Memory Efficient Signature Matching in Deep Packet Inspection Applications at Line Rates

Shiva Shankar Subramanian

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr.-Ing. Georg Sigl

Prüfende der Dissertation:
1. Prof. Dr. sc.techn. Andreas Herkersdorf
2. Prof. Dr.-Ing. Ulf Schlichtmann

Die Dissertation wurde am 19.12.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 19.08.2019 angenommen.

Abstract

Deep Packet Inspection (DPI) is the process of inspecting all the headers and the payload in a network packet. Various content-aware networking applications such as intrusion detection/prevention, load balancing, copyright enforcement and content aware Quality-of-Service (QoS) drive the need for performing DPI in the next-generation networks. Comparing the payload bytes against the signatures, which are defined through string and regular expression patterns is the most time-critical function in DPI. So, it is essential to accelerate the signature matching function to perform DPI at multi gigabit rates ($\sim$10 Gbps) in modern embedded network processors.

The signatures or patterns which are used for payload inspection are either represented through the Non-Deterministic Finite Automaton (NFA) or the Deterministic Finite Automaton (DFA). The DFA is generally preferred to represent the signatures, especially for high speed signature matching applications as it is highly processing efficient in comparison to the NFA. On the other hand, the DFA is highly storage inefficient due to the presence of redundant state transitions and the exponential state blow-up corresponding to certain signature sets. So, the DFA is generally compressed before being stored in the memory, while the payload bytes are actually compared against the compressed DFA.

The transition compression algorithms play a significant role in defining the memory footprint of the compressed DFA, as well as in mandating the rate at which the signature matching function can be performed. The state-of-the-art transition compression algorithms either focus on effectively compressing the DFA [1, 2, 3] or focus on compressing the DFA in such a way that the signature matching can be performed at multi gigabit rates [4, 5], but not both. Addressing this shortcoming, two transition compression methods, the MSBT and the LSCT are proposed and evaluated in this dissertation which can effectively compress the DFA as well as perform the signature matching function at about $\sim$ 10 Gbps. The proposed methods efficiently compress the redundant transitions through a combination of bitmaps and bitmasks. Additionally, the linearity in the arrangement of state transitions in the DFA is further leveraged to compress the redundant bitmasks in the compressed DFA. The MSBT and the LSCT are capable of achieving transition compression rates of the order of over 98% primarily due to the introduction of bitmasks, which is 4-5% higher than the state-of-the-art bitmap based compression algorithms. The improvement in the transition compression rates and the bitmask compression reduces the memory footprint of the compressed DFA by $\sim$70% in comparison to the state-of-the-art transition compression algorithms.

A bitmap based signature matching engine called BiSME is proposed and implemented which performs the decompression corresponding to the MSBT. BiSME encompasses two efficient storage architectures, the *Packed Storage Methodology* and the *Shared Memory Methodology* to effectively store the compressed DFA in the on-chip memories in a configurable manner without resulting in memory wastage. The Bitmask Optimized BiSME (BOBiSME) extends the BiSME architecture to support bitmask compression further reducing the area requirements of BiSME. The BiSME and the BOBiSME were effectively pipelined to perform the signature matching function at 9.3 Gbps and 10.6 Gbps, respectively. The signature matching engines were designed and synthesized on the TSMC 28nm technology node and are capable of storing a maximum of 1000 string signatures. The BiSME and BoBiSME require 1.43 mm^2 and 1.2 mm^2 of silicon area and only consume 155 mW and 170 mW of power, respectively making them *area-efficient* and *power-efficient* hardware coprocessor architectures. The BiSME and the BOBiSME architectures were prototyped on the cadence palladium platform and network traffic traces of over 2 GBytes consisting of different traffic characteristics were injected to validate the hardware implementation.

Acknowledgments

This dissertation would not have been possible without the support and encouragement from a number of people to whom I would like to express my gratitude from the bottom of my heart.

First and foremost, I would like to thank Prof. Dr. sc. techn. Andreas Herkersdorf for agreeing to be my doctoral advisor. The profound technical discussions which I had with him greatly helped me to structurally formulate my research problems and propose innovative solutions during my research work. His extensive experience in working with industry based research programs enabled me to not only make quality contributions to the academic community, but also to conceive effective solutions which could be easily adopted by the industry. Additionally, I would like to thank him for accommodating me as a visiting researcher in his institute, which also allowed me to experience the research environment in TUM.

I would like to thank Dr. Pinxing Lin from Intel Technology Asia Pte. Ltd. for accepting to be my industry supervisor for the research program. He has always been the go-to person during all the challenging times and I would like to thank him for all the great discussions which he has had with me over these times. His calm demeanor always put me at ease and is a quality which I have always admired in him.

I would also like to thank Dr.-Ing. Thomas Wild for his technical insights during various stages of my research work. I really enjoyed the discussions which I had with him and I really appreciate his prompt and timely feedback over the course of the research project.

I would like to express my gratitude to Mr. Mario Traeber for believing in me and giving me an opportunity to be the first candidate for the pioneering Industry PhD Program (IPP). I would like to thank Ms. Ko Kah Goh and Mario for the enormous effort which they put in to define the organization of this program. I would also like to thank Mr. Bing Tao Xu, for supporting my career aspirations and allowing me to proceed with the IPP. My sincere thanks to Mr. Daniel Artusi for his continuous moral support throughout the course of the IPP. Finally, I would like to thank Intel Technology Asia Pte. Ltd., Singapore for funding this research program and providing me an opportunity to work on such a great research topic.

I would like to acknowledge the technical contribution of certain members within Intel without whose help this work would not have been possible. My thanks to Roshini John who setup the software simulations on automata models which helped me to get a deeper understanding on automata based signature matching. My thanks to Hariharan for helping me to prepare the synthesis setup for the hardware implementation of the signature matching engine which was developed in this research work. My thanks to Anmol Prakash Surhonne, Nihar Sriram and Sagar Paramesh for their support in the preparation of the evaluation setup to verify the hardware implementation. Additionally, I would like to thank all my colleagues in Intel Technology Asia Pte. Ltd who have helped me over the course of this program. My special thanks to Abishek Godaa for all the thought provoking discussions which he has had with me. Many thanks to Jyothi, Kavitha, Dharmesh, Seshagiri, Giridhar, Jincy, Rekha, Suresh Nagaraj, Suresh Kumar for being great friends and colleagues at work. Many thanks to all my colleagues in Institute for Integrated Systems in TUM who made my stay memorable in Munich. Especially, my special thanks to Michael Vonbun for all his help during my stay in Munich and during the course of the preparation of this dissertation. I would also like to thank Mrs. Doris Zeller, Mrs. Draga Verena and Mrs. Gabriele Spöhrle for all their secretarial support during the course of the program.

Last but not least, I would like to thank my family for their unconditional love and affection which gave me the strength to take up such a monumental journey. I would also like to thank my wife Varsha Sharma for her love. Her support and encouragement was in the end what made this dissertation possible. Knowing that these words are wholly inadequate, I express my deepest thanks to them for bearing with me, for being so constant and supportive, and for giving me their faith and trust. It is to them that I dedicate this dissertation.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The evolution of the Internet of Things (IoT) and the possibility to connect numerous electronic devices through a multitude of communication technologies is revolutionizing the way in which we interact with these devices. The ability to remotely monitor and manage these devices enable systems to make data driven decisions; further saving time for people and businesses to improve the quality of our lives. According to a report from Accenture [7], the evolution of the IoT devices is still in its nascent stages and the total number of connected devices is expected to hit about 40 billion by 2024. The growth in the number of IoT devices is spread across a multitude of environments such as industrial automation, wearables, transportation, infrastructure applications, home automation etc.

A wide range of IoT devices and applications are emerging for use in home, which include connected devices such as personal computers, mobile phones, tablets, smart televisions, gaming consoles, smart camera and many other smart appliances as shown in Figure 1.1. Though these devices bring additional convenience to our lives, the networking capability in these devices present cybercriminals an opportunity to exploit them [8]. Issues such as competitive cost, technical constraints and the time to market pressure challenge the manufacturers to adequately design effective security features into these devices. Distributed Denial-of-Service (DDoS) attacks such as mirai botnet [9] is an example of the devastation that could be caused by cybercriminals who capitalize on the always on networking connectivity in these devices. Additionally, the sheer growth in the number of IoT devices kindles their interest to target these devices. The connected devices in the home network, not only allow the attackers to execute complex attacks from the compromised devices, but also allow them to access precious user data within the devices and the other devices which are connected to the home network. Various studies by researchers have found that home appliances such as smart refrigerators or smart televisions are susceptible to be compromised [10, 11], and the biggest worry is that the end users are completely caught unaware of it. Cisco's annual security report states that the patching rate for these IoT devices is minimal and in most cases, the vulnerabilities are not even patched [12]. So, securing the IoT devices in the home network is a major issue which has to be addressed before converting our home, really into a *Smart Home*.

The Residential Gateway Router (RGR) is a centralized networking hub in the connected home. The RGR has also been the networking node which is used to manage the network communication between the network service provider and the devices connected to the network. However, due to the explosion in the number of connected devices in the

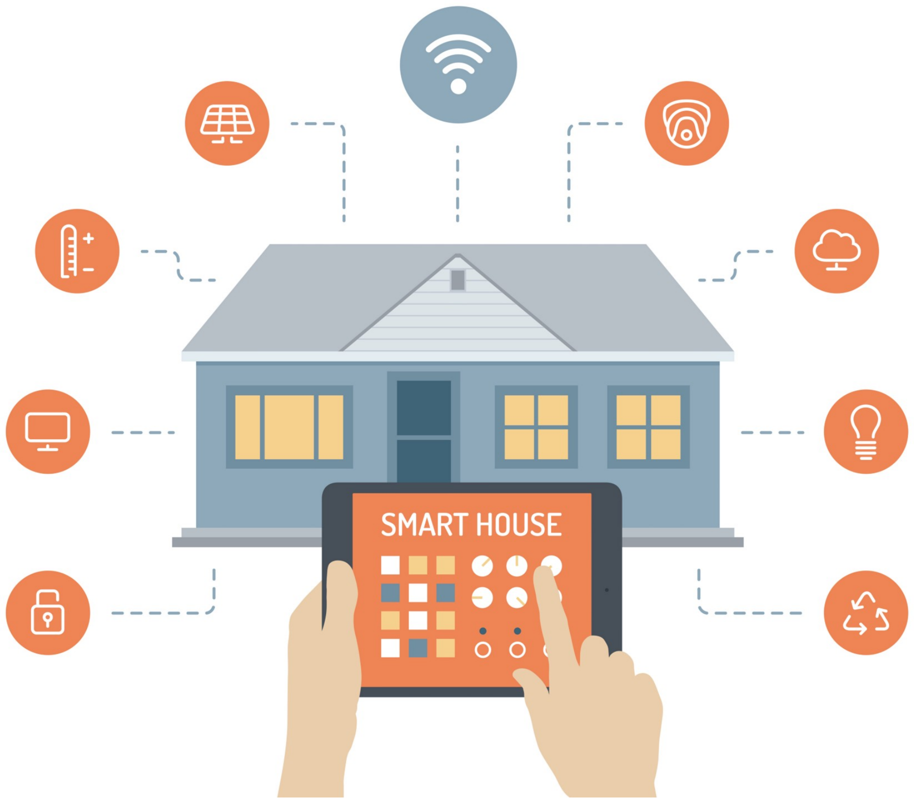


Figure 1.1: A smart home which has many IoT devices connected to it. (Image downloaded from [6]).

home network, the RGR now has to take the additional responsibility of securing and managing the devices connected to it. So, it is essential to implement Content-Aware Networking (CAN) services such as intrusion detection/prevention, load balancing, content aware Quality-of-Service (QoS), copyright enforcement etc. in the RGR, in addition to enabling connectivity with the network service provider. In this way, the RGR becomes the central networking device in the home network through which the various connected devices can be effectively secured and managed.

A Network Processor (NP) is the heart of the RGR, which is responsible to analyze the network packets which are generated by the devices connected to the home network. The data which is communicated in a network packet is generally classified into the header and the payload. The payload consists of the actual application data which is exchanged by the network devices, while the header information in a packet helps to identify the next-hop device to which the packet is sent to. In order to perform the packet forwarding function (switching/routing), the NP in the RGR generally inspects the packet header across various layers specified as part of the Open Systems Interconnection (OSI) specification [13]. However, in order to perform the content aware networking functions, it is essential to inspect the complete network packet including the headers at various layers and the content of the payload in a packet. So, it is essential to perform Deep Packet Inspection (DPI) in the NP, the technology which allows to completely inspect a network packet to enable content-aware networking in the next generation RGR.

A typical network processor which is used in the RGR consists of multiple wireline[1] and wireless network interfaces[2] through which it can enable communication between multiple connected devices. The incredible development in the wireline (2.5/5 Gbps Ethernet [17]) and the wireless (802.11ax [18]) communication technologies which are used in the home networking ecosystem are allowing devices to communicate at multigigabit rates, even within the home network. Moreover, the deployment of services such as fiber-to-the-home (FTTH) [19, 20], mandates the network processors used in the RGR to perform packet processing at multi-gigabit rates with the network service provider [21]. As more applications begin to leverage the available bandwidth in the home network, it is crucial to perform DPI at multi-gigabit rates to process packets in a content aware manner.

The next section provides a short introduction to various packet inspection methodologies and further describes the challenges associated in performing DPI.

1.2 Introduction to Deep Packet Inspection

1.2.1 Packet Inspection Methodologies - Overview

Figure 1.2 shows the various layers[3] which are inspected as part of network packet processing. Based on the depth of the layers until which a packet is inspected, the packet inspection methodology is classified into the *Shallow Packet Inspection* (SPI), *Medium depth Packet Inspection* (MPI) and the *Deep Packet Inspection* (DPI) [22].

The SPI and the MPI methodologies primarily inspect the packet header alone. The difference between the two methodologies arises depending on the specific header layers which are inspected in a packet. The packet headers corresponding to layers 2, 3 and 4 alone are inspected as part of the SPI, while the header information across all the header layers are inspected as part of the MPI. Various rules are defined to inspect specific portions of the headers in a packet, while the rules also consist of actions which have to be applied after a rule match. It should be noted that the location of the header fields which are inspected in a packet are known a priori, as the headers adhere to standard network protocols. So, the processing associated with respect to the SPI

[1]The current generation network processors in an RGR consists of 4 Ethernet ports each of which is capable of communicating at 10/100/1000 Mbps [14, 15, 16].

[2]Next-generation network processor architectures used for the RGR are enabled with multiple (typically 4) WiFi baseband processors which provide a consolidated wireless throughput of more than multiple gigabits ($\sim$6 Gbps) per second [14, 15, 16].

[3]According to the OSI model, the information in a network packet is primarily split into 7 layers; physical layer, data link layer, network layer, transport layer, session layer, presentation layer and the application layer. The physical layer forms the lowest level of abstraction while the application layer is highest. The physical layer represents the medium in which the data is communicated. So the header information is generally added for the other 6 layers and the packet inspection is also performed corresponding to these 6 layers.

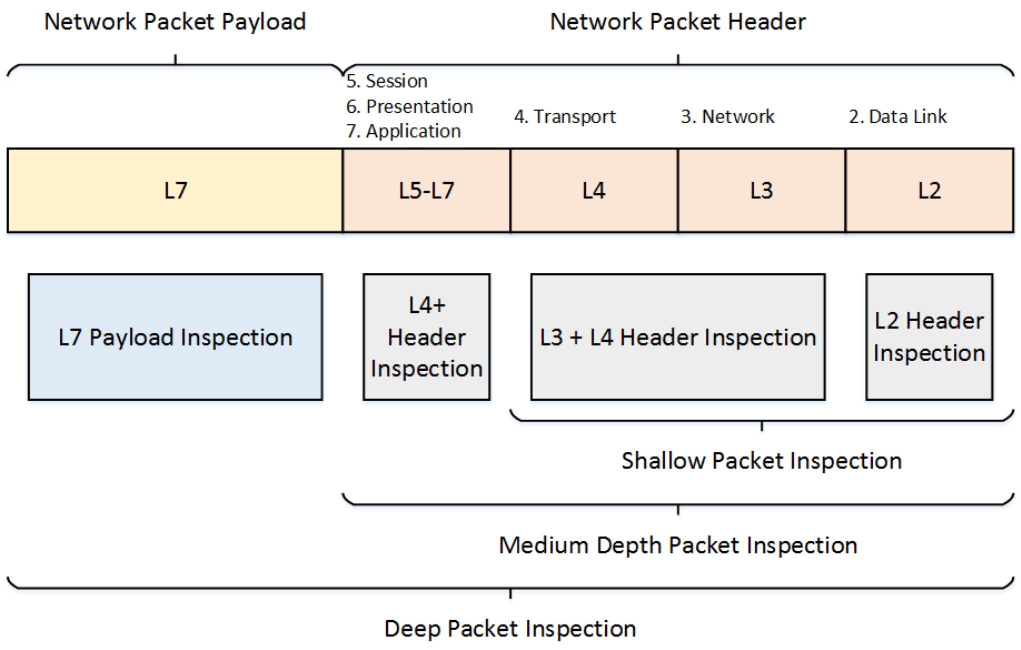


Figure 1.2: An overview of various packet inspection methodologies

and the MPI primarily involve searching for specific fields in the header whose location is known.

As shown in Figure 1.2, in the case of the DPI, the packet payload is also inspected in addition to the packet headers across various layers. So, the rules which are used for packet inspection in DPI consist of two parts. The first part consists of specific fields which have to match in the packet header. After the required fields in the packet headers match, a *signature* is searched in the payload portion of the packet. The signatures which are used for payload inspection are either composed of strings or regular expressions. Unlike the header inspection in which the location of the inspected portion in the header is known, the location of the signature which is being searched in the payload is unknown. Moreover, the signature can start at any byte location within the sequence of payload bytes and can even be split across the payload portions of multiple packets. So, all the payload bytes across multiple packets have to be sequentially inspected to check if the signatures are found within the payload byte sequence. Thus, the payload inspection is a computationally challenging task in DPI in comparison to header inspection.

1.2.2 Steps in Deep Packet Inspection

The various functions which are performed as part of DPI can be broadly classified into packet normalization, packet reordering, packet prefiltering, signature matching and the postprocessing [23]. Figure 1.3 shows a logical ordering of these functions and a short description of these functions is outlined below.

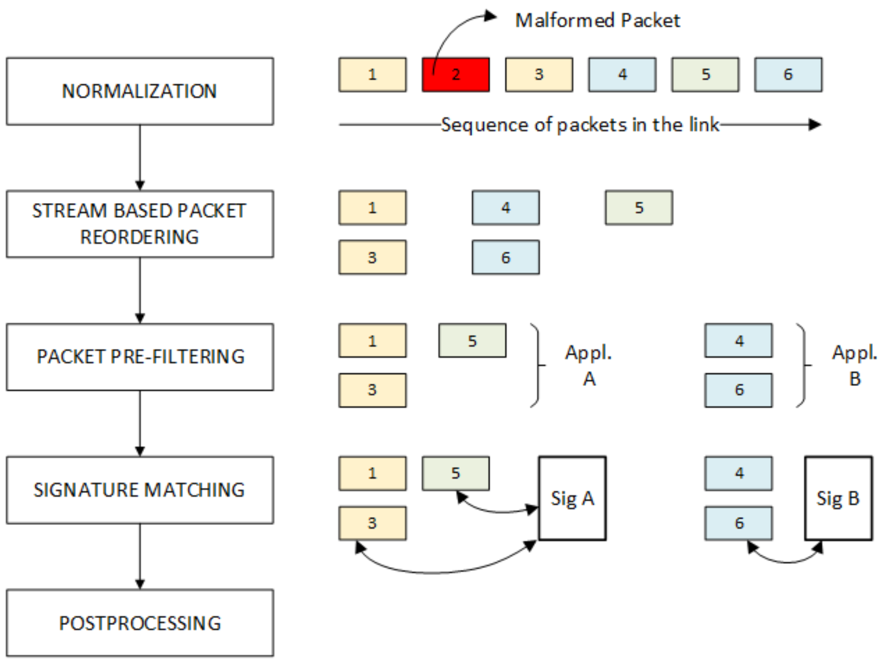


Figure 1.3: Overview of the various steps to be performed in deep packet inspection

- **Normalization:** Normalization [24] is the first step in DPI, in which various sanity checks are performed on a network packet before performing the payload inspection. Various researchers [24, 25] have identified that the network and the transport layer headers can be synthetically configured to evade the signature matching function. So, the main aim of normalization is to eliminate those malformed packets and to ensure that only those packets which comply to the network protocols are inspected.

- **Packet Reordering:** The packets which are processed by a network processor are generated by different network devices and have to be initially split into network packet streams before performing the payload inspection. Furthermore, the packets corresponding to a stream should also be reordered to make sure that they are inspected in the right sequence [26]. The normalization and the packet reordering functions have to be performed on each and every individual packet.

- **Packet Prefiltering:** Since the main aim of DPI is to enable content aware networking, the signatures are generally associated with a certain higher layer application protocol, e.g., HTTP, FTP etc. So, the signature database is also divided into subsets of signatures, where each of the subsets identify the signatures associated with a specific application protocol. The application protocol information is extracted after inspecting the header in the transport layer or the application layer headers. The extracted information can be used to compare the packet payloads against a signature subset, instead of comparing the payload bytes against all the signatures. The process of comparing the packet payload against a signature subset after inspecting the packet headers is called packet pre-filtering and has been used by various DPI implementations [27, 28, 29]. The packet pre-filtering is a one time step which is generally performed during the packet classification stages where a new stream is categorized into a specific higher layer protocol.

- **Signature Matching:** The signature matching is the most important step in DPI in which the payload bytes are compared against the signatures. In this step, each and every byte in the payload is compared against the database of signatures. Since there can be multiple signatures in a signature subset, it is essential to match the payload bytes against all the signatures at once to perform the function in a computationally effective manner. Moreover, since the application data is split into multiple packet payloads, there are chances that a signature is split across multiple network packets. So, it is essential to match signatures across multiple network packets. To summarize, the signature matching is the most computationally challenging task in DPI and is also the apt function for hardware acceleration to perform DPI at multi gigabit line rates [30]. So, considering the fact that the network traffic at such high rates have to be inspected by the network processors in the RGR, hardware acceleration of signature matching becomes quintessential to perform line rate signature matching within the home network [31].

- **Postprocessing:** Once a signature is successfully matched, the action corresponding to a signature is performed on the packet or the stream in this step. The action associated with a signature is generally described as part of its definition, while the criticality of the postprocessing function varies depending on the individual signatures.

To summarize, the signature matching function is the most time critical function which enables the network processors to perform line rate signature matching. The next section discusses the computational and storage challenges associated with the signature matching function.

1.2.3 Signature Matching - Requirements & Challenges

The signatures which are used for DPI are either represented through strings or regular expressions. Since the signatures cannot be directly processed by modern processor architectures, they are either converted into the Deterministic Finite Automaton (DFA) or the Non-deterministic Finite Automaton (NFA). Both these are machine readable state machine representations which are functionally equivalent to the signature set. Since the processing complexity of comparing a payload byte against the DFA is constant ($\mathcal{O}(1)$), it is the preferred form to represent the signatures, especially for high speed signature matching applications [32]. On the contrary, the DFA is highly storage ineffective due to the presence of redundant state transitions in it. Moreover, when the signatures are described through regular expressions, the total number of states generated in the DFA explodes exponentially and this problem is referred to as the state explosion problem [33].

So, the standard approach proposed by the academia is to compress and store the DFA in the memory and eventually perform the signature matching against the compressed DFA [34]. The state explosion problem during the DFA generation is addressed by performing state compression, while the redundant transitions are compressed through the transition compression algorithms. Since both these approaches address orthogonal problems, the transition and the state compression algorithms are orthogonal to each other [35].

The transition compression algorithms play an important role in defining the memory footprint of the compressed DFA [1]. Achieving high transition compression rates reduces the memory footprint of the compressed DFA. However, it is also important to make sure that the algorithmic approach towards transition compression enables the decompression to be performed in a dedicated hardware accelerator. This would enable the signature matching function to be performed at line rates which in turn allows to perform DPI at line rates.

The transition compression algorithms proposed in the literature either focus on efficiently compressing the DFA or focus on performing the decompression at line rates, but not both. The transition compression algorithms [1, 32, 36, 3] which belong to the former category achieve high transition compression rates typically of the order of over 95%. However, in these techniques, the redundant transitions are compressed at the cost of increased memory bandwidth which doesn't allow the decompression to be performed through dedicated hardware accelerators [4]. On the other hand, the transition compression algorithms which belong to the latter [4, 5] focus on compressing the DFA through the bitmaps. Compressing the redundant state transitions through the bitmap maintains the constant processing complexity as that of the DFA, which allows the decompression to be performed through dedicated accelerators. However, these algorithms require the bitmap to be stored together with the compressed DFA to identify the location of the compressed transition. So, in order to reduce the number of bitmaps stored together with the compressed DFA, the bitmap based compression solutions proposed in the literature compromise on the transition compression rates and only achieve ∼90-95%. Considering the strategic importance of transition compression, there is a requirement for a class of transition compression algorithms which can *effectively compress the DFA* as well as enable *hardware acceleration* of the decompression function to perform signature matching at line rates.

The signature matching throughput achieved by a hardware accelerator engine in the case of DPI applications completely depends on the organization of the compressed state transitions in the memory. As in the case of any computing system, the data access latency plays a big role in deciding the throughput that is achieved in the case of signature matching applications. The signature matching throughput achieved is the highest when the compressed DFA is completely stored in the on-chip memories as the data can be accessed at low latencies. However, if the compressed DFA is stored in a combination of on-chip and off-chip memories, the throughput of the hardware accelerator completely depends on the latency to fetch the data from the off-chip memories [37]. There is a possibility to reduce the access latencies through the introduction of cache memories as in the case of [37]. However, loading the specific portions of the compressed DFA into the cache memories can become highly volatile, as this completely depends on the characteristics of the payload bytes being inspected. Thus, a great amount of effort will be required to effectively design a caching algorithm to load the data into the cache memories. So, if the DFA is compressed efficiently in such a way that it is completely stored in the on-chip memories, the hardware accelerator can be effectively used to perform the signature matching function at line rates.

Considering that the hardware accelerator will be part of the network processor used in the RGR, the following requirements should be satisfied so that the architecture becomes reusable across multiple generation of systems.

- **Scalability:** The signature matching engine should be scalable to support increasing network bandwidth requirements and increasing signature counts. While there is no specific data that is currently available regarding the specific signature counts required for the RGR, the significant growth in the new malware [38] will require to support signature counts of the order of few thousands. The signature matching throughput to be supported by the engine also depends on the network bandwidth and the application scenarios. Considering that multiple network interfaces are available in the RGR which can support network bandwidth at multiple gigabits per second, the signature matching throughput to be supported will cross the 10 Gbps barrier in the future for the RGR.
- **Flexbility & Programmability:** When the compressed DFA is completely stored in the on-chip memories, it is essential to make sure that the data is effectively stored in the physical memories. Effective storage architectures should be proposed to store the compressed DFA in a flexible way so that the on-chip memory space is effectively utilized. Additionally, the access to the physical memory and sharing of physical memory space should be made programmable to introduce a certain level of flexibility with respect to compressed DFA storage. In this way, when the accelerator is integrated with a network processor, the processor can configure the functionality of the accelerator depending on specific application requirements.

1.3 Thesis Contributions

Addressing the challenges mentioned above, two transition compression methods are proposed in this dissertation which can efficiently compress the DFA and also enable the decompression to be performed in dedicated hardware accelerators. Following up on the compression algorithms, a flexible, scalable, programmable hardware accelerator is proposed which enables to perform the signature matching functions at ∼10 Gbps. The following are the key contributions of this dissertation:

- **Transition Compression Through Bitmaps & Bitmasks:** Though the bitmap based transition compression methods proposed in the literature perform the signature matching through dedicated hardware accelerators, they compromise on the transition compression rates to reduce the number of bitmaps stored together with the compressed DFA. Addressing this drawback, two bitmap based transition compression methods called the *Member State Bitmask Technique* (MSBT) and the *Leader State Compression Technique* (LSCT) are proposed in this dissertation. These methods achieve transition compression rates of about 97-98%, a 4-5% improvement over the state-of-the-art solutions [39, 40]. The redundant state transitions in the DFA are compressed through a combination of bitmaps and *bitmasks*, a secondary layer of indexing introduced in these methods. Though the bitmasks have to be stored together with the compressed transitions, the improvement in the transition compression rates reduces the memory footprint of the compressed DFA by 50% in comparison to the state-of-the-art bitmap based solutions. The compressed DFA generated after bitmap based transition compression mainly consists of the compressed transitions and the control data (bitmaps, bitmasks etc.), which enable to locate the compressed transitions. The main idea behind the proposed solutions is to add more control data in the form of bitmasks to effectively index the redundant state transitions. Thus, even a small improvement in the transition compression rate considerably reduces the overall memory footprint of the compressed DFA.

- **Optimizing the Memory Footprint of the Compressed DFA after the MSBT & the LSCT:** Since, the compressed DFA is intended to be completely stored in the on-chip memories after the MSBT and the LSCT, it is paramount to make sure that the memory footprint of the compressed DFA is well optimized. The following three methods are proposed to further improve the memory footprint of the compressed DFA.

  1. **Divide & Conquer State Grouping Method:** The state grouping is one of the integral steps in the proposed methods in which the states are grouped into subsets after which the redundant transitions are compressed. So, a compression-aware *Divide and Conquer* (DC) state grouping method is proposed for this step, through which the transition compression rates are improved by a variable factor of 0.5-2%, thus reaching overall compression rates of about 98-99% [41].

  2. **Bitmask Compression:** As part of the MSBT and the LSCT, the bitmask is generated for all the states in the DFA which enables to effectively compress the redundant state transitions after bitmap based compression. However, a majority of the bitmasks which are generated in this process are redundant and are compressed through the bitmask compression process. Experimental evaluation of the bitmask compression shows that about ∼60-70% of the bitmasks are redundant which are efficiently compressed through the proposed method.

  3. **Combining Alphabet Compression with Bitmap Compression:** Alphabet compression is a well known method which is used to compress those indistinguishable characters in an alphabet, i.e, the ASCII character set in the

case of DPI applications. The alphabet compression is proposed to be combined together with the MSBT and the LSCT, as the bitmap alone cannot compress certain redundant state transitions in the DFA.

All of the methods proposed above are orthogonal to each other and can be implemented together to optimize the memory footprint of the compressed DFA together with the MSBT and the LSCT. When compared with the state-of-the-art solutions, the overall memory footprint of the compressed DFA reduces by 70%, an additional improvement of 20%, when combined together with the MSBT and the LSCT.

- **Hardware Coprocessor for Signature Matching:** Utilizing the proposed methods to compress the DFA, a Bitmap based Signature Matching Engine called BiSME is proposed to perform the signature matching function in a dedicated accelerator to achieve line rate DPI [42]. BiSME is a *programmable*, *flexible* and *scalable* coprocessor which stores the compressed DFA in on-chip memories after performing the MSBT. The compressed DFA is flexibly stored in the on-chip SRAMs through various efficient storage architectures. The *Shared Memory Methodology* efficiently stores the dynamically varying compressed transitions, whose count is signature dependent, in the predefined on-chip memory boundaries. The *Packed Storage Methodology* enables to store the unstructured *bitmasks* flexibly, without wasting precious on-chip memory resources. The BiSME is effectively pipelined to achieve a signature matching throughput of 9.3 Gbps. Furthermore, an extension to BiSME called BOBiSME is proposed which implements the bitmask decompression in the hardware and is capable of performing signature matching at 10.6 Gbps. The proposed signature matching engines were synthesized on a commercial 28nm technology library operating at 0.81V. The BiSME consumes 1.43 mm^2 of silicon area and consumes 0.155W power while the BOBiSME consumes 1.18 mm^2 of silicon area and consumes 0.167W power. A compiler was designed to convert the signature sets into BiSME and BOBiSME memory formats. The functionality of the hardware implementation was thoroughly verified in the Cadence Palladium platform by injecting over 2 GB of identical traffic to BiSME, BOBiSME and a DFA based signature matching engine. The identical signature matching results verified the functional correctness of the hardware implementation.

1.4 Dissertation Organization

The dissertation is organized as follows. Chapter 2 first provides a detailed overview of the state-of-the-art signature matching engines and summarizes the advantages and disadvantages of the various methods proposed in the literature.

Chapter 3 proposes and evaluates the MSBT and the LSCT in which the bitmasks are introduced to significantly improve the transition compression rates in comparison to existing bitmap based solutions.

Chapter 4 proposes and evaluates three different methods, i.e, the Divide and Conquer state grouping method, the alphabet compression and the bitmask compression which focus on further reducing the memory footprint of the compressed DFA.

Chapter 5 first discusses the two flexible storage architectures through which the compressed DFA can be efficiently stored in the on-chip memories and then the resulting hardware engines through which the transition decompression is performed after compressing the DFA through the Member State Bitmask Technique. Furthermore, this chapter discusses the key results with respect to the achievable signature matching throughput and the further the functional evaluation of the proposed hardware accelerators.

Chapter 6 finally concludes the dissertation and discusses the directions for some of the future work.

2 State-of-the-Art

String and regular expression based signature matching is a problem which has been well addressed by the research community [34]. The signatures which have been used for DPI have greatly evolved over time. Initially, the signatures were primarily composed using simple string patterns. However, due to the high false positive rate observed during string based signature matching, Sommer et al., [43] proposed the usage of regular expressions to describe the signatures. Since the regular expressions offer better flexibility in comparison to strings, most of the applications prefer to use the regular expressions to define the signatures [44].

The signature matching engines have correspondingly evolved in relation to the evolution in the signature representations. Based on the type of signatures which are processed, the signature matching engines can primarily be classified into *string based* and *automata based* signature matching engines as shown in Figure 2.1 [34]. The string based signature matching engines are capable of processing string signatures alone, while the automata based signature matching engines are capable of processing both string and regular expression signatures. Due to this advantage, a majority of the research work has primarily focused on automata based signature matching engines [34].

Existing research work on automata based signature matching engines primarily focuses on two aspects, i.e, automata compression and line-rate signature matching implementations [34, 23]. This chapter provides an overview of solutions which focus on both of these aspects.

This chapter is organized as follows. Section 2.1 provides an overview of the various string based signature matching engines proposed in the literature. Section 2.2 provides an introduction to the theory of automata and the basics of automata based signature matching. Section 2.3 provides an overview of the various DFA compression approaches while Section 2.4 provides an overview of the state-of-the-art solutions which target the problem of line rate signature matching. Finally, Section 2.5 provides an overall summary of the state-of-the-art approaches with respect to automata based signature matching.

2.1 String Based Signature Matching Engines

The very first signature matching engines which were proposed in the literature were targeted to process string based signatures and are classified under the string based signature matching engines [34]. Initially, various classical string matching algorithms such as Knuth Morris Pratt (KMP) [45], Aho-Corasick (AC) [46], Boyer-Moore (BM) [47], etc., were used to match the payload bytes against the string signatures. Dharmapurikar et al., [48] identified that these algorithms are primarily tuned towards software based

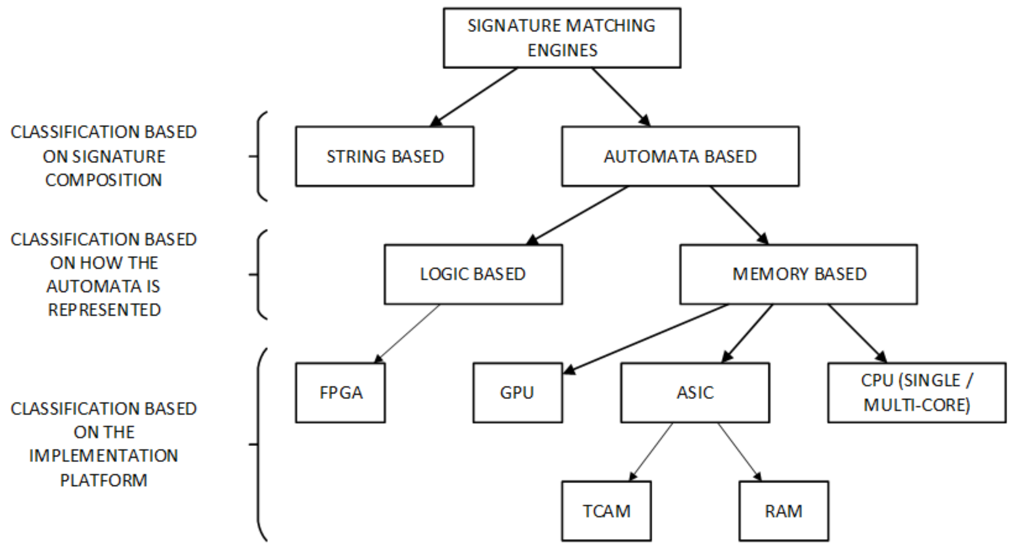

SIGNATURE MATCHING ENGINES
CLASSIFICATION BASED ON SIGNATURE COMPOSITION
STRING BASED
AUTOMATA BASED
CLASSIFICATION BASED ON HOW THE AUTOMATA IS REPRESENTED
LOGIC BASED
MEMORY BASED
CLASSIFICATION BASED ON THE IMPLEMENTATION PLATFORM
FPGA
GPU
ASIC
CPU (SINGLE / MULTI-CORE)
TCAM
RAM

Figure 2.1: Classification of various signature matching engines proposed in the literature

signature matching implementations and are not useful for line rate signature matching. So, they proposed to use the bloom filters [49] for string based line rate signature matching implementations. The bloom filter is a time and space efficient probabilistic data structure that stores a database of strings compactly in a memory vector. However, the result of the querying process using the bloom filter primarily indicates if an element (signature) is likely to be present in the bloom filter or not present, i.e., false positives are possible as part of the querying process but false negatives will never occur. So, a secondary source is required to confirm if the signatures were identified after the querying process.

