNodeStack + pc$

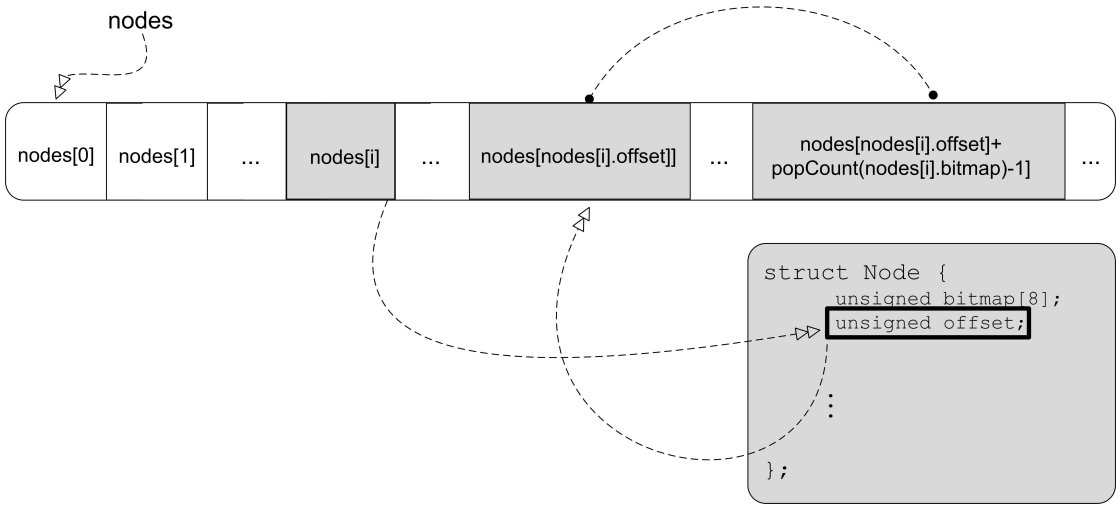


Figure 3.20: The stack-based memory model (with the offsets vector) used for efficient storage of the pattern-matching automaton in the GPU.

7: **for** $i \leftarrow 0$ to $pc - 1$ **do**
8: addNode($node.child[i]$, $old + i$)
9: **end for**

If the node structures contain other pointers to different nodes (e.g. for instance, for the *failure* function in the Aho-Corasick automaton), the approach can be improved using a hash table that stores a list of all the nodes in the automaton, together with their corresponding offset in the serialized stack representation. A second preprocessing step would parse the GPU tree in pre-order (although any other type of parsing can be used), and would replace the pointers in the stack with the offsets, by locating them in the hash.

The storage requirements for the Aho-Corasick automaton using an alphabet Σ of size $\mid \Sigma \mid$ with a total of S states (nodes), and using bitmapped nodes, would be $S \times (2log_2S + \mid \Sigma \mid)$ bits. The calculus was performed keeping in mind that for each node, a failure offset occupying log_2S bits needs to indicate the new state in case of a mismatch; the children offset pointer would also require log_2S bits and would be used to determine the starting offset of the current node's children (it's important to notice here that it is not necessary to have the children of a node immediately follow the node at consecutive memory locations in the representation); finally, the bitmapped representation of the alphabet would be occupying $\mid \Sigma \mid$ bits and would be the one used when performing popcount computations. Compared to the PFAC approach in [Liu12], where the *AC-Compact* representation used by the authors occupied 24.18 MB for a total of 1,703,023 states (with about 15 bytes per state, assuming - since in the paper it is not specified - an alphabet size of $\mid \Sigma \mid = 32$), our approach would require a storage of just $2 \times 21 + 32 = 74$ bits per node, or 10 bytes per node, resulting

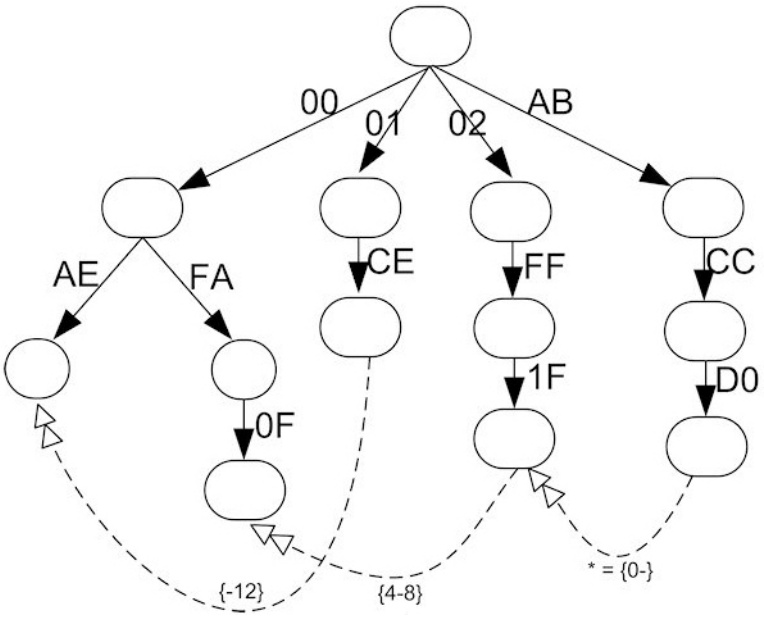


Figure 3.21: Our constraint-based Aho-Corasick automaton for the set of patterns {00FA0F{4-8}*02FF1F*ABCCD0, 00AE{-12}01CE} [PN12]. Dashed lines represent links between leaves and failures have been omitted.

in a total of 15.02 MB of memory consumed - which is 1.6 times less memory compared to the PFAC storage methodology. Comparing our methodology with the one from GrAVity in Section 3.3.3, for 352,921 nodes, our storage model requires 18.63 MB of memory, while GrAVity with the inefficient storage of 1 KB per node, would use about 345 MB, almost 19 times more memory.

3.4.4 The Constraint-Based Aho-Corasick Algorithm

In order to accommodate for the different types of regular expressions which the patterns from the ClamAV database contain, as discussed in Section 3.2.1, we have extended the Aho-Corasick automaton to support the last four most common types of constraints, which represent more than 96.5% from the entire ClamAV database [Pun13].

Each regular expression in the ClamAV signature database is being trimmed into its subexpression parts (none of the subexpressions will contain any regular expressions, they will be plain patterns only). Our algorithm adds pointers from the leaf of a subexpression at position i of such type to leaf of the previous subexpression at position *i-1* in the pattern. We also store, for each leaf, the position of partial matches for it in the input data. Whenever a leaf is encountered at position j, the leaf of the subexpression at position *j-1* is checked for a match; if a match is found, and if the constraint is verified, the current leaf is being marked as a match as well, and a

partial match is added for this leaf also. If a leaf is belonging to the last subexpression of a pattern, a full match has been found and is reported (Figure 3.21).

3.4.5 Towards a Parallel Implementation for the GPU

Applying pattern matching in the GPU relies on heavy parallelism, which requires changes or adaptations of the classic algorithms in order to be used in data-parallel, task-parallel or hybrid-parallel manners. For the Aho-Corasick and Commentz-Walter automata, one of the simplest parallelization method is a data-parallel decomposition of the input string, and its applicability to the algorithm itself.

One of the first attempts to parallelize the Aho-Corasick algorithm was proposed by Intel in 2007 [XW07], when Xu et al. had used Intel threading and profiling tools to implement a data-parallel version. They were able, as a consequence, to improve the speed of the algorithm by 1.714 times on a dual-core system.

Data-parallel decomposition refers to splitting the input data into several, smaller chunks, and scanning each of them separately in its own thread. However, for pattern matching algorithms this could pose problems when scanning at the border of two different chunks of data, because the match might be split between those chunks (Figure 3.22).

In order to solve the situation, several techniques could be employed. One of the simplest would be extending each chunk of data with *D-1* bytes, where *D* is the length of the longest pattern in the dictionary, as discussed in our approach from [Pun09] and by Zha et al. in [Zha]. This approach may produce duplicate results, especially if there are patterns of smaller length which exist in the first *D-1* bytes of the next chunk: for instance in the case of *acad* from Figure 3.22, if the pattern *ad* also exists in the dictionary, it would be found by both thread i and also by thread $i+1$, so it would be reported twice. A possible solution here would be to use a hash table to store match positions. Another approach to avoid border-line issues with data-parallel pattern matching would be to store, for each node, its depth d, and when reaching the end of the chunk to extend the search by D-d characters, however this approach is also prone to duplicate results and would also increase the storage requirements for the automata, by a significant amount when dealing with large structures. Throughout the implementations in this thesis, we have always followed the first one presented above, to avoid introducing additional memory overhead.

3.4.6 Performance Evaluation

We have implemented the Aho-Corasick, Commentz-Walter and an extended version of the Aho-Corasick algorithm (presented in Section 3.4.4). The constraint-based automaton uses an additional array, called *links*, which is used for indicating the leaf of a previous subexpression in the pattern. Construction of the automaton is not the

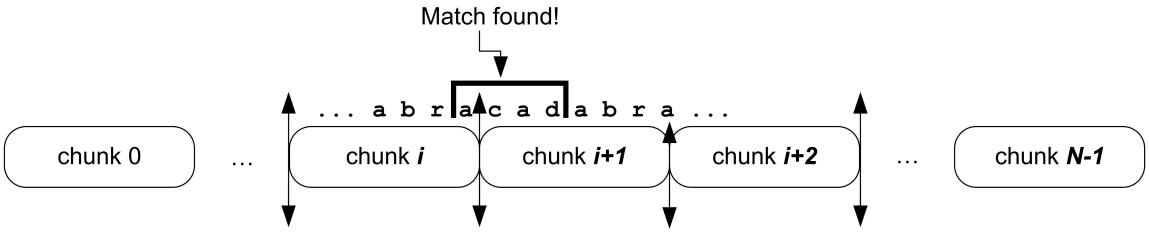


Figure 3.22: A situation where the match *acad* is split over two different chunks of data being processed in parallel, with none of the chunks matching the string.

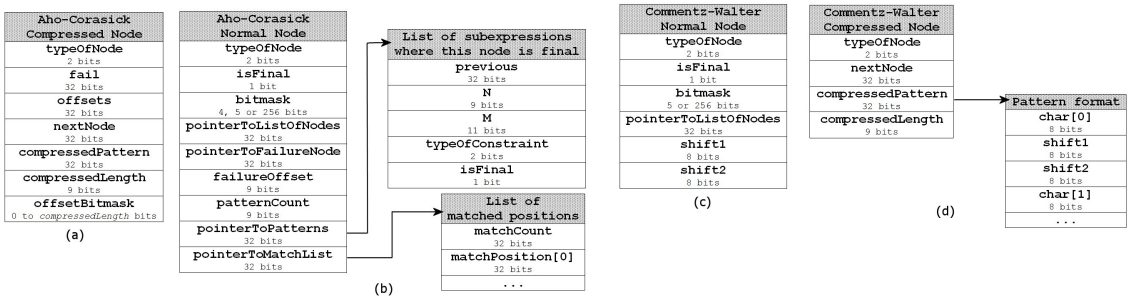


Figure 3.23: a) Aho-Corasick compressed nodes; b) Aho-Corasick normal nodes in the constraint-based automaton supporting RegEx signatures; c) Commentz-Walter normal nodes; d) Commentz-Walter compressed nodes.

subject of this performance evaluation, since the automaton is usually delivered in real-world environments as alread being pre-processed or constructed, however it has been thoroughly discussed in Chapter 4. Given that the automaton is now stored as a continuous block of memory, the transfer between the host memory and the device memory is instant.

The Aho-Corasick automaton has been subject to numerous improvements in terms of efficiency over the past years. Some of the early attempts belonged to Tuck et al. in [TSCV], which had presented a *path compression* approach and applied it to the Snort [sno] network intrusion detection system. The model had been further improved to work with bitmapped nodes by Zha et al. in [ZS08]. Later on, Liu et al. in [LYLT09] had proposed a table compression approach for the same automaton, by eliminating equivalent rows in the state table and grouping state rows into equivalent classes.

In our approach, we implement CPU-based version of the Aho-Corasick, Commentz-Walter and Wu-Manber algorithms, and GPU versions as well. For the CPU implementations we used path compression and bitmapped nodes (as shown in Figure 3.23), and for the GPU implementation we used the stack-based representation discussed in Section 3.4.3. A data parallel approach was employed for performing scanning of the input data.

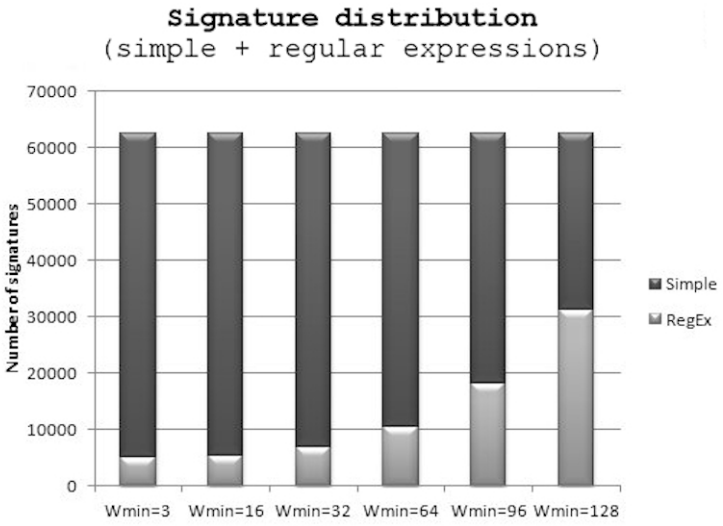


Figure 3.24: The signatures and their distribution when performing single-core and multi-core scanning on both the CPU and the GPU.

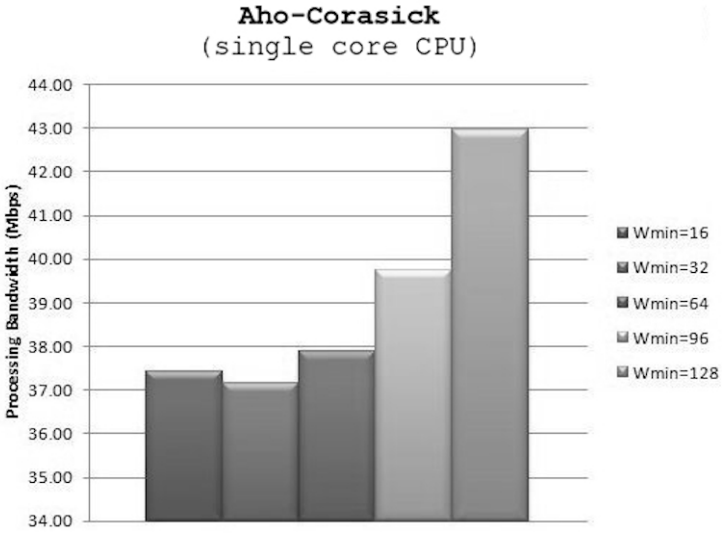


Figure 3.25: The throughput of the Aho-Corasick automaton when using a single-core CPU implementation.

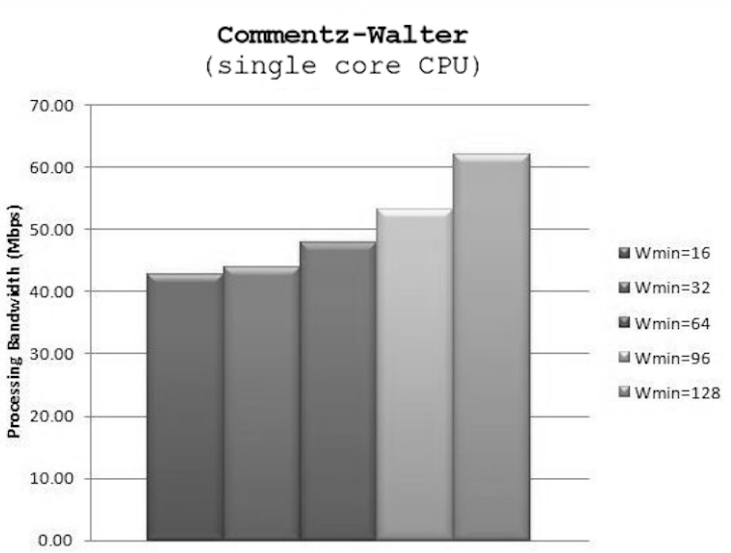


Figure 3.26: The throughput of the Commentz-Walter automaton when using a single-core CPU implementation.

The data structures used in our implementation from [PN12] for benchmarking purposes are presented in Figure 3.23.

Although it would be possible to implement path compression in GPU-based implementation, the memory relocation routines in device memory are much slower compared to host memory. Equivalent functions to *memmove()* or *realloc()* in host memory, when applied to device memory have a significant on performance. By avoiding path compression, coding complexity also greatly reduces, which eventually results in better performance and better error resilience.

Given that the Commentz-Walter algorithm heavily relies, as discussed in Section 2.1.2.2, on the length of the shortest keyword in the dictionary, w_{min}, we have trimmed the signatures chosen from the ClamAV database (62,302 in total) and delimited them according to the several different values for w_{min}, particularly $\{16, 32, 64, 96, 128\}$. Some of the signatures in the ClamAV database contain regular expressions, which is why we have counted them accordingly for reference, with the results being presented in Figure 3.24.

3.4.6.1 The CPU Implementation

We then measured the performance of the Aho-Corasick, Commentz-Walter and Wu-Manber algorithms based on the different values chosen for w_{min}, and the results are shown in Figures 3.25, 3.26 and 3.27, respectively. The results have shown the following:

- The Commentz-Walter algorithm has outperformed the Aho-Corasick approach from a lowest of more than 14% (for $w_{min} = 16$) to more than 44% (for $w_{min} =$

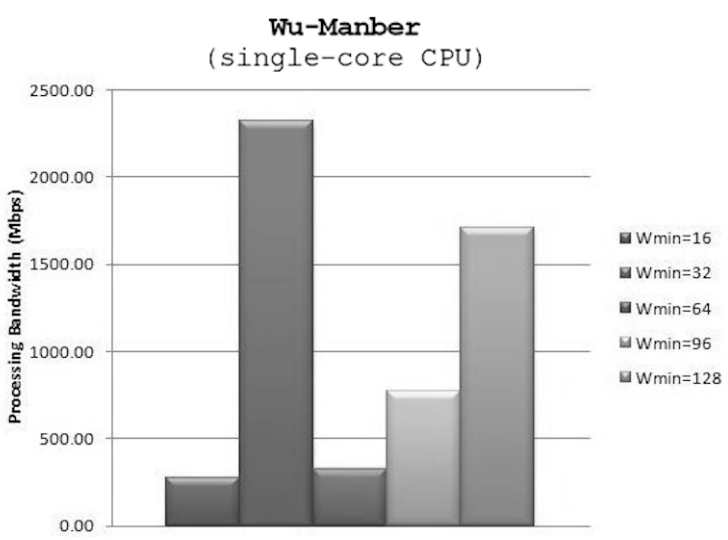


Figure 3.27: The throughput of the Wu-Manber automaton when using a single-core CPU implementation.

128), as expected given the higher w_{min} values which eventually translates in higher jumps in the input data.

- The Wu-Manber has outperformed the Commentz-Walter approach from a lowest of more than 6.8× the performance (for $w_{min} = 64$) to more than 53× (for $w_{min} = 32$). The highest throughput speeds-ups were obtained in the Wu-Manber implementation for $w_{min} = 32$ (53× faster) and $w_{min} = 128$ (27× faster).

Based on the observations made by Lin et al. in [LLL11], where an improved version of Wu-Manber was used, called Backward Hashing, for scanning simple signature patterns, and Aho-Corasick for scanning regular-expression-based ones, and given the observation made in GrAVity (Section 3.3.3) where the authors have used a limited tree depth of 8 (which achieved a very low number of false positives), we have built a multi-threaded scanner aiming to perform a similar operation, however also aiming to maximize throughput and minimize memory requirements when running on a GPU architecture. Such an approach would be the most feasible to implement in a real-time antivirus scanning engine, where low memory and high bandwidth are best desired.

Similar to Lin in [LLL11], in our approach we have combined, using our own storage model, the three most common types of multiple pattern matching algorithms, Aho-Corasick, Commentz-Walter and Wu-Manber into a hybrid-parallel, multi-threaded scanner performing regular expression scanning in one thread and regular virus pattern scanning in the other. We implemented three variants for performing the scan-

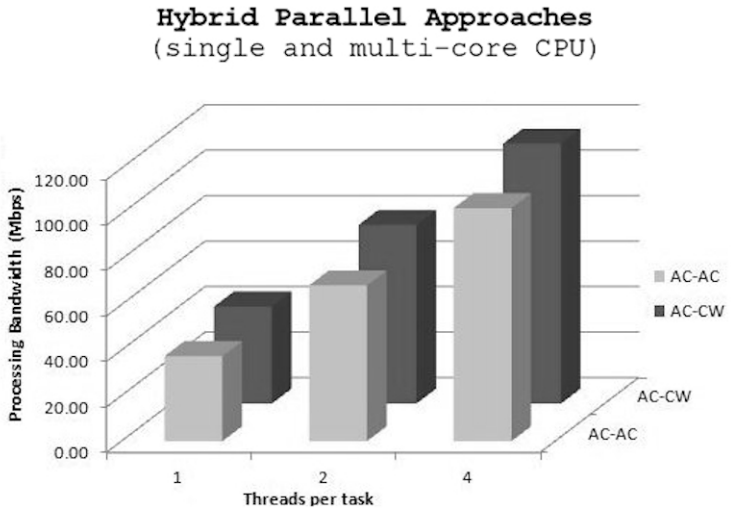


Figure 3.28: The throughput of the different hybrid parallel approaches tested using our storage model.

ning and measured the overall throughput of the system, using the following combinations:

- One thread scanning regular expressions using Aho-Corasick and the other scanning regular virus signatures using again Aho-Corasick (depicted as AC-AC).
- One thread scanning regular expressions using Aho-Corasick and the other scanning regular virus signatures using Commentz-Walter (depicted as AC-CW).
- One thread scanning regular expressions using Aho-Corasick and the other scanning regular virus signatures using Wu-Manber (depicted as AC-WM).
- We tried out combinations using 1, 2 and 4 threads for both scanners, using a data-parallel scanning approach for each thread (so for instance for 4 threads per scanner, we would have 4 threads performing data-parallel scanning regular expressions using Aho-Corasick, and another 4 threads performing data-parallel scanning of regular virus signatures using one of the other algorithms).

The results obtained for AC-WM and AC-CW were identical, which is no surprise given that the Wu-Manber bandwidth is much higher than that of Aho-Corasick as shown earlier, however although given that the regular signature scanning process finishes first, it must still wait for the completion of the regular expression scanning, which becomes the bottleneck of the system. We have therefore not included the AC-WM results in our graphical representation of the results shown in Figure 3.28, for better clarity.

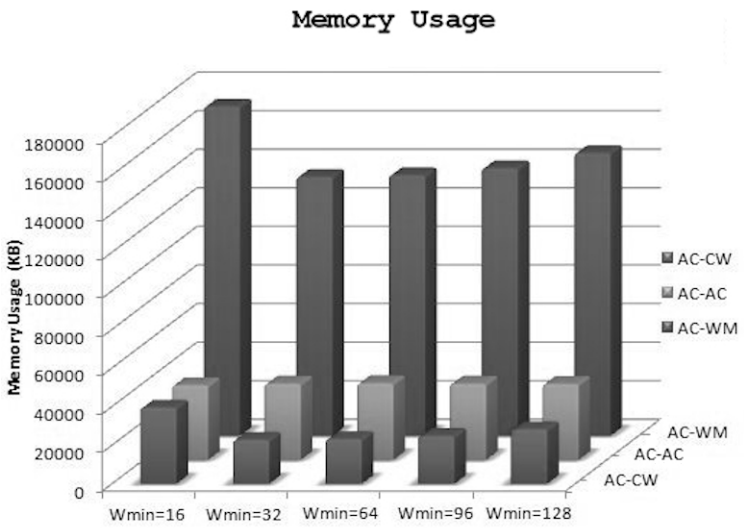


Figure 3.29: The memory usage measured for our hybrid parallel implementations.

Finally, we measured the memory storage requirements for all hybrid implementations and the results are present in Figure 3.29. The AC-AC approach had produced almost constant memory usage throughout the test, which is normal given that the same structures were used for the same number of nodes in the automaton. The AC-WM approach had the most memory usage, given the hash-table structures implicitly required by the Wu-Manber algorithm. For the Wu-Manber implementation specifically, since we used a block size of 3 (as recommended by the authors in [WM94] for a large number of patterns, to avoid collisions) and the result produced $2^{24} = 16.7$ million elements in the hash table. For storing the two elements that comprise the hash table described in Section 2.1.2.3, considering 32 or 64 bits per integer value (we used a 64-bit integer in our implementation), a total amount of 64 MB or 128 MB of memory is used, not counting any additional structures.

3.4.6.2 The GPU Implementation

We implemented the same algorithms as in the CPU implementation in Section 3.4.6.1 on our test GPU and started from the same premise as in GrAVity by limiting the depth of the tree. We tested our algorithms with three different depths, of 8, 12 and 16.

We started by implementing the classic Aho-Corasick algorithm for the three different depths, and one of the first concerns was memory usage and the number of states/nodes in the automaton. Figure 3.30 shows that there were about 350,000 nodes at a depth of 8, close to 600,000 nodes for a depth of 12 and over 800,000 nodes for a depth of 16. Their corresponding memory usage using our proposed methodology is shown in Figure 3.31.

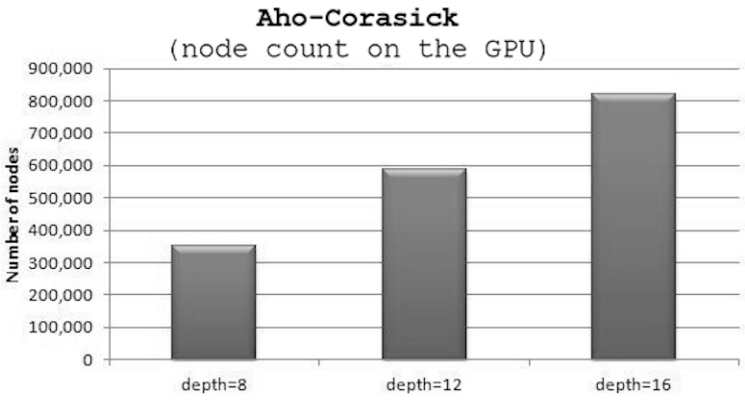


Figure 3.30: The state/node count for the depth-limited Aho-Corasick automaton.

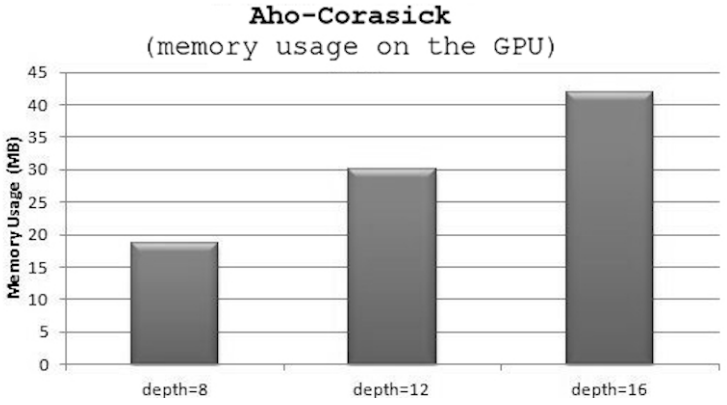


Figure 3.31: The memory usage for the depth-limited Aho-Corasick automaton.

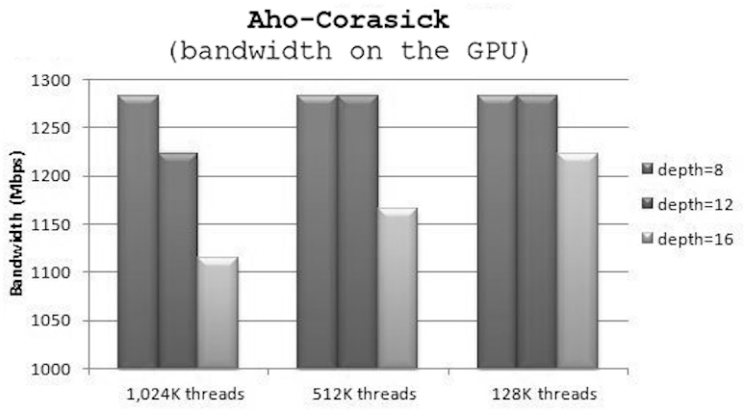


Figure 3.32: The bandwidth performance for the depth-limited Aho-Corasick automaton.

