Approximate search in misuse detection-based intrusion detection by using the q-gram distance

Sverre Bakke

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2008

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Approximate search in misuse detection-based intrusion detection by using the q-gram distance

Sverre Bakke

2008/05/20

Abstract

Intrusion detection systems have in the last years become a commonly used and important component in the perimeter security for many organizations. Such systems often use a misuse detection-based model for detection of known attack patterns from a finite signature database. By applying fault-tolerant approximate search to the IDS, unknown attacks may also be detected if they are similar enough to known attacks. This is especially efficient if the unknown attack is based on existing, known, attack. However, as the amount of attacks increases, so does the time needed to search through the signature database. Fault-tolerant search also adds complexity to the search and may therefore increase the time needed to search the signature database even further.

Approximate search often uses a distance measure in its pattern matching for finding the distance between the input data and the signatures in the database. If the distance is below a pre-defined threshold, the input data and the signature is considered a match. Previous research have used the edit distance or the constrained edit distance as a distance measure for approximate search. However, both of these have a high time complexity. In this thesis we look at a fast alternative for measuring the distance in approximate search for misuse detection-based intrusion detection, called the q-gram distance. The q-gram distance is known to estimate the edit distance well for some applications. We look at how the q-gram distance can be implemented in intrusion detection as a distance measure in approximate search. We suggest a two-stage approximate search method in which the q-gram distance can be applied to approximate search in intrusion detection, and compare its data reduction, performance and accuracy against the ordinary edit distance and the constrained edit distance using SNORT rules as test data. Experimental results indicate that the q-gram distance can be used as an alternative to the constrained edit distance, and by far outperform both the ordinary edit distance and the constrained edit distance in terms of performance for intrusion detection.

Sammendrag

Innbruddsdeteksjonssystemer har i de siste årene blitt en ofte brukt og viktig komponent i perimetersikringen for mange organisasjoner. Denne typen systemer bruker ofte en signaturbasert deteksjonsmodell for å oppdage spor etter kjente angrep som er lagret i en signaturdatabase. Ved å benytte feil-tolerant tilnærmet søk i innbruddsdeteksjonssystemet kan også ukjente angrep bli oppdaget hvis disse angrepene ligner nok på eksisterende angrep. Dette er spesielt effektivt i de tilfeller hvor det ukjente angrepet er basert på eksisterende kjente angrep. Forøvrig, ettersom mengden med angrep vokser så vil også tiden som er nødvendig for å søke gjennom signaturdatabasen øke. Feil-tolerant søk vil også legge til kompleksitet til søket og kan derfor også ytterligere øke tiden som er nødvendig for å søke gjennom signaturdatabasen.

Tilnærmet søk benytter ofte et avstandsmål i sin mønstergjenkjenning for å finne avstanden mellom inndata og signaturene i databasen. Hvis avstanden er under en forhåndsdefinert terskel vil inndata og signaturen bli beregnet som like. Tidligere forskning har benyttet såkalt “edit distance” og “constrained edit distance” som et avstandsmål for tilnærmet søk, men begge disse avstandsmålene har høy tidskompleksitet. I denne masteroppgaven ser vi på et raskere alternativ for å måle avstand i tilnærmet søk for signaturbasert innbruddsdeteksjon kalt “q-gram distance”. Som avstandsmål er q-gram distance kjent for å kunne estimere vanlig edit distance for enkelte bruksområder. Vi tar en titt på hvordan q-gram distance kan bli implementert i innbruddsdeteksjon som et avstandsmål for tilnærmet søk. Vi foreslår en to stegs metode hvor q-gram distance blir benyttet i tilnærmet søk i innbruddsdeteksjon, og sammenligner ytelsen og nøyaktigheten til denne i forhold til vanlig edit distance og constrained edit distance som avstandsmål i et tilsvarende søk. I disse sammenligningene vil vi benytte regler fra innbruddsdeteksjonssystemet SNORT som vår signaturdatabase. Resultater fra våre eksperimenter tyder på at vanlig q-gram distance kan bli brukt som et alternativ til constrained edit distance, og at den med god margin kan utkonkurrere både vanlig edit distance og constrained edit distance med tanke på ytelse for innbruddsdeteksjon.

Acknowledgements

I would like to thank my supervisor, Professor Slobodan Petrović, for excellent guidance throughout this thesis. His help and comments about the thesis, and our many discussions about the topic, have been invaluable for the quality of this thesis and for my motivation during the thesis. I would also like to thank all my friends at the master's lab (A220) at Gjøvik University College for making this thesis an enjoyable and memorable experience. Especially I would like to thank my opponent, Lars Olav Gigstad, for giving me valuable feedback on my thesis and presentation. Last, but not least, I would like to thank my friends and family for being supportive throughout the course of the thesis, especially in the hectic periods.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topics covered

The purpose of this thesis is to look at how various methods for approximate search can be applied for improving the performance of approximate search in misuse detection-based intrusion detection signature database querying. We will implement the various search methods, with focus on one particular distance measure for approximate search, called the q-gram distance. We will measure the approximate search performance against the SNORT signature database in order to see how intrusion detection systems can benefit from these methods. The results will be used to compare the efficiency of the different methods and see which method is better for intrusion detection.

1.2 Keywords

Intrusion detection, intrusion detection system, intrusion prevention system, misuse detection, IDS, IPS, performance analysis, algorithms, algorithm design and analysis, performance evaluation of algorithms, similarity measures, query processing, search process.

1.3 Problem description

By using intrusion detection systems, operators can detect attacks against their networks. The most commonly used type of intrusion detection systems is misuse detection-based. A misuse detection-based intrusion detection system searches in a signature database for known attack patterns to identify and alert the operator about attacks in the content of network traffic. In such systems, every data packet needs to be compared in real-time against each known attack pattern in order to guard against each and every possible known attack that the operator wishes to detect. The problem with misuse detection-based intrusion detection systems is that they can only detect previously known attacks. New attack signatures are added to the intrusion detection *after* the attacks become known and at this point it may already have been used against the protected domain. Many new attacks are often based upon existing previously known attacks that there already exist intrusion detection signatures for. Unfortunately, with exact pattern matching, which is used in most of the current misuse detection-based intrusion detection systems, these new attacks are not detected as they will not constitute an exact match for the existing signatures. Furthermore, different techniques (e.g. inserting x86 NOP operations to hide known buffer overflow attacks) for evading the intrusion detection systems [7] will not be detected by exact pattern matching. By adding fault-tolerant search to the misuse detection, these new attacks may be detected as a variation of the attacks they are based upon (i.e. detected as a match with a certain number of errors).

Another problem with misuse detection-based intrusion detection is the size of the signature database. Attacks that are published (e.g. on Bugtraq [8]) will often be transformed into an IDS signature after just a few hours [9]. The more attacks are added, the larger the IDS database will be. As the database grows, the time needed to search it increases. Furthermore, the need for fault-tolerant search may increase the computational complexity of the search even further. New attacks are discovered every day and each of these attacks must get its own entry in the signature databases if they are to be detected. With exact search, even every variation of existing attacks must get its own entry. New network equipment with higher transfer rates also adds to this problem since the interval between packets may become significantly lower and therefore give the system less time to finish the search before a new packet is inspected. If the time needed to search the database for a pattern exceeds the interval between incoming packets, then packets may be ignored by the intrusion detection and as a result lead to false negatives where the intrusion detection system do not alert about an attack.

Various methods for approximate search have been proposed for fault-tolerant document and Internet searches. Some of these methods can also be applied to intrusion detection. Unfortunately, the computational complexity of these methods may vary, and as a result may add a significant amount of time to the IDS search. In this thesis we look at how some of these methods can be applied to intrusion detection and try to find out how efficient they are for the purpose of intrusion detection. We look particularly at one distance measure, called the q-gram distance, that is known to be possible to compute efficiently.

1.4 Justification, motivation and benefits

Intrusion detection and prevention systems are an important part of security for many computer networks and organizations. Such systems must analyze the incoming data in real-time in order to detect and respond to attacks. With real-time inspection of the data, the operator or the system itself may perform the appropriate countermeasures for attacks in progress. However, with the rapid increase in network transmission rates and the increasing amount of attack patterns that needs to be searched for each incoming packet, real-time processing may be difficult to achieve. When real-time analysis fails, such attacks may go unnoticed until after it is too late to stop them. Furthermore, when the limitations of the processing capacity are reached, the system may start dropping packets, including packets containing attacks.

Approximate search solves the problem with misuse detection-based intrusion detection systems where only previously known attacks are detected. By adding a fault-tolerant pattern matching to the search, the IDS can make sure that some unknown attacks are also detected. This is an advantage for users and owners of such networks since the IDS will cover more attacks. Approximate search may also reduce the amount of data that needs to be searched considering that similar attacks do not need their own entry in the signature database. However, approximate search is slow. Since the q-gram distance is known to be possible to compute very fast, we hope to overcome the performance issues of approximate search in intrusion detection. This may allow intrusion detection systems to include approximate search techniques while working in real-time at high transmission rates and without dropping packets. The advantage of this is first and foremost that the IDS will not drop packets that may contain attacks that should be discovered. Furthermore, the ability to process more attacks in real-time will obviously benefit the response time when an attack is launched against the network. For an organization this may have economical benefits as a result of less downtime as a result of attacks, or the possibility to increase the bandwidth without risking that the IDS may ignore attacks as a consequence. We hope to provide research that will help intrusion detection and prevention system developers to create systems that can handle the performance issues of approximate search better and by this also, both directly and indirectly, benefit all the users of such systems.

1.5 Research questions

To find out which approximate search method is better for intrusion detection and prevention, the following research questions need to be answered:

1. How can the so-called q-gram distance be applied in approximate search for intrusion detection?
2. How does the q-gram distance compare with other approximate pattern matching algorithms in terms of accuracy and performance?

1.6 Contributions

The contributions of this thesis are the results of the study on how approximate search algorithms can be applied in order to improve the accuracy and performance of intrusion detection systems. Our main contribution is the accuracy and efficiency comparison of various approximate search algorithms for use with misuse detection. The results are presented in form of a chapter that can be used as reference for later research, design, development or evaluation of intrusion detection and prevention systems.

2 State of the art

In this chapter we present the available theory that is relevant for the topic of approximate search in misuse-based intrusion detection. By doing so we hope to provide an overview on where this thesis belongs in the field on intrusion detection and search algorithm theory.

2.1 Intrusion detection theory

Before we can start to discuss search techniques and approximate search we briefly discuss the basic theory of intrusion detection. This will hopefully provide a short introduction to how search and approximate search is relevant for intrusion detection, and by this give a better understanding for the later sections.

2.1.1 Classification

Intrusion detection is the process of identifying and responding to actions set to compromise or act in contrast to the security policy. This process is often automated by one or more intrusion detection or intrusion prevention systems placed at different locations within the protected domain. These systems need a stream of event records (e.g. network packets), an analysis engine for finding attacks in the event stream, and a response component to handle detected attacks (e.g. alerting an operator) [10]. There are many different products [11, 12] that can be used for intrusion detection. The efficiency of these products in terms of accuracy and performance may vary depending on what they are designed to protect and how they detect intrusions. A common method to classify such products is according to what they protect (scope of protection), by how they analyze the data (detection model), and how they respond to an attack.

Classification by scope of protection

Every intrusion detection system monitors a data source and detects any sign of attacks in it. Events that occur in these data sources are later passed to the analysis engine for inspection. The data source is generally decided by the asset that is to be protected, and often represents the location of the IDS within the protected domain. Although an intrusion detection system may have multiple data sources, a general classification of these can be used to categorize the intrusion detection system [10]:

Host-based IDS A host-based IDS is an intrusion detection system that resides internally at a specific host/computer. These systems often look at the operating system level events, and may include data sources such as audit trails and system logs.

Network-based IDS A network-based IDS is located on the network. These systems will look at passing network packets for any sign of an attack. These may look at data aimed at multiple hosts rather than focusing on a single host.

Application-based IDS An application-based IDS also resides at a specific host/computer, but uses application generated data as data source. This includes logs and events generated by specific applications (e.g. a Web server log).

Target-based IDS A target-based IDS differ from the other types of intrusion detection systems. These generate their own data to monitor by looking at the state of protected objects. These systems may for instance use cryptographic measures (e.g. hash functions) to detect modifications in system objects (e.g. system files and processes).

The best known and most used intrusion detection systems are network-based. These systems inspect incoming and outgoing traffic that passes through the system, and try to detect any present patterns of attacks. As a result these systems' efficiency is highly dependent on where in the network they are located (i.e. how much and what type of traffic they can inspect) and the performance and accuracy of the detection techniques. SNORT [11] is an example of a typical network-based IDS, and is the de-facto standard for intrusion detection.

Classification by detection model

When we have a data source that feeds events into the analysis engine, we need to decide how the IDS should examine the data when looking for attacks. Intrusion detection systems can be classified into two main categories based on how they analyze the data [10]:

Misuse detection A misuse-based IDS will look for "known bad" patterns. This is often achieved by comparing the events with rules that define what is considered an attack or violation of the security policy. Misuse detection often uses pattern matching techniques to find whether or not the event matches any of the predefined rules. As a result, misuse detection-based systems can only find known attacks, or variations of known attacks, and are highly dependent on a large and accurate set of rules.

Anomaly detection An anomaly-based IDS will look for events that are rare or unusual. This is achieved by defining what is normal behavior, and events that appear abnormal will be considered an attack. These systems often use statistical methods to find activity that is inconsistent with what the system considers normal.

Misuse-detection is the most commonly used intrusion detection model by current commercial intrusion detection systems because it is much easier to implement compared with anomaly detection. Defining what is normal behaviour is difficult, and anomaly detection systems will therefore generate a large number of false positives (i.e. false intrusion alerts). In this thesis we will focus on misuse detection-based intrusion detection systems.

Passive and reactive systems

After an attack is detected by the analysis engine, an intrusion detection needs to decide how it should respond to the attack. There are two main categories which define the behavior of the intrusion detection systems after an attack is discovered [2]:

Passive A passive IDS will log any attack and alert an operator. The operator can then take a decision how he/she wants to deal with the attack (e.g. block the source of the attack in the firewall) based on the logs.

Reactive A reactive IDS (or intrusion prevention system - IPS) will automate the task of responding to attacks. The IDS will automatically respond to the event by performing an action without relying on manual inspection of the event by an operator. A reactive IDS will be much faster than a passive IDS at responding to attacks, but will not have the intelligence of a human and may therefore lead to a large number of false positives (e.g. block an address that should not be blocked).

2.1.2 Case study: SNORT

Most misuse-based intrusion detection systems are based on exact pattern matching. This means that when some event occurs (e.g. a network packet arrives) the content is compared with a set of pre-defined rules. In this thesis we use the open source IDS SNORT [11] as a case study and in our experimental work, and therefore also look at the current state of the SNORT intrusion detection system architecture and search techniques. SNORT is currently the de-facto standard IDS, and is commonly used in education when learning about intrusion detection. Its flexibility, simplicity and performance makes it the perfect choice of IDS for many research and educational purposes.

Architecture

Although SNORT is first and foremost a misuse detection-based IDS, it can be extended to support different types of intrusion detection techniques. In addition to inspecting the content of packets using a raw pattern matching engine, different plug-ins can provide a more specific analysis which for instance can look at packets with knowledge regarding the operation of the specific protocol. To allow such an extensible architecture, SNORT is built using several components. A general overview of the SNORT architecture/components can be seen in Figure 1.

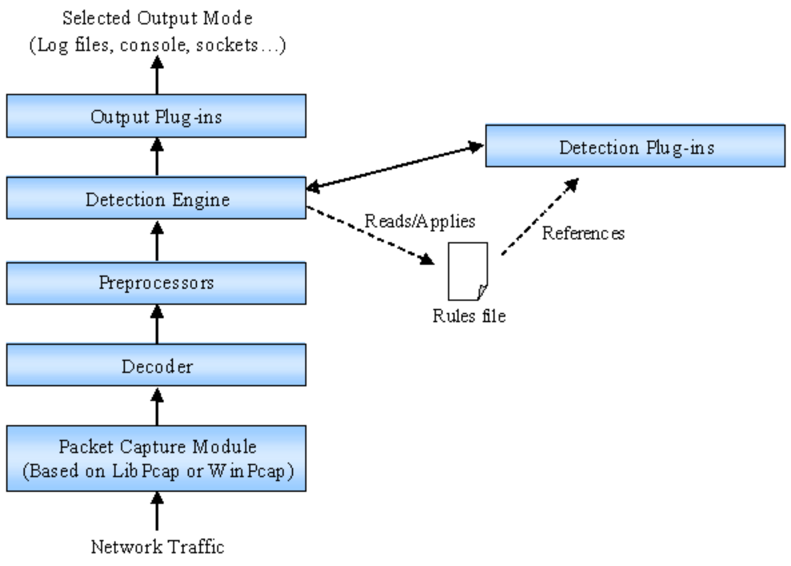


Figure 1: SNORT components [1]

The components are ordered as a chain where each component will produce some output that is used as input for the next component in the chain. The main components of SNORT are the following [1]:

Packet Decoder After capturing packets from the network interface using the libpcap library [13], the packet decoder will read the packets and extract all the information that is needed by the intrusion detection into a more easy to use data structure. This information is then passed to the preprocessor. Already at this component, malformed packets are detected and will trigger alerts.

Preprocessor The preprocessors will take the captured data and preprocess it for later analysis. This can be looked at as a sort of filter that will identify and store information that is needed by the different detection engines. Examples of information that can be found during preprocessing is for instance traffic patterns that can be used to detect denial of service and portscanning, and can be used by the corresponding detection engines.

Detection Engine The detection engine will perform the actual intrusion detection analysis like pattern matching against the signature database. It also utilizes different detection plug-ins for finding more specialized attacks like portscanning, denial of service, and high-level protocol attacks. Even though SNORT is first and foremost a misuse detection-based IDS, even anomaly detection plug-ins can be used.

Output Engine The output engine is used to present the actual alerts that are raised when an intrusion attempt is identified. Also this component is modular so that the notifications can for instance be issued as alerts to an operator or logged to a file/database depending on the context.

Search technique

SNORT is a network-based intrusion detection system that first and foremost uses a misuse-based detection model to find attacks in incoming and outgoing packets. It compares packets against a finite set of rules that state what it should be looking for in the packets. These rules are searched using an exact pattern matching algorithm when a packet arrives at the IDS, and an appropriate action defined in the rule is executed if a match is found. An example of a typical SNORT rule is presented below [14]. This particular rule will detect attempts of remote execution of perl code on a local webserver.

```
alert tcp !$HOME_NET any -> $HOME_NET 80 (msg:“IDS219 - WEB-CGI-Perl access attempt”;
flags:PA; content:“perl.exe”; nocase;)
```

A SNORT rule consists of two parts: a rule header and the rule options [2]. The rule header defines the scope in which the rule applies. It decides what action is to be performed (e.g. log or alert), and what protocols (e.g. TCP/IP and UDP/IP), addresses and portnumbers (source and destination) it should be applied to. Any incoming data are first checked against the rule header to see if the rule applies to the specific packet. If the rule applies to the packet, then the rule options are used to search for a match. The rule options say what content to look for and if any parameters (e.g. packet direction, case sensitivity, etc.) are to be used when searching for content.