The usage of bloom filters for signature matching involves a filter programming step, in which k different hash functions are used to set k different bits in an *m-bit* vector. For each of the signatures in the signature set, the hash functions generate a set of indices at which the bits in the vector are set to 1. As part of signature matching, the payload bytes are passed to the k different hash functions to generate indices at which the memory vector is queried. If all the indices in the memory vector are set, a signature is set to be identified within the payload byte stream. Since a single *m-bit* vector is used to store multiple signatures, the identified index could have possibly been set by a different signature other than the one that matched. So, a secondary analyzer is used to verify the signature matching results. However, the most important property of this data structure is that the computation time involved in performing the query is independent of the number of strings stored in the vector.

The very first bloom filter based signature matching methodology was proposed in [48], where multiple bloom filters in parallel were used to accelerate the string matching function. A hash based secondary analyzer was used to distinguish between a false-positive and a true positive. On the other hand, Nourani et al. [30] proposed a bloom filter based prefilter, to enable a first level filtering and a hash based hardware engine to perform signature matching at multi-gigabit rates. There were various other enhancements which were proposed using the bloom filter based architectures for signature matching following [48] and [30]. However, due to the emergence of regular expressions as the preferred mode to describe the signatures, the research community has primarily focused on automata based signature matching engines which are capable of processing both string and regular expression signatures [44, 34, 23, 50, 33]. The next section provides an introduction to the theory behind the automata and further describes the basics of automata based signature matching approaches.

2.2 Automata Based Approaches

2.2.1 Introduction to NFA & DFA

An automaton or a Finite State Machine (FSM) is a state machine which can understand the language expressed by a set of signatures comprising both string and regular expression signatures. To simplify, an automaton is an equivalent description of a signature set in the machine readable format. A finite automaton [51] is represented as a 5-tuple(Q, Σ, δ, q_0, F) as shown below:

- Q, is a finite set of states,
- Σ is a finite set of characters called alphabet,
- δ: $Q \times\Sigma \rightarrow Q$[1], is the state transition function,
- $q_0 \in Q$, is the initial state,
- $F \subseteq Q$, is the set of accepting states.

An automaton primarily consists of a finite set of nodes called the states, Q and labeled directed edges between the states which are called the state transitions. The labels in the directed edges correspond to either one or multiple characters in the alphabet, Σ. An automaton consists of a single initial state called the *root state*, q_0 from which the state machine starts the signature matching function. The state transition function, δ takes a state and a character[2] as an input and generates a single or multiple next states as the output. In an automaton, F is the subset of all the states in Q, which identify a signature match and is referred to as the set of *accepting states.*

As part of the signature matching, the root state is first assigned as the current state. Subsequently, each and every payload byte is input to the state transition function which computes the next state transition corresponding to a current state and payload byte combination. The computed next state is assigned as the current state and the

[1]In the case of the Non-Deterministic Finite Automaton, the state transition function generates a set of next states P(Q) corresponding to a state-character combination, i.e, δ: $Q \times\Sigma \rightarrow P(Q)$

[2]As part of the dissertation, the terms character and payload byte are used interchangeably to represent an individual payload byte in the network packets.

subsequent character is input to the machine to compute the next state transition. This process continues until all the payload bytes have been compared against the automaton. At any point of time in this process, if the computed next state belongs to the accepting states, then a signature is set to be detected by the automaton.

Since the 8-bit extended American Standard Code for Information Interchange (ASCII)[3] character set is used for the internet communication, the signatures for DPI applications are constructed using the ASCII character set. A signature set can either be represented through the the Non-deterministic Finite Automaton (NFA) or the Deterministic Finite Automaton (DFA). The primary difference between the representations is the output of the state transition function, which also affects the total number of states in the machine[4]. In the case of the NFA, the number of next states generated by the state transition function is non-deterministic, i.e, the state transition function can either generate a single next state, multiple next states or even no next state. On the other hand, in the case of the DFA, the state transition function always generates a single next state resulting in a deterministic output. It should be noted that both the machines (NFA & DFA) are equivalent representations of the language represented by a signature set.

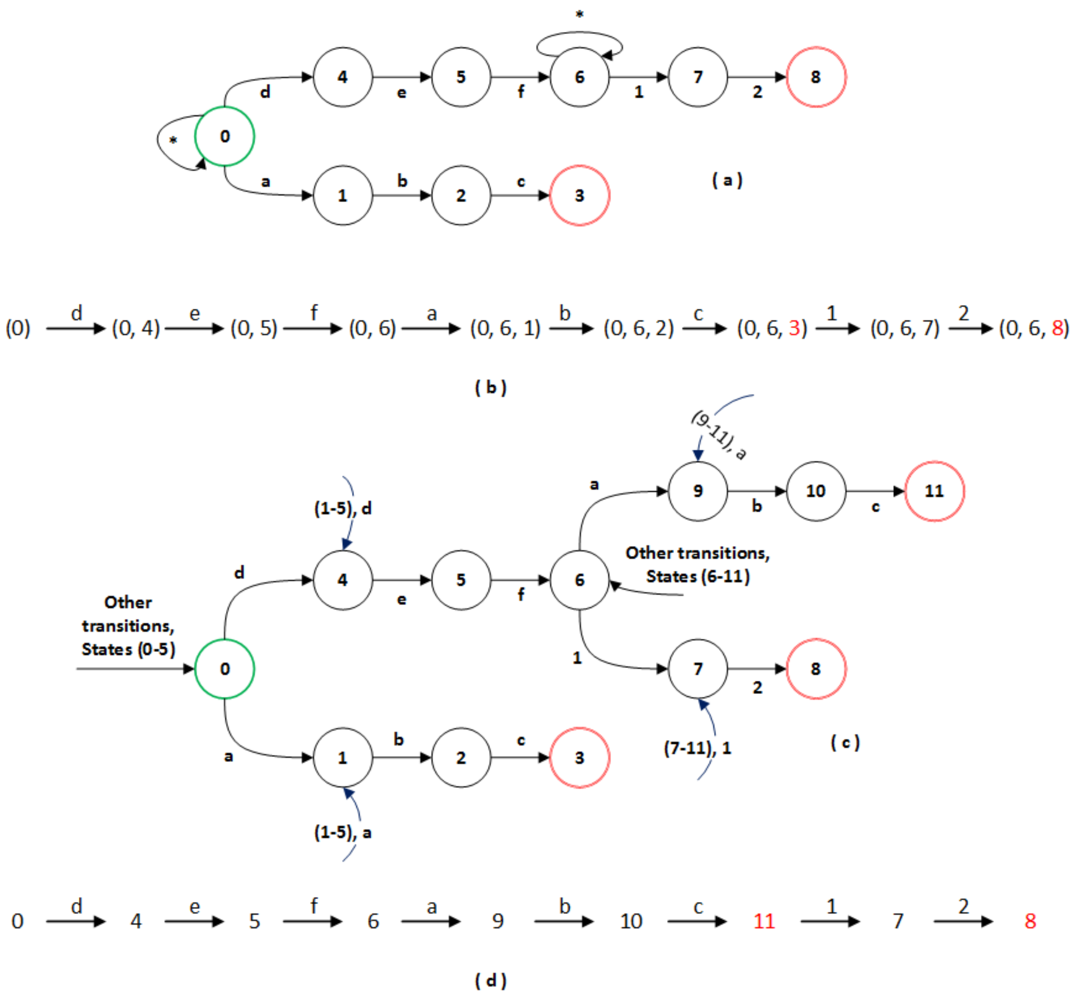
A signature set which consists of two different signatures, *abc* and *def.*12* is further used to explain the concepts behind the NFA & DFA representations. The former is a string signature and is only matched if there is a sequence of an 'a', followed by a 'b' and a 'c' in the payload. On the other hand, the latter is a regular expression which is only matched if there is a sequence of a 'def', followed by a '12' with zero or more characters in between 'def' and '12'. Figure 2.2(a) and (c), respectively show the NFA and the DFA representations corresponding to the signature set. The initial state in both the automaton representations have been shown using a green circle, while the red circles represent the accepting states.

Figure 2.2(b) and (d), respectively show the sequence of states traversed in the NFA and the DFA when the byte sequence to be inspected is composed of 'defabc12'. It can be seen that the byte sequence being inspected consists of both the signatures which are part of the signature definitions. In the case of the NFA, due to the non-determinism associated with the state transitions, multiple states are active at the same time during the signature matching as seen in Figure 2.2(b). The states which are active during the signature matching operation are called the 'active set' of states. So, the state transition function corresponding to a character has to be executed for all of the states in the active set. On the other hand, it can be seen from Figure 2.2(d) that only a single state is active during DFA based signature matching. Irrespective of the automaton representations, it can be seen from Figure 2.2 that the signatures matched by the sequence of bytes is identical in both the NFA and the DFA, which further prove their functional equivalence.

Converting the signatures into an automaton consists of various steps. A variety of algorithms [52, 53, 54] have been proposed in the literature to convert a signature into

[3]With respect to DPI applications, the extended ASCII character set is generally referred to as the ASCII character set. So as part of the dissertation, the usage of ASCII character set refers to the 8 bit extended ASCII character set.

[4]In the rest of the dissertation, the term *state-character combination* will be used to refer the inputs to the state transition function.



*
4
5
6
7
8
d
e
f
1
2
*
0
a
b
c
1
2
3
(a)
(0) d (0, 4) e (0, 5) f (0, 6) a (0, 6, 1) b (0, 6, 2) c (0, 6, 3) 1 (0, 6, 7) 2 (0, 6, 8)
(b)
(9-11), a
a
9
10
11
b
c
(1-5), d
Other transitions, States (6-11)
d
4
5
6
e
f
Other transitions, States (0-5)
0
1
7
8
2
a
1
b
2
c
3
(7-11), 1
(c)
(1-5), a
0 d 4 e 5 f 6 a 9 b 10 c 11 1 7 2 8
(d)

Figure 2.2: A signature set with 2 signatures abc, def.*12 represented as (a) NFA and (c) DFA. The states traversed during (b) NFA and (d) DFA based signature matching.

the NFA, which is the first step in converting the signatures into an automaton. After converting the signatures into the NFA, the subset construction algorithm [52] is used to convert the NFA into the DFA. As part of the subset construction algorithm, all the unique active set combinations seen in the NFA are converted to unique DFA states. In theoretical worst-case scenarios, if there are ‘N’ states in the NFA, the equivalent DFA could consist of a maximum of 2^N states. However, this exponential state blow-up called the *state explosion*, only results when the regular expressions in the signatures consist of constrained and unconstrained repetitions of wildcards or large character ranges [33, 50, 44]. Once the DFA is generated from the signature set, the state minimization algorithm [55] is used to generate the most compact DFA equivalent to the signature set.

2.2.2 Automata Based Signature Matching - Complexity Analysis

Table 2.1 shows a comparison of the processing complexity and the storage cost incurred to store the NFA and the DFA. Since, the state transition function in the DFA results in a single next state, the processing complexity associated in comparing an individual payload byte is always constant. On the other hand, since the state transition function has to be executed across multiple states in the active set, the worst case processing complexity in comparing a payload byte in the case of the NFA is exponentially related to the number of states[5]. However, with respect to automata storage, in worst case scenarios, the amount of memory required to store the DFA exponentially grows with respect to the number of states while it is linear in the case of the NFA. To summarize, the DFA is processing efficient and is storage inefficient while the NFA is processing inefficient and is storage efficient.

Table 2.1: Processing and storage complexity associated with NFA & DFA

	Processing complexity / payload byte	Storage cost
NFA	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
DFA	$\mathcal{O}(1)$	$\mathcal{O}(2^{\mathrm{N}})$

Due to the constant processing complexity associated to process a payload byte, the DFA is generally preferred for high speed signature matching applications [32]. Though, the constant processing complexity in the DFA enables to achieve signature matching at high rates, the storage problem has to be addressed for efficient signature matching implementations [1]. Addressing the storage problem in the DFA, the research community has primarily focused on compressing the DFA before it is stored in the memory [23, 34]. The next section provides a detailed overview of the various DFA compression algorithms discussed in the literature.

2.3 DFA Compression

Since the DFA is a deterministic state machine representation, it stores a single state transition for each of the characters in the alphabet, Σ for every state. Consequently, the total number of state transitions which are stored in the DFA, linearly depends on the total number of states generated as well as the total number of characters in the alphabet. Assuming that the DFA consists of 'S' states[6], the total number of state transitions (DFA_Trans) that have to be stored in the memory is represented by (2.1).

$$\mathrm{DFA\_Trans} = \mathrm{S} \times \Sigma \tag{2.1}$$

[5]The number of states here represents the total number of states generated in the NFA when a signature set is converted into the automata.

[6]Its a convention in the filed of automata theory to represent a DFA with 'S' states corresponding to an NFA with 'N' states.

The DFA compression methods proposed in the literature can be broadly classified into the *state compression* and the *transition compression* approaches [34]. The state compression algorithms primarily focus on reducing the number of states in the DFA, while the transition compression algorithms focus on compressing the redundant state transitions in the DFA. A detailed description of both the approaches are described below.

2.3.1 State Compression

2.3.1.1 State Explosion Problem

Under certain circumstances, the conversion of an NFA into the DFA can potentially result in an exponential explosion in the number of states created in the DFA. This is called the state explosion problem [50] and is typically addressed by the state compression algorithms. The exponential explosion of states primarily occurs when the regular expressions consist of constrained or unconstrained repetitions either over the wildcard character (.) or over character ranges. Various analyses [50, 33] have identified that the state explosion problem can be narrowed down to the following 3 scenarios[7] as described below:

- **Scenario 1: Length Restriction on an Anchored Regular Expression** State explosion occurs when the signature contains the regular expression of the following format: ˆA+[A-Z]{k}B. This specific regular expression pattern can be broken down into 3 parts. The first part is the prefix (A+), the middle part is the portion with the length restriction on the character range or the wildcard ([A-Z]k) and the final part is the suffix (B). The state explosion in this case occurs due to the overlap in the character(s) seen in the prefix portion with the character range seen in the middle portion. In this scenario, the generated DFA consists of states which record all possible combinations in the overlap occurrence in order to maintain functional correctness to the original signature representation. In addition to the states generated as part of the prefix and the suffix, $\mathcal{O}(k^2)$ states are further generated to track the overlap occurrence. For example, Figure 2.3(a) shows this scenario where the DFA corresponding to the signature ˆA +[ˆ\n]{3}B consists of 9 states (states 1-10) to keep track of all possible combinations which potentially lead to a signature match. It should be noted that an anchored regular expression (expressions which have ˆ as the first symbol) is matched in the payload bytes, if and only if the pattern exactly matches from the first byte within the payload byte sequence.

- **Scenario 2: Length Restriction on an Un-anchored Regular Expression** The second scenario is very similar to the first, but results in a case where the regular expression does not have the anchor symbol, i.e., the regular expression is of the format, .*A[A-Z]{k}B. In this scenario, the state explosion also occurs because of the overlap between the characters in the prefix and the ones in the character

[7]The examples associated to the 3 scenarios have been used from [34]

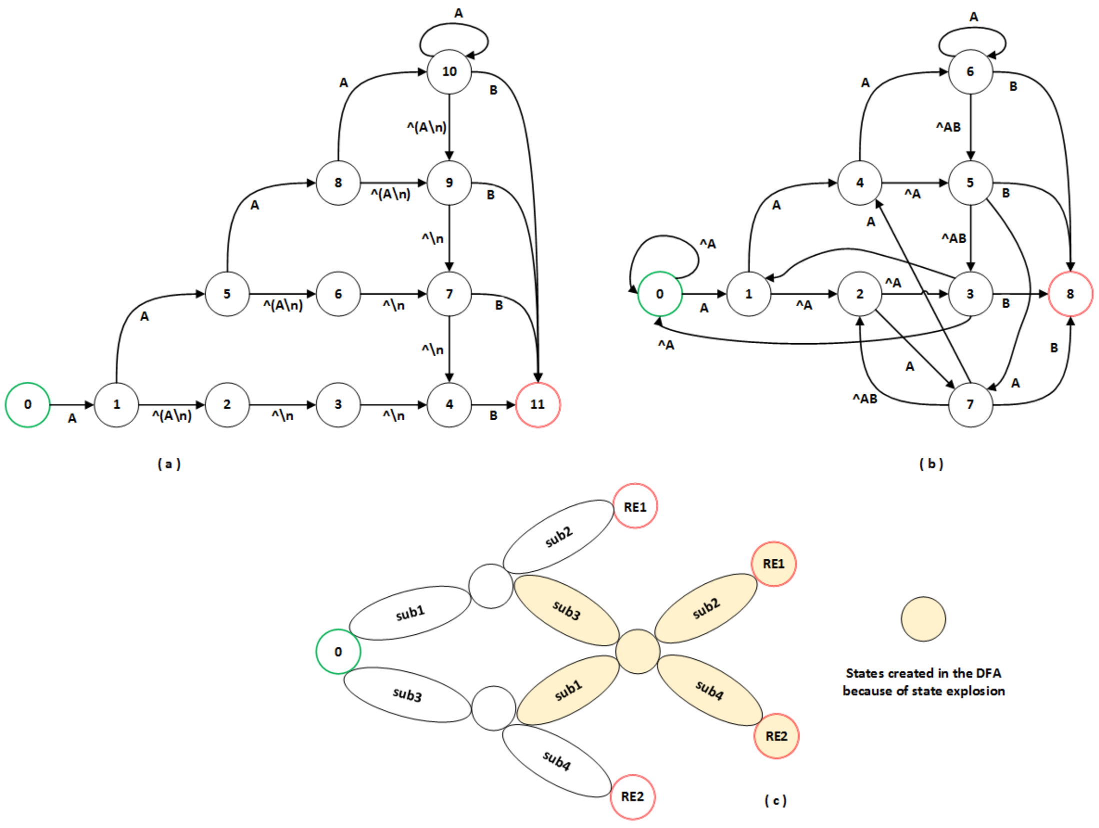


Figure 2.3: (a) state explosion scenario 1 (b) state explosion scenario 2 (c) state explosion scenario 3

class in the middle portion. Since, this is not an anchored regular expression, there is no restriction with respect to the occurrence of the signature within the sequence of payload bytes. For example, an 'A' seen anywhere in the sequence of bytes would potentially lead the DFA to search for the signature, irrespective of whether the previous few bytes resulted in a partial signature match. In this scenario, $\mathcal{O}(2^{\mathrm{k}})$ additional states are required to keep track of all possible combinations leading towards a potential signature match. Figure 2.3(b) shows an example of this scenario, where a DFA is shown corresponding to the regular expression .*A.{2}B.

- **Scenario 3: Regular Expression Combinations** Unlike the previous scenarios where the state explosion resulted because of the specific characteristics of the individual signature, this scenario occurs due to the interaction of multiple regular expressions with wildcard terms in it. Two regular expressions of the format RE1=sub1.*sub2 and RE2=sub3.*sub4 are used to show the state explosion in this scenario, where each sub$i$ could refer to a sequence of characters. The states corresponding to the sub-expressions are duplicated as shown in Figure 2.3(c) resulting in state explosion. This scenario occurs as the unrestricted wildcard character in a signature could potentially match the sub expressions of other signatures. So, the resulting DFA should be able to keep track of the following scenarios:

1. After matching a sub-expression within a regular expression (e.g. sub1 in RE1 / sub3 in RE2), the second sub-expression should be tracked for a possible signature match (e.g. sub2 in RE1/sub4 in RE2).
2. After matching the first sub-expression in a certain regular expression (e.g. sub1 in RE1 / sub3 in RE2), the sub-expressions which lead to a different signature should be tracked simultaneously (e.g. sub3 in RE2 / sub1 in RE1).

Thus when multiple regular expressions with unrestricted wildcard character repetitions are converted into a single DFA, the number of states in the resulting DFA grows rapidly.

2.3.1.2 State Compression Solutions

Various solutions have been proposed in the literature targeting the state explosion problem. Based on the algorithmic approach used to solve the state explosion problem, the solutions can be classified into *rule grouping*, *semi-deterministic FAs* and the *decomposed FAs* [34]. A brief overview of each of the approaches is described below:

- **Rule Grouping:** Since the state explosion primarily happens when multiple signatures are combined together to create a single DFA, one of the ways to avoid this problem is to create multiple DFAs corresponding to a signature set. Solutions such as [33, 56] have proposed rule grouping algorithms to split the signatures into *'m'* subsets, so that a single DFA is created for each of the subsets. Since the payload bytes corresponding to the network streams have to be compared against the original signature set, they have to be matched against each of the 'm' individual DFAs. So, the cost paid for the rule grouping is the increase in the processing complexity, i.e., $\mathcal{O}(m)$, to process each of the individual payload bytes. For example, the solution proposed in [56], which is the state-of-the-art rule grouping algorithm generates 7 DFAs corresponding to a signature set consisting of only 107 regular expressions. When the signature set is directly converted into a single DFA, it consists of more than a million states. After rule grouping, the 7 DFAs only consist of a total of 15k states. However, with an increase in the number of signatures to be supported, the total number of groups to which the signatures have to be split also increases, which linearly reduces the achievable signature matching throughput. One of the methods which has been discussed in the literature is the usage of multicore processors to perform the signature matching function against multiple DFAs in parallel to hide the linear reduction in the signature matching throughput. However, the number of DFAs to which the signatures are split completely depends on the characteristics of the signature set. So, the scalability of this approach to support increasing number of signatures is a major concern with this approach [34].

- **Semi-deterministic FA:** Solutions such as the Hybrid Finite Automata (HFA) [50], the Tunable Finite Automata (TFA) [57], the DFA with extended character set (DFA-eC) [58], etc., fall into this category and are semi-deterministic automata representations. In the case of the HFA, a certain portion of the automaton is deterministic while the signature structures which lead to state explosion are made non-deterministic. So, the number of states in the active set during signature matching varies depending on whether the deterministic or the non-deterministic portion of the HFA is traversed. On the other hand, solutions such as [57, 58] propose to have a fixed number states which are always active during signature matching. In the case of the TFA, the number of states which are active is greater than '1' and is predefined to a fixed value which is configurable (i.e., 2/3/4), while in the case of DFA-eC, it is always set to 2. With respect to state compression, more than 90% of the states on average are compressed by all of these methods while the individual number of states compressed vary depending on the characteristics of the specific signature sets. Since there is no standard signature set which is used for evaluation purposes, evaluating the state compression results across the techniques discussed above is not straight foward. In general, the cost paid for the state compression in these solutions is the increase in the memory bandwidth due to the semi-deterministic nature of the generated automata. In the case of the HFA, the memory bandwidth varies dynamically depending on the specific portion of the automata that is being traversed. In worst case scenarios, the memory bandwidth required in the case of the HFA is equivalent to that of the NFA. In the case of the DFA-eC, the memory bandwidth doubles which also reflects through the 50% reduction in signature matching performance in comparison to that of the DFA.

- **Decomposed FA:** Decomposed FAs are a collection of solutions in which certain additional control information is augmented to the state transitions to combat the state explosion. Solutions such as the eXtended Finite Automata (XFA)[44], the Jump Finite Automata (JFA)[35], etc., belong to this category. For example, if the regular expression is of the format ab.*cd, a set bit is used to indicate that the sub expression *ab* has been matched in the state transition. The state transition leading to the state after matching 'c' is only transitioned to, after checking the status of the set bit. In this way, the state explosion resulting from the combination of regular expressions with unrestricted wildcard characters is easily combated. The additional information which is augmented into the state transitions varies depending on the characteristics of the signature set. On the other hand, Bechhi et al. [59] proposed the usage of multiple counters to keep track of the characters seen in the length restricted regular expressions[8], to evade the state explosion in the DFA. So, the decomposed FAs introduce additional control information along with the state transitions, to solve the state explosion problem. Unlike the semi-deterministic FAs, the memory bandwidth associated with the state transition fetch in the case of the decomposed FAs is the same as that of the DFA. However, the additional processing which is introduced as part of the execution of the state

[8]Explained as part of scenario 1 and scenario 2 in the previous section.

transition function depends on the information augmented in the state transition [34].

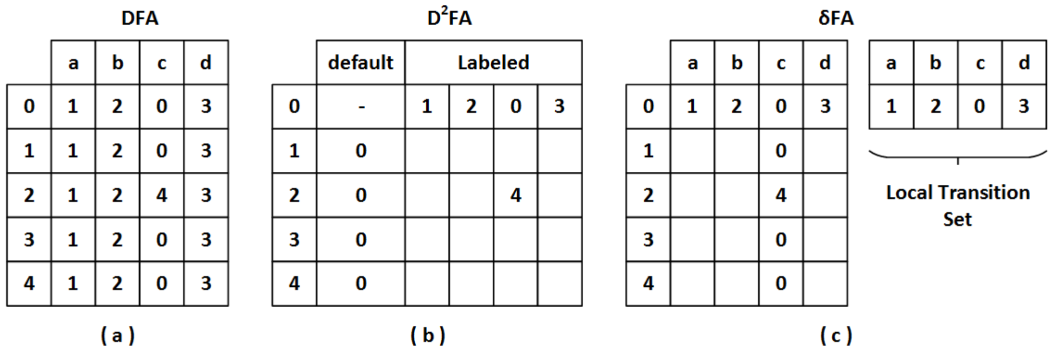
After performing the state compression, the total number of states in the DFA reduces from S to S′ (S′ ≪ S). However, each of the states in the state compressed DFA stores the state transitions for all the characters in the alphabet Σ. So, even after performing the state compression, a major portion of the state transitions are redundant and have to be compressed through the transition compression algorithms. So, performing the transition compression is mandatory in the case of the DFA irrespective of whether the state compression is performed or not.

2.3.2 Transition Compression

Though there are 256 state transitions for each of the states in the DFA, a majority of these state transitions are redundant and leads the machine towards the root state or one of states closer to the root state [1]. The process of compressing these redundant state transitions in the DFA is called as transition compression. As discussed previously, transition compression can either be performed on the DFA or a state compressed DFA. Various transition compression algorithms have been proposed in the literature and they can be broadly classified into the hardware oriented and software oriented algorithms [39]. This section also discusses the alphabet compression mechanism which compresses the redundant transitions by compressing the indistinguishable characters in the alphabet. A short overview of all these algorithms is described further.

2.3.2.1 Software Oriented Algorithms

The concept of transition equivalence between multiple states was first used by Kumar et al. [1] to compress the redundant state transitions in the DFA. The compressed DFA representation was called the Delayed-input DFA (D^2FA). Figure 2.4(a) shows an uncompressed DFA with 5 states and the state transitions corresponding to 4 characters ($\Sigma = \{a, b, c, d\}$), while Figure 2.4(b) shows the compressed D^2FA corresponding to the DFA. As part of the transition compression, the state transitions between the states are compared and compressed if they are identical. After compression, the compressed state transitions in the states are classified into the default and the labeled state transitions. For example, with respect to states 0 and 2 in Figure 2.4(a), the state transitions corresponding to characters a, b and d in state 2 are compressed as they are identical to those in state 0. On the other hand, the state transition corresponding to character ‘c’ in state 2 is not compressed, since it is different from the state transition corresponding to character ‘c’ in state 0. So, the state transition corresponding to character ‘c’ in state 2 is added to the labeled transitions. Moreover, a new transition called a default transition is introduced in state ‘2’ which is directed towards state ‘0’. This implies that the state transitions corresponding to the characters whose transitions were compressed, i.e., the characters a, b and d should be fetched from state 0. As part of signature matching, in order to fetch a state transition corresponding to a state-character combination, the labeled transitions are searched first. If the state transition is not found among the labeled



DFA

	a	b	c	d
0	1	2	0	3
1	1	2	0	3
2	1	2	4	3
3	1	2	0	3
4	1	2	0	3

(a)

D^2FA

	default	Labeled			
0	-	1	2	0	3
1	0				
2	0			4	
3	0				
4	0				

(b)

δFA

	a	b	c	d
0	1	2	0	3
1			0	
2			4	
3			0	
4			0	

a	b	c	d
1	2	0	3

Local Transition Set

(c)

Figure 2.4: (a) A DFA with state transitions for 5 states and 4 characters (b) The compressed D^2FA representation of the DFA (c) δFA representation of the DFA

transitions, the search is further directed to the state pointed by the default transition, which introduces an additional table lookup. However, it should be noted that the default transition doesn't consume an additional character, but introduces additional memory fetches as part of signature matching.

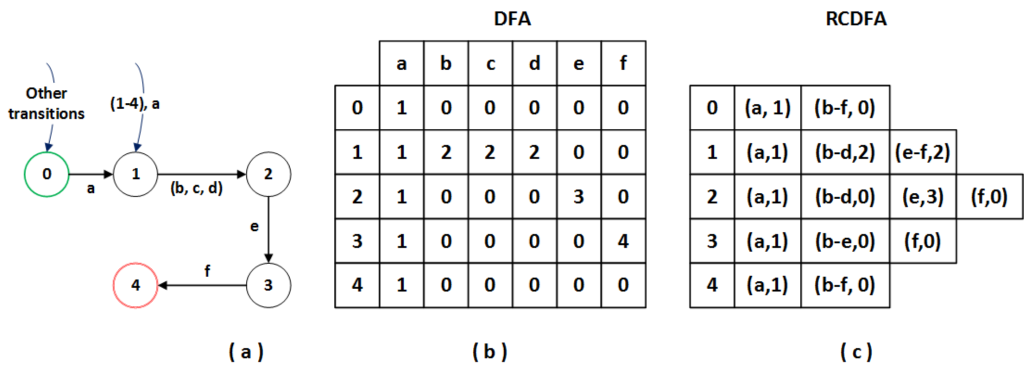
The D^2FA employs the Kruskal's algorithm [60] to identify the states which have the maximum number of identical transitions between the states. Though the D^2FA representation can achieve transition compression rates of the order of 95%, the associated default transition paths can be long. In turn, the signature matching throughput achieved through D^2FA completely depends on the traffic characteristics which determines the number default and labeled transitions that are traversed.

Addressing the limitation associated with the default transition chains, Becchi et. al. [32, 2] introduced an improved version of the D^2FA in which the default transitions were directed to states which were of lower depth in comparison to the state of interest. The depth of the state was defined as the number of hops taken from the initial state to the specific state of interest. The choice of assigning the default transition in this way, arises from the observation that most of the state transitions in a state either lead to the initial state (root state) or one of its neighboring states, which reduced the length of the default transition paths. However, the reduction in the default path also resulted in a reduced transition compression in certain cases. On average, transition compression rate of about 90% was achieved by the improved D^2FA proposal.

Ficara et al. proposed the δFA [3], which is also a compressed representation of the DFA. In δFA, if a state 's' has a state transition directed towards another state 't', then 's' was defined as the parent state of 't'. In this way, the identical state transitions between a state and its parent state were compressed. Additionally, a local transition set is maintained in the cache memories from which the compressed state transitions are fetched during signature matching to minimize the memory bandwidth. The local transition set is first prefilled with the state transitions from the initial state. As the payload bytes are inspected, the local transition set is periodically updated from the state transitions in the δFA representation, which is stored in the main memory. Figure 2.4(c) shows the δFA corresponding to the DFA shown in Figure 2.4(a). The authors in [3] argue that the memory bandwidth required to fetch the state transition is constant in the case of the δFA, due to the usage of the local transition set. However, the additional memory bandwidth which is required to periodically update the local transition set is not considered by the authors which varies depending on the sequence of the payload bytes being inspected. On average, the δFA method only achieves transition compression rates of the order of 90%.

Finally, Antonello et al. proposed the Ranged Compressed DFA (RCDFA) [36] in which the state transitions are represented corresponding to the character ranges. Figure 2.5(a) shows the DFA corresponding to the signature a[b-d]ef, while Figure 2.5(b) and (c), respectively show the state transition table and the RCDFA representation corresponding to the DFA. The RCDFA leverages the fact that a majority of the state transitions in a state are always directed to the same next state. This property is exploited to store the state transitions for a group of characters, instead of storing the state transitions for individual characters as in the case of the DFA. In this way, a majority of the state transitions are compressed. Experimental evaluation of the RCDFA resulted in transition compression rates of the order of 97%. As in the case of other compression algorithms [1, 32], the state transition corresponding to a state character combination has to be sequentially searched in each of the states in the RCDFA. So, even in this method, the memory bandwidth required to search the state transitions in a sequential fashion is the cost paid for transition compression.

The signature matching throughput that is achieved by the methods described above was compared by Antonello et al. in [36]. The signature matching engines were implemented as software programs and the evaluation was performed on an Intel core i7-2600 [61] running at 3.4 GHz with 8GB of memory. The signature matching throughput achieved when the payload is compared against the uncompressed DFA representation is used as the benchmark for comparison purposes. The average signature matching throughput achieved by the DFA (uncompressed) based signature matching engine was ~400 Mbps. On the other hand, the average signature matching throughput achieved through the D^{2}FA and the RCDFA implementations were of the order of 10 Mbps and



DFA

	a	b	c	d	e	f
0	1	0	0	0	0	0
1	1	2	2	2	0	0
2	1	0	0	0	3	0
3	1	0	0	0	0	4
4	1	0	0	0	0	0

RCDFA

0	(a, 1)	(b-f, 0)		
1	(a,1)	(b-d,2)	(e-f,2)	
2	(a,1)	(b-d,0)	(e,3)	(f,0)
3	(a,1)	(b-e,0)	(f,0)	
4	(a,1)	(b-f, 0)		

Figure 2.5: A signature a[b-d]ef represented as a DFA and through RCDFA

200 Mbps, respectively. It was pointed out in the evaluation that the reduction in the signature matching throughput is due to the fact that multiple additional state transitions have to be sequentially fetched from the memory in order to identify the compressed state transition corresponding to a state character combination. The δFA based signature matching engine was not considered for the evaluation in [36], as a previous study [62] had identified that the δFA based signature matching engine resulted in 100 times lower throughput in comparison to the DFA.

To summarize, the various approaches [1, 32, 3, 36] described in this section are capable of achieving transition compression rates of the order of 90$\sim$95%. However, the cost paid for the transition compression is the additional memory bandwidth to fetch the compressed state transition from the memory. Moreover, a study in [4] identified that the algorithms such as [1, 32] cannot be directly used in hardware implementations, since it would require multiple Tbps of memory bandwidth to perform signature matching even at 10 Gbps, when the transitions are compressed using the [1, 32]. They also mention that it would not be feasible to design systems supporting such high memory bandwidth. Alternately, the next section describes the bitmap based transition compression algorithms which can be used for line rate signature matching implementations.

2.3.2.2 Hardware Oriented Algorithms

In comparison to the implementations discussed in the previous sections, the algorithms discussed in this section primarily allow the signature matching to be performed in a dedicated hardware accelerator. The algorithms which are discussed in this section use the *bitmap* to compress the redundant state transitions in the DFA. A short introduction to the bitmap is provided first, followed by a detailed description of the various bitmap based transition compression algorithms.

A Bitmap is a simple, but an efficient method to compress the redundant state transitions in the DFA. The bitmap primarily compresses those identical state transitions which are adjacent to each other within a state. If the state transition corresponding to a character is compressed, a '0' is stored corresponding to the character position in the bitmap, while a '1' is stored if not.

Figure 2.6 shows an example of the bitmap based transition compression. Figure 2.6(a) shows a sequence of 8 transitions in a state corresponding to the alphabet, Σ=a, b, c, d, e, f, g & h. When the bitmap is used to compress the transition sequence, the transitions corresponding to the characters b, e, f and h (represented in purple blocks) are compressed, while the others (represented in green blocks) are not compressed. Figure 2.6(b) shows the bitmap corresponding to the transition sequence shown in Figure 2.6(a). The bits corresponding to the character positions b, e, f and h in the bitmap are set to '0', since they are compressed while the bits corresponding to other character positions are set to '1' in the bitmap. The compressed transitions are stored in the unique transition list as shown in Figure 2.6(c), in which each of the indices is called the unique transition index. If there are 'K' characters in an alphabet, a K-bit bitmap is generated for each state in the DFA, while the number of entries in the unique transition list varies depending on the number of transitions compressed through the bitmap.

As part of the decompression, the bitmap corresponding to a state is used to identify the compressed transition from the unique transition list. The example shown in Figure 2.6(d) describes the compressed state transition fetch corresponding to character ‘d’ from the unique transition list. The unique transition index corresponding to the character is calculated by performing the *population count* operation on the bitmap, which identifies the total number of occurrences of ‘1’ in a bit vector. As shown in Figure 2.6(d), the population count operation is only performed over a subset of bits in the overall bitmap, i.e., from the first character (‘a’) until the character of interest (‘d’). With respect to the example shown above, there are three 1’s in the bitmap which when added results in the unique transition index of 3. The state transition which is fetched from the index ‘3’ in the unique transition list is identical to that of the transition in the uncompressed representation seen in Figure 2.6(a). The location of the state transition in the compressed state table can be directly computed from the bitmap. So, the memory bandwidth that is required to fetch the state transition from the compressed state table is always constant. Due to this advantage, the decompression can be performed in a hardware accelerator to achieve line rate signature matching. However, the cost paid for the compression is the additional memory that is required to store the control information such as the bitmaps along with the compressed state transitions.