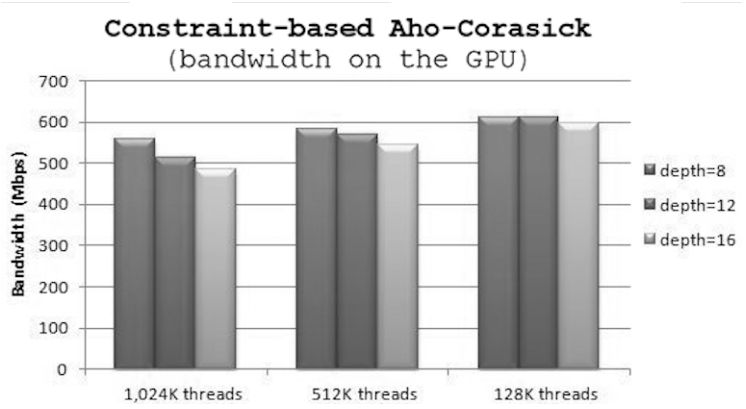


Figure 3.33: The bandwidth performance for the depth-limited Aho-Corasick automaton.

The bandwidth performance of the Aho-Corasick GPU implementation is shown in Figure 3.32. In the experiment performed, the GPU implementation had been about 38 times faster than our single-core CPU implementation of the same algorithm.

We also tested the bandwidth performance of our proposed extended Aho-Corasick algorithm (Figure 3.33) and the throughput was about half of the original GPU implementation of the automaton.

Given that in the previous attempt the best memory usage pattern was obtained when implementing a hybrid-parallel combination of the Aho-Corasick and Commentz-Walter algorithms (AC-CW), for the GPU implementation we focused on the same ones, however in our situation, with the trees being depth-limited (we imposed a permanent limit of 8 to the depth of both trees this time), we recomputed the total number of nodes in the automata and the memory usage of both combined. Results are shown in Figures 3.34 and 3.35. In the hybrid implementation, one kernel (kernel 1) in the GPU was running the regular expression scanner, and the

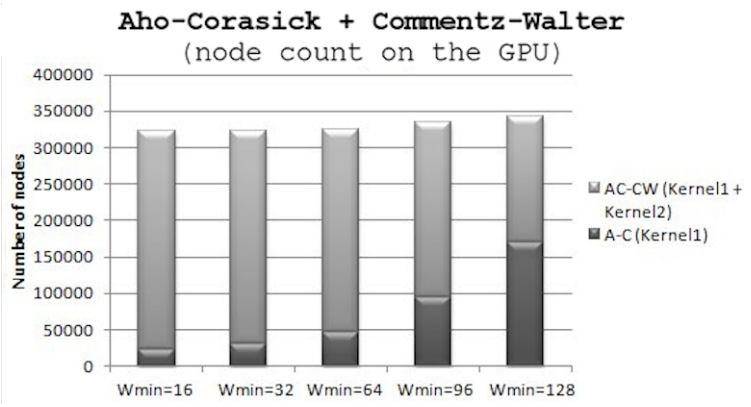


Figure 3.34: The total number of nodes for the both types of the automata used.

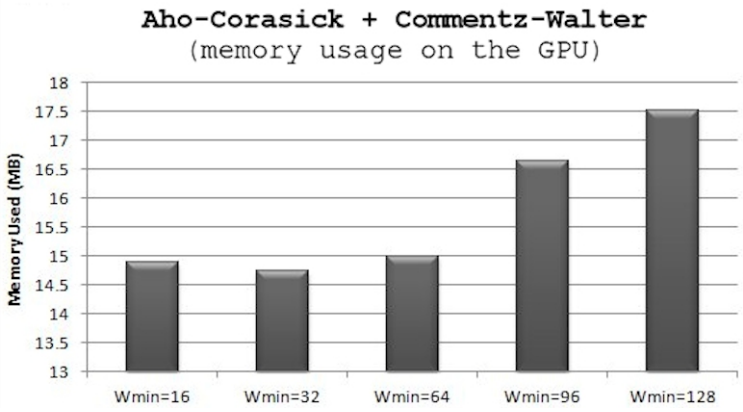


Figure 3.35: The total memory usage for the hybrid-parallel GPU implementation of AC-CW.

other (kernel 2) was running the regular virus signature scanner.

The memory usage was the lowest for $w_{min} = 32$, being about 14.75 MB.

In order to test the bandwidth, we have chosen to test with three different scenarios, each employing a different number of threads to run on the GPU - the first scenario used 128K threads, the second used 512K and the third used 1,024K threads (over 1 million threads in total). We performed a test where the AC-CW approach was used with a default Aho-Corasick implementation (not supporting regular expression scanning, Figure 3.36), and a second test where we had used the extended Aho-Corasick implementation in the first kernel (Figure 3.37).

For 1 million threads, for AC-CW in Figure 3.36 the bandwidth reached 1,420Mbps, almost half the SATA-II maximum theoretical throughput, and about 34 times faster than the equivalent CPU implementation for a value of $w_{min} = 128$. When using the extended Aho-Corasick though, performance dropped from 50% for $w_{min} = 16$ to about 37% when using 1,024K threads and $w_{min} = 128$ (for the rest of the values, the performance was about half as the standard Aho-Corasick, similar to the CPU

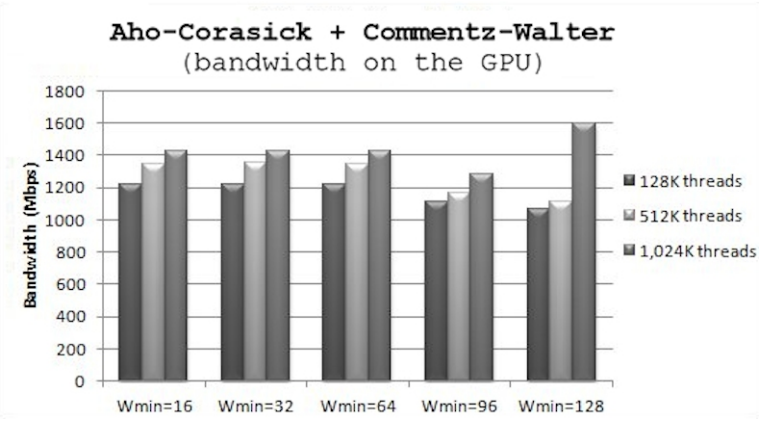


Figure 3.36: The bandwidth performance of the AC-CW implementation on the GPU.

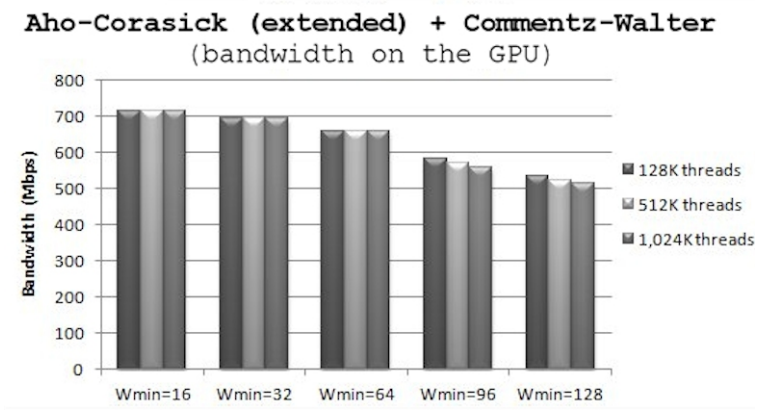


Figure 3.37: The bandwidth performance of the extended Aho-Corasick + Commentz-Walter implementation on the GPU.

implementation).

3.5 Summary

This chapter focused on describing the particular challenges involved when performing virus scanning and malicious code detection, starting with a short motivation on why we consider this is still an important subject today, continuing with the ClamAV database signature format and discussing a series of known and recent CPU and GPU-accelerated frameworks to virus scanning, such as Split-Screen, GrAVity, MIDeA or PFAC. Next, we focused on the challenges involved in achieving efficient and real-time virus scanning in modern systems today, with the final aim being to formalize, implement and test an efficient storage model which may be successfully used in achieving higher bandwidth performance for transfers between the host and the device and vice-versa, as well studying which hybrid-parallel memory models are the most

efficient for performing virus scanning, both on the CPU and on the GPU.

In order to build an efficient storage model, we had to take into consideration several facts representing the current state-of-the-art in virus scanning:

- we start with the open-source ClamAV antivirus, and continue by analyzing its memory storage format, algorithms employed and the virus signature formats used in describing malware;
- we discuss the challenges of using GPU-based acceleration and the limitations imposed by the resources of the GPU when employing it in a hybrid system;
- we present some of the most recent frameworks used in malicious code detection and network intrusion detection systems, analyzing their performance and focusing on their efficiency compared to existing solutions;

Based on the above, we conclude that our proposed storage model offers additional benefits compared to other existing solutions, such as:

- it offers the maximum possible bandwidth performance for transfers between the host memory and the device memory, and vice-versa;
- it offers an efficient storage model for completely storing the entire automaton into memory, allowing it to be parsed entirely through simple offset references;
- it imposes no limitation on the order of nodes, its only restriction being related to the consecutive storage of all immediate descendants of a given node;
- it offers better performance than other memory models, by always employing constant storage space, as discussed in Section 3.4.3;
- it bridges the gap between CPU and GPU-based hybrid algorithms involving pattern matching automaton, as discussed in Chapter 4;
- it opens up new research opportunities for building self-modifying automaton which can adjust their size and structure based on dynamic demands;

We have proposed and tested a hybrid-parallel model which employs multi-threaded scanning using different variants of multiple pattern matching algorithms, and have concluded that the best performance is obtained using our storage model for a hybrid-parallel combination of the Aho-Corasick and Commentz-Walter algorithms, with the first scanning regular expressions in patterns and the second scanning for regular virus signatures. We also tested the performance of such a model on both the CPU and the GPU, and issued a few remarks about their performance and behavior at runtime.

The summary of the work conducted herein shows that our approach is a highly viable method for implementing and storing the most common types of pattern matching automaton used in both network intrusion detection systems and virus scanners, offering flexibility, high performance and efficiency in the implementation.

Chapter 4

An Efficient Model for Hybrid CPU/GPU-Accelerated Construction of Very Large Automata

Contents

Several research areas work with large or very large pattern matching automata. One of the most conclusive examples is the virus signature matching process, where

the number of viral signatures is in the order of tens of thousands or more, with the corresponding automata reaching a high number number of nodes, in the order of millions or tens of millions. While the algorithms themselves only require the automata to be built when running, without the need to construct them on-the-fly, accelerating their construction could have important benefits in the long run, allowing not only self-adjustable automata to be built, but also permitting implementation of such automata on different hardware platforms and devices.

In this chapter we are proposing a very fast, low-storage and innovative approach to efficiently constructing pattern matching automata using a hybrid CPU/GPU-accelerated architecture, and we are proposing, to the best of our knowledge, the first parallel model to allow such a construction, that can fully benefit from the best of both computational units.

Heavy applications of pattern matching automata may be found in the intrusion detection system industry. Both network IDS [DPP07, VAP+08, Yun12, MK06, CY06] and virus scanners [Lee07, Pun09, VI10, PN12, Pun13] make use of these data structures for matching suspicious data packets or finding viral infections. Performance in such situations is critical for building real-time environments and offering responsive solutions which do not slow down the system considerably. The GPU-industry had made processing power more accessible to the end-consumer, allowing more algorithms to be parallelized and rewritten so they fully benefit from the GPU acceleration. Besides the IDS applications of pattern matching, there are numerous others in the field of bioinformatics [Dud06, Tum10, BS11], as well as natural language processing [Fre06, ZSS11] and many others.

4.1 Background

While there are many studies on improving the performance of the pattern matching automata discussed, there are a very limited number of such studies dealing with the construction performance of these algorithms and ways of optimizing that. The reason why this happens is caused by the run-time performance of both the Aho-Corasick and Commentz-Walter algorithms, which is completely separated from the construction phase and therefore it is assumed that the construction is performed upfront. While this is true and applicable, the uncompromising fact remains that for very large automata, the construction phase remains quite important due to the tremendous amount of computational power involved when dealing with a very large number of nodes in the tree.

Kouzinopoulos et al. in [KM11] have tempered with the preprocessing performance of pattern matching algorithms. They had used a binary alphabet and an alphabet of size 8, which is very low and not applicable to many fields of research nowadays (by comparison, the virus signature matching uses an alphabet of size 256).

Their keywords, in total ranging from 10,000 to 100,000, were having lengths of 8 to 32 and were taken from the E.coli genome, the SWISS-PROT Amino Acid sequence database, the FASTA Amino Acid (FAA) and FASTA Nucleidic Acid (FNA) sequences of the A-thaliana genome. One important observation made by the authors as a result of the test, regarding the Commentz-Walter algorithm, was related to the increase by 50 times in the preprocessing time when larger keyword sets were used. In practice, the Commentz-Walter algorithm preprocessing stage is very complex, as discussed in Section 2.1.2.2.

One important direct application of the construction phase performance is related to the update procedure in real-time antivirus scanners: a common approach to virus vendors when issuing updates to the virus signature database is to remove the currently loaded automaton from memory and re-map it from disk, where it is already pre-computed and ready to use. This does not require the structures to be reconstructed, possibly only a re-mapping of the pointers if needed, and eludes the preprocessing stage therefore entirely. However, whenever preparing the disk-mapped version of the automaton, antivirus vendors still need to rebuild the automaton on every change performed to it, and the pre-processing stage can no longer be ignored. Furthermore, as we have discovered in our experiments using the ClamAV [cla12] virus signature database, the construction of a large Commentz-Walter automata can take hours even on some of the fastest consumer CPUs today, which makes the update procedure of such automata very difficult and unfeasible to use in real-world scenarios.

4.2 A Hybrid GPU-Accelerated Architecture

In [PN12], also discussed in Chapter 3, we have proposed an efficient storage mechanism for the two most common types of pattern matching automata, Aho-Corasick [AC75] and Commentz-Walter [CW79]. Using this approach, the entire automaton can be stored efficiently in the memory of the device, where it can be parsed allowing multiple GPU threads to access it as needed.

The approach presented as follows offers a model for the fast construction of the Aho-Corasick automaton, and two algorithms (noted *algorithm A* and *algorithm B*) for the construction of the Commentz-Walter automaton. We are going to test its performance by using the ClamAV [cla12] virus database and a set of 62,302 virus signatures from the version of the database dated March, 2012. Figure 4.1 shows the overall architecture of the system along with the steps involved. The basic idea is to combine the workload performed on the CPU with the computational resources offered by the GPU in an attempt to increase the efficiency of the overall system.

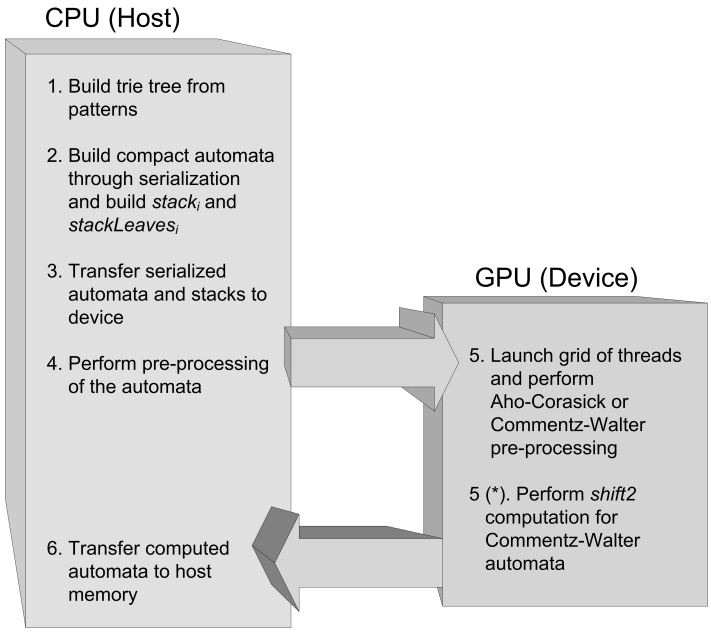


Figure 4.1: Our hybrid CPU/GPU approach to the fast construction of the two types of automata. Steps marked (*) are only applicable to the Commentz-Walter automaton.

4.3 The Storage Format

Since both types of the automata rely on the trie-tree concept, their nodes will have common elements. In our case, these elements are: a bitmap of size $\mid \Sigma \mid$, where Σ is the input alphabet (in our situation, the alphabet size is 256), the offset pointing to the start location in the stack of the children of the current node (which may be 0, or any other value to that matter, if the popcount of the bitmap is zero). At this stage, two different approaches may be used for performing suffix computations: the first would store the patterns in GPU memory, and additionally storing at each node the pattern ID, as in Figure 4.2; a second approach would avoid storing the patterns in the device memory, but requires that we store, for each node, an offset to its parent (which would occupy 4 bytes if using a 32-bit unsigned integer value) and an additional character (occupying 1 byte), specifying the last matched character that led to the current node (e.g. for a node v of depth d belonging to pattern ID p_{id}, i is the character at position v in the pattern $patterns[p_{id}]$). Both approaches used do not affect the runtime performance, but they affect the storage space required for the automaton, and are suitable each in different situations: for instance, the second approach is only preferable when $\Omega > N \times (sizeof(unsigned\ int) + 1)$, where Ω is the total length of all patterns and N is the number of nodes in the automaton; if the inequality is not verified, the first approach is the preferable one, as it uses less memory.

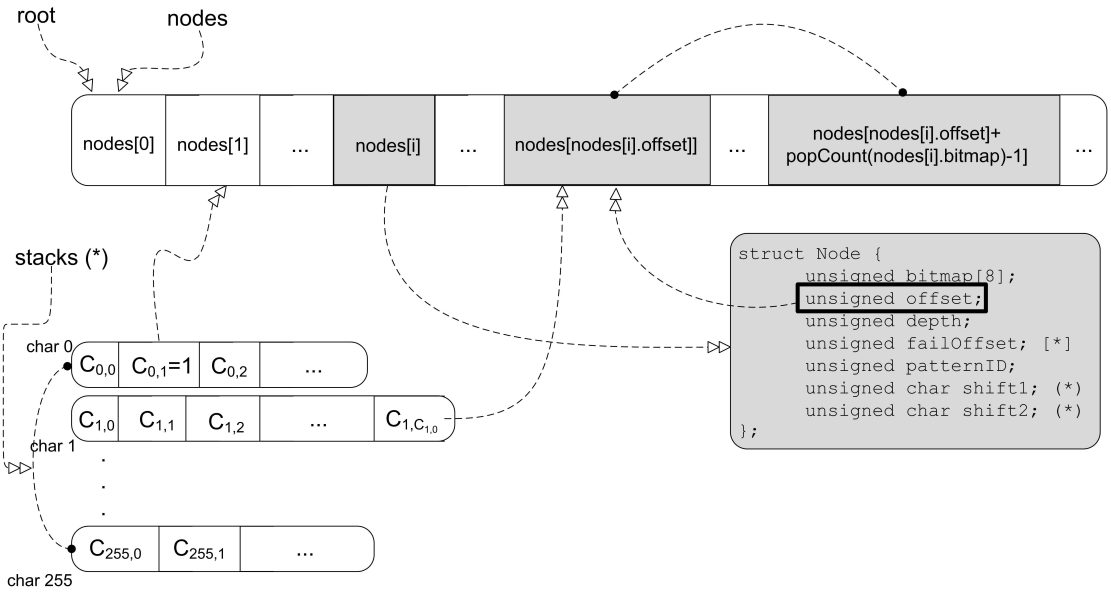


Figure 4.2: Our approach to storing the automata in memory. Elements marked with [*] correspond to the Aho-Corasick automata only, while those marked (*) correspond to the Commentz-Walter automata only.

Having P patterns, we are creating an array called *patternIndexes* of size *P+1* which stores, the each pattern p_i, the starting offset of the pattern in the stack of nodes we have built with our model (Figure 4.3).

4.3.1 The Aho-Corasick Automaton

For the Aho-Corasick automaton, we are using an extended version of the approach we proposed in [PN12] and we are storing, in addition to the fields in the original approach, the *pattern ID number* and the *depth of the node* in question. This information is used for parsing the patterns in Figure 4.3 in order to perform failure

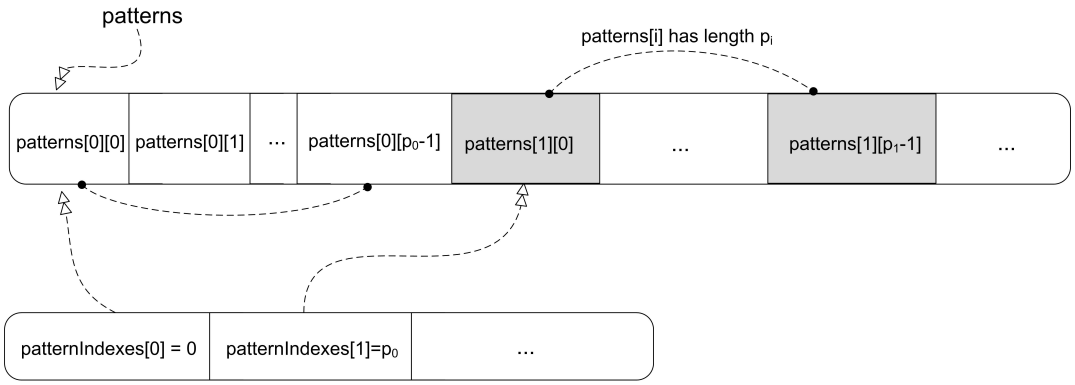


Figure 4.3: Storing patterns in the GPU (device) memory for both types of automata.

function computation.

4.3.2 The Commentz-Walter Automaton - Algorithm A

The storage approach for the Commentz-Walter algorithm A is presented in Figure 4.2. Starting with the model we proposed in [PN12], we are using an additional stack for storing only the leaves of the tree built, which will later on be used to help compute the *shift2* distances faster. The adapted algorithm from Section 3.4.3 for constructing the stack of nodes along with the stack of leaves is presented as follows:

- *Initialization*
  1: $topOfNodeStack \leftarrow 1$ (top of node stack)
  2: $topOfLeavesStack \leftarrow 1$ (top of leaves stack)
  3: $currentPos \leftarrow 0$ (position in the stack)
  4: $node$ (the currently processed node)

- *function addNode(node, currentPos)*
  1: $nodeStack[currentPos] \leftarrow node$
  2: $nodeStack[currentPos].offset \leftarrow topOfNodeStack$
  3: add in hash ($key \leftarrow node$, $value \leftarrow currentPos$)
  4: $pc \leftarrow popCount(node.bitmap)$
  5: **if** $pc = 0$ **then**
  6: $\quad topOfLeavesStack \leftarrow topOfLeavesStack + 1$
  7: $\quad leavesStack[topOfLeavesStack - 1] \leftarrow currentPos$
  8: **end if**
  9: $old \leftarrow currentPos$
  10: $topOfNodeStack \leftarrow topOfNodeStack + pc$
  11: **for** $i \leftarrow 0$ to $pc - 1$ **do**
  12: $\quad$ addNode($node.child[i]$, $old + i$)
  13: **end for**

We are employing here the same hash-based technique for restoring pointers, by inserting all pointers of the original tree in the hash and then, on re-parsing the serialized tree, replacing each pointer with its offset from the hash.

4.3.3 The Commentz-Walter Automaton - Algorithm B

The algorithm B proposed for the Commentz-Walter automaton represents an improved version of algorithm A, and uses the same storage format (Figure 4.2). In this approach, we are going to reduce the computation of the *shift1* jumping distance by building a stack for each character in the alphabet (in total, at most 256 stacks for an alphabet of size 256, such as the ASCII charset). Using this methodology,

$stack_i$ represents the list of nodes L of the automaton, so that if *v is a node in L* then *word(v) ends in character "i"*. As the construction of the tree relies heavily on the suffix computation stage at every node in the tree, we are going to use the stacks to limit the number of searches performed for a node whenever performing the suffix computation. Furthermore, using stacks allows us to better distribute the workload in a data-parallel implementation of the algorithm as will be shown later on.

Following the same analogy, the *shift2* jump distance computation can be reduced if using another set of stacks, of the same size as the alphabet, however in this case each stack will only contain the leaf nodes in the automata. A formal description of the approach is: $stackLeaves_i$ represents the list of nodes L_{leaves} in the automaton, so that if *v is a node in* L_{leaves} then *word(v) ends in character "i"* and *v is a leaf node.*

4.4 Optimizing the Performance of the Pre-Processing Stage for the Commentz-Walter Automaton

Based on the definitions of the *shift1* and *shift2* distances in [CW79], we will demonstrate as follows a few properties which will help significantly reduce the workload during the pre-processing stage of this automaton.

Proposition 4.4.1 (Parent nodes have the smallest shift1 values) *Assuming that we are located at node v of depth* Δ *in the Commentz-Walter tree, then all shift1 values of all child nodes at all higher depths in the tree have a value of at least shift1(v).*

Proof We will demonstrate this by *reductio ad absurdum*. Let's assume that there is a node v of depth Δ having $shift1_v{=}s_1$. This means by definition that there is at least one node *v'* of depth $\Delta + s_1$, for which word(v) is a suffix of word(*v'*), and that s_1 is the minimum value possible for which this property is verified. Assuming now that there is a second node, z at depth $\Delta + d$, where $d > 0$ for which $shift1_z{=}s_2 < s_1$, that means that there is at least one node *z'* in the tree of depth $\Delta + d + s_2$ for which word(z) is a suffix of word(*z'*). Since word(v) is a prefix of word(z) by construction, that results in word(v) being a prefix of word(*z'*) as well. The ending position of the prefix word(v) in word(*z'*) can be then computed as: depth of *z'* minus the distance between z and v, which translates to $\Delta + d + s_2 - ((\Delta + d) - \Delta) = \Delta + s_2$, meaning that the node w at depth $\Delta + s_2$ verifies the property: word(v) is a suffix for word(w), which would contradict the assumption that s_1 is the smallest possible positive integer value for which that property is verified.

Proposition 4.4.2 (Depth-limited search for shift1 computation) *Assuming that we are located at node v of depth* Δ *in the Commentz-Walter tree, then we only need to search from nodes of depth* $\Delta + 1$ *to* $\Delta + W_{min} - 1$ *to correctly compute shift1.*

Proof By definition, *shift1*=min{ W_{min}, {l; l=d(*v*')-d(*v*)}}, and since the first member of the *min* function is W_{min}, it results that if the other other member has higher values, the expression would simplify to the first member. It is therefore sufficient to look, from the current node depth, at most $W_{min} - 1$ extra depths to find the smaller possible values for *shift1*, if they exist.

Proposition 4.4.3 (Limiting searches for shift2 computation) *For each node v in the Commentz-Walter tree, then we only need to determine if word(v) is a suffix of word(z), where z is a leaf node in the tree, to correctly compute shift2.*

Proof By definition, *shift2*=min{*shift2*(parent node of *v*), {l; l=d(v')-d(v)}}, where *v'* is a leaf node in the tree. If none of the leaves in the tree have word(*v*) as a suffix, then it is sufficient to recursively parse the tree when all nodes have been processed and recursively compute $shift2_v$=*min(shift2 value of parent node of v).*

4.5 Achieving Automaton Construction Parallelism

For both types of automata, we will follow a data-parallel decomposition model, applied to the different types of scenarios as follows.

4.5.1 The Aho-Corasick Automaton

The most computationally intensive part of the Aho-Corasick pre-processing stage is the failure function computation. In order to parallelize this process, each node in the three is assigned a thread in the GPU, executing the same kernel. For each thread *t* assigned to node *v* of depth *depth > 1*, we verify if suffix s_l of length *l* for *word(v)* exists in the tree, beginning with the longest suffix length, which is *depth(v)-1* and if found, the failure function indicates to the offset in the stack where this suffix is found, and the thread terminates.