When a SNORT rule is loaded into the memory, each rule header is stored in a RTN (Rule Tree Nodes) list that represents all the unique rule headers. Each RTN in the list contains an OTN (Option Tree Nodes) list, which holds the different rule options for the corresponding RTN. When an incoming packet arrives the header is first checked against the RTN list. If a matching RTN entry is found, the OTN list for this RTN entry is searched. A result of this approach is that if a packet does not match any entry in the RTN list there is no need for pattern matching of the content against the OTN lists. In Figure 2 we can see an example of the lists in SNORT. We can see that the RTN lists are organized by which action to perform and which protocol it applies to. Each unique source/destionation pair (e.g. $HOME_NET any -> $HOME_NET 80 like in the previous example) are added as an entry to the appropriate RTN list. Each of these RTN entries has their own OTN list where the option content (e.g. flags:PA and content:"perl.exe" as in the previous example) is added as an entry.

For the actual searching, SNORT initially used a sequential brute-force search algorithm. This algorithm was later, due to performance issues, replaced by the faster Boyer-Moore [15] pattern matching algorithm. Multiple efforts [2, 14] have been made to further improve the Boyer-Moore based search in SNORT. For instance, in [2] a new Boyer-Moore based algorithm is applied to a set of keywords in an Aho-Corasick [16] like keyword tree structure. Since the original Boyer-Moore algorithm is unable to perform multi-pattern matching, this improved technique [2] is suggested due to the great similarities between many SNORT rules.

2.2 Search techniques

In this section we introduce elements of the contemporary search theory. Although most theory regarding searching is focused at finding words and phrases, the same techniques are relevant for information security and intrusion detection.

Searching is about finding a given pattern inside a dataset. It may be used either to find whether or not some data exists in the dataset or at which position in the dataset the pattern is located. Applications of search techniques are everywhere from searching in local documents, searching for documents on the Internet, or even searching internal datastructures stored in memory by most programs.

When searching for a given string in a large dataset there are two main methods of searching: sequential search and index structures [17, 18]. The advantages and cost of both approaches depends on the size and volatility of the dataset. For medium sized datasets hybrid methods that combine both sequential search and index structures are often used [17]. We discuss here the most relevant research related to both methods.

2.2.1 Sequential search

The easiest search method is the sequential search (online search). In sequential search the dataset is iterated sequentially, usually from top to bottom, and for each item in the dataset the search string is compared for matches. The search will iterate through the dataset until a match is found or until the entire dataset is iterated. At the end of the search we will have found whether or not the string exists and its location within the dataset. If we want to find all matches instead of just the first, the entire dataset needs to be iterated. In situations where no further

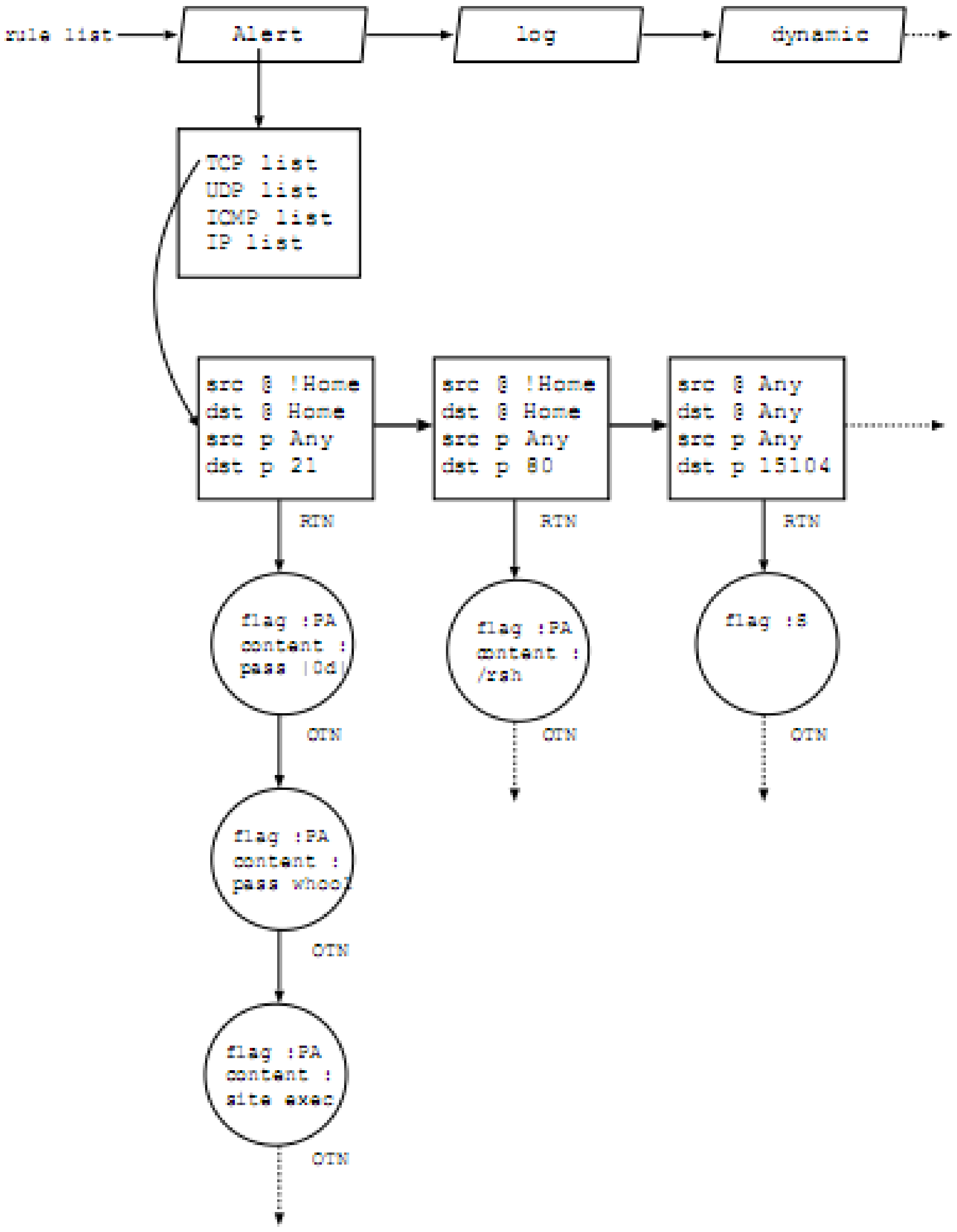


Figure 2: Example of the signature lists in SNORT [2]

#	t	h	i	s		i	s		a		t	e	s	t		s	t	r	i	n	g
1	t	e	s	t																	
2		t	e	s	t																
3			t	e	s	t															
4				t	e	s	t														
5					t	e	s	t													
6						t	e	s	t												
7							t	e	s	t											
8								t	e	s	t										
9									t	e	s	t									
10										t	e	s	t								
11											t	e	s	t							

Figure 3: Example of brute-force search with the search string “test”

information about the dataset is given, sequential search is the obvious approach for searching in the dataset [19]. It is also the only choice if the dataset is very volatile (changes frequently) or if there are strict space requirements [17]. For small datasets sequential search is sufficient. However, since sequential search in many cases needs to search the entire dataset, the run-time may be too slow for many applications.

We look at a few commonly used sequential search algorithms. Unfortunately, these algorithms can only find exact matches, i.e. they are not fault-tolerant.

Brute-Force

The Brute-force algorithm [17, 20] is the simplest of the sequential search algorithms. The idea is to compare the search pattern with every pattern position within the dataset in order to find whether or not the pattern is present at each possible position. Many applications use a variation of this algorithm in which a sliding window of length m (where m is the search pattern length) slides over the dataset and the content of the window is compared with the search pattern. When the search pattern has been matched against the current window the window is shifted forward to the next position. This is an easy way to find if a given search pattern is a substring of the dataset at a given position. An illustration of this process is described in Figure 3. Here we can see that the algorithm needs to shift the search string “test” 11 times before finding a match in the dataset.

If we assume a dataset of length n and a search pattern of length m the brute-force algorithm needs to check $O(n)$ positions where each position has the worst-case cost of $O(m)$. As a result the worst case search cost of the brute-force algorithm is $O(nm)$, which is not efficient enough [21]. However, since the pattern can be found anywhere in the dataset, even at the first position, the average cost is $O(n)$ [17].

Knuth-Morris-Pratt

Considering that brute-force search has the worst-case complexity of $O(nm)$, this is a slow algorithm. An alternative algorithm is the Knuth-Morris-Pratt algorithm [17, 20, 22]. The Knuth-Morris-Pratt algorithm was the first sequential search algorithm with linear worst-case complexity. However, on average it is not that much faster than the brute-force algorithm [17]. Like the

#	0	1	2	3	4	5	6	7	8	9	..
	a	b	c	a	b	c	a	b	d		
1	a	b	c	a	b	d					
2				a	b	c	a	b	d		

Figure 4: Example of the Knuth-Morris-Pratt algorithm [3]

brute-force algorithm, it uses a sliding window, but unlike the brute-force algorithm it reuses information from previous checks. This information is used to decide how many positions the sliding window can skip in case of a mismatch. As a result it does not need to check every possible position in the dataset.

The algorithm starts by preprocessing the search pattern and building an auxiliary *next* table. The *next* table's i-th value then tells us which is the longest prefix in the search pattern $P_{1,\dots,j-1}$ that is also a suffix. When we encounter a mismatch between the i-th character in the sliding window and the i-th character in the search pattern we can then find how many positions m the sliding window can skip, without missing the prefix of possible pattern matches, using the expression:

$$m = i - next[i] + 1 \tag{2.1}$$

In Figure 4 we can see an example of how the Knuth-Morris-Pratt algorithm is an improvement over the brute-force algorithm. In this example we want to find the string “abcabd” inside a dataset. In the first comparison, every character in the search string is a match at the position in the dataset except the last character. At this point, the match will fail and if we had been using the brute-force algorithm we would have shifted the dataset one position (e.g. one character). However, with the Knuth-Morris-Pratt algorithm we know from our previous comparison (using our *next* table) that the prefix of the string does not match any position in the dataset until position 3. As a result, the dataset is shifted 3 positions instead of just 1, and we avoid unnecessary comparisons.

Building the *next* table has the complexity of $O(m)$, while the search given the *next* table has the worst-case complexity of $O(n)$ and takes at most $2n$ comparisons [17, 22]. As a result, the total complexity of the Knuth-Morris-Pratt algorithm is $O(m+n)$ [22]. An extension of the same algorithm is the Aho-Corasick algorithm [16, 17, 23], which uses the same concept, but uses a trie-like data structure in order to match a set of patterns. Like the Knuth-Morris-Pratt algorithm, the Aho-Corasick algorithm has the search complexity of $O(n)$ and uses at most $2n$ character comparisons.

Boyer-Moore

The Boyer-Moore family of algorithms [15, 17, 20, 24] is another alternative for sequential search. As with the Brute-force algorithm and the Knuth-Morris-Pratt algorithm, the Boyer-Moore algorithm also uses a sliding window. The Boyer-Moore algorithm uses two different heuristics (depending on the variant of the algorithm) for finding the longest possible shift after a mismatch: the bad character heuristic and the good suffix heuristic. The advantage of this approach over the Knuth-Morris-Pratt algorithm is that both heuristics can lead to shift of the sliding win-

#	0 a	1 b	2 b	3 a	4 d	5 a	6 b	7 a	8 c	9 b	.. a
1	b	a	b	a	c						
2						b	a	b	a	c	

Figure 5: Example of the Boyer-Moore algorithm [3]

dow one entire pattern/window length. The idea with the Boyer-Moore algorithm is that the matching between the search pattern and the sliding window is processed backwards since this produces more information. First of all, if it finds a character that does not exist in the pattern, the window can be shifted all the way past the position of this character. This means that if the last character in the window is known not to exist in the pattern, the pattern cannot occur in the entire window, and we can slide the window forward a whole window-length after just checking one single character. This is called the bad character heuristic and is the reason why the performance of the algorithm increases with a longer search pattern. Like the Knuth-Morris-Pratt algorithm, it also excludes positions from the search where the algorithm decides that the search pattern cannot occur based on previous knowledge (information gained by a mismatch), thus saving it from searching every position in the dataset. Since the algorithm is rather complex, simpler variants have been proposed, like the Boyer-Moore-Horspool algorithm [25] and the Sunday algorithm [26], where only bad character heuristic is used.

Furthermore, like the Knuth-Morris-Pratt algorithm, the Boyer-Moore algorithm uses preprocessed jump tables in order to find how many positions to skip when a mismatch occurs. However, unlike the Knuth-Morris-Pratt algorithm it uses two tables instead of just one. The first table holds the characteristics for each of the different characters in the alphabet used in the pattern. If the character $char$ does not occur in the pattern then the corresponding entry will hold the pattern-length $patlen$ (i.e. shift one entire pattern-length) while the entry will hold $patlen - j$ where j is the maximum value such that $pattern[j] = char$. The other table holds the characteristics of the suffix occurrences in a similar manner to the Knuth-Morris-Pratt *next* table. If a mismatch is found the algorithm will skip $\max(table1[i], i - table2[dataset[i + j]])$ positions where i is the current position in the pattern/window while j is the current position within the dataset.

In Figure 5 we can see an example of how the improvements in the Boyer-Moore algorithm are beneficial over the Knuth-Morris-Pratt algorithm. In the first comparison, the search string “babac” is searched from the rightmost position to the left for a match. The match will fail straight away since the character at that position in the dataset is not only a mismatch compared with the search string, but it is not even in the pattern at all. A mismatch at this point will look up the table that tells the algorithm to shift the dataset one whole window (pattern length) since it is impossible to find this character anywhere in the pattern. If the character that causes the mismatch had occurred somewhere else in the pattern, then the dataset would be shifted to the position of the mismatch. We can easily see how this can improve the performance since the sliding window will be able to shift longer distances than allowed by the Knuth-Morris-Pratt algorithm.

For preprocessing the Boyer-Moore algorithm requires $O(m + \sigma)$ time and space (where σ is

Document	Occurrences
1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

Figure 6: Example of a dataset with six documents with a single line of text each [4]

the size of the alphabet). The search time for the best variant of the Boyer-Moore algorithm is in the worst case $O(n)$ [24], while other variants of the Boyer-Moore algorithm may have the worst case search time of $O(mn)$ [17]. However, on average the algorithm is "sublinear", thus making it one of the fastest algorithms on average [17]. An interesting note is that a property of the algorithm is that as the search pattern gets longer, the algorithm gets faster. This means that it performs worse for short search patterns, and binary string in particular because of the short alphabet.

2.2.2 Indexed search

The problem with sequential search is that the search may need to inspect the entire dataset, thus making it slow. A solution to this is to use so-called index structures and indexed search. In situations where the dataset is large and semi-static, using an index structure can speed up the search significantly [17]. The idea behind an index is that we do not need to search the entire dataset, but only a subset that is large enough to represent the location of data inside the dataset. However, building and maintaining such structures can be slow, and for large datasets with frequent updates, the overhead cost for maintaining the structure may be too high. There are many different indexing techniques for different purposes. We will briefly look at three main indexing techniques: inverted files, suffix arrays, and signature files.

Inverted files

Inverted files [4, 27, 28] (inverted index) are the most popular data structures for document search [29] and are in literature described as the currently best choice for most applications [17] due to its simplicity to maintain compared with other alternatives. The idea is to create a mapping between some content and all its positions within a dataset. An inverted index consists of a vocabulary and the occurrences. The vocabulary is a set of all the items in the dataset (e.g. all words in a document) [17]. Each of the items in this vocabulary is then coupled with a set of occurrences which represents the items positions (i.e. address) within the dataset. In [23] the inverted index is defined as follows:

> An index into a set of texts of the words in the texts. The index is accessed by some search method. Each index entry gives the word and a list of texts, possibly with locations within the text, where the word occurs.

In Figure 6 we see a sample dataset to be indexed using an inverted file [4]. In this sample dataset we have six documents where each document contains one single line of text. In Figure 7 we can see the corresponding inverted file [4]. Each term (i.e. word) from the sample dataset

Term t	Frequency	Inverted list for t
and	1	(6,2)
big	2	(2,2) (3,1)
dark	1	(6,1)
did	1	(4,1)
gown	1	(2,1)
had	1	(3,1)
house	2	(2,1) (3,1)
in	5	(1,1) (2,2) (3,1) (5,1) (6,2)
keep	3	(1,1) (3,1) (5,1)
keeper	3	(1,1) (4,1) (5,1)
keeps	3	(1,1) (5,1) (6,1)
light	1	(6,1)
never	1	(4,1)
night	3	(1,1) (4,1) (5,2)
old	4	(1,1) (2,2) (3,1) (4,1)
sleep	1	(4,1)
sleeps	1	(6,1)
the	6	(1,3) (2,2) (3,3) (4,1) (5,3) (6,2)
town	2	(1,1) (3,1)
where	1	(4,1)

Figure 7: Example of an inverted file [4]

has its own entry in the inverted file. Each of these entries has a frequency counter which tells us in how many documents the term occurs. Last, but not least, each entry has a list of tuples that tell us which documents the term occurs in, and the frequency of the term within the document. Another variation of this list could be to replace the frequency counter with the position of each occurrence of the term within the document. If we want to search for a certain term in this inverted file, we can find the entry corresponding to the term. From here we know that the term occurs in each of the documents that is listed in the inverted list of this entry. As an example we can see that the term “big” occurs in 2 documents: 2 times in document number 2, and 1 time in document number 3. If we want to search for the terms “big”, “had”, and “keep”, we can use the intersection of their inverted lists that would return a set with all documents that contains all three terms (only document is used in our example): $\{2, 3)\} \cap \{3\} \cap \{1, 3, 5\} = 3$. This tells us that all three terms only occur in the third document.

To build an inverted file the dataset needs to be scanned and the position of each item in the vocabulary is then added to the set of occurrences. The cost of building such an index is $O(n)$ (where n is the size of the dataset). Inverted files can hold different levels of addressing granularity. Each occurrence in the set can represent which documents contain the item, a specific block within a document (e.g. the i-th block), a specific word within a document (e.g. the i-th word), or a specific character position in a document (e.g. the i-th character) [17]. With a finer level of granularity more accurate searches like phrase searches can be achieved. The space requirements of such a structure are according to Heaps’ law that the vocabulary will grow as $O(n^\beta)$ where β is a constant between 0 and 1 dependent on the text (in practice between 0.4

#	Suffix
1	AACCACG
2	ACCACG
3	CCACG
4	CACG
5	ACG
6	CG
7	G

Figure 8: Suffixes for string s = "AACCACG"

and 0.6) [17]. However, the occurrences will take up more space as each word in the dataset will be pointed to once. The occurrence space overhead will in practice be between 30% and 40% of the text size. Searching an inverted file is as simple as searching the vocabulary for a match and retrieving the set of occurrences when a match is found. These occurrences can then be processed as locations where the search query was found. Although one can search the vocabulary by iterating a list sorted in lexicographical order, different techniques and structures, like hashing [29], tries, or trees [30], can be used to speed up the search. The use of such techniques gives a search cost of $O(m)$ (where m is the search pattern size) while the use of lexicographical ordered lists gives a cost of $O(\log n)$.

Suffix trees

Suffix trees [31, 32, 33] are another category of index structures. It is more costly to build the index compared with inverted files, but it allows for answering complex queries really fast. The idea is that the dataset is a long string of data. The substrings between each position and the end in this string are then called suffixes. All positions used in suffix trees are called index points, and those positions which are not index points cannot be retrieved. Suffix trees work on either entire words or single characters. When the set of all suffixes is found, a suffix tree, containing all the suffixes, is built where each path from the root to a leaf represent one unique suffix. Suffixes that have a similar beginning will share a path down the tree until they differ. At the point they differ they will split into different subtrees. When a path can uniquely represent a suffix (i.e. it will no longer fork into multiple subtrees since no other suffixes share the same beginning) no more characters from the suffix need to be stored in the tree. The leaf node at the end of each path will contain a pointer to the position in the dataset where the suffix the path represents is located.