The usage of the bitmap to compress the redundant transitions in the DFA was first proposed by Tuck et al. in [63]. In this proposal, the authors compress the redundant state transitions within each of the states along the character axis using the bitmap. So, a 256-bit[9] bitmap is required for each of the states to compress the redundant state transitions. However, the downside of this approach is the linear increase in the memory

[9]There are a total of 256 state transitions in a state corresponding to the 256 ASCII characters in the alphabet. So, a 256-bit bitmap is required to identify if each of the individual state transitions is compressed or not.

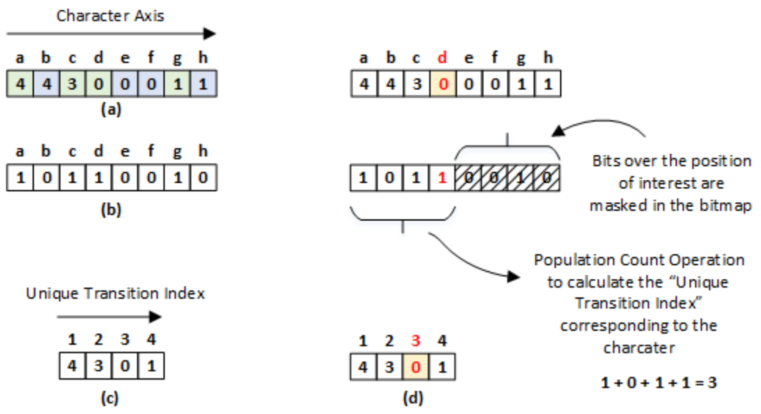


Figure 2.6: (a) A sequence of 8 Transitions within a state (b) The bitmap corresponding to the state transitions to identify if a state transition is compressed (c) Compressed state transition representation after bitmap based compression stored in a unique transition list (d) Transition decompression - Example

requirement to store the bitmaps corresponding to each of the states in the compressed DFA. Addressing this issue, Qi et al. [4] proposed a two-dimensional bitmap based transition compression algorithm called FEACAN[10]. Since the authors in [4] observed identical bitmaps in the DFA after bitmap based compression in [63], they proposed to only store those unique bitmaps in the DFA in the algorithm which they proposed. FEACAN employs a 3-step sequence to compress the redundant state transitions in the DFA. The first is the intra-state transition compression, followed by the second state grouping step and the last and final inter-state compression step. The intra-state compression step focuses on compressing the redundant transitions within each state using the bitmaps. The next is the state grouping step in which the states are grouped into subsets of states, based on the following clustering constraints as described below:

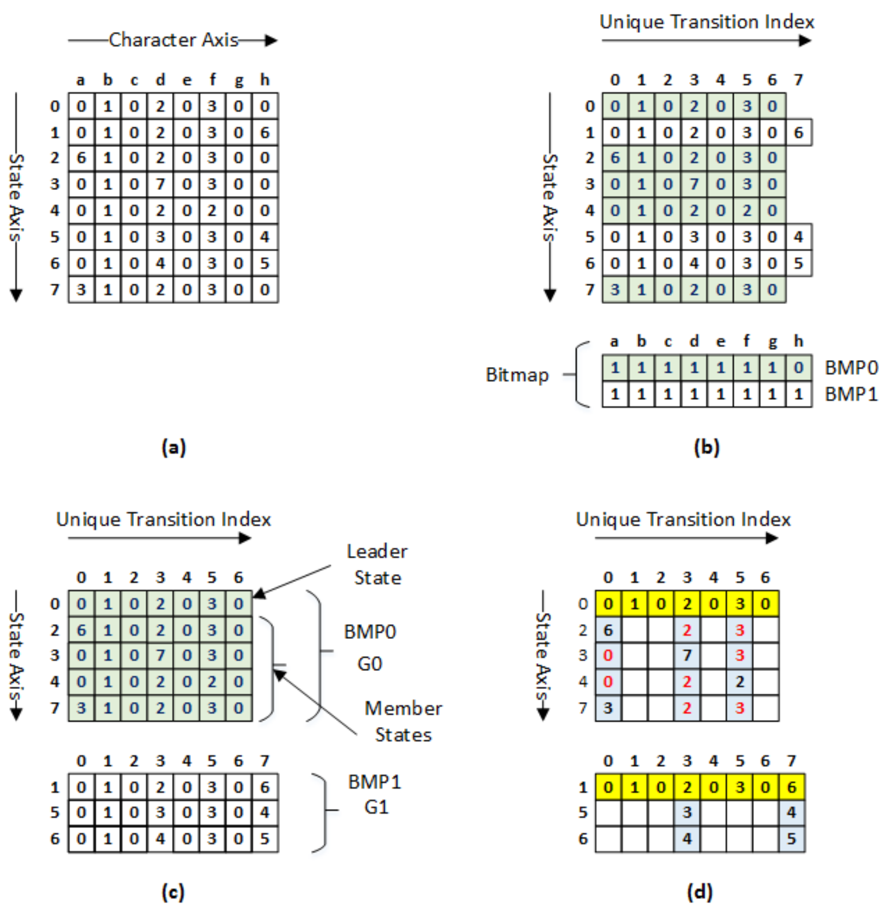
- The bitmap of the states which are clustered into a subset should be identical. This allowed to only store those non-distinguishable bitmaps in the compressed DFA.
- The states which are clustered into a subset of states, should have a certain minimum number of identical state transitions between them. The minimum number of identical transitions is defined as the transition threshold 'T'. This allowed to group the states which share a certain number of identical transitions which were compressed in the inter-state compression step.

After the state grouping step, one of the states in each group is assigned as the leader state while the other states are called as the member states. Finally, as part of the inter-state compression step, the state transitions corresponding to the member states at each of the index are compressed, if and only if they are identical to the state transition in the leader state at the same index. Even if one of the state transitions among the member states is different from that of the leader state at the unique transition index, none of the member state transitions corresponding to the index are compressed which results in inefficient transition compression.

Figure 2.7 describes the FEACAN transition compression algorithm using an example. Figure 2.7(a) shows a DFA with 8 states and state transitions corresponding to the character set Σ=a,b,c,d,e,f,g,h. Figure 2.7(b) shows the DFA after the intra-state compression step in FEACAN. As observed in Figure 2.7(b), the bitmap for each of the state either belongs to BMP0 or BMP1, where BMP represents a unique bitmap pattern. After the intra-state compression step, the states are clustered into two different groups (G0 & G1) as part of the state grouping step as shown in Figure 2.7(c). Figure 2.7(d) shows the compressed transitions after the inter-state compression step. It can be seen from Figure 2.7(d) that not all redundant transitions are efficiently compressed as part of the inter-state compression step through FEACAN. The state transitions in the member states in G0, corresponding to indices 1, 2, 4 and 6 are compressed as all the transitions in the aforementioned indices are identical to the state transition in the leader state.

[10]A signature matching engine was proposed by Qi et. al., The system was called as the Front End Acceleration for Content Aware Network processing (FEACAN), which resulted in the algorithm also to be called as FEACAN.

On the other hand, the state transitions in the member states corresponding to indices 0, 2 and 5 are not compressed at all, as some state transitions corresponding to the member states are not identical to that of the leader state. In this way, although certain state transitions are redundant, yet they are not compressed because of the inter-state compression policy used in FEACAN which is a limitation of the compression algorithm. On average, the transition compression rates achieved by FEACAN is of the order of 90%.



(a)

	a	b	c	d	e	f	g	h
0	0	1	0	2	0	3	0	0
1	0	1	0	2	0	3	0	6
2	6	1	0	2	0	3	0	0
3	0	1	0	7	0	3	0	0
4	0	1	0	2	0	2	0	0
5	0	1	0	3	0	3	0	4
6	0	1	0	4	0	3	0	5
7	3	1	0	2	0	3	0	0

(b)

	0	1	2	3	4	5	6	7
0	0	1	0	2	0	3	0	
1	0	1	0	2	0	3	0	6
2	6	1	0	2	0	3	0	
3	0	1	0	7	0	3	0	
4	0	1	0	2	0	2	0	
5	0	1	0	3	0	3	0	4
6	0	1	0	4	0	3	0	5
7	3	1	0	2	0	3	0	

	a	b	c	d	e	f	g	h
BMP0	1	1	1	1	1	1	1	0
BMP1	1	1	1	1	1	1	1	1

(c)

	0	1	2	3	4	5	6
0	0	1	0	2	0	3	0
2	6	1	0	2	0	3	0
3	0	1	0	7	0	3	0
4	0	1	0	2	0	2	0
7	3	1	0	2	0	3	0

	0	1	2	3	4	5	6	7
1	0	1	0	2	0	3	0	6
5	0	1	0	3	0	3	0	4
6	0	1	0	4	0	3	0	5

(d)

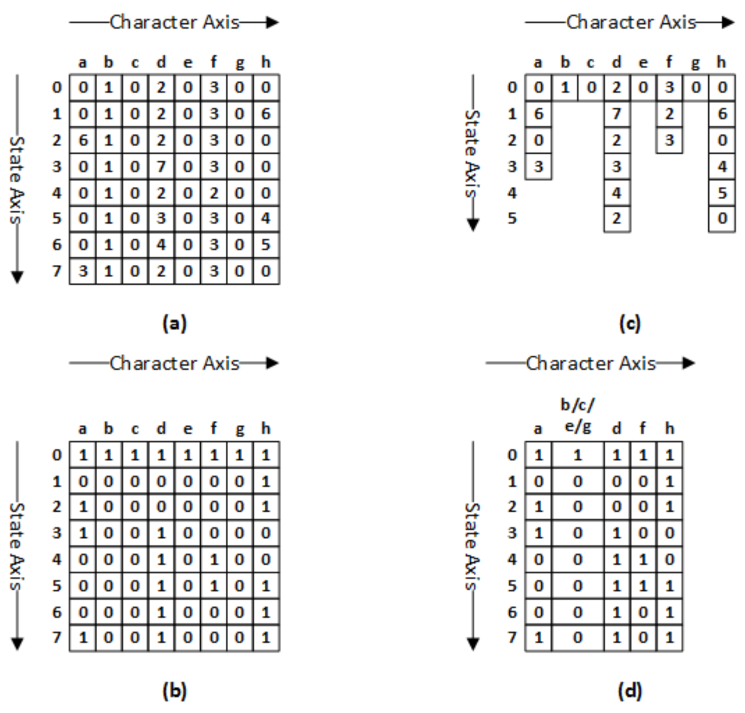
	0	1	2	3	4	5	6
0	0	1	0	2	0	3	0
2	6			2		3	
3	0			7		3	
4	0			2		2	
7	3			2		3	

	0	1	2	3	4	5	6	7
1	0	1	0	2	0	3	0	6
5				3				4
6				4				5

Figure 2.7: Example to explain the FEACAN transition compression (a) An uncompressed DFA with 8 states and state transitions for 8 characters for each state (b) The compressed DFA after bitmap based intra-state transition compression (c) The DFA states grouped into subsets of states after intra-state compression (d) The compressed DFA after inter-state transition compression

Reorganized and Compact DFA (RCDFA) [5] is another transition compression algorithm which uses the bitmaps to compress the redundant transitions in the DFA. Unlike FEACAN which performs bitmap based transition compression along the character axis, the RCDFA performs transition compression along the state axis using the bitmaps. The RCDFA primarily proposes a state grouping algorithm in which the DFA states are reorganized in such a way that the states with identical state transitions are organized next to each other. In this way, the states are reorganized before the transition compression is performed.

Figure 2.8 describes the transition compression using the RCDFA, with the same example used to describe the FEACAN transition compression algorithm. Figure 2.8(b) shows the bitmap's along the state axis for the example used in Figure 2.8(a), while Figure 2.8(c) shows the compressed transitions after the RCDFA based transition compression. Since the RCDFA performs bitmap based compression along the state axis, the number of states in the DFA defines the width of the bitmap for each of the characters. To generalize, if there are 'S' states in the DFA, an S-bit bitmap is required for each and every character. However, since the number of states in the DFA varies depending on the characteristics of the signature set, it would be tricky to architect an efficient storage architecture for the bitmaps generated from the RCDFA. On the other hand, RCDFA also observed that the bitmaps corresponding to some of the characters are identical and need not be stored multiple times in the memory. This is shown in the Figure 2.8(d), where the bitmaps corresponding to characters b, c, e and g are combined into a single bitmap entry. Each bitmap corresponding to the characters can be identified uniquely by assigning a bitmap identification to each of the unique bitmaps.



(a)

	a	b	c	d	e	f	g	h
0	0	1	0	2	0	3	0	0
1	0	1	0	2	0	3	0	6
2	6	1	0	2	0	3	0	0
3	0	1	0	7	0	3	0	0
4	0	1	0	2	0	2	0	0
5	0	1	0	3	0	3	0	4
6	0	1	0	4	0	3	0	5
7	3	1	0	2	0	3	0	0

(b)

	a	b	c	d	e	f	g	h
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	1
2	1	0	0	0	0	0	0	1
3	1	0	0	1	0	0	0	0
4	0	0	0	1	0	1	0	0
5	0	0	0	1	0	1	0	1
6	0	0	0	1	0	0	0	1
7	1	0	0	1	0	0	0	1

(c)

	a	b	c	d	e	f	g	h
0	0	1	0	2	0	3	0	0
1	6			7		2		6
2	0			2		3		0
3	3			3				4
4				4				5
5				2				0

(d)

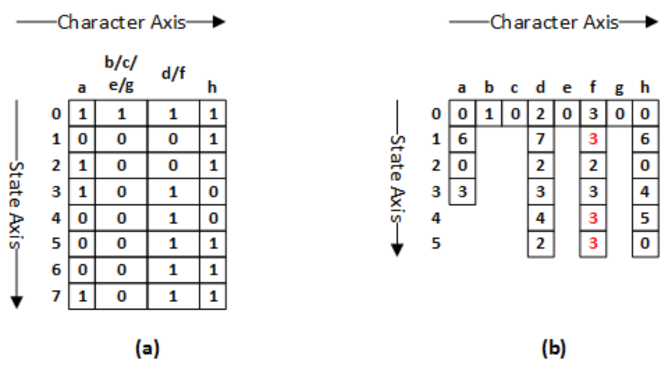
	a	b/c/ e/g	d	f	h
0	1	1	1	1	1
1	0	0	0	0	1
2	1	0	0	0	1
3	1	0	1	0	0
4	0	0	1	1	0
5	0	0	1	1	1
6	0	0	1	0	1
7	1	0	1	0	1

Figure 2.8: (a) An uncompressed DFA with 8 states and state transitions for 8 characters for each state (b) The compressed DFA after RCDFA (c) The bitmaps across the vertical state axis (d) Unique bitmap after being combined

Based on the experimental evaluation, the authors claim that the RCDFA can achieve transition compression rates of the order of 95%.

Unlike the proposal in [63], both FEACAN and RCDFA do not store all the bitmaps together with the compressed DFA. Especially in the case of the RCDFA, in worst case scenarios, the memory required to store the bitmaps can increase exponentially since the bitmap is created along the state axis. So, in order to reduce the storage requirements associated with the bitmap, both FEACAN and RCDFA propose a mechanism to combine the bitmaps. Two bitmaps are combined into a single bitmap by performing a *logical 'OR'* operation on the concerned bitmaps. After combining multiple bitmaps, it is enough if the combined bitmap alone is stored in the memory. However, the bitmap combination comes at the cost of more redundant transitions stored in the unique transition list corresponding to the combined bitmaps. An example of the bitmap combination in the case of the RCDFA is shown in Figure 2.9. The bitmaps corresponding to characters 'd' and 'f' are combined into a single bitmap which results in an additional three redundant transitions added to the unique transition list corresponding to character 'f'. The state transitions which are additionally stored due to the bitmap combination are marked in red in Figure 2.9(b).

All the algorithms which were discussed in this section [63, 4, 5] propose a hardware accelerator to perform the signature matching at line rates. The compressed DFA is typically stored in the on-chip memories, so that the hardware accelerator can fetch them at low latencies to perform the signature matching function. Unlike the software oriented algorithms, the bitmap based algorithms maintain the constant memory bandwidth associated with a compressed transition fetch corresponding to a payload byte, even after the transition compression. This property allows the decompression to be performed in a dedicated hardware accelerator. On the contrary, the cost paid by the bitmap based transition compression techniques is the additional memory required to store the bitmaps along with the compressed state transitions. The various methods discussed


Character Axis
State Axis
(a)
(b)

	a	b/c/e/g	d/f	h
0	1	1	1	1
1	0	0	0	1
2	1	0	0	1
3	1	0	1	0
4	0	0	1	0
5	0	0	1	1
6	0	0	1	1
7	1	0	1	1

	a	b	c	d	e	f	g	h
0	0	1	0	2	0	3	0	0
1	6			7		3		6
2	0			2		2		0
3	3			3		3		4
4				4		3		5
5				2		3		0

Figure 2.9: (a) Bitmaps corresponding to character 'd' and 'f' combined (b) Additional transitions stored corresponding to the unique transition list for character 'f'

in this section address the storage problem associated with the bitmaps by combining the bitmaps, which comes at the cost of reduced transition compression rates. This in turn results in inefficient transition compression results and inefficient on-chip memory usage as part of the compressed DFA storage. This is the major drawback of the bitmap based transition compression methods and is a major problem which is addressed in this dissertation.

2.3.2.3 Alphabet Compression

Alphabet Compression is the process of combining multiple indistinguishable characters in the character set Σ to generate an encoded character set. Through alphabet compression, the redundant state transitions corresponding to the indistinguishable characters are also compressed without compromising on the structural equivalence to the original signature set. The motivation for alphabet compression is further described through the scenarios as discussed below.

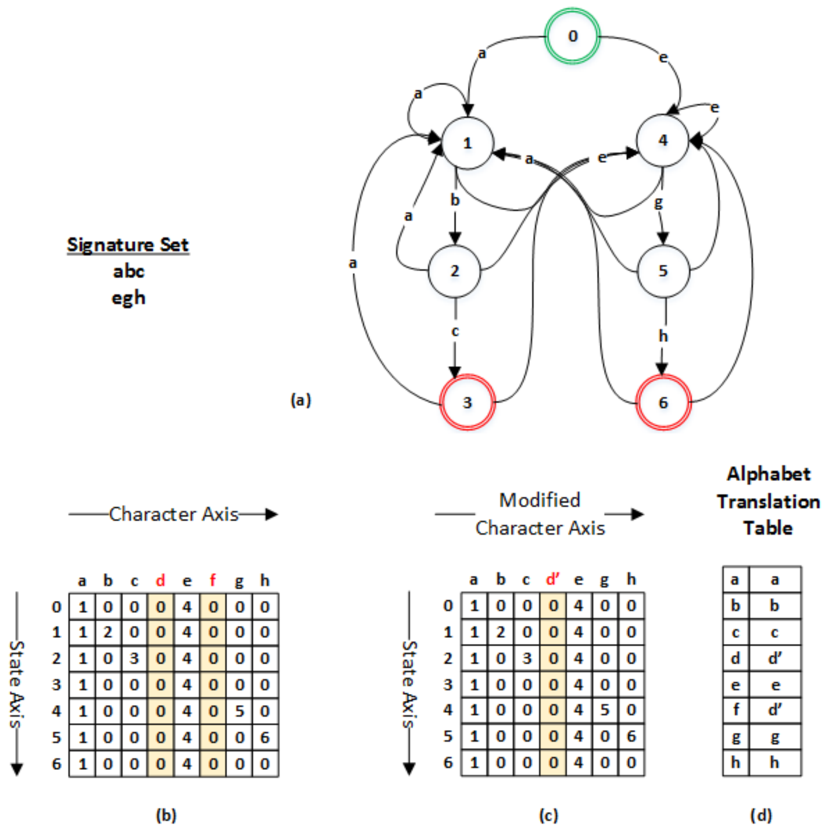
Scenario 1: Though there are 256 unique characters in the ASCII character set, not all the characters are always used to construct the signatures for DPI applications. The choice of characters generally vary depending on the individual signature sets. The state transitions corresponding to the characters which do not occur in the signature set typically lead to the same next state in the DFA making these characters indistinguishable.

Scenario 2: The expressiveness offered by regular expressions is one of the primary reasons to use them to define signatures for DPI applications. One of the features in regular expressions is the possibility to use character classes to define the signatures. For example, the signature *abc[d-h]* matches a sequence of characters a, b and c followed by any character between the characters d to h. In such a case, from a signature matching point of view, the characters from d until h are generally indistinguishable, unless and until there is some other signature within the signature set which uses a character between d and h[11]. Due to the common usage of character classes in real life signatures, there is a high chance that the state transitions associated with these characters are redundant as well [32].

The common occurrence of indistinguishable characters in the signature sets makes alphabet compression an essential part of transition compression. A character set with 8 characters, i.e., Σ={a, b, c, d, e, f, g, h} is further used to explain the concept of alphabet compression. Figure 2.10(a) shows a signature set, 'abc' and 'egh' and the DFA corresponding to the signature set. The resulting DFA corresponding to the signature set consists of 7 states in total, with state '0' being its root state, while states '3' and '6' are the accepting states corresponding to the signatures 'abc' and 'efg', respectively. Figure 2.10(b) shows the state table representation of the DFA shown in Figure 2.10(a).

As seen in Figure 2.10(b), the state transitions corresponding to the characters, 'd' and 'f' are identical across all the states in the DFA. Since the characters 'd' and 'f' are not part of the signatures, the state transitions corresponding to these 2 characters

[11]In this scenario, one of the states in the DFA will have a forward state transition corresponding to the character between the character class, which will be different from the state transition in the other states which are identical.



	a	b	c	d	e	f	g	h
0	1	0	0	0	4	0	0	0
1	1	2	0	0	4	0	0	0
2	1	0	3	0	4	0	0	0
3	1	0	0	0	4	0	0	0
4	1	0	0	0	4	0	5	0
5	1	0	0	0	4	0	0	6
6	1	0	0	0	4	0	0	0

	a	b	c	d’	e	g	h
0	1	0	0	0	4	0	0
1	1	2	0	0	4	0	0
2	1	0	3	0	4	0	0
3	1	0	0	0	4	0	0
4	1	0	0	0	4	5	0
5	1	0	0	0	4	0	6
6	1	0	0	0	4	0	0

a	a
b	b
c	c
d	d’
e	e
f	d’
g	g
h	h

Figure 2.10: Transition Compression through Alphabet Compression in a DFA

across all the states in the DFA lead to the root state. So, with respect to the DFA, the characters, ‘d’ and ‘f’ are completely indistinguishable. As part of the alphabet compression, the characters ‘d’ and ‘f’ are combined to create d', which is an encoded representation of both these characters. This modifies the original character set Σ to Σ'={a, b, c, d$'$, e, g and h} which consists of 7 characters in comparison to 8 characters in the original character set. Figure 2.10(c) represents the DFA after implementing the alphabet compression. It can be seen from the figure that the alphabet compression mechanism not only compresses the character set, but also the state transitions associated with the characters that are compressed. For example, the DFA corresponding to the original character set consists of 7$\times$8=56 transitions, while the DFA corresponding to the encoded character set only consists of 7$\times$7=49 transitions which is a direct result of the reduction in the character set. To formalize, two characters a, b $\subset \Sigma$ are compressed through alphabet compression, if δ(s,a) is identical to δ(s,b) across all the states S in the DFA. Assuming that the character set Σ is reduced to a character set with ‘k’ characters after alphabet compression, the resulting DFA consists of S$\times$ k transitions. The cost paid for the alphabet compression is the need to maintain an Alphabet Translation Table (ATT) as shown in Figure 2.10(d). As part of the signature matching, the encoded representation of the incoming character is first looked up from the ATT, before further proceeding with the transition lookup in the alphabet compressed DFA.

Various methods have been proposed in the literature which use alphabet compression to remove the transition redundancy. The method proposed by Brodie et al. [64] introduced the idea of alphabet compression, but only verified the idea on a small set of signatures. Becchi et al. [32] proposed the idea of combining alphabet compression with the improved D^2FA, which showed the possibility of combining alphabet compression with transition compression. Finally, Kong et al. [65] proposed the idea of using multiple alphabet translation tables to compress the redundant transitions in the DFA, wherein alphabet compression alone was used to compress the redundant state transitions in the DFA. However, the transition compression rates achieved by Kong et al. is only of the order of 75%, which is a little low in comparison to the other transition compression mechanisms proposed in the literature.

To summarize, the alphabet compression mechanisms leverage the idea of compressing the indistinguishable characters in a signature set to compress the redundant state transitions in the DFA. The biggest advantage of alphabet compression is the possibility to combine the procedure with other transition compression mechanisms to improve the overall transition compression rate in the resultant compressed DFA.

2.3.3 Summary of DFA Compression

The biggest advantage of using the DFA to represent the signatures is the ability to perform the signature matching function at predictable rates, since the processing complexity associated with the payload byte is always constant. Though the DFA is processing efficient, it is highly storage inefficient. The storage inefficiency associated with the DFA is primarily addressed through the state compression and the transition compression algorithms. The solutions proposed as part of the state compression algorithms primarily focus on compressing the number of states generated in the DFA, while the transition compression algorithms focus on compressing the redundant state transitions. The major focus of the existing research in automata based signature matching has been on efficient representation of the DFA through various compression algorithms. So, the associated signature matching engines are evaluated on the general purpose processor platforms, which results in flexible and scalable implementations with respect to storage; but do not necessarily perform the signature matching function at line rates [66]. The next section discusses the various signature matching engines which have been proposed in the literature primarily targeting line rate signature matching engine implementations.

2.4 Line Rate Signature Matching Engine Implementations

Various signature matching engines have been proposed in the literature to perform signature matching function at line rates. The solutions which have been proposed in the literature are generally targeted across a multitude of platforms such as FPGA, ASIC and GPU. Smith et al. [66] classified these platforms with respect to their capabilities as shown in Table 2.2. The yardsticks which are used for comparison are power efficiency, area efficiency, scalability, flexibility, performance and the cost of implementing the signature matching engine in a specific platform. The various signature matching engines which belong to each of the target platforms have been compared based on the yardsticks mentioned above in this section. The suitability of each of the specific platforms for signature matching in the RGR is also discussed in this section.

Table 2.2: Comparison of various platforms with respect to signature matching engine implementation

Yardstick	CPU	FPGA	GPU	ASIC
Power Efficiency	Lowest	Medium	High	Highest
Area Efficiency	Low	Worst	High	Highest
Scalability	High	Low	Medium	High
Flexibility	Best	Medium	Medium	Worst
Performance	Lowest	Medium	Medium	Highest
Cost	Lowest	Medium	Medium	Highest

2.4.1 FPGA - Logic Based Implementations

FPGAs have been one of the target platforms which has been used by line rate signature matching implementations, mostly to perform logic based signature matching. The solutions which have been proposed in this category primarily use the NFA to represent the signatures and perform NFA based signature matching [67, 68, 69] in the FPGA. These solutions typically use the inherent parallelism available in the FPGA to configure the NFA state transitions as logic circuits. The hardware parallelism available in the FPGA is primarily used to paralellize the computations associated with multiple active states in the NFA. Consequently, the logic based implementations primarily focus on effectively encoding the NFA state transitions into logic circuits [34].

Despite the high-parallelism which is on offer in the FPGAs, there are few challenges in using the FPGAs for multigigabit line rate signature matching applications [34]. Firstly, since the state transitions are directly configured into logic circuits, the maximum number of signatures which can be configured in the FPGA is restricted by the logic resources available in the specific FPGA device. Secondly, the signature definitions with respect to various DPI applications are frequently updated by various sources. Whenever there is an update in the signature database, the corresponding logic representation of the signature sets have to be regenerated for the FPGA, which is quite a time consuming process. Moreover, the clock frequency achieved after mapping the NFA into the FPGA logic elements depends on the characteristics of the signature set. Lastly, the network traffic which is inspected by DPI applications primarily constitutes of a multitude of network streams. However, a single signature set compiled into the FPGA as a logic representation, can only be used to compare a single stream at a time. So, if multiple streams have to be inspected in parallel, multiple copies of the hardware circuits are required to represent the signatures which results in inefficient usage of the FPGA resources affecting the scalability of the signature matching engine. Thus, as shown in Table 2.2, the resulting FPGA implementation is not an area-efficient and a power-efficient solution. On the other hand, the parallalism available through the logic elements in the FPGAs can be used to perform signature matching at gigabit rates.

To summarize, as seen in Table 2.2, though the inherent parallelism in the FPGA allows the signature matching to be perform at line rates, the primary challenges associated with the FPGA based signature matching engines is the scalability. Moreover, for the logic based FPGA implementation approaches to be used in the network processors for the RGR, the logic based elements have to be prefabricated together with the network processor used in the RGR which is also an architectural challenge to be considered.

2.4.2 GPU Based Signature Matching Engines

The GPU primarily consists of multitude of processing cores with abundant memory bandwidth to access the internal and external memories. The parallelism offered by the processing cores and the availability of huge memory bandwidth, which are critical requirements for automata based signature matching, have kindled interest in the research community to use the GPU as a possible platform to perform line rate signature matching. The very first signature matching engine using the GPU was proposed in [70] and only implemented a string based signature matching engine. Various other solutions proposed in the literature have either been NFA based [71, 72] or DFA based implementations [73, 74, 66].

After comparing the various GPU based signature matching engines, Xu et al. [34] concluded that the current GPU based solutions are capable of performing signature matching at gigabit rates only for a small number of signatures. Moreover, Yu et al. [73] pointed out that the GPUs cannot be used for signature matching against complex regular expression patterns where the memory availability in the GPU becomes a bottleneck. Though the GPU provides a possibility to implement a fairly scalable implementation in comparison with the FPGAs, the restriction with respect to automata storage is still a concern to use the GPUs for line rate signature matching solutions. Moreover, not all the current network processor architectures consist of a GPU integrated into the system which is also a challenge to use the GPUs to accelerate the signature matching function in RGR.

2.4.3 Hardware Accelerators - ASIC

There are various advantages when the signature matching function is performed through a dedicated hardware accelerator implemented in an ASIC. Since the signature matching function is performed in a dedicated hardware accelerator, it is possible to perform the signature matching function at line rates. Multiple instances of the accelerator can be used to support increasing throughput as well as increasing signature counts making the implementation scale efficiently. Moreover, the resulting signature matching engine can be optimized to support the stringent area and power requirements associated with the implementation. On the downside, there is a huge cost involved in designing and implementing the accelerator. Furthermore, the signature matching engine implemented as a hardware accelerator is not always flexible, as any changes required in the implementation has to go through a redesign and the fabrication process. Nevertheless, the advantages with respect to high performance, area efficiency, power efficiency, scalability outweigh the disadvantages. Since the network processors designed for the residential gateway routers is a highly cost sensitive market, the advantages associated with an ASIC implementation make it the natural choice for accelerating the signature matching function in these devices.

Various signature matching engines have been proposed in the literature in which the signature matching function is performed in a dedicated hardware accelerator. When the signature matching is performed in a dedicated hardware accelerator, the automata is typically stored in the TCAM or the RAM. Since the processing associated with the signature matching function is accelerated, the latency associated in fetching the automata from the memory makes a huge difference in the throughput achieved by these systems [4, 34]. A short overview of the various implementations are discussed further.

2.4.3.1 Automata Storage - TCAM

The TCAM has been widely used in various network processors for high speed applications such as network packet classification and routing. Since the data which is stored in the TCAM is content addressable, the data from the TCAM can be fetched in a single clock cycle. This feature is especially beneficial for signature matching applications, since the signatures can be stored in the TCAM and fetched in high speeds to perform line rate signature matching. Various solutions have been proposed in the literature which propose the usage of TCAMs to store the automata and the major research focus has been on encoding and storing the state transitions in the content addressable memories. Furthermore, the associated accelerators primarily focus on fetching the state transitions from the TCAMs and checking for the signature match in the sequence of payload bytes.

Initial TCAM based signature matching hardware accelerators [75, 76, 77] were primarily used for string based signature matching solutions, where the signatures were stored using the Aho-Corasick automaton [46]. Later, various DFA based solutions [78, 79, 80] were proposed to also include the regular expressions. The DFA based solutions primarily focused on storing the DFA in its uncompressed form, while the compressed DFA representations such as D^{2}FA, δFA cannot be stored in the TCAM making them memory inefficient solutions [34].

The biggest problem with respect to the usage of TCAMs for signature matching is the issues associated with scalability, power and cost. The TCAMs which are generally available for integration are constrained with a certain fixed size. So multiple TCAMs have to be used to store the signatures to support increasing signature sets which makes the proposal expensive to implement. Moreover, the TCAMs consume 150 times more power per bit than the Static RAM and costs about 30 times more money per bit than the DDR [34]. So, the majority of the TCAM based solutions have only been research proposals while the challenges associated with cost and power make them an impractical solution for commercial implementation.

2.4.3.2 Automata Storage - RAM

Various hardware accelerators [37, 81, 82, 4, 5] have been proposed in the literature which primarily store the automata in the RAM. These accelerators either store the automata in the on-chip memories alone [82, 4, 5] or in a combination of the on-chip and off-chip memories [37, 81]. Similar to the accelerators described previously, the hardware accelerator is primarily responsible for fetching the state transitions corresponding to the state character combination and to check for signature matches in the stream of bytes.

The RegX engine [37] was the very first complete hardware acceleration system which was proposed to perform line rate signature matching and was part of the IBM PowerEN processor. The RegX accelerator leveraged the BFSM [83], a programmable state machine architecture to represent the signatures. The BFSM architecture represents the state transitions as rules, where the rules provide the next state information corresponding to either individual characters or a set of characters. As part of RegX based signature matching, multiple rules corresponding to the payload bytes are searched, from which the specific rule corresponding to the state-character combination is identified. The RegX hardware engine consists of four physical lanes of signature matching engines with each lane consisting of four BFSM engines. Each BFSM engine is equipped with 16KB of on-chip cache memory to store the rules corresponding to the signature sets. If the rules corresponding to the signatures cannot be completely stored in the on-chip SRAM, they are stored in a combination of on-chip SRAM and the off-chip DRAM. Each of the physical lane is optimized to perform signature matching at a peak rate of 18.4 Gbps. Though, this represents the theoretical maximum throughput which the accelerator can achieve, the actual signature matching throughput achieved by each of the physical lanes completely depends on the characteristics of the payload byte sequence inspected by the system and the number of signatures stored in the system. The throughput achieved by the RegX depends on whether the rules are fetched from the on-chip or the off-chip memories. In worst-case scenarios, the hardware accelerator only achieves signature matching throughput of about $\sim$2 Gbps due to the high latency associated with the off-chip memory access [37].

Fang et al., proposed the Unified Automata Processor (UAP) [81] which is also a hardware accelerator that is capable of processing different automata representations such as the NFA, the DFA, the A-DFA[2], the JFA[35], etc. The UAP's architecture is also similar to that of the [37], where multiple individual accelerators are proposed with 16KB of dedicated on-chip memory to store the compressed state transitions. Though, the authors do not specifically discuss about how the automata is stored when the size of the signature sets exceed beyond the on-chip memory boundaries, the UAP will also suffer from throughput limitations similar to that of [37].

On the other hand, FEACAN [4] and RCDFA [5], the bitmap based transition compression algorithms discussed in Section 2.3 also propose a dedicated hardware accelerator to perform signature matching. Unlike the hardware accelerators proposed in

[37, 81], the hardware accelerators proposed in [4, 5] store the complete compressed DFA in the on-chip SRAM alone. In these proposals, since the compressed DFA is completely stored in the on-chip memories, they can be fetched and processed in low latencies. So, the signature matching throughput achieved by these systems is identical to the peak throughput for which they were designed for. On the downside, since these methods compromise the transition compression rates in order to keep the storage requirements associated with the bitmaps to a bare minimal, the hardware implementations typically leads to inefficient on-chip memory usage.

HAWK [82] is a hardware accelerator which also performs signature matching at multi-gigabit rates. Unlike all the accelerators described above which either use the NFA or the DFA to represent the signatures, HAWK uses the bit-split automata representation [84]. In the case of the bit-split automata representation, a single automaton corresponding to the signature set is split and represented as a combination of 8 different bit-split automata. So, each state in the original automata corresponds to a vector of states in the bit-split automata. Each of the bits in the payload byte[12] is compared against the corresponding bit-split automaton representation. A signature match is identified, if and only if an accepted state is triggered by all the states in the bit-split automata. Unlike the other hardware accelerators [37, 81, 4, 5] which consume a single payload byte every clock cycle, HAWK consumes multiple payload bytes every clock cycle. However, the size of the automata explodes exponentially corresponding to the number of payload bytes consumed every clock cycle, which makes HAWK quite a difficult approach to scale. Moreover, in comparison to the normal compressed DFA representation, the bit-split automata representation proposed in HAWK is highly area and power inefficient. Based on the studies reported by the authors in [82], a HAWK engine which is implemented to consume 32 characters in a single clock cycle consumes about 90 mm^2 area when synthesized on a commercial 45nm CMOS technology node. On the other hand, the HAWK engine completely stores the bit-split automata in the on-chip memories and the hardware accelerator is pipelined to run at 1GHz which enables to achieve a signature matching throughput of 8Gbps if 1 byte is consumed every clock cycle. If the HAWK accelerator is architected to consume 32 characters in a single clock cycle, it will be capable of achieving a signature matching throughput of 32Gbps.