1: **if** $node.depth <= 1$ **then**
2: $node.failOffset \leftarrow 0$
3: stop
4: **end if**
5: $node.failOffset \leftarrow 0$
6: **for** $l \leftarrow node.depth - 1 downto 1$ **do**
7: **if** suffix s_l of $word(node)$ is matched by the automata at offset $matchOffset$ **then**
8: $node.failOffset \leftarrow matchOffset$
9: stop
10: **end if**
11: **end for**

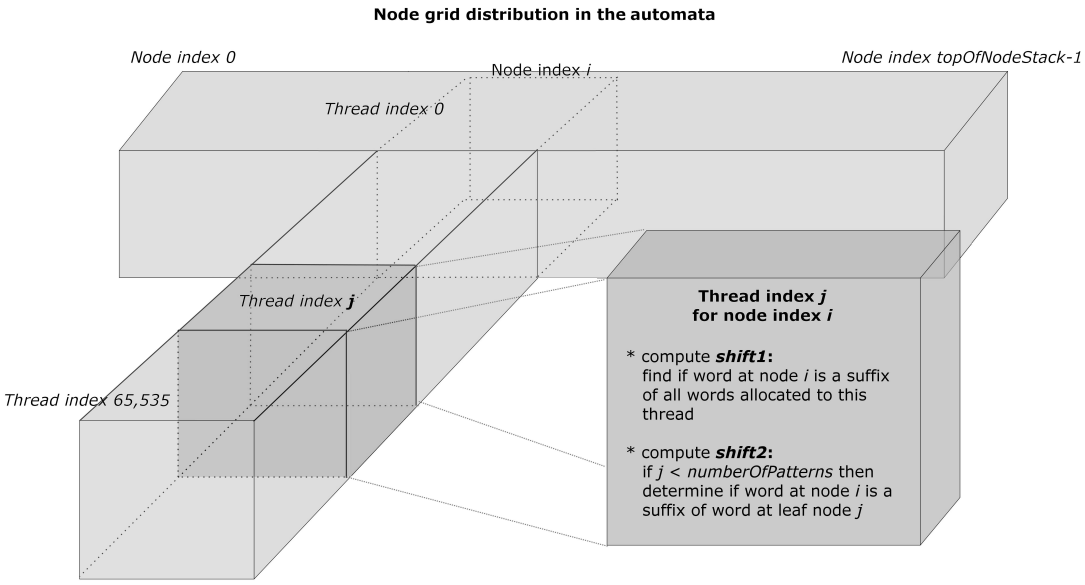


Figure 4.4: Grid structure of our hybrid-parallel model for Commentz-Walter algorithm A.

4.5.2 The Commentz-Walter Automaton - Algorithm A

Following the same data-parallel decomposition approach for the Commentz-Walter automaton, we have built a hybrid-parallel model that uses a data-parallel level approach to parse the stack of nodes we had constructed in Section 4.3.2 for computing the *shift1* jump distance, and a task-parallel approach which assigned part of the available threads in the GPU to the *shift2* jump distance computation. For each node of the automaton, we have assigned 65,536 threads to each node (the maximum allowed as a grid dimension in CUDA is 65,535), where each thread of index i performs comparisons between the node that it is associated with, and nodes from $i \times \frac{topOfNodeStack}{65{,}536}$ to $(i+1) \times \frac{topOfNodeStack}{65{,}536} - 1$, where $topOfNodeStack$ is the total number of nodes in the stack. In order to achieve this, we created a kernel that uses 128 threads per block and a grid of blocks of dimensions $\frac{topOfNodeStack}{128} \times 65,535$ (Figure 4.4).

4.5.3 The Commentz-Walter Automaton - Algorithm B

Algorithm B reduces the computational workload by assigning only a limited number of nodes to a thread in the data-parallel decomposition as opposed to algorithm A. In this improved approach, we are assigning to each node v a number of T threads (where T was chosen from a wider set as discussed later), which are used for parsing the two stacks used (Figure 4.2), $stack_{node(v).c}$ $stackLeaves_{node(v).c}$ using a data-parallel approach. After finishing the processing, we are parsing the automata once more

recursively and setting *shift2* values based on the relationship defined in the original algorithm from Section 2.1.2.2, *shift2=min(shift2(parent), current value)*.

- *Initialization*
  1: $nodes[0].shift1 \leftarrow 1$ (for root), $nodes[0].shift1 \leftarrow W_{min}$ (for others)
  2: $nodes[0].shift2 \leftarrow W_{min}$ (shift2 initialization)
  3: *node* (the currently processed node), T (number of threads per node)
  4: T (number of threads per node), N (total number of nodes)
  5: *patterns* (the list of patterns), $patternIndexes$ (the index list)

- *Jump distance computation*
  1: **if** ThreadID $> N \times T$ **then**
  2: Stop
  3: **end if**
  4: $nodeIndex \leftarrow \frac{ThreadID}{T}$
  5: $toProcess \leftarrow ThreadID \mod T$
  6: $charIndex \leftarrow node.depth - 1$
  7: $patternStart \leftarrow patternIndexes[node.patternID]$
  8: $c \leftarrow patterns[patternStart + charIndex]$
  9: call $computeShift1()$
  10: call $computeShift2()$

- *setShift1(node, value)*
  1: $atomicMin(node.shift1, value)$
  2: **if** $node.shift1 = W_{min}$ **then**
  3: for all *child* of *node*: $child.shift1 \leftarrow W_{min}$ (recursive)
  4: **end if**

- *computeShift1()*
  1: **if** $node.shift1 <= W_{min}$ **then**
  2: Stop
  3: **end if**
  4: **if** $T > stack[c][0]$ **then**
  5: **if** $toProcess = 0$ **then**
  6: $from \leftarrow 1$
  7: $to \leftarrow stack[c][0]$
  8: **else**
  9: $from \leftarrow 1$
  10: $to \leftarrow 0$
  11: **end if**
  12: **else**

13: $from \leftarrow 1 + \frac{toProcess \times stack[c][0]}{T}$
14: $to \leftarrow \frac{(toProcess+1) \times stack[c][0]}{T}$
15: **end if**
16: **if** $to > stack[c][0]$ **then**
17: $to \leftarrow stack[c][0]$
18: **end if**
19: $found \leftarrow false$
20: **for** $i \leftarrow from$ up to to **do**
21: **if** $node.shift1 = W_{min}$ or $node.shift1 = 1$ **then**
22: break
23: **end if**
24: $j \leftarrow stack[c][i]$
25: **if** $nodes[j].depth > node.depth$ and $nodes[j].depth < node.depth + min(W_{min}, node.shi$ **then**
26: **if** $isSuffix(node.patternID, charIndex, nodes[j].patternID, nodes[j].depth - 1)$ **then**
27: **if** $node.shift1 > nodes[j].depth - node.depth$ **then**
28: $found \leftarrow true$
29: call $setShift1(node, min(nodes[j].depth - node.depth, W_{min}))$
30: **end if**
31: **end if**
32: **end if**
33: **end for**
34: **if** $node.shift1 > W_{min}$ or $found = false$ **then**
35: call $setShift1(node, W_{min})$
36: **end if**

- *computeShift2()*

1: **if** $node.shift2 = 1$ **then**
2: Stop
3: **end if**
4: **if** $T > stackLeaves[c][0]$ **then**
5: **if** $toProcess = 0$ **then**
6: $from \leftarrow 1$
7: $to \leftarrow stackLeaves[c][0]$
8: **else**
9: $from \leftarrow 1$
10: $to \leftarrow 0$
11: **end if**
12: **else**
13: $from \leftarrow 1 + \frac{toProcess \times stackLeaves[c][0]}{T}$

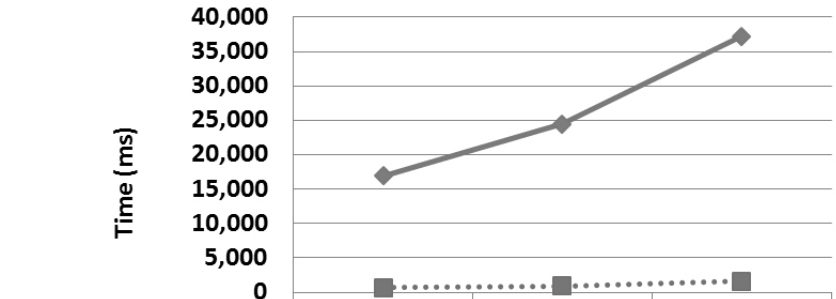
```
    to ← ((toProcess+1)×stackLeaves[c][0]) / T
end if
if to > stackLeaves[c][0] then
    to ← stackLeaves[c][0]
end if
for i ← from up to to do
    if node.shift2 = 1 then
        break
    end if
    j ← stackLeaves[c][i]
    if nodes[j].depth > node.depth and nodes[j].depth < node.depth+min(W_min, node.shi
    then
        if isSuffix(node.patternID, charIndex, nodes[j].patternID, nodes[j].depth−
        1) then
            if node.shift2 > nodes[j].depth − node.depth then
                atomicMin(node.shift2, nodes[j].depth − node.depth)
            end if
        end if
    end if
end for
```

4.6 Performance Evaluation

The pre-processing stage for both types of automata studied in this chapter is the most computationally-demanding part of the construction stages. As the other stages of the construction, as defined in Figure 4.1 are performed on the CPU, we have benchmarked the preprocessing time in our performance evaluation and have compared these with the single-core CPU implementations, in order to determine the degree of speedup achieve by our proposed parallel architecture.

4.6.1 The Aho-Corasick Automaton

In order to experiment with a high number of nodes in our automata, we have used a set of 56,900 signatures extracted from the ClamAV virus database, where each pattern was of a minimum length of 16 bytes, and we have created two additional datasets by concatenating the initial file with the same set of patterns, but in reversed order. This way, we obtained a total of 113,800 unique patterns to test with, and we ended up choosing three different datasets for the benchmark: the initial dataset of 56,900 virus signatures (in total, over 6,8 million nodes), the second dataset representing the 113,800 signatures (in total, over 13.5 million nodes) and a third dataset,



	56,900 patterns	90,000 patterns	113,800 patterns
"Single-core CPU"	16,926	24,461	37,238
"GPU"	686	873	1,576

Figure 4.5: Results for the single-core CPU vs. GPU implementation of the Aho-Corasick automaton.

representing the first 90,000 signatures out of the second (totalling about 10,5 million nodes).

The results obtained have shown a similar trendline in the running time for both the CPU and the GPU implementations (Figure 4.5), with better throughput in the GPU implementation by almost 24× for the first (smaller) dataset, more than 28× for the intermediate dataset and more than 23× for the third (largest) dataset (Figure 4.6). There is an important remark to be made here: for the largest dataset, the GPU implementation had to be built as a 64-bit application given the large memory requirements; as a consequence, the performance has degraded since words in memory have double-lengths in 64-bit implementations versus the 32-bit implementations. However, the rising trend in the acceleration is certain to continue until it reaches saturation (in our case, for as low as 6,8 million nodes, meaning 6,8 million threads in the GPU, the bandwidth was not saturated, which was immediately observable in the next test employing over 10.5 million threads, which caused a performance increase compared to the previous dataset of 12%), assuming that we compile both CPU and GPU implementations using the same 32 or 64-bit architecture.

4.6.2 The Commentz-Walter Automaton - Algorithm A

Given the high-dependence of the Commentz-Walter algorithm on the length of the smallest keyword in the dictionary w_{min}, we have opted to use the same set of signatures as in Figure 3.24, having $w_{min} \in \{16, 32, 64, 96, 128\}$. The number of nodes obtained in each of these situations is shown in Figure 4.7, and their corresponding memory usage on the device is shown in Figure 4.8.

In order to test the performance of the classic approach to the precomputation stage for this automaton and the new, optimized one, based on the mathematical

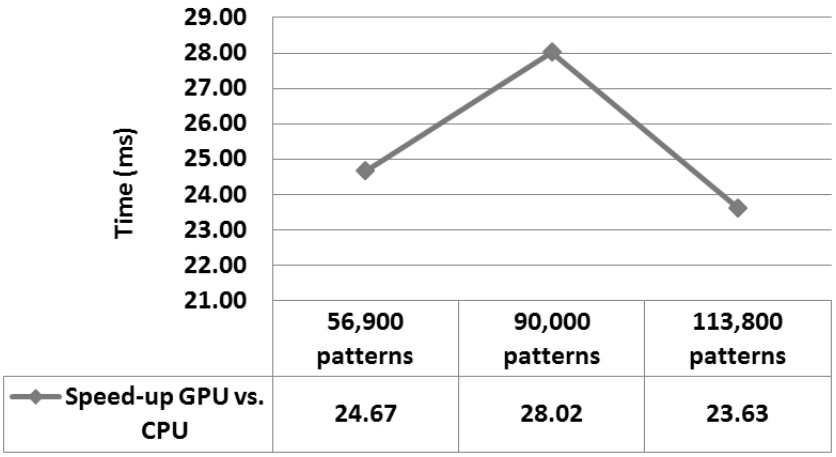


Figure 4.6: The speed-up obtained in the GPU implementation of the Aho-Corasick automaton, compared to the CPU single-core version.

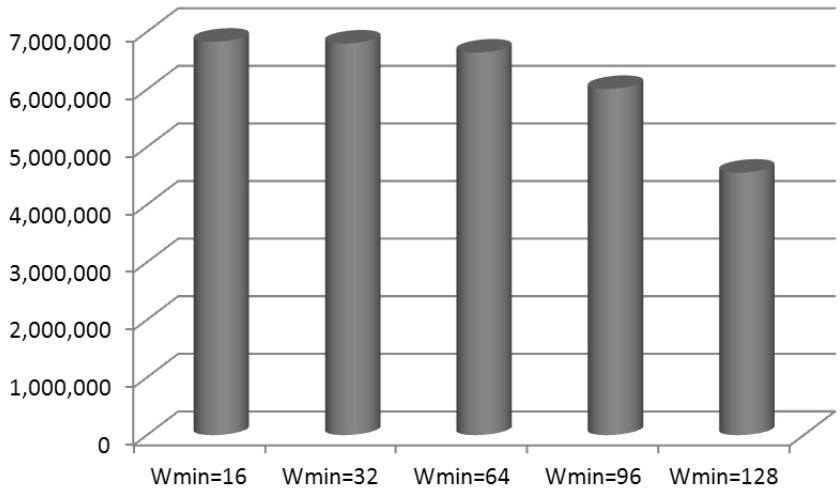


Figure 4.7: The number of nodes obtained in the Commentz-Walter automaton when using different w_{min} values.

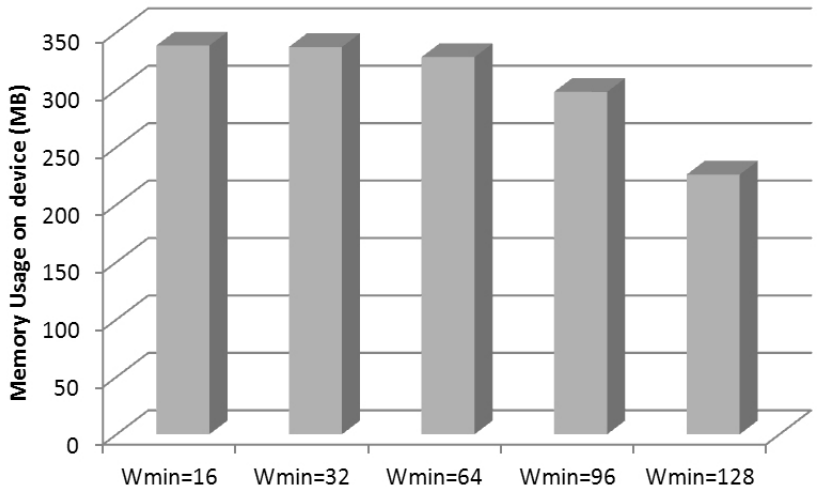


Figure 4.8: The memory usage on the device (GPU) for the Commentz-Walter automaton when using different w_{min} values.

demonstrations presented in Section 4.4, we have conducted a series of two benchmarks: the first had implemented the classic, non-optimized version of the algorithm on the CPU, and tested it against the optimized GPU implementation; the second benchmark tested the optimized version on both the GPU and the CPU, and measured the performance improvement.

The running times for the non-optimized CPU single-core implementation are shown in Figure 4.9, while for the optimized GPU implementation are shown in Figure 4.10. The overall speed-up achieved here is presented in Figure 4.11.

While the results may seem somewhat side-tracked, they are perfectly explainable here for two reasons: the first, we tested a parallel, mathematically-improved version of the algorithm on the GPU, while the single-core CPU version was implemented using the classic approach from the original algorithm definition with no improvements of any kind, and the second, much of the performance increase here is thanks to the low number of computations performed by the algorithm based on the propositions discussed earlier. In fact, by applying the *shift1* and *shift2* computation limitations, only about 700,000 nodes get processed and are updated through suffix computation for a 6.8 million-node automaton, which is less than 10%; the remaining nodes are updated through recursive calls based on proposition 1 from Section 4.4. A more fair comparison between the optimized CPU single-core implementation and the parallel GPU implementation and the resulting speedup is shown in Figure 4.12, where we noticed throughput speedups from 40× (for $w_{min} = 96$) to 50× (for $w_{min} = 64$) for the GPU implementation.

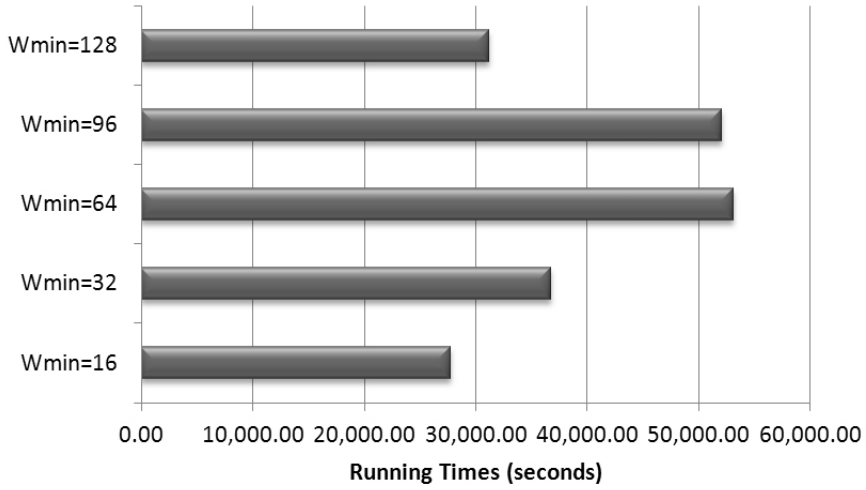


Figure 4.9: The running times for the classic (non-optimized) version of the Commentz-Walter pre-processing stage on a single-core CPU.

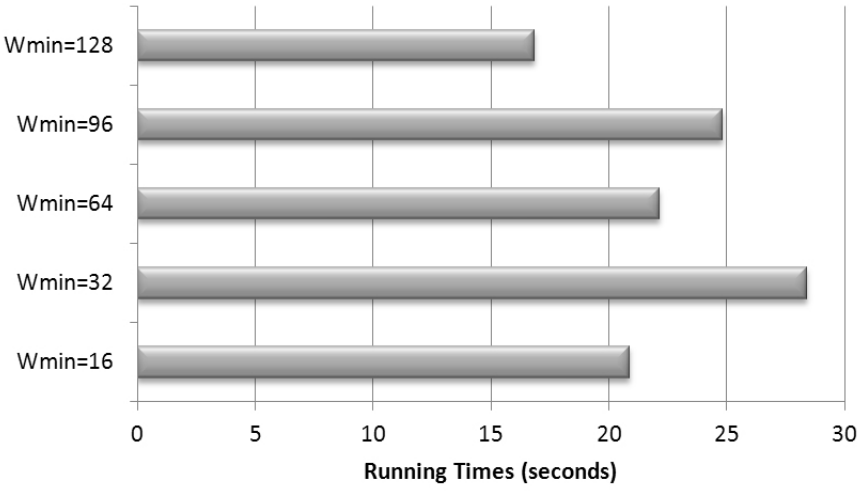


Figure 4.10: [Algorithm A] The running times for the optimized version of the Commentz-Walter pre-processing stage running on the GPU.

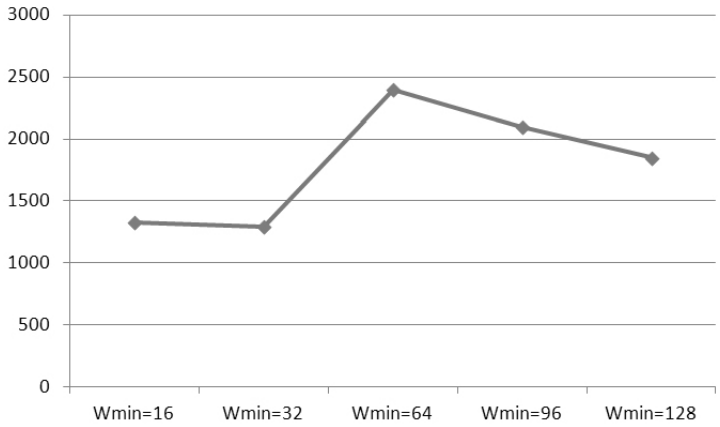


Figure 4.11: [Algorithm A] The speed-up of the optimized GPU implementation versus the non-optimized CPU single-core implementation.

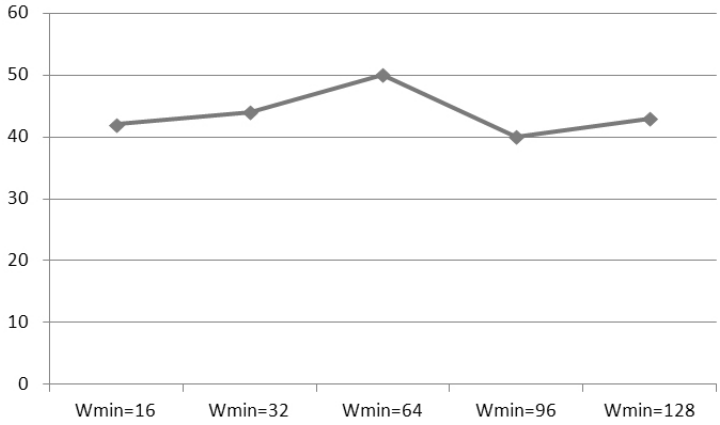


Figure 4.12: [Algorithm A] The speed-up of both optimized implementations in the GPU versus a single-core CPU.

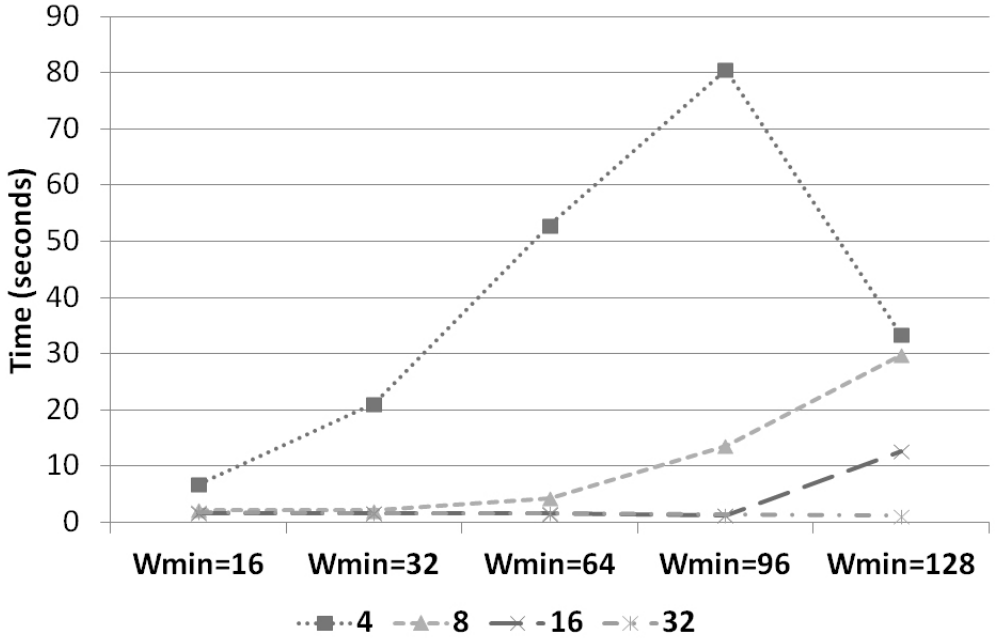


Figure 4.13: [Algorithm B] The running times on the GPU.

4.6.3 The Commentz-Walter Automaton - Algorithm B

As discussed in Section 4.5.3, for this improved version of the algorithm we have chosen different values for T, the number of threads in the GPU per node in the automaton. We started with a value of 4 and went up to 4,096. The primary interest was measuring the speed-up compared to the speed of algorithm A discussed previously. Running times of the improved algorithm B are presented in Figures 4.13 and 4.14.

The worst running times were obtained for $T = 4$, which is explainable by a non-saturation of the GPU pipelines. The best performance for $w_{min} \in \{16, 32, 64, 96\}$ was obtained when using 16 threads per node: in this situation, the algorithm finished in 1.67s for $w_{min} = 16$ (12× faster than algorithm A), 1.623s for $w_{min} = 32$ (17× faster than algorithm A), 1.498s for $w_{min} = 64$ (14× faster than algorithm A) and 1.295s for $w_{min} = 96$ (19× faster than algorithm A). The best running time for $w_{min} = 128$ was obtained for $T = 64$, when the algorithm finished the processing in 0.89s (again, almost 19× better performance than algorithm A). The results are impressive and highly useful, considering a reduction of processing time from the magnitude of hours in the classic algorithm, to just a couple of seconds or less in the optimized GPU-based version in algorithm B, and offer the first parallel approach to the pre-computation stage in the Aho-Corasick and Commentz-Walter automata, with the best improvements in the latter due to the different properties of the nodes in the automaton.

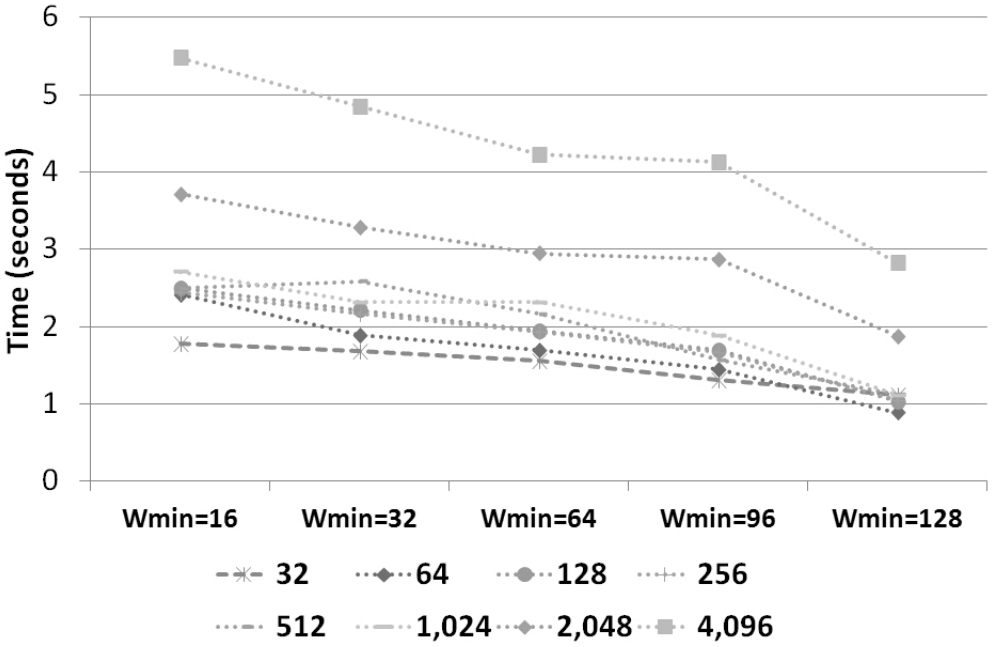


Figure 4.14: [Algorithm B] The running times on the GPU (continued).

4.7 Summary

This chapter focused on presenting the first hybrid-parallel CPU/GPU-accelerated architecture for the fast construction of very large Aho-Corasick and Commentz-Walter automata, two of the most common types of automata used in pattern matching applications in several fields of research. Starting with the efficient storage model proposed in Chapter 3, we designed and implemented an architecture which relies on the CPU-based construction of both types of automata, while employing the pre-processing stage to be computed in the GPU using a hybrid-parallel approach. We proposed an algorithm for the Aho-Corasick automaton in Section 4.5.1 and two different variants for the Commentz-Walter approach in Sections 4.5.2 and 4.5.3, with the latter offering much better performance. We based our improvements on a mathematical model outlining several properties of the jumping distances which could be used to reduce the computational workload, and which we have proven to be correct in Section 4.4. We conducted several benchmarks for determining the performance of our architecture in Section 4.6.

Our proposed architecture adds the following benefits to the existing approaches:

- it proposes the first hybrid-parallel architecture for constructing very large automata through an efficient collaboration between the CPU and the GPU, with the GPU handling the most computationally-intensive task, the pre-processing stage;

- it builds a formalism which can be applied to any type of pattern matching automata, not just to intrusion detection systems;
- it offers an efficient way to store the automata in device memory, fully benefitting from the bandwidth transfers between the host and device memory and offering direct access to the nodes of the automata in a serialized structure;
- it offers the first pre-processing results on the set of virus signatures in the ClamAV database for both the Aho-Corasick and Commentz-Walter automata;
- it reduces the processing time for such automata from the magnitude of hours in the case of the Commentz-Walter automaton to less than a couple of seconds;
- it opens up new research opportunities for implementing self-adjusting automata in real-time.