Let us assume we want to build a suffix tree for the string s = "AACCACG". First we need to find all suffixes. The suffixes for string s are presented in Figure 8. These suffxies are then inserted into a tree so that each of the unique suffixes have their own unique path from the root and down to a leaf node. This leaf node will then hold the position for the suffix within the dataset. The compact suffix tree for string s is presented in Figure 9.

A suffix tree will in many cases produce a space overhead of 120% to 240% over the dataset size and therefore uses a lot of memory [17]. For large datasets the entire tree may not even fit in memory. To solve this problem, a suffix array can be used instead of a suffix tree. A suffix array is simply an array that contains pointers to all suffixes sorted in lexicographical order. A suffix

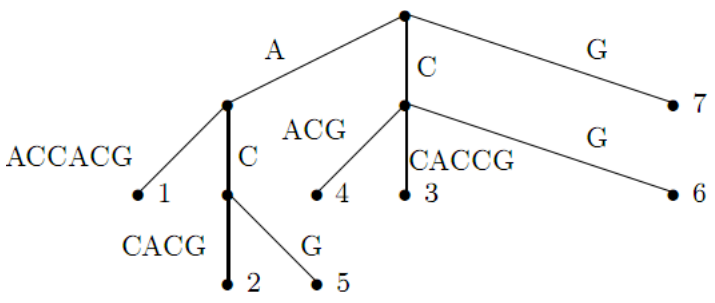
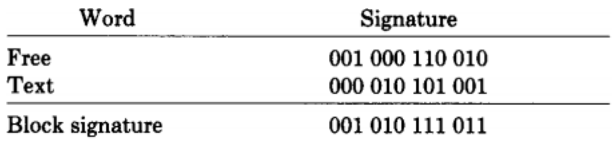


Figure 9: Compact suffix tree for the string s = “AACCACG” [5]



Word	Signature
Free	001 000 110 010
Text	000 010 101 001
Block signature	001 010 111 011

Figure 10: Example of signature files with blocks of 2 words [6]

array will use about the same space as an inverted file index which is around 40% of the size of the dataset [17]. While suffix trees can search the structure in $O(m)$ time by a simple trie search, a suffix array can perform the same search in $O(\log n)$ time by doing a binary search [17].

Signature files

Signature files [6, 27, 28] is yet another indexing structure. Unlike inverted files or suffix trees, signature files have a very low storage overhead with only 10% to 20% overhead over the dataset size [17]. However, as a result the search complexity is increased to linear instead of sublinear like the other approaches. Signature files are suitable for small datasets, but are often outperformed by inverted files [17]. When using signature files the dataset is divided into blocks of the same number of words. Each of the words in a block is then mapped to a fixed size bitmask (signature) by using a hash function. All the bitmasks inside a block are then OR'ed together by using boolean algebra, thus creating the signature of the entire block. An example of such a bitmask signature can be seen in Figure 10.The idea is that if a word is in a block then the signature of the block should have the same bits set to 1 as the signature of the search word. In some cases the same bits will be set to 1 even though the block does not contain the search word and as a result creates false matches. The accuracy of this index is therefore dependent on the quality of the hash function and the size of the blocks [17]. Signature files were mostly used in the 1980's and are today mostly replaced by inverted files [17].

2.3 Approximate search

As we have seen, search techniques with sublinear complexity have been found by researchers decades ago and are today widely deployed in modern applications. Algorithms like the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm as previously discussed can be very efficient, but will only find exact matches. For situations where fault-tolerant search is needed, these regular search methods cannot be used. The solution is to use approximate string/pattern matching.

One situation where approximate search is helpful over regular search is for Internet search. With regular search methods the search algorithm can give a list of webpages that contain the exact search pattern. If the user has misspelled the search query then this may ignore all the relevant webpages. By using approximate search, the user may also get all the similar pages. This scenario also applies in other situations like data communications where some data may be lost during transfer, in spam e-mail messages where the text is deliberately distorted in order to evade the automated spam filters, in spellcheckers where the program can make suggestions that resemble the misspelled word, and in molecular biology where even the same proteins may differ slightly within the same species [34].

In approximate search the idea is to find one or more matches for a short pattern P in the dataset T that only contains a maximum number of k errors. This means that if the distance between the search pattern P and some substring of the dataset T is lower than or equal to the threshold k, the substring will be accepted as a match. If the distance is higher than k, the substring will be rejected as a mismatch. Although approximate search is a newer problem than exact string matching, researchers have already suggested multiple solutions [35, 36, 37] to the problem.

2.3.1 Measures

The central concept of approximate search is to find a way to measure distance between two strings. This distance is later tested against the threshold value to see if the strings are similar enough to accept them as a match. If we were dealing with integer values we could subtract the two values and use the result as the difference. However, finding the distance between strings is a more complicated procedure. Many measures have been suggested as a way of finding the distance between two strings. A distance function $D(X, Y)$, where X and Y are two strings, is required to satisfy the following conditions:

- $D(X, Y) \geqslant 0$ (non-negativity)
- $D(X, Y) = 0$ iff $X = Y$ (identity of indiscernibles)
- $D(X, Y) = D(Y, X)$ (symmetry)
- $D(x, z) \leqslant D(x, y) + D(y, z)$ (triangle inequality)

Perhaps the most commonly used distance functions are the Hamming distance and edit distance, but other measures may in many situations prove to be better for certain problems. We look at a few existing distance measures that were shown to be suitable for string comparison.

Hamming distance

The Hamming distance [38] is perhaps the easiest way to measure the distance between two strings. The idea is that for two strings of equal length, the distance between these are the number of positions in the strings where the characters at that position do not match [17]. In other words, for two strings of equal length, the Hamming distance is the number of substitutions that is required in order to transform one of the strings into the other, e.g. it adds one to the distance for each position in the strings where the characters at this position are not equal. This

A	1	0	1	0	0	1	1
B	1	1	0	0	0	0	1
$\lvert A_i - B_i \rvert$	0	1	1	0	0	1	0

$$\sum \lvert A_i - B_i \rvert = 3$$

Figure 11: Example of Hamming distance between two binary strings

A	s	e	n	a	t	o	r
B	s	e	m	i	n	a	r
$\lvert A_i - B_i \rvert$	0	0	1	1	1	1	0

$$\sum \lvert A_i - B_i \rvert = 4$$

Figure 12: Example of Hamming distance between two character strings

means that for two completly different strings with no equal characters, the distance is the same as the length of the strings. As a result, to compute the Hamming distance we only need to compare n characters at most, where n is the length of the strings.

In Figure 11 we can see an example of using the Hamming distance to find the distance between two binary strings. The bits that differ will add 1 to the distance while bits that are equal will add 0 to the distance. The Hamming distance can also be used between character strings. In Figure 12 we can see an example where the Hamming distance concept is implemented on character strings. Like the binary strings, each character is matched and if they differ 1 is added to the distance while 0 is added to the distance if they are equal. When a mismatch is found the actual character values are of no importance, only that they differ.

Hamming distance is easy to implement and its computation is quite fast. However, it can only work with equally sized strings. This limits the number of situations where the Hamming distance can be used to implement approximate searches. Hamming distance is often used for bit-level analysis in telecommunication, information theory, coding theory and cryptography.

Edit distance

To solve the problem with Hamming distance and its inability to find the distance between strings of different lengths, the edit distance (often called Levenshtein distance) is often used. The edit distance can be defined as follows [23]:

> The edit distance is the smallest number of insertions, deletions, and substitutions required to change one string or tree into another.

Unlike the Hamming distance that only counts substitutions (and as a result only can work on equally long strings), the edit distance also counts insertions and deletions. This means that two strings of any two lengths can be matched. If no substitutions, insertions or deletions are needed to transform one string into the other the distance will be zero. Furthermore, the distance will never be a greater value than the size of the longest of the two strings, and will always be at least the difference in length between the two strings. For two strings of equal length, the Hamming distance will be an upper bound of the edit distance.

A	t		u	e	s	d	a	y
B	t	h	u	r	s	d	a	y
		+		=				

Edit operations = 2

Figure 13: Example of Edit distance between two character strings

		t	u	e	s	d	a	y
	0	1	2	3	4	5	6	7
t	1	0	1	2	3	4	5	6
h	2	1	1	2	3	4	5	6
u	3	2	1	2	3	4	5	6
r	4	3	2	2	3	4	5	6
s	5	4	3	3	2	3	4	5
d	6	5	4	4	3	2	3	4
a	7	6	5	5	4	3	2	3
y	8	7	6	6	5	4	3	**2**

Figure 14: Example of Edit distance matrix between two character strings

The Levenshtein distance algorithm can be computed by means of a bottom-up dynamic programming [17] method for finding the distance. Dynamic programming is a method to solve problems by dividing the problem into simpler subproblems. When these subproblems are solved the results can be combined to solve the original problem.

When the algorithm for finding the edit distance begins, a matrix consisting of n rows and m columns, where n and m are the lengths of the two strings A and B, respectively, is created. Each of the cells in this matrix represents the comparison between the two characters located at the corresponding positions in the strings. The first row is then initialized to $0 \dots n$ and the first column to $0 \dots m$. Once the matrix is initialized, each character in the two strings (positions 1 to n and 1 to m) are examined by the algorithm. If both characters $A[i]$ and $B[j]$ are equal ($A[i] == B[j]$) then the cost of the transformation is set to 0 and if they differ ($A[i] \mathrel{!=} B[j]$) to 1. The value of the cell $d[i, j]$ (the cell corresponding to the match between the character at positions i in the string A and j in the string B) is then set to the minimum/lowest value of the following three computations:

- $d[i-1, j] + 1$ (deletion)
- $d[i, j-1] + 1$ (insertion)
- $d[i-1, j-1] + \mathrm{cost}$ (substitution)

Once each of the characters in the string A is matched against each of the characters in the string B, the entire matrix has been populated. At this point, $d[n, m]$ (i.e. the bottom right cell) contains the minimum number of operations needed to transform one of the strings into the other.

In Figure 13 we can see an example of the computation of the edit distance between two character strings. The minimal number of basic edit operations needed in this example is 2 as only one insertion (or deletion depending on which string is being transformed into the other) and one substitution are needed to transform A to B. In the example, the letter “h” is inserted into the string (denoted in Figure 13 as “+”) and the letter “e” is replaced with the letter “r” (denoted in Figure 13 as “=”). There are of course other edit operation combinations that can achieve the same result (e.g. substitution of all characters), but the two operations described is the absolute minimum. The corresponding populated distance matrix for the example can be seen in Figure 14.

Another variation of the edit distance is the Damerau-Levenshtein distance [39, 40]. This is really only an extension of the original Levenshtein distance where another elementary edit operation is counted. In addition to substitution, insertion and deletion, the Damerau-Levenshtein distance has a character transposition operation that swaps two characters in the string as one operation. In his paper [39] about the Damerau-Levenshtein distance, the author states that the four elementary edit operations correspond to 80% of human misspellings. Other possible variations of the edit distance include adding different penalty costs to the different edit operations [41] or to the different characters.

The edit distance (with different variations) is one of the most common measures used today in approximate string matching and are often used in areas such as text databases and molecular biology. Unfortunately, its computation is rather slow with the time complexity $O(mn)$. This is because each character in the first string needs to be compared with each character in the second string during the population of the matrix.

One of the ways to increase the efficiency of the approximate search is to apply a special index structure called a V-tree [34]. Instead of computing the distance between the entire dataset or a vocabulary in an inverted index and the search pattern sequentially word by word, a V-tree can discard parts of the dataset. Because computing the distance is a slow procedure, reducing the number of items in the dataset that needs to be compared will increase the performance significantly. A V-tree solves this by using a tree structure and a filtering technique. By partitioning the dataset into subgroups and clustering these by distance from each other in a tree structure, the search algorithm can skip all items from the dataset that are not likely to match. In [34] this method proved to be very successful in increasing the performance of approximate search.

Constrained edit distance

Depending on the context and application, different variations of the edit distance may provide better results than the original Levenshtein distance. One of the variations is the constrained edit distance [42, 43]. This distance uses constraints to the edit distance regarding the maximum number of consecutive insertions and deletions. Like with the ordinary edit distance we use three elementary edit operations substitution, insertion and deletion. However we also add the following constraints to the operations [42, 43]:

C_1 The number of insertions belongs to a set T given in advance.

C_2 The maximum lengths of runs of deletions and insertions are F and G, respectively.

C_3 The edit sequence is ordered in a sense that every substitution is preceeded by at most one run of deletions followed by at most one run of insertions.

Given these constraints, the constrained edit distance is defined as the minimum number of elementary edit operations needed to transform one string into another. The elementary distances associated with the edit operations deletions, insertions and substitutions can then be defined as follows, given the alphabet $\mathcal{A}$ and the "empty" symbol ϕ, where $\mathcal{A}^* = \mathcal{A} \cup \{\phi\}$ [42]:

- $d(a, \phi)$ is the elementary distance when deleting a symbol $a \in \mathcal{A}$.
- $d(\phi, b)$ is the elementary distance when inserting a symbol $b \in \mathcal{A}$.
- $d(a, b)$ is the elementary distance when substituting a symbol a with b, where $a, b \in \mathcal{A}$

We can also define a compression operator $C(Z')$ that will transform an *edit sequence* Z' into a string Z by removing all empty symbols ϕ. An edit sequence is every permitted editing transformation (from string X to string Y) as a sequence of editing operations, that are allowed by the constraints C_1, C_2, C3. An edit sequence is denoted as $S = (X', Y')$, where $(X', Y') \in \mathcal{A}^* \times \mathcal{A}^*$ (X' and Y' are the strings X and Y where ϕ has been inserted to denote edit operations), and where the following rules hold:

- $C(X') = X, C(Y') = Y$.
- $\mid X' \mid = \mid Y' \mid = L$, $\max\{N, M\} \leqslant L \leqslant N + M$.
- for all $i, 1 \leqslant i \leqslant L$, the characters x'_i and y'_i cannot both have the value ϕ simultaniously.
- S satisfies the constraints C_1, C_2, C_3.

Furthermore, $\mathcal{G}_T$ is the set of all permitted edit sequences S, where S transforms the string X into the string Y. The constrained edit distance can then be computed using the following expression [42]:

$$D(X, Y) = min \left\{ \sum_{i=1}^{L} d(x'(i), y'(i)) | (X', Y') \in \mathcal{G}_T(X, Y) \right\}, \tag{2.2}$$

where $\max\{N, M\} \leqslant L \leqslant N + M$.

Like the ordinary edit distance, the constrained edit distance also populates a matrix while computing the distance, and as a result also has the same time complexity problem. The constrained edit distance was also suggested for use with digital computer forensics [42] and intrusion detection [43].

q-gram distance

Although the performance of the edit distance computation is rather slow, it is still one of the most commonly used approximate search distance measures. However, for many applications where performance is important the computation of the edit distance is too slow. As a result, an alternative measure called the q-gram distance [44, 45, 46] has been found to estimate the edit distance quite well [47].

The q-gram distance between two strings is the difference between the occurrence count for each substring of length q (a q-gram) within the strings. This means that the two strings count the occurrences of every possible substring of length q and the difference between the count for each of these substrings are added to the distance between the strings. The idea is that similar strings will share a lot of common q-grams. The accuracy of the q-gram distance is dependent on the value for q and a higher value will give more accurate results. When using q-grams of length 1, we will only be counting individual letters. Since the q-gram distance only counts occurrences with no regards to their position within the strings, two strings may have a low distance without actually being similar. As a result, the q-gram is only considered a pseudo-metric considering that two different strings with the same q-gram occurrence count will have a distance of 0.

The advantage with the q-gram distance over edit distance is that it is possible to compute it quite fast. With a complexity of $O(n + m)$ the computation of the q-gram distance will by far outperform the edit distance that has a quadratic complexity $O(nm)$ where n and m are lengths of two strings X and Y.

One important application of the q-gram distance is that it can be used as a filter to speed up the edit distance [47] by first applying the q-gram distance and later increase the accuracy by inspecting the results that got accepted by the q-gram distance using the edit distance. During the filtering, the q-gram distance will reject any data that has a distance above the threshold value. When a data item is rejected by pattern matching using the q-gram distance we know that this would also be rejected by pattern matching using the edit distance. However, all items accepted by pattern matching using the q-gram distance will not be accepted by pattern matching using the edit distance. As a result, all items accepted by the q-gram distance pattern matching will therefore be further inspected by the edit distance pattern matching in order to reject false matches. This is possible since the q-gram distance can be used as a lower bound of the edit distance as $D_q(X, Y)/2q \leqslant D_{ed}(X, Y)$ for two strings X and Y [44, 47].

Like the edit distance, also the q-gram distance can be coupled with index structures to increase the performance. In [47] a so-called Monotonous Bisector Tree (MBT) is used to store the words from a vocabulary in an inverted index file. This tree structure is then used to ignore parts of the vocabulary, thus reducing the number of items where the q-gram distance needs to be computed. This allows us to quickly find any matches without computing the distance between the pattern and every item in the dataset. In [48], the q-gram distance has also been implemented by researchers as user-defined functions in DBMS in order to implement approximate string processing without major changes in the underlying database system.

We look further into the concept of q-grams, and how to apply this in approximate search for misuse detection-based intrusion detection systems in the next chapter.

2.3.2 Information security and intrusion detection

Although research on approximate search for many decades had focused on document and database search, more recent research has extended this effort to other fields of computer science. Information security is one of the areas where approximate search techniques can be highly beneficial.

In [49] approximate search techniques have been applied to computer forensics in order to study the problem of data reduction. As the amount of data in modern computer systems increases, more data needs to be processed when performing computer forensics. This is especially a problem since investigators often need to manually inspect the data. By using the Levenshtein distance algorithm, approximate search could be used to filter the data and significantly reduce the dataset for further inspection manually or with the help of slower algorithms. The tests showed that the amount of data could be reduced by approximately 50% using the Levenshtein distance as a filter.

In [42] this concept was studied further by applying the constrained edit distance instead of the Levenshtein distance as in [49]. A two stage indexless search procedure was suggested where the first stage was using the constrained edit distance for data-reduction and the second stage used a slower exhaustive search for fine grained inspection of the data. Using approximate search can in this setting find data that have been deliberatly distorted in an attempt to prevent the data from being detected by means of ordinary search methods. By applying constrained edit distance instead of the edit distance, the amount of data that needs to be inspected by a exhaustive search method is significantly reduced and as a result the search time is also reduced. Experimental work showed that by applying constrained edit distance it was possible to achieve up to 80% dataset reduction over the ordinary edit distance.

The same two stage approximate search procedure was in [43] applied to intrusion detection. The Snort IDS signature set was used as test-data in the experimental work, and by using the constrained edit distance on the dataset a 70% data reduction was achieved during the first stage of the two stage search procedure. The first stage of the procedure will find the packets that are candidates for a closer inspection, and the second stage will use slower exhaustive search mechanisms to further inspect the data. For intrusion detection, fault-tolerant search is efficient to find new variations of existing attacks without the need for explicit signatures preloaded. This is especially useful when attacks have been deliberatly distorted in order to bypass the intrusion detection pattern matching. As intrusion detection needs to complete pattern matching as fast as possible in order to process all data on time, reducing the dataset is an excellent way of increasing the performance.

3 Approximate search in misuse-detection using the q-gram distance

In this chapter we introduce an approach to approximate search in misuse-based intrusion detection using the q-gram distance. The approach makes use of the previously suggested two stage indexless approximate search algorithm [42, 43] in a combination with the q-gram distance for pattern matching. This represents the theoretical contribution of this master thesis, and is the basis for our experimental work. Our focus is mainly on the q-gram distance for pattern matching. However, we will also use both the ordinary edit distance and the constrained edit distance as references for our results.