2.5 State-of-the-Art Summary

Of the various steps to be performed in DPI, the signature matching is the most time critical step which defines the rate at which DPI can be performed by modern network processors. Since the signature representations are primarily a combination of stings and regular expressions, the automaton has become the preferred choice to represent the signatures. Majority of the solutions proposed in the literature use the DFA to represent the signatures, as the worst case complexity associated in processing a payload byte is always constant. Due to this property, the DFA has been the preferred choice to represent

[12]Since the extended ASCII is used to represent the payload, a single automaton corresponding to the signature set is split into 8 bit split automata.

the signatures, especially for high speed signature matching. However, the DFA is highly storage inefficient due to the state explosion problem associated with complex signature patterns and the transition redundancy. So, a majority of the solutions proposed in the literature focus on reducing the number of generated states during the NFA to DFA conversion process and to compress the redundant transitions in the DFA. Though the automaton compression has been the primary focus of the research community, various solutions have been proposed to target the problem of line rate signature matching. The solutions which have been proposed can be classified into GPU-based, FPGA-based and ASIC based signature matching engines, primarily based on the target platform of implementation on which the signature matching function is accelerated.

Due to the early adoption of the GPU by the research community, various proposals have suggested the usage of the GPU as a potential platform to accelerate automata based signature matching. Though the GPU consists of a large number of processing cores and high internal memory bandwidth, the memory availability in the GPU becomes a bottleneck in processing complex regular expression patterns. On the other hand, though the inherent hardware parallelism in the FPGA can be utilized to accelerate the signature matching function, the signature matching engines implemented as logic circuits in an FPGA are not easily scalable. Moreover, it is quite a time consuming process to map the signatures into hardware circuits in the FPGA. Furthermore, the graphics processors and the programmable logic have to be prefabricated inside the network processors for them to be used as a platform to accelerate the signature matching function.

Hardware accelerators targeting ASIC implementations have been one of the platforms which is targeted for line rate signature matching engine implementations. Of the various solutions proposed in this category, generally the automata have been stored in the RAM or the TCAM with a hardware accelerator performing the signature matching function. Though the TCAMs have been used for various other networking applications, the high power consumption and high cost makes them unsuitable for signature matching applications. Alternately, in some of the hardware accelerators, the compressed DFA is either completely stored in the on-chip SRAMs or in a combination of the on-chip SRAMs and the off-chip DRAMs. When the compressed DFA is stored in a combination of on-chip and off-chip memories, the signature matching throughput achieved by the accelerator is variable and depends on how quickly the data can be fetched from the off-chip memories.

On the other hand, when the compressed DFA is completely stored in the on-chip memories as in the case of bitmap based accelerators, the hardware can access the data in low latencies to efficiently perform the signature matching function at *guaranteed* multi-gigabit line rates. However, the memory inefficiency in the bitmap based accelerators is a downside of using these accelerators for signature matching and is a topic which has to be addressed to complement the line rate signature matching ability of the bitmap based accelerators. The primary problem associated with the bitmap based compression algorithms is the necessity to store the bitmaps along with the DFA. So, the bitmap based compression algorithms proposed in the literature reduce the number of bitmaps by storing additional redundant transitions in memory which heavily increases the memory footprint of the compressed DFA. However, this problem can be addressed in a different way by using a secondary layer of indexing in addition to bitmap based compression. The secondary layer of indexing can greatly reduce the number of redundant transitions stored in the memory, thereby greatly reducing the memory footprint of the compressed DFA. Based on this approach, two different transition compression algorithms are proposed in Chapter 3. The subsequent chapters further propose additional optimization schemes for the proposed bitmap based algorithms before defining a bitmap based hardware accelerator to perform signature matching multigigabit rates.

3 Bitmask: A Secondary Indexing Layer for Bitmap based Transition Compression

As seen in Chapter 2, the bitmaps possess attractive properties which make them a good choice to compress the redundant state transitions in the DFA. In comparison to the software oriented transition compression methods such as [1, 32, 3, 36], the approach towards transition compression in the bitmap based methods [4, 5] allow the signature matching function to be performed in a dedicated hardware accelerator. This in turn enables to perform deep packet inspection at multi gigabit line rates.

After bitmap based transition compression, the content of the generated compressed DFA can be classified into the transition and the control data. The transition data primarily refers to the compressed state transitions, while the control data refers to information such as bitmaps which help to locate the compressed state transition corresponding to a state-character combination. As part of the compressed DFA storage, both the transition and the control data is stored in the on-chip memories. The existing bitmap based transition compression algorithms primarily focus on reducing the number of unique bitmaps which are stored together with the compressed state transitions. However, in this process, a heavy compromise is made on the transition compression rates which results in excessive redundant transitions stored as part of the transition data. So, the biggest challenge associated with the bitmap based compression methods is the need to balance the storage of the transition and the control data.

Addressing this problem, this chapter introduces the idea of using *bitmasks* to compress the redundant state transitions in the DFA after performing bitmap based transition compression. A bitmask is a secondary layer of indexing which effectively identifies the redundant state transitions in the DFA after performing bitmap based transition compression. Though the usage of bitmasks increases the overall control data in the compressed DFA, the redundant transitions are effectively indexed and compressed resulting in high transition compression rates. Consequently, the reduction in the transition data resulting from the efficient compression of the DFA will result in a memory efficient representation of the overall compressed DFA.

In this chapter, two different transition compression methods are first introduced which use the combination of the bitmap and the bitmask to efficiently compress the DFA. After introducing the two methods, this chapter further evaluates the effectiveness of the proposed methods against the state-of-the-art bitmap based transition compression algorithms. The first method is called the *Member State Bitmask Technique* (MSBT) while the second approach is called the *Leader State Compression Technique* (LSCT). The MSBT and the LSCT are further explained in detail in the following sections.

3.1 Member State Bitmask Technique

The MSBT is a three step compression mechanism which eliminates the intra-state and inter-state transition redundancy in the DFA. The three steps which are part of the MSBT are described briefly below:

- **Intra-State Transition Compression:** The first step in the MSBT uses the bitmap to compress the redundant state transitions along the character axis. In this step, the bitmap compresses the identical state transitions which are adjacent to each other in each of the states in the DFA, thus effectively eliminating the intra-state transition redundancy.
- **State Grouping:** In the second step, the states are grouped into subsets of states in order to prepare the DFA for the third inter-state compression step. The state grouping algorithm proposed in FEACAN [4] is used to group the states in this step. As described in the previous chapter, the state grouping algorithm groups those states which have identical bitmaps and a certain minimum number identical state transitions defined by the transition threshold, 'T'. In this step, one of the states in the group is defined as the leader state, while the rest of the states are called the member states.
- **Inter-State Transition Compression:** As part of the inter-state transition compression step, the redundant state transitions in the member states are compressed using the *Member Transition Bitmask* (MTB). In this step, the state transition at each and every index in a member state is compared against the corresponding state transition in the leader state. If the state transitions are identical, they are compressed. The member transition bitmask is generated for each of the member state to identify those specific indices at which the state transitions were compressed as part of the inter-state compression step.

The various steps in the MSBT and the bitmask generation process are explained further through an example below.

3.1.1 MSBT - An Example

Figure 3.1(a) shows an uncompressed DFA with state transitions for 8 states and 8 characters[1]. Figure 3.1(b) shows the compressed DFA after the intra-state transition compression step. In this step, the state transitions corresponding to character 'h' in states 0, 2, 3, 4 and 7 are compressed as they are identical to the state transition corresponding to character 'g' in their respective states. After the intra-state compression step, the bitmap corresponding to the states either belong to the bitmap pattern BMP0 (11111110) or to the bitmap pattern BMP1 (11111111). The bitmaps are represented in Figure 3.1(b) through separate colors for easy identification.

[1]It should be noted that the example considered to explain the MSBT transition compression is identical to the example which was used in Chapter 2 to describe the bitmap based signature matching solutions.

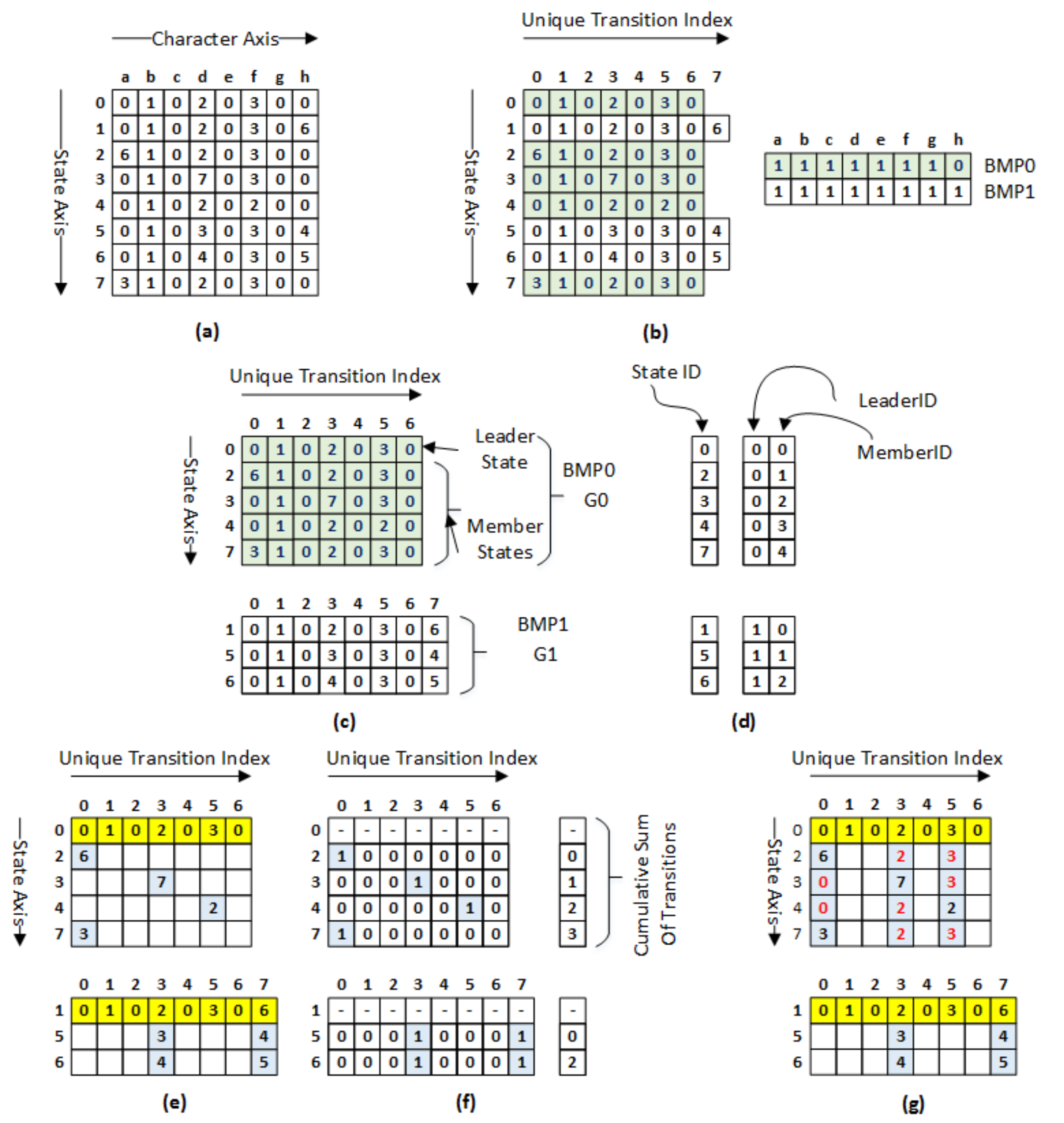


Figure 3.1: (a) Original DFA before compression (b) The DFA after bitmap based intra-state transition compression (c) The DFA states grouped into subset of states (d) Encoded state representation of the DFA states (e) The DFA after inter-state transition compression (f) The MTB and the cumulative sum of transitions after inter-state compression (g) Compressed DFA generated after FEACAN

Figure 3.1(c) shows the states being grouped into subsets of states as part of the state grouping step. The transition threshold T, is set to 75%, i.e., the states are grouped into subsets of states if and only if they share a minimum of 75% of identical state transitions. After the state grouping step, the states 0, 2, 3, 4 and 7 are grouped into a subset (G0), while states 1, 5 and 6 are grouped into another subset (G1) based on the predefined clustering constraints. After the state grouping step, the states '0' and '1' are designated as the leader states for groups G0 and G1, respectively. The other states in the groups are referred to as the member states.

The last and final step in the MSBT is the inter-state compression step. In this step, the state transitions between the member states and the leader state are individually compared at every unique transition index. If the state transition at an index is identical to the state transition in the leader state, then the state transition in the member state is compressed. In order to identify the indices at which the state transitions are compressed in the member states, a member transition bitmask (MTB) is generated for each of the member state. If there are k state transitions in a member state after the intra-state compression step, a k-bit MTB is populated to identify the indices at which the state transitions are compressed as part of the inter-state compression step. If the state transition in the member and the leader state are identical at a unique transition index, then the bit position corresponding to the index is marked '0' in the MTB. If not, the bit position is marked '1' in the MTB.

Figure 3.1(e) shows the compressed DFA after the inter-state compression step, while Figure 3.1(f) shows the MTB which is populated for the member states. For example, the state transition at the unique transition index *0* in state *2* is *6* while the state transition at the corresponding index in state *0* is *0*. Since these two state transitions are different, the state transition at index 0 in member state 2 is not compressed and the bit position corresponding to index 0 in the MTB for state 2 is set to 1. On the other hand, the state transition at the unique transition index *1* in state *2* and state *0* is *1* and are identical. So, the state transition corresponding to the unique transition index 1 in state 2 is compressed and the bit position corresponding to this index is set to 0 in the MTB. In this way, the MTB for all the member states is populated as shown in Figure 3.1(f). The empty blocks in Figure 3.1(e) are the state transitions in the member states which are compressed as part of the inter-state compression step in the MSBT in the considered example. Thus, in this way, the MTB is used as the secondary layer of indexing to compress the redundant state transitions after performing bitmap based transition compression.

The state transitions which remain uncompressed after the MSBT are the ones which are stored in the memory as part of the compressed DFA. The compressed state transitions belonging to the leader states (yellow blocks in Figure 3.1(e)) are called as the leader transitions, while the compressed state transitions which belong to the member states are called as the member transitions (blue blocks in Figure 3.1(e)). It can be seen from Figure 3.1(e) that a total of 41 state transitions are compressed using the MSBT in comparison to 33 transitions that were compressed in FEACAN seen in Figure 3.1(g) on the same DFA. The primary difference between both the compression methods is the introduction of the member transition bitmask, which is the secondary layer of indexing that is used to identify the redundant state transitions between the states.

The cost paid for the additional compression achieved is the memory required to store the MTB. As mentioned earlier, the number of unique transition indices in the states after the intra-state compression step decides the width of the MTB for each of the member states. For example, since there are 7 unique transition indices for the states in G0, this results in a 7-bit MTB for each member state in the group. In order to efficiently store the MTB, additional 0's are padded to make them byte aligned. This helps to easily store and fetch the MTB from the memory.

As part of the MSBT, since the states are clustered into groups, the original state identifiers used for the states cannot be directly used to identify the states after state grouping. Consequently, the states are encoded and represented as a combination of a leader identifier (leaderID) and a member identifier (memberID). The leaderID identifies the group to which a state belongs to, while the memberID is used to identify each of the individual states within a group. The memberID for the leader state is set '0' to easily differentiate between the leader state and the member states in a group. Figure 3.1(d), shows the encoded state representation.

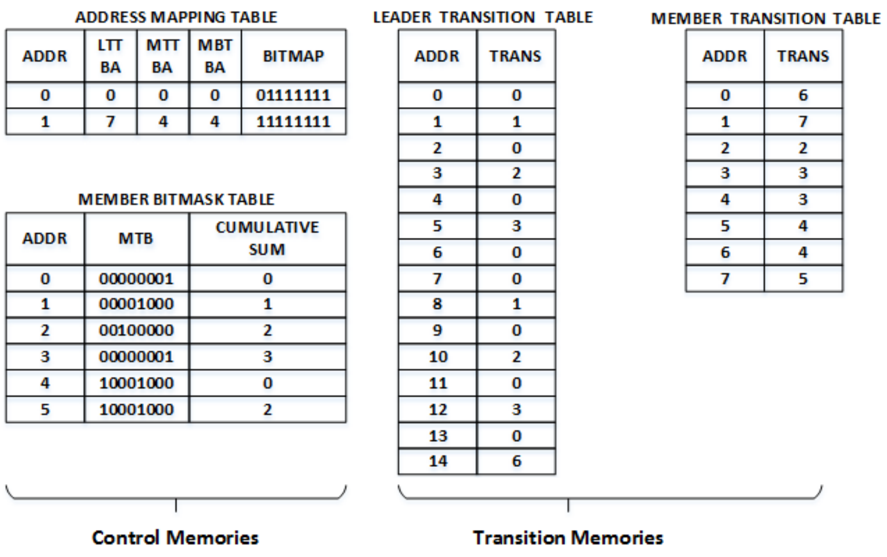
3.1.2 Compressed DFA Organization

The compressed DFA which is generated after the MSBT is organized into four different tables (or memories). The leader transitions across all the leader states are consolidated and stored in the Leader Transition Table (LTT), while the member transitions across all the member states are consolidated and stored in the Member Transition Table (MTT). Since the LTT and the MTT only store the compressed transitions, they are collectively called as the *'Transition Memories'*.

Apart from the LTT and the MTT, two other tables are used to only store the control information which is used to locate the compressed transitions from the transition memories. The MTBs across all the member states, across all the groups are consolidated and stored in the Member Bitmask Table (MBT). The Address Mapping Table (AMT) stores a multitude of control information for each of the groups. The control information include the bitmap, the address of the first compressed leader and member state transition in the group and the address of the first MTB in the group. The address location of the first leader transition, member transition and member transition bitmask are also called as the leader base address, member base address and the bitmask base address for the group.

As far as the member base address, the address of the first member transition in the first member state is stored in the address mapping table. So, a cumulative sum of all the member transitions which remain uncompressed until each member state in a group is generated along with the MTBs as shown in Figure 3.1(f). For example, a cumulative sum of '3', corresponding to state 7 represents that 3 member transitions are stored in memory before the first uncompressed member transition belonging to state 7 is stored. This information is used to precisely locate the compressed member transition in the state. The MBT and the AMT are collectively called as the *'Control Memories'*, since they store the control information that is primarily used to locate the compressed transition from the transition memories.

Figure 3.2 shows the compressed DFA organization corresponding to the example shown in Figure 3.1. The compressed transitions which belong to the leader states shown in yellow in Figure 3.1(e) are sequentially arranged in the leader transition table. Similarly, the compressed transitions which belong to the member states shown in blue



ADDRESS MAPPING TABLE

ADDR	LTT BA	MTT BA	MBT BA	BITMAP
0	0	0	0	01111111
1	7	4	4	11111111

MEMBER BITMASK TABLE

ADDR	MTB	CUMULATIVE SUM
0	00000001	0
1	00001000	1
2	00100000	2
3	00000001	3
4	10001000	0
5	10001000	2

LEADER TRANSITION TABLE

ADDR	TRANS
0	0
1	1
2	0
3	2
4	0
5	3
6	0
7	0
8	1
9	0
10	2
11	0
12	3
13	0
14	6

MEMBER TRANSITION TABLE

ADDR	TRANS
0	6
1	7
2	2
3	3
4	3
5	4
6	4
7	5

Control Memories
Transition Memories

Figure 3.2: The compressed transitions stored in the Leader and Member Transitions Tables while the control information is stored in Address Mapping Table and Member Bitmask Table

in Figure 3.1(e) are sequentially arranged in the member transition table. It can be seen that the MTBs seen in Figure 3.1(f) are sequentially stored in the member bitmask table seen in Figure 3.2. It can be seen in Figure 3.2 that the address mapping table stores the bitmap and the base addresses for each of the groups in it. It should be noted that all the data in the tables are stored first for G0, followed by G1 in a successive manner.

The transitions shown in Figure 3.2 are shown in their pre-encoded form to explain the idea behind the memory organization. In actual implementations, the transitions are stored in their encoded form, i.e., as a combination of the leaderID and the memberID. Figure 3.2 shows the bitmap and the bitmask in the binary format while all the other data such as the stateIDs, addresses etc., are represented in the decimal number format for simplicity. The bitmap corresponding to the character ‘a’ is shown in the right (least significant bit position), while the bitmap corresponding to the character ‘h’ is shown in the left (most significant bit position). Similarly, the MTB corresponding to the unique transition index ‘0’ is shown in right, while the one corresponding to the index ‘7’ is shown in the left.

3.1.3 Decompression

This section details how the compressed state transition corresponding to a state-character combination is fetched from the above mentioned tables. As part of the decompression, the current state is first decoded into its leaderID and the memberID. If the memberID corresponding to the current state is ‘0’, then the the current state is a leader state, while it is a member state if not. If the current state is a leader state, then the compressed state transition is identified from the leader transition table. The location of the compressed state transition is calculated by performing the population count operation on the bitmap which is added to the leader base address fetched from the address mapping table.

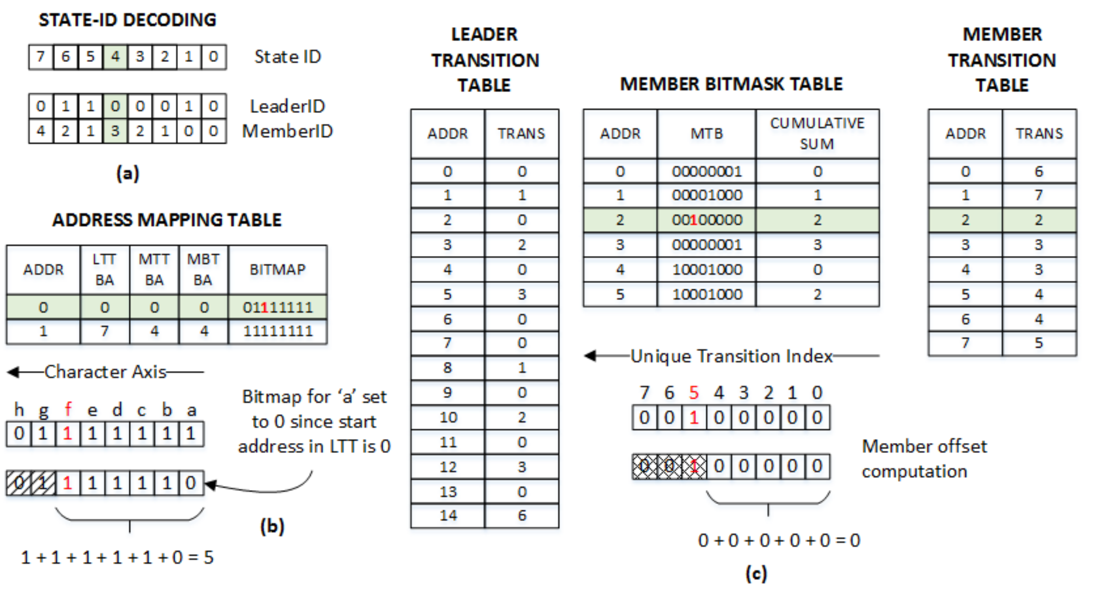
On the other hand, if the current state is a member state, the bitmask bit corresponding to the unique transition index in the MTB decides the next state. If the bitmask bit at the unique transition index is ‘1’, it signifies that the transition corresponding to the index is not compressed. So, the compressed state transition is fetched from the member transition table. The location of the transition in the table is calculated by performing a population count operation on the MTB, which is added to the cumulative sum of transitions and the member base address. If the bitmask bit corresponding to the unique transition index is ‘0’, it signifies that the state transition in the member state was compressed during inter-state compression step. In such a case, the compressed state transition is fetched from the leader transition table similar to how the compressed transition was fetched in the case of the leader state.

3.1.3.1 Decompression - An Example

The compressed state transition fetch for character ‘f’ in state ‘4’ is used as an example to explain the decompression process. In Figure 3.3, the specific locations in the tables which are accessed are highlighted in green, while the bits of interest in the bitmap and the MTB are highlighted in red.

As shown in Figure 3.3(a), the leaderID and the memberID corresponding to state ‘4’ are ‘0’ and ‘3’, respectively. Since the memberID of the state has a non-zero value, the current state is a member state. The leaderID, i.e., ‘0’ is used as the address to fetch the various base addresses and the bitmap from the address mapping table. As shown in Figure 3.3(b), the population count operation is performed on the bitmap to generate the unique transition index corresponding to the character. This is also referred to as the leader offset.

Since state ‘4’ is a member state in G0, the bitmask bit at the leader offset position is first checked to identify the location of the compressed transition. So, the MTB is first extracted from the member bitmask table. The address of the MTB is directly calculated by adding the memberID to the bitmask base address (MBT BA). Since the memberID and the bitmask base address for G0 are 2 and 0 respectively, the MTB and the cumulative sum of transitions are fetched from location ‘2’ in the member bitmask table. After locating the MTB, it is examined to identify if the state transition of interest was compressed during the inter-state compression step. Since the bitmask bit at the leader offset position is ‘1’, the transition corresponding to the state character combination is not compressed and has to be fetched from the member transition table. The location of the compressed state transition is calculated by adding an offset address to the member base address. The offset address consists of two components, i.e., the cumulative sum of transitions which identifies the total number of member transitions stored prior to the member state of interest and the member offset, which is calculated by performing the population count operation on the MTB. The member offset calculation shown in Figure 3.3(c) identifies the offset address of the compressed state transition of interest



	7	6	5	4	3	2	1	0
State ID	7	6	5	4	3	2	1	0
LeaderID	0	1	1	0	0	0	1	0
MemberID	4	2	1	3	2	1	0	0

ADDRESS MAPPING TABLE

ADDR	LTT BA	MTT BA	MBT BA	BITMAP
0	0	0	0	01111111
1	7	4	4	11111111

h	g	f	e	d	c	b	a
0	1	1	1	1	1	1	1
		1	1	1	1	1	0

LEADER TRANSITION TABLE

ADDR	TRANS
0	0
1	1
2	0
3	2
4	0
5	3
6	0
7	0
8	1
9	0
10	2
11	0
12	3
13	0
14	6

MEMBER BITMASK TABLE

ADDR	MTB	CUMULATIVE SUM
0	00000001	0
1	00001000	1
2	00100000	2
3	00000001	3
4	10001000	0
5	10001000	2

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
			0	0	0	0	0

MEMBER TRANSITION TABLE

ADDR	TRANS
0	6
1	7
2	2
3	3
4	3
5	4
6	4
7	5

Figure 3.3: Representation of the compressed DFA with respect to MSBT decompression system

within the member transitions corresponding to the member state. In this example, the member base address, the member offset and the cumulative sum of transitions are '0', '0' and '2', respectively. So, the compressed state transition is fetched from address location '2' in the member transition table, which is the same as the transition corresponding to the state transition for '4', 'f' seen in Figure 3.1(a).

3.1.3.2 Hardware Decompression Engine for MSBT - Logical Block Level Description

Similar to the bitmap based transition compression techniques proposed in the literature, the decompression through the MSBT can be performed in a dedicated hardware accelerator. A short logical block level description of the hardware accelerator is provided in this section while a detailed architecture of the accelerator is described in Chapter 5. Figure 3.4 shows the block level representation of the decompression engine. The hardware accelerator is composed of three processing stages and four memory blocks. The focus of this representation is to show how the various functions described above are mapped to each of the hardware processing stages. Each of the processing stage consists of a memory lookup followed by a combinatorial function to calculate the address for the memory lookup in the subsequent stage. The sequential logic in the processing stages and the memories are synchronized using the same clock signal. So, it would take a minimum of three clock cycles to fetch the compressed transition from the transition memories corresponding to a state-character combination.

The first stage is called as the *Address Lookup Stage* (ALS). In this stage, the current state and the character are taken as the inputs, to primarily generate the address

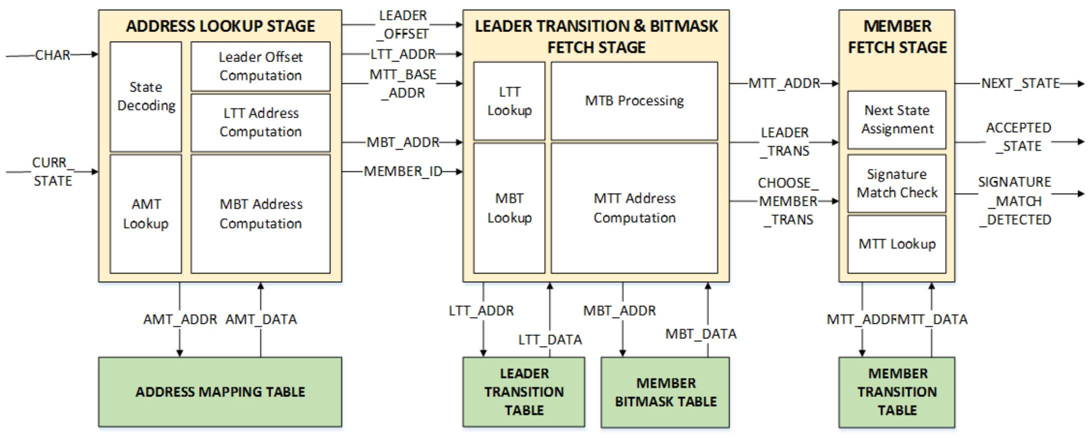


Figure 3.4: Functional description of the hardware based decompression architecture for MSBT

locations for the leader transition table and the member bitmask table. The leaderID, decoded from the current state is first used to fetch the data from the address mapping table. The logic calculations related to the leader transition table and the member bitmask table address calculations are performed in parallel in separate hardware blocks.

The second stage is called the *Leader Transition and Bitmask Fetch Stage* (LTBFS). The leader transition corresponding to the unique transition index is fetched from the leader transition table in this stage. Further, the MTB from the memory is fetched from the pre-calculated member bitmask table address location. Then the bitmask bit corresponding to the leader offset is identified in this stage. Simultaneously, the address location of the compressed state transition to be fetched from the member transition table is also calculated in a separate hardware block in this stage.

The third stage is called the *Member Fetch Stage* (MFS). The third stage fetches the member transition from the computed member transition table address location, if required. As explained earlier, the transition is fetched from the member transition table, if and only if the current state is a member state and the bitmask bit corresponding to the unique transition index is ‘1’. Subsequently, the next state is assigned from the state transition which was either fetched from one of the transition tables in this stage. Moreover, if a signature match is detected after the next state assignment, the signature match detected signal is driven to ‘1’ for a single clock cycle.

3.1.4 Summary

There are various advantages when the bitmap is used to compress the redundant state transitions in the DFA. Performing the transition compression using the bitmap allows the decompression function to be performed in a dedicated hardware accelerator at fixed, but guaranteed rates. Though the bitmap based transition compression has this advantage, the various bitmap based techniques proposed in the literature do not efficiently compress the DFA. In order to reduce the memory requirements associated with the bitmap storage, the existing bitmap based compression algorithms do not compress the DFA efficiently and store a significant number of redundant transitions in the memory.

Considering the advantages of bitmap based transition compression techniques, a novel two-dimensional transition compression method called MSBT was proposed in this section. The MSBT uses a combination of bitmaps and an additional secondary indexing layer called bitmask to efficiently compress the redundant state transitions in the DFA. In the next section, an extension to the MSBT called the Leader State Transition Technique is proposed to further improve the transition compression rates through the usage of the bitmap and the bitmask.

3.2 Leader State Compression Technique

The state transitions belonging to a state in the DFA can primarily be split into the forward transitions and the failure transitions. The forward transitions are those state transitions which progressively lead towards a signature match and the failure transitions are those which do not. A majority of the state transitions in a state are the failure transitions as only a few ASCII characters within a state lead towards a signature match. The failure transitions in a state either leads the DFA to the root state or to one of the states which is closer to the root state. The organization of the forward and the failure transitions in a state completely depends on the combination of characters which are found in the signature set. Given that there is an uncertainty in the organization of the forward and failure transitions within a state, there is always a possibility that a forward transition occurs between a sequence of failure transitions. In such a case, the bitmap alone may not always be effective in compressing the redundant state transitions in a state.

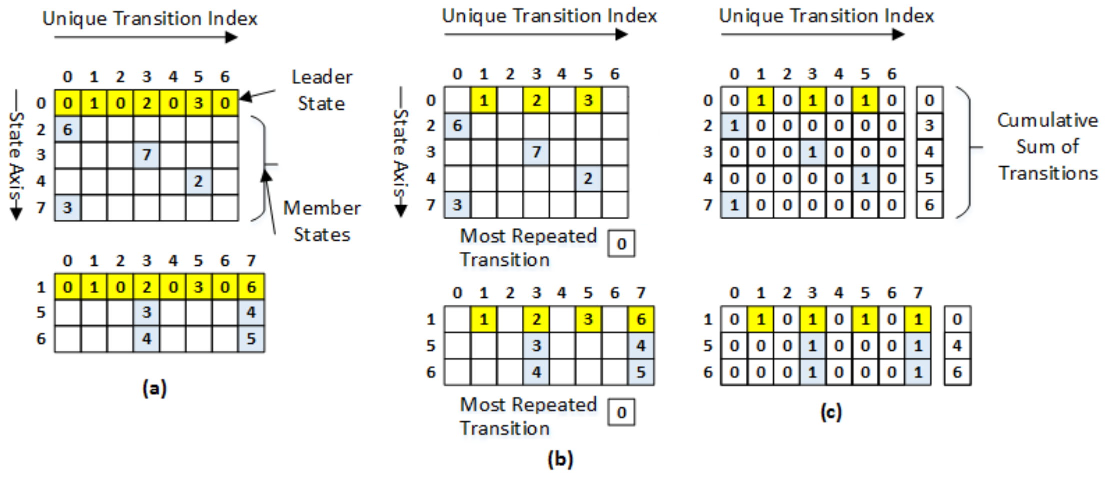


Figure 3.5: (a) Compressed DFA after the MSBT (b) Compressed DFA after LSCT (c) The LTB & the MTB for each state

In the MSBT, the redundant state transitions in the member states are first compressed using the bitmap and then through the MTB. So, if a redundant state transition is not compressed in a member state just because it is blocked by a forward transition, it can possibly be compressed during the inter-state compression step through the MTB. On the other hand, in the case of the leader state, there is no secondary indexing after bitmap based compression. Due to this reason, some of the redundant state transitions in the leader states are not effectively compressed. Figure 3.5(a) shows the compressed transitions generated after the MSBT for the DFA considered in Figure 3.1(a). The state transition corresponding to the unique transition indices 0, 2, 4 and 6 in state 0, which is the leader state of G0 leads to the same next state. It should be noted that these state transitions are identical but are not adjacent to each other and are not compressed just because they are blocked by several state transitions which are different from them.

In order to address this inefficiency in the transition compression in the leader states, a secondary layer of indexing called the *Leader Transition Bitmask* (LTB) is proposed. The idea behind the usage of the LTB to compress the redundant state transitions in the leader states is an extension of the MSBT and is called as the Leader State Compression Technique (LSCT). The details behind the LSCT and the LTB are detailed below.

3.2.1 Leader Transition Bitmask

As part of the LSCT, a single most repeated state transition is identified in the leader state by sorting all the state transitions in the unique transition list based on the frequency of their occurrence. Generic sorting algorithms such as quick sort [85] can be used to identify the most repeated transition. After determining the single most repeated state transition, an LTB is created for the leader state. A state transition in the unique transition list is compressed if it is identical to the most repeated state transition. The bit position corresponding to the unique transition index in the LTB is set to 0, if the index contains the most repeated state transition. If not, the bit position corresponding to the LTB is set to 1. So, in this way, the most repeated transition which is likely to be the failure transition need not be stored multiple times. The LTB is populated after performing the inter-state compression step.

Since the LSCT is a logical extension of the MSBT, the intra-state compression and the state grouping steps are also performed in the LSCT. However, as part of the inter-state compression step in the LSCT, the redundant state transitions are first compressed using the MTBs and then through the LTB. Figure 3.5(b) shows the compressed DFA after the inter-state compression step in the LSCT. In this particular example, the most repeated state transition in both the leader states is ‘0’. After the LSCT, the LTB and the MTBs corresponding to the leader and the member states are shown in Figure 3.5(c). For example, the entry corresponding to the unique transition index ‘2’ for state ‘0’ has a ‘0’ in the LTB. This represents that the leader transition corresponding to the entry is the most repeated transition. On the other hand, the entry corresponding to the unique transition index ‘3’ in state ‘0’ has a ‘1’ in the LTB, which represents that the leader transition corresponding to the entry is different from the most repeated transition.

As seen in Figure 3.5(b), the compressed DFA after the LSCT only consists of a total of 17 compressed transitions[2] in comparison to 23 compressed transitions in the case of the MSBT. The cost paid for the additional compression is the control memory used to store the LTB for each leader state. However, this additional storage cost results in an improved transition compression, and in turn results in a reduced transition memory usage. Since the LSCT is an improvement over the MSBT, the core ideas with respect the state encoding and the bitmask storage are retained for implementation purposes.