It is our belief that in the future, self-adjusting automata (containing millions or tens of millions of states) which are capable of dynamically updating their contents according to the user's requirements, will be easily implementable using parallel, GPU-accelerated approaches. We are also confident that the approach has applicability in several other fields of research where a very large number of patterns is required, such as bioinformatics or approximate pattern matching algorithms in natural language processing.

Chapter 5

Implementing Behavioral Heuristics and Hybrid Compression Techniques in the Aho-Corasick Automata

Contents

In this chapter we are going to present known approaches to using the Aho-Corasick pattern matching automata for implementing *behavioral heuristics* and *hybrid compression techniques*, in different fields of research. We will focus primarily on the intrusion detection field and particularly on the virus scanning process, but the formal models used can be easily applicable to any other type of applications involving automaton-based heuristics or requiring low-storage space.

5.1 Behavioral Heuristics

Behavioral heuristics are focusing on reducing the time allocated to a decision-making process and are applied when there is insufficient time for assessing the ideal choices in each possible circumstance. In a game of chess for instance, a player may choose to take some non-trivial choices, such as sacrificing the rook in order to gain a tactical advantage with no other immediate benefit. While this rule of thumb may seem harsh at the time, such compromises are often beneficial in the long run. Of course, there are numerous situations as well when such behavior is rendered useless in the long run (e.g. by a smarter opponent which may use the sacrifice for winning the game) and may cause more harm than actually help.

Gigerenzer et al. in [GG11] have described heuristics as being "efficient cognitive processes, conscious or unconscious, that ignore part of the information". He points out that formal models for answering the *descriptive* question (*which heuristics are the most appropriate given a certain situation?*) and the *prescriptive* question (*why are heuristics to be considered when there may be better choices?*) are essential.

5.1.1 Towards Behavioral Analysis

Using the Aho-Corasick automaton from [AC75] for dictionary-based storing has the immediate advantage of low-storage space thanks to the trie-based tree structure and the ability to convert the trie into an automaton by adding the *failure* function to it. We have presented two different applications of the behavioral analysis approach discussed in Section 5.1.3 in [Pun09] (which is also the base of our discussion) and [Pun10], using as a starting point the study performed in [PAF09], where we have analyzed and benchmarked the best approaches for storing the data collected from the wireless sensors installed in a smart-home in order to efficiently be able to collect, analyze and interpret such data. As a conclusion to this initial partially-empirical research, in [Pun10] where we have presented a model for the behavioral analysis of electrical apparatus in an intelligent home (using the approach presented in Section 3.4.4) by storing and learning their power consumption profiles. In this approach, the dictionary is comprised of a custom-built alphabet, used for storing the energy patterns in the tree. The automaton ensures accurate matching and also allows us to store the power profiles after an adaptation period (which is variable for each home) and recognize different power profiles of the apparatus afterwards by matching future consumption values through our extended automaton approach. With this implementation it is therefore easier to build additional heuristics, on top of the automaton, for classifying power consumption profiles and offer informed suggestions to the end-user about how they can improve their power consumption habits.

5.1.2 Applications to Malicious Code Detection

Heuristics have been used in intrusion detection systems in several topics of interest, such as network IDS [CH11, PAP12], cloud-security [AX11, HZB11] and virus scanning [Pun09, Pun13].

There are two major parts to the analysis of executable code in order to assess its threat level: the *static analysis* component and the *dynamic analysis* component, related to the classification of intrusion detection systems in Section 2.2.1 in *misuse detection* and *anomaly detection*. Rozinov in his Master's thesis [Roz05] discussed the challenges of efficient malicious code behavior and proposes an approach that utilizes only static analysis in order to assess the threat level of an executable, without requiring it to be ran upfront.

As Choen [Coh87] and Chess-White [CW00] have pointed out, it is not possible to construct an algorithm that can detect all possible viruses. Landi in [Lan92] has shown that the problem of static analysis can be undecidable and uncomputable. These are the premises for building formal models which can assess the threat levels of executable code, either by analyzing them before execution (through techniques such as sandboxing or emulation, as discussed in Section 2.2.2), or by analyzing them at run-time through watchdog-alike mechanisms. Several anomaly-detection-based techniques for performing malicious code detection have been proposed [SBDB01, ALJ+95, HFS98, PN97, FKF+03, Roz05, CWL07, Pun09], with one of the most common methodologies involving system-call analysis: in short terms, a program is being traced and the sequence of system calls it makes at run-time is being analyzed and a threat level is computed for it; if the result exceeds a maximum threshold permitted, the program is considered to be dangerous, otherwise it is considered to be safe.

5.1.3 A Metric-Based Heuristic for Malicious Code Detection

This section represents a review of the research we have performed in [Pun09]. We present as follows the general overview of our approach and outline its most important results and contribution.

In [SBDB01], Sekar et al. have discussed an approach for the static extraction of sequences of system calls performed by a program, directly from the program's executable code and a storage model for them. In [BGI+12], Barak et al. have shown that it is impossible to completely obfuscate a program in a bounded computational environment. A direct consequence is that no virus can completely hide itself from all possible detection techniques. Modern techniques for malicious code detection do not require access to the source code, but instead rely on different heuristics or

prediction models for classifying the programs into malicious or safe. The architecture for performing such an analysis requires the following components:

- a module to disassemble the executable code and transform the opcode mnemonics into a humanly-readable form (e.g. assembler code);
- a static analysis module for parsing the source code obtained in the previous step and determine all possible paths of execution for the given program, assembling all possible sequences of system calls as a consequence thereof;
- an algorithm for analyzing the sequences extracted and applying different heuristics for assessing the threat level and classifying it into malicious or safe.

5.1.3.1 Program Slicing

The system-call sequence extraction process mentioned earlier is based on the concept of *program slicing*, commonly found in static analysis research. Weiser in [Wei81] had proposed a slicing technique which was used later on by Sung et al. in [SXCM04] to build an architecture which captures every system-call made by the running program. Weiser had defined *program slicing* as being a method to build an abstract form of a program which is still functionally identical to the original; the reduced program, or "slice", is an autonomous program independent of the original, but preserving the behavior. His approach involved control flow analysis by parsing the program's source code. Barak et al. in [BGI+12] have also proposed a slicing algorithm which only extract the relevant parts of an executable program. Weiser's approach had been also used by Kiss et al. and Cifuentes et al. in [kKJLG03, CF97] for performing static analysis on executable code.

We will demonstrate the program slicing technique with the help of a small C program, based on the example from [Gra].

```
1  int main() {
2      int sum = 0;
3      int i = 1;
4      while (i < 1) {
5          sum = sum + i;
6          i = i + 1;
7      }
8      printf("\%d\n", sum);
9      printf("\%d\n", i);
10 }
```

The first step to performing slicing is the construction of the control dependence (CDG) and flow dependence graphs (FDG) [Gra]. Based on these two, the program dependence graph (PDG) can be constructed (Figure 5.1). At this stage, *backward slicing* is performed by determining all incoming arcs for a given node in the PDG, identifying all nodes from where these arcs are originating, and continuing the process

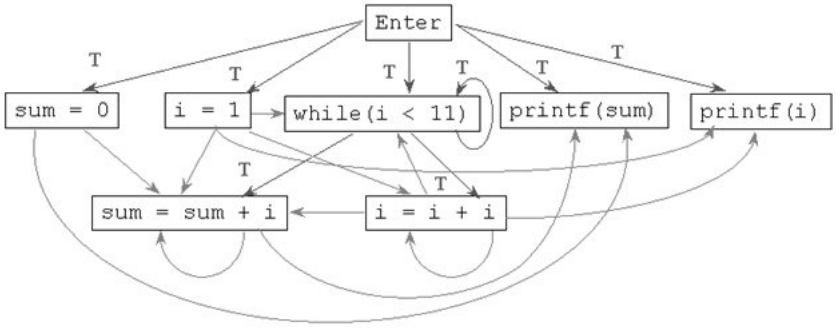


Figure 5.1: The program dependence graph (PDG) for the example code in Section 5.1.3.1 [Gra].

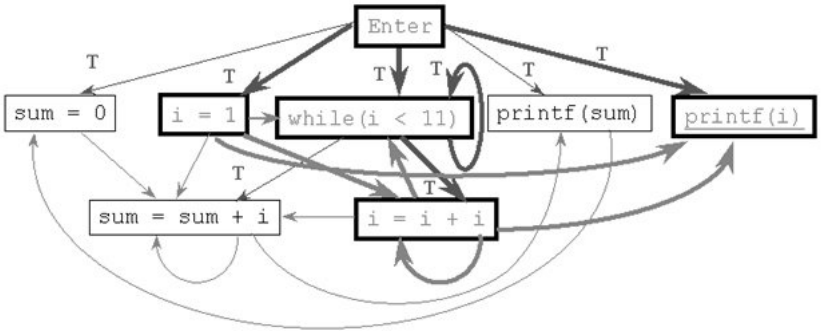


Figure 5.2: A slice starting from the last printf() statement for the PDG in Figure 5.1 [Gra].

recursively until no more new nodes are added to the list of nodes (Figure 5.2). At this stage, a program slice (as defined by Weiser in [Wei81]) has been identified. *Interprocedural slicing* builds a *system dependence graph (SDG)* for the code (Figure 5.3). We now perform backward slicing again, taking extra care to the intersecting paths of any two calls in different calling contents - such intersections must not occur, or the algorithm might run into an infinite loop. The result of the backward slicing is the program slice containing all the calls inside that portion of the code of the program (Figure 5.4).

This approach is being used to extract the sequences of system-calls that a program makes during its execution. It is easy at this stage to assign ordinal numbers to each system-call. Usually, an additional *cleanup* process removes trivial (non-malicious) system calls with no real impact on security and the remaining result is the sequence of interest for being analyzed. The cleanup process does not affect the ordinal numbers assigned to the system calls however.

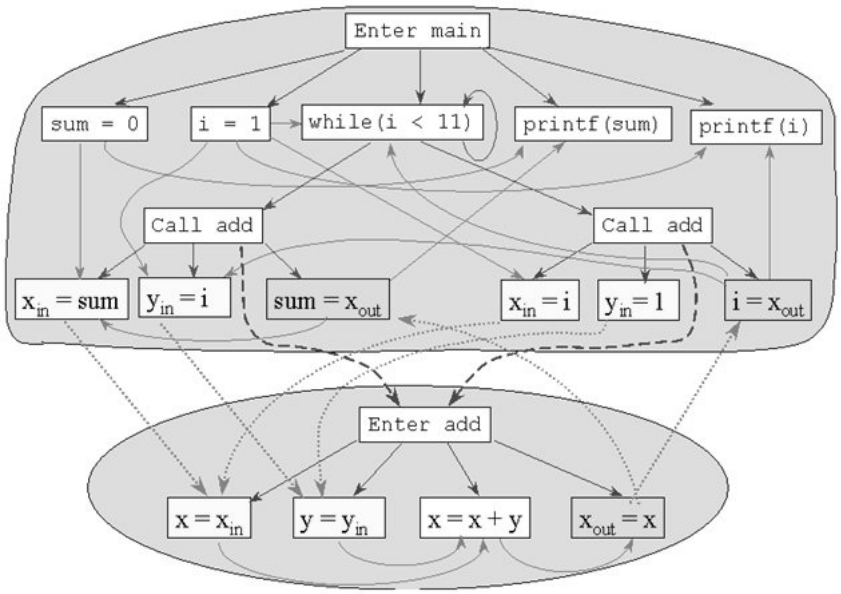


Figure 5.3: System dependence graph (SDG) for the program in Figure 5.1 [Gra].

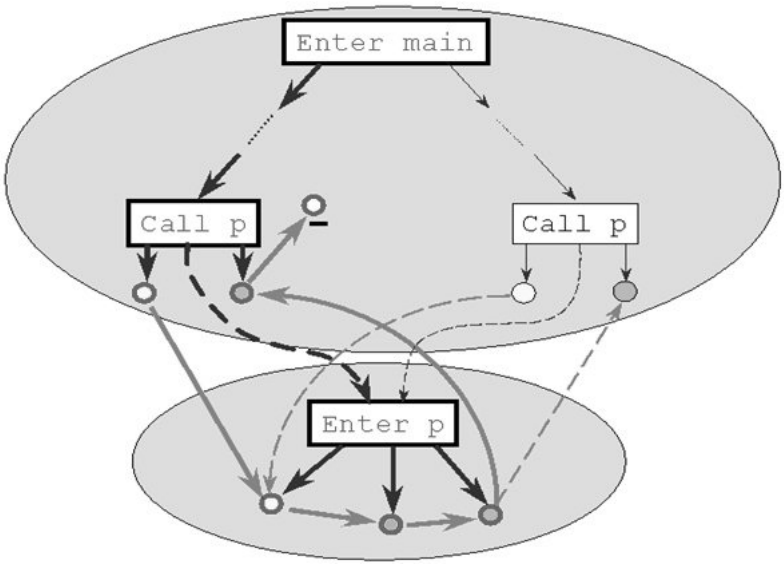


Figure 5.4: Interprocedural backward slicing for the program in Figure 5.1 [Gra].

5.1.3.2 Time Dependency

While there are many studies conducted showing how to extract, analyze and classify program slices from a binary program's executable, there is little research done related to the time-dependency of the system-calls. In practice, this dependency relation can prove to represent the difference between detecting a false positive or not. For instance, considering the following scenario: if a program partially matches a malicious system-call sequence, but stops to perform additional harmless operations for a very long time, and after a significant amount of time (when the effect of the previous sequences had been neutralized) he performs what remains of the sequence, the result would be a false positive in the dynamic analysis engine of an antivirus, although the program did not perform any malicious activity at all.

Some approaches that we proposed in [Pun09] for solving the problem mentioned are:

1. Associating a system-call with the time of the machine at the moment the call is made. This is a hardware-dependant approach which is not portable and therefore prone to errors.

2. Computing the number of instructions between system calls in the decompiled assembler source and associating an ordinal number to each system call. The number is computed according to the *program counter (PC)* value of the program, or the number of the instructions reached, but poses additional challenges with CISC architectures (which have variable length instructions, making them difficult to backtrace in static analysis).

3. One other possible approach, which we have used also, is computing the number of harmless system-calls between consecutive suspicious calls, and associating each suspicious system call an ordinal number.

5.1.3.3 Measuring Similarity Using the Bray-Curtis Metric

The Bray-Curtis distance [Bra] (also called the Sorensen distance) is a normalization method commonly used in botany, ecology and environmental sciences, viewing space as a grid in a manner similar to the city block distance. An important property is that if all coordinates are positive, the value of the distance is between 0 and 1, with 0 representing exact similar coordinates. The Bray-Curtis distance is defined as:

$$\frac{\sum_{k=1}^{n} |x_{ik} - x_{jk}|}{\sum_{k=1}^{n} (x_{ik} + x_{jk})} = \frac{\sum_{k=1}^{n} |x_{ik} - x_{jk}|}{\sum_{k=1}^{n} x_{ik} + \sum_{k=1}^{n} x_{jk}}$$

and we define the Bray-Curtis function as follows:

$$f_{BC} : \mathcal{R}_+ \to [0,1], x_{ij}\epsilon\mathcal{R}_+, \forall i,j\epsilon\mathcal{N}$$
$$\text{and } x_{jk} \geq x_{ik}\forall j,k\epsilon\mathcal{N}$$
$$f_{BC}(x_{j,n+1}) = \frac{\sum\limits_{k=1}^{n} |x_{ik}-x_{jk}|+|x_{i,n+1}-x_{j,n+1}|}{\sum\limits_{k=1}^{n}(x_{ik}+x_{jk})+(x_{i,n+1}+x_{j,n+1})}$$

Let us prove the following:

Theorem 5.1.1 (monotonicity of f_{BC}) *f_{BC} is a monotonically increasing function, that is*

$$\forall x_1, x_2\epsilon\mathcal{R}_+, x_1 > x_2, f_{BC}(x_1) > f_{BC}(x_2)$$

Proof If we make the following notations: $\sum\limits_{k=1}^{n} |x_{ik} - x_{jk}| = \alpha$, $\sum\limits_{k=1}^{n}(x_{ik} + x_{jk}) = \beta$ and $x_{i,n+1} = a$ we will need to prove that:

$$\frac{\alpha+|a-x_2|}{\beta+(a+x_2)} < \frac{\alpha+|a-x_1|}{\beta+(a+x_1)}$$

where we know that $a < x_2 < x1$. Since both the nominator and the denominators are positive, we obtain:

$$\alpha(a+x_1)+\alpha\beta+\beta(x_2-a)+(x_2-a)(a+x_1) < \alpha\beta+\alpha(a+x_2)+\beta(x_1-a)+(a+x_2)(x_1-a)$$

which later on, after performing the calculus, is reduced to:

$$\beta(x_1 - x_2) + a(x_1 - x_2) + a(x_1 - x_2) > 0 \Leftrightarrow (x_1 - x_2)(\beta + 2\alpha) > 0$$

which is true since from the hypothesis $a < x_2 < x1$.

Theorem 5.1.2 (convergence of f_{BC}) *f_{BC} is convergent towards 1, that is*

$$\lim_{x\to\infty} f_{BC}(x) = 1$$

Proof Since f_{BC} is monotonous and $f_{BC}(x)\epsilon[0,1], \forall x\epsilon\mathcal{R}_+$ from Weierstrass's theorem we deduct that it is superiorly convergent towards 1.

Let us now define the extended Bray-Curtis function, f_{BCX} as follows:

$$f_{BCX} : \mathcal{R}^{n+1} \to [0,1], x_{ij}\epsilon\mathcal{R}, \forall i,j\epsilon\mathcal{N}$$
$$\text{and } x_{jk} \geq x_{ik}\forall j,k\epsilon\mathcal{N}$$
$$f_{BCX}((x_{j0}, x_{j1}, .., x_{jn})) = \frac{\sum\limits_{k=0}^{n} |x_{ik}-x_{jk}|}{\sum\limits_{k=0}^{n}(x_{ik}+x_{jk})}$$

Theorem 5.1.3 *f_{BCX} is monotonously increasing, meaning: $f_{BCX}(v_1) > f_{BCX}(v_2)$ where $v_1 = (x_0, x_1, .., x_n)\epsilon\mathcal{R}^{n+1}$, $v_2 = (y_0, y_1, .., y_n)\epsilon\mathcal{R}^{n+1}$ and $x_i \leq y_i, \forall i\epsilon\{0, 1, .., n\}$*

Proof We base the proof on the fact that if $v = (x_0, x_1, .., x_n)\epsilon\mathcal{R}_+^{n+1}$, by fixing the last n values of the vector we obtain $f_{BCX}(v) = f_{BC}(x_0)$ and we know from the previous theorem that f_{BC} is a monotonically increasing function. Therefore, if all but the first variable of the input vector for f_{BCX} are fixed, the theorem is proven. If we now consider fixed all the values in v except x_1, we see that $f_{BCX}(v) = f_{BC}(x_1)$, etc. which proves that whenever one variable $x_i, i\epsilon\{0, 1, .., n\}$ from v is variable and all the others are fixed, f_{BCX} is monotonically increasing. In order to prove for all the variables, one can use mathematical induction, by considering fixed all variables except one and by using permutations between the variables.

We are encoding the system-calls by assigning them ordinal numbers, encoded as hexadecimal numbers - at this stage, the encoding methodology is similar to that used in the ClamAV signature database and which we have addressed in Section 3.2.1. Assuming that no sequence of system calls has a length shorter than 5, up to $2^{16} = 65,536$ can be inserted in to the dictionary, a sufficient number for all modern operating systems today. There is no limit imposed on the length of a rule. Each system call which is part of a rule will have an ordinal number assigned to it (the ordinal number of the system call before the resampling process mentioned in Section 5.1.3.1 takes place). Using an empirically determined threshold value for the degree of similarity between two sequences of system calls, we can use the above functions for performing the similarity comparison between the sequences existing in our dictionary and the ones detected in the program code.

As a consequence, the metric that we have proposed, which is based on the Bray-Curtis distance, can be used only when the ordinal values associated to the system calls detected in the program being analyzed have greater values than the corresponding positions in the dictionary (caused by the function f_{BCX} being monotonously increasing). To that effect, a rule in the dictionary which has the ordinal values {*1, 3, 6, 7, 11, 20, 21, 23, 29*} associated with it would determine an input sequence of {*1, 2, 5, 7, 11, 20, 21, 23, 29*} to cause a mismatch unless a proper threshold Θ is used, because $2 \leq 3$ and $5 \leq 6$. However, having an input sequence of {*1, 4, 8, 10, 13, 22, 25, 29, 33*} preserves the conditions required to achieve function convergence above, showing that the higher the dispersion of system-calls in time, the probability of the sequence's similarity to a rule decreases (basically, f_{BC} is increasing towards 1 as the system calls are more and more dispersed in the binary code, and we assume that as time passes the probability of the program having a malicious behavior reduces; therefore, by carefully selecting a proper threshold value Θ so that a match is obtained if $f_{BC} < \Theta$ we can ensure an accurate identification of such malicious patterns of sequences of system calls).

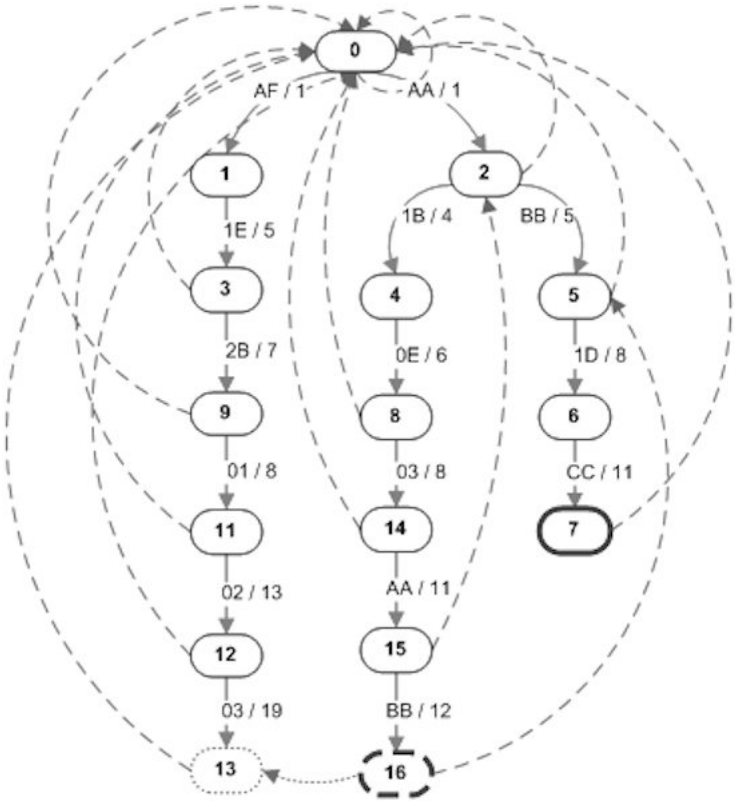


Figure 5.5: The extended Aho-Corasick automaton for the set of rules **{AF 1E 2B 01 02 03 ∗ AA 1B 0E 03 AA BB, AA BB 1D CC}**. Failure transitions are dashed lines.

5.1.3.4 A Weighted Automaton Model

The extended Aho-Corasick automaton model for our approach, which assigns weights to each transition (with each weight representing the ordinal number of the system-call), is comprised of the following elements:

1. a goto function *g(s,a)* that gives the next state entered in case of a match of the character *a*
2. each transition in the automaton from state *s*, in case of a match of the character *a*, has a weight associated to it given by a function *h(s, a)*
   - if the *edge(u,v)* is labeled by *a*, then *g(u, a)=v*
   - *g(0, a) = 0* for each *a* that does not label an edge out of the root (the automaton remains at the initial state while scanning non-matching characters)
3. a failure function *f(s)* that shows what state to enter in case of a mismatch
   - search the longest proper suffix *ps* of *word(s)* so that *ps* is a prefix of a pattern, *f(s)* is the node labeled by *ps*
4. an output function *out(s)* that gives the set of patterns matched after entering state *s*
5. we are proposing two different approaches that may be used for when performing sequence matching in the tree
   - the first is taking into consideration a minimum depth Δ_{min} of the tree, so that when the current matching is performed at a depth $\Delta \geq \Delta_{min}$ (or when we are reaching a leaf in the tree) we compute the Bray-Curtis function f_{BC} defined earlier and if $f_{BC} < \Theta$, where Θ is the threshold value, we go on with the match, otherwise we call in the failure function and report a mismatch; taking into account the minimum length of the rules in the knowledge base, we conclude that $\Delta_{min} \geq 5$.
   - a second approach is computing the Bray-Curtis distance f_{BC} only when we have reached a leaf in the tree, independent of the current depth; if $f_{BC} < \Theta$, where Θ is the threshold value, we continue the algorithm as in the normal extended automaton, otherwise we call in the failure function and report a mismatch.

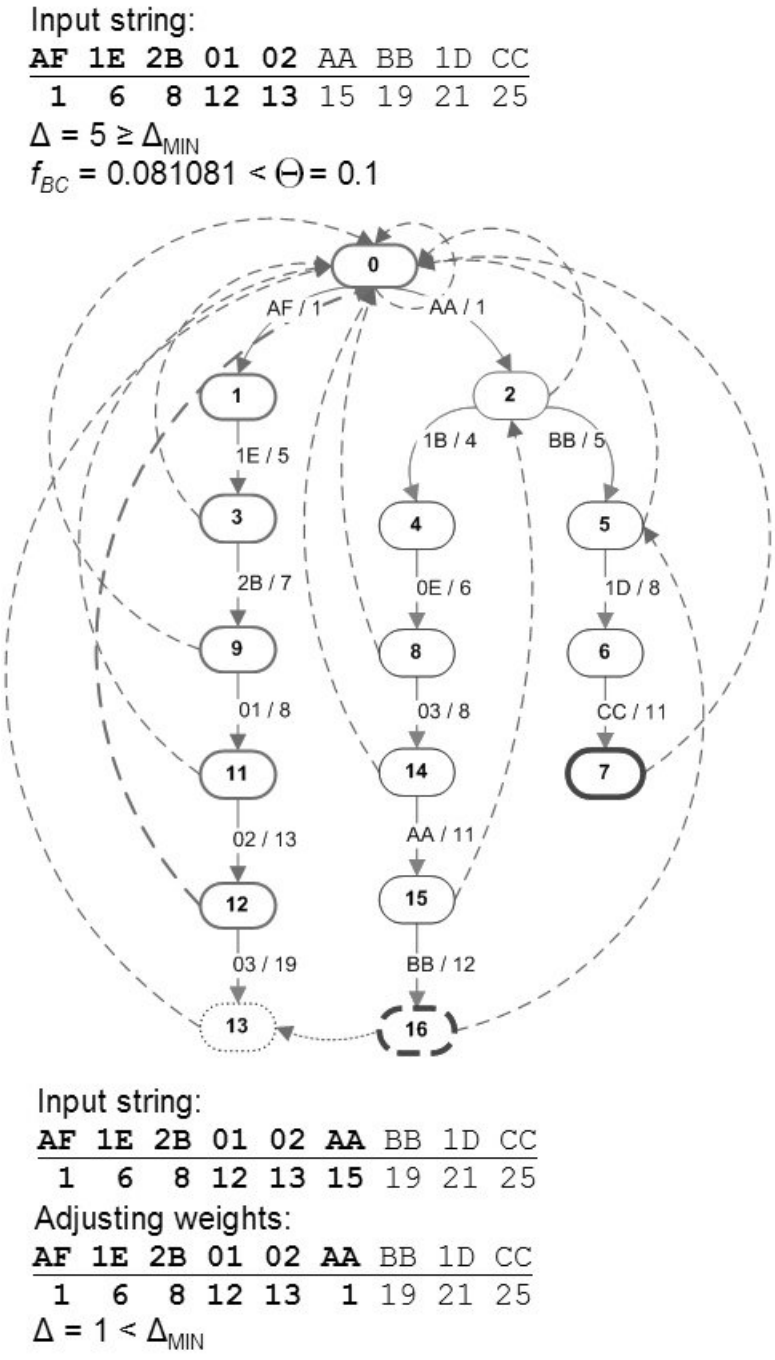


Figure 5.6: A step in the matching process for the automaton in Figure 5.5.

6. in both of these cases, the weights are automatically adjusted as the matching is being performed: if the next state (at position μ) where the matching is being accomplished is of depth λ, and the weight associated to the $(\mu - \lambda + 1)$th last accepted system call in the automaton is η, then for each system call that follows, including the current one, substract $\eta - 1$ from its associated weight and assign it as its new weight.