3.1 Approximate search for misuse detection

A general misuse-based intrusion detection system often receives some activity data (e.g. network packets), processes this data in an analysis engine using pattern matching against a set of rules (e.g. a signature database), and if a match is found alerts the operator or log the result. In Figure 15 we can see such a system where an analysis engine compares the incoming activity with every signature in the signature database in an attempt to find a match. A simple method to implement approximate search could then be just to replace the analysis engine's exact pattern matching algorithm (in Snort's case the Boyer-Moore sequential search algorithm) with an approximate search algorithm. Such an approximate search could use a sliding search window over the dataset where the content of this window could be compared with the input data using a fault-tolerant pattern matching algorithm. This search could then be considered exhaustive.

However, signature databases are often large, and grow at a rapid rate in order to include all currently known attacks. This makes the exhaustive search on the entire database a rather slow process, especially considering that intrusion detection systems may have problems already at high transfer rates with exact pattern matching. The solution to this problem is to reduce the

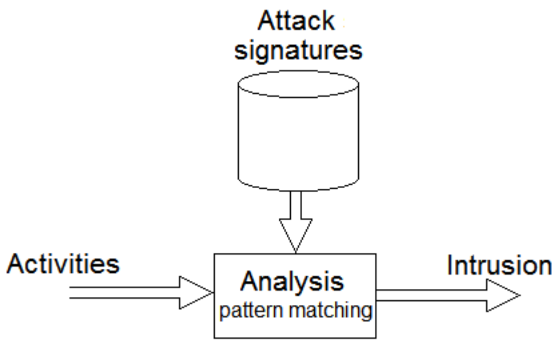


Figure 15: Architecture of a generic misuse-based IDS

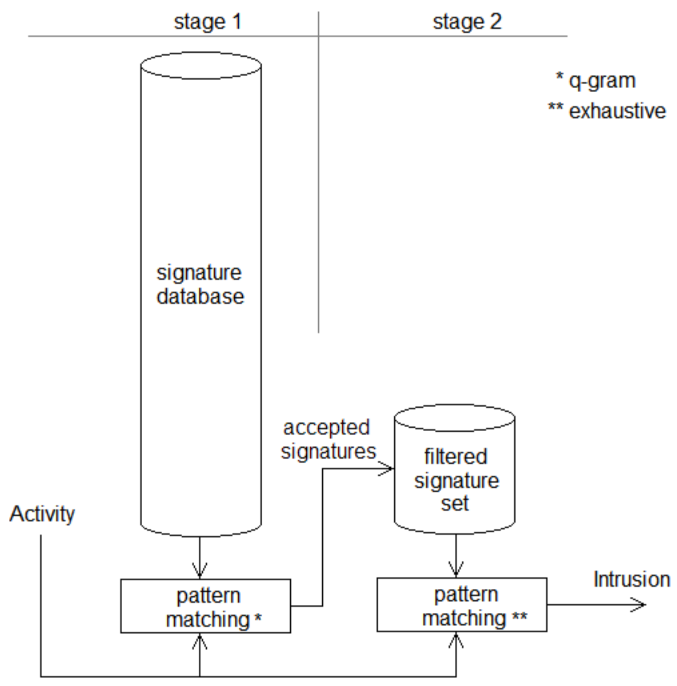


Figure 16: Architecture for two-stage approximate IDS search

signature dataset that needs exhaustive search to include only the signatures that are likely to match, or in other words, exclude all signatures that are unlikely to match.

The method to achieve this is to implement a two-stage pattern matching process. Figure 16 shows a system based on this two-stage search procedure. Like the misuse-based intrusion detection system in Figure 15, the system reads activity data as input. In the first stage, the system compares the activity data with every signature in the entire signature database using approximate pattern matching based on the q-gram distance. If the signature from the database matches the activity data with a distance lower than the error treshold k, the signature is accepted into the filtered signature dataset for further inspection in stage two. If the distance between the signature and the activity data is above the error threshold k, the signature is rejected and does not qualify for the second stage. In the second stage, the reduced dataset that was accepted in previous stage will be matched against the activity data once again, but this time using a finer, exhaustive, pattern matching algorithm.

The advantage of this method is that the q-gram distance is fast with a complexity of $O(n+m)$, while exhaustive search is slow. By using a fast algorithm to reduce the dataset, much less data needs to be processed by the exhaustive search algorithm, and as a result the entire search should be faster. In this thesis we focus on the first stage as exhaustive search is straightforward.

This concept can further be enhanced by using index and tree structures (e.g. Monotonous Bisector Trees [47] and inverted index files). However, this is beyond the scope of this thesis.

3.2 q-gram distance pattern matching

Different variations of the edit distance are some of the most common measures for finding the distance between two strings. Unfortunately, the edit distance is computed in quadradic time and as a result its computation is slow. For situations where the running time of the computation is important, quadratic time is too slow. One alternative that has been proposed as a simpler alternative to the edit distance is the q-gram distance, which is reported to estimate the ordinary edit distance well [44]. Unlike the edit distance whose complexity is quadratic, the q-gram distance is computed in linear time [47].

The q-gram (sometimes also known as n-gram [44, 46]) distance is used to measure the difference between two strings of variable length. Given a generic string S, a q-gram is any substring of length q within this string. The idea behind the q-gram distance is that similar strings will have a lot of common q-grams while strings that differ will have fewer common q-grams. The q-gram distance is found by counting the number of occurrences of all q-grams and find the difference in the q-grams occurrence count between the two strings. Because we find the q-gram distance by counting occurrences of q-grams without regard to their position, the q-gram distance is considered a pseudo-metric [44]. This means that although equal strings always will have zero distance, zero distance will not always mean that the strings are exact matches. In cases where two distinct strings have exactly the same q-gram count for all q-grams but at different locations within the strings (i.e the strings are anagrams to each other), the q-gram distance will also say that they are exact matches (i.e. have 0 distance). Furthermore, if we use $q = 1$ we will only count occurrences of single letters.

More formally a q-gram can be defined as follows [44]:

Definition 1 *Let $\mathcal{A}$ be a finite alphabet. Let $\mathcal{A}^*$ denote the set of all strings in $\mathcal{A}$ and $\mathcal{A}^q$ denote the set of all strings of length q over $\mathcal{A}$, sorted in lexicographical order, where q equals a positive integer $q = 1, 2, \ldots, n$. A q-gram is any string $v = a_1, a_2, \ldots, a_q$ in $\mathcal{A}^q$.*

Following the definition of the q-gram, we can formally define the q-gram distance. However, we first need to define the q-gram profile, which is used to compute the q-gram distance:

Definition 2 *Let S be a string in $\mathcal{A}^*$. The q-gram profile $G_q(S)$ is a vector of length $z = |\mathcal{A}|^q$ whose i-th element equals the number of occurrences of the i-th q-gram of $\mathcal{A}^q$ in the string S. E.g. $G_q(S)[i]$ equals the number of occurrences of $\mathcal{A}^q[i]$ where i is a integer between 0 and $|\mathcal{A}|^q$. The value i is then called the* fingerprint *of the q-gram it corresponds to in $\mathcal{A}^q$.*

Now that we have defined the prerequisites we can formally define the q-gram distance:

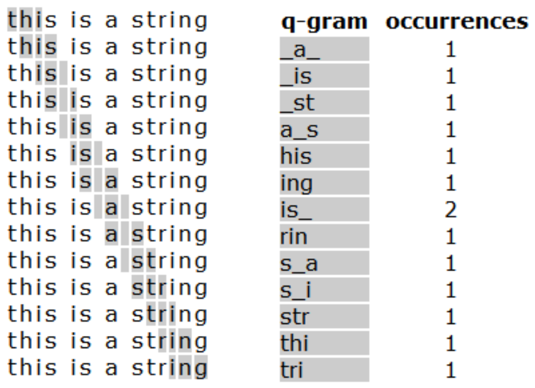
Definition 3 *Let S_1 and S_2 be two arbitrary strings. The q-gram distance $D_q(S_1, S_2)$ between S_1 and S_2 is defined as the L_1-distance (absolute distance, Manhattan distance or city block distance) between their corresponding q-gram profiles $G_q(S_1)$ and $G_q(S_2)$.*

We will later take a closer look at how we can generate q-gram profiles and how we can compute the distance between these.

3.2.1 Generating q-gram profiles

When computing the q-gram distance between two strings we are in reality computing the distance between two vectors called q-gram profiles. A q-gram profile $G_q(S)$ is a vector in which each item corresponds to the occurrence count of a unique q-gram in $\mathcal{A}^q$. As a result, a q-gram profile contains the occurrence count for each possible q-gram in $\mathcal{A}^q$ even for the q-grams that the string does not contain.

In [48] the process of obtaining the q-grams from a string is described as using a "sliding window" technique. This concept is perhaps the easiest abstraction of the process of creating a q-gram profile. The idea is that for any character at position $1, 2, \ldots, n - q + 1$ (where $n = |S|$) in a string S we can fetch q consecutive characters from this position. The way to do this is to iterate the string from the start to the length n of the string minus the last $q - 1$ characters. For each of these iterations we fetch the character at position i plus the next $q - 1$ consecutive characters. The sliding window abstraction can be seen in Figure 17.



```
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
this is a string
```

q-gram	occurrences
_a_	1
_is	1
_st	1
a_s	1
his	1
ing	1
is_	2
rin	1
s_a	1
s_i	1
str	1
thi	1
tri	1

Figure 17: A sliding window technique for counting q-grams ($q = 3$)

To find the mapping between a specific q-gram and the corresponding item in the q-gram profile vector the fingerprint of the q-gram is computed. The fingerprint is a unique numerical representation for the specific q-gram and represents the number (index) in which the corresponding q-gram appears in $\mathcal{A}^q$ when sorted in lexigraphic order. As a result, this fingerprint can identify the position of the specific q-gram occurrence count in the q-gram profile vector. The fingerprint of a q-gram $v = (v_1, v_2, \ldots, v_q)$ can be interpreted as an integer $f(v)$ in base-z (where $z = |\mathcal{A}|$) notation using the following expression:

$$f(v) = \sum_{i=1}^{q} \bar{v}_i z^{q-i} \tag{3.1}$$

where $\bar{v} = j$ if $v_i = a_i$, $i = 1, 2, \ldots, q$, $j = 0, \ldots, z - 1$. $f(v)$ is a bijection since $f(v) = f(v')$ if and only if $v = v'$.

Computing a q-gram profile $G_q(S)$ is all about extracting all q-grams from a string, finding the index for each q-gram in the q-gram profile by using the fingerprint, and setting the value at this index to the number of occurrences of the q-gram. Using the sliding window technique

(Figure 17) we can compute the fingerprints of all q-grams $W_i = s_i, s_{i+1}, \ldots, s_{i+q}$ where $i = 1, \ldots, n-q+1$ in string $S = s_1, \ldots, s_n$ $(n = |S|)$. The concept of finding the fingerprint of a q-gram using expression 3.1 is illustrated in Figure 18.

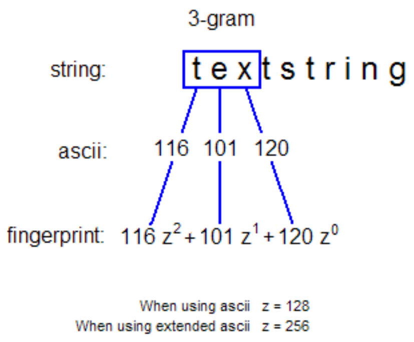

3-gram
string:
t e x t s t r i n g
ascii:
116 101 120
fingerprint: 116 z^2 + 101 z^1 + 120 z^0
When using ascii z = 128
When using extended ascii z = 256

Figure 18: Finding a fingerprint using expression 3.1

Although expression 3.1 is sufficient to find all the q-grams alone by just iterating each position in S for $i = 1, \ldots, n-q+1$, this is a process that has a a lot of duplicate computations. Instead, we will use another method that is similar, but spares us for unnecessary computations. We start by finding the first fingerprint W_1 in the string $S = s_1, s_2, \ldots, s_n$ by applying the abovementioned expression $f(W_1) = \sum_{i=1}^{q} \bar{s}_i z^{q-i}$. When this fingerprint is found, it is used to increase the corresponding counter in the q-gram profile by 1 in order to say that "this q-gram occurred". After the first q-gram is found we can find the rest of the fingerprints using the following expression for $1 <= i <= n-q+1$:

$$f(W_{i+1}) = (f(W_i) - \bar{s}_i z^{q-1})z + \bar{s}_{i+q} \tag{3.2}$$

For each and every one of these q-gram fingerprints $f(W_i)$ we increase the occurrence counter value $G_q(S)[f(W_i)]$ by 1. When all q-grams have been iterated the q-gram profile should contain the number of occurrences of all possible q-grams in the alphabet, even those that are not found in the string S.

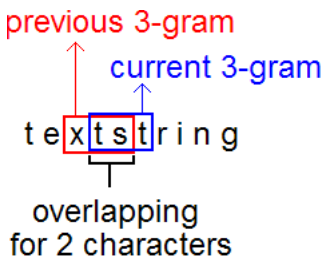


Figure 19: Re-use of previous q-gram fingerprint

As mentioned before, using a combination of these two expressions to compute the fingerprints rather than only expression (3.1) alone we will save a lot of duplicate computations. The expression 3.2 will re-use the fingerprint for the previous q-gram since they will share $q-1$

characters. Since consecutive q-grams share $q - 1$ characters, this method is especially beneficial for higher values of q. The concept of q-gram fingerprint re-use can be seen in Figure 19. In order to transform the previous q-gram fingerprint into the next q-gram fingerprint W_{i+1} all that is needed is the previous fingerprint W_i, the previous character $\bar{s}_i$, and the new character $\bar{s}_{i+q}$ that is not already a part of the previous fingerprint. Expression (3.2) then removes the character that is no longer a part of the fingerprint, multiplies the remaining $q - 1$ characters by z in order to shift all letters one spot to the left, and then add the new character. When this process is complete the previous fingerprint is transformed into the current fingerprint without the need to do unnecessary computations.

3.2.2 Finding the distance

After we have established the q-gram profiles for the strings we wish to compare, these can be used to compute the distance between the two strings. The q-gram algorithm uses the L_1-distance [50] (also known as the absolute distance, Manhattan distance or city block distance) between the two q-gram profiles. Since both of the q-gram profiles $G_q(S_1)$ and $G_q(S_2)$ for strings S_1 and S_2 must be computed using the same alphabet and q-value, they will both be vectors of the same length where identical index values represent the same exact q-gram within the vectors. This makes the vectors easy to iterate and sum the absolute distance between the q-grams.

Given two strings X and Y represented as q-gram profiles $G_q(S_1)$ and $G_q(S_2)$, the q-gram distance $D_q(S_1, S_2)$ can be formally defined as follows:

$$D_q(S_1, S_2) = \sum_{v \in \mathcal{A}^q} | G_q(S_1)[v] - G_q(S_2)[v] | \tag{3.3}$$

The algorithm for computing the q-gram distance will iterate through the q-gram profiles $G_q(S_1)$ and $G_q(S_2)$, and for each item finds the absolute value of the difference between the two profiles using the absolute distance. Each item in the profile is a numerical value that represents the number of times a specific q-gram occurs in the string the profile represents. Getting the same item from two different profiles allows us to find the difference in the number of times the corresponding q-gram occurs between the two strings. This distance will then be added to the total distance, thus yielding the total q-gram distance between the strings S_1 and S_2 at the end of the algorithm.

3.2.3 Examples

To fully understand how q-gram works we will take a quick look at a couple of q-gram examples. In these examples we will adjust the alphabet and q in order to provide a better understanding of the possibilities with q-grams. To keep the examples understandable we will use small values for q.

Example 1 — *2-gram with binary alphabet*
Let us assume that we want to find the 2-gram (q-gram where $q = 2$) of two strings X and Y using the binary alphabet. The first string $S_1 = 10101010$ and $S_2 = 1101100$. Because we are using the binary alphabet and the only two allowed characters are 0's and 1's, the alphabet $\mathcal{A} = \{0, 1\}$ with cardinality $|\mathcal{A}| = 2$. Both S_1 and S_2 consist only of 0's and 1's, and are both therefore also in the set $\mathcal{A}^*$ as required. This means that the set of all possible 2-grams (every

variation of $q = 2$ consecutive characters) $\mathcal{A}^q = \{00, 01, 10, 11\}$. The first thing we need to do is to compute the 2-gram profiles for both strings S_1 and S_2. We can see that S_1 contains 0 occurences of 00, 3 occurences of 01, 4 occurences of 10, and 0 occurences of 11. This results in the 2-gram profile $G(S_1) = (0, 3, 4, 0)$. If we repeat the procedure for S_2 we see that S_2's 2-gram profile $G(S_2) = (1, 1, 2, 2)$. At this point all that remains is to compute the L_1-distance between the two 2-gram profiles $G_q(S_1)$ and $G_q(S_2)$. Using $G_q(S_1)$ and $G_q(S_2)$ we find that $D_q(S_1, S_2) = |0 - 1| + |3 - 1| + |4 - 2| + |0 - 2| = 7$.

As we can see, computing the 2-gram distance of a small alphabet like the binary is rather simple to do manually by hand. Each 2-gram profile is a vector of $| \mathcal{A} |^q = 2^2 = 4$ elements.

Example 2 — *3-gram with ASCII alphabet*
Let us assume we want to find the 3-gram (q-gram where $q = 3$) of two strings S_1 and S_2 using the US-ASCII alphabet. The first string S_1 = "Kitten" and S_2 = "Sitting". The regular 7-bits US-ASCII alphabet consists of 128 different characters. However, because 33 (the first 32 and the last one) of the characters are control characters and not used in simple text strings without linefeeds and carriage returns, these can be discarded. This leaves an alphabet consisting of lower case letters (a-z), upper case letters (A-Z), numbers (0-9) and special characters (mathematical symbols, comma, question mark, exclamation mark, etc.). As a result the alphabet $\mathcal{A} = \{␣, \ldots, a, b, \ldots, \sim\}$ with cardinality $|\mathcal{A}| = 95$. With this alphabet and q=3 we get the set of all 3-grams $\mathcal{A}^q = \{␣␣␣, \ldots, aaa, aab, \ldots, \sim\sim\sim\}$ consisting of all possible 3-gram permutations in the alphabet $\mathcal{A}$. For the 3-gram profile $G_q(S_1)$ we see that it contains "Kit", "itt", "tte", "ten" with only one occurrence of each thus giving the 3-gram profile vector for X the value 1 in the fields belonging to each of these 3-grams, while all other fields are set to 0. For the 3-gram profile $G_q(S_2)$ we see that it contains "Sit", "itt", "tti", "tin", "ing" with only one occurrence of each, thus giving the 3-gram profile vector for Y the value 1 in the corresponding fields while all other fields are set to 0. When calculating the $D_q(S_1, S_2)$ using the previously calculated $G_q(S_1)$ and $G_q(S_2)$ profiles we see that they have different values for 7 fields where each has the value 1, thus giving the distance $D_q(S_1, S_2) = 7$.

Computing the 3-gram distance with a large alphabet like the US-ASCII is a much more tedious task to do manually by hand, compared with the 2-gram distance with the binary alphabet in the first example. The reason is that as the alphabet and the q-value increases in size, so does the vectors used to represent the q-gram profile. In example 2 each of the profiles $G_q(S_1)$ and $G_q(S_2)$ needs to store $| \mathcal{A} |^q = 95^3 = 857375$ items each. Although this may seem like a lot of numbers, this is still no problem for a computer to handle. As we will see later, this problem can be reduced using the so-called compact q-profile.

3.2.4 Performance

The biggest advantage of the q-gram distance is the performance of the algorithm for its computation compared with the performance of computation of other distance measures used in approximate pattern matching algorithms. When using the q-gram distance the performance will be heavily dependent on the generation of the q-gram profile.