3.2.2 Compressed DFA Organization - LSCT

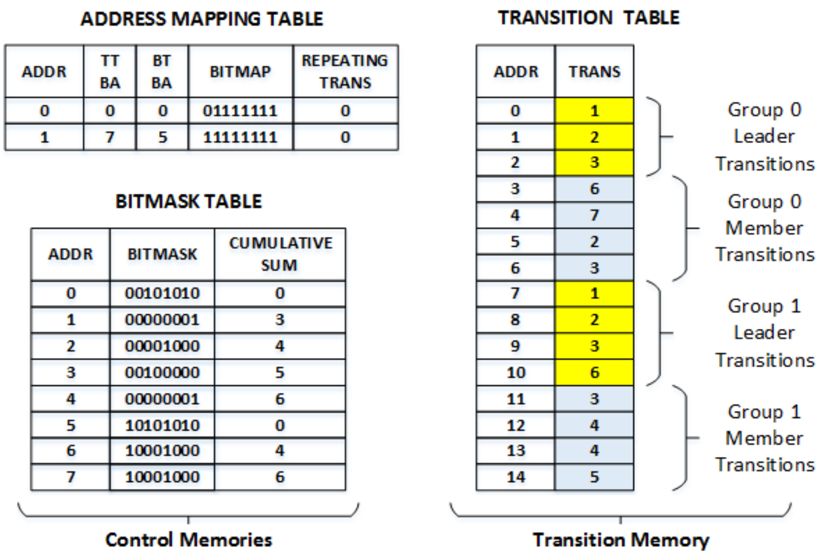
The compressed DFA generated after the LSCT is organized into three tables. The compressed DFA organization in the LSCT is very similar to that of the MSBT and the tables are split into the transition and the control memories. The compressed state transitions across all the states (both leader and member) from all the groups are consolidated and stored in a single Transition Table (TT). The Bitmask Table (BT) and the Address Mapping Table (AMT) belong to the control memories. The bitmask table consolidates and stores the LTBs and the MTBs across all the states from all the groups. Similar to the bitmask storage in the MSBT, the cumulative sum of transitions is stored together with the MTB and the LTB. The address mapping table stores the address of the first bitmask (LTB in this case), the address of the first compressed transition, the bitmap and the most repeated state transition for each of the groups. The address of the first bitmask and the first compressed transition are referred to as the bitmask base address and the transition base address respectively.

Figure 3.6 shows the organization of the compressed DFA into the transition and control data for the example considered in Figure 3.5. The storage of the compressed transitions in the transition table is showed on the right in Figure 3.6. For each of the groups, the leader transitions are stored first, followed by the member transitions. Similar to the MSBT, the compressed transitions corresponding to G0 are stored first followed by G1. As far as the bitmask table, the LTB is stored first followed by the MTBs for all the member states along with the cumulative sum of transitions. As seen in Figure 3.6, the address mapping table stores the base addresses along with the bitmap and the most repeated transition.

3.2.3 Decompression

The decompression process in the LSCT is very similar to that of the one proposed in the MSBT. As a first step, the current state is decoded into its leaderID and the memberID to identify if the current state is a leader or a member state. Irrespective of whether the current state is a leader or a member state, the population count operation is performed on the bitmap to identify the unique transition index corresponding to the incoming character. The unique transition index is also referred to as the transition offset (similar to the leader offset in the MSBT) in the case of the LSCT.

[2]The total number of compressed transitions generated is the sum of the leader transitions, the member transitions and the most repeated state transition per leader state



ADDRESS MAPPING TABLE

ADDR	TT BA	BT BA	BITMAP	REPEATING TRANS
0	0	0	01111111	0
1	7	5	11111111	0

BITMASK TABLE

ADDR	BITMASK	CUMULATIVE SUM
0	00101010	0
1	00000001	3
2	00001000	4
3	00100000	5
4	00000001	6
5	10101010	0
6	10001000	4
7	10001000	6

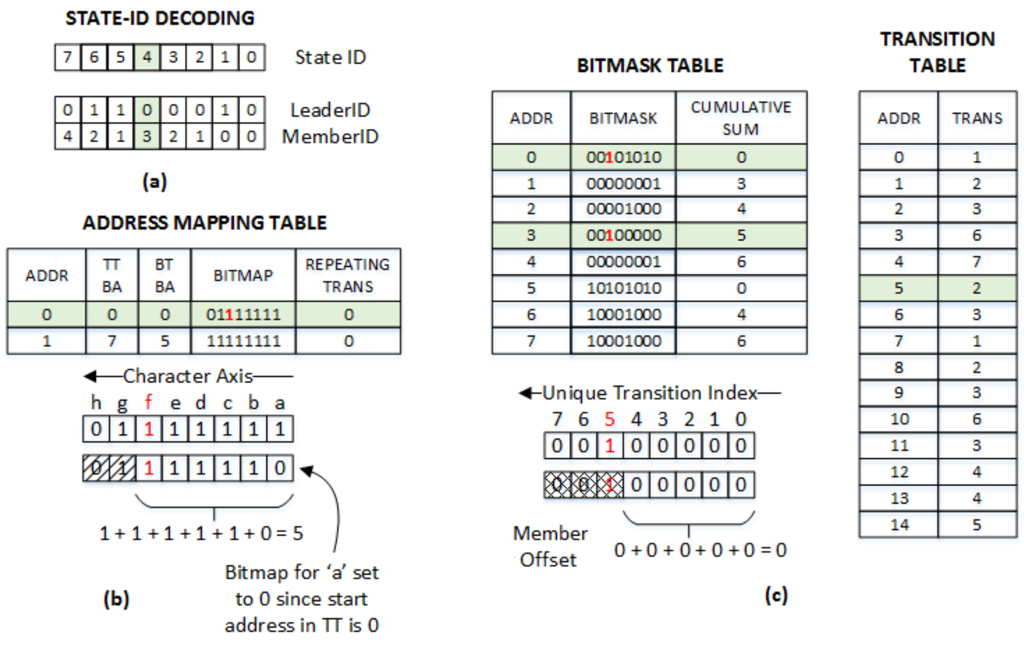
TRANSITION TABLE

ADDR	TRANS	
0	1	Group 0 Leader Transitions
1	2	
2	3	
3	6	Group 0 Member Transitions
4	7	
5	2	
6	3	
7	1	Group 1 Leader Transitions
8	2	
9	3	
10	6	
11	3	Group 1 Member Transitions
12	4	
13	4	
14	5	

Figure 3.6: The compressed DFA organized into the AMT, BT and the TT after LSCT transition compression

If the current state is a leader state, then the bit position corresponding to the unique transition index in the LTB is first investigated. If the bitmask bit is '1', this represents that the state transition corresponding to the character is not compressed during the inter-state compression step and the compressed transition is fetched from the transition table. The location of the compressed transition is calculated by performing the population count operation on the LTB which is added to the transition base address of the group. If the bit position at the unique transition index in the LTB is '0', then the most repeated state transition is directly assigned as the next state.

If the current state is a member state, then the bit position at the unique transition index is first investigated in the MTB, and then in the LTB if needed. If the bit position corresponding to the unique transition index in the MTB is '1', this represents that the state transition of interest was not compressed during the inter-state compression step. So, the compressed state transition is fetched from the transition table. The location of the compressed transition is calculated by performing the population count operation on the MTB which is added to the cumulative sum of transitions and the transition base address of the group. If the bitmask bit in the MTB is '0', this represents that the state transition corresponding to the state character combination was compressed during the inter-state compression step. So, in this case, the state transition in the leader state corresponding to the unique transition index is assigned as the next state. The steps which were carried out to identify the compressed transition in the leader state are followed in this scenario.



State ID	7	6	5	4	3	2	1	0
LeaderID	0	1	1	0	0	0	1	0
MemberID	4	2	1	3	2	1	0	0

ADDR	TT BA	BT BA	BITMAP	REPEATING TRANS
0	0	0	01111111	0
1	7	5	11111111	0

ADDR	BITMASK	CUMULATIVE SUM
0	00101010	0
1	00000001	3
2	00001000	4
3	00100000	5
4	00000001	6
5	10101010	0
6	10001000	4
7	10001000	6

ADDR	TRANS
0	1
1	2
2	3
3	6
4	7
5	2
6	3
7	1
8	2
9	3
10	6
11	3
12	4
13	4
14	5

Figure 3.7: Representation of the compressed DFA with respect to the LSCT decompression system

3.2.3.1 Decompression - An Example

The transition fetch corresponding to character 'f' in state '4' in the DFA shown in Figure 3.1(a) is used as an example to explain the decompression process. The memory locations which are accessed as part of the decompression are shown in green, while the specific bits of interest are shown in red in Figure 3.7.

As shown in Figure 3.7(a), state '4' is first decomposed into its leaderID and the memberID. Since the memberID corresponding to state '4' is 3, it is a member state in group 0 (since the leaderID is 0). The leaderID is used to fetch the base addresses, the bitmap and the most repeated transition from the address mapping table. The bit position corresponding to character 'f' in the bitmap is '1'. This represents that the state transition corresponding to character 'f' in state '4' was not compressed in the intra-state compression step. The unique transition index corresponding to character 'f' is '5' and is calculated by performing the population count operation as shown in Figure 3.7(b).

As a next step, the LTB and the MTB are fetched from address locations '0' and '3' from the bitmask table, respectively. The LTB address is the same as the bitmask base address stored in the address mapping table, while the memberID added to the bitmask base address generates the address location from which the MTB is fetched. Since state '4' is a member state, the MTB is first investigated. The bit position corresponding to the unique transition index is in the MTB is '1'. So, this represents that the state transition of interest was not compressed during the inter-state compression step. So, the location of the state transition to be fetched from the transition table is calculated by adding 0, 0 and 5, which are the transition base address, member offset and the cumulative sum of transitions, respectively. The member offset is calculated by performing the population count operation on the MTB as shown in Figure 3.7(c). The compressed state transition is fetched from the transition table address location '5'. The compressed transition, '2' which is fetched from the transition table is the same as seen in the uncompressed DFA shown in Figure 3.1(a).

3.2.3.2 Hardware Decompression Engine for LSCT - Logical Block Level Description

Figure 3.8 shows the logical block level description of the accelerator which is used to perform the decompression corresponding to the LSCT. Similar to the one shown in the MSBT, the intention of this section is to show the mapping of the specific functions which are performed as part of the decompression into a hardware accelerator system. The decompression engine consists of three processing stages. Similar to the MSBT, each stage consists of a memory lookup followed by a logic function to generate the address for the memory access in the subsequent stage. The sequential elements in the logic block and the memories are synchronized using the same clock signal. So, it would take a minimum of three clock cycles to identify the compressed transition corresponding to the state character combination.

The first stage is called as the *Address Lookup Stage (ALS)*. In this stage, the state identifier is first decoded into the leaderID and the memberID from which the leaderID is used to fetch the data from the address mapping table. The transition offset computation is performed from the bitmap, as it is required to scrutinize the data in the LTB and the MTB. Simultaneously, the address locations from which the LTB and the MTB have to be fetched are calculated in parallel to the transition offset computation.

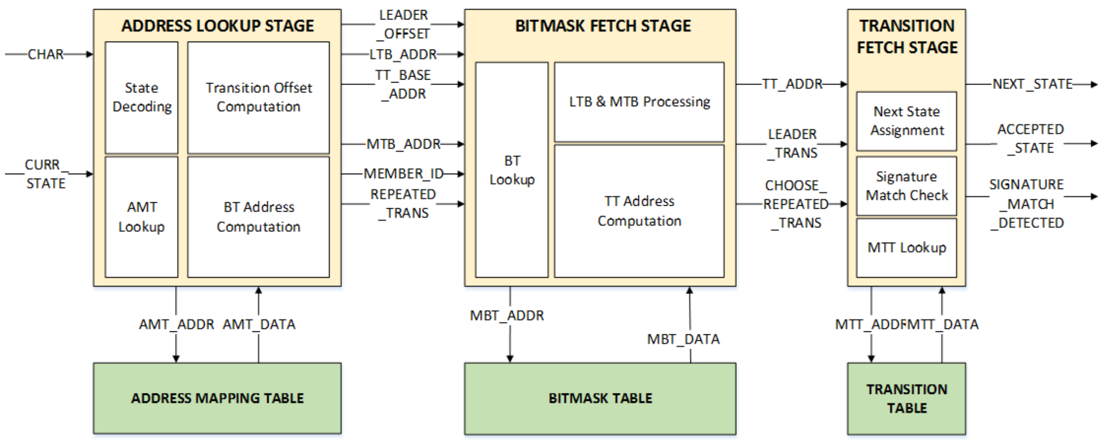


Figure 3.8: Logical block level description of the hardware based decompression engine for the LSCT

The second stage is called as the *Bitmask Fetch Stage (BFS)*. The addresses computed in the previous stage are used to fetch the bitmasks from the bitmask table. Unlike the member bitmask table in the MSBT which was a single port memory, the bitmask table is proposed to be architected using a dual-port memory. This would allow the LTB and the MTB to be simultaneously fetched in a single clock cycle. Once the bitmask data is fetched from the table, the LTB and the MTB are scrutinized to identify the bitmask bits corresponding to the incoming character. Simultaneously, the address location of the compressed state transition to be fetched from the transition table is also calculated in this stage.

The third stage is called as the *Transition Fetch Stage (TFS)*. The compressed state transition is fetched from the transition table based on the computations performed in the previous stages. Depending on whether the incoming state is a leader or a member state and the bitmasks, the next state is assigned either from the transition fetched from the transition table or the registered most repeated transition. Furthermore, it is also checked if the identified next state results in a signature match in this stage.

3.2.4 Summary

In the case of the MSBT, since the bitmap alone is used to compress the redundant transitions in the leader state, some of the failure transitions are not effectively compressed in the leader state. So, the leader transition bitmask was proposed to effectively compress the redundant state transitions in the leader state after performing the MSBT. The idea of compressing the redundant transitions in the leader state in combination with the MSBT is called as the LSCT. Though an additional cost is incurred to store the bitmask for the leader states, multiple redundant state transitions in the leader states can effectively be compressed through this proposal. The next section describes the experimental evaluation of the MSBT and the LSCT.

3.3 Experimental Evaluation

3.3.1 Signature Sets

The MSBT and the LSCT are thoroughly evaluated using a combination of 5 different signature sets. Table 3.1 shows an overview of the signature sets which are used for the

Table 3.1: Signature sets used for evaluation

Signature Set	# Signatures	# DFA States	# State Transitions
Snort34	34	13834	3541504
Snort31	31	19522	4997632
Snort24	24	13882	3553792
Exact_Match	500	15149	3878144
Bro217	217	6533	1672448

evaluation. *Exact_match* is a set of 500 synthetic string signatures generated using the regex tool [86] whose average signature length is about 50 bytes. *Bro217* is a set of 217 regular expressions extracted from the Bro intrusion detection system [87]. The other 3 signature sets, i.e., the *snort24*, *snort31*, *snort34* are a set of 24, 31 and 34 regular expressions extracted from the Snort intrusion detection system [88]. The signatures were first converted into their corresponding DFA representations using the regex tool [86]. Columns 3 and 4 in Table 3.1 show the total number of states and the state transitions in the generated DFA.

After the signature sets were converted into the DFA, the redundant transitions are compressed using the MSBT and the LSCT. The various steps in the MSBT and the LSCT were implemented using unix AWK scripts. The experimental evaluation discussed in this section was performed in an Intel Xeon server CPU running at 4.4 GHz with 500GB of memory. The state grouping algorithm proposed in FEACAN [4] is used to group the DFA as part of the state grouping step in the MSBT and the LSCT. The authors in [4] mention that the transition compression results do not vary when the transition threshold T is varied between 80-95%. So, based on this observation, the transition threshold, T is set to 80% for the state grouping step in this evaluation.

Table 3.2 shows the compressed DFA separated into the leader and the member states after the state grouping step. Column 2 in Table 3.2 shows the total number of groups generated, while columns 3 and 4 in Table 3.2, respectively show the total number of leader states and the member states generated after the state grouping step. Since there is only a single leader state for each group, the total number of leader states generated is identical to the total number of groups. The remaining states in the DFA are organized across various groups as member states. The actual number of member states in each group varies depending on multiple factors such as the bitmap, the transition threshold, T and the characteristics of the signature set.

Table 3.2: Signature characteristics after bitmap compression and state grouping

Signature Set	# groups	# Leader States	# Member States
Snort34	575	575	13259
Snort31	584	584	18938
Snort24	538	538	13344
Exact_Match	298	298	14851
Bro217	173	173	6360

3.3.2 Transition Compression Rate (TCR)

The transition compression rate is the percentage of the number of transitions that are compressed to the total number of transitions in the original DFA. A higher compression rate infers that a higher number of redundant state transitions are compressed in the DFA. Figure 3.9 compares the compression rate achieved by the MSBT and the LSCT in comparison to the theoretical maximum compression limit. Additionally, the compres-

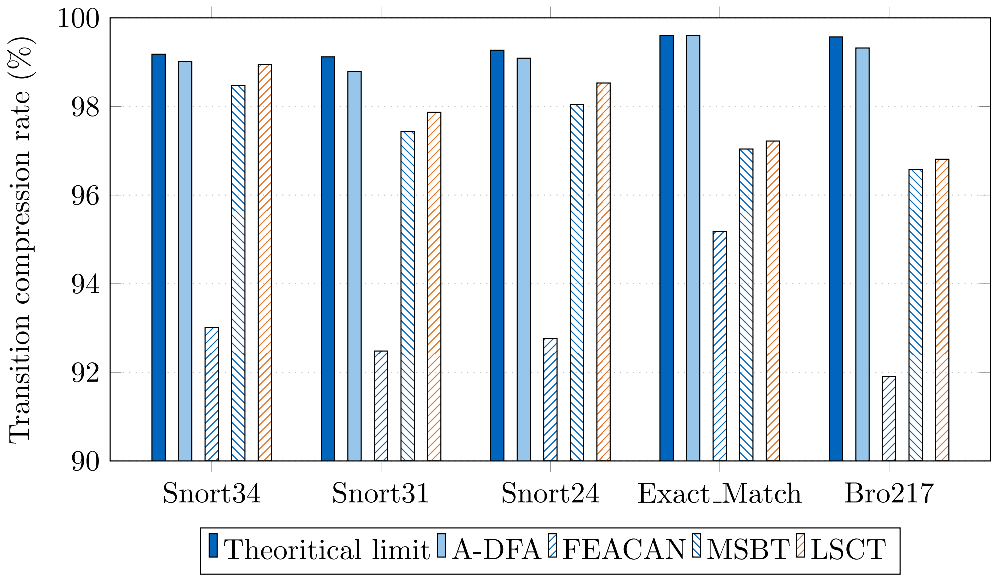


Figure 3.9: Comparison of transition compression rate across various techniques

sion results are also compared against that of the A-DFA and FEACAN. The A-DFA and FEACAN represent the state-of-the-art software and hardware oriented transition compression techniques and are subsequently chosen for comparison. The algorithmic pseudocode available in [4] was used to reproduce the transition compression results for FEACAN [4]. The software implementation of the A-DFA which is available as part of the regex tool [86] is used to generate the compression results for the A-DFA[3].

As seen in Figure 3.9, a 4-5% improvement in the transition compression rate is observed in the MSBT in comparison with FEACAN in most of the signature sets. An additional 0.2-0.5% improvement in the transition compression rate is observed in the LSCT in comparison to the MSBT. In order to better understand the results, Table 3.3 compares the average number of compressed transitions generated per state across various methods. Columns 2 and 3 in Table 3.3, respectively show the average number of member transitions generated per state member state after FEACAN and the MSBT. It is clearly seen from the table that the improvement in the compression rate in the MSBT is a direct consequence of using the MTBs to compress the redundant state transitions in the inter-state compression step. About 50-80% of the member state transitions in FEACAN are redundant which are efficiently compressed by the introduction of the MTB in the MSBT. Similarly, the improvement in the compression rate in the LSCT is attributed to the introduction of the LTB in the LSCT. Columns 4 and 5 in Table 3.3, respectively show the average number of leader transitions per leader state after the MSBT and the LSCT. It can be observed from the table that about 30-60% of the leader transitions are redundant and are effectively compressed in the LSCT.

[3]The parameter k in A-DFA was set to 1 which allows to achieve the best transition compression results in the A-DFA [2].

Table 3.3: Comparison of the average number of transitions in the member state after compression

	#Avg. Member Trans./state		#Avg. Leader Trans/state	
Signature Set	FEACAN	MSBT	MSBT	LSCT
Snort34	17.12	2.16	43.53	13.98
Snort31	18.38	5.18	52.66	14.63
Snort24	17.49	3.14	51.07	19.13
Exact_Match	10.62	5.84	104.96	82.42
Bro217	19.27	6.74	79.66	56.80

The A-DFA achieves the best transition compression results when compared with all the bitmap based compression techniques and is also close to the theoretical limit. However, a closer look at the results from Table 3.4 will detail the cost paid for the high transition compression rate in the case of the A-DFA. Columns 3, 4, 5 and 6 represent the average number of transitions that have to be fetched from the memory before identifying the compressed transition in the A-DFA, FEACAN, MSBT and LSCT, respectively. Column 2 represents the maximum number of transitions that have to be fetched from the memory in the case of the A-DFA before identifying the compressed transition. The numbers shown in columns 3 to 6 are directly calculated from the respective compressed automata representations. The data in column 2 and 3 are calculated by analyzing the compressed DFA generated through the A-DFA compression mechanism. The software implementation of the A-DFA was used to calculate these numbers.

Table 3.4: Average number of transitions fetched before fetching the compressed state transition

Signature Set	A-DFA		FEACAN	MSBT	LSCT
	Max.	Avg.	Avg.	Avg.	Avg.
Snort34	1036	9.8	1.06	1.01	1.01
Snort31	288	12.84	1.07	1.02	1.02
Snort24	1109	21.01	1.07	1.01	1.01
Exact_Match	26	6.65	1.04	1.02	1.03
Bro217	44	10.05	1.07	1.03	1.03

Columns 2 and 3 clearly show that multiple tens of transitions have to be fetched from the memory before identifying the compressed transition corresponding to the state character combination in the case of the A-DFA. Simulations performed in [4] also show similar results to what has been shown in Table 3.4 in the case of the A-DFA. The additional memory bandwidth that is required to search the compressed state transition is the cost paid by the software oriented techniques such as the A-DFA. On the other hand, in the case of the bitmap based techniques, only a maximum of 2 memory accesses are required to fetch the compressed transition. The best case scenario to identify the compressed transition is a single memory fetch in all the cases. However, the ability to

Table 3.5: Parameters used for memory usage estimation

Parameter	Bit-Width (in bits)		
	FEACAN	MSBT	LSCT
LTT Base Address	18	16	-
MTT Base Address	18	16	-
TT Base Address	-	-	18
MBT Base Address	-	16	-
BT Base Address	-	-	16
MTB & LTB	-	Variable	Variable
Bitmap	256	256	256
Transition	20	20	20

fetch the compressed transition without introducing additional memory bandwidth is the differentiating factor in the case of bitmap based transition compression techniques.

3.3.3 Estimated Memory Usage

After evaluating the achieved transition compression rates, the amount of on-chip SRAM space required to store the compressed DFA is estimated in this section. Table 3.5 lists the assumptions pertaining to the width of the various information that is stored in the memories. The details associated with the compressed DFA storage is not discussed in detail in FEACAN [4]. So, the memory space required to store the compressed DFA generated by FEACAN is also estimated together with the MSBT and the LSCT.

Since the ASCII character set is used for DPI, the width of the bitmap in all the methods is set to 256 bits. The compressed state transition is represented through 20 bits which allows to uniquely represent a DFA consisting of a maximum of 2^{20} states, which is much higher than the number of states in the DFA's used for evaluation. The base address widths for the various tables is chosen in such a way that the compressed information is addressable using the assumed values. Since the width of the bitmask varies depending on the number of transitions compressed in the intra-state compression step, it is rounded-off to the nearest byte which varies depending on different signature sets[4].

Figure 3.10 shows a comparison of the estimated memory footprint of the compressed DFA generated across various methods. The estimated memory footprint is differentiated into the transition and the control memory. The memories which store the compressed state transitions are referred to as the transition memories while the others are referred to as the control memories. It can be seen from Figure 3.10 that a small improvement of 4-5% in the transition compression rate in the case of the MSBT results in a 50% reduction in the overall memory footprint of the compressed DFA in comparison to

[4]The theoretical maximum width of the bitmask can be 256 bits and this corresponds to a scenario where none of the state transitions in a state are compressed as part of the inter-state compression step.

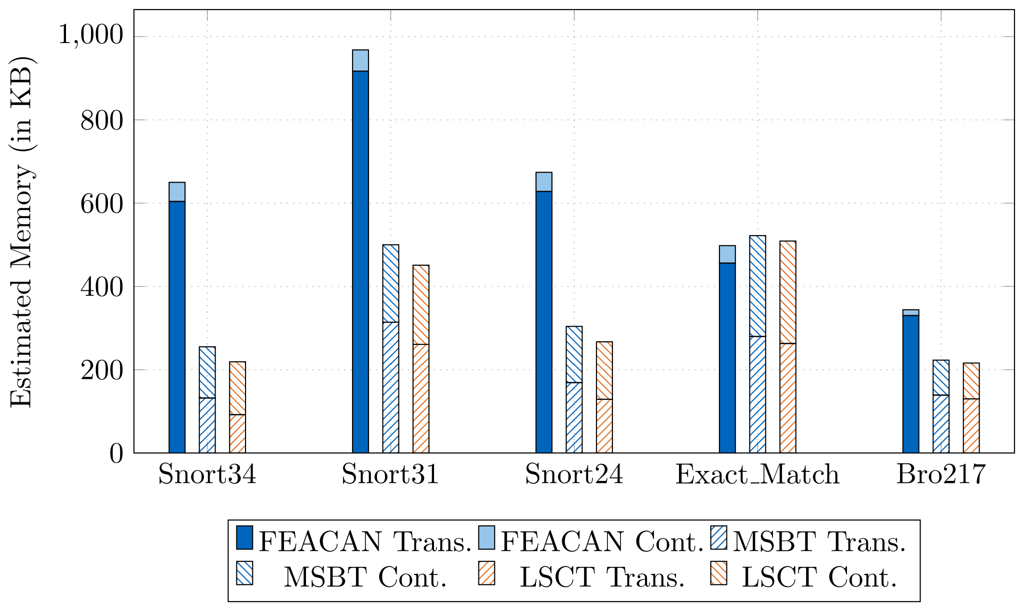


Figure 3.10: Comparison of transition and control memory usage across various techniques

FEACAN. Similarly, in the case of the LSCT, even a very minute (∼0.5%) increase in the transition compression rate results in a significant 5 – 10% reduction in the memory footprint when compared with the MSBT.

It should also be noted from Figure 3.10 that there is a significant difference between the transition and control memory usage between FEACAN and the proposed techniques. FEACAN only stores the base addresses and the bitmap for each of the groups as part of the control data. So, the control memory portion only represents a very negligible portion of the overall memory usage, while a major portion of the memory is used to store the compressed state transitions as shown in Figure 3.10. On the other hand, in the case of the MSBT and the LSCT, a considerable portion of the overall memory is used by the control memories to store the bitmasks, which in turn efficiently compresses the redundant state transitions.

As seen in Figure 3.10, the Exact_Match signature set is an exception where the estimated memory required to store the compressed DFA in the case of the MSBT and the LSCT is slightly higher than FEACAN. The average length of the bitmask in the case of the Exact_match signature set was about 112 bits, which when stored for each member state required a considerable amount of on-chip memory. Nevertheless, the overall transition compression achieved in the case of the Exact_Match signature set in the case of the MSBT and the LSCT is higher than FEACAN.

The memory space required to store the compressed transitions in the case of software based techniques such as the A-DFA, depends on the chosen memory layout [89] and so is not compared with the proposed techniques.

3.3.4 Functional Evaluation - Software Model

The functionality of the MSBT and the LSCT based signature matching engines was verified through a software based model. Verifying the functional correctness of the proposed techniques is paramount as the DPI applications cannot afford a true negative during the signature matching step. Figure 3.11 shows an overview of the software based simulation setup which is split across four steps.

In the first step, the signature set is first converted into the DFA using the regex tool [86]. This tool first converts the signatures into the NFA and then to its corresponding DFA representation.

The second step consists of two different functions. Firstly, the DFA is compressed through the MSBT and the LSCT and the corresponding compressed DFA is generated. A compiler was developed using the UNIX AWK script to perform this process. The resulting compressed DFA was converted into the table formats as described in sections 3.1.2 and 3.2.2 through the compiler. Secondly, a synthetic payload byte sequence is generated by the traffic generator which is injected into the signature matching engines to verify their functionality. The traffic generation methodology proposed by Becchi et al. [90] is used to generate the byte sequence for each of the signature sets. The model proposed in [90] generates the payload byte sequence based on the probability of maliciousness, P_M. In this process, the traffic generator takes the NFA or the DFA as the input and generates a sequence of characters, where each character in the sequence is decided based on the defined P_M value by simultaneously traversing the automata[5]. The P_M value is generally chosen between 0 and 1, where a lower P_M value indicates a lower probability of the generated trace consisting of signatures in it. The traces were

[5]A detailed description of the traffic generation function is described in Appendix A

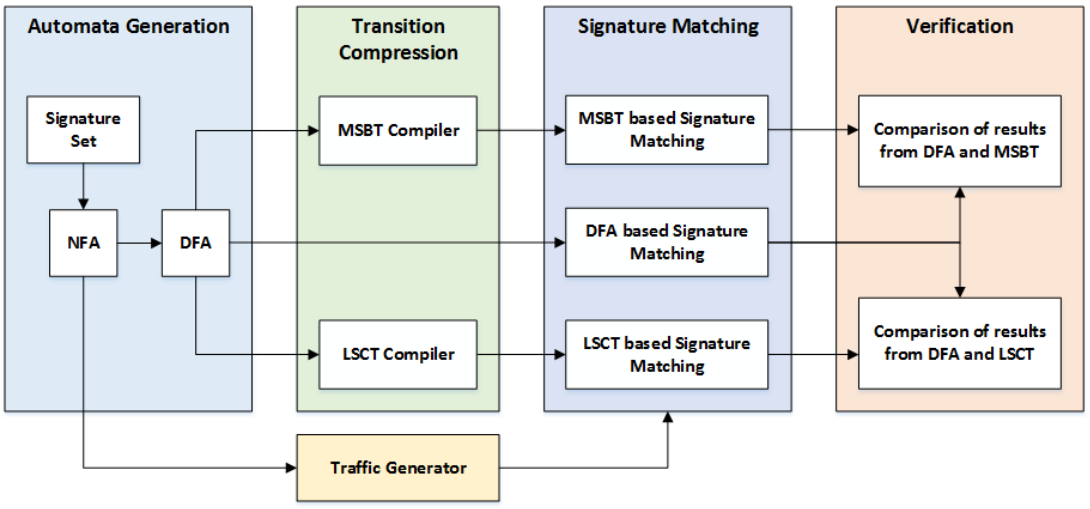

Automata Generation
Transition Compression
Signature Matching
Verification
Signature Set
NFA
DFA
MSBT Compiler
LSCT Compiler
MSBT based Signature Matching
DFA based Signature Matching
LSCT based Signature Matching
Comparison of results from DFA and MSBT
Comparison of results from DFA and LSCT
Traffic Generator

Figure 3.11: Overview of the software based simulation environment to verify the transition compression

individually generated for each of the signature sets corresponding to 4 different P_M values, i.e., 0.35, 0.55, 0.75 and 0.95. The traffic trace generated with the P_M value of 0.35 represents the average case, while a traffic trace generated with the P_M value of 0.95 represents the worst case scenario [90]. A total of 1MB payload byte sequence was generated corresponding to each of the P_M values. The process of verifying the signature matching engines through the synthetic traces allows to verify the signature matching engines across a multitude of scenarios ranging between the average case to the worst case scenario.

In the next step, the generated traces are injected into the MSBT, LSCT and the DFA based signature matching engines. The core part of the signature matching in both the MSBT and the LSCT is the decompression function. The signature matching starts with the root state being assigned as the current state. The compressed state transition corresponding to a current state character combination is identified during the decompression process and is assigned as the next state. For the subsequent character, the identified next state is assigned as the current state and the process continues further. The MSBT and the LSCT based signature matching engines generate an alert if a signature match is identified in the trace. The generated traces are injected into a DFA based signature matching engine implemented using the software available in [86]. In this way, the signature matching results from the DFA based signature matching engines are used as the reference to compare the results with the MSBT and the LSCT based signature matching engines.

In the last step, the total number of signature matches generated in the MSBT and the LSCT based signature matching engines are compared to that of the DFA based signature matching engine. The step by step comparison across the traces were performed for all the signature sets across all the P_M values. Table 3.6 shows the signature matching results across the different signature sets across the different P_M values. It can be seen from the table that the total number of signatures matched in the MSBT and the LSCT based signature matching engines are identical to the signature matching results seen in the case of the DFA. In addition to the total number of signature matches, the location of the signature match occurrence in the byte stream was also identical between the proposed methods and the DFA. The identical signature matching results seen in the simulation results further show that the MSBT and the LSCT only compresses the redundant state transitions in the DFA and the functional equality is maintained during the transition compression process.

3.4 Conclusion

The major advantage of using the bitmap to compress the DFA is the possibility of performing the decompression in a dedicated hardware accelerator, in turn enabling to perform line rate signature matching. However, in the case of bitmap based compression approaches, the bitmaps corresponding to the states have to be stored together with the compressed state transitions. In order to minimize the storage cost associated with the bitmaps, the existing bitmap based compression algorithms store more redundant state

Table 3.6: Comparison of signature matching results across the compression methods across different P_M values

P_M	Method	Signature Sets				
		Snort34	Snort31	Snort24	Exact_Match	Bro217
0.35	DFA	5	1	29	46	11631
	MSBT	5	1	29	46	11631
	LSCT	5	1	29	46	11631
0.55	DFA	7	2	45	117	6347
	MSBT	7	2	45	117	6347
	LSCT	7	2	45	117	6347
0.75	DFA	8	1	22	331	28701
	MSBT	8	1	22	331	28701
	LSCT	8	1	22	331	28701
0.95	DFA	14591	1286	22818	7841	69976
	MSBT	14591	1286	22818	7841	69976
	LSCT	14591	1286	22818	7841	69976

transitions in the transition memories, resulting in inefficient transition compression of the DFA.

Addressing this problem, two different bitmap based transition compression methods called the MSBT and the LSCT were proposed and evaluated in this chapter. The MSBT and the LSCT compress the redundant state transitions efficiently by introducing a secondary layer of indexing called bitmasks after bitmap based transition compression. Though, the bitmasks have to be stored along with the compressed DFA, the resulting improvement in the transition compression rates effectively reduces the memory footprint of the compressed DFA. Since the proposed techniques are bitmap based, the decompression can be performed in a dedicated hardware accelerator to perform signature matching at multi gigabit line rates. The proposed methods were experimentally evaluated to show the improvements resulting in the transition compression rates through the introduction of bitmasks. The MSBT and the LSCT consistently achieved transition compression rates of 97-98%. This corresponds to a 4-5% improvement in the transition compression rates in comparison to the existing bitmap based techniques. The simulations performed as part of the memory usage estimation further showed that the memory required to store the compressed DFA after the MSBT and the LSCT is $\sim$50% lesser than the existing bitmap based approaches.

Additionally, as part of the experimental evaluation, a software implementation of the signature matching engine was designed to verify the functional correctness of the proposed approaches. The signature matching was performed across various signature sets against multiple synthetic traffic traces using the MSBT, the LSCT and the DFA to represent the signatures. The signature matching results obtained after MSBT and LSCT were identical to that of the DFA, which further validated the functional correctness of the approach. The next chapter details additional optimizations which are performed on top of the MSBT and the LSCT to further reduce the memory footprint of the compressed DFA.

4 Memory Footprint Optimizations - MSBT & LSCT

4.1 Motivation

Since the compressed DFA is proposed to be stored in the on-chip memories after the MSBT and the LSCT, it is paramount to reduce the memory footprint of the compressed DFA through additional optimizations. Moreover, when a signature matching engine is designed to perform the transition decompression, the physical boundaries of the memories which are used to store the compressed DFA will definitely be predefined. So, it is essential to make the memory footprint of the compressed DFA as small as possible, as this will allow more signatures to be stored in the on-chip memories.

As discussed in Chapter 3, the memories which store the compressed DFA can broadly be classified into the transition and the control memories. The transition memories store the compressed state transitions, while the control memories store the control data such as the bitmap and the bitmasks. In order to reduce the transition memory usage in the compressed DFA, it is essential to make sure that the DFA is compressed as efficiently as possible. Similarly, it is also important to make sure that only the essential control data is stored in such a way that the transition decompression is performed in a lossless manner[1]. So, three different optimizations are proposed for the MSBT and the LSCT in this chapter which focus on reducing the overall memory footprint of the compressed DFA. A short introduction to the optimizations is provided below.

- **Divide & Conquer State Grouping:** The Divide & Conquer state grouping method proposes a structured methodology for the state grouping method in the MSBT and the LSCT. The proposed state grouping method is compression-aware, i.e., the state grouping decision is optimized to generate the best transition compression results. Organizing the states into state clusters with the knowledge of the resulting number of compressed transitions greatly reduces the number of state transitions generated after the inter-state compression step in the proposed transition compression methods.
- **Alphabet Compression:** Although there are 256 characters in the ASCII character set, yet not all the characters are used to compose the signatures. Due to this reason, when a signature set is converted into the DFA, there are certain

[1]The compressed DFA should always be equivalent to the original DFA in terms of the signature representation. The compression algorithm should not compress the state transitions or the associated control data in such a way that it results in incorrect signature matching results.

indistinguishable characters in the DFA, i.e., those characters whose state transitions are identical across all the states in the DFA. Alphabet compression is a well known technique which is generally used to compress the redundant state transitions corresponding to these indistinguishable characters and is also orthogonal to transition compression. So, performing the MSBT and the LSCT after performing the alphabet compression will further improve the transition compression rates.