For example, let's take the following rules as part of our dictionary: {*AF/1 1E/5 2B/7 01/8 02/13 03/19 ∗ AA/1 1B/4 0E/6 03/8 AA/11 BB/12, AA/1 BB/5 1D/8 CC/11*} (a rule in the form *AA/10* means that the system call with the code *AA* has an ordinal value of 10 associated to it) and let's build the automaton for it (Figure 5.5). Figure 5.6 shows the automaton as it is fed the following sequence of system calls and their associated weights: *(AF/1 1E/6 2B/8 01/12 02/13 AA/15 BB/19 1D/21 CC/25)*. Considering $\Delta_{min} = 5$, once the depth in the tree is reached, the value of the Bray-Curtis function is compared to the threshold and since it verifies the similarity condition, the matching continues (Figure 5.7) until the matching is complete.

5.1.4 Performance Evaluation

We have evaluated the performance of our implementation and used sequences of variable length (up to a maximum of 100) as patterns in our dictionary, with $\Theta \in \{0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5\}$. We have then generated different input strings that were to be fed as input to the automaton and for which we have computed the Bray-Curtis normalization in order to determine the maximum average distribution of non-suspicious system calls between any two consecutive suspicious system calls so that $f_{BC} < \Theta$. This is useful for knowing the maximum average number of system calls that can exist between two consecutive suspicious system calls which are part of a sequence, before the threshold condition is no longer verified. The results are presented in Table 5.1 (a visual representation is reproduced also in Figure 5.8) and show that as the threshold value increases, so does the maximum average.

One advantage to our approach is that the threshold value can be either statically determined *a priori*, or it can be dynamically adjusted as the system requires. Good performance was obtained for $0.1 \leq \Theta \leq 0.2$, when the maxim average varies from about 6 to 13 for rules of length 50, and from approximately 9 to 20 for rules of length 75.

We have tested our implementation against the classic Aho-Corasick algorithm, using a set of 38,439 signatures from the ClamAV database [cla12] containing both simple signatures and regular-expressions. We associated weights to our model before beginning the computation, by generating them randomly. The input data used in our experiment was 7 MB in size (again, with randomly generated weights). Table 5.2

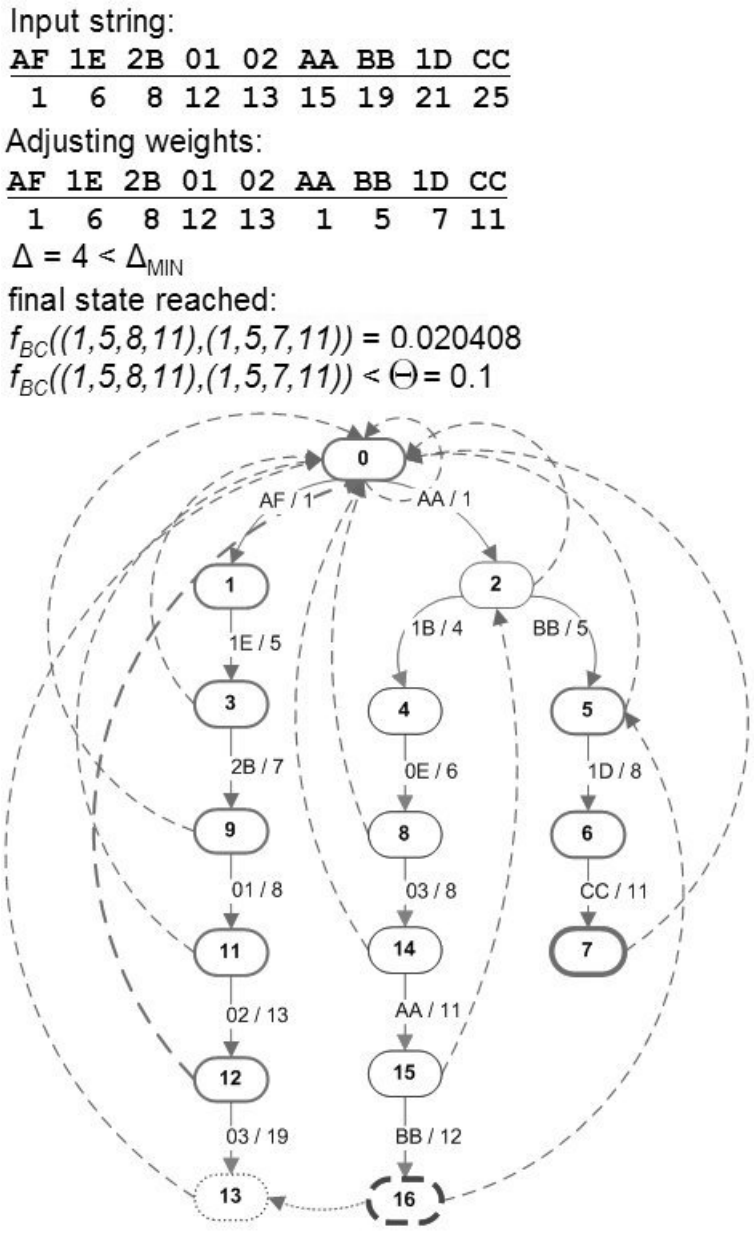


Figure 5.7: Finding matches in the automaton in Figure 5.5.

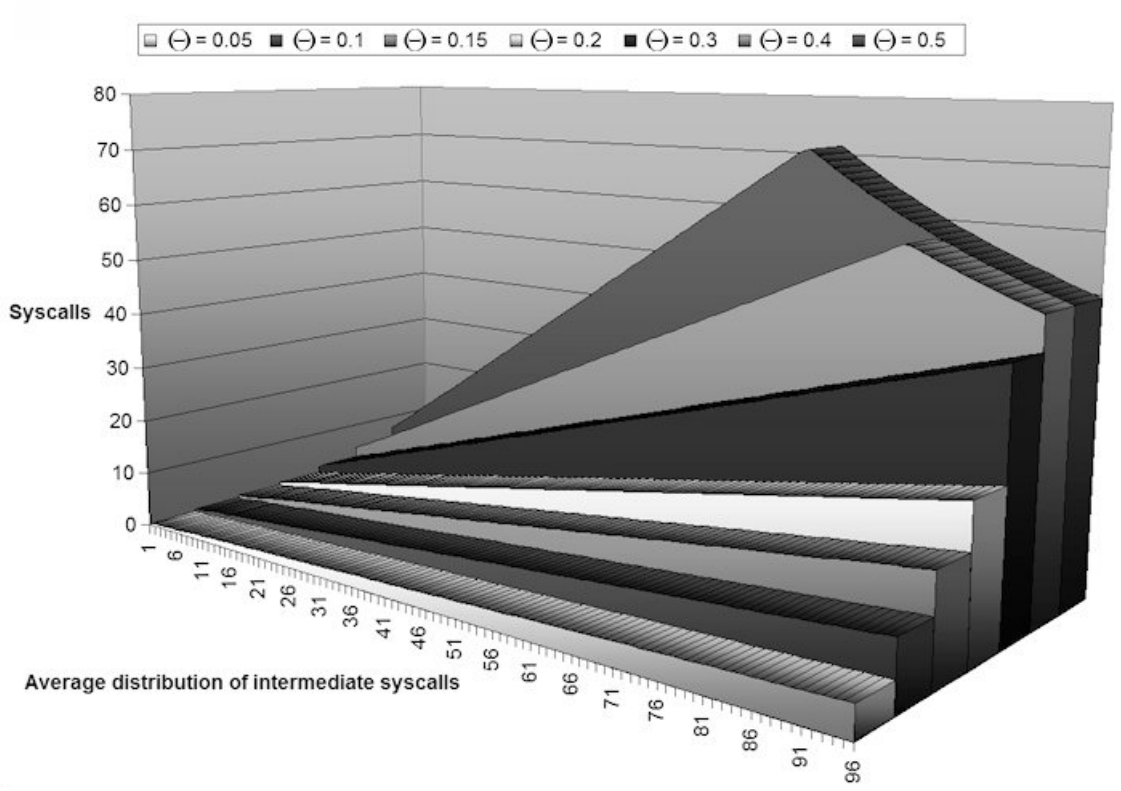


Figure 5.8: Maximum average distribution of non-suspicious system calls at different lengths for the rules in the knowledge base and for different threshold values as shown in [Pun09].

Table 5.1: Maximum Average Distribution of Intermediary Non-Suspicious System Calls for Bray-Curtis Normalization

	Rule Length				
Threshold Value	*5*	*25*	*50*	*75*	*100*
Θ = *0.05*	0.25	1.42	2.73	4.05	5.36
Θ = *0.10*	0.75	3.00	5.78	8.56	11.33
Θ = *0.15*	1.25	4.75	9.18	13.58	18.00
Θ = *0.20*	1.75	6.75	13.00	19.26	25.50
Θ = *0.30*	3.00	11.58	22.29	33.00	43.72
Θ = *0.40*	5.00	18.04	34.69	51.35	49.49
Θ = *0.50*	7.5	27.08	52.04	66.55	49.49

shows the benchmark results that had been obtained when testing the two automata on a quad-core Q6600 CPU with 4 cores.

Table 5.2: Benchmark Results of Standard AC Machine, Extended AC Machine and Bray-Curtis Weighted AC-Machine

	Aho-Corasick Implementation Results		
	Standard	*Extended*	*Bray-Curtis-weighted*
Scanning time	1.248 s	1.311 s	13.246 s
Memory consumption	119.41 MB	144.81 MB	239.04 MB
Processing bandwidth (syscalls/second)	$5{,}743.59 \cdot 10^3$	$5{,}467.58 \cdot 10^3$	$541.14 \cdot 10^3$

In the weighted implementation, additional information is being stored for each node in the automaton (almost double the size compared to the classic approach), and the processing complexity also increases whenever a partial match (at a leaf in the tree) is found. Our approach produced a throughput of 541,000 system-calls being analyzed per second, which is fully compliable with contemporary programs which usually make at most a few thousand system calls per second. We conclude therefore that our approach is suitable for implementation in a real-time heuristic malicious code detection engine, having very little impact on the performance of the overall system.

5.2 Hybrid Compression

Compressing the Aho-Corasick automaton in order to reduce its storage space has been an idea long tackled with in intrusion detection systems. Tuck et al. in [TSCV] have proposed a method for compressing the automaton by reducing single-childed consecutive nodes to a single, compressed node. Their technique was called *path-compression* and could reduce the storage requirements for the automaton if there

were many nodes have one single direct descendant. The authors have applied their approach to the Snort [sno] signature database, and was later on improved slightly by Zha et al. in [ZS08], where bitmapped nodes were used together with path-compression to achieve even lower storage requirements. While path-compression produces good results in general, it increases coding complexity for the automaton, as for instance there may be nodes having failure functions pointing to a path-compressed node; this would require us to store an additional offset for the failure pointer, which is only used when pointing inside a path-compressed node.

Intrusion detection systems build very large automata, as we have discussed in Section 4.6, which requires the development of efficient storage methodologies for these structures. In [PN12] (detailed in Chapter 3) we have proposed such an efficient storage mechanism for both CPU and GPU implementations. As follows, we will propose a hybrid compression mechanism which aims to use dynamic programming concepts in order to reduce the memory requirements for very large automata. This section represents an outline of the research performed by us in [Pun13].

5.2.1 The Smith-Waterman Local Sequence Alignment Algorithm

The Smith-Waterman local sequence alignment algorithm was proposed by Smith et al. in [SW81] as a dynamic programming approach commonly used in bioinformatics [bio] for detecting and locating the longest common subsequences in two DNA strands. It works by building a two-dimensional matrix, with each cell corresponding to a pairing of a letter from each sequence, and begins by parsing the matrix from the top-left corner, following a mostly diagonal path, as it moves to the right or down in the matrix. In case of a match for the two letters compared, the path traverses the matrix diagonally. The algorithm is comprised of two steps: the computation of a similarity score and the search for the optimal alignment. A value $v_{i,j}$ in the matrix for sequences A (of length n) and B (of length m) is being computed as below in *O(mn)*, where *sub*, *ins* and *del* are penalties for substitution, insertion and deletion:

$$v_{i,j} = min \begin{cases} v_{i-1,j-1} & A_i = B_j \\ v_{i-1,j-1} + sub & A_i \neq B_j \\ v_{i,j-1} + ins & \\ v_{i-1,j} + del & \end{cases} \tag{5.1}$$

An example is shown in Table 5.3 as follows:

For our implementation, we used the common values of $ins = 1, del = 1$, $sub = 2$. An important application is in the problem of finding the longest common subsequence (LCS) and finding all common subsequences between two given sequences.

Table 5.3: An example of the Smith-Waterman local sequence alignment algorithm [SW81]

		A	**C**	**G**
	0	1	2	3
A	1	0	1	2
T	2	1	2	3
C	3	2	1	2

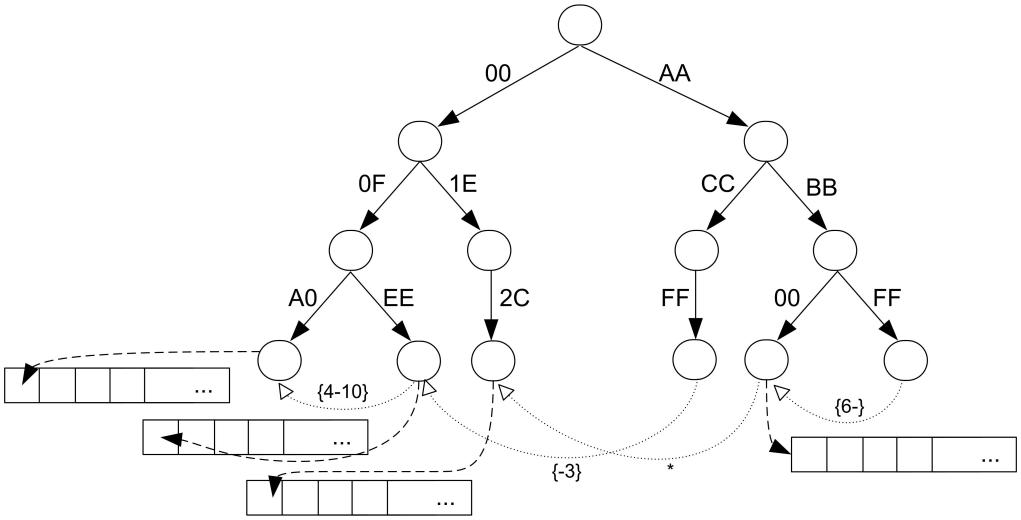


Figure 5.9: Our constraint-baseds Aho-Corasick automaton (failures not included) for the regular expressions 001E2C*AABB00{6-}AABBFF and 000FA0{4-10}000FEE{-3}AACCFF.

5.2.2 Towards An Efficient Compression Mechanism

As mentioned in Section 3.4.4, the majority of the ClamAV signature database (in total, 62,302 signatures, with an average length of 45 bytes and a maximum length of 804 bytes), which we have used for testing purposes of our approach, is represented (by an overwhelming percentage of 96.5%) by the last four types of constraints defined by us in Section 3.2.1. In order to implement the automaton, we are using the extended, constraint-based Aho-Corasick automaton described in Section 3.4.4, while as a storage model we are using the approach discussed in Section 3.4.3. In essence, the algorithm works by splitting regular expressions into their regular pattern parts and creating links between the different parts in the automaton. An outline for this approach is shown in Figure 5.9.

The basic idea for the compression mechanism we are proposing lies in the virus development process itself: many viruses, of related families, share similar portions of code, which eventually leads to their identification inside executable files based on highly similar (yet not entirely equivalent) signatures in virus databases. Based

on this idea, we are aiming to extract common subsequences of code from the set of virus signatures in the ClamAV database and *fragment the signatures* so that they are becoming regular expressions themselves. We are further encouraged to take this approach by the results obtained in GrAVity [VI10] (discussed in Section 3.3.3), where limiting the tree to a depth of only 8 characters had produced false positives at a very low rate of 0.0001%.

Since we used a very high number of signatures, we sorted them in lexicographical order of their characters and then we have applied the Smith-Waterman algorithm for identifying all common subsequences of code in the signatures, in groups of 1,000, progressively, up to the maximum tested. Assuming the common subsequence is C, and that the words in the tree are $P_1 = S_1CS_2$ and $P_2 = T_1CT_2$, the resulting modification will imply transforming the patterns into $P_1 = S_1\{0-0\}C\{0-0\}S_2$ and $P_2 = T_1\{0-0\}C\{0-0\}T_2$, so that C is stored only once in the automaton. We have used minimum subsequence lengths of 2 to 8, 10, 12, 16 and 32 and also two variations of the algorithm, one that performs splits starting with the longest common subsequences first (referenced from here onwards as *max*), while the other begins with the shortest (referenced from here onwards as *min*):

$start \leftarrow 0$, $LIMIT \leftarrow 1000$, $LENGTH \leftarrow$ {2,3,4,5,6,7,8,10,12,16,32}
$beginFrom \leftarrow 2, endAt \leftarrow maxKeywordLength$ (for min, reversed for max)
repeat
  $lowerLimit = start \times LIMIT$
  $upperLimit = lowerLimit \times LIMIT - 1$
  for i from $lowerLimit$ to $upperLimit$ **do**
    for j from $i + 1$ to $upperLimit$ **do**
      $pArray \leftarrow$ getAllCommonSubsequences(pattern[i], pattern[j])
      for k from 0 to pArray.count()-1 **do**
        if pArray[k].length() $> LENGTH$ **then**
          insertSubSeqInArray(pArray[k].length(), pArray[k], i, j)
          incrementOccurences(pArray[k])
        end if
      end for
    end for
  end for
  $start \leftarrow start + 1$
until all sequences processed
for i from $beginFrom$ to $endAt$ **do**
  for j from 0 to subArray[i].count()-1 **do**
    splitPattern(pattern[subArray[i][j].id1], subArray[i][j].subsequence)
    splitPattern(pattern[subArray[i][j].id2], subArray[i][j].subsequence)
  end for

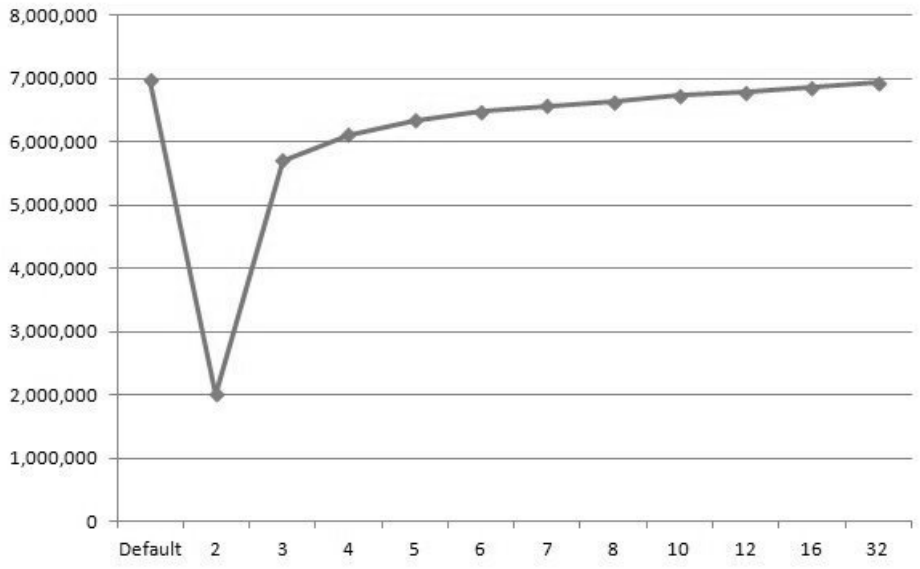


Figure 5.10: The total number of nodes obtained for our automaton, when using the ClamAV signatures and also when using our **min**-based common signature extraction approach. For **max**, a very similar distribution was obtained and was not included for preserving clarity.

end for

It is easily deductable that when splitting signatures by groups of two bytes, the automata will shrink in size considerably. This has been observed in the number of nodes we have obtained after performing our analysis, shown in Figure 5.10.

The density of children distribution obtained as a resulting of the signature fragmentation process presented above is shown in Figure 5.11. Table 5.4 shows the actual numbers for different **min** values considered.

Table 5.4: The number of nodes having from 0 up to 6 children in the constraint-based Aho-Corasick automaton [Pun13]

No. of children	0	1	2	3	4	5	6
Default	67,046	6,882,919	14,174	3,431	1,091	533	262
min=2	618,980	1,316,209	9,138	2,141	2,942	4,562	6,504
min=3	197,492	5,462,903	28,309	11,394	5,357	2,918	1,775
min=4	141,090	5,930,441	25,379	8,112	3,425	1,736	961
min=5	118,409	6,190,955	22,634	6,679	2,654	1,374	743

An empirical observation from Table 5.4 shows that the leaf nodes in the default tree are over 67,000 which is more than the number of signatures (62,302). The difference is explainable in that there are many signatures using regular expressions, which themselves contain different patterns that are inserted in the tree - therefore, a

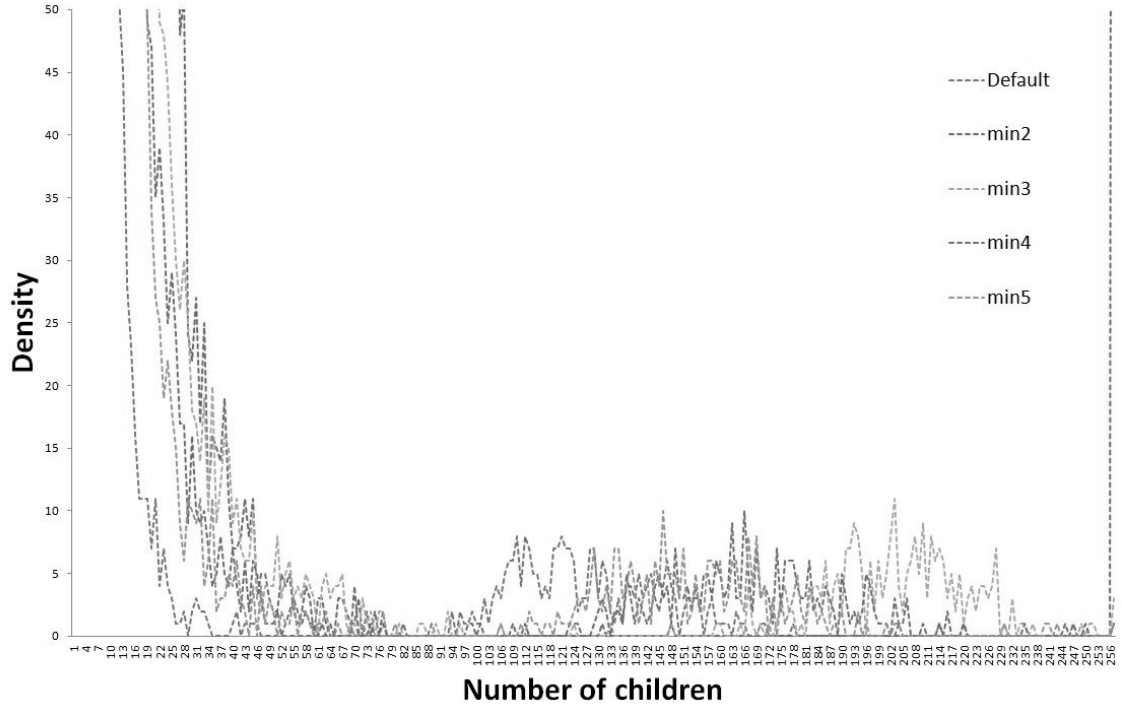


Figure 5.11: Children density distribution in the automaton built from the ClamAV database. Similar distributions (not shown) as for **min** were also obtained for **max**.

single regular expression pattern can produce more than one leaf node. Furthermore, when fragmenting the signatures using a minimum signature length of 2, the number of leaf nodes increases 9 times, while the number of single-childed nodes is reduced 5 times. It is interesting to observe that there are thousands of nodes having from 2 to 6 children, although not comparable to the previous numbers which are in the magnitude of millions. The high-number of single-childed nodes in the automaton shows that employing path-compression here could produce much lower memory usage for this automaton.

As also discussed in Section 3.4.4, the lists used for storing match positions are dynamically modified: when a match is found, if the match is part of a regular expression, a check is being made to determine if the previous subexpression node has been matched before, and if the constraints are being met (constraints of types $\{n\text{-}\}$ are the only ones that, when matched, permanently store the match position), store the match position at the current node in the list; if constraints are not met, remove all permitted previous matches.

We have implemented two different approaches to storing the information for each node: the first uses a sorted linked list with pointers to all children of the automaton (an approach also discussed in Section 6.2.4), which requires logarithmic matching time for verifying a transition using a *divide-et-impera* approach, and a second which uses bitmapped nodes (similar to the idea proposed in [ZS08, PN12]),

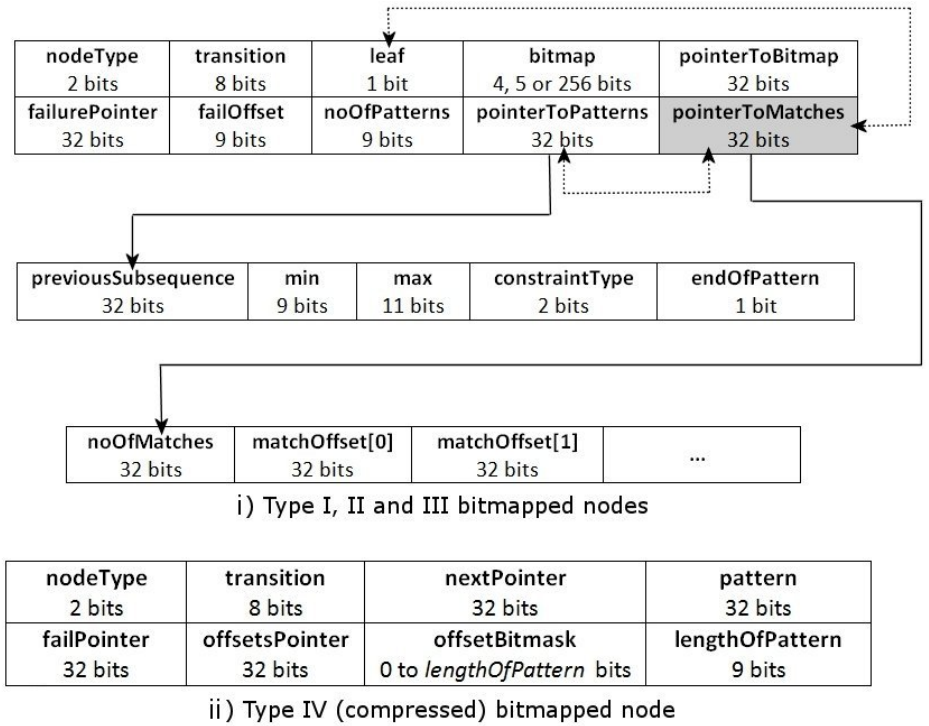


Figure 5.12: Children density distribution in the automaton built from the ClamAV database. Similar distributions (not shown) as for **min** were also obtained for **max**.

which uses slightly more memory but at the benefit of a throughput increase. Based on the empirical observations from Figure 5.11 and Table 5.4, for the bitmapped implementation we have built four different types of nodes: type I, for nodes having from 0 to 15 transitions, type II for nodes having from 16 to 48 transitions, type III, for generic bitmapped nodes and type IV for compressed nodes. In fact, in the logarithmic matching time implementation, to even further reduce memory usage, types I, II and III all have two sub-types: the first includes the *bitmapSequences, pointer* and *matches* fields (which only apply to nodes that are leaves in a subexpression), while the second sub-type does not have these fields (this only applies to nodes that are not leaves). In conclusion, the bitmapped implementation has 4 types of nodes, while the non-bitmapped implementation uses 7 types of nodes (4 primary types and 3 sub-types), as shown in Figure 5.12. Additionally, the *transition* field is used for nodes whose parent node is type I or II. The *leaf* field determines the presence of the *pointerToMatches* and *pointerToPatterns* fields. To avoid creating gaps in memory, type IV (bitmapped) nodes are followed in the actual stack by the pattern itself (as a normal sequence of bytes), after which the next node structure follows. Our implementation followed the data-parallel approach presented in Section 3.4.5.