The suggested algorithm for creating q-gram profiles generates a vector of z^q elements (where

z is the size of the alphabet used) although each string can contain only a minimum number of 1 and maximum number of $n - q + 1$ (where n is the length of the string) distinct q-grams. As a result most elements in the q-gram profile will hold the value 0 (i.e. there are 0 occurrences of most of the possible q-grams). This means that the time and space complexity of computing the q-gram profile are as follows [47]:

Lemma 1 The q-gram profile $G_q(S)$ of a string S with length n can be computed in $O(n)$ time and occupies $O(z^q)$ space.

We can see that storing all the 0's is inefficient with regards to space complexity, and inefficient with regards to computing the L_1-distance as the $D_q(S_1, S_2)$ would have to iterate through all the z^q possible q-grams even though most of them are not used. This can be solved by using the so-called compact q-gram profile [47] where only the q-grams that occurred within the string (i.e. elements that have a value higher than zero) are stored in the q-gram profile. This can be implemented using standard hashing techniques and data structures. As a result the space needed to store the profile and the time needed to iterate the profile will be significantly reduced [47]:

Lemma 2 The compact q-gram profile $G_q(S)$ of a string S with length n can be computed in $O(n)$ expected time and occupies $O(n)$ space.

We can see that the time/space complexity efficiency of computing the q-gram distance is highly dependent on which method we use to store the q-gram profile. If we assume we use the more efficient compact q-gram profile, the time complexity of computing the distance between two strings is linear [47]:

Lemma 3 Given the q-gram profile of two strings X and Y with length n and m respectively, the q-gram distance $D_q(X, Y)$ between X and Y can be computed in $O(n + m)$ time.

Performance advantages for misuse detection

In addition to the lower time complexity and thus faster running time of the q-gram distance over the edit distance, the separation between the computation of the profiles and the distance is a huge performance advantage. This means that the q-gram profiles can actually be generated at any point before the distance is computed. The computation of the q-gram profiles is time consuming, and by doing this only once for each string rather than for each time the distance is computed, we can save a lot of time. This is especially the case when the same strings (e.g. input data and signatures) are used multiple times in different distance computations. Since the q-gram profiles only need to be computed once, this can for static strings be done when loading the strings into memory or even be loaded from a file on the harddrive containing strings and pre-computed q-gram profiles. For intrusion detection, this ability can be highly beneficial since the signatures/rules can have their q-gram profiles computed when they are inserted into the database. Furthermore, for incoming data that are to be checked for intrusion, the q-gram profile can be computed only once when the data is received instead of each time it is compared against a new signature/rule.

3.2.5 Disadvantages

Although computation of the q-gram distance has performance advantages over the computation of alternative distances, it has one big disadvantage. Unlike the edit distance, the q-gram distance cannot be computed for strings that have a length less than the q-value. The reason is that such strings will have no q-grams (substrings of length q) since the entire string is shorter than q. Although this most definitely is a problem since the algorithm is incapable of processing such a string, there are ways to circumvent this. Perhaps the most obvious way to solve the problem is to pad the string until it is of length q (i.e. holds exactly 1 q-gram/substring of length q). Another suggestions to circumvent the problem may be to process these short strings with another pattern matching algorithm that is capable of processing such strings. Another disadvantage is, as mentioned, that the q-gram distance is considered a pseudo-metric. This means that the q-gram distance may be less accurate than alternative distance metrics. However, how much of a problem this really is in our particular case is open for discussion. Since our search is approximate, such inaccuracies should be found by the second stage of our two-stage approximate search algorithm, and as a result have little effect on the number of false positives.

3.3 Exhaustive pattern matching

When the first stage in our two-stage approximate search algorithm has reduced the dataset, an exhaustive search algorithm may inspect the reduced dataset closer. As previously explained, an exhaustive search over the dataset may use a sliding search window where the content of the entire window is compared with the input data. The pattern matching algorithm used for comparing the window with the input data may vary depending on the application. Perhaps the simplest algorithm for exhaustive pattern matching is the brute-force algorithm discussed in the state of the art chapter. However, considering that the brute-force algorithm is an exact search algorithm without fault-tolerance, this may not be sufficient for approximate search. For fault-tolerant search, the window could for instance use one of the previously suggested distance measures (e.g. the edit distance) for comparing the content of the window with the input against a pre-defined threshold. Other possibilities for exhaustive fault-tolerant search include data mining techniques such as dtSearch [51]. The exhaustive search in the second stage of our approximate search algorithm is straightforward and because of that it is not discussed here.

3.4 The IDS data reduction algorithm

At this point we have presented the theoretical concepts of the general two-stage approximate search intrusion detection and the q-gram distance pattern matching. We now look at how the q-gram distance can be implemented into the first stage of the two-stage intrusion detection analysis engine.

3.4.1 Generating q-gram profiles

The complete algorithm for computing the q-gram profile $G_q(S)$ for a string S is given in pseudo-code as follows:

Algorithm 1 $\mathrm{QProfile}(S)$ - Get the q-gram profile
Input:

- S - An arbitrary string of which we want to find the q-gram profile.
- q - Size of the q-grams used.

Output:

- A vector that contains the q-gram profile of the string S.

```
begin
  $profile[] = \text{new Vector}[z^q]$
  $wi = 0$
  comment Computing the first q-gram fingerprint
  for $i \longleftarrow 1$ to $q$ do
    begin
      $wi = wi + (S[i] * z^{(q-i)})$
    end
  comment Count the occurrence of the first q-gram fingeprint
  $profile[wi] = 1$
  comment Computing the rest of the q-gram fingerprints
  for $i \longleftarrow 1$ to $n - q + 1$ do
    begin
      $wi = wi + ((((wi - S[i]) * z^{(q-1)}) * z) + S[i + q])$
      comment Count the occurrence of the next q-gram fingeprint
      $profile[wi] = profile[wi] + 1$
    end
  return $profile$
end
```

The algorithm above is a conceptual implementation of the previously discussed expression (3.1) and expression (3.2) for finding, and counting, each q-gram in string S. The idea is to, for each q-gram, compute the fingeprint, which is used as the position within the q-gram profile to increase when counting the specific q-gram. The algorithm can be summarized with the following steps:

1. Initialize a new empty q-gram profile (all fields equal to zero).
2. Compute the first q-gram fingerprint in S using the expression (3.1).
3. Count the first q-gram using the q-gram fingerprint.
4. Compute the next q-gram fingerprint in S using the expression (3.2).
5. Count the next q-gram using the q-gram fingerprint.

6. Repeat the steps 4–5 until all q-grams are counted.
7. Return the q-gram profile.

3.4.2 Computing the q-gram distance

The algorithm for computing the L_1-distance given two q-gram profiles $G_q(X)$ and $G_q(Y)$ is given below as pseudo-code:

Algorithm 2 $\mathrm{QDist}(px, py, q)$ - Get the q-gram distance

Input:

- px - The q-gram profile $G_q(X)$ of the string X.
- py - The q-gram profile $G_q(Y)$ of the string Y.
- q - Size of the q-grams used.

Output:

- A positive integer that holds the q-gram distance between the strings X and Y.

```
begin
  $dist = 0$
  for $i \longleftarrow 0$ to $|\mathcal{A}|^q - 1$ do
    begin
      comment Finding the distance between the strings for each q-gram
      $dist = dist + \mathrm{Abs}(px[i] - py[i])$
    end
  return dist
end
```

The algorithm above is used to find the distance between two strings X and Y by using their q-gram profiles px and py. This algorithm is basically just an implementation of the L_1-distance over two equally sized vectors. We assume that the two q-gram profiles have already been pre-computed using the previous algorithm. The idea is that the algorithm will iterate the two q-gram profiles at the same time and for each element (which represents the occurrence count of a unique q-gram) find the difference between the values stored in the two profiles. The algorithm can be summarized in the following steps:

1. Set the initial distance $dist$ to 0, so that if all q-gram counts are equal we have 0 as distance.
2. Find the absolute difference between the first q-gram in the two profiles and add it to the total distance $dist$.

3. Repeat step 2 for all possible q-grams (i.e. every possible q-gram in $\mathcal{A}^q$).
4. Return the distance between the strings X and Y.

3.4.3 The search algorithm

Now that we have defined the algorithms for generating the q-gram profiles and finding the distance between these, we can finally define the search algorithm:

Algorithm 3 - Finds which signatures are eligible for the exhaustive search

Input:

- P - Traffic packet/activity data.
- D - Signature database denoted as a list of IDS rules.
- q - Size of the q-grams.

Output:

- A sorted list F of accepted signatures for finer, exhaustive, inspection.

```
begin
  comment Computing the q-gram profile of P
  $pp = QProfile(P, q)$
  comment Comparing P with every signature in D
  for $i \longleftarrow 1$ to $| D |$ do
    begin
      $S = D[i]$
      comment Compute the q-gram profile of the current signature
      $ps = QProfile(S, q)$
      $d = QDist(ps, pp, q)$
      comment Store the distance and the position of the signature into F
      Store (d, i) into F sorted ascending by d
    end
  return F
end
```

This algorithm represents the first stage in our two-stage approximate search process as described in figure 16. It takes the content of the incoming packet and matches it against the complete signature database. The distance between the packet P and the signature S is then saved together with the position of the signature in the database into a list. This list is sorted in ascending order based on the value of d, thus making a ranked list where better results come first. The list is then returned for use with a finer, exhaustive, search algorithm. The exhaustive algorithm can then check all items in the list until the d value exceeds the error threshold k. We can summarize the algorithm in the following steps:

1. Compute the q-gram profile pp for the input P.
2. Fetch a signature S from the signature database D.
3. Compute the q-gram profile ps for the signature S.
4. Find the distance between their q-gram profiles pp and ps.
5. Store a tuple consisting of the position/index of the signature within the database and the distance in a list sorted by the distance.
6. Repeat the steps 2–5 for all the signatures in the signatures.
7. Return the sorted list for use in the exhaustive search in the second stage.

One important part of this algorithm is that the q-gram profile of P is computed before the signature database iteration. As the q-gram distance of P remains the same for each comparison, this only has to be computed once, and as a result will save us some time. This concept can also be applied to the signature database. The q-gram profiles of all the signatures could be pre-computed and stored with the signatures. As a result we could also avoid the time needed to compute the profile of all the signatures for each incoming packet.

4 Experimental work

In this chapter we present our experimental work based on the theoretical basis presented in the previous chapter. The experimental work is first and foremost designed in order to answer our research questions, and will hopefully tell us how efficient approximate search using the q-gram distance is for misuse-based intrusion detection.

4.1 Scope

In this thesis we have looked at how the q-gram distance and approximate search can be applied to misuse-based intrusion detection from a theoretical point of view. In order to find the efficiency of q-gram distance and approximate search as a method for intrusion detection we need to carry out several experiments. We look at the accuracy and performance of the algorithm for distance computation in order to gain a better understanding of how well it is suited for approximate search in intrusion detection. The results are matched against other previously suggested methods for approximate search as a reference. Our focus in this experiment is on the first stage in the two-stage approximate search procedure. Furthermore, we limit our experiments to sequential searches.

4.2 The method

In order to find the efficiency of approximate search for misuse-based intrusion detection we need to conduct several experiments. The method we have chosen is to implement the previously discussed algorithms and let them process actual signatures from an intrusion detection system against a set of input data. We will discuss each of our different experiments in further detail later, but first we give a general overview of our chosen method.

We implement the following three algorithms for distance measures for pattern matching algorithms and compare their properties against a signature database and an input dataset:

- q-gram distance
- ordinary edit distance (for comparison)
- constrained edit distance (for comparison)

Our primary focus in the experiments is on the q-gram distance. However, we also need to implement the edit distance and the constrained edit distance for comparison. For the q-gram distance, the q-value (i.e. the size of the q-grams) is in our experiments adjusted to different values in order to see how this value influences the results. The same applies for the parameter of the constrained edit distance: the maximum number of consecutive deletions constraint variable F. Since the input data in our experiments is always shorter than the rules, the insertion constraint variable G is not needed in the constrained edit distance. As a result we will have a lot of different variations of the algorithms that need to be compared. In our experiments we used q-grams of sizes 1, 2 and 3. If we were to use larger q-grams we would have to use another q-gram algorithm due to memory constraints. Furthermore, larger q-gram would have had issues with shorter search patterns, and thus excluded a lot of patterns from the search. We also adjusted the constrained edit distance's deletion constraint F between 1 and 5 in order to allow the length of runs of deletions to be between 1 and 5. This particular range was chosen because 1 is the minimal value if we are to allow deletions while 5 is a rather long length of runs of deletions which should account for all patterns that are "similar enough". As a result, a deletion constraint between 1 and 5 should cover the most realistic cases of the constrained edit distance. Furthermore, the threshold values Δ were adjusted between 0 and 3 in order to allow both few and many errors. For the q-gram distance, this was later increased to 15 for reasons we will look into in our results chapter.

We have previously presented the concept of approximate search for misuse-based intrusion detection where a signature was accepted as a candidate for finer inspection in the second stage if it matched the input data with no more than k errors. In our experiment we use this concept to match each rule in the signature database against each item in an input dataset. This means that we perform approximately 43 000 comparisons (i.e. each input matched against each rule). Since the SNORT rules are much longer than the input data (i.e. contain a lot more data than just the content field), we use $N-M+\Delta$, where N is the length of the rule, M is the length of the input, and Δ is the predefined threshold value, as our threshold in order to take this difference in lengths into account. This concept is presented in Figure 20. If the distance between the input data and the rule is less than or equal to $N - M + \Delta$, we accept the rule for further inspection. In our experiments we vary the Δ threshold between 0 and 3 in order to see how the threshold value affects the results.

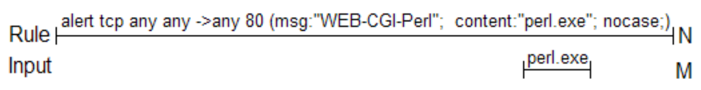

Rule
alert tcp any any ->any 80 (msg:"WEB-CGI-Perl"; content:"perl.exe"; nocase;)
N
Input
perl.exe
M

Figure 20: Difference in length between rule and input

4.3 The dataset

We have chosen to use standard predefined Snort rules as our signature database considering that SNORT is not only the de-facto standard intrusion detection system, but also because it is freely available. More specifically we have chosen the standard web-misc ruleset (*web-misc.rules*) as this set contains a large number of rules and all of the rules have a content field representing the actual signatures. As SNORT can detect many types of attacks, including those that do not use signatures, the web-misc ruleset is a safe bet since all the rules have this content field. This allows us to perform our experiment on real signatures that are actually in use, thus making our results more illustrative. The rules are also padded (using the character "#") to equal length (i.e. all rules have the same length as the longest rule) to make sure that a shorter rule does not get an unfair advantage (i.e. shorter distance) over the others when computing the distance.

In addition to the signature database we also need to have input data to match against the signature database. The input will represent the content (i.e. body) of a network packet. Because the headers of the packet are irrelevant for our pattern matching we only care about the actual content. We have chosen to use the content of the selected rules as the input data. We extract each unique content field (124 in total) from the web-misc rules and use these as our input/content. That way we get at least one match for every rule in the dataset, but also get a lot of situations where a specific input does not match the specific rule. The advantage of this approach rather then using real network traffic is that random packets do not influence our results (i.e. we have complete control over the data and environment), and that the experiment is easily repeatable.

4.4 The experiments

In order to highlight different aspects of the efficiency we have implemented the described experimental method in different experiments. We look at the distance computation algorithms both isolated and compared with each other. As a result we hope to present the selected algorithms differences in performance and accuracy.

4.4.1 Data reduction

In our first experiment, we look at the data reduction properties of the q-gram distance. The results we want to find are how much the q-gram distance reduces the dataset in the first stage of the approximate search procedure. A larger reduction (i.e. a smaller dataset in the second stage) is obviously better for performance as the exhaustive search will have fewer records to search. The goal is not only to see how much data that can be removed from the exhaustive search, but also how well the q-gram distance does this compared with the ordinary edit distance and the constrained edit distance. A general diagram of this experiment can be seen in figure 21.

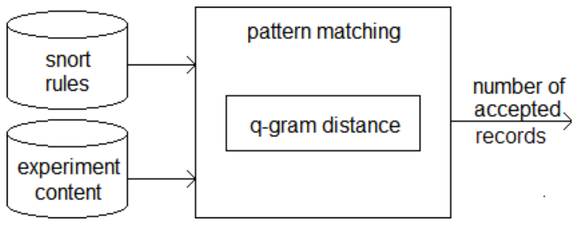


Figure 21: Data reduction experiment

In the experiment we compare each input item with each rule in the signature database (i.e. the Snort *web-misc.rules*) using the q-gram distance for pattern matching. If the distance is less than or equal to the threshold $N - M + \Delta$ we will, as described earlier, accept the rule for further inspection. We count the number of comparisons that the pattern matching accepts as items for the reduced dataset. This count is used to compute the data reduction compared with the original signature database. This experiment is repeated for all values of q and all values of Δ in order to see the reduction for different variations of the algorithm and with different acceptance threshold. It is also repeated for the ordinary edit distance and the constrained edit distance (with all selected variants of deletion constraint F) to find the difference in data reduction properties.

4.4.2 Accuracy comparison

Another experiment tests accuracy of the approximate search methods implementing q-gram and edit distance. As we have seen, the ordinary edit distance and the constrained edit distance have been successfully applied in [43]. In this experiment, we look at how accurate q-gram distance is for pattern matching when compared with these two algorithms. We use the ordinary edit distance and the constrained edit distance as the “correct” answer to what should be accepted in the first stage, and measure the q-gram distance’s accuracy accordingly. If the q-gram distance can provide results similar to the ordinary edit distance and the constrained edit distance we can use previous theory to see the q-gram distance’s accuracy and data reduction capabilities. A general diagram of this experiment can be seen in Figure 22.

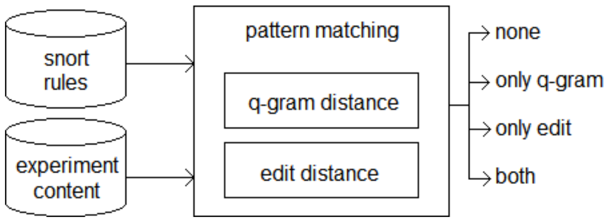


Figure 22: Algorithm accuracy comparison experiment

In the experiment we compare each input item with each rule in the signature database using the q-gram distance and compare the result with the edit distance. As with the previous experiment, the rule will be accepted if the q-gram distance is less than or equal to the threshold $N - M + \Delta_q$. Furthermore, the same input item and rule is also matched using the edit distance with its independent threshold value $N - M + \Delta_e$.

We use the Hamming distance as a metric for comparing the algorithms. If both algorithms (using the q-gram distance and the edit distance) accept or reject the rule (i.e. have the same result) the comparison will add 0 to the Hamming distance. If one algorithm accepts while the other does not (i.e. have different results) the comparison will add 1 to the Hamming distance. After comparing all input items with all rules we have the Hamming distance between the vectors corresponding to the algorithms, telling us for how many comparisons they differ. This allows us to see how similar results the algorithms produce with regard to data reduction and which rules they accept.

The experiment is repeated for all combinations of the q-value, and the threshold values Δ_q and Δ_e. Furthermore, it is also repeated for the q-gram distance compared with the constrained edit distance for all values of q, deletion constraint F, threshold value Δ_q and threshold value Δ_e.

After using all the variations of the experiment we are able to tell if the q-gram distance can for any q, F, Δ_q or Δ_e provide similar results as either the ordinary edit distance or the constrained edit distance.