- **Bitmask Compression:** Unlike the two methods mentioned above which focus on improving the transition compression rate, the *bitmask compression* focuses on compressing the redundant bitmasks generated during transition compression. Due to the predictable organization of the state transitions in the member states, the bitmasks corresponding to certain member states are redundant and can be compressed. The compression of these bitmasks helps to reduce the control data generated during transition compression. However, the cost paid for the bitmask compression is the additional processing that is incurred to identify if the bitmask corresponding to a member state is compressed or not.

All of the above mentioned methods are explained with the MSBT. Since the LSCT is a logical extension over the MSBT, each of the proposed techniques is applicable to LSCT also. The proposed optimizations are explained in detail in the following sections.

4.2 Improved Transition Compression through State Grouping

4.2.1 Background

The state grouping step plays a significant role in efficiently organizing the states for the inter-state compression step in the MSBT and the LSCT. The state grouping algorithm proposed in FEACAN [4], which was originally used in the MSBT and the LSCT, does not focus on clustering the states which would result in the best compression results. Figure 4.1 shows a scenario where the state grouping step proposed in FEACAN when used together with the MSBT results in sub-optimal transition compression. Figure 4.1(a) shows a DFA with 8 states and 8 characters with a total of 64 state transitions. After the intra-state compression step, the states are clustered into 3 groups as shown in Figure 4.1(c). The transition threshold T, is set to 75%[2] in this particular example. Figure 4.1(d) shows the final compressed DFA after the inter-state compression step.

As discussed in Chapter 3, all the transitions in the leader state generated after the intra-state compression step are consolidated and stored in the memory after the MSBT. So, more groups generated after the state grouping step will result in more leader transitions stored in the memory[3]. Moreover, a state should be clustered into a group with

[2]The Transition threshold, T of 75% is just set to explain the problem associated with FEACAN state grouping.

[3]FEACAN does not discuss the implications of varying T to the number of groups generated in the state grouping step. However, the experimental evaluation performed in this chapter identified that the higher the value of set to T, a higher number of groups are generated during the state grouping step.

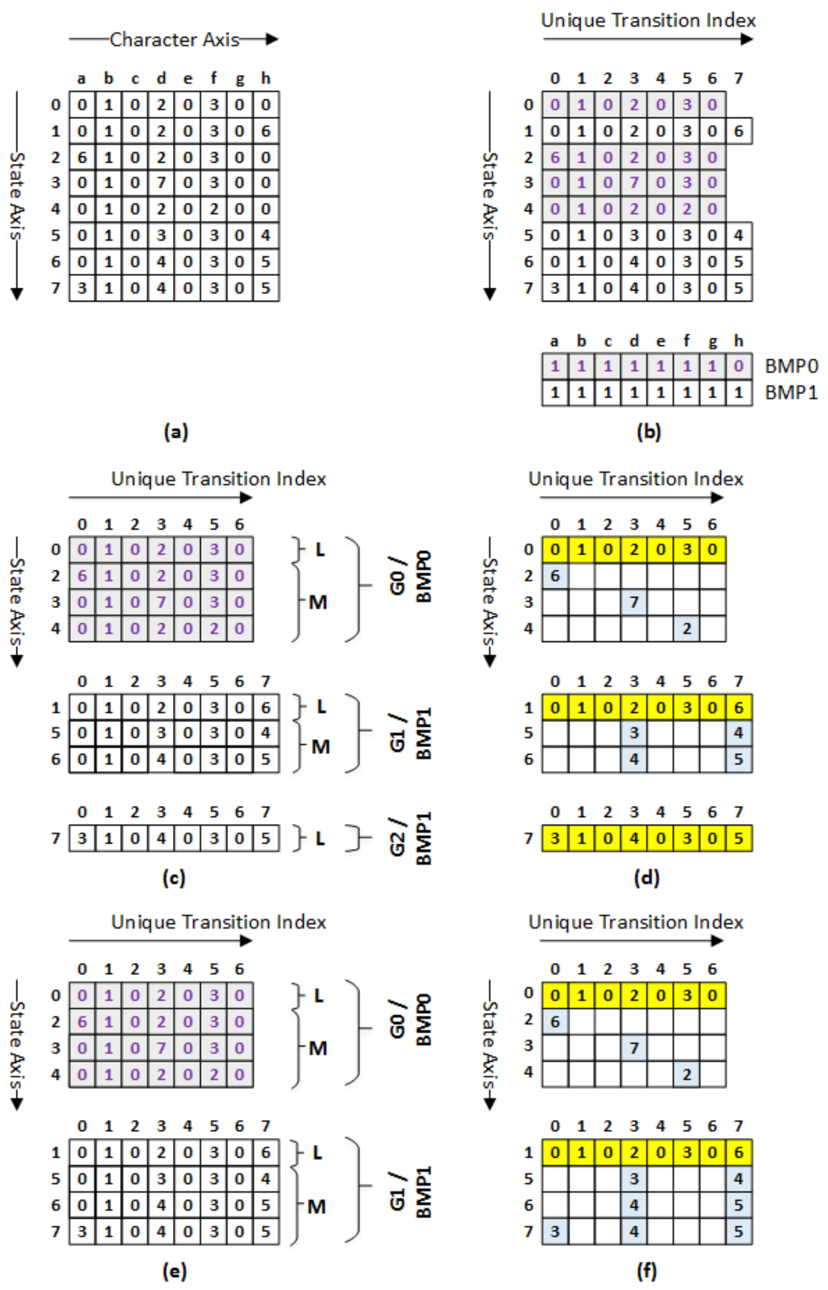


Figure 4.1: (a) An uncompressed DFA with 8 states and 8 charcaters (b) DFA after Intra-state transition compression step in MSBT (c) The DFA split into 3 groups after the state clustering step (d) DFA after Inter-state transition compression step in MSBT (e) State 7 merged with G1 as it shares the same bitmap defying the constraint with transition threshold (f) More transitions compressed when Inter-state transition compression is performed after merging G1 and G2

whose leader state it shares the maximum number of identical transitions, as this will result in the least number of member transitions from the state. So, clustering a state into the right group and simultaneously balancing the overall number of groups is very important to achieve high transition compression rates. Though, the FEACAN grouping algorithm clusters the states into groups, it is neither compression-aware, nor it optimizes the state clusters to achieve the best transition compression results. For example, as shown in Figure 4.1(d), performing the inter-state compression on the groups generated after FEACAN results in a total of 30 compressed transitions. However, if state 7 shown in G2 in Figure 4.1(c) is merged with G1 as shown in Figure 4.1(e), the total number of compressed transitions generated reduces to 25 as shown in Figure 4.1(f). This state merging is possible as the bitmap of state 7 is identical to the bitmaps of the states in G1. FEACAN grouping algorithm creates a separate group with state 7 as the leader state since it does not satisfy the clustering constraint associated with the transition threshold, as state 7 only has 5 (62.5% < T=75%) state transitions identical to the leader state in G1. Although merging state 7 into G1 violates the clustering constraint associated with the transition threshold, it results in fewer compressed state transitions in comparison to state 7 being added into a separate group.

4.2.2 Divide & Conquer State Grouping

Addressing the shortcomings associated with the FEACAN grouping algorithm, a *compression-aware* Divide and Conquer state grouping method is proposed in this section. The proposed method consists of two major steps, a divide step and a conquer step as described below:

- **Divide Step:** The divide step compares a state with all the available groups before clustering the state into a group, with which the state has the maximum number of identical transitions. In this way, the number of member transitions generated from the state is minimized.

- **Conquer Step:** The conquer step combines multiple groups of states into a single group, if and only if, the combined group results in fewer compressed transitions than the compressed transitions generated from the groups before combining. Moreover, an optimal leader state is identified for each of the groups which results in the least number of member transitions per group after performing the inter-state compression step.

The overall motive of the proposed method is to make sure that the states are grouped optimally to achieve efficient transition compression results. Although an additional processing cost is incurred to make the state step grouping compression-aware, yet this results in an improved transition compression. The DC state grouping method consists of four different algorithms and are described in detail below.

4.2.2.1 The Divide Step

The divide step performs a preliminary grouping of the states using a set of clustering constraints. A state is clustered into a group if and only if the following clustering constraints are met.

(i) The bitmap of the state to be clustered is the same as the bitmap of the leader state in the group. Having an identical bitmap across all the states in the group enables to compress the redundant state transitions between the states.

(ii) The number of states in the group is lesser than the predefined maximum number of states, B. The typical values of B ranges anywhere between 128 and 512. The smaller the value set to B, the higher the number of groups generated in comparison to a higher value set to B. However, the transition compression rates do not vary greatly depending on the chosen value of B.

(iii) A state has at least T identical transitions with the leader state of a group, where T is called the transition threshold. A function called similarity (sim), is used to calculate the number of identical transitions (it) between any two states. The transition threshold is identical to the transition threshold which is used in FEACAN.

The clustering constraints used for the divide step are similar to the ones used in the FEACAN grouping algorithm. After running through the divide procedure, a DFA with S states is split into G groups with each group ‘g’ represented as a 4-tuple (L, M, BMP, ST), where

- L ∈ S, is the leader state of the group
- M ∈ S, is the set of states in the group apart from the leader state, called the member states
- BMP, the bitmap of the leader state in the group
- ST, the number of states in the group

During the divide step, each of the states in the DFA is compared against a set of groups during which the clustering constraints are checked. If all the clustering constraints are met, a state is clustered to the group with which it has the maximum number of identical transitions among all the groups at that time instance. Even if one of the clustering constraint is not met, a new group is created with the state being added as the leader state. Since the divide step is an incremental process, the number of groups with which a state is compared during the divide procedure varies over time. For example, at time instance t, a state $s \in S$ is only compared against j groups, which have formed over time t. On the other hand, at a later time instance $t+\Delta$, a state $r \in S$ is compared with $j+\delta$ groups, where δ is the number of groups formed within the Δ time.

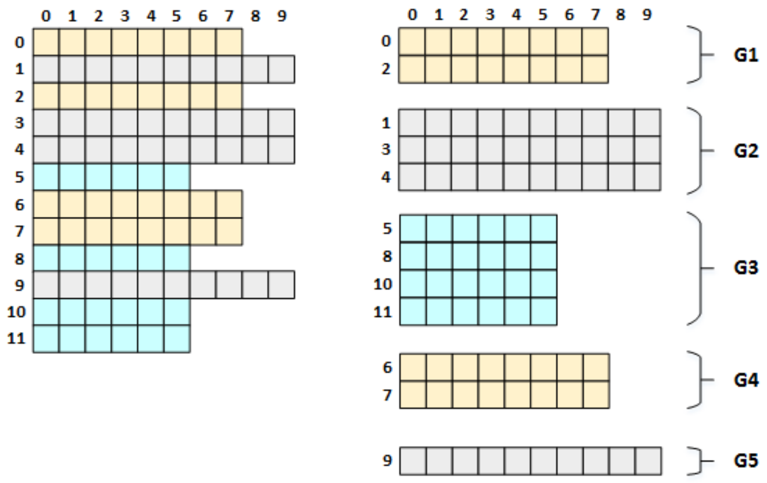


Figure 4.2: Example of DFA states clustered into a set of 5 groups using the divide step

Figure 4.2 shows an example of the DFA being clustered into groups using the divide step. A total of 12 states are clustered into 5 groups represented through G1 to G5. Three different colors are used to represent the bitmap of the states after the intra-state compression. The state grouping starts with state 0 and ends with state 11. State '0' is added as the leader state of G1 as there are no other groups to compare the state. After grouping state '0', state '1' is added as the leader state of G2 as the bitmap of the states '0' and '1' do not match. This increases the overall number of groups to 2. In the case of state '2', it is compared with groups G1 and G2 before eventually deciding on the state grouping for state '2'. In this way, as part of the divide step, the grouping decision is made by comparing the state with the available groups, one at a time. By making sure that the state is added into the group with which it has the maximum number identical transitions, the number of member transitions generated after inter-state compression is kept to a minimal.

Algorithm 1 shows the pseudocode for the divide procedure. The processing complexity associated with the algorithm is $\mathcal{O}(S \log S)$.

4.2.2.2 State Reorganization

As part of the divide step, a state is clustered into one of the groups with which it has the highest number of identical transitions during the time of comparison. However, there are certain groups with which the state is not compared, i.e, the groups which were created after the state was clustered, with which it may have a higher number of identical transitions than the one to which it is clustered into. So, as part of the state reorganization procedure, each member state in a group is compared only with those groups which were created after the member state was clustered into its group.

Algorithm 1 Divide Automata States into Groups

```
1:  for all $s \in S$ do
2:      $s.grp \leftarrow \emptyset$                                    ▷ Initialization
3:      for all $g \in j \mid j \in G$ do
                                   ▷ Clustering Constraint Check (i) & (ii)
4:          if $(g.ST < B)$ & $(s.BMP = g.BMP)$ then
5:              $it \leftarrow sim(s, g.L)$
                                       ▷ Clustering Constraint Check (iii)
6:              if $it > T$ then
7:                  if $(s.grp = \emptyset) \parallel (s.it < it)$ then
8:                      $s.it \leftarrow it$
9:                      $s.grp \leftarrow g$
10:     if $s.grp = \emptyset$ then                  ▷ New group with s as leader
11:         $j \leftarrow j + 1$
12:         $j.L \leftarrow s$
13:     else                                 ▷ s added as member state in s.grp
14:         $s.grp(M) \leftarrow s.grp(M) \cup s$
```

The state $s \in S$, which is already clustered into one of the groups in $j \in G$, is compared with the set of groups $h \in G \mid h \cap j \rightarrow \emptyset$, where h is the set of groups created after s is clustered. After the state reorganization step, the state s is added to the group with which it has the maximum number of identical transitions, either in j or h. Since, a state is added into the group with which it has the highest number of identical transitions, this results in the least number of member transitions from the state during the inter-state compression step. Algorithm 2 shows the pseudocode for the state reorganization procedure and the processing complexity associated with the algorithm is $\mathcal{O}(S \log S)$.

Figure 4.3 shows an example of the state reorganization step. As part of the state reorganization step, state '2', which was originally clustered into G1 is now moved into G4, after being compared with G3, G4 and G5. These were the groups which were created after state '2' was clustered into G1 after comparison with G1 and G2. As shown in Figure 4.3, state '2' has 2 non-identical state transitions at indices '3' and '4' in G1, while it only has '1' non-identical state transition in G4. By reorganizing state '2' from G1 to G4, an additional state transition in state '2' is compressed.

4.2.2.3 The Conquer Step

More groups resulting from the divide and the state reorganization steps will result in more leader transitions stored in the memory. So, it is necessary to combine the groups whenever possible to reduce the number of leader transitions. Any two groups g, $h \in G$ can be combined into a single group g, if the resulting number of compressed transitions after combining them is fewer than the sum of the compressed transitions when they remain as separate groups. If the above mentioned property is satisfied for the groups g and h, all the states in the group h are added as member states in the group g. In this way,

Algorithm 2 State Reorganization

```
1: for all $g \in G$ do
2:     for all $s \in g(M)$ do
                                          ▷ h created after s added to g(M)
3:         for all $h \in G$ do
                                          ▷ Clustering Constraint Check (i) & (ii)
4:             if $(h.ST < B)$ & $(s.BMP = h.BMP)$ then
5:                 $it \leftarrow sim(s, h.L)$
6:                 if s.it < it then
                                          ▷ Clustering Constraint Check (iii)
7:                     $s.it \leftarrow it$
8:                     $s.grp \leftarrow h$
                                          ▷ State Reorganization Decision for state s
9:         if $s.grp \neq g$ then
10:            $s.grp(M) \leftarrow s.grp(M) \cup s$
```

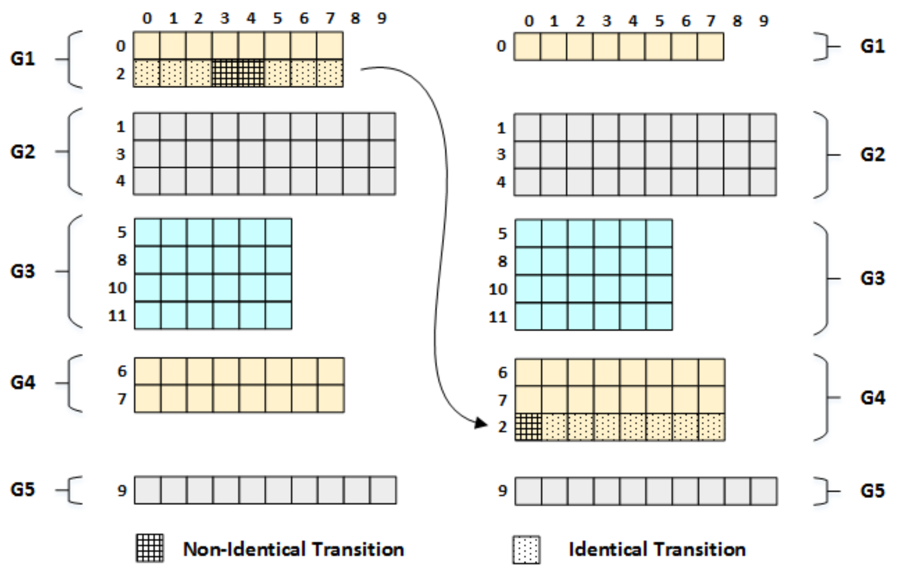


Figure 4.3: Example of state reorganization step performed on the states after the divide step

the groups which only improve the transition compression rate are actually combined together to reduce the number of leader transitions generated. The above mentioned property is defined as the *conquer criteria* along with the clustering constraints (i) and (ii) which were defined as part of the divide procedure. A group is iterated through all the other groups to find the best combination that will result in the least number of compressed transitions when combined. The conquer step continues until it reaches a point where it cannot continue further. Algorithm 3 shows the pseudocode of the conquer step. The processing complexity associated with algorithm 3 is $\mathcal{O}(G^2)$.

Algorithm 3 Conquer Step

```
1: $conquer\_done \leftarrow 0$
2: $grps \leftarrow G$
3: for all $g \in G$ do
4:     $g.trans \leftarrow \#CTrans(g)$
5:     $g.mergetrans \leftarrow 0$
6:     $g.combine \leftarrow g$
7: while $conquer\_done = 0$ do
8:     $grps\_before\_combine \leftarrow grps$
9:     for all $g \in G$ do
10:        for all $h \in G \mid h \neq g$ do
11:            if $((g.BMP = h.BMP)\ \&\ (g.ST + h.ST) < B))$ then
                                                ▷ Conquer Condition Check
12:                if $(g.trans + h.trans) > [g \cup h].trans)$ then
13:                    if $((g.mergetrans > [g \cup h].trans)\ \|\ (g.mergetrans = 0))$ then
14:                        $g.mergetrans \leftarrow [g \cup h].trans$
15:                        $g.combine \leftarrow h$
16:        if $g.combine \neq g$ then
17:            $g(M) \leftarrow g(M) \cup h.L \cup h(M)$          ▷ Merge Groups
18:            $grps \leftarrow grps - 1$
19:            $g.trans \leftarrow g.mergetrans$
20:    if $(grps\_before\_combine = grps)$ then
21:        $conquer\_done \leftarrow 1$
```

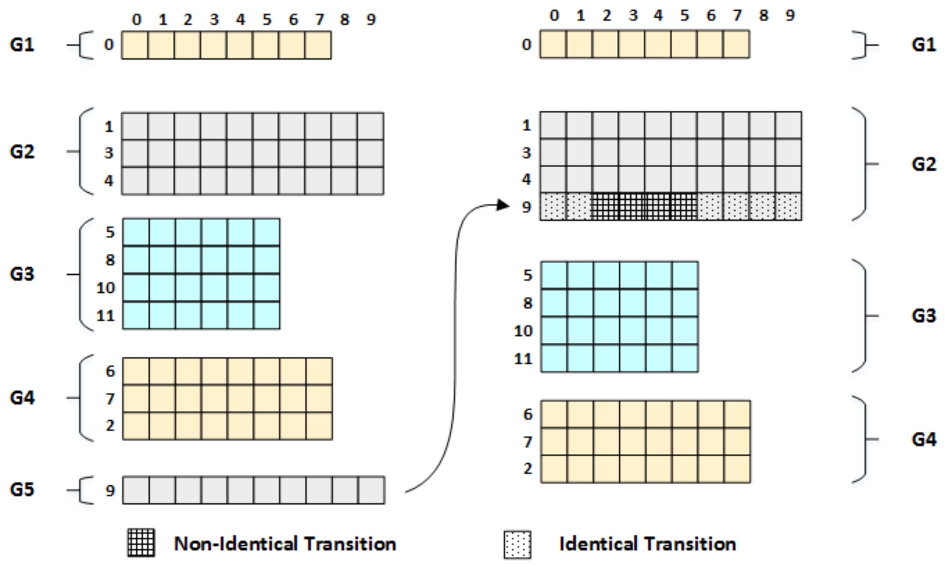


Figure 4.4: State 9 merged with G2 as part of conquer step

Figure 4.4 shows an example of the conquer step performed as part of the DC state grouping method. After the divide and the state reorganization steps, state '9' is originally clustered into G5 as the clustering constraint pertaining to the transition threshold did not satisfy in the divide step. Since state '9' is the only state in G5, it is also the leader state in the group which necessitates all the 10 state transitions to be stored in the memory after the divide step. However, after the conquer step, the state is merged with G2 with which it shares 6 identical transitions. Thus, by combining G5 and G2, a total of 6 transitions are additionally compressed even though the clustering constraint pertaining to the transition threshold is violated. Since, the priority of the DC state grouping is to reduce the number of compressed transitions, the violations can be waived. It should be noted that two groups are merged, if and only if, the clustering constraints pertaining to the bitmap and the number of states are strictly maintained after merging the groups.

4.2.2.4 Optimal Leader State Identification

When the divide step is originally performed, the first state added to the group is designated as the leader state by default and is never changed over the grouping process. However, there could be a state among the member states, which when made as the leader state is capable of producing fewer member transitions in the group than the default leader state. Identifying this optimal leader state for the group improves the overall transition compression. So, each of the member states in a group is temporarily made as the leader, to calculate the resulting number of member transitions in the group. The state which results in the least number of member transitions will be the optimal leader and is finally chosen as the leader state of the group. If none of the states generate the least number of member transitions in comparison to the default leader state, the default leader state is maintained as the leader state in the group. Algorithm 4 shows the pseudocode to modify the leader state. Assuming that there are k states per group on average, the worst case time complexity to identify the best leader state in a group is $\mathcal{O}(k)$.

Algorithm 4 Identify Best Leader State

```
1: $min\_trans \leftarrow \#member\_trans\ with\ g.L$
2: $Add\ g.L\ to\ g(M)$
3: for all $t \in g(M)$ do
4:     $g.L \leftarrow t$                                  ▷ Make t as the Leader State
5:     $t.trans \leftarrow \#member\_trans\ with\ g.L$
6:     if $min\_trans \geq t.trans$ then
7:         $leader \leftarrow t$
8:         $min\_trans \leftarrow t.trans$
9: $g.L \leftarrow leader$                                ▷ Optimal Leader State
```

Figure 4.5 shows the optimal leader identification step performed on G3. The figure in the left shows G3 with state '5' set as the leader state which generates a total of 6

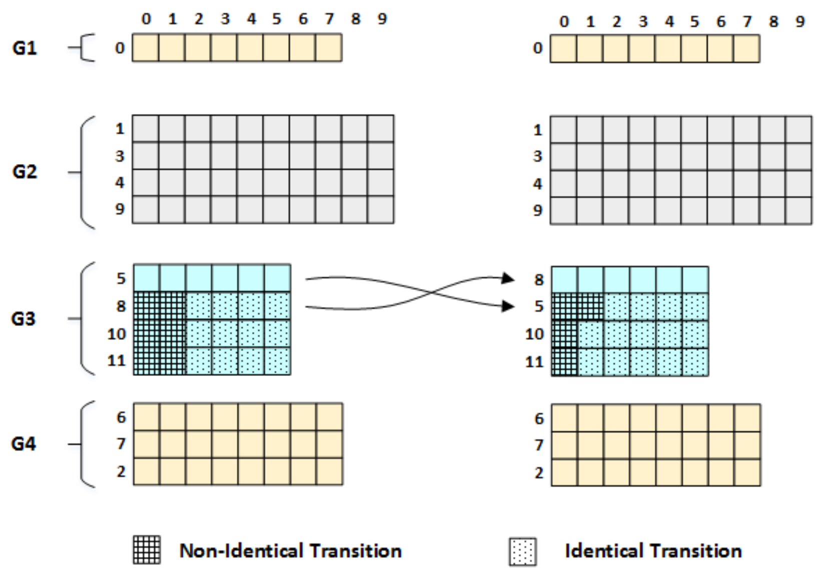


Figure 4.5: States 5 and 8 swapped in G3 as part of optimal leader state identification

member transitions within the group. On the other hand, after the optimal leader state identification, state '8' is chosen as the leader state which results in 4 member transitions in comparison to 6 with the default case. Thus, this is a step in which the states are reorganized internally within a group to improve the transition compression.

4.2.3 Complexity Analysis Comparison

The complexity associated with the various algorithms proposed as part of the DC is summarized in Table 4.1. The DC method increases the complexity of the grouping process in comparison to the FEACAN which achieves the state grouping in $\mathcal{O}(S \log S)$. However, the increased time complexity is the cost paid to achieve to improve the transition compression rate in the case of the DC state grouping method.

The RCDFA [5] also uses a state reorganization algorithm to reorganize the states before performing bitmap based transition compression. As part of this algorithm, the states are reorganized within the DFA, so that the states which have identical transitions are placed next to each other. The authors claim the complexity of the algorithm to be $\mathcal{O}(S \log S)$. However, when a state is reorganized within the DFA, the state identifier corresponding to the reorganized states will also have to be swapped and the complete DFA has to be updated to reflect the changes in the state modifier. The complexity associated with this step is not accounted by the authors which increases the complexity of the algorithm to $\mathcal{O}(S^3 \log S)$. This is very high in comparison to the FEACAN and the DC state grouping algorithms.

The time complexity associated with the A-DFA [2], the state of the art software oriented algorithm is $\mathcal{O}(S^2)$. In comparison to [2], the DC method splits the states into groups in logarithmic time in the divide step and further improves the grouping in the

Table 4.1: Algorithmic complexity of various algorithms proposed in DC

Algorithm	Complexity
Divide Step	$\mathcal{O}(S \log S)$
State reorganization Step	$\mathcal{O}(S \log S)$
Conquer Step	$\mathcal{O}(G^2)$
Optimal Leader State Identification	$\mathcal{O}(k)$

conquer step in quadratic time. Since, the number of groups is very small in comparison to the total number of states after the divide step, the quadratic complexity associated with the conquer step doesn't affect the performance of the overall grouping process. Moreover, as mentioned in [2], a huge amount of memory is required by the A-DFA algorithm to store the intermediate data structures which makes it impractical to build a single compressed automaton.

4.2.4 Experimental Evaluation

The signature sets which were used to evaluate the MSBT and the LSCT in Chapter 3 are also used to evaluate the DC state grouping method. In order to evaluate the effect of the state grouping on the transition compression, the states are grouped into subsets using the FEACAN and the DC state grouping methods separately. After this step, the respective inter-state compression steps proposed in the MSBT and the LSCT are performed on the generated subsets of states. The transition compression rate achieved through the MSBT and the LSCT under both the grouping methods are calculated and compared to show the improvement which the DC state grouping method brings. After evaluating the transition compression rates, the estimated memory footprint of the compressed DFA is calculated similar to the evaluations performed in Chapter 3. This helps to understand the reduction in the memory footprint in the compressed DFA after using the DC state grouping method.

4.2.4.1 State Grouping Results

Table 4.2 compares the number of groups (G) and the average number of Member states Per Group (MPG) generated after performing the state grouping through the DC and FEACAN. The G and MPG values are reported for different values of the transition threshold (T) which is represented as a percentage of the minimum number of identical transitions within the group. This experiment intends to show the relationship between T and the state grouping results. So, the maximum number of permitted states per group (B) is set to 256 in this experiment. After evaluating the effect of the transition threshold on the state grouping results, the effect of varying the maximum number of states is evaluated separately in the later part of this section.

In the case of FEACAN grouping, as the value of T increases, there is an increase in the number of groups generated with a corresponding reduction in the MPG. Since the FEACAN grouping is not compression-aware, a state is not always clustered into the best group due to the strict grouping policy associated with the transition threshold. Moreover, as the value of T increases, more new groups are formed due to the clustering constraint imposed by the transition threshold. This can be seen in Table 4.2 in which the G value increases with a reduced MPG corresponding to an increase in the transition threshold, T across all the signature sets. On the other hand, in the case of the DC state grouping method, the conquer step combines the state groups with a liberal policy associated with the transition threshold. This can be seen through state groups converging to similar G and MPG values across different values of T. So, the G and the MPG values are independent of the chosen T value in the DC state grouping method.

Table 4.2: Comparison of the Number of Groups and the Average Members per Group between FEACAN & DC, B=256

T	**70%**				**80%**				**90%**			
Grouping	**FEACAN**		**DC**		**FEACAN**		**DC**		**FEACAN**		**DC**	
	G	**MPG**	**G**	**MPG**	**G**	**MPG**	**G**	**MPG**	**G**	**MPG**	**G**	**MPG**
Snort34	564	24	519	26	575	23	518	26	1163	11	519	26
Snort31	496	38	410	47	584	32	420	45	5456	3	431	44
Snort24	488	27	416	32	538	25	428	31	857	16	439	31
Exact_Match	298	50	109	138	298	50	109	138	304	49	107	141
Bro217	137	47	91	71	173	37	98	66	273	23	94	67

4.2.4.2 Transition Compression Rate

Table 4.3 compares the transition compression rate achieved in the MSBT, when the FEACAN and the DC grouping are used in the state grouping step. Columns 2, 4 and 6 show the transition compression rate achieved through the MSBT across different values of T when the FEACAN state grouping algorithm is used for state grouping. Similarly, columns 3, 5 and 7 show the transition compression rate achieved through the MSBT when the states are grouped using the DC state grouping method across different values of T. It can be seen from Table 4.3 that an improvement of the order of 0.1-2% is seen in the transition compression rates in the MSBT, when the state grouping method is modified from FEACAN to DC. The improvement is directly attributed to the compression-aware state grouping approach proposed in the DC. Figure 4.6 compares

Table 4.3: Comparison of transition compression rate achieved in MSBT with FEACAN and DC state grouping, B=256

T	**70%**		**80%**		**90%**		**A-DFA**
Grouping	**FEACAN**	**DC**	**FEACAN**	**DC**	**FEACAN**	**DC**	
Snort34	98.50	98.60	98.49	98.60	97.92	98.60	99.02
Snort31	97.42	97.46	97.42	97.96	93.61	97.96	98.79
Snort24	97.92	98.61	98.05	98.61	98.09	98.62	99.09
Exact_Match	96.96	99.06	96.96	99.06	97.42	99.07	99.60
Bro217	95.34	98.43	96.61	98.44	97.06	98.43	99.57

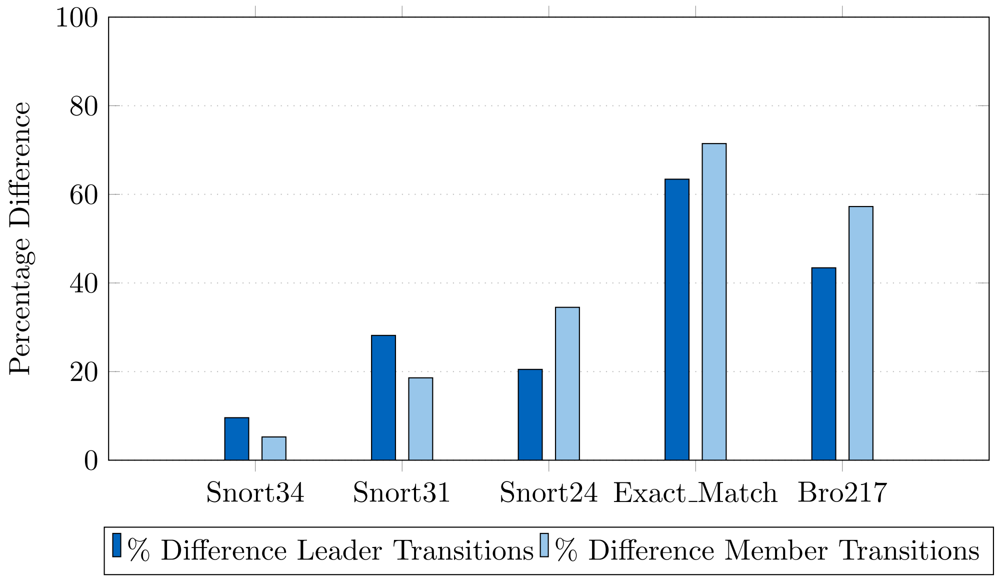


Figure 4.6: Percentage reduction in the Leader and Member transitions between FEACAN and DC grouping in MSBT, T=80%, B=256

the percentage reduction in the number of leader and member transitions generated after the MSBT, when FEACAN and the DC are used for state grouping. The reduction in the leader transitions is directly attributed to the conquer step, in which the groups are merged to optimize the number of generated leader transitions. Similarly, the percentage reduction in the member transitions seen in Figure 4.6 is attributed to the state reorganization and the optimal leader identifications steps, as these steps focus on reducing the number of member transitions in the group.

Table 4.4 compares the transition compression rate achieved through the LSCT, when FEACAN and the DC state grouping methods are used in the state grouping step. Since the LSCT is an improvement over the MSBT, the improvement in the transition compression rate seen in the LSCT is very similar to the results in the MSBT. It can also be seen from Table 4.4 that the DC state grouping, in combination with the LSCT achieves compression results close to that of the A-DFA. Moreover, when the DC state grouping method is used in the MSBT and the LSCT, the transition compression rate that is achieved always converges to an optimal ceiling, irrespective of the value set to T. This shows that the DC state grouping method is compression-aware and clusters the DFA efficiently to result in the best transition compression results.

After evaluating the effect of the transition threshold, additional experiments are performed to evaluate the relationship between the choice for the maximum number of states in a group, B and the resulting transition compression rate. In this experiment, the transition threshold is set to 80%, while the value for B is varied between 128 and 512. Table 4.5 shows the grouping and the transition compression results when B is changed in the DC state grouping. It can be seen that the number of groups generated slightly varies based on the choice of B. As the value for B increases, the number of

Table 4.4: Comparison of transition compression achieved in LSCT with FEACAN and DC state grouping, B=256

T	**70%**		**80%**		**90%**		**A-DFA**
Grouping	**FEACAN**	**DC**	**FEACAN**	**DC**	**FEACAN**	**DC**	
Snort34	98.97	99.04	98.97	99.04	98.85	99.04	99.02
Snort31	97.80	98.29	97.87	98.29	97.58	98.31	98.79
Snort24	98.36	98.99	98.53	99.00	98.85	99.02	99.09
Exact_Match	97.13	99.13	97.13	99.13	97.60	99.13	99.60
Bro217	95.52	98.55	96.85	98.57	97.43	98.55	99.57

Table 4.5: Comparison of transition compression achieved in MSBT & LSCT when the maximum number of states in the groups, T=80%

	B=128				**B=256**		**B=512**			
	G	**MPG**	**MSBT**	**LSCT**	**MSBT**	**LSCT**	**G**	**MPG**	**MSBT**	**LSCT**
Snort34	536	26	98.58	99.03	98.60	99.04	516	27	98.60	99.04
Snort31	462	42	97.93	98.31	97.96	98.29	411	48	97.98	98.31
Snort24	450	31	98.59	99.00	98.61	99.00	424	33	98.61	99.00
Exact_Match	165	92	98.91	99.01	99.06	99.13	85	178	99.14	99.19
Bro217	110	59	98.36	98.51	98.44	98.57	92	71	98.47	98.60

groups generated decreases with an increase in the average number of states per group. Since the value of B defines the room for the maximum number of states in the group, a higher value for B enables more states with similar characteristics to be clustered together. Though there is a very minor improvement in the transition compression rate with a higher B, there is no drastic change in the transition compression rates due to the variation in the value of B.

4.2.4.3 Estimated Memory Usage

Figure 4.7 shows a comparison of the estimated memory required to store the compressed DFA, both in the case of the MSBT and the LSCT. Since the transition compression rates achieved doesn't depend on the value of T and B in the DC state grouping, the estimated memory usage comparison is shown for T set to 80 and B set to 256. Although the DC state grouping only improves the transition compression rates by 0.5-2% in comparison to FEACAN, yet this results in a substantial memory savings of the order of 10-30% in the compressed DFA representation. It can be clearly seen from Figure 4.7 that the reduction in the compressed DFA's memory footprint primarily results from the reduced transition memory usage due to the increase in the transition compression rates. Though the DC state grouping method generates an increased number of member states, this doesn't alter the associated control memory usage in the compressed DFA representation. The observed reduction in the estimated memory usage further verifies the fact that the optimal state grouping plays a significant role in improving the memory footprint of the compressed DFA.