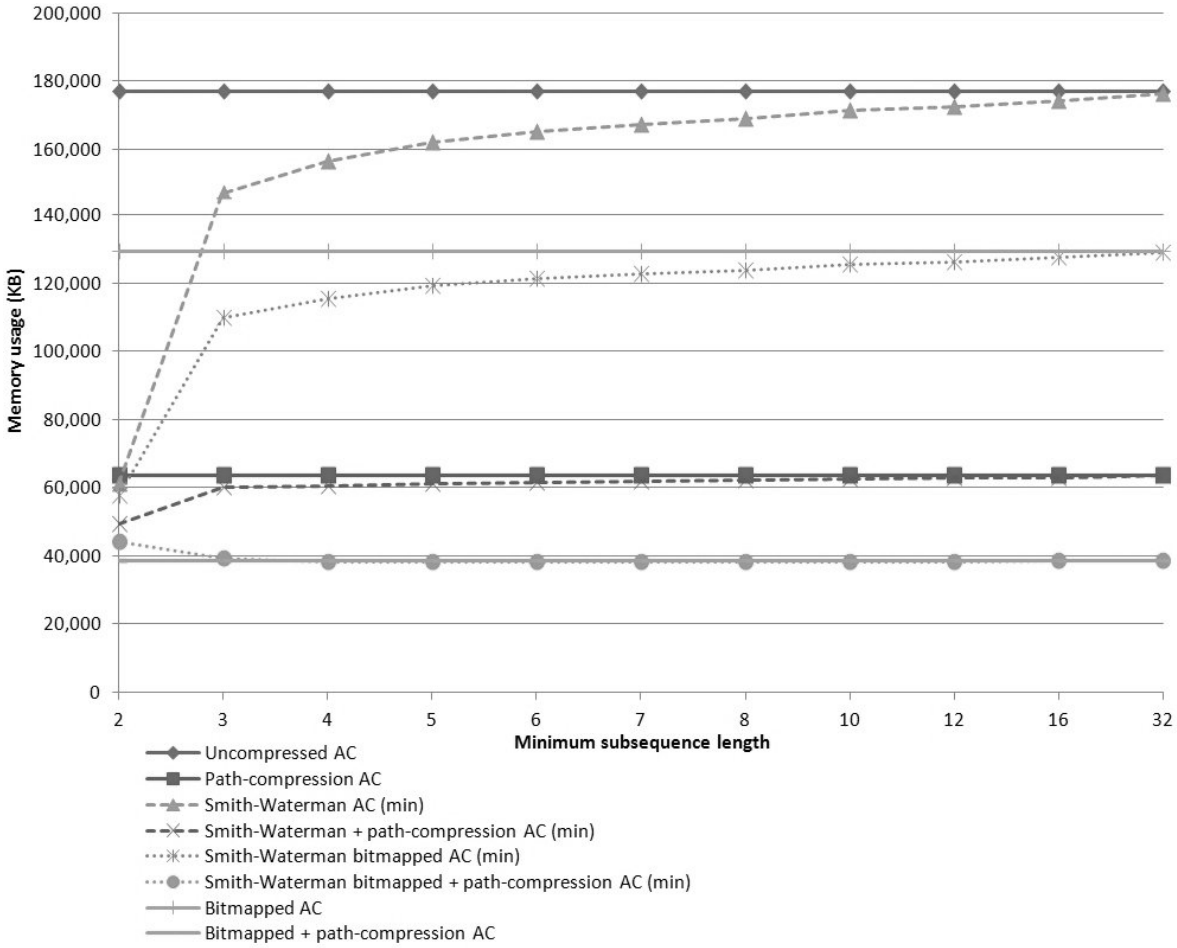


Figure 5.13: Memory usage for different implementations of the hybrid-compression mechanisms in the Aho-Corasick automaton.

5.2.3 Performance Evaluation

We tested the throughput of the automaton on a file comprised of 50 MB random binary data. We have applied different compression techniques in order to determine the most efficient, as follows: we tested the default (uncompressed) automaton against a bitmapped implementation, a path-compression implementation, an extended bitmapped implementation using path-compression, and a Smith-Waterman-based implementation featuring no other compression techniques, but also featuring, separately and combined, path-compression and bitmapped implementations. We have obtained various results which are depicted in Figure 5.13.

Applying path-compression to the ClamAV virus signature dataset used in the unoptimized automaton had determined a decrease in memory usage of more than 64%, while a bitmapped implementation (without path-compression) had decreased memory usage by 27%. A bitmapped implementation featuring path-compression re-

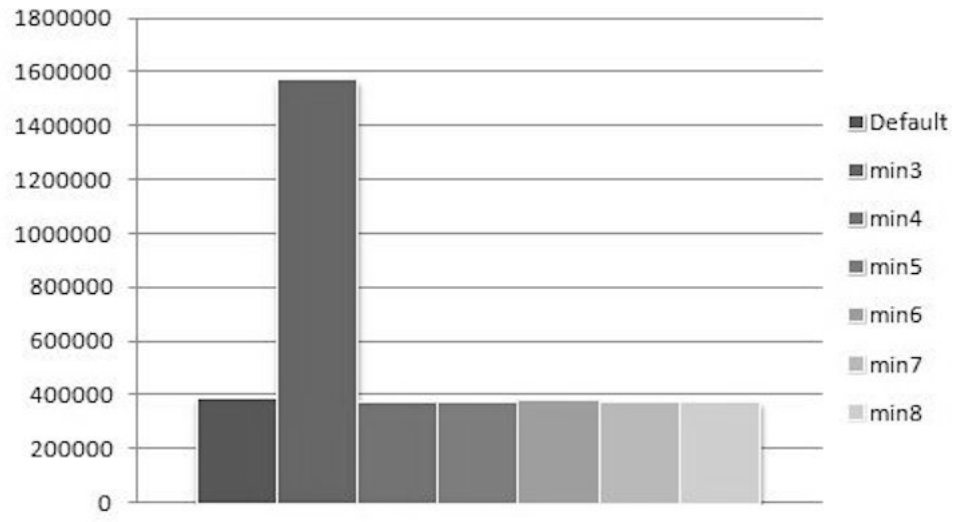


Figure 5.14: The number of *local hits* achieved at run-time.

duced the memory usage by more than 78%. Applying the Smith-Waterman-based algorithm we proposed, the compression ratios were even better, with the best occurring for a minimum subsequence of length 2; the primary drawback here was that the *local hit ratio* (the total number of hits in leaves of the automaton) had increased exponentially, which was to be expected given that the tree was highly dense at a depth of 2, making almost every possible two-byte combination a hit. The fragmentation approach we proposed becomes however efficient for a high number of patterns to be matched, with a large number of common subsequences of minimum lengths of 3 or 4, where the probability of a local hit decreases as the minimum length increases (Figure 5.14).

The *local hit* rate was very high when using a minimum subsequence length of 2, with the throughput decreasing to about 0.1 Mbps for the single-threaded implementation and 0.4 Mbps for the 8-threads implementation. For a minimum length of 3 (Figure 5.15), the throughput decreased by 25% compared to the standard implementation (although the local hit ratio was 4 times higher than that of the default implementation), while for consecutive lengths it was kept at the same level as the initial approach, because of lower local hit rates.

It is worth observing a close similarity between the Wu-Manber [WM94] hashing technique when using 2-bytes and our Smith-Waterman-based approach for a minimum subsequence length of 2. The Wu-Manber algorithm can be successfully used when using 2-bytes hashing for a small number of patterns only, while for a large number the authors suggest using 3-bytes hashing; a similar empirical observation can be made for our approach, since for a minimum subsequence length of 3 or more, the number of local hits had been reduced considerably and the run-time performance

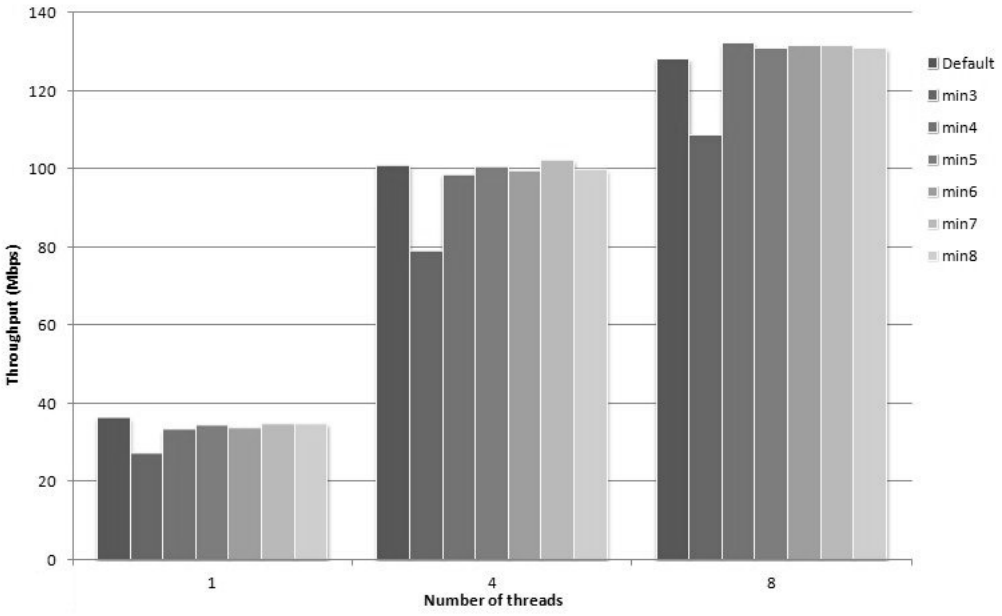


Figure 5.15: The throughput of the automaton when using our Smith-Waterman-based approach.

of the automaton had increased up to the level of the uncompressed implementation, although the memory usage had decreased considerably. An open-problem here still lies in the ability to quickly perform the Smith-Waterman decomposition of patterns, which is not a trivial problem and requires significant processing power.

In conclusion, we believe that our approach can be successfully applied to different pattern datasets wherever memory usage is a critical factor, assuming that the proper analysis is performed upfront on the dataset and the proper implementation decisions are taken.

5.3 Summary

This chapter had discussed two common problems in the Aho-Corasick pattern matching automaton: the first, presented in Section 5.1 deals with implementing behavioral heuristics in pattern-matching automata, and how such heuristics can be applied to malicious code detection in particular, and we have also proposed a new heuristic, based on a distance metric, for assessing the threat level of a program by analyzing the system-call sequences that the program contains; the second problem, discussed in Section 5.2 deals with the various approaches existing today for reducing memory usage in the Aho-Corasick automaton, and we have proposed (in Section 5.2.2) a new, hybrid-approach to achieving better compression by employing a dynamic-programming algorithm (presented in Section 5.2.1) and performing pattern decomposition.

The benefits of our approach, when it comes to performing malicious code detection, are outlined as follows:

- the algorithm proposed is very fast and has a throughput of several hundreds of thousands of system-calls per second, which is a lot more than any contemporary program would use;
- our approach does not require the code to be executed upfront, making it suitable for both static and dynamic (run-time) analysis in malicious code detection;
- our approach is, to the best of our knowledge, the first to use time-dependency in the heuristic employed;
- our approach is based on a metric which can be easily changed and adapted to the model desired.

In regards to the hybrid compression mechanism, our approach's particular characteristics are outlined below:

- our approach is the first to use the dynamic programming aspect for performing pattern decomposition, and the first to propose a model for efficiently storing the automaton using the new datasets of patterns obtained;
- our formalized model can be applied to any type of pattern, with any alphabet and of any size;
- our approach represents the first to analyze the impact of hybrid compression techniques on the virus signatures existing in ClamAV database, and obtain conclusive experimental results about the degree to which the memory efficiency can be improved in real-world implementations.

We believe that there is still room for much improvement here in both areas, as we have already pointed in the different sections throughout this chapter, therefore both the behavioral analysis and compression problems of the Aho-Corasick automaton remain open to debate. As a final thought, we are hoping that we have made small, yet significant steps towards improving the performance of the problems we tackled with throughout this chapter.

Chapter 6

Improving Carving Analysis in the Digital Forensics Process

Contents

The data recovery process is vital in the business environment and although safety measures can be employed to reduce the risks of data loss or corruption, sometimes emergent data can still be lost as a result of mechanical defect or faulty human intervention. In digital forensics, data recovery is often used to collect forensic evidence to be used by prosecutors against criminal suspects.

In this chapter we are going to discuss the challenges involving data recovery in the digital forensics process and propose and test a new and efficient method for identifying and collecting the metadata necessary to the human operator in the forensics analysis process.

6.1 Challenges of File-Carving in Digital Forensics

The data recovery process in digital forensics is based on the *file-carving* process. In general, the term *carving*, as also pointed out by Kloet in his Master's Thesis [Klo07], refers to extracting file from raw, unformatted and unordered binary data, based on the particular characteristics of that data. The extraction process does not use the filesystem information to reconstruct the file, just the information contained within. Mikus had pointed out in his Master's Thesis [Mik05] that disc carving is "an essential aspect of Computer Forensics".

Metz and Mora of Hoffman Investigations had formed one of the team competing in the Digital Forensics Research Workshop (DFRWS) in 2006 and responding to the challenge raised at that time, of building an efficient file carving algorithm that recovers more files and reduces the number of false positives. While the first attempts were mainly performed manually, which is not scalable to higher data sizes as Mora pointed out, in 2007 the challenge bar was raised to design completely automated tools that perform the same task. The team formed from Metz and Mora had initially made an empirical observation on the speed of a file-carver, stating it should proceed a theoretical 100 Gb of unallocated data daily, or about 1.16 MB per second, with tools handling less than half of that amount being considered unusable.

6.1.1 File Fragmentation

Databases (or *datasets*, as they are called in [Klo07]) are clones, bit-by-bit, of the hard-disk drives (or any other type of storage device, for that matter) to be examined through the digital forensics process. The nature of such databases is *dynamic*, since data is continuously read and written from and to the device in time. One major challenge in the investigation process is represented by file fragmentation (Figure 6.2). A fragmented file is a file whose contents has been spread over several different, non-consecutive areas of the disk, in other words, a file which has, between the starting address of its contents and its ending address, one or more gaps which are empty or occupied by other files.

Kloet in [Klo07] classifies files as being of two types: with linear fragmentation or with non-linear fragmentation. In the case of linear fragmentation, the parts of the file are present in the database in their original order, while in the non-linear fragmentation the order is different. Modern filesystems contain a file allocation table, where information about each file and directory structure is usually kept (e.g. access time, creation time, size, access rights, etc.). Whenever deleting a file, it would be impractical (and slow, for larger files) to erase its contents on disk byte-by-byte, which is why the only operation usually performed by a filesystem is the removal of the file information from the file allocation table. While this approach is fast, it has two potential effects: the first, the file could be recovered by restoring the entry in

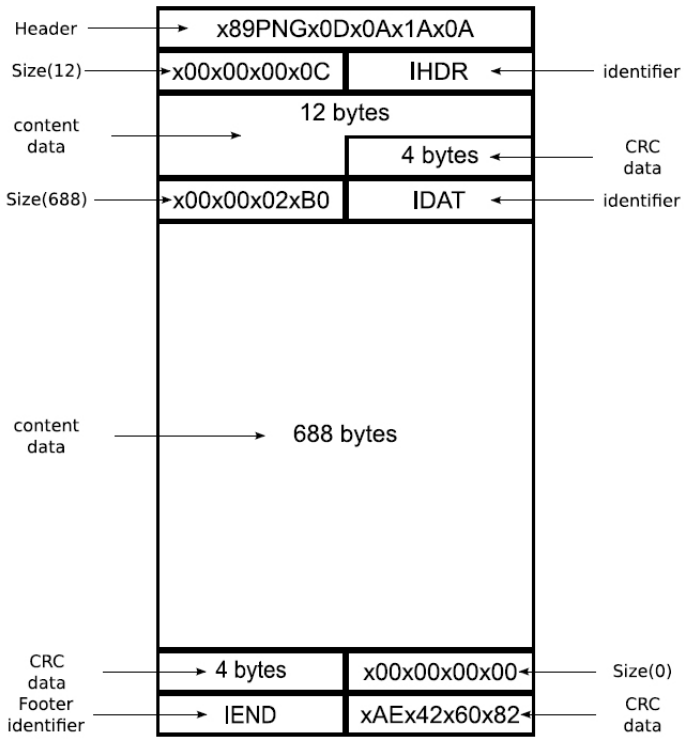


Figure 6.1: An example of the PNG image file header, footer and contents, as discussed by Kloet in [Klo07].

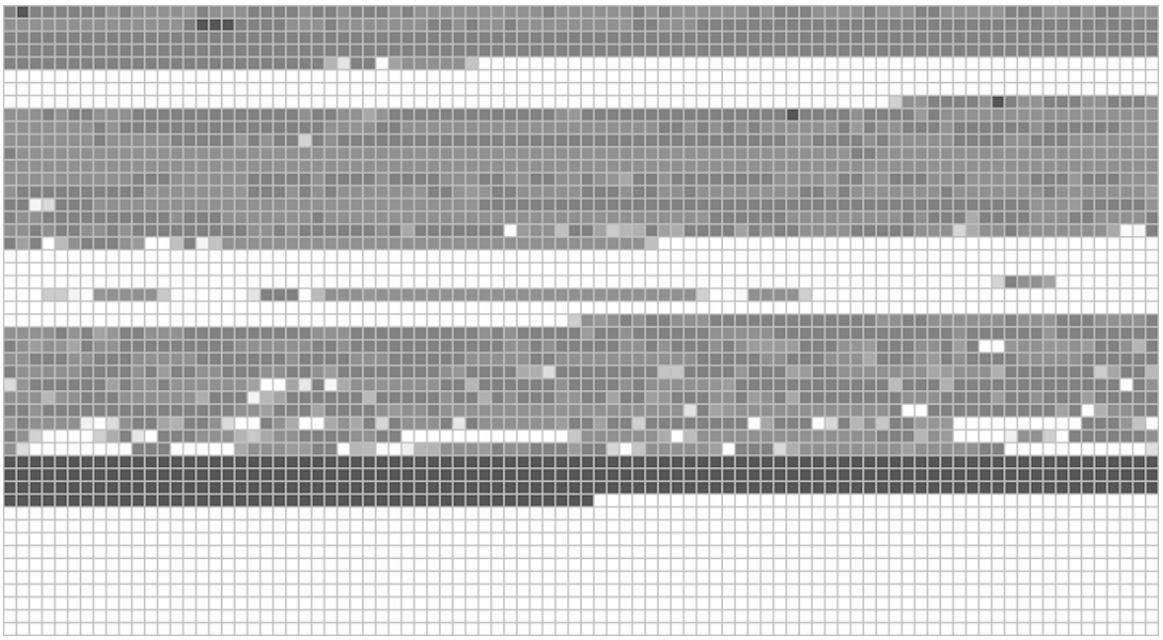


Figure 6.2: An example of how a disk can have fragmented files (disk was analyzed using our custom-built program). Each block represents a sector on the disk.

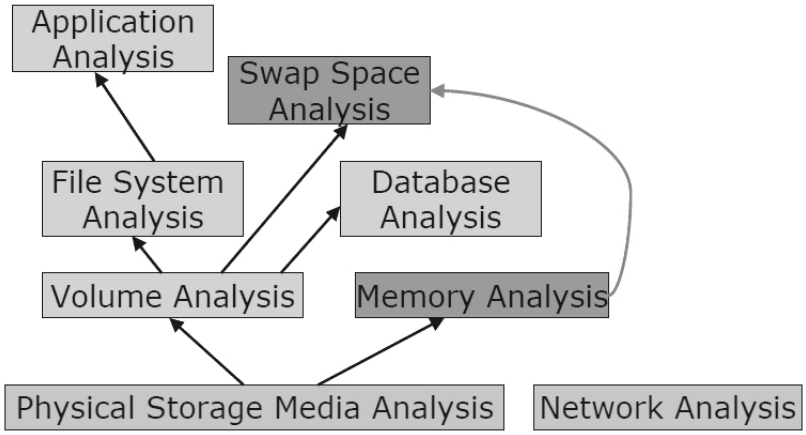


Figure 6.3: Different forensic analysis types as proposed by Carrier in [Car05].

the file allocation table, and the second, the file's contents (which is now marked as free space on the disk in the filesystem) may be overwritten by other files created in the meantime. Therefore, certain deleted files in this situation are only partially recoverable.

6.1.2 Forensic Analysis

The recovery techniques can vary, from *header & footer* based carving, to structural and content-based analysis. Each of these methods has it own disadvantages, which we will discuss as follows. Nevertheless, it is important to notice that the forensic analysis process is in fact another application of pattern matching algorithms, as it mainly involves locating pieces of the data of interest. Additionally, the results provided by the pattern matching algorithms may be used by different heuristic approaches to reconstruct the original file of interest to the investigator.

Carrier in [Car05] had proposed an organizational chart for the different types of forensic analysis, depicted in Figure 6.3.

6.1.2.1 Header and Footer Analysis

Analyzing the header and footer assumes that the forensic investigator has a precompiled list of known headers of different types of files (e.g. JPEG, PNG, TIFF images or other types of documents), along with a potential footer, which could identify that file accurately. The contents between the header and the footer would be relevant to the investigator because it could potentially be the actual contents of the file. Figure 6.1 shows the structure of a PNG image file.

However, for large files or for a highly fragmented storage device, additional data may be placed between the header and the footer. Furthermore, assuming the non-linear fragmentation pattern we have discussed above, it may happen that a file's

footer comes before the actual header, which could cause false positives in the forensic analysis or could even cause the entire process to fail.

The primary problem of this type of analysis is the low accuracy obtained in highly fragmented file systems. Kloet in [Klo07] states that another direct consequence of this limitation is that the file size of the recovered file is at least the same as that of the original, which is not entirely true: if a non-linear fragmentation pattern is employed for the file, and the footer is found before the header on the storage device, the recovery process would heavily rely on additional heuristics for reconstructing the original file, which may produce a smaller file or no file at all even, if insufficient data exists.

6.1.2.2 Structural Analysis

Structural analysis starts from the concept of analyzing and identifying the structural elements of the file in question. Given for instance the PNG image structure in Figure 6.1, the structural analysis would take into consideration the header and footer location, along with the different locations where the identifier *IDAT* (which defines a size of a number of bytes in the file) appears and attempts to recover the same amount of data as the identifier specifies from that location. This would of course not be a perfect reconstruction, especially since the identifier size value may have already been rewritten by another file's contents, but it gives a good shot to the investigator to obtaining a partially valid content for that file.

By adding heuristics to the investigation process, an automated tool would be capable of *estimating* how the file can be reconstructed, and attempt its reconstruction. If the estimation fails, the file resulted would be invalid or incomplete, which would result partially recoverable content, and the file would become a *false positive*. If the heuristics would fail, the file would be discarded, leading to a lost file in the database, or a *false negative*.

This method works best when there is sufficient information about a file's structure which could be carved and therefore reconstructed. In essence, structural analysis builds a map of the original file by putting together many pieces of the puzzle, and from that point onwards it employs diverse heuristics for *guessing* what would fit in the missing spots. The more information a puzzle contains, the easier it is to reconstruct the big picture. Keeping the analogy, the more quantitative information a portion of the file holds, the easier it is for the heuristics employed to reconstruct it with greater accuracy.

6.1.2.3 Storage-Based Analysis

Storage based analysis (also called *block content based carving* in [Klo07]) works best by analyzing the type of storage the database had been retrieved from. Different storage devices, such as hard-disk drives, SSDs, floppy disk drives, etc. use different block

sizes (e.g. a block size could have 512 bytes in early stage magnetic disks, or 4,096 bytes in the NTFS filesystem) to read and write information from and to. Information is mostly written in sectors (e.g. blocks of 512 bytes for some hard-disks), meaning that over a certain number of blocks in some storage devices, data is guaranteed not be fragmented in any way. This type of analysis performs a statistical analysis on the bytes of the block and employs different heuristics to determine whether they are relevant to the forensic investigation or not.

A common approach to performing statistical evaluation in storage-based analysis is based on *entropy computation*. The *entropy*, according to [Wika], refers to the average number of bits used to encode a symbol - for instance, compressed or randomly generated data has much higher entropy than plain text data.

6.1.3 Quantitative Criteria for Measuring Quality

One of the first quantitative criteria for quality measurement was proposed by Manning in [MS99]. Manning defines the *recall* as the proportion of the items that the system selected, the *precision* as being the number of items that the system correctly identified, and a combined metric of the two for evaluating the overall performance, called the $F_{measure}$:

$$recall = \frac{p}{p + m_{neg}}, \tag{6.1}$$

where p is the number of matches being positives and m_{neg} the number of matches being false negatives.

$$precision = \frac{p}{p + m_{pos}}, \tag{6.2}$$

where m_{pos} the number of matches being false positives.

$$F_{measure} = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}, \tag{6.3}$$

where P is the precision, R is the recall and $\alpha \in [0, 1]$ (weight indicating preference for precision or recall). Based on the above, Kloet [Klo07] proposes a metric for measuring the qualitative performance and defines normative scores for the performance of file carving tools, which are presented in Table 6.1 and which are used to evaluate several such tools.

Hoffman and Metz have proposed the initial empiric metric discussed in Section 6.1 for measuring the speed and performance as follows:

Kloet [Klo07] had also performed a benchmark of several file-carving tools, including Foremost, FTk, PhotoRec, Recover My Files and Scalpel. The results, based on similar metrics as those defined above, have shown that *Recover My Files* was the best performer, having a very good carving precision and almost perfect carving

Descriptive score range	Evaluation
$0 \leq$ score < 0.5	Bad
$0.5 \leq$ score < 0.75	Mediocre
$0.75 \leq$ score < 0.85	Good
$0.85 \leq$ score < 0.95	Very good
$0.95 \leq$ score < 1.0	Almost perfect
score $= 1.0$	Perfect

Table 6.1: The qualitative scores as proposed by Kloet in [Klo07]

Throughput (MB/s)	Evaluation
bandwidth < 0.58	Not useful
$0.58 \leq$ bandwidth < 1.16	Useful in testing only
$1.16 \leq$ bandwidth	Useful in real-world environments

Table 6.2: The empiric evaluation of the carving speed, proposed by Hoffman and Metz

recall, followed closely by *Foremost* and *PhotoRec*. These last two programs use a combined methodology for performing header and footer analysis for some files, and structural analysis as well for other (limited) number of files. *FTk* and *Scalpel* were the worst performers in the field, mainly because the first does not know all the file types and is less efficient, while the latter only uses header and footer analysis when carving. The author also proposes *MultiCarver*, a file carver aiming to support multiple different types of carving in one single application (Figure 6.4).

6.1.4 Fighting Fragmentation

Nemon et al. in [BM06] have proposed a method to reassemble the fragments of an image using greedy algorithms. By constructing a weighted graph, they reduce the problem to a maximization or minimization problem, and propose three different approaches (pixel matching, sum of differences and median edge detector) using variants of greedy heuristics for achieving a rate of reconstruction of up to 86% for their tested datasets. An immediate disadvantage of this work is that the approach works for images mainly, and needs adaptation or rewriting for additional filetypes or formats.

A *sequential hypothesis* approach for solving the fragmentation problem was proposed by Pal et al. in [PSM08]. Their work focuses on determining the fragmentation point of a file by sequentially analyzing adjacent pairs of blocks in the file until the point of interest is reached. Their approach relies on an adaptation of Dijkstra's shortest path algorithm proposed in [Dij59], called *Parallel Unique Path (PUP)*, that can reassemble a file by analyzing the headers of a file, then performing a sequential analysis where the best match for each header is chosen and the best match is chosen

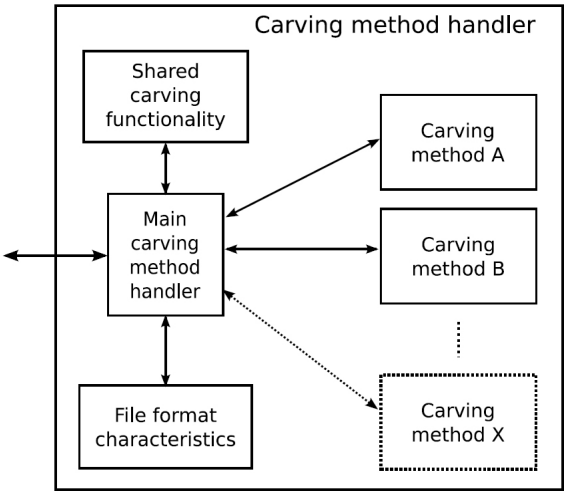


Figure 6.4: The Multi-Carver architecture as proposed by Kloet in [Klo07].

and merged with the header. The process finishes until the file is fully reconstructed. They propose a few additional heuristics which enhance the detection rate and reduce processing time by a few times. Thing et al. in [TCC11] have proposed an architecture for the detection and recovery of fragments for forensic purposes, where they were able to reduce the processing time to half of that of a known commercial solution.

6.2 Improving File-Carving Through Data-Parallel Header and Structural Analysis

In this section we are going to present the file-carving architecture proposed in [Pun12] for improving the structural analysis of files in digital forensic investigations.

6.2.1 Background

The motivation for improving file-carving begins with the history of carvers: one of the first file-carvers built was Foremost [For], which was the basis for the popular open-source alternative Scalpel [Ric05]. The Scalpel set of rules is described in the external file **scalpel.conf**, which allows the user to configure how the data is to be carved.