4.4.3 Performance comparison

The previous experiments have looked at the data reduction and accuracy properties of the algorithms. The data reduction increases performance in the exhaustive search phase. However, we also wish to find the raw performance of the different algorithms in the first stage in order to reduce the overall execution time of the entire two-stage procedure. In this experiment we look at the difference in performance between the different pattern matching algorithms used in the first phase. A general diagram of this experiment can be seen in Figure 23.

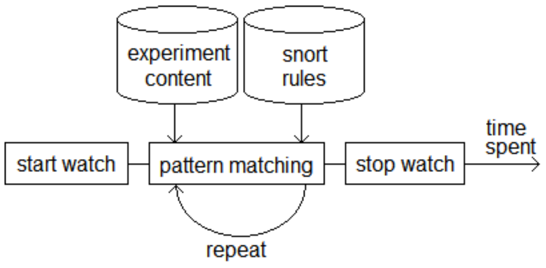


Figure 23: Algorithm performance comparison experiment

In this experiment, each input item is matched against each rule in the signature database and the execution time used for the comparisons is measured using a “stop watch”. Before we start the process of comparing all input with all the rules we store the current timestamp of the system. After storing the timestamp we start execution of the pattern matching algorithm for all inputs and rules as described earlier. Once all input items are compared with all rules we repeat this process 19 more times so each input item and each rule is compared 20 times in total. The reason for this is that we wish to reduce random noise (e.g. unfortunate process scheduling) that may influence the results. After finishing the processing 20 times the new current timestamp is stored. The difference between the timestamps represents the elapsed time in the processing interval and tells us how much time was spent processing. Since we repeat the pattern matching 20 times we divide the result by 20 in order to get the average runtime. This experiment is repeated for the ordinary edit distance, the constrained edit distance, and for the q-gram distance for all previously chosen values of q.

Since we are only interested in the performance difference between the algorithms, we only run the pattern matching on the data without checking whether rules are accepted or rejected. After the moment at which the distance is computed, the rest of the search algorithm is the same for all the pattern matching algorithms and as a result will not vary in execution time between the algorithms, and can therefore be discarded from our experiment.

The problem with short strings is also taken into consideration by removing strings shorter than 3 characters (our biggest q-gram size) from the dataset. The reason for this is that all algorithms should process exactly the same amount of data, and since the q-gram distance cannot process these strings we need to remove them from our benchmark so that the q-gram distance does not get an unfair advantage by skipping these. Furthermore, all data is preloaded into the memory so that the IO performance of the test environment should not slow down the algorithms. It is a well known fact that IO is a bottleneck in modern computer systems, and we do not want the performance of the algorithms to be slowed down by waiting for IO operations. For the q-gram distance, all the q-gram profiles for all the rules are pre-computed (as this is one of the performance benefits), but the input data's q-gram profile is computed once during the processing. Furthermore, we will use the compact q-gram profiles so that we can keep all q-gram profile in memory coupled with the rules they represents. We believe that this will well represent the way the q-gram distance would be used in an intrusion detection setting and consequently will give us the most illustrative results.

4.5 The environment

Many intrusion detection products, like Snort, run as software on regular computers. For our experiments we use a computer with components that should have about the same processing power and resources as current computer systems. The major components and software versions are listed below:

- AMD X2 3800+ 2.0 GHz, Dual Core processor
- 2 GB of internal memory
- 74 GB Western Digital Raptor 10 000 RPM hard drive
- Windows XP Pro SP2 (32 bits) with all current updates
- .NET Framework version 3.5

Both the speed of the processor, amount of internal memory, and the speed of the harddrive are properties that will influence the performance of the algorithms. It is therefore important to keep in mind that these components should roughly represent a normal computer system by today's standards. Windows XP SP2 with .NET Framework 3.5 should also represent the current software versions (i.e. XP being the most popular OS at the time of this writing and .NET 3.5 being the latest .NET Framework being released on windows update). During the experiments, no other applications are running except built-in services and drivers. All of the experiments have been implemented using the C# language running on the .NET Framework with no additional debugging information attached to the processes.

4.6 Reliability

Since our method of choice for this thesis is to implement different algorithms and compare their properties, we must make sure the algorithms work as intended. If the algorithms do not work correctly, our results and our conclusions will be wrong as a consequence. In order to make sure we have the maximum level of reliability, every algorithm is implemented as described formally according to their original papers or papers describing improvements and applications for the algorithms. In addition, we have used the Visual Studio.NET debugging capabilities to step by step make sure each statement and control sequence within the algorithm implementation behave as intended with a given input and an expected output. Furthermore, we have used the algorithm implementations on test data described in the papers and compared the results with the expected results (i.e. reimplemented previous experiments described in previous papers). We have also tested the algorithms with small data items that can, and have been, verified manually. This should all help making sure that our experiments and results have the required level of reliability.

4.7 Expectations

Based on the properties of the different pattern matching algorithms we have certain expectations for the results based on the known theory. First and foremost we know that the q-gram distance has successfully been used to estimate the edit distance as a lower bound. We believe that, based on this theory, it is reasonable to believe that the q-gram distance can be used in our method as a means for approximate search. However, there has been no such comparison with regards to the constrained edit distance. Furthermore, the q-gram distance is said to have a time complexity of $O(n + m)$ while the edit distance's time complexity is $O(nm)$ so it is safe to expect that q-gram will be faster than the edit distance. Considering that the constrained edit distance is built upon many of the same principles as the edit distance, we also expect that the q-gram distance should be faster than the constrained edit distance.

5 Results

In this chapter we present the results from our experiments. Because of the large amount of results generated from all the variations of the algorithms in our experiments (in some cases several hundreds) we have to limit the amount of data to present to only the best and the worst cases in some of the experiments.

5.1 Data reduction

In our data reduction experiment we wanted to see how many comparisons we could discard from exhaustive search in the second phase of the two-stage approximate search algorithm. The more comparisons we can remove from the search, the better the performance of the entire search algorithm. As a result, a higher data reduction is better for approximate search with regards to performance. We have looked at the data reduction properties for all three algorithms, but our primary focus is on the q-gram distance. The edit distance and the constrained edit distance are included as a reference. The data reduction R is the percentage of the number of accepted input/rule comparisons a out of the total number of input/rule comparisons c:

$$R = (1 - \frac{a}{c}) * 100 \tag{5.1}$$

In Figure 24 we can see the amount of data that is accepted by the q-gram distance, and thus selected to be searched using the exhaustive search in the second stage of our approximate search algorithm. What we can see here is that using a low threshold value Δ will give us a 99.3% data reduction for q-grams of size 3, and 99.2% data reduction for q-grams of size 2, in the first stage. This means that only 0.7% and 0.8% of the original data needs to be inspected by a slow, exhaustive, search algorithm in the second stage. By increasing the threshold, the amount of data will be slightly increased. With a threshold of 2 and 3, q-grams of size 2 and 3 will accept 4.9% and 4.2% of the original dataset for finer inspection in the second stage. For q-grams of size 1, the results differ a bit. For a low threshold value (0 and 1), 23.9% of the original data remains in the second stage. This is only a 76.1% data reduction. Compared with the 99.3% reduction for q-grams of size 3, this is not very good. It becomes even worse when increasing the threshold to 2 and 3. At this point, 55.5% of the original dataset is accepted for the second stage, thus only giving a 49.5% data reduction. The unsatisfactory results from q-grams of size 1 may be related to the fact that when using 1 letter q-grams we are only counting characters, thus making the filter much more inaccurate. However, despite the fact that q-grams of size 1 give a rather low data reduction, the larger q-gram sizes give us a high data reduction. This should be considered a good result and thus should improve the performance of the exhaustive search in the second

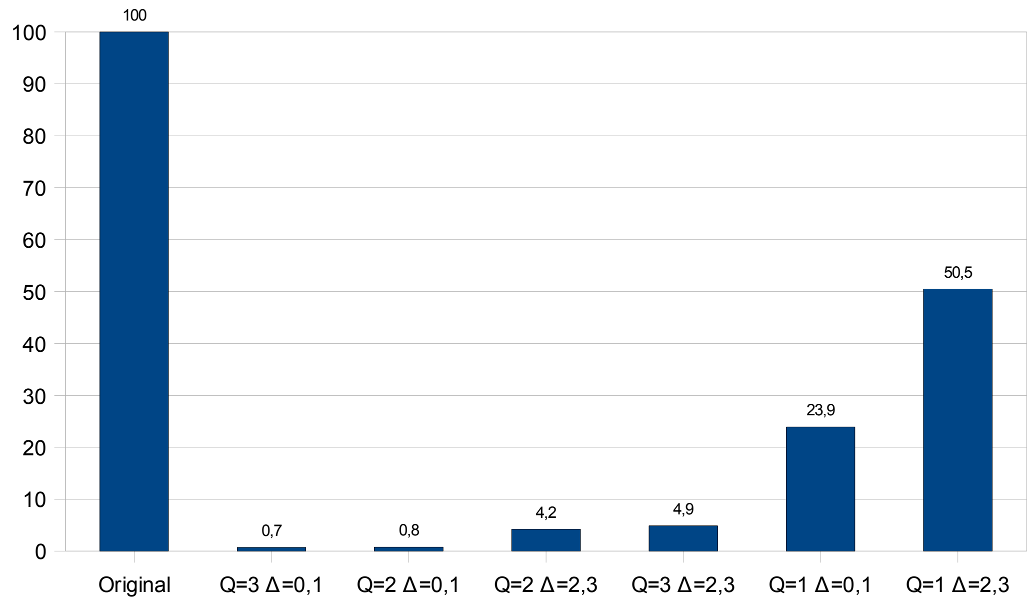


Figure 24: Data needed to be searched with exhaustive algorithm after data reduction

stage significantly. Another observation that can be made from this figure is that two and two threshold values give the same amount of data reduction.

However, although the data reduction is high, the results from the q-gram distance looks slightly different when compared with the other distance measures. In Figure 25 we can see the amount of data that is removed during the first stage for all the different distance measures at different threshold values. Here we can clearly see the data reduction properties of the different measures compared with each other. The q-gram distance has the highest data reduction of all the measures at low threshold values, which is good. The q-gram distance gives a 99.7% data reduction at this threshold, while the constrained edit distance and the ordinary edit distance follow with 98.4% and 91.9% respectively. Considering that all distance measures accept all the items that would be a match when using an exact match, the q-gram distance provides the best data reduction at this threshold. However, as we increase the threshold, and by this expect more data to be accepted and thus get a lower data reduction, the q-gram distance keeps a high data reduction. The ordinary edit distance and the constrained edit distance, on the other hand, have a steady decrease in the data reduction as we increase the threshold. As we can see from Figure 25, the constrained edit distance and the ordinary edit distance will give us a data reduction of 63.7%–71.3% (depending on F) and 40.9% when using 3 as our threshold value. The q-gram distance, on the other hand, will at the same threshold give us a 95.1%–95.8% data reduction depending on the size of the q-grams.

The point of the threshold value is to allow data with more errors to be accepted, and thus a behavior like the ordinary edit distance and the constrained edit distance seems desirable. One theory is that the q-gram distance is more strict, and thus could achieve the same behavior by

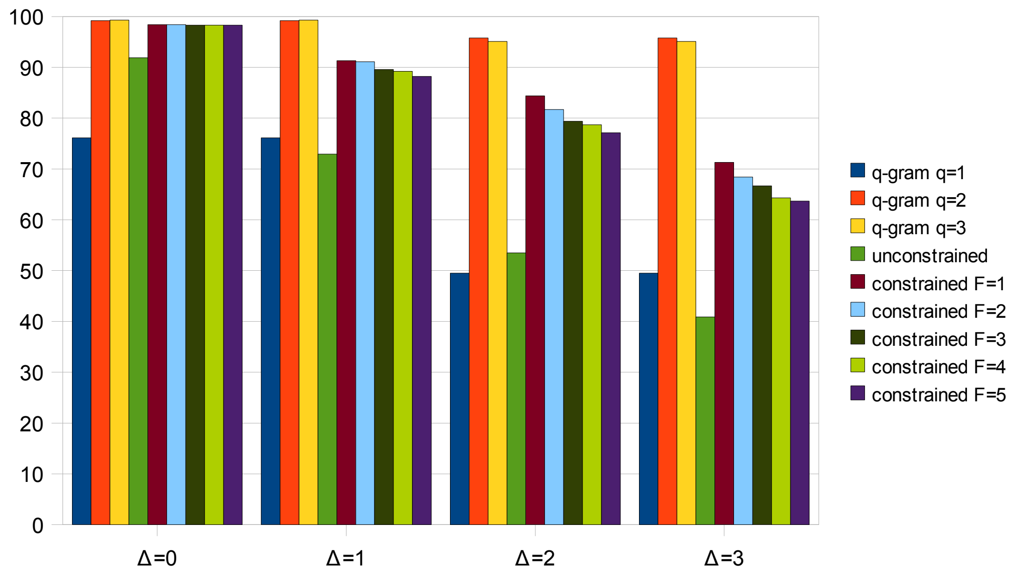


Figure 25: Data reduction comparison

increasing the threshold value further beyond 3.

In Figure 26 we have increased the threshold value from 3 to 15 for the q-gram distance (with q-gram size 2 and 3). As we can see, the q-gram distance will, like the ordinary edit distance and the constrained edit distance, accept much more data. If we compare Figure 26 with Figure 25 we can see that the q-gram distance using the threshold 8 or 9 is roughly comparable with the constrained edit distance using the threshold 3. This further strengthens our belief that the q-gram distance is stricter than the other two distance measures. This may be a positive property as this will allow the q-gram distance to be adjusted more fine grained than the other measures, as each step on the threshold scale will have a smaller impact on the amount of data to be accepted.

As we can see from the figures, all three algorithms provide a significant data reduction. In all cases the threshold value is a key variable that significantly influences the size of the data reduction. Unfortunately, the q-gram distance is not as good as the other two algorithms at scaling with the threshold value Δ. What this means is that when increasing the threshold value to allow a more fault-tolerant search, the q-gram distance becomes more strict than the other algorithms, and accepts far less input/rule comparisons for further inspection. This can, however, be adjusted by increasing the threshold even further, and as a result get a reduction behavior that is more similar to the constrained edit distance and the ordinary edit distance. In the case of the q-gram distance, the q-value also seems to have an impact on the results, especially for q-grams of size 1 compared with higher values. Although 40% to 60% data reduction results (i.e. the worst cases from all algorithms) may appear to be bad, this still means that the second stage of our intrusion detection approximate search algorithm will have to search 40% to 60% less data, and may therefore still be significantly better than searching the entire dataset. Unfortunately, although

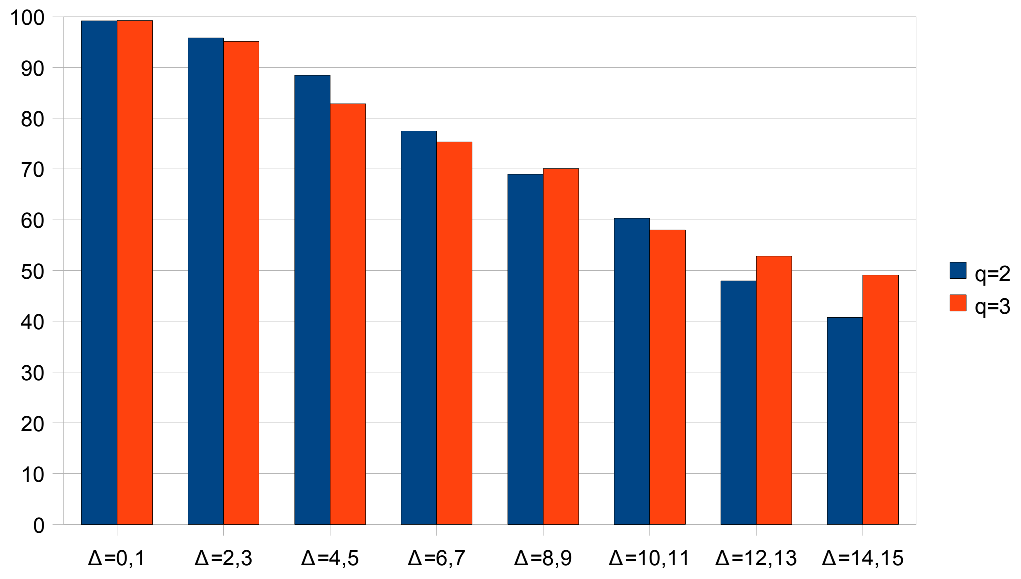


Figure 26: Data reduction for the q-gram distance with higher threshold

the data reduction properties will benefit the performance, this fact says little about the accuracy of the algorithms. For fault-tolerant search we want to accept signatures that resemble a match with no more than Δ errors, so the highest possible data reduction will not necessarily mean that all similar matches will be accepted. As a result it may be beneficial to find a compromise where we trade part of our data reduction (and therefore the performance) for better accuracy.

5.2 Accuracy comparison

In our accuracy comparison experiment, we wanted to find out how well the q-gram distance could be used to accurately estimate the edit distance and the constrained edit distance for approximate search in misuse detection. The accuracy in this case is defined as the number of input/rule comparisons that give the same result when using the q-gram distance as when using the ordinary edit distance and constrained edit distance. In this experiment we looked at for how many such input/rule comparisons the q-gram distance gives a different result compared with the other two distance measures. The error percentage is defined as the percentage of the number of accepted input/rule comparisons that differ d_H (i.e. the Hamming distance) between the two distance measures out of the total number of input/rule comparisons c:

$$E = (1 - \frac{d_H}{c}) * 100 \tag{5.2}$$

5.2.1 Constrained edit distance

First we look at how accurate the q-gram distance is when used in pattern matching compared with the constrained edit distance. In Table 1 we can see the results of the comparison between the different variations of the q-gram distance and constrained edit distance. Since all the variations would give us a table with hundreds of different entries we have chosen to look only at the best results for each variation of the constrained edit distance, considering that this is the measure we wish to estimate using the q-gram distance.

In Table 1 we see all the different variations of the constrained edit distance (i.e. all variations of the threshold value Δ_e and the F constraint) and which variations of the q-gram distance (i.e. variations of the q-gram size and threshold value Δ_q) can provide the most similar results. Each entry in Table 1 represents one of the different constrained edit distance variations with the q-gram distance variation that gives the best result when comparing the q-gram distance with this particular variation of the constrained edit distance. The first two columns, q and Δ_q, represent the parameters used in the q-gram distance variation that best estimated the current constrained edit distance variation. The next two columns, Δ_e and F, represents the parameters the current constrained edit distance variation uses. The last column is the percentage of the input/rule comparisons that give a different result when using the q-gram distance. An entry in the table can thus be read as: "the q-gram distance with q-grams of size q and a threshold Δ_q can be used to estimate the constrained edit distance with deletion constraint F and threshold Δ_e, with no more than E percent errors". In Figure 27, these results are presented visually as the lowest percent of errors we can achieve when using the q-gram distance instead of the constrained edit distance.