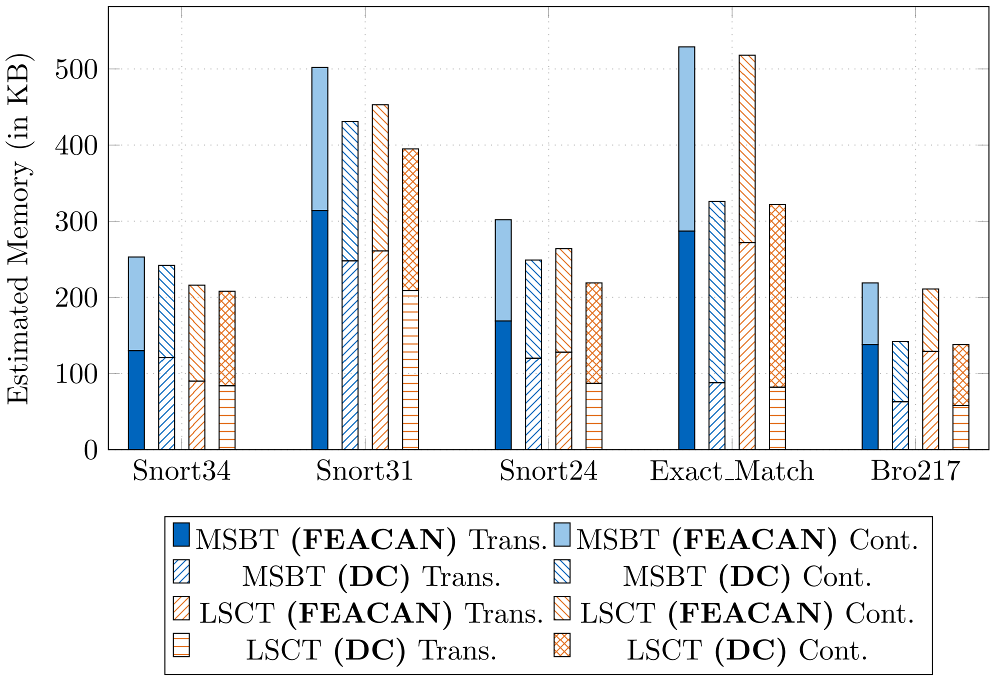


Figure 4.7: Estimated on-chip SRAM memory usage to store the compressed DFA

4.2.5 Discussion & Summary

The state grouping step plays a critical role in defining the transition compression rate that is achieved in the MSBT and the LSCT. The primary aim of the state grouping step is to cluster the DFA and prepare it for the inter-state compression step. However, making the state grouping algorithm compression-aware will effectively group the states to result in better transition compression results. The existing algorithms which have been proposed in the literature do not group states in a coherent manner and the generated groups do not always achieve the best transition compression results. So, a Divide and Conquer state grouping method was proposed in this section which is a group of algorithms that cluster the states in a compression-aware manner. The proposed method is split into the divide and the conquer steps, where the divide step focuses on efficiently clustering a state into the best possible group. After the divide step, the conquer step combines multiple groups into a single group, provided the resulting number of compressed transitions in the combined group is lesser than the sum of the compressed transitions when they are stored separately. Experimental evaluation of the proposed DC with the MSBT and the LSCT, across various signature sets showed an increase in transition compression by about 0.5-2%. Furthermore, the improvement in the transition compression rate reduced the memory footprint of the compressed DFA by about 10-30%, which is a very significant reduction.

The additional cost paid to make the state grouping method compression-aware is the increased complexity in the algorithms proposed as part of the DC state grouping.

Considering that the state grouping is a one-time step performed during the DFA compression, the additional complexity is acceptable. Moreover, performing the transition compression in a compute platform with high processing and memory capabilities can remove the processing cost involved in the state grouping step.

The signature matching engine designed based on the MSBT would require a logical maximum value on the total number of groups that can be supported to fix a boundary for the AMT. In such a case, an additional extension in the conquer step can limit the maximum number of groups generated by combining groups together. However, this may result in a reduced transition compression rate which is a cost that may be incurred to limit the number of groups to be supported. So, using the DC method in this way also grouping opens up an avenue to limit the maximum number of groups in the compressed DFA.

4.3 Alphabet Compression

As described in Chapter 2, alphabet compression is the process of compressing the indistinguishable characters in the character set and in turn compressing the redundant state transitions associated with the indistinguishable characters. One of the biggest advantages of alphabet compression is its is orthogonality to other forms of transition compression [32]. Utilizing this advantage, this section discusses the need to combine alphabet compression with bitmap based compression techniques and details how the alphabet compression is combined with the MSBT and the LSCT.

4.3.1 Combining Alphabet Compression with MSBT & LSCT

The inherent idea behind bitmap based transition compression mechanisms is to compress the identical transitions which are adjacent to each other. However, if a sequence of identical transitions are blocked by a transition which is different from the sequence, the state transition following the non-identical transition[4] cannot be compressed.

In a DFA corresponding to the character set Σ, the identical transitions corresponding to the indistinguishable characters in a state are generally distributed over the character axis. Moreover, the organization of the indistinguishable characters among the character set also depends on the characters used in the signature set. So, there are chances that the identical state transitions corresponding to the indistinguishable characters are blocked by the forward transitions in a state. In such a scenario, bitmap based transition compression mechanisms alone cannot compress all the redundant transitions corresponding to the indistinguishable characters in the state.

So, a two step solution is required to compress the redundant state transitions. As a first step, the alphabet compression should be performed to compress the redundant state transitions corresponding to the indistinguishable characters. As a second step, the transition compression methods such as the MSBT or the LSCT should be performed on the alphabet compressed DFA. Performing the MSBT or the LSCT after the alphabet

[4]The state transition which blocks the sequence the identical transitions

compression removes the intra-state and the inter-state redundancy in the alphabet compressed DFA. So, combining alphabet compression together with the MSBT and the LSCT further reduces the memory footprint of the compressed DFA. However, the cost paid to combine these two methods is the requirement to store the Alphabet Translation Table (ATT) which stores the encoded representation of the original character set.

Figure 4.8 shows a comparison of the compressed DFA corresponding to the signature set *abc* and *egh* generated before and after alphabet compression[5]. Figure 4.8(a) shows the original uncompressed DFA while Figure 4.8(b) shows the compressed DFA after performing the MSBT transition compression alone. The uncompressed DFA consists of 64 state transitions, while the compressed DFA after performing the MSBT consists of 26 state transitions. Figure 4.8(c) shows the transitions which are compressed and the ones which are not compressed in relation to the uncompressed DFA. The empty boxes in Figure 4.8(c) are those state transitions which have been compressed through the MSBT, while the others are the transitions which remain uncompressed. As pointed out in Figure 4.8(c), the state transition corresponding to δ(4,f)=0 is not compressed since it is blocked by the transition δ(4,e)=4, which is different from the state transition corresponding to δ(4,d)=0. This is exactly the scenario which was identified as the drawback when the bitmap alone is used to compress the redundant state transitions in a state.

Figure 4.8(d) and (e), respectively show the compressed DFA after alphabet compression and the compressed DFA after performing the MSBT on the alphabet compressed DFA. As seen in Figure 4.8(e), the compressed DFA consists of 23 transitions in comparison to the 26 compressed transitions in the previous case. As seen in Figure 4.8(f), the scenario observed in Figure 4.8(c) does not happen when both alphabet compression and the MSBT are combined together to compress the redundant state transitions in the DFA.

Since the LSCT is an improvement over the MSBT, performing the LSCT after alphabet compression further enables to compress more redundant state transitions. As seen in Figure 4.8, the failure transitions in the leader states are not fully compressed after the MSBT and the LTB removes these redundant transitions. Figure 4.9(a) shows the compressed DFA after performing the MSBT over the alphabet compressed DFA, while Figure 4.9(b) shows the compressed DFA after performing the LSCT over the alphabet compressed DFA. It can be seen that an additional 5[6] transitions are compressed after the LSCT which are highlighted through the dashed blocks in Figure 4.9(b).

As explained earlier, the cost incurred to perform alphabet compression is the need to store the ATT. For the MSBT and the LSCT implementations, the ATT will be composed of 256 entries each of which is 8-bit wide. The width of each of the character in the encoded character set varies depending on the number of characters that are compressed after alphabet compression. Since the MSBT and the LSCT perform the

[5]The signature set consisting of the signatures abc and egh were used to explain the idea behind alphabet compression in Chapter 2. So the same DFA is used to explain the combination of the transition compression and alphabet compression techniques.

[6]9 transitions are compressed after the MSBT, while 4 of them need to be stored, i.e, one most repeated state transition per leader state.

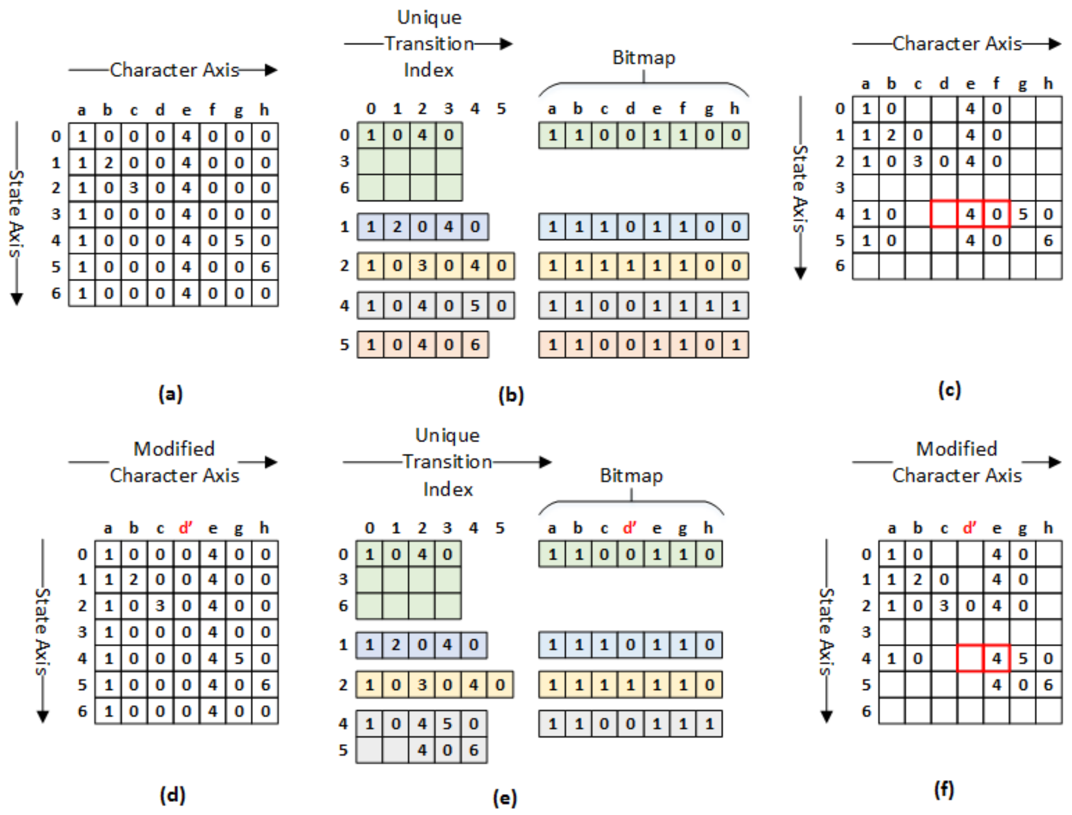


Figure 4.8: (a-c) MSBT transition compression in a DFA without Alphabet Compression (d-f) MSBT transition compression in a DFA after Alphabet Compression

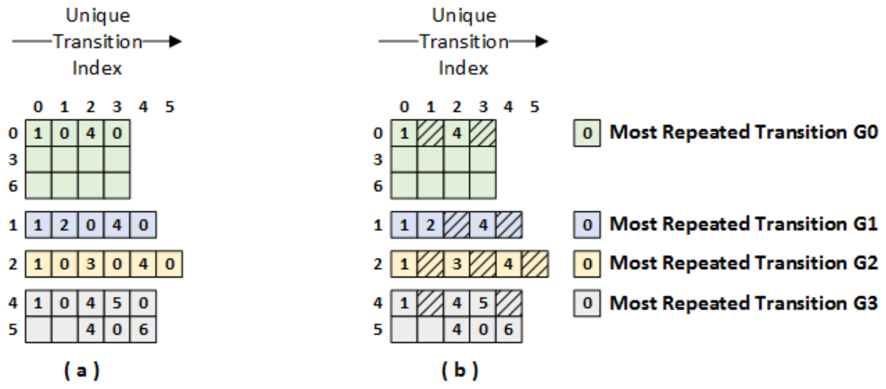

Unique Transition Index
Most Repeated Transition G0
Most Repeated Transition G1
Most Repeated Transition G2
Most Repeated Transition G3
(a)
(b)

Figure 4.9: Example showing the LSCT implemented after Alphabet Compression

transition decompression in a hardware accelerator, 8 bits are required to represent the encoded character set to support worst case scenarios. However, this is a negligible cost in comparison to the additional storage savings achieved due to the improvement in the transition compression.

Combining alphabet compression together with the MSBT and the LSCT does not introduce an additional latency in the corresponding decompression engines. As discussed in Chapter 3, the first stage in the transition decompression decomposes the encoded state representation into the leaderID and the memberID. After this step, the further processing in the transition decompression only continues after fetching the relevant base addresses and the bitmap from the AMT which requires a memory fetch. So, the encoded character corresponding to the original character can be fetched from the ATT in parallel to the AMT memory fetch. In this way, all the information for the transition decompression is available at the same time for the transition decompression to continue.

4.3.2 Experimental Evaluation

There are two algorithms which have been proposed in the literature to perform the alphabet compression. The algorithm proposed by Brodie et al. [64] performs alphabet compression in $\mathcal{O}(|\Sigma|^2 S)$, while the one proposed by Becchi et al. [32] performs in $\mathcal{O}(S^2|\Sigma|)$[7]. Since the ASCII character set is used to define the signatures for DPI applications, the size of the character set is always constant. On the other hand, the number of states generated in the DFA varies depending on the characteristics of the signature set. So, the proposal by Brodie et al. is a better choice for alphabet compression since its time complexity is quadratically related to the size of the character set and not the number of states. Due to this advantage, the alphabet compression algorithm proposed by Brodie et al. has been used for the experimental evaluation in this proposal.

The signature sets which were previously used to evaluate the MSBT and the LSCT are also used to evaluate the effect of combining alphabet compression with the MSBT and the LSCT. Table 4.6 shows a summary of the characteristics of the signature sets before and after alphabet compression. Since, the signatures are composed of the characters from the ASCII character set, there are a total of 256 characters in the DFA before performing alphabet compression. Column 5 in Table 4.6 shows the number of unique characters in the encoded character set generated after performing the alphabet compression. Over 50% of the characters are compressed in the Exact_Match and the Bro217 signature sets, while about 70% of the characters are compressed in the Snort signature sets. Among all the signature sets, a majority of the characters within the extended ASCII character range (128-255) were compressed. The signature sets from Snort (24/31/34) have a smaller signature count in comparison to the Exact_Match and the Bro217 signature sets. So, they contained fewer distinguishable characters from the first half of the ASCII character set (between 0-127), which resulted in a higher number of characters being compressed from the first half of the ASCII character set.

The MSBT and the LSCT was performed on the DFA generated after alphabet compression. The DC state grouping method was used in the state grouping step during

[7]It should be remembered that S represents the overall number of states in the DFA while Σ represents the total number of characters in the character set.

Table 4.6: Summary of the Signature set before and after alphabet compression

	Bef. Alpha. Comp.			**Aft. Alpha. Comp.**		
Signature Set	$\|\Sigma\|$	**G**	**MPG**	$\|\Sigma\|$	**G**	**MPG**
Snort34	256	518	26	74	509	26
Snort31	256	420	45	77	458	42
Snort24	256	428	31	67	453	30
Exact_Match	256	109	138	112	107	141
Bro217	256	98	66	111	100	64

the transition compression. The transition threshold (T) was set to 80% while the maximum number of states per group (B) was set to 256 for this experiment. Column 6 and 7 in Table 4.6, respectively show the number of groups and the average number of member states per group in the compressed DFA that is generated after performing the MSBT/LSCT on the alphabet compressed DFA. It can be seen from Table 4.6 that there are no major variations in the characteristics of the generated state clusters when alphabet compression is introduced in the overall compression process.

4.3.2.1 Transition Compression Rate

Table 4.7 compares the number of compressed transitions generated when the MSBT and the LSCT are used to compress the redundant transitions before and after performing alphabet compression. Columns 2 and 3 respectively show the number of compressed transitions generated when the MSBT is used to compress the DFA before (BAC)[8] and after performing alphabet compression (AAC)[9], respectively. Column 4 in Table 4.7 shows the percentage reduction in the compressed transitions after combining alphabet compression in the case of the MSBT. It can be seen that there is a reduction of about 10% in the compressed transition count, primarily in the case of the DFA's generated from the Snort signature sets. Since the snort signature sets consist of regular expressions with character classes, the redundant state transitions associated with these are efficiently compressed with MSBT performed after alphabet compression. On the other hand, since the Exact_Match and the Bro217 signature sets do not have character classes, the reduction in the compressed transition count is not as substantial as seen in the case of the Snort signature sets. Moreover, the organization of the indistinguishable characters were linear in Exact_Match and Bro which were effectively compressed through the bitmaps alone.

Columns 5 and 6 in Table 4.7 compare the number of compressed transitions generated when the LSCT is used to compress the DFA, before and after alphabet compression. Column 7 shows the additional reduction in the compressed transition count in the case of LSCT after combining alphabet compression. The trend of a higher percentage of

[8]MSBT alone is performed as part of transition compression
[9]MSBT is performed on the alphabet compressed DFA

Table 4.7: Compressed Transitions Before and After Alphabet Compression

	MSBT			LSCT		
Signature Set	**BAC**	**AAC**	**% Diff.**	**BAC**	**AAC**	**% Diff.**
Snort34	49711	44228	9	34175	31087	9
Snort31	101924	93631	8	85444	77467	9
Snort24	49300	44938	11	35433	33060	6
Exact_Match	36182	35055	3	33718	33017	2
Bro217	26117	25911	1	23853	23854	0

transitions compressed in the Snort signature sets is also observed in the case of the LSCT, when it is performed on the alphabet compressed DFA.

4.3.2.2 Estimated Memory Usage

Figure 4.10 compares the estimated memory footprint of the compressed DFA when the transition compression techniques are combined with the alphabet compression. Before Alphabet Compression (BAC) refers to the scenario where the MSBT and the LSCT alone are performed on the DFA, while After Alphabet Compression (AAC) refers to the combination of alphabet compression and the transition compression methods. As in the previous scenarios, the estimated memory usage is split into transition and control memory usage. The reduction in the transition memory seen in Figure 4.10 is a direct consequence of the increased number of redundant transitions compressed by combining alphabet compression together with the MSBT and the LSCT. For example, an additional 9% reduction in the compressed transition count in the case of Snort34 signature set seen in Table 4.7 reduces the memory footprint of the compressed DFA by 20KB. This is roughly about a 10% reduction in the memory footprint corresponding to the compressed DFA. On the other hand, there is no major change in the control memory usage when the transition compression is performed after the alphabet compression. The ATT which only needs 256 bytes to store the encoded character set results in a substantial memory savings of about 20-50 KB which is the result of the additional redundant transitions compressed due to alphabet compression combined with the MSBT and the LSCT.

4.3.3 Discussion & Summary

Though there are 256 characters in the ASCII character set, not all the characters are used to construct the signatures for DPI applications. So, when the signatures are converted into the DFA, some of the characters are indistinguishable, i.e., the state transitions corresponding to these characters are identical across all the states. Since the bitmap based methods focus on compressing the redundant state transitions which are only adjacent to each other, the redundant transitions corresponding to the indistinguishable characters are not always efficiently compressed through the bitmaps alone.

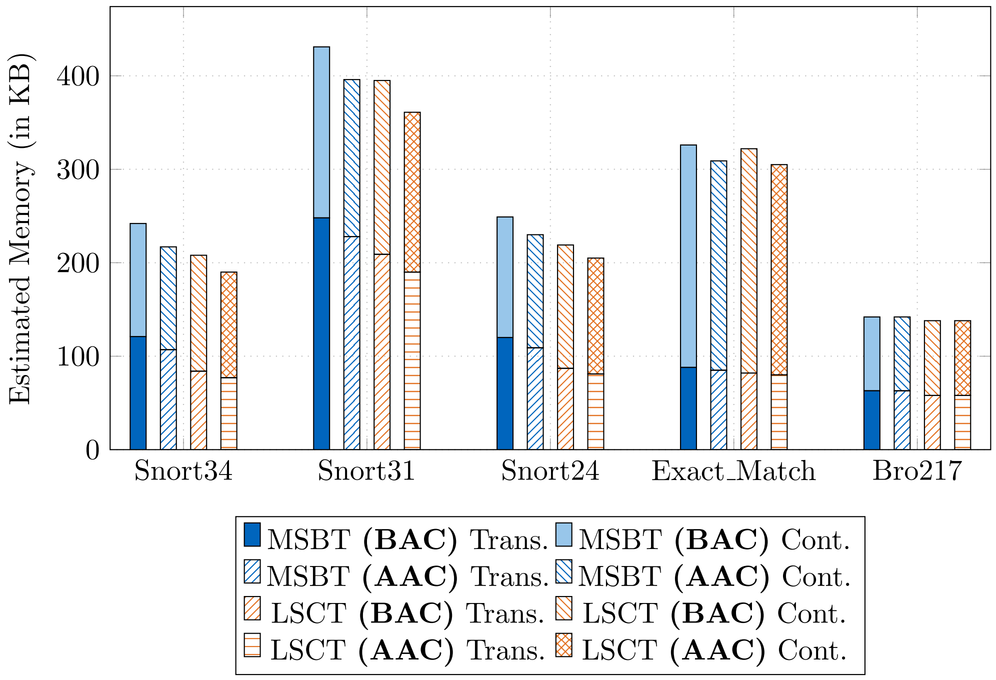


Figure 4.10: Estimated on-chip SRAM memory usage to store the compressed DFA

Alphabet compression is a well known method to compress the state transitions corresponding to the indistinguishable characters in the DFA. In order to efficiently compress the redundant transitions corresponding to the indistinguishable characters, a method was proposed to combine the alphabet compression mechanism with the MSBT and the LSCT. Experimental evaluation further showed that the reduction in the number of redundant transitions further reduces the memory footprint of the compressed DFA by about 10%. The reduction in the memory footprint also depends on the characteristics of the signature set, i.e., the effect of the combined compression methods improves the compression rate, specifically in the signature sets which consist of regular expressions with character classes.

4.4 Bitmask Compression

The MSBT and the LSCT use a combination of bitmaps and bitmasks to effectively compress the redundant state transitions in the DFA. As seen in Chapter 3, the introduction of bitmasks resulted in a considerable increase in the control memory usage in comparison to the previous bitmap based transition compression methods. Since the bitmasks should also be stored in the on-chip memories together with the compressed state transitions, reducing the number of bitmasks will further result in a reduced memory footprint corresponding to the compressed DFA. So, this section proposes a mechanism which leverages the inherent arrangement of characters in the signatures to reduce the number of bitmasks generated after the MSBT and the LSCT.

4.4.1 Background

To further understand the background behind bitmask compression, the DFA corresponding to the signature set 'acd', 'bh' and 'gh' is used as an example as shown in Figure 4.11. The characters in the signatures belong to the alphabet Σ={a,b,c,d,e,f,g,h}. It should be noted that the signature set has multiple occurrences of character *'h'* among the three signatures. Figure 4.11(a) shows the conversion of the signatures into the DFA. The state table corresponding to the DFA is shown in Figure 4.11(b). Figure 4.11(c) and (d), respectively show the intra-state compression and the state grouping steps in the MSBT. The compressed DFA after the inter-state compression step and MTBs are shown in Figure 4.11(e) and (f), respectively.

It can be seen from Figure 4.11(f) that the MTBs corresponding to states 3, 5 and 7 are identical to each other. Similarly, the MTBs corresponding to states 4 and 6 are also identical to each other. In order to further understand the occurrence of the identical MTBs among the states, it is necessary to understand the organization of the state transitions within the DFA. The state transitions within the DFA can be categorized into one of the following subsets, depending on where a state transition leads to during the signature matching function.

- **Root state diverter:** Not all the characters in a character set are always used to construct the signatures in a signature set. So, the state transitions associated with those characters which are not used by the signature set generally direct the DFA towards the root state. For example, the state transitions corresponding to characters *e* and *f*, seen in Figure 4.11(b) belong to this category and the state transitions associated with these characters divert the DFA towards the root state, i.e., state *0*.

- **Initiators:** The state transitions corresponding to the first character in each of the signatures direct the DFA to a single unique state across all the states in the DFA[10]. For example, the state transitions corresponding to the characters *a*, *b* and *g* belong to this category and lead to states 1, 4 and 6, respectively. These state transitions make sure that the first character in a signature is promptly matched irrespective of the current state in which the signature matching function is in.

- **Partial match:** As part of the signature matching function, a series of characters could have led the DFA to a partial match and the state transition belonging to this category further continues this trend, potentially resulting in a plausible signature match. For example, the state transition corresponding to character *h* in states 4 and 6, and the state transition corresponding to character *c* in state 1 belong to this category.

- **Failure match:** Although a sequence of characters can lead the DFA towards a plausible signature match, yet in certain cases after a partial match, an incoming

[10]This is applicable when the first character is not repeated in the same or other signatures. If it occurs, that specific transition will be a part of the *partial match* category.

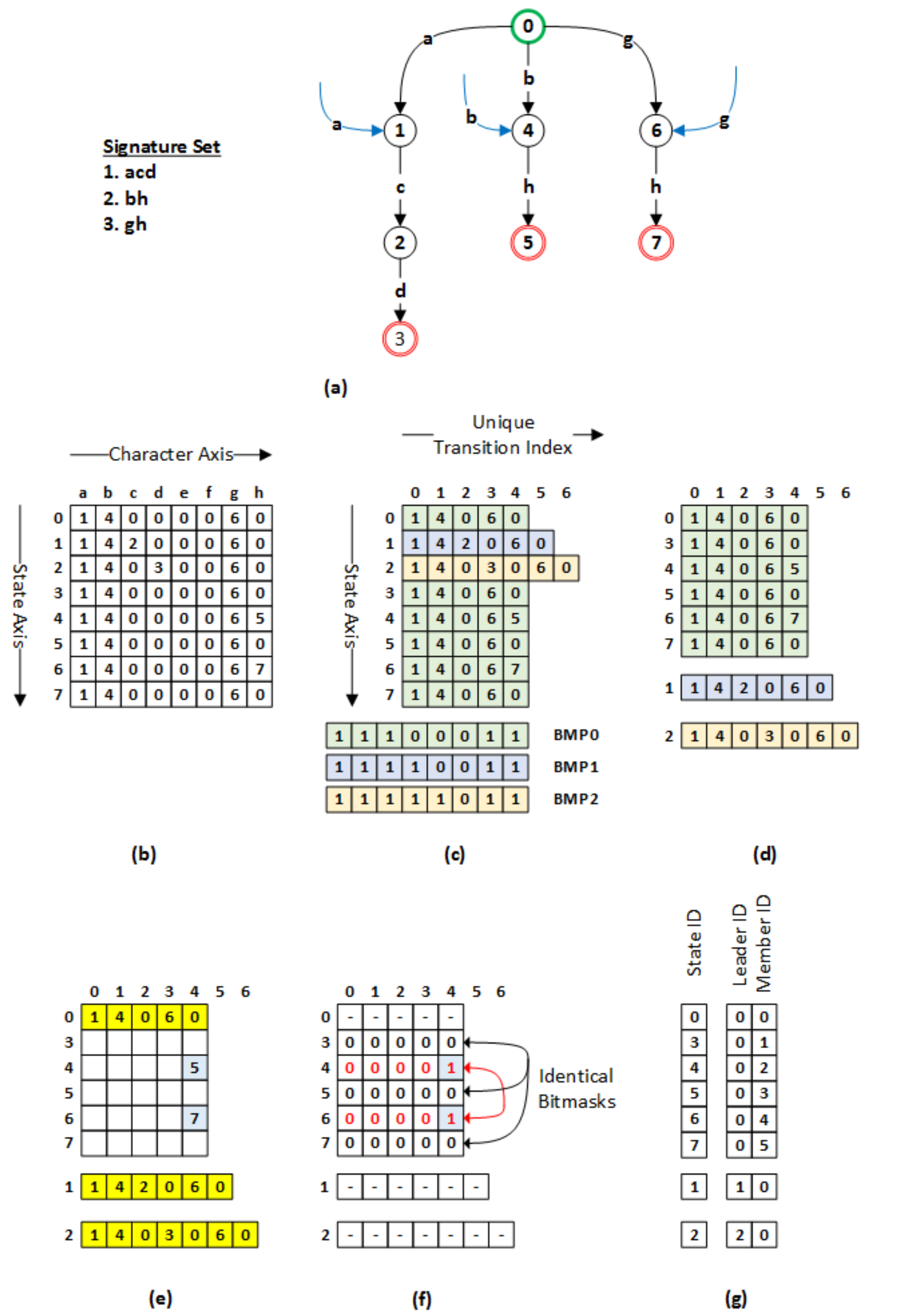


Figure 4.11: Example of redundant bitmasks generated during MSBT transition compression

character may not lead the automata towards a signature match. The state transitions corresponding to the characters apart from the root state diverters and the initiators, which do not continue with the partial (or even full) signature match belong to this category. For example, the state transition corresponding to the character *c* in state 4 belongs to this category. State 4 is reached after successfully matching the character *b* and the DFA generates a signature match if and only if the subsequent character is *h*.

The state transitions which belong to the initiator and the root state diverter are very linear across all the states and the MTB bits resulting from these transitions are generally *0*. On the other hand, the state transitions from the partial and the failure matches are the ones which differ from the state transitions at the leader state, thus generating a *1* in the corresponding MTB bit position. When same characters occur across multiple signatures, the state transitions which belong to the *partial match* will differ from the leader state and generate an identical MTB pattern across different member states. This scenario can be seen in the case of states '4' and '6' corresponding to unique transition index 4 (character *'h'*), where the state transition belongs to a partial match. On the other hand, the state transitions which belong to the *failure match* will more or less be identical with the leader state resulting in identical MTB patterns across the member states, i.e., as seen in the case of states 3, 5 and 7.

The organization of the state transitions within the DFA is highly dependent on the specific characters which are used in the signature set. To summarize, majority of the state transitions in the DFA are linearly organized and the state transitions specific to certain states vary depending on the organization of the individual characters in a signature set. Due to the repetitive usage of certain characters within signature sets, even if the state transitions differ, it corresponds to a specific index resulting in identical MTB patterns during the inter-state compression step in the MSBT. Leveraging this observation, a mechanism to compress the redundant MTBs is proposed to further reduce the memory footprint of the compressed DFA. The combination of bitmask compression together with the MSBT is called as the the Bitmask Optimized Member State Bitmask Technique (BOMSBT). The only difference between the MSBT and the BOMSBT is the bitmask compression step which is performed after the inter-state transition compression step. So, the transition compression rate achieved through BOMSBT is identical to that of the MSBT, while the only difference between the two methods is the total number of MTBs generated in the compressed DFA. The next section provides the details on how the redundant MTBs are compressed in BOMSBT.

4.4.2 Bitmask Optimized Member State Bitmask Technique

4.4.2.1 Bitmask Compression

As seen in Figure 4.11(f), the identical MTBs do not always occur next to each other. So the member states have to be reorganized within a group in such a way that the identical MTBs occur next to each other, so that the redundant MTBs are easily compressed. Figure 4.12(a) shows the member states after reorganization. The states 3, 5 and 7 which had identical MTBs are organized first followed by the states 4 and 6 which also have identical MTBs. The reorganization of the states also require the memberID's corresponding to the states to be modified. For example, as seen in Figure 4.11(g), the original memberID corresponding to state 4 is 2. However, after reorganizing the states, the memberID corresponding to 4 is modified to 4.

After reorganizing the states, the bitmask compression is performed similar to how the redundant transitions were removed in the intra-state compression step using the bitmap. A *unique_bitmask* is generated to identify if the MTB of the member state within a group is compressed during the bitmask compression. The unique_bitmask is as wide as the number of member states in a group. If there is a maximum of B states in a group[11], the unique_bitmask consists of B bits. If the MTB corresponding to a member state is different from its predecessor, the bit position corresponding to the memberID is set to 1 in the unique_bitmask. If the MTB corresponding to a state is identical to its predecessor, then the bit corresponding to memberID in the unique_bitmask is set to 0.

Figure 4.12(b) shows the bitmask compression performed in the example shown in Figure 4.11. The MTB's corresponding to states '5' and '7' are compressed as they are identical to the MTB of state '3'. Similarly, the MTB of state '6' is compressed as it is identical to that of state '4'. Since there is a maximum of 6 states in the first group, a 6-bit unique_bitmask is generated for all of the groups. The bit position corresponding to indices '1' (state ID 3) and '4' (state ID 4) in the unique_bitmask are set to 1 in G0,

[11]The maximum number of states, B was defined as a clustering constraint in the Divide & Conquer State grouping method in Chapter 4.

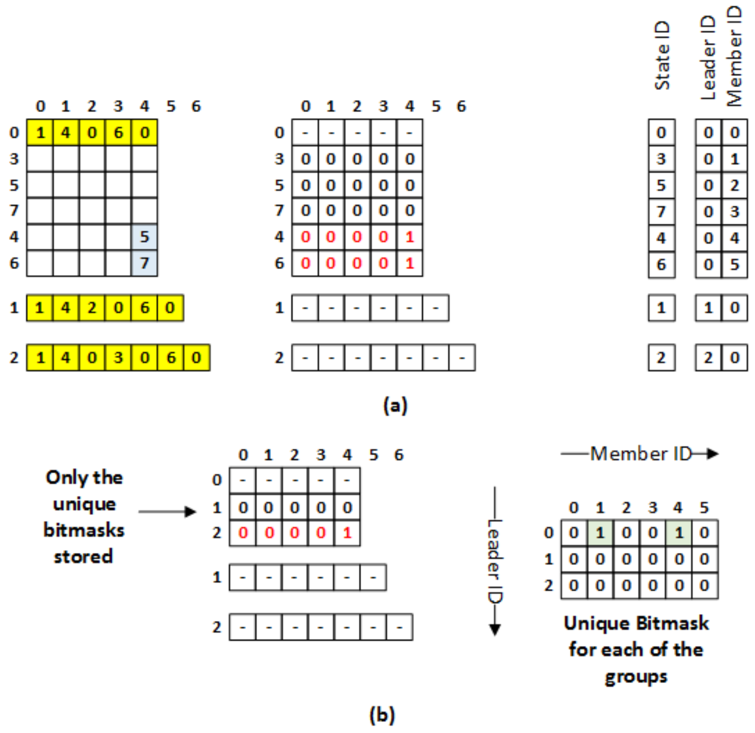

State ID
Leader ID
Member ID
(a)
Only the unique bitmasks stored
Member ID
Leader ID
Unique Bitmask for each of the groups
(b)

Figure 4.12: Compressing the MTBs to reduce the storage requirements of bitmasks

as the MTBs corresponding to these memberID's are not compressed, while the other bits are set to 0. The unique_bitmask bit corresponding to the leader state is always set to 0 as it doesn't have an MTB. Even though the other groups do not have any member states, the unique_bitmask is created for them to maintain the uniformity in the unique_bitmask construction.

Algorithm 5 shows the pseudocode of the bitmask compression and the state reorganization steps performed on each of the groups. As a first step, the MTB corresponding to each of the member state is examined and organized into the 'unique bitmask' set. The unique bitmask set only consists of those MTBs which are indistinguishable. As a next step, each of the MTB in the unique bitmask set is compared across all the member states, so that the member states which have the same MTBs are organized next to each other. The modified memberID for each of the member state is set after a successful MTB match. After assigning the new memberID to a state, the corresponding bit is also set to a 1 or 0 in the unique_bitmask entry.

Algorithm 5 State Reorganization

```
 1: for all $s \in g(M)$ do
                                              ▷ Unique Bitmask Identification
 2:   $s.bmsk\_match \leftarrow 0$
 3:   $s.reorganized \leftarrow 0$
 4:   for all $b \in uniq\_bmsk$ do
 5:     if $b = s.bmsk$ then
 6:       $s.bmsk\_match \leftarrow 1$
 7:   if $s.bmsk\_match = 0$ then
 8:     $uniq\_bmsk \leftarrow uniq\_bmsk \cup s.bmsk$
                                              ▷ State Reorganization
 9: $reorg\_memberID \leftarrow 1$
10: for all $b \in uniq\_bmsk$ do
11:   $set\_unique\_bitmask \leftarrow 1$
12:   for all $s \in g(M)$ do
13:     if $s.reorganized = 0$ & $s.bmsk = b$ then
14:       $s.memberID \leftarrow reorg\_memberID$          ▷ Assign the new memberID
15:       if $set\_unique\_bitmask = 1$ then
16:         $unique\_bitmask[reorg\_memberID] \leftarrow 1$     ▷ Assign Unique Bitmask
17:         $set\_unique\_bitmask \leftarrow 0$
18:       else
19:         $unique\_bitmask[reorg\_memberID] \leftarrow 0$
20:       $s.reorganized \leftarrow 1$                  ▷ Member State is reorganized
21:       $reorg\_memberID{+}{+}$
```

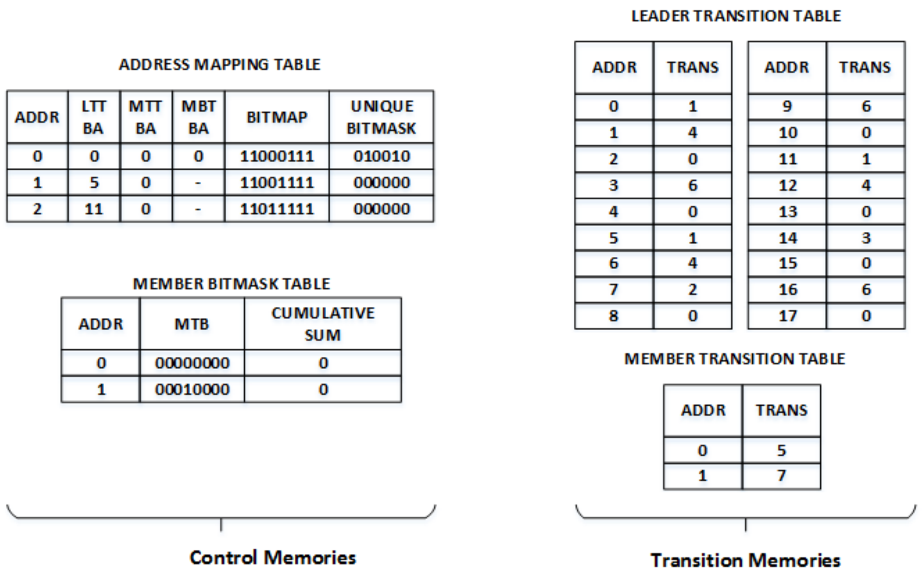
4.4.2.2 Memory Organization

Figure 4.13 shows the organization of the compressed DFA after the BOMSBT. Since the BOMSBT is an extension of the MSBT along with the bitmask compression, the organization of the compressed DFA is very similar to that of the MSBT.