Scalpel (which may be downloaded from [Sca]) performs header and footer analysis, as discussed in Section 6.1.2.1, by specifying patterns for the header of the files it is capable of carving (as being unique sequences of bytes to be identified in the raw data contents, for instance <**html** for HTML documents) and also for the footer (for instance, <**html/**> for HTML documents). In order to solve the false positives issue which most file carvers can produce, an approach known as *in-file carving* had been

proposed by Richard et al. in [RI07]. Their approach generates sufficient metadata to allow a human expert operator to interfere later on and determine if the recovered contents is accurate or not. The slow performance of this methodology, which is still faster and uses less storage than other similar solutions, has been pointed out by Zha et al. in [Zha], where they have also proposed a different approach for speeding up the algorithm by employing the Aho-Corasick [AC75], Boyer-Moore [BM77] and Wu-Manber [WM94] algorithms in a multi-threaded implementation. As a result, the worst-case optimality of the Aho-Corasick automaton had produced significantly better results compared to the initial approaches.

One freeware solution that had existed since 2003 in the online community, and which is partially community-based, is the TrID pattern scanner [tri]. The software's purposes is identifying file types by analyzing their contents, and is complemented by the TrIDScan software, a module to create signatures for new file types, by analyzing multiple files belonging to the same type in groups, extracting relevant information (or metadata) particular to those file types (based on the properties of the files and on their contents) and compiling it into a definition to be used by the TrID scanner. The resulting signatures produced by the TrIDScan software are rewritten as XML-based sequences of bytes, that uniquely identify that type of file from that point onwards. Of course, the approach can be prone to false positive matching, assuming that:

- a small number of files was analyzed, with highly similar contents - in this case, two scenarios may happen: the program may produce long signatures which may also produce false positives if the extracted information is not relevant (for example, when dealing with text files with common and repetitive textual patterns in them), or may be so strict that the identification process would fail for other files of that type;

- a high number of files was analyzed, with a high degree of differences in their contents - in this situation, there is a high probability that the signatures would be invalid or would produce false positives, since it would be very difficult to extract fixed-length patterns of data which exists in both files.

Memory investigation has been of strong forensic interest in the past few years. With the potential ability to implement intrusion detection systems or detect perpetrators, this topic becomes more interesting, even more as it relies on the file-carving techniques presented in Section 6.1.2. Interesting approaches to this problem have been proposed by Hejazi et al. in [HDT08], where a methodology for an automated extraction of files from Windows memory had been proposed. The authors have been able to extract several filetypes using their method, which is similar to the file-carving technique although it is applied to the Windows memory layout, and once obtained they could be correlated with other sources of data found on disk. Butler

and Murdock have proposed in [BM] a method for reconstructing the process space in physical memory, by outlining EXE and DLL (Dynamic Link Library) file structures and performing structural analysis (described in Section 6.1.2.2) on the physical memory areas. They had also discussed the ability to extract files from the shared cache, extending the discussion to other file-types.

6.2.2 A Comparison Between Scalpel and the TrID Pattern Scanner

Scalpel's [Ric05] limitations are mostly present in the number of pattern signatures existing in its database. At the time of this writing, the number of definitions in the Scalpel configuration file was about 60, which is not only low, but lacks the ability to detect other file types. While Scalpel focuses mainly on the ability to detect the most common file types known today, by employing multiple pattern matching, a forensics investigator often could use additional document types in a thorough research (e.g. spreadsheet documents, compressed archives, videos of different formats and encodings, etc.). Another important limitation of Scalpel is related to the way the program scans for files - currently, the program supports only the ability to define raw header and footer contents, as mentioned in Section 6.1.2.1, and bases its matching strategies on the premise that they can both be successfully identified. However, as pointed out in Sections 6.1.2.2 and 6.1.2.3, this type of strategy is not sufficient and could fail in many scenarios and situations, because it does not take into account any structural features of the filetype, or any relevant metadata existing in the file, which may later on be used for identifying pieces of the file throughout the storage device (if the file is, for instance, fragmented). A direct consequence is that the post-analysis process involved in digital forensics is only going to be lengthier and more difficult, with additional human operator intervention required.

The TrID pattern scanner has had a long history since it had appeared online. Although it is not open-source, it is freeware (available for both the Windows and Linux operating systems) and, most importantly, it offers a community-based definitions database, where people can contribute whenever they have a new file-type to add. This does not make TrID suitable for direct file-carving, since the tool had not been built for such a purpose and it cannot read through raw data like a file-carver does, but given that its database has grown considerably ever since its appearance in 2003, to over 4,800 patterns for known filetypes along with their definitions, this denotes a high-degree of interest in this tool from the online community which could, ultimately, lead to a beneficial withstand in the file-carving process.

6.2.3 Defining Signatures for File-Carving

The Scalpel configuration stored in **scalpel.conf** allows definitions in a simple form and also using simple wildcards, such as *?*. An example of a PNG image file taken from its database is listed below:

```
png y 2500000 \x89\x50\x4e\x47\x0d\x0a\x1a\x0a
```

The TrID pattern scanner uses an XML-based format for defining signatures, and has several areas in the definition. The first is the general information about the file, having two sections. The first section is the **Info** section:

```
<TrID ver="2.00">
    <Info>
        <FileType>Java Bytecode</FileType>
        <Ext>class</Ext>
        <ExtraInfo>
            <Rem></Rem>
            <RefURL></RefURL>
        </ExtraInfo>
        <User>Marco Pontello</User>
        <E-Mail>marcopon@nospam@gmail.com</E-Mail>
    </Info>
```

where information is being stored about the filetype, the type of extension it usually has and any extra information relevant to this file's type (e.g. URL), and also the user who had provided this file definition (in this case, the author of TrID). The next section is **General**, and contains information about the definition itself: the date when the definition was added and whether to check for internal strings (metadata) in the file once the header information is found.

```
    <General>
        <FileNum>80</FileNum>
        <CheckStrings>True</CheckStrings>
        <Date>
            <Year>2003</Year>
            <Month>11</Month>
            <Day>14</Day>
        </Date>
        <Time>
            <Hour>03</Hour>
            <Min>10</Min>
            <Sec>51</Sec>
        </Time>
    </General>
```

The actual definition being used by the TrID software lies in the **FrontBlock**, **Pattern** and **GlobalStrings** areas of the XML.

```
    <FrontBlock>
        <Pattern>
            <Bytes>CAFEBABE00</Bytes>
```

```
4             <Pos>0</Pos>
5         </Pattern>
6         <Pattern>
7             <Bytes>00</Bytes>
8             <Pos>6</Pos>
9         </Pattern>
10        <Pattern>
11            <Bytes>00</Bytes>
12            <Pos>11</Pos>
13        </Pattern>
14    </FrontBlock>
15    <GlobalStrings>
16        <String>CODE</String>
17        <String>INIT</String>
18        <String>JAVA</String>
19    </GlobalStrings>
```

In here, the software finds information about the header of the file (in the **FrontBlock** area), each of the unique patterns found in the header along with their relative position to the header (e.g. **Pattern** describes such a patterns, with the elements **Bytes** describing the actual sequence of bytes, for instance *CAFEBABE00*, and **Pos** describing the relative position to the header, in this case *0*, meaning the file's header begins with this text).

Currently, TrID does not seem to offer a way to define the footer of the file, but as we had already discussed earlier in Section 6.1.2 about the different fragmentation types, footers can be difficult to detect in non-linear fragmentation models.

There is however a downside to the community-based submission process implemented in TrID: certain submissions are false positives, which we have also probed in our implementation and experimental research using them. We have found that many patterns are completely unusable since they are too generic (the number of false positives was too high), or because they have been incorrectly submitted or analyzed (for instance, when performing the analysis on an insufficient number of sample files in the group).

6.2.4 Signatures as Regular Expressions

For our own approach as presented in [Pun12], we have written a XML parsing tool which processes the XML definitions of TrID and stores them in our internal format. There is a high-resemblance between the ClamAV [cla12] regular expression virus signatures and the TrID XML-based file-signature storage format. For a start, the same alphabet is used in both definitions for both programs, and all strings are expressed in the hexadecimal format as well.

The approach we have proposed in [Pun12] uses a similar background to the extended, constraint-based Aho-Corasick automaton from Section 3.4.4. As a reminder,

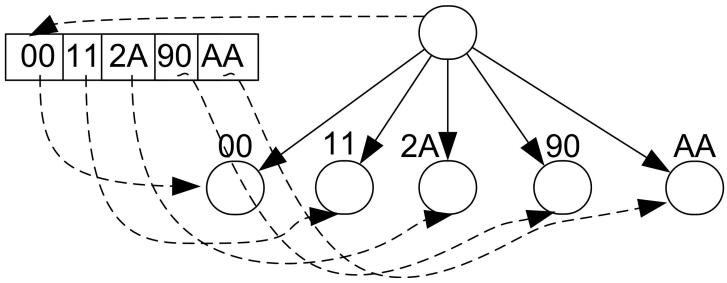


Figure 6.5: The sorted linked-list of children in a tree-based structure.

this automaton supports regular expressions with four types of constraints, most commonly found in the ClamAV definitions database as pointed out in Section 3.2.1. We have analyzed the TrID definitions database and have concluded that it is feasible to convert the signatures from the XML-based format they are currently in, in a new, regular-expression-based format similar to the four types of constraints discussed. For the implementation in [Pun12], we have used a different approach for storing the automata, that stores at each node a sorted linked list of its children (Figure 6.5). This way, locating an element is equivalent to the problem of locating an item in a sorted list, of complexity $O(log_2(N))$, where N is the number of elements in the list (in our case, the number of children of a node).

We have written a parser for converting the XML-based signatures into a regular expression format similar to that used in the ClamAV signature definitions in Section 3.2.1, by combining all **Pattern** elements into a single expression, as follows:

- We start with an empty regular expression.
- For each **Pattern** element of index i, we store the **Bytes** element (of length l_i) and the **Pos** value, which represents the relative position in the file where this pattern should occur, as in $p^i_{pos_i}$, and concatenate it to the final expression.
- For the pattern at index $i+1$, the final expression would be concatenated with the previous pattern as follows: $p^i_{pos_i} \mid p^{i+1}_{pos_{i+1}-l_i}$
- The final regular expression would represent a concatenation of all patterns found in the XML description of the TrID signature database.

For example, the patterns below:

```
1     <FrontBlock>
2         <Pattern>
3             <Bytes>4D4D</Bytes>
4             <Pos>0</Pos>
5         </Pattern>
6         <Pattern>
7             <Bytes>0002000A000000030000003D3D</Bytes>
```

```
            <Pos>5</Pos>
        </Pattern>
        <Pattern>
            <Bytes>00</Bytes>
            <Pos>21</Pos>
        </Pattern>
        <Pattern>
            <Bytes>0A000000</Bytes>
            <Pos>24</Pos>
        </Pattern>
        <Pattern>
            <Bytes>000000</Bytes>
            <Pos>29</Pos>
        </Pattern>
        <Pattern>
            <Bytes>000000</Bytes>
            <Pos>36</Pos>
        </Pattern>
    </FrontBlock>
    <GlobalStrings>
        <String>SOME</String>
        <String>MORE</String>
        <String>DATA</String>
    </GlobalStrings>
```

would be transformed into the regular expression:

```
4D4D{3-3}0002000A00000003000000 3D3D{3-3}00{2-2}0A000000
    {1-1}000000{4-4}000000*534F4D45*4D4F5245*44415441
```

In order to assemble the regular expression, the starting **4D4D** regular pattern is inserted, and its length is $l_0 = 2$. The next pattern, **0002000A00000003000000 3D3D**, begins at position $pos_1 = 5$, which means that it will be inserted after the previous pattern and a distance of $pos_1 - l_0 = 3$ must be kept, which is why the **{3-3}** expression had been inserted between the two, denoting that exactly any three characters must exist between the two patterns. The logic continues for the remaining patterns also. After the **FrontBlock** element has been fully processed, the remaining patterns in the **GlobalStrings** element are inserted (still as hexadecimal characters) using * separators to denote any number of characters between the header and this metadata. It is worth noticing that this approach assumes that the **GlobalStrings** patterns are mentioned in the logical order of their appearance, but the algorithm does not necessarily need to use the same logic when applying for matches, especially considering the potential file fragmentation in the filesystem. Instead, for the **GlobalStrings** patterns, it would suffice to mark their matching location in the automaton leaf nodes instead and use that information in an automated tool for reconstructing the file, or in order for it to be analyzed by a human operator supervising the process.

An immediate benefit of this advantage is the high scalability of other, commonly intrusion detection-related algorithms, to the problem of file-carving, with immediate

performance benefits and advantages. Another benefit is the reduction of storage space required for the patterns.

6.2.5 Extracting Relevant Patterns

After building the regular expressions and designing the mechanism for employing pattern matching using our modified version of the constraint-based Aho-Corasick algorithm, the final step before performing the experiments was the elimination of false positives from the TrID definitions database.

Using patterns of length 2, even if they are part of regular expressions, employed many false positives in our tests using a set of regular files on a Windows 7 disk, and even decreased the throughput performance of the automaton, since many local hits occurred at depth 2 in the tree. In order to avoid such situations, we have worked with a minimum pattern length of 3, and have conducted different experiments to determine how the pattern length would affect the number of definitions. We started with a set of 4,610 definitions and the number of valid signatures obtained after removing the false positives are shown in Tables 6.3 and 6.4. For a minimum length of 12, we observed that the same number of signatures was obtained when using a minimum length of 16 also, which is why our research stopped at this threshold.

Table 6.3: Number of patterns of lengths 3 to 7, extracted after pre-processing XML definitions

Pattern-length	*3*	*4*	*5*	*6*	*7*
Patterns extracted	3,052	2,838	2,031	1,864	1,666

Table 6.4: Number of patterns of lengths 8 to 12, extracted after pre-processing XML definitions

Pattern-length	*8*	*9*	*10*	*11*	*12*
Patterns extracted	1,556	1,263	1,129	1,049	992

6.2.6 Performance Evaluation

Our testing methodology used an intentionally damaged USB stick of 128 MB storage space, using a NTFS filesystem. We initially filled the disk with different filetypes of different sizes, after which the raw data of the disk was read and parsed by our processing algorithm and all partial and full matches were reported.

In order to accurately determine the performance of the modified constraint-based Aho-Corasick algorithm, we had tested it against the classic Aho-Corasick implementation, which does not support matching of regular expressions (instead, it provides all partial matches for all parts of a regular expression only). We have used a data-parallel approach similar to the one described in Section 3.4.5.

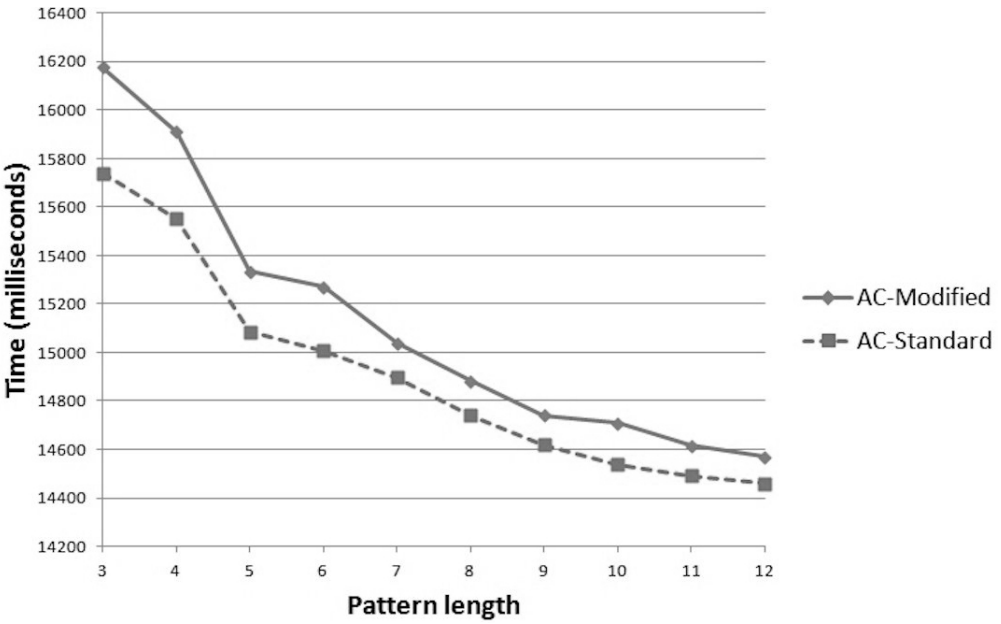


Figure 6.6: The results of the single-threaded file-carving process on a 128 MB USB flash disk using definitions from TrID and the classic and modified versions of the Aho-Corasick algorithm.

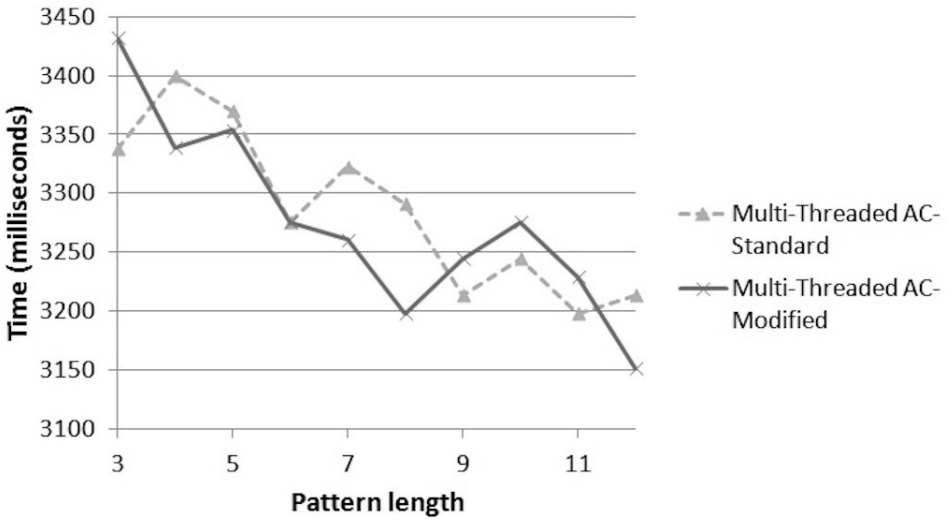


Figure 6.7: The results of the multi-threaded file-carving process on a 128 MB USB flash disk using definitions from TrID and the classic and modified versions of the Aho-Corasick algorithm.

Results in Figure 6.6 for the single-threaded implementation show that the modified version, although slower, behaved about 3% slower than the standard Aho-Corasick implementation. A multi-threaded, data-parallel implementation, featuring 8 threads was also tested and the results are shown in Figure 6.7. This experiment was less conclusive, given the low amount of data and the high throughput of the system used, however the same sequential approach shows that the performance degradation is small, given the low number of local hits at the leaves of the automaton.

In conclusion, the primary benefits of our proposed model are the following:

- extended support for both header and footer analysis and structural analysis, as part of the same carving process;
- support for many file types (over 4,600 in our experiments, with the ability to add many more);
- easier metadata collection by enhancing the partial results and storing their location, building a map of all the logical pieces of a file;
- partial-carving of files, by supporting only partial contents through regular-expression-formatted definitions;
- partially fighting file fragmentation (although not supporting corrupted files yet) by building a logical map of a file's structural contents through the analysis of all local hits in the Aho-Corasick automaton.

6.3 Summary

This chapter had discussed some of the most common challenges involved when it comes to improving the file-carving process in digital forensics analysis, proposing different classifications for the different analysis types and showing how they can be accomplished. We have also proposed a new model for improving the file-carving process, which bases its grounds on a popular, freeware software program which has a permanently growing database for identifying the different filetypes, outlining its benefits and immediate application to the process of file-carving.

The general outline of this chapter is presented as follows:

- In Section 6.1 we have discussed about file fragmentation and how it affects the carving process, outlining different fragmentation types and outlining the particular characteristics for each of them.

- We moved on to classifying the different types of forensic analysis and discussing a few proposed solutions in literature throughout recent years in Section 6.1.2, also discussing some quantitative criteria for measuring quality and how they can be applied to file-carving procedures and methodologies proposed so far (Section 6.1.3).

- We have discussed some common file-carvers existing today and outlined the advantages and disadvantages of each.

- We have proposed a new model for employing file-carving using header and structural analysis, by constructing regular expressions similar in format to that used in Section 3.2.1 for the ClamAV virus signatures, and have written a parser for building the expressions from the database of a freeware, multi-platform software for identifying filetypes, called TrIDScan.

- We have outlined the performance and benefits of using our approach, by showing how an efficient file-carver performing both header and structural analysis can be successfully used by the human operator in the post-analysis process for an efficient reconstruction of the original file.

The summary of the work and model proposed in our approach shows that performing header, structural and storage-based analysis as part of the carving process is vital to the post-analysis process, where the human operator is usually employed or where automated tools using different heuristics are reconstructing the original file's contents by assembling the pieces in the data in their logical order. Our proposed model uses thousands of signatures compared to tens used in equivalent file-carvers as Scalpel [Ric05], therefore offering higher flexibility. Furthermore, it shows that the model can be successfully used to perform both single-threaded and multi-threaded data-parallel pattern matching, without significant performance degradation, offering a scalable approach to extending our research to other platforms and/or hardware devices.

Chapter 7

Conclusions

Contents

Given the highly dynamic nature of the cyberspace today and the continuously growing number of threats that are released, with the newest ones targeting the emerging mobile platforms, such as Android, MacOS or iOS, the need for fast and efficient malicious code detection engines is becoming more stringent than ever, especially considering the very limited computational and storage requirements imposed by such platforms. The plural nature of pattern matching algorithms and their high applicability to various degrees in numerous fields of research has determined the appearance of significant improvements for the classic algorithms known, with some of the latest approaches looking to benefit from GPU acceleration in modern systems and others aiming to develop hybrid solutions that offer a good compromise between storage space and run-time performance.

As a highly active computer security research area, data forensics is still a widely open subject to innovations, where improvements and performance optimizations could still be found for increasing the reliability, feasibility and integrity of the existing approaches known today. Its high applicability to cyber-crime prevention, either it being for tracking down vital clues or pieces of information in crime-related trials, or for recovering lost, yet highly valuable business data for medium or large enterprises, shows that this chapter is far from being closed in the coming years and that contributions here could help improve the way we look at the world in the near future. The divergent applicability of pattern matching algorithms in this area has shown numerous improvements in recent years, with breakthroughs that simplify the work of the human operators, increase the processing speed of the processes involved and help assess important and relevant information easier than ever.

7.1 Discussions

At the beginning of this thesis we outline related work in the field of pattern-matching, starting with the area of single-pattern matching (and presenting the most important contributions here) and ending with multiple-pattern matching (with special emphasis on the Aho-Corasick, Commentz-Walter and Wu-Manber implementations, the most widely used today in most pattern-matching applications). We then move on to present the field of intrusion detection systems, presenting a classification of generic intrusion detection systems and outlining ways to detect malicious code behavior. The state of the art ends by presenting the recently emerging field of digital forensics, outlining its applicability in cyber-crime investigations and data forensics.

The next part of the thesis discusses our own contribution to improving the storage efficiency in heterogeneous hardware platforms, such as those relying on hybrid CPU/GPU computations, by presenting a model used to significantly reduce the storage requirements for very large pattern-matching automata and discussing how the model may be successfully be applied in the virus signature matching process. We outline the primary benefits of the approach and show the overall improvement compared to other existing methodologies at this time.

We continue our dissertation by showing how the previous efficient storage model may be used for the very fast, highly-parallel and throughput-optimized construction of very large pattern-matching automata in any type of applications, with specific focus on intrusion detection systems and, particularly, malicious code detection. We also present a few mathematical proofs for showing how the number of computations may be dramatically reduced, and apply the model to our approach, showing the great impact on performance that this has in hybrid-parallel implementations and discussing the primary benefits of the approach. We observe that our approach may also be applied without any restrictions to any type of pattern-matching applications using high amounts of data which require highly intensive computational tasks to be performed as part of the construction stage.

The next topic the thesis tackles with is focusing on implementing behavioral heuristics in large pattern-matching automata, and we propose a new heuristics, based on a weighted automaton, which is taking into account the moment of time of a system-call during program execution for classifying malicious code behavior. We also discuss for mathematical formalism behind the approach, outlining the very good performance results obtained in real-world scenarios, and concluding that the approach is suitable for both static (offline) and dynamic (online, at run-time) analysis of the code.

Another contribution we have made is discussed next, outlining a hybrid compression approach of large pattern-matching automata through the means of dynamic programming. Our focus was to reduce the storage required for the automaton by compacting the tree sufficiently, by converting regular patterns into regular-expressions

and using our own constraint-based model for matching them. We discuss the performance evaluation of the algorithm and show that for sufficiently-sized patterns, the performance suffers almost no impact in the approach, allowing much better storage efficiency than the default naive implementation.

The final contribution of this thesis is outlined last, showing how we can adapt the constraint-based regular-expression matching automata to the problem of file-carving, focusing in particular on the header and structural analysis process as part of data forensics. We discuss the challenges involved in resolving this problem, outlining the major problems posed by file fragmentation on modern file-systems, also showing how we can improve the file-carving process by converting a popular, open-database of file-types to a regular-expression format used by our implementation. Furthermore, we outline the potential future benefits of our approach and conclude that the performance impact of the implementation is very small, compared to the initial algorithm.

The final chapter of the thesis outlines its main contributions and future research topics which have emerged as a result of the work and results presented herein.

7.2 Contributions

The main contributions of this thesis are the following:

- We have built a highly-efficient storage mechanism (outlined in Chapter 3) for hybrid CPU/GPU memory storage, with the following characteristics:
  - it can be easily constructed by parsing the tree associated to the automaton;
  - it stores the entire automaton in both host and/or device memory while still allowing it to be traversed entirely;
  - it uses a single pointer to external memory, with all other child pointers being offsets to different areas in the internal memory;
  - it offers the ability to transfer data back and forth (full-duplex) between the host and the device at maximum throughput allowed by the PCI-Express architecture;
  - it imposes no limitation on the ordering of nodes, with the single exception that all children of a node must be stored in consecutive memory locations in the layout;
  - it always employs constant storage space, an improvement over other previous similar approaches discussed.

- We have proposed, formalized and implemented (in Chapter 4) the first hybrid-parallel architecture for constructing very large automata through efficient CPU/GPU collaboration in heterogeneous hardware environments, as follows:
  - we have proposed and proved that the computation can be significantly reduced through the use of parallelism and a few mathematical properties of the resulting automata;
  - we have improved the performance significantly as a result of our formalism, dropping the processing time from the magnitude of a few hours to just under a few seconds, on high-end consumer hardware;
  - our approach may be successfully used in any type of pattern-matching applications involving very large automata;
  - the approach uses our own-proposed highly-compact storage format described earlier, which allows access to nodes in a single serialized memory area;
  - it offers the first pre-processing performance results for the ClamAV database for both the Aho-Corasick and Commentz-Walter automata;
- We have proposed, to the best of our knowledge, the first time-dependant system-call analysis heuristic approach for detecting malicious code behavior (in Section 5.1):
  - the methodology uses the first known approach to tracing malicious code behavior by analyzing system-calls, while also taking into account the moment in time when the call has been made;
  - our approach has very fast performance (of several hundreds of thousands of system-calls being analyzed per second on average consumer hardware), which makes it suitable for both static and dynamic (at run-time) analysis, as well as implementation in real-time monitoring engines;
  - the approach does not necessarily require the code to be executed upfront;
  - the algorithm is based on a metric which may be easily adapted and changed to comply with various requirements during the implementation stage;
- We have proposed a hybrid compression mechanism for the Aho-Corasick automaton to be used in the virus signature matching process (Section 5.2):
  - our approach is the first to use a Smith-Waterman-based, dynamic programming approach to converting regular patterns into regular-expressions in order to create a more compact tree representation;

  - our formalized model may be applied to any other type of applications using pattern-matching automata, with any alphabet of any size;
  - we have presented the first real-world results of the impact of our hybrid compression approach over the ClamAV database of virus signatures, through conclusive experiments of the degree to which storage efficiency can be improved as a result of its application.

- We have proposed an innovative model and approach to use regular-expressions for performing carving analysis in the data forensics process, in Chapter 6:
  - our approach may be successfully used for performing both header and structural data analysis, as part of the carving process;
  - our implementation can support several tens or hundreds of thousands of file-types and structural definitions of files;
  - our model can output a structural map of the pieces of a file, as they are spread throughout the disk's physical layout;
  - partial-carving is also permitted, allowing fragments of files to be recovered easier through accurate regular-expression matching;
  - it supports heuristics on top of the implementation, for reconstructing the initial file by matching the found file pieces and restoring their logical order.

7.3 Future Research

Future research may be structured according to the chapters debated in this thesis, as follows:

- One possible improvement would be to design a methodology for storing the very-large pattern-matching automata discussed in Chapter 3 in GPU texture memory, which is faster since it has a cache mechanism behind. The approach may however limit the throughput performance if a large number of transfers are required between the host and the device, which makes the approach suitable for environments where cross-collaboration is minimal.