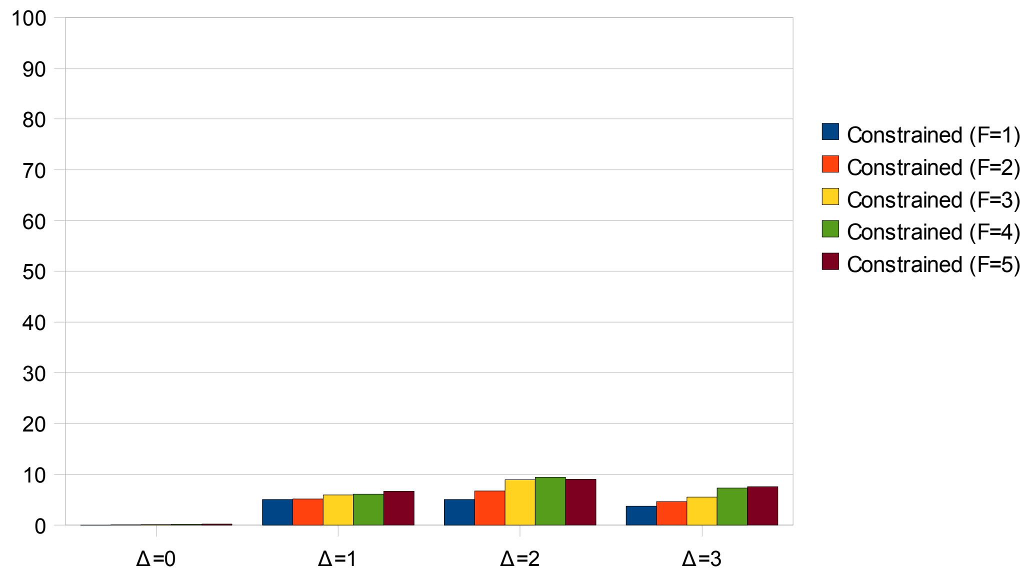


Figure 27: q-gram/constrained edit distance accuracy comparison differences

What we can see from Table 1 is that for any threshold value Δ_e and deletion constraint F, the constrained edit distance can be replaced by at least one variation of the q-gram distance with less than 10% errors (i.e. less than 10% input/rule comparisons with different result). If we look at the best result, we can see that this is achieved when using a threshold value $\Delta_e = 0$ and when using large q-grams. In this case only 6 out of more than 43 000 input/rule comparisons were wrong compared to the constrained edit distance. This is no more than 0.014% of the total amount of input/rule comparisons. As observed during our data reduction experiment, we need a higher threshold value Δ_q than the threshold value Δ_e in order to achieve a good estimate of the constrained edit distance when using the q-gram distance. As an example we can see that when using $\Delta_e = 3$ we need to set the q-gram distance's threshold value $\Delta_q = 8$ in order to get the best estimate.

In addition to the results in Table 1, which represent only the best cases, we have som general statistics about the rest of the q-gram/constrained edit distance comparisons The best results were achieved when we used low threshold value Δ_e. In the complete results table, 20 of the q-gram/constrained edit distance combinations only differed in less than 0.025% of the input/rule comparisons. Furthermore, 40 of the q-gram/constrained edit distance combinations only differed with less than 5% errors. In addition, more than 150 of the q-gram/constrained edit distance combinations managed to differ in less than 10% of the input/rule comparisons. An interesting observation from the results is that only q-grams of size 2 and 3 would provide the best estimates. In the complete results table, q-grams of size 1 were dominating the results with the highest amount of errors compared with the constrained edit distance. This does not come entirely as a surprise since we know that the q-gram distance with q-grams of size 1 is just a matter of counting single letters, and after seeing the results (Figure 25) in our data reduction experiment. Since q-grams of size 1 already in the data reduction experiment appeared to be inaccurate, and allowed a lot more data to be accepted compared with the other distance measures, it is certainly no surprise that it differs too much from the constrained edit distance.

5.2.2 Ordinary edit distance

In comparison with the constrained edit distance we saw that the q-gram distance in many cases could produce similar results as the constrained edit distance with only a few errors. In addition to the constrained edit distance, we compared the q-gram distance with the ordinary edit distance to see whether or not the q-gram distance could be used to estimate the ordinary edit distance. In Table 2 we can see the results of the comparison between the different variations of the q-gram distance and ordinary edit distance. As with the q-gram/constrained edit distance accuracy comparison, we have also chosen here to only look at the best results for each variation of the ordinary edit distance, considering that this is the measure we wish to estimate using the q-gram distance.

In Table 2 we see all the different variations of the ordinary edit distance (i.e. all variations of the threshold value Δ_e) and which variations of the q-gram distance (i.e. variations of the q-gram size and threshold value Δ_q) can provide the most similar results. Each entry in Table 2 represents one of the different ordinary edit distance variations with the q-gram distance varia-

q	Δ_q	F	Δ_e	E %
3	0	0	1	0,014
3	0	0	2	0,060
3	0	0	3	0,118
3	0	0	4	0,150
3	0	0	5	0,208
2	2	1	1	5,060
2	2	1	2	5,152
3	2	1	3	5,949
2	4	1	4	6,118
2	4	1	5	6,679
2	4	2	1	5,035
3	4	2	2	6,752
3	4	2	3	8,929
3	6	2	4	9,433
3	6	2	5	9,068
3	6	3	1	3,758
2	8	3	2	4,659
2	8	3	3	5,506
2	8	3	4	7,308
2	8	3	5	7,583

Table 1: Accuracy comparison between q-gram and constrained edit distance (best results)

tion that gives the best result when comparing the q-gram distance with this particular variation of the ordinary edit distance. The first two columns, q and Δ_q, represent the parameters used in the q-gram distance variation that best estimated the current constrained edit distance variation. The next column, Δ_e, represents the threshold value that the current ordinary edit distance variation uses. The last column is the percentage of the total number of input/rule comparisons that gave a different result when using the q-gram distance. An entry in the table can thus be read as: "the q-gram distance with q-grams of size q and a threshold Δ_q can be used to estimate the ordinary edit distance with threshold Δ_e, with no more than E percent errors". In Figure 27, these results are presented visually as the lowest percent of errors we can achieve when using the q-gram distance instead of the the ordinary edit distance.

What we can see from Table 2 is that unlike the q-gram/constrained edit distance comparison, the q-gram distance has quite a few errors when estimating the ordinary edit distance. Although the ordinary edit distance with a threshold value Δ_e can be estimated with less than 6.6% errors, an increase in the threshold value will significantly increase the number of errors. For all the variations of the ordinary edit distance with a threshold greater than 0, the percent of errors is between 16%–19%. This is almost twice as many errors as the worst case when comparing the q-gram distance with the constrained edit distance. However, we can see that although the results are worse than the comparison with the constrained edit distance, there are still q-grams of size 2 and 3 that gives the best estimate. Furthermore, as with the constrained edit distance comparsion, also when compared against the ordinary edit distance, the q-gram distance needs to increase the threshold value Δ_q significantly in order to best estimate a much lower threshold

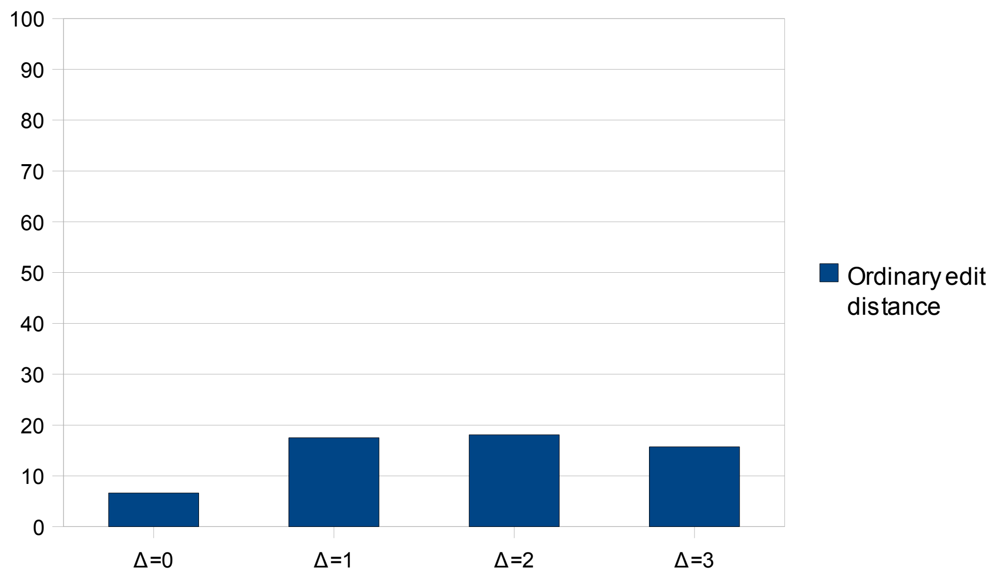


Figure 28: q-gram/ordinary edit distance accuracy comparison differences

Δ_e for the ordinary edit distance.

q	Δ_q	Δ_e	E %
3	2	0	6,598
2	8	1	18,756
2	10	2	18,065
2	14	3	15,697

Table 2: Accuracy comparison between q-gram and ordinary edit distance (best results)

5.3 Performance comparison

In our previous experiments we compared the data reduction and accuracy properties of the algorithms. However, the known advantage of the q-gram distance for pattern matching is its time complexity. In our performance comparison experiment we wanted to find the difference in performance between the distance measure algorithms for our particular use case: intrusion detection. In this experiment we measured the running time of each distance measure algorithm when comparing all input data with all rules. This was done 20 times for each algorithm so that any noise, like unfortunate processor scheduling, would not influence the results significantly.

Table 3 and Figure 29 present the total elapsed time and the average elapsed time for all algorithms during the experiment. In the Table 3, each of the different variations (q-gram size and deletion constraint F) are treated as individual algorithms. The threshold value is of no importance in this context as this will influence all the algorithms equally and are therefore not a part of the benchmark. Since the threshold computation is the same for all algorithms and is built in the same manner on top of the actual distance measure algorithms it is of no importance for highlighting the differences in running time between the algorithms. The table is sorted in ascending order with the shortest total elapsed time and shortest average elapsed time at the top. The elapsed time is presented as minutes, seconds, and milliseconds, which should provide a sufficient level of detail. From Figure 29 we can see that the q-gram distance is, as expected, significantly faster than the other algorithms. In the best case where we use q-grams of size 1, the average elapsed time for comparing all inputs with all rules is only 30 milliseconds. Even when increasing the size of the q-grams to 3 we use on average less than a second for all comparisons. In comparison, the ordinary edit distance uses more than 10 seconds on average while the constrained edit distance uses a minute more than the other algorithms on average. Based on these results and previous theory there should be no doubt that q-gram holds an enourmous advantage over the other algorithms for real-time approximate pattern matching when using typical intrusion detection rules as input.

Name	Total	Average
q-gram distance ($q = 1$)	00:00,696	00:00,034
q-gram distance ($q = 2$)	00:02,417	00:00,120
q-gram distance ($q = 3$)	00:14,134	00:00,706
Edit distance	03:32,510	00:10,625
Constrained edit distance ($F = 1$)	23:18,723	01:09,936
Constrained edit distance ($F = 2$)	24:05,748	01:12,287
Constrained edit distance ($F = 3$)	24:54,024	01:14,701
Constrained edit distance ($F = 4$)	25:39,548	01:16,977
Constrained edit distance ($F = 5$)	27:04,763	01:21,238

Table 3: Performance comparison results

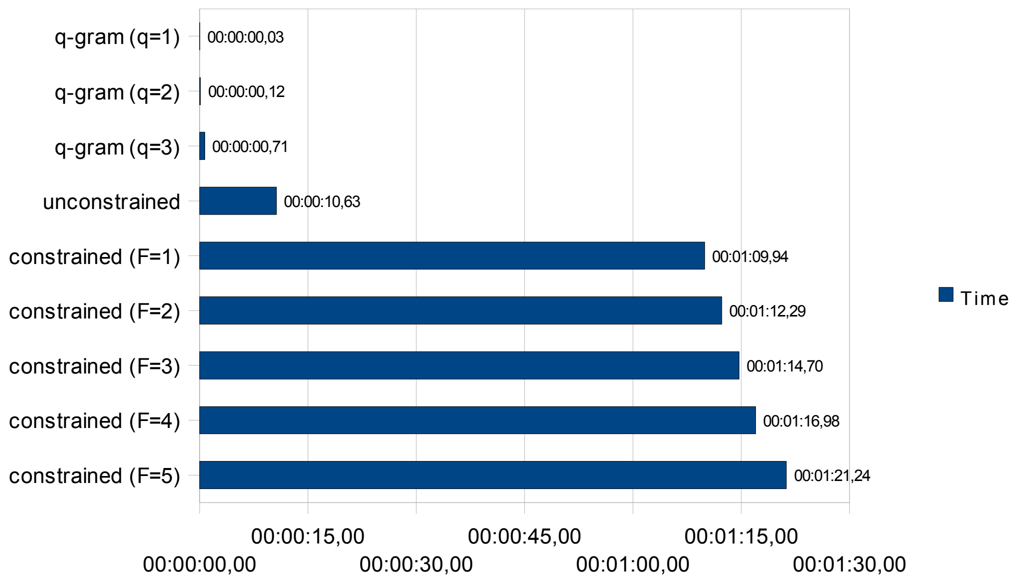


Figure 29: Performance comparison results

6 Discussion

In this chapter we summarize and discuss the results and possible explainations for the results found in our experiments. The different experiments look at different aspects of the algorithms that are all important for the feasibility of our approach for approximate search in misuse detection-based intrusion detection.

6.1 Performance

In our approximate search algorithm we used the q-gram distance as a means for gaining better performance in an exhaustive search within the dataset. Performance is vital in any intrusion detection system, and any problems in processing data fast enough will lead to reduced reliability (i.e. false negatives).

In our data reduction experiment we wanted to find out how large data reduction we could expect from the q-gram distance used in the pattern matching algorithm in the first stage of the two-stage search procedure. A larger data reduction will allow the exhaustive search algorithm to process a smaller dataset, and as a result lead to a huge performance gain in the second stage. What we found was that all the algorithms, both the q-gram distance and our reference algorithms, provided a significant data reduction. The amount of data that needs to be searched in the second stage of the algorithm, after the data reduction using the q-gram distance, can be seen in Figure 24. What we can see is that the amount of data that needs to be searched by the exhaustive algorithm is significantly decreased. Previous work [42, 43] has shown that the constrained edit distance provides better data reduction properties than the ordinary edit distance. In our experiment we could see that the q-gram distance at low threshold values and for large q-gram sizes would provide an even larger data reduction than the constrained edit distance, and especially the ordinary edit distance. However, when we increased the threshold value (i.e. tried to make the algorithms accept more data and therefore have a smaller data reduction), the q-gram distance would not scale at the same rate as the other algorithms. However, by increasing the threshold value even further we were able to get the same data reduction as the other distance measures. This may indicate that the q-gram distance is much more strict (i.e. rejects more items) compared with the other distance measures. However, by increasing the threshold value for the q-gram distance far beyond the threshold values used in the other two algorithms we actully managed to achieve the same effect, and thus the same amount of data reduction. The possibility of scaling the dataset using the threshold value is useful when we decide how tolerant the intrusion detection system should be for errors. To see that the q-gram distance actually needs a higher threshold for reaching the same level of accuracy as the other algorithms makes us believe that the threshold value for the q-gram distance can be adjusted more fine grained compared with the other two distance measures, thus giving us more control of what is accepted and what is not. This is, of course, yet to be proven as there is no guarantee that the data reduction procedure actually removes the correct items by just looking at the amount of data alone.

Unlike the data reduction experiment where we wanted to improve the performance in the second stage of the search algorithm, our performance comparison experiment looked at improving the performance in the first stage. Since we already have algorithms that can perform the data reduction properly, the performance in the first stage is the core of our thesis. The q-gram distance is already known for its linear time complexity of $O(n + m)$ so we wanted to know exactly how much performance gain we could expect. We found that the q-gram distance could outperform the other algorithms by a significant amount of time (i.e. matter of seconds) using our IDS dataset. In our experiment we managed to check 124 input data items in about two thirds of a second (710 ms) on average using the q-gram distance in the worst case. The same amount of data took almost 10.5 seconds using the edit distance. To further refine the results we see that the q-gram distance for $q = 3$ can process more than 190 input items on average while the edit distance can process only 11 items on average in the same time period. To make matters even worse, the constrained edit distance spend more than a minute on average to compare the 124 input items against our ruleset. What this tells us is that the q-gram distance is superior to other algorithms if we only look at the raw performance alone and not the accuracy. The constrained edit distance, which has shown to have excellent data reduction properties, was much slower than the alternatives. This can perhaps be explained partially by the fact that both the ordinary edit distance and the q-gram distance have been subject to much research regarding optimizations during the last decades, while the constrained edit distance computation algorithm as defined in [42, 43] is a rather new algorithm that has been proposed just recently. The research regarding this algorithm has focused only on the accuracy, so it is possible that this algorithm has a disadvantage over the other algorithms regarding the performance optimizations. We have used the reference implementation used in the previously discussed accuracy experiments [42, 43]. It should also be noted that we did not use any index structures to further optimize the performance. We do not know how implementation of suitable index structures would affect the results. It could be especially interesting to to see whether or not such index structures would make our reference algorithms perform closer to the q-gram distance, or if this would make the difference even greater.

6.2 Accuracy

Although the advantage of the q-gram distance is its low time complexity and therefore high performance, the method is useless if it has a low accuracy. It is difficult to make a formal definition for which rules it is correct to accept and which should be rejected in order to have high accuracy in approximate search. The whole concept is that similar items should be recognized, but we don't have an absolute definition of what characteristics should be present in order to be "similar enough". In our accuracy comparison experiment we used our reference algorithms as the basis for our accuracy definition. We compared the results from the q-gram distance with both of these algorithms to see how much they differed. We experiment under the assumption that the ordinary edit distance and the constrained edit distance is the “correct” way to do approximate search and that the q-gram distance should perform as close to these as possible in terms of accuracy. In our results we could see that for large q-grams and low threshold values, the q-gram distance could provide the same results as the constrained edit distance with only a few errors (only 6 out of 43 000 comparisons disagreed). In 20 of the cases the q-gram distance could be used to estimate the constrained edit distance with less than 1% errors, in 40% of the cases with less than 5% errors, and in more than 150 of the cases with less than 10% errors. This indicates that the q-gram distance could actually be used instead of the constrained edit distance in some very important cases. If we only look at only the best results when trying to estimate each variation of the the constrained edit distance we see that the q-gram distance actually can give the same results with less than 10% errors for all variations of the constrained edit distance. This is a really good result, and is a clear indication that the q-gram distance has potential as a replacement or addition to the constrained edit distance.

For the ordinary edit distance, the results were not quite as good. In the best case the q-gram distance and the edit distance would differ in about 6.6% of the cases. This was a bit disappointing considering that the q-gram distance is known to estimate the edit distance well. This may be a result of the method we used to check the distance against the threshold value ($d \leqslant N - M + \Delta$) since this was designed for the ordinary edit distance and constrained edit distance, and may not be as suitable for the q-gram distance. One observation was that for higher values of q, the q-gram distance was more strict than either of the other algorithms, and most errors were items that the other algorithms accepted and the q-gram distance rejected. When using lower values for q on the other hand, the q-gram distance would become very inaccurate (since its only counting occurrences of single letters) and accept much more items than any of the other algorithms. Another observation we made during our experiments was the fact that all algorithms would accept every comparison that would match if we had used exact pattern matching instead of approximate search. This is a desirable property considering that the algorithms will not wrongly reject perfect matches.

In our experiment we have used real intrusion detection rules as our data, and used input data we knew would be accepted for at least one rule each. We believe that this makes our experiment more reliable since we know each input item should be accepted at least once, and should be rejected by most other rules. Considering we have used real IDS rules (Snort) for our signature database, we also believe that our results should reflect real life usage. However, we have only looked at one ruleset (i.e. the *web-misc.rules*), but our results should not differ that much for other rulesets from SNORT as the structure of the rules remains the same with only the actual content field replaced. However, our experiments works only with SNORT rules, and it is difficult to say whether or not we would achieve the exactly same results when using another IDS as our case study. However, considering that SNORT is one of the most commonly used intrusion detection systems, we still believe that our results should represent a typical IDS environment.