The compressed transitions from the leader and the member states are organized into the Leader Transition Table (LTT) and the Member Transition Table (MTT), respectively. In comparison to the MSBT, the Member Bitmask Table (MBT) only stores the indistinguishable MTBs corresponding to various groups along with the cumulative sum of transitions. The major difference between the MSBT and the BOMSBT is with respect to the organization of the data in the Address Mapping Table (AMT). In addition to the base addresses and the bitmap, the AMT also stores the unique_bitmask for each of the groups in the AMT. Moreover, as far as the MBT base address is concerned, the location of the first compressed MTB in the group is stored in the AMT. Similar to the memory organization in the MSBT, the LTT and the MTT are categorized into the transition memories while the MBT and the AMT are categorized into the control memories.

4.4.2.3 Transition Decompression

The current state-character combination is taken as the input to identify the compressed state transition. The processing associated with the transition decompression depends on whether the current state is a leader or a member state. The computations associated with the leader transition fetch is identical to the ones performed in the MSBT. So, this is not explained again in this section. On the other hand, the computations performed



ADDRESS MAPPING TABLE

ADDR	LTT BA	MTT BA	MBT BA	BITMAP	UNIQUE BITMASK
0	0	0	0	11000111	010010
1	5	0	-	11001111	000000
2	11	0	-	11011111	000000

MEMBER BITMASK TABLE

ADDR	MTB	CUMULATIVE SUM
0	00000000	0
1	00010000	0

LEADER TRANSITION TABLE

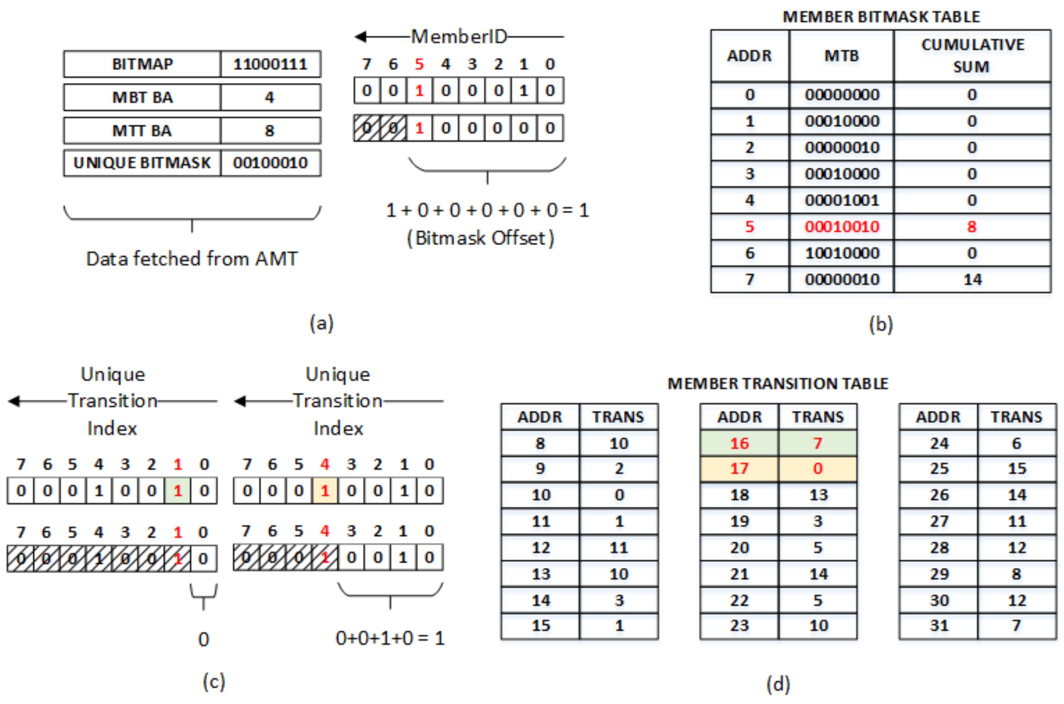
ADDR	TRANS	ADDR	TRANS
0	1	9	6
1	4	10	0
2	0	11	1
3	6	12	4
4	0	13	0
5	1	14	3
6	4	15	0
7	2	16	6
8	0	17	0

MEMBER TRANSITION TABLE

ADDR	TRANS
0	5
1	7

Control Memories
Transition Memories

Figure 4.13: Organization of the compressed DFA between the control and the data memories



MEMBER BITMASK TABLE

ADDR	MTB	CUMULATIVE SUM
0	00000000	0
1	00010000	0
2	00000010	0
3	00010000	0
4	00001001	0
5	00010010	8
6	10010000	0
7	00000010	14

MEMBER TRANSITION TABLE

ADDR	TRANS	ADDR	TRANS	ADDR	TRANS
8	10	16	7	24	6
9	2	17	0	25	15
10	0	18	13	26	14
11	1	19	3	27	11
12	11	20	5	28	12
13	10	21	14	29	8
14	3	22	5	30	12
15	1	23	10	31	7

Figure 4.14: Compressed Transition fetch from the MTT when the unique_bitmask bit for the member state is 1

to identify the compressed transition corresponding to a member state varies depending on whether the MTB of the member state is compressed or not.

Figure 4.14 shows an example of the scenario in which the MTB corresponding to the member state is not compressed as part of bitmask compression. In the case of the MSBT, the memberID and the MBT base address were used to locate the MTB corresponding to the member state. In the case of the BOMSBT, the unique_bitmask is used to locate the compressed MTB corresponding to the member state. Figure 4.14(a) shows the data fetched from the AMT to initiate the compressed transition fetch[12]. Assuming that the memberID corresponding to the current state (which is a member state) as 5, the calculation of the MTB address from the unique_bitmask is shown in Figure 4.14(a). The population count operation is performed on the unique_bitmask which generates the bitmask offset. The bitmask offset provides the relative position of the MTB of the member state among the compressed bitmasks within the group. The MBT BA stores the address of the first compressed MTB in the group, while the bitmask offset added to the MBT BA provides the location of the MTB of the member state among the compressed MTBs. In the example being discussed, the compressed MTB corresponding to MemberID's 1 through 4 are fetched from MBT address location

[12]It should be noted that the example considered to explain the member transition fetch in Figure 4.14 and Figure 4.15 are not connected to the previous example shown to explain bitmask compression.

'4', while the MTB corresponding to MemberID's 5 through 7 are fetched from MBT address location '5'[13].

Once the compressed MTB is fetched, it is further inspected to decide if the compressed transition has to be fetched from the MTT or the LTT. If the bitmask bit corresponding to the unique transition index in the MTB is '0', then the compressed state transition is fetched from the LTT. On the other hand, if the bitmask bit corresponding to the unique transition index in the MTB is '1', then the compressed state transition is fetched from the MTT. The location of the compressed state transition in the MTT is the sum of the MTT base address, the cumulative sum of transitions stored together with the MTB and the member offset as shown in (4.1). The member offset is calculated by performing the population count operation on the MTB, similar to that in the case of the MSBT. So, as far as the example shown in Figure 4.14(c), the compressed state transitions corresponding to the unique transition indices '1' and '4'[14] are fetched from the MTT address locations 16 and 17, respectively as shown in Figure 4.14(d).

$$MTT\ Address = MTT\ BA + Cumulative\ Sum + Member\ Offset \tag{4.1}$$

If the unique_bitmask bit corresponding to the memberID is '0', the MTT address calculation includes an additional offset computation in comparison to the one shown in (4.1). The *'additional offset'* computes the total number of transitions stored in the MTT, from the state preceding the member state of interest until the member state with which the MTB is identical to. For example, if the state with memberID '7' is considered as the member state of interest as shown in Figure 4.15, the additional offset to be added is '4', i.e., 2 transitions each corresponding to states with memberID '6' and '5', respectively[15]. The additional offset is computed by multiplying the total number of compressed transitions in a member state generated after the inter-state compression step (Trans/State) and the number of member states to be offset with (MID_to_offset) as shown in (4.2).

$$Additional\ Offset = (Trans/State)\ * MID\_to\_Offset \tag{4.2}$$

$$\begin{aligned} MTT\ Address = MTT\ BA + Cumulative\ Sum + Member\ Offset \\ Additional\ Offset \end{aligned} \tag{4.3}$$

[13]The location of the MTB for member states with memberID 1 through 4 are located from MBT address 4 as the MTBs corresponding to the states with memberID 2, 3 and 4 are compressed. The same applies for the states with memberID 6 and 7.

[14]These are the only indices in the member state at which the state transitions are not compressed in the member state.

[15]As shown in the previous example in Figure 4.14(c) & (d), the compressed transitions corresponding to the unique transition indices '1' and '4' in the state with memberID '5' are stored in MTT address locations 16 and 17, respectively. The compressed transitions corresponding to the same indices for the state with memberID '6' are stored in MTT address locations '18' and '19' respectively. These are the 4 transitions which are offset by the 'additional offset' which have to be taken into consideration when the compressed transitions in the state with memberID '7' is fetched from the MTT.

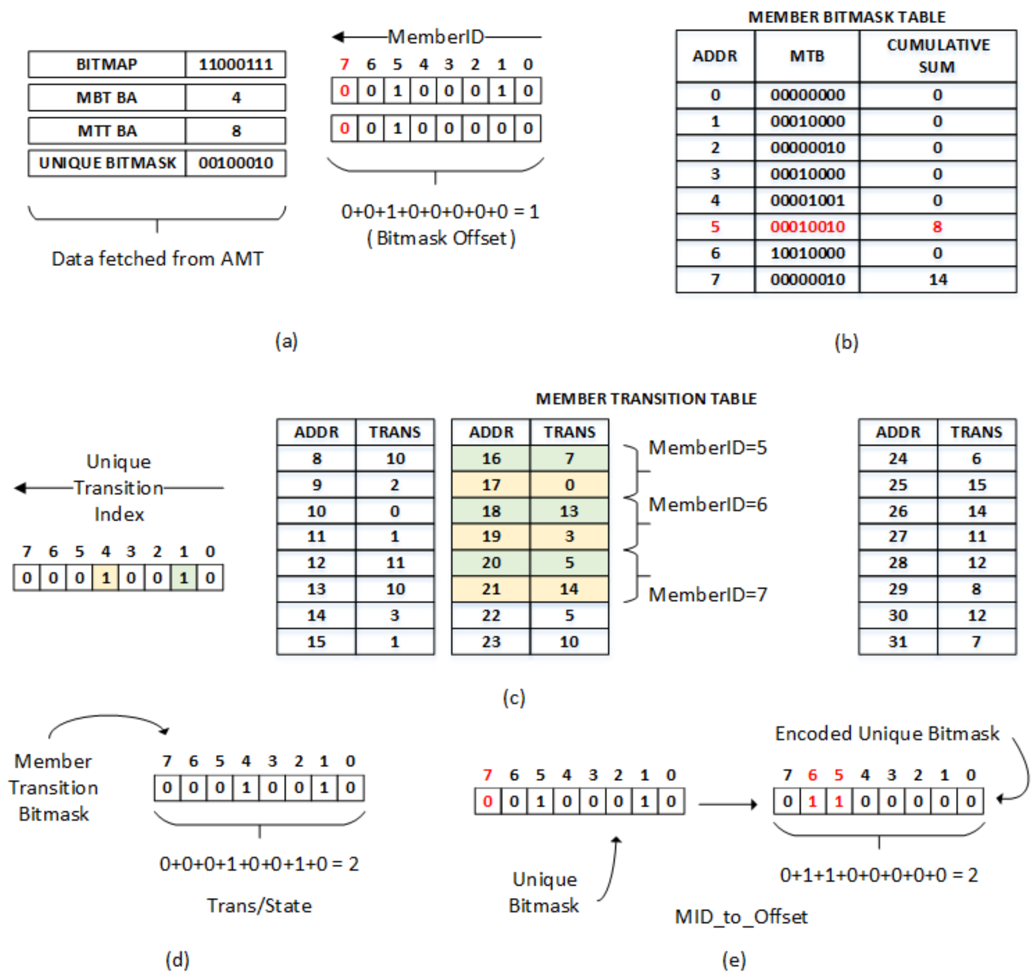


Figure 4.15: Compressed Transition fetch from the MTT when the unique_bitmask bit for the member state is 0

As shown in Figure 4.15(a), since the unique bitmask bit corresponding to the state with memberID ‘7’ is ‘0’, the MTB corresponding to the member state is fetched from MBT location ‘5’[16]. If the bitmask bit corresponding to the unique transition index of interest is ‘0’, then the compressed state transition is fetched from the LTT. If not, the additional offset has to be computed to identify the location of the compressed state transition. Figure 4.15(d) shows the population count operation being performed on the MTB which generates the Trans/State information while Figure 4.15(e) shows the calculation of the MID_to_offset. The unique_bitmask is first converted into an encoded representation on which the population count operation is performed to calculate the MID_to_offset. The MTT address location is computed as shown in (4.3). Based on the

[16]It should be remembered that the bitmask offset is added with the MBT BA from which the MTB is fetched.

equation described above, the address locations of the compressed member transitions corresponding to the unique transition indices ‘1’ and ‘4’ are computed to be ‘20’ and ‘21’ in the MTT.

The additional cost paid for the bitmask compression is the cost to identify the compressed MTB corresponding to the member state during the transition decompression. When the decompression is performed through a hardware accelerator, additional hardware logic circuits will be required to perform the processing associated with the bitmask decompression. The additional hardware cost incurred is further detailed in Chapter 5.

4.4.2.4 Hardware Engine for Decompression - Logical Block Level Description

Similar to the MSBT and the LSCT, the decompression in BOMSBT can be performed in a hardware accelerator. Figure 4.16 shows the logical block level description of the hardware accelerator. Similar to the MSBT, the decompression is performed across three stages. The additional processing which is introduced in the BOMSBT in comparison to the MSBT due to the bitmask compression is highlighted in the blue boxes in Figure 4.16.

The ‘Address Lookup Stage’ (ALS) is the first stage in the transition decompression. The current state identifier is broken down into its leaderID and the memberID from which the leaderID is used to fetch the data from the AMT. The base addresses, the bitmap and the unique_bitmask that is fetched from the AMT is used to calculate the address locations of the leader transition and the compressed MTB which is used in the next stage. Since the bitmask offset is necessary to locate the compressed MTB, the bitmask offset is computed from the unique_bitmask in this stage.

The ‘Leader Transition Bitmask Fetch Stage’ (LTBFS) forms the second stage in the decompression. The leader transition is fetched from the LTT based on the address location calculated in the previous stage. The compressed MTB and the cumulative sum of transitions are fetched from the MBT from the address location calculated in the previous stage. The various offsets (additional offset & member offset) are computed in parallel which are used to calculate the location of the compressed transition to be fetched from the MTT.

The ‘Member Fetch Stage (MFS)’ is the last of the 3 stages, in which the compressed member transition is fetched from the MTT, if required. Depending on whether the current state is a leader or a member state, the next state is assigned accordingly. If the identified next state is an accepted state, a signature match detection signal is set to logic high for a clock cycle.

The idea of bitmask compression can be combined together with the LSCT to form the Bitmask Optimized Leader State Compression Technique (BOLSCT). The LSCT primarily consists of the MTBs and a single LTB for each of the groups. Since the bitmask compression focuses on compressing the redundant MTBs alone, the LTB is not compressed as part of the bitmask compression. So, the resulting processing associated with the bitmask decompression is identical in the case of the BOMSBT and the BOLSCT and is not described further in detail.

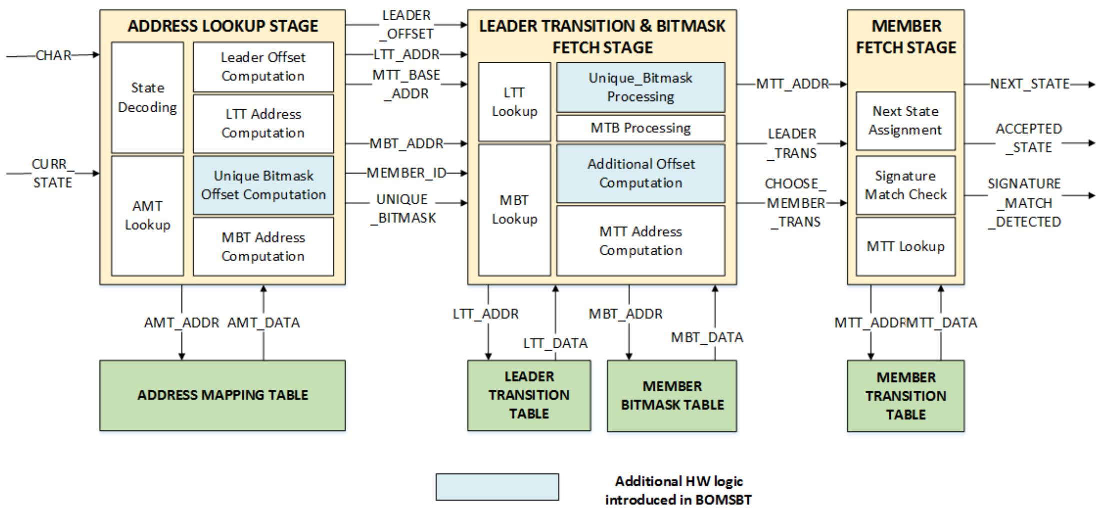


Figure 4.16: Functional Description of the Hardware Accelerator to perform the Transition Decompression

4.4.3 Experimental Evaluation

The BOMSBT and BOLSCT methods were evaluated using the same signature sets that were used to evaluate the other methods described in this dissertation. The DC state grouping method proposed in the previous section was used to group the states in the BOMSBT and the BOLSCT. As part of the evaluation, the transition threshold, T was set to 80% and the maximum number of states within each of the groups, B was set to 256.

Figure 4.17 compares the number of MTBs generated in the MSBT and the BOMSBT, respectively. In the case of the MSBT, the total number of MTBs generated is identical to the total number of member states in the DFA. On the other hand, in the case of the BOMSBT, a majority of the MTBs are compressed through the bitmask compression. It can further be seen from Figure 4.17 that about 60-70% of MTBs are redundant and are compressed through the bitmask compression. However, it should be noted that the individual number of MTBs compressed varies depending on the individual signature sets.

4.4.3.1 Estimated Memory Usage

Figure 4.18 shows a comparison of the estimated memory required to store the compressed DFA generated after the MSBT, the BOMSBT, the LSCT and the BOLSCT, respectively. The memory estimation results are explained in detail through the MSBT and the BOMSBT first. Since the bitmask compression primarily focuses on compressing the redundant MTBs alone, the total number of compressed transitions generated before and after bitmask compression doesn't vary. This can be confirmed from the identical transition memory usage between the MSBT and BOMSBT. On the other hand, the reduction in the number of MTBs after the bitmask compression, reduces the control memory usage in the case of the BOMSBT. The bitmask compression reduces the con-

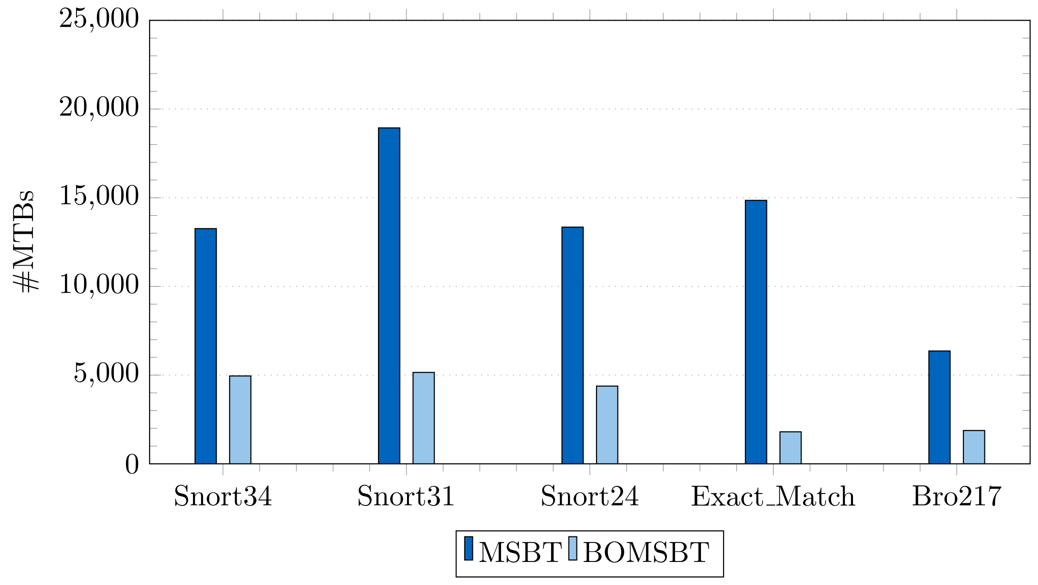


Figure 4.17: The total number of MTBs generated before and after Bitmask Compression

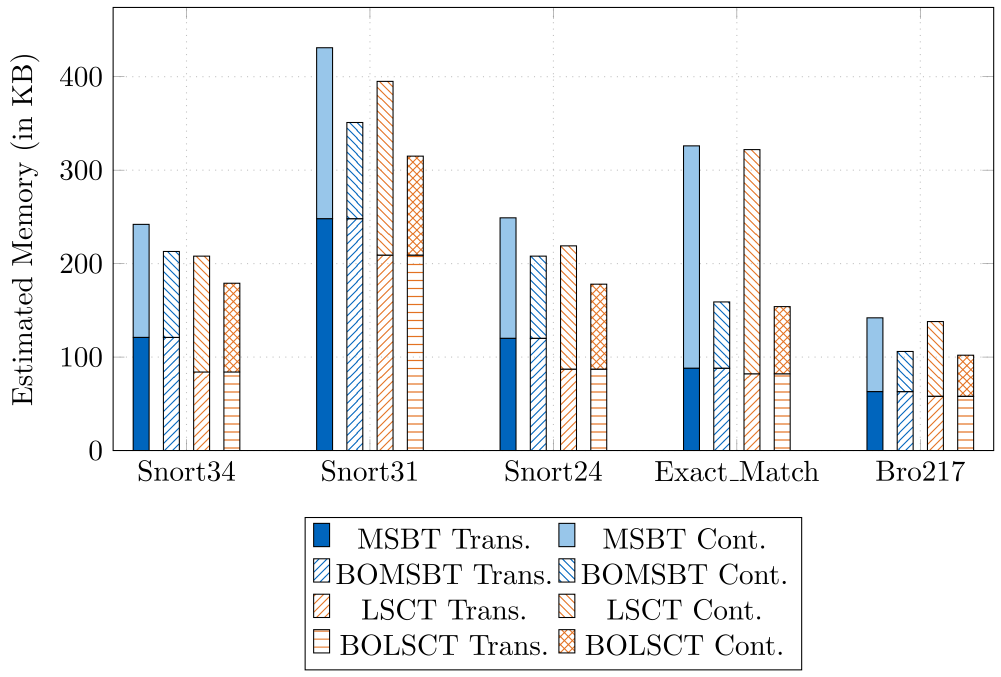


Figure 4.18: Comparison of estimated on-chip SRAM memory usage to store the compressed DFA before and after bitmask compression

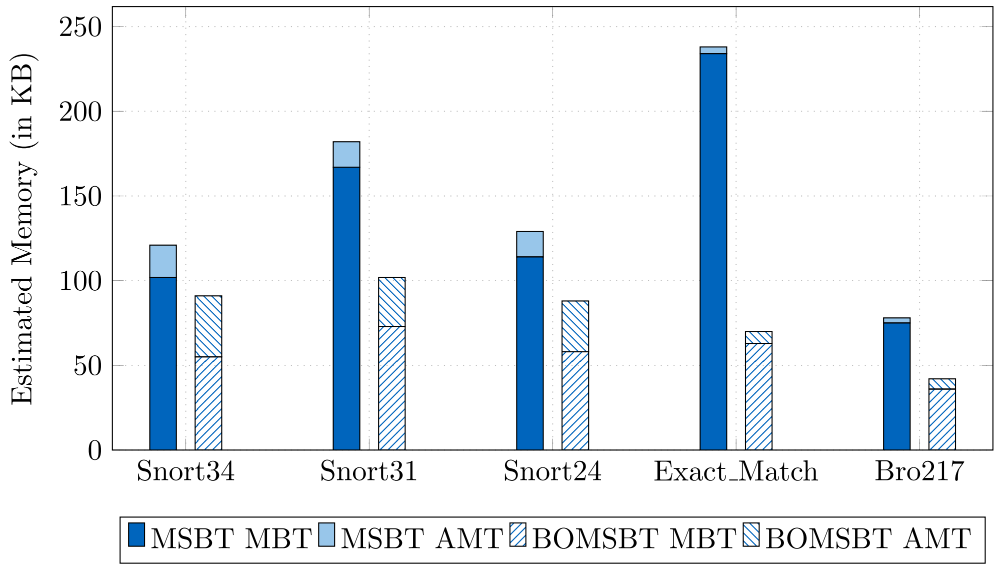


Figure 4.19: Comparison of the control memory usage between MSBT and BOMSBT

trol memory usage by 30-50%, as seen from Figure 4.18. Moreover, it can also be seen from Figure 4.18 that the control memory reduction in BOMSBT is not directly proportional to the reduction in the number of MTBs as seen in Figure 4.17. This is primarily due to the increase in the memory required to store the unique_bitmask for each of the groups to identify if the MTBs corresponding to the member states are compressed or not. Since the unique_bitmask is stored for each of the groups in the AMT, there is a small increase in the memory required to store the AMT, which offsets the reduction in the MBT memory usage. The increase in the AMT usage is clearly seen in Figure 4.19 which compares the estimated memory used for the AMT and the MBT between the MSBT and the BOMSBT. The number of groups generated after the state grouping step are much higher in the case of the signature sets extracted from Snort (between 400-500), in comparison to the other signature sets (∼100). This is the reason why the AMT memory usage is high in the case of the signature sets from Snort in the case of the BOMSBT, while the others are relatively negligible.

Figure 4.18 also shows a comparison of the estimated memory required for the transition and control memories in the case of the LSCT and BOLSCT. As seen in Figure 4.18, the reduction in the memory usage is very similar to that seen in the case of the BOMSBT. Since the most repeated state transition in the leader state is compressed as part of the LSCT and BOLSCT, the transition memory usage is slightly lesser than the MSBT and the BOMSBT, respectively.

4.4.3.2 Functional Evaluation - Software Model

Though the bitmask compression is performed to compress the redundant MTBs in the member states, it is mandatory to make sure that the generated compressed DFA is functionally equivalent to the original uncompressed DFA. In order to verify the functional equivalence, a software model of the BOMSBT and the BOLSCT based signature matching engines were implemented using the UNIX AWK scripting language. Since the DC state grouping is used to perform the state grouping step in the BOMSBT and the BOLSCT, the DC state grouping method was also verified in this evaluation step.

The synthetic traffic generator models which were originally used to verify the MSBT and the LSCT were reused to verify the BOMSBT and the BOLSCT implementations. It should be remembered that the synthetic traffic traces of 1MB were generated for 4 different P_M values, i.e., 0.35, 0.55, 0.75 and 0.95 for each of the signature sets. After the traffic was injected into the system, the total number of signature matches generated across different P_M values were extracted from the system. Table 4.8 shows the signature matching results which were obtained from the DFA, the BOMSBT and the BOLSCT based signature matching engines. It can be seen from Table 4.8 that the total number of signature matches are identical between all the methods. In addition to the total number of signature matches, the individual character positions at which the signature matches occurred were also identical between all the three methods. Thus, this experiment verified the functional correctness of the bitmask compression and the DC state grouping method.

Table 4.8: Comparison of signature matching results across the compression methods across different P_M values

P_M	Method	Signature Sets				
		Snort34	Snort31	Snort24	Exact_Match	Bro217
0.35	DFA	5	1	29	46	11631
	BOMSBT	5	1	29	46	11631
	BOLSCT	5	1	29	46	11631
0.55	DFA	7	2	45	117	6347
	BOMSBT	7	2	45	117	6347
	BOLSCT	7	2	45	117	6347
0.75	DFA	8	1	22	331	28701
	BOMSBT	8	1	22	331	28701
	BOLSCT	8	1	22	331	28701
0.95	DFA	14591	1286	22818	7841	69976
	BOMSBT	14591	1286	22818	7841	69976
	BOLSCT	14591	1286	22818	7841	69976

4.4.4 Discussion & Summary

The content of the compressed DFA generated after the MSBT and the LSCT are classified into the transition and the control data. The primary focus of the MSBT and the LSCT is to effectively compress the redundant state transitions in the DFA by introducing additional control information in the form of the MTBs and the LTBs. However, a majority of the MTBs are redundant due to the linearity in the state transition organization and they can be compressed to reduce the memory footprint of the compressed DFA. So, the bitmask compression mechanism was proposed in this section to effectively compress the redundant MTBs in the compressed DFA.

As part of the bitmask compression, a state reorganization algorithm was proposed to organize the member states in such a way that the identical MTBs are placed next to each other. After bitmask compression, a unique_bitmask was generated to easily idenitfy the member states with the redundant MTBs. The bitmask compression mechanism was effectively integrated with the MSBT and the LSCT to result in two new compression methods called the BOMSBT and the BOLSCT. Experimental evaluation of the proposed methods further showed that about 60-70% of the redundant MTBs could be compressed through the bitmask compression method. The compression of the redundant MTBs further reduces the control memory footprint of the compressed DFA in the BOMSBT and the BOLSCT by about 30-50%. However, the cost paid for the bitmask compression is the additional processing which has to be performed during the decompression in order to identify the compressed MTB corresponding to a member state.

4.5 Conclusion

The primary focus of the MSBT and the LSCT is to remove the transition redundancy in the DFA through the bitmaps and the bitmasks. Performing the transition compression through the bitmaps and bitmasks allows the decompression to be performed in a hardware accelerator to achieve line rate signature matching. In this process, the compressed DFA is stored in the on-chip memories which allows them to be fetched at low latencies. Since, the on-chip memories are expensive resources due to their area and power consumption in comparison to the off-chip memories, it is important to compress the DFA as efficiently as possible to reduce the memory footprint of the compressed DFA. This will allow more signatures to be stored in the predefined on-chip memory boundaries.

So, three different techniques were proposed in this chapter to reduce the memory footprint of the compressed DFA generated after the MSBT and the LSCT. The first two techniques were targeted towards improving the transition compression rates without compromising on the functional equivalence to the original uncompressed DFA. The compression-aware DC state grouping method added additional intelligence to the state grouping step in the MSBT and the LSCT, which resulted in more redundant state transitions being compressed. Though the DC state method comes at the cost of an increased algorithmic complexity, this is a one time step which is performed during the transition compression and can be managed by performing the function in a processor heavy compute cluster. The second method proposed in this chapter combined the alphabet compression together with the MSBT and the LSCT. The combination of alphabet compression and the proposed bitmap based methods focused on compressing those redundant state transitions in the DFA which could not be compressed through the bitmaps alone. Experimental evaluation showed that the DC state grouping method reduced the memory footprint of the compressed DFA by about 10-30%. Furthermore, combining the alphabet compression together with the MSBT and the LSCT, in combination to the DC state grouping method further reduced the memory footprint by an additional 10%.

The third technique focused on reducing the memory footprint of the bitmasks, which played a key role in improving the transition compression rates in comparison to the state-of-the-art bitmap methods. The bitmask compression primarily focused on compressing those indistinguishable MTBs generated in each of the groups. The redundant MTBs are generated due to the linearity in the organization of the state transitions in the DFA. Experimental evaluation showed that the bitmask compression reduces the memory footprint of the control data in the compressed DFA by about 30-50%.

When all these three methods are combined with the MSBT and the LSCT, the overall memory footprint of the compressed DFA reduces by a factor of about 70% in comparison to the state-of-the-art bitmap based compression method [4]. The importance of the proposed optimizations becomes extremely valuable, when the compressed DFA is downloaded into a signature matching engine with predefined memory boundaries. In such a scenario, the reduction in the memory footprint paves way for more signatures to be packed and stored in the on-chip SRAM in an effective manner. This also allows the signature matching to be performed at line rate with an increased signature count.

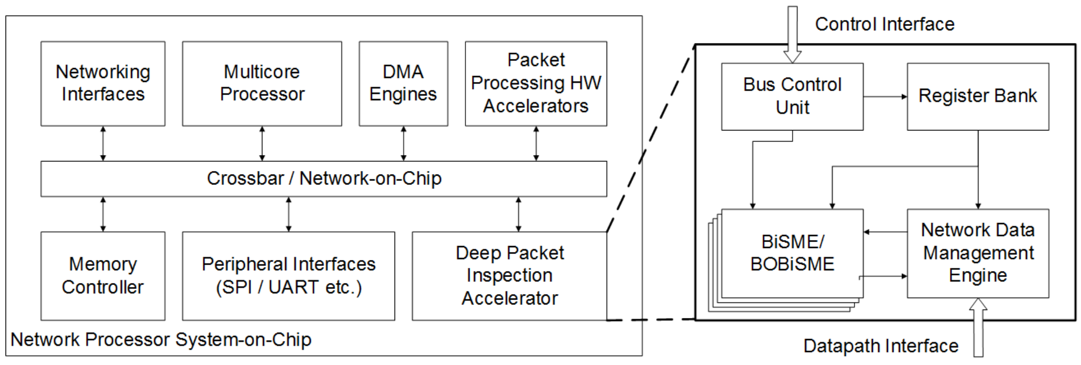
The next chapter describes the architecture of the signature matching engine which is developed based on the MSBT and the BOMSBT compression techniques. The next chapter details the nuances required to design a signature matching engine which is programmable, scalable and flexible which are the key requirements for it to be part of a network processor.

5 Hardware Coprocessors for Signature Matching

5.1 Overview

This chapter details the internal hardware architecture of the signature matching engines proposed in this dissertation. Figure 5.1 shows an overview of the Deep Packet Inspection Accelerator (DPIA) which consists of the signature matching engines (BiSME/BOBiSME) which perform the signature matching function at multi-gigabit line rates. Though the DPIA consists of various blocks, the signature matching engines form the primary focus of this dissertation. The internal architecture of the signature matching engines is described from a bottom up approach. The storage methodologies associated with the compressed DFA is described first, after which the logical processing blocks are described.

As discussed in the previous chapters, the compressed DFA generated after the MSBT/BOMSBT is split into the transition and the control memories. The number of compressed state transitions and the control information generated are highly non-linear and heavily depend on the characteristics of the signature sets. However, in order to design a programmable and a flexible signature matching engine, it is important to define efficient storage architectures to store the compressed DFA. So, this chapter first proposes two different storage methodologies which can flexibly store the compressed DFA. Section 5.2 discusses the *Packed Storage Methodology*, which proposes an efficient method to store the bitmasks in the on-chip memories. Later, Section 5.3 describes the *Shared Memory Methodology* through which the compressed state transitions are stored in a programmable and a configurable manner in the on-chip memories.



Networking Interfaces
Multicore Processor
DMA Engines
Packet Processing HW Accelerators
Crossbar / Network-on-Chip
Memory Controller
Peripheral Interfaces (SPI / UART etc.)
Deep Packet Inspection Accelerator
Network Processor System-on-Chip
Control Interface
Bus Control Unit
Register Bank
BiSME/ BOBiSME
Network Data Management Engine
Datapath Interface

Figure 5.1: Overview of the Deep Packet Inspection Accelerator

A bitmap based signature matching engine called BiSME is proposed in section 5.4 which is capable of performing signature matching at 9.3 Gbps. The BiSME utilizes the storage methodologies discussed in sections 5.2 and 5.3 to efficiently store the compressed DFA and performs the decompression through dedicated logic circuits. Furthermore, the BiSME is optimized to support the bitmask compression and the resulting signature matching is called as the Bitmask Optimized BiSME (BOBiSME). The BOBiSME is capable of performing signature matching at 10.6 Gbps. Section 5.5 proposes the *Deep