- Another possible future research topic is given by the specific *morphing* ability of the architecture presented in Chapter 4, where the pattern-matching automata can change their own structure dynamically in real-time. Such an approach could be successfully used in other algorithmic implementations, with potential applicability to different heuristics (behavioral ones, in particular), where structures change all the time as a result of the impact of one or more external factors.

- Another potential improvement would represent an extension or adaptation of the Bray-Curtis weighted automaton presented in Section 5.1 to more complex regular-expressions or other similarity measurements and comparing the results obtained. A careful development of work may be required if a preservation of the *system-call time dependency* is desired in the new implementation as well.

- Although widely discussed, compression of pattern-matching automata remains a challenging subject and it is our belief that there is still room for improvement in the topic. The approach we presented in Section 5.2 could be for instance extended to use a much-better, more efficient and parallel (by construction) approach to pre-processing the patterns and determining the longest common subsequences, which is the most laborious step of the work performed as part of that research. Although there have been studies on the topic, a heterogeneous hardware approach to solving the LCS problem for multiple patterns at once is still a potential subject to be researched further.

- The algorithmic backgrounds discussed in Chapter 6 for improving the carving analysis process could be extended further to accommodate hybrid-parallel implementations where one task is given the header analysis, another is given the structural analysis and a potentially new task is employed to apply different heuristics for merging file-fragments together into a single data piece. Additionally, heterogeneous implementations of structural analysis engines for data forensics is still an active subject of the currently on-going research in the field.

7.4 Final Thoughts

The future of computing lies in the hybrid, heterogeneous systems, where the integrated computational devices, such as the CPU and the GPU, work together flawlessly for the greater benefit of the end-user and for highly improved processing power. Improved nano-technology and improved electrical and chemical properties of semiconductors make this scenario a closer reality with each passing day, making research in this topic a viable source of interest for developing, building and improving on the existing architectures and algorithms, either by further parallelizing those known so far or by constructing new ones that benefit from the advantages offered by the new platforms.

In conclusion, we argue that in this thesis we have identified and solved some of the pattern matching challenges commonly found nowadays in some of the most challenging fields of research in computer security, particularly related to the malicious code detection and digital data forensics problems and especially concerning heterogeneous hardware platforms making use of hybrid implementations. Given the fast changing pace of the world, it is our belief that the primary focus from here onwards

needs to carve deeper into building heterogeneous systems that are efficient in terms of performance and storage, which mixed together with the technological progress in the upcoming years would represent the foundational background of computational technology for the next couple of decades or more.

Bibliography

[AC75] A. V. Aho and M. J. Corasick, *Efficient string matching: an aid to bibliographic search*, Communications of the ACM, ACM Press, 1975, pp. 333–340.

[ALJ+95] Debra Anderson, Teresa Lunt, Harold Javitz, Ann Tamaru, and Alfonso Valdes, *Next-generation Intrusion Detection Expert System (NIDES): A Summary*, 1995.

[AMD] *Amd firestream*, http://en.wikipedia.org/wiki/AMD_FireStream.

[AX11] Townend P. Arshad, J. and J. Xu, *A novel intrusion severity analysis approach for clouds*, Future Generation Computer Systems, 2011, doi: 10.1016/j.future.2011.08.009.

[Axe00] S. Axelsson, *Intrusion detection systems: A survey and taxonomy*, Technical Report 99-15, Chalmers University of Technology, Dept. of Computer Engineering, Goteborg, Sweden, 2000, http://www.ce.chalmers.se/staff/sax/taxonomy.ps.

[Ben11] M. et al. Benn, *Digital forensics and data leakage protection*, https://www.bewglobal.com/pdfs/digital-forensics-data-leakage.pdf.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang, *On the (im)possibility of obfuscating programs*, J. ACM **59** (2012), no. 2, 6:1–6:48.

[bio] *Bioinformatics explained: Smith-waterman*, http://www.clcbio.com/sciencearticles/Smith-Waterman.pdf.

[Blo70] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Commun. ACM **13** (1970), no. 7, 422–426.

[BM] J. Butler and J. Murdock, *Physical memory forensics for files and cache*, BlackHat.com, http://media.blackhat.com/bh-us-11/Butler/BH_US_11_ButlerMurdock_Physical_Memory_Forensics-WP.pdf.

[BM77] R.S. Boyer and J.S. Moore, *A fast string searching algorithm*, Communications of the ACM, 1977, ISSN 0001-0782, pp. 762–772.

[BM06] J. Butler and J. Murdock, *Automated reassembly of file fragmented images using greedy algorithms*, IEEE Transactions on Image Processing, 2006, ISSN 1057-7149, pp. 385 – 393.

[Bra] *Kardi teknomo's page: Bray curtis distance*, http://people.revoledu.com/kardi/tutorial/Similarity/BrayCurtisDistance.html.

[Bro93] A.Z. Broder, *Some applications of rabin's fingerprinting method*, Sequences II: Methods in Communications, Security, and Computer Science, Springer-Verlag, 1993, pp. 143–152.

[BS11] R. Bhukya and D.V.L.N. Somayajulu, *Exact multiple pattern matching algorithm using dna sequence and pattern pair*, International Journal of Computer Applications, 2011, pp. 32–38.

[Car02] Brian Carrier, *Defining digital forensic examination and analysis tools*, International Journal of Digital Evidence **1** (2002), 2003.

[Car05] B. Carrier, *File system forensic analysis*, Addison-Wesley Professional, 2005, ISBN 0321268172.

[CF97] Cristina Cifuentes and Antoine Fraboulet, *Intraprocedural static slicing of binary executables*, Proceedings of the International Conference on Software Maintenance (Washington, DC, USA), ICSM '97, IEEE Computer Society, 1997, pp. 188–.

[CH11] Emilio Corchado and Álvaro Herrero, *Neural visualization of network traffic data for intrusion detection*, Appl. Soft Comput. **11** (2011), no. 2, 2042–2056.

[cla12] *Clamav antivirus*, 2012, http://www.clamav.net/lang/en/.

[CMJ+10] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen, *Splitscreen: enabling efficient, distributed malware detection*, Proceedings of the 7th USENIX conference on Networked systems design and implementation (Berkeley, CA, USA), NSDI'10, USENIX Association, 2010, pp. 25–25.

[Coh87] F. Cohen, *Computer viruses: theory and experiments*, Comput. Secur. **6** (1987), no. 1, 22–35.

[Col91] R. Cole, *Tight bounds on the complexity of the boyer-moore string matching algorithm*, Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 1991.

[Cor02] Leiserson C.E. Rivest R.I Stein C. Cormen, T.H., *Introduction to algorithms, second edition*, The MIT Press and McGraw-Hill Book Company, 2002, ISBN 978-0262032933.

[CR94] Czumaj A. Gasieniec L. Jarominek S. Lecroq T. Plandowski W. Crochemore, M. and W. Rytter, *Speeding up two string matching algorithms*, Algorithmica, 1994, pp. 247–267.

[Cre] *Creating signatures for clamav*, http://www.clamav.net/doc/latest/signatures.pdf.

[CUDa] *The cuda occupancy calculator*, http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[CUDb] *The cuda thread model*, http://pg-server.csc.ncsu.edu/mediawiki/index.php/CSC/ECE_506_Spring_2011/ch2a_mc.

[CUDc] *nvidia cuda c programming guide*, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[cud12] *The cuda parallel computing platform*, 2012, http://www.nvidia.com/object/cuda-parallel-computing-platform.html.

[CW79] B. Commentz-Walter, *A string matching algorithm fast on the average*, Proceedings of the 6th Colloquium, on Automata, Languages and Programming, Springer-Verlag, 1979, ISBN 3-540-09510-1, pp. 118–132.

[CW00] D.M. Chess and R.S. White, *An undetectable computer virus*.

[CWL07] Mohamed R. Chouchane, Andrew Walenstein, and Arun Lakhotia, *Statistical signatures for fast filtering of instruction-substituting metamorphic malware*, Proceedings of the 2007 ACM workshop on Recurring malcode (New York, NY, USA), WORM '07, ACM, 2007, pp. 31–37.

[CY06] Hongke Z. Chunyue, Z. and L. Yun, *A pattern matching based network intrusion detection system*, 9th International Conference on Control, Automation, Robotics and Vision (ICARCV), IEEE Computer Society, 2006, ISBN 1-4244-0341-3, pp. 1–4.

[cyb] *Cybercrime*, http://www.interpol.int/Crime-areas/Cybercrime/Cybercrime.

[CZ04] Watson B.W. Cleophas, L. and G. Zwaan, *A new taxonomy of sublinear keyword pattern matching algorithms*, Technical Report CS-TR 04-07, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2004.

[DH00] et al. Debar H., *A revised taxonomy for intrusion detection systems*, Annales des Telecommunications (vol. 55), 2000, pp. 361–378.

[DHW99] Dacier M. Debar H. and A. Wespi, *Towards a taxonomy of intrusion detection systems*, Computer Networks (vol. 31), 1999, pp. 805–822.

[Dij59] E.W. Dijsktra, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1959, pp. 269–271.

[Dir] *Microsoft directcompute*, http://developer.nvidia.com/cuda/directcompute.

[DPP07] Vassilis Dimopoulos, Ioannis Papaefstathiou, and Dionisios N. Pnevmatikatos, *A memory-efficient reconfigurable aho-corasick fsm implementation for intrusion detection systems*, ICSAMOS'07, 2007, pp. 186–193.

[Dud06] L. Dudas, *Improved pattern matching to find dna patterns*, IEEE International Conference on Automation, Quality and Testing, Robotics, IEEE Xplore, 2006, ISBN 1-4244-0360-X, pp. 345–349.

[ea11] Baker W. et al., *2011 data breach investigations report*, 2011, http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2011_en_xg.pdf.

[Esm95] Safavi-Naini R. Pieprzyk J. Esmaili, M., *Computer intrusion detection: A comparative survey*, Technical Report 95-07/06, Center for Computer Security Research, University of Wollongong, Wollongong, NSW, Australia, 1995.

[FC10] Kanonov-U. Elovici Y. Dolev S. Fledel, Y. and Chanan, *Google android: A comprehensive security assessment*, IEEE Security & Privacy, IEEE, 2010, ISSN 1540-7993.

[Fer] *nvidia fermi architecture*, http://www.nvidia.com/object/fermi-architecture.html.

[FKF+03] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong, *Anomaly detection using call stack information*, Proceedings of the 2003 IEEE Symposium on Security and Privacy (Washington, DC, USA), SP '03, IEEE Computer Society, 2003, pp. 62–.

[fla12] *Rt.com*, 2012, http://www.rt.com/news/flame-stuxnet-kaspersky-iran-607/.

[For] *Foremost*, http://foremost.sourceforge.net/.

[Fre06] Kimmo Fredriksson, *On-line approximate string matching in natural language*, Fundam. Inf. **72** (2006), no. 4, 453–466.

[Fre09] K. Frederiksson, *Succinct backward-dawg-matching*, Journal of Experimental Algorithmics (JEA), ACM, 2009, ISBN 1084-6654.

[GG11] G. Gigerenzer and W. Gaissmaier, *Heuristic decision making*, Annual Review of Psychology, vol. 62, 2011, pp. 451–482.

[GPG] *Gpgpu*, http://ro.wikipedia.org/wiki/GPGPU.

[Gra] *Control & flow dependence graphs*, http://www.grammatech.com/research/papers/staticAnalysis/imgSlides/sld021.html.

[HA] S. Holewinski and G. Andrzejewski.

[Har11] S.L. Harrington, *Collaborating with a digital forensics expert: ultimate tag-team or disastrous duo?*, William Mitchell Law Review **38** (2011), 353–396, ISSN 0270-272X.

[HDT08] Seyed Mahmood Hejazi, Mourad Debbabi, and Chamseddine Talhi, *Automated windows memory file extraction for cyber forensics investigation*, J. Digit. Forensic Pract. **2** (2008), no. 3, 117–131.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji, *Intrusion detection using sequences of system calls*, J. Comput. Secur. **6** (1998), no. 3, 151–180.

[HZB11] Amir Houmansadr, Saman A. Zonouz, and Robin Berthier, *A cloud-based intrusion detection and response system for mobile phones*, Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (Washington, DC, USA), DSNW '11, IEEE Computer Society, 2011, pp. 31–32.

[int12] *Hexus*, 2012, http://hexus.net/tech/news/cpu/39381-intel-currently-developing-14nm-aiming-towards-5nm-chips/.

[IoM] *Data recovery pricing*, http://iomega.com/data-recovery/datarecoveryfaq-prices.html.

[Jac99] K. Jackson, *Intrusion detection system (ids) product survey*, Technical Report LA-UR-99-3883, Los Alamos National Laboratory, Los Alamos, NM, 1999, http://libwww.lanl.gov/la-pubs/00416750.pdf.

[Kas] *Kaspersky lab utilizes nvidia technologies to enhance protection*, http://www.kaspersky.com/news?id=207575979.

[kKJLG03] kos Kiss, Judit Jsz, Gbor Lehotai, and Tibor Gyimthy, *Interprocedural static slicing of binary executables*, Source Code Analysis and Manipulation, IEEE, 2003, pp. 118–127.

[Klo07] S.J.J. Kloet, *Measuring and improving the quality of file carving methods*, Master's Thesis, 2007.

[KM11] Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis, *A performance evaluation of the preprocessing phase of multiple keyword matching algorithms*, Informatics, Panhellenic Conference on **0** (2011), 85–89.

[Knu77] Morris-J.H. Pratt V. Knuth, D., *Fast pattern matching in strings*, SIAM Journal on Computing, 1977, ISSN 0097-5397, pp. 323–350.

[Knu09] D.E. Knuth, *The art of computer programming volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams*, Addison-Wesley Professional, 2009.

[KR87] R.M. Karp and M.O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development, 1987, pp. 249–260.

[Lan92] William Landi, *Undecidability of static analysis*, ACM Lett. Program. Lang. Syst. **1** (1992), no. 4, 323–337.

[LC04] T. Lecroq and C. Charras, *Handbook of exact string matching algorithms*, College Publications, 2004, ISBN 978-0954300647.

[Lee07] T.H Lee, *Generalized aho-corasick algorithm for signature based anti-virus applications*, Proceedings of 16th International Conference on Computer Communications and Networks, IEEE Computer Society, 2007, ISBN 978-1-4244-1251-8, pp. 792–797.

[LH08] T.H. Lee and N.L. Huang, *An efficient and scalable pattern matching scheme for network security applications*, Proceedings of 17th International Conference on Computer Communications and Networks (ICCN), IEEE, 2008, pp. 1–7.

[LH12] Chang-S.C. Lin, C.H. and W.K. Hon, *Memory-efficient pattern matching architectures using perfect hashing on graphic processing units*, Proceedings of the INFOCOM, IEEE Computer Society, 2012, ISBN 978-1-4673-0773-4, pp. 1978–1986.

[Lin06] P.C. Lin, *Profiling and accelerating string matching algorithms in three network content security applications*, IEEE Communications Surveys & Tutorials vol. 8, issue 2, IEEE, 2006, pp. 24–37.

[Liu12] Chien-L.S. Chang S.C. Hon W.K. Liu, C.H., *Pfac library: Gpu-based string matching algorithm*, PU Technology Conference (GTC), 2012, http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0054-GTC2012-PFAC-GPU-Algorithm.pdf.

[LLL11] Po-Ching Lin, Ying-Dar Lin, and Yuan-Cheng Lai, *A hybrid algorithm of backward hashing and automaton tracking for virus scanning*, IEEE Trans. Comput. **60** (2011), no. 4, 594–601.

[LTL+10] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu, *Accelerating string matching using multi-threaded algorithm on gpu*, GLOBECOM'10, 2010, pp. 1–5.

[Lun88] T.F. Lunt, *Automated audit trail analysis and intrusion detection: A survey*, 11th National Computer Security Conference, Baltimore, MD, 1988, pp. 65–73.

[LYLT09] Yanbing Liu, Yifu Yang, Ping Liu, and Jianlong Tan, *A table compression method for extended aho-corasick automaton*, Proceedings of the 14th International Conference on Implementation and Application of Automata (Berlin, Heidelberg), CIAA '09, Springer-Verlag, 2009, pp. 84–93.

[Mala] *18.2-152.4:1 penalty for computer contamination*, Joint Commission on Technology and Science, http://jcots.state.va.us/2005 Content/pdf/-Computer Contamination Bill.pdf.

[Malb] *National conference of state legislatures virus/contaminant/destructive transmission statutes by state*, http://www.ncsl.org/programs/lis/cip/viruslaws.htm.

[MDWZ04] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok, *Avfs: an on-access anti-virus file system*, Proceedings of the 13th conference on USENIX Security Symposium - Volume 13 (Berkeley, CA, USA), SSYM'04, USENIX Association, 2004, pp. 6–6.

[MH90] Wolcott-D. Schaefer L. Kelem N. Hubbard B. McAuliffe, N. and T. Haley, *Is your computer being misused? a survey of current intrusion detection system technology*, Sixth Computer Security Applications Conference, 1990, pp. 260–272.

[mic12] *Microsoft safety & security center*, 2012, http://www.microsoft.com/security/pc-security/virus-whatis.aspx.

[Mik05] N.A. Mikus, *An analysis of disc carving techniques*, Naval Postgraduate School - Master's Thesis, 2005, http://handle.dtic.mil/100.2/ADA432468.

[MK06] Muthprasanna-M. Mansoor, A. and V. Kumar, *High speed pattern matching for network ids/ips*, Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols, IEEE Computer Society, 2006, ISBN 1-4244-0593-9, pp. 187–196.

[MP70] J.H. Morris and V.R. Pratt, *A linear pattern-matching algorithm*, Technical Report TR 40, University of California, Berkley, CA, 1970.

[MS99] Christopher D. Manning and Hinrich Schütze, *Foundations of statistical natural language processing*, MIT Press, Cambridge, MA, USA, 1999.

[NC10] Philips-A. Nelson, B. and Steuart C., *Guide to computer forensics and investigations*, Cengage Learning, 2010, ISBN 9781435498839.

[Nor04] M. Norton, *Optimizing pattern matching for intrusion detection*, SourceFire, Inc., 2004.

[nVI] *Cuda-powered anti-virus scanning is in development*, http://news.softpedia.com/news/CUDA-Powered-Anti-Virus-Scanning-Is-in-Development-123648.shtml.

[Oba09] Barrack Obama, *Remarks by the president on securing our nation's cyber infrastructure*, 2009, http://www.whitehouse.gov/the_press_office/Remarks-by-the-President-on-Securing-Our-Nations-Cyber-Infrastructure/.

[Ope] *Khrinis group: Opencl*, http://www.khronos.org/opencl/.

[PAF09] Ciprian. Pungila, Ovidiu. Aritoni, and Teodor-Florin. Fortis, *Benchmarking Database Systems for the Requirements of Sensor Readings*, IETE Technical Review **26** (2009), no. 5, 342–349.

[Pal01] G. Palmer, *A road map for digital forensic research*, DFRWS 16 (2001), http://www.dfrws.org/2001/dfrws-rm-final.pdf.

[PAP12] Mrutyunjaya Panda, Ajith Abraham, and Manas Ranjan Patra, *A hybrid intelligent approach for network intrusion detection*, Procedia Engineering **30** (2012), no. 0, 1 – 9, ¡ce:title¿International Conference on Communication Technology and System Design 2011¡/ce:title¿.

[pci] *Pci express*, http://en.wikipedia.org/wiki/PCI_Express.

[pe-] *pe-sig*, http://vrt-blog.snort.org/2009/03/generating-virus-signatures-automated.html.

[PN] C. Pungila and V. Negru, *Towards building efficient malware detection engines using hybrid cpu/gpu-accelerated approaches*, Architectures and Protocols for Secure Information Technology, IGI Global, (accepted to be published in 2013).

[PN97] Phillip A. Porras and Peter G. Neumann, *EMERALD: Event monitoring enabling responses to anomalous live disturbances*, Proceedings of the 20th National Information Systems Security Conference, 1997, pp. 353–365.

[PN12] C. Pungila and V. Negru, *A highly-efficient memory-compression approach for gpu-accelerated virus signature matching*, Proceedings of 8th International Information Security Conference (ISC), Lecture Nodes In Computer Science, 2012.

[Pri04] R. W. Price, *Roadmap to entrepreneurial success: Powerful strategies for building a high-profit business*, AMACOM, 2004, p. 42.

[PSM08] Anandabrata Pal, Husrev T. Sencar, and Nasir Memon, *Detecting file fragmentation point using sequential hypothesis testing*, Digit. Investig. **5** (2008), S2–S13.

[Pun09] Ciprian-Petrisor Pungila, *A bray-curtis weighted automaton for detecting malicious code through system-call analysis*, Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (Washington, DC, USA), SYNASC '09, IEEE Computer Society, 2009, pp. 392–400.

[Pun10] Ciprian Pungila, *A model for energy-efficient household maintenance through behavioral analysis of electrical appliances*, E-Business Engineering, IEEE International Conference on (2010), 409–414.

[Pun12] C. Pungila, *Improved file-carving through data-parallel pattern matching for data forensics*, 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2012, ISBN 978-1-4673-1013-0, pp. 197–202.

[Pun13] Ciprian Pungila, *Hybrid compression of the aho-corasick automaton for static analysis in intrusion detection systems*, International Joint Conference CISIS12-ICEUTE12-SOCO12 Special Sessions (lvaro Herrero, Vclav Snel, Ajith Abraham, Ivan Zelinka, Bruno Baruque, Hctor Quintin, Jos Luis Calvo, Javier Sedano, and Emilio Corchado, eds.), Advances in Intelligent Systems and Computing, vol. 189, Springer Berlin Heidelberg, 2013, 10.1007/978-3-642-33018-6_8, pp. 77–86.

[Rab81] M.O. Rabin, *Fingerprinting by random polynomials*, Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[Rac] *Race condition*, http://en.wikipedia.org/wiki/Race_condition.

[Rah00] S.A. Rahman, *Network intrusion detection systems*, CS594-Computer and Network Security, 2000.

[red12] *Redorbit*, 2012, http://www.redorbit.com/news/technology/1112543496/flame-virus-most-sophisticated-cyber-weapon-ever-used/.

[Rei02] Carr-C. Gunsch G. Reith, M., *An examination of digital forensic models*, International Journal of Digital Evidence, 2002.

[RI07] Roussev V. Marziale L. Richard III, G., *In-place file carving*, Science Direct, 2007.

[Ric05] Roussev V. Richard, G. III, *Scalpel: A frugal, high performance file carver*, Digital Forensics Research Workshop, 2005.

[Riv92] R. Rivest, *The md5 message-digest algorithm*, RFC 1321, 1992.

[Roz05] K. Rozinov, *Efficient static analysis of executable for detecting malicious behaviors*, Polytehnic University, Brooklyn, NY, 2005, Master Thesis.

[SA07] Networks J. Seamans, E. and T. Alexander, *Gpu gems 3: Chapter 35 - fast virus signature matching on the gpu*, first ed., Addison-Wesley Professional, 2007.

[San] *The sandy bridge microarchitecture*, http://en.wikipedia.org/wiki/Sandy_Bridge_(microarchitecture).

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, *A fast automaton-based method for detecting anomalous program behaviors*, Proceedings of the 2001 IEEE Symposium on Security and Privacy (Washington, DC, USA), SP '01, IEEE Computer Society, 2001, pp. 144–.

[Sca] *Scalpel*, http://www.digitalforensicssolutions.com/Scalpel/.

[She08] A. Shevchenko, *Malicious code detection technologies*, Kaspersky Lab, 2008, http://brazil.kaspersky.com/sites/brazil.kaspersky.com/files/knowledge-center/malicious code detection technologies.pdf.

[SM07] K. Scarfone and P. Mell, *Guide to intrusion detection and prevention systems (idps)*, Computer Security Resource Center (National Institute of Standards and Technology), 2007.

[sno] *Snort*, http://www.snort.org.

[SQ10] Vermaat M. E. Shelly, G. B. and J. J.(CON) Quasney, *Living in a digital world, introductory*, Discovering Computers 2010, Cengage Learning, 2010, p. 144.

[stu11] *Wired.com*, 2011, http://www.wired.com/threatlevel/2011/07/how-digital-detectives-deciphered-stuxnet/all/1.

[SW81] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences.*, Journal of molecular biology **147** (1981), no. 1, 195–197.

[SXCM04] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, *Static analyzer of vicious executables (save)*, Proceedings of the 20th Annual Computer Security Applications Conference (Washington, DC, USA), ACSAC '04, IEEE Computer Society, 2004, pp. 326–334.

[Tan10] M. Tanase, *Detecting and removing malicious code*, 2010, http://www.symantec.com/connect/articles/detecting-and-removing-malicious-code.

[TCC11] Vrizlynn L. L. Thing, Tong-Wei Chua, and Ming-Lee Cheong, *Design of a digital forensics evidence reconstruction system for complex and obscure fragmented file carving*, CIS'11, 2011, pp. 793–797.

[Tec] *Techreport.com: The gf114*, http://techreport.com/articles.x/20293/6.

[Tom] *Tom's hardware: Geforce gtx 560 ti*, http://www.tomshardware.com/reviews/nvidia-geforce-gtx-560-ti-gf114,2845.html.

[tri] *Tridscan*, http://mark0.net/soft-tridscan-e.html.

[TSCV] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese, *Deterministic memory-efficient string matching algorithms for intrusion detection*, In IEEE Infocom, Hong Kong, pp. 333–340.

[Tum10] A. Tumeo, *Accelerating dna analysis applications on gpu clusters*, IEEE 8th Symposium on Application Specific Processors (SASP), IEEE Xplore, 2010, ISBN 978-1-4244-7953-5, pp. 71–76.

[VAP+08] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis, *Gnort: High performance network intrusion detection using graphics processors*, Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (Berlin, Heidelberg), RAID '08, Springer-Verlag, 2008, pp. 116–134.

[VI10] Giorgos Vasiliadis and Sotiris Ioannidis, *Gravity: a massively parallel antivirus engine*, Proceedings of the 13th international conference on Recent advances in intrusion detection (Berlin, Heidelberg), RAID'10, Springer-Verlag, 2010, pp. 79–96.

[vir12] *Zdnet*, 2012, http://www.zdnet.com/blog/bott/the-malware-numbers-game-how-many-viruses-are-out-there/4783/.

[vN66] J. von Neumann, *Theory of self-reproducing automata*, Essays on Cellular Automata, University of Illinois Press, 1966, pp. 66–87.

[VPI11] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis, *Midea: a multi-parallel intrusion detection architecture*, Proceedings of the 18th ACM conference on Computer and communications security (New York, NY, USA), CCS '11, ACM, 2011, pp. 297–308.

[Wat94] B. Watson, *The performance of single-keyword and multiple-keyword pattern matching algorithms*, Technical Report CS TR 94-19, Department of Computing Science, Eindhoven University of Technology, 1994.

[Wat95] B.W. Watson, *Taxonomies and toolkits of regular language algorithms*, Eindhoven University of Technology, 1995, pp. 53–116.

[Wat04] et al. Watson, B.W., *Spare parts: a c++ toolkit for string pattern recognition*, Software Practice and Experience, 2004.

[wdm] *Windows driver model*, http://msdn.microsoft.com/en-us/library/windows/hardware/ff565698(v=vs.85).aspx.

[Wei73] P. Weiner, *Linear pattern matching algorithms*, 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 1–11.

[Wei81] Mark Weiser, *Program slicing*, Proceedings of the 5th international conference on Software engineering (Piscataway, NJ, USA), ICSE '81, IEEE Press, 1981, pp. 439–449.

[Wika] *Wikipedia - entropy*, http://en.wikipedia.org/wiki/Information_entropy.

[Wikb] Wikipedia, *Cih - computer virus*, http://en.wikipedia.org/wiki/CIH_(computer_virus).

[WM94] S. Wu and U. Manber, *A fast algorithm for multi-pattern searching*, Technical Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[WZ96] B.W. Watson and G. Zwaan, *A taxonomy of sublinear multiple keyword pattern matching algorithms*, Science of Computer Programming, 1996.

[XW07] X. Xu and L. Wang, *Parallel software development with intel threading analysis tools*, Intel Technology Journal, vol. 11, issue 04, 2007.

[Yun12] SangKyun Yun, *An efficient tcam-based implementation of multipattern matching using covered state encoding*, IEEE Trans. Comput. **61** (2012), no. 2, 213–221.

[Zha] Sahni S. Zha, X., *Fast in-place file carving for digital forensics*, e-Forensics.

[ZS08] Xinyan Zha and Sartaj Sahni, *Highly compressed aho-corasick automata for efficient intrusion detection*, ISCC'08, 2008, pp. 298–303.

[ZSS11] Xinyan Zha, D.P. Scarpazza, and S. Sahni, *Highly compressed multi-pattern string matching on the cell broadband engine*, Computers and Communications (ISCC), 2011 IEEE Symposium on, 28 2011-july 1 2011, pp. 257 –264.