6.3 Approximate search algorithm

We have looked at both the performance properties on the first and second stage of our two stage approximate search, and compared the accuracy of our pattern matching algorithm with the algorithms it is supposed to replace. As we can see, the q-gram distance will in terms of performance do an excellent job, but with regards to the accuracy it will vary depending on which distance measure we compare against and their provided variables. In some cases the q-gram distance will produce almost the same accuracy as the constrained edit distance, but much faster. In the cases where the q-gram distance is tweaked (using the q-value and the threshold value) to accept a lot of the rules for further inspection, the benefit of using the q-gram distance will be directly linked to the performance of the exhaustive algorithm in the next phase. The question is whether or not the time penalty of matching the input with all the rules in the first phase is less than it would be to just use exhaustive search on the rules that are filtered. If we can achieve a 50% data reduction (which was the case when using q-grams of size 1), then the exhaustive algorithm needs to spend at least twice as much time processing the data compared with the q-gram distance for it to be beneficial. In such cases perhaps we could extend our two-stage approximate search with yet another stage. In this case the q-gram distance could first filter the dataset, then followed by the constrained edit distance or the ordinary edit distance, which again filters the data before an exhaustive search in the last stage. As a result the exhaustive search would only have to search the data that has been accepted by both the q-gram distance and the edit distance.

A point that should be mentioned is the obvious possibility of distorting the data to such an extent that the distance becomes too large for the approximate search to accept, and as a result avoid the exhaustive search. By inserting enough extra data into the data packet, the distance will be too great and thus not considered similar enough for a more thorough search. However, this is not a flaw that exists exclusively when using q-gram as our distance measure for fault-tolerant pattern matching. As long as we even use a distance metric for fault-tolerance, this will be a possible problem, regardless of the choice of distance measure. However, how easy the approximate search algorithm is to circumvent is dependent on the threshold value. A higher threshold value will allow more distorted attacks to be subject for further inspection. At this point, it is dependent on the exhaustive algorithm whether such attacks are detected or not. As a result, to which extent such distorted attacks will be accepted or not is really a matter of finding the best balance between a high data reduction and sensitivity (i.e. adjusting the threshold). Furthermore, our approximate search algorithm will to a larger extent, compared with exact search, find such distorted attacks, which is the whole point of approximate search in IDS. Compared with the other distance measures like the ordinary edit distance and the constrained edit distance, our data reduction and accuracy experiments indicates that the q-gram distance would perform similar to these with regards to such distorted attacks.

We believe that our approximate search algorithm using the q-gram distance will be feasible considering that the q-gram distance would perform more than ten times faster than the edit distance and even more compared with the constrained edit distance on average, while still managing to achieve a large data reduction. Furthermore, considering that the q-gram distance could, in comparison with the constrained edit distance, achieve a rather good accuracy, this further strengthen our belief that this is a feasible distance measure for approximate search in IDS. However, the concept of using the q-gram distance as a measure for approximate search is still in its initial phase and should probably be subject to more research before implementing it in a corporate environment.

7 Conclusions

As time goes by, more and more attacks towards computer systems are discovered. All of these attacks must be added to the intrusion detection systems in order for them to be detected when using a misuse dection-based IDS. The increase in the signature database size will also increase the amount of data that needs to be searched every time the IDS receives activity data. Adding fault-tolerant search techniques to the IDS may allow new variations of known attacks to be detected without a pre-defined entry in the signature database. However, such techniques are often time consuming, and this adds complexity to the already time constrained search procedure.

In this thesis we have looked at how we can apply the q-gram distance as a means for distance measure in approximate search for misuse detection-based intrusion detection systems. We have described a two-stage search procedure where the first stage uses a fast q-gram distance based pattern matching for filtering the signature database. In the second stage the filtered dataset can be further inspected with a slower, exhaustive, algorithm. The first stage will reduce the dataset by removing rules that it finds to differ too much from the input data for it to be accepted in the second stage. A large data reduction will increase the performance of the second stage as the amount of data that needs to be inspected will be significantly reduced. Furthermore, by applying the fast q-gram distance as a distance measure instead of the ordinary edit distance and the constrained edit distance, the performance in the first stage will be significantly reduced, thus making the overall performance of the two-stage procedure much better.

In our experimental work we looked at exactly how efficient the q-gram distance was as a measure for distance in our approximate search algorithm, compared with the ordinary edit distance and the constrained edit distance. In our first experiment we could see that we could achieve a high data reduction using the q-gram distance and large q-grams. We could also see that by increasing the threshold value of the q-gram distance, we could achieve about the same data reduction as the other distance measures for any of the tested thresholds. In our next experiment we looked at the accuracy of the q-gram distance compared with the other distances. This is probably our most significant result since we could see that in some very important cases the q-gram distance would only differ from the constrained edit distance in 6 out of almost 43 000 input/rule comparisons (i.e. 0.014%). Furthermore, for 40 of our variations of the algorithms, it would differ in less than 5% of the cases. This makes us believe that the q-gram distance can for many variations of the algorithms computational parameters be used as an improvement over the constrained edit distance for approximate search in intrusion detection. Unfortunately, we did not get the same significant results when we compared the q-gram distance with the edit distance when using our algorithm. As a result we believe that the q-gram distance is most suitable for replacing the constrained edit distance as a distance measure in the first phase of our approximate search algorithm. This impression was further enhanced in our third experiment, the performance comparison, since our implementation of the q-gram distance was able to compare all the input data with all the SNORT rules in about $\frac{2}{3}$ of a second on average, while the implementation of the constrained edit distance used more than a minute. Our results indicate that the q-gram distance will perform the search many times faster than the constrained edit distance. At the same time it can in many cases estimate the constrained edit distance very well for misuse detection. As a result we believe that the q-gram distance could actually replace the constrained edit distance as a measure for finding the distance in approximate search for intrusion detection.

8 Further work

In this thesis we have seen results that indicate that the q-gram distance can be used as a distance measure for approximate search in misuse detection-based intrusion detection systems. The use of approximate search in IDS is definitely an interesting topic, and q-gram distance may be a solution for the performance problems associated with such search methods. However, although our research looks promising, there are many areas that should be looked further into when using q-gram distance as an approximate search distance measure.

Perhaps the most interesting topic to look further into when using the q-gram distance for IDS is to see exactly which records are accepted and which are not. From this, one could manually map which input should really match which rules and as a result see exactly how accurate all the different variants of the distance measures really are in our approximate search. This is, however, a rather tedious task considering that it would involve manually inspecting tens of thousands of input/rule comparisons.

In our experiments we have used rules from the SNORT intrusion detection system as our test dataset. A suggestion for further work could be to look at whether or not the q-gram distance would provide the same results when using rules from another intrusion detection system than only SNORT. This could be further enhanced by testing real network traffic or other well known IDS test-datasets (e.g. KDD cup 99 [52]) as input data. Another suggestion is to try another programming language/architecture for the experiments. Considering that we have used the C# programming language running on the .NET Common Language Runtime (CLR) during our experiments, it could be interesting to see how well the algorithms perform on the same language/-platform as real intrusion detection systems. An even more interesting variant of this suggestion would be to implement the approximate search in the analysis engine of an actual IDS. We also briefly mentioned that the q-gram distance was incapable of processing short strings. This is also a topic that could be studied further, and especially the accuracy of the approximate search when using our previously mentioned suggestions for avoiding the problem (i.e. padding the data, or using another algorithm for short strings).

To gain a better overview of the performance of the entire two-stage approximate search algorithm, an improvement to the performance experiments could also be to include the second stage (i.e. an exhaustive search algorithm) in the benchmark. This would allow us to see how fast our search is, not only the first stage using the q-gram distance, but also how fast the second stage is, given the data reduction of the q-gram distance from the first stage. It could also be interesting to see how index structures would influence the performance results in the first stage, and whether or not such structures could increase or decrease the difference in performance between the q-gram distance and the reference algorithms. Examples of such index structures could be to implement a MBT [47] or a V-tree [34] as briefly mentioned in the state of the art chapter.

Bibliography

[1] Arboleda, A. F. & Bedón, C. E. April 2005. Snort diagrams for developers. `http://afrodita.unicauca.edu.co/~cbedon/snort/snortdevdiagrams.html`.

[2] Abbes, T., Bouhoula, A., & Rusinowitch, M. 2004. On the fly pattern matching for intrusion detection with snort.

[3] Flensburg, F. 2001. String matching algorithms description. `http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/`.

[4] Zobel, J. & Moffat, A. 2006. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 6.

[5] Sagot, M.-F. 1998. Spelling approximate repeated or common motifs using a suffix tree. In *LATIN '98: Proceedings of the Third Latin American Symposium on Theoretical Informatics*, 374–390, London, UK. Springer-Verlag.

[6] Faloutsos, C. & Christodoulakis, S. 1984. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4), 267–288.

[7] Ptacek, T. H. & Newsham, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[8] Bugtraq mailing list, archives and vulnerability data base are available at security focus. `http://www.securityfocus.com`.

[9] Roesch, M. 1999. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, 229–238, Berkeley, CA, USA. USENIX Association.

[10] Bace, R. G. 2000. *Intrusion detection*. Macmillan Technical Publ., Indianapolis, Ind.

[11] The open-source intrusion detection system snort. `http://www.snort.org`.

[12] Check point intrusion prevention. `http://www.checkpoint.com`.

[13] Tcpdump/libpcap. `http://www.tcpdump.org/`.

[14] Fisk, M. & Varghese, G. 2001. Fast content-based packet handling for intrusion detection.

[15] Sedgewick, R. 1997. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[16] Aho, A. V. & Corasick, M. J. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6), 333–340.

[17] Baeza-Yates, R. 1999. *Modern information retrieval*. ACM Press, New York.

[18] Goodrich, M. & Tamassia, R. 2001. *Data structures and algorithms in Java*. Wiley.

[19] Wirth, N. 1986. *Algorithms & data structures*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] Sedgewick, R. 1984. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[21] Sedgewick, R. 1992. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[22] Knuth, D., Morris, J., & Pratt, V. 1977. Fast pattern matching in strings. *SIAM J. Computing*, 6(2), 323–350.

[23] Black, P. E. Dictionary of algorithms and data structures. `http://www.nist.gov/dads/`.

[24] Boyer, R. S. & Moore, J. S. 1977. A fast string searching algorithm. *Commun. ACM*, 20(10), 762–772.

[25] Horspool, R. N. 1980. Practical fast searching in strings. *Software-Practice & Experience*, 10(6), 501–506.

[26] Sunday, D. M. 1990. A very fast substring search algorithm. *Commun. ACM*, 33(8), 132–142.

[27] Zobel, J., Moffat, A., & Ramamohanarao, K. 1998. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4), 453–490.

[28] Carterette, B. & Can, F. 2005. Comparing inverted files and signature files for searching a large lexicon. *Inf. Process. Manage.*, 41(3), 613–633.

[29] Knuth, D. E. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[30] Knuth, D. E. 1997. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[31] Weiner, P. October 1973. Linear pattern matching algorithms. In *IEEE 14th Ann. Symp. on Switching and Automata Theory*.

[32] McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2), 262–272.

[33] Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3), 249–260.

[34] Shi, F. & Mefford, C. 2005. A new indexing method for approximate search in text databases. In *Proceedings of the The Fifth International Conference on Computer and Information Technology*, 70–76. IEEE Computer Society.

[35] Landau, G. M. & Vishkin, U. 1988. Fast string matching with k-differences. *J. Comput. Syst. Sci.*, 37(1), 63–78.

[36] Lu, M. & Lin, H. 1994. Parallel algorithms for the longest common subsequence problem. *IEEE Trans. Parallel Distrib. Syst.*, 5(8), 835–848.

[37] Allison, L. & Dix, T. I. 1986. A bit-string longest-common-subsequence algorithm. *Inf. Process. Lett.*, 23(6), 305–310.

[38] Hamming, R. 1950. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2), 147–160.

[39] Damerau, F. J. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3), 171–176.

[40] Bard, G. V. 2007. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *Proceedings of the fifth Australasian symposium on ACSW frontiers - Volume 68*, 117–124, Ballarat, Australia. Australian Computer Society, Inc.

[41] Kurtz, S. 1996. Approximate string searching under weighted edit distance. In *Proceedings of the 3rd South American Workshop on String Processing*, Ziviani, N., Baeza-Yates, R., & Guimarães, K., eds, 156–170, Recife, Brazil. Carleton University Press.

[42] Petrović, S. & Franke, K. 2007. Improving the efficiency of digital forensic search by means of the constrained edit distance. In *Proceedings of the Third International Symposium on Information Assurance and Security*, 405–410.

[43] Petrović, S. & Franke, K. 2007. A new two-stage search procedure for misuse detection. In *Proceedings of the 2007 International Conference on Future Generation Communication and Networking (FGCN 2007)*, 418–422.

[44] Ukkonen, E. 1992. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92, 191–211.

[45] Navarro, G. 2001. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 31–88.

[46] Owolabi, O. & Mcgregor, D. 1988. Fast approximate string matching. *Software - practice and experience*, 18, 387–393.

[47] Shi, F. 2004. Fast approximate search in text databases. In *Advances in Web Age Information Management, Lecture Notes in Computer Science, LNCS*, 259–267. Proceedings of WAIM.

[48] Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., Pietarinen, L., & Srivastava, D. 2001. Using q-grams in a DBMS for approximate string processing. *IEEE Data Engineering Bulletin*, 24(4), 28–34.

[49] Mangnes, B. The use of levenshtein distance in computer forensics. Master's thesis, Gjovik University College, 2005.

[50] Krause, E. F. 1986. *Taxicab Geometry*. Courier Dover Publications.

[51] dtSearch Corp. dtsearch text retrieval products. `http://www.dtsearch.com/`.

[52] Stolfo, S. J., Fan, W., Lee, W., Prodromidis, A., & Chan, P. K. Kdd cup data set from the knowledge discovery and data mining tools competition. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

A Source code

In this chapter we present samples of the q-gram distance source code used in our experiments. All code is written using the C# programming language. The source code presented is the actual implementation of the concept pseudo-code presented in the approximate search chapter, and will therefore not be given a further explanation. For a description of the concept please refer to the approximate search concept chapter. Some of the source code samples have been slightly changed compared with our actual experiments in order for them to look better in this thesis. In case of the approximate search itself we have only included a sample from the q-gram data reduction experiment considering that this is the easiest source code of any of our experiments. This source code should still be sufficient to see an actual implementation of our approximate search concept.

A.1 Generating a q-gram profile

```
using System;

namespace ApproximateSearchIDS
{
    partial class Program
    {
        static int[] GQ(string s, int q)
        {
            const int z = 95;          //the last 95 characters are used in
                the original ascii table
            int size = Power(z, q);
            int[] vector = new int[size];

            int maxpow = q - 1;
            int n = s.Length - q;

            int x = 0;
            for (int j = 0; j < q; j++)
            {
                x += ((s[j] - 32) * Power(z, maxpow - j));
            }
            vector[x]++;

            for (int i = 0; i < n; i++)
            {
                x = ((x - (((s[i] - 32) * Power(z, maxpow)))) * z) + (s[i + q
                    ] - 32);
                vector[x]++;
            }
            return vector;
        }
    }
}
```

A.2 Finding the q-gram distance

```
using System;

namespace ApproximateSearchIDS
{
    partial class Program
    {
        static int DQ(int[] v1, int[] v2)
        {
            int distance = 0;
            for (int i = 0; i < v1.Length; i++)
            {
                int x = v1[i] - v2[i];
                distance += (x < 0 ? -x : x);
            }
            return distance;
        }
    }
}
```

A.3 Data reduction experiment

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace ApproximateSearchIDS
{
    partial class Program
    {
        class Summary
        {
            public int q, DeltaQ;
            public int count;
            public int accept_count;
        }

        static int MinDelta = 0, MaxDelta = 15;
        static int MinQ = 1, MaxQ = 3;

        static void Main(string[] args)
        {
            StreamReader srRecord, srSearch;

            List<Summary> summaryList = new List<Summary>();
```

```
//trying every combination of the q-gram distance with different
    thresholds
for (int DeltaQ = MinDelta; DeltaQ <= MaxDelta; DeltaQ++)
{
    for (int q = MinQ; q <= MaxQ; q++)
    {
        srSearch = new StreamReader("contents.txt");

        int searchNo = 0;

        int total_comparison_count = 0;
        int total_accept_count = 0;

        while (!srSearch.EndOfStream)
        {
            searchNo++;
            string search = srSearch.ReadLine();

            srRecord = new StreamReader("padded_rules.txt");

            int[] vector = GQ(search, q);

            int recordNo = 0;
            while (!srRecord.EndOfStream)
            {
                recordNo++;
                string record = srRecord.ReadLine();

                total_comparison_count++;

                int d = DQ(GQ(record, q), vector);
                int limit = record.Length - search.Length +
                    DeltaQ;

                if (d <= limit)
                {
                    total_accept_count++;
                }
            }
            srRecord.Close();
        }
        srSearch.Close();

        Summary sum = new Summary();
        sum.q = q;
        sum.DeltaQ = DeltaQ;
        sum.count = total_comparison_count;
        sum.accept_count = total_accept_count;

        //inserting the summary into a sorted list
        bool inserted = false;
        for (int i = 0; i < summaryList.Count; i++)
        {
```

```
                        if (sum.accept_count < summaryList[i].accept_count)
                        {
                            summary.Insert(i, sum);
                            inserted = true;
                            break;
                        }
                    }
                    if (!inserted) summaryList.Add(sum);
                }
            }
            writeSummary(summaryList);
        }
    }
}
```

B Dataset

In this chapter we present samples of the dataset used in our experiments. These data are stored in plain textfiles, with one entry on each line. Since the amount of data is far beyond what we can present here, we only present a very limited sample in order to give an impression of how the data looks like. In the case of the SNORT rules, these are presented unpadded. In the experiments, the character “#” was added at the end of each rule until they were of the same length as the longest rule in the dataset.

B.1 Unpadded SNORT *web-misc.rules*

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC cross
     site scripting HTML Image tag set to javascript attempt"; flow:to_server
    ,established; content:"img src=javascript"; nocase; reference:bugtraq
    ,4858; reference:cve,2002-0902; classtype:web-application-attack; sid
    :1667; rev:7;)
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 443 (msg:"WEB-MISC PCT
    Client_Hello overflow attempt"; flow:to_server,established; flowbits:
    isnotset,sslv2.server_hello.request; flowbits:isnotset,sslv3.server_hello
    .request; flowbits:isnotset,tlsv1.server_hello.request; content:"|01|";
    depth:1; offset:2; byte_test:2,>,0,5; byte_test:2,!,0,7; byte_test
    :2,!,16,7; byte_test:2,>,20,9; content:"|8F|"; depth:1; offset:11;
    byte_test:2,>,32768,0,relative; reference:bugtraq,10116; reference:cve
    ,2003-0719; reference:nessus,12205; reference:url,www.microsoft.com/
    technet/security/bulletin/MS04-011.mspx; classtype:attempted-admin; sid
    :2515; rev:14;)
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC /
    ecscripts/ecware.exe access"; flow:to_server,established; uricontent:"/
    ecscripts/ecware.exe"; nocase; reference:bugtraq,6066; classtype:web-
    application-activity; sid:1944; rev:3;)
```

B.2 Input data

```
ls%20-l
/etc/passwd
cd..
/....
cat%20
////////
/GWWEB.EXE
hdr=/
form=
template=../../../
HEAD/./
Authorization|3A|
icatcommand=
GET x HTTP/1.0
includedir=
pccsmysqladm/incs/dbconnect.inc
DELETE
secure_site, ok
b2inc
query=%00
/../../
whois|3A|//
/nstelemetry.adp
|EB|_|9A FF FF FF FF 07 FF C3|^1|C0 89|F|9D|
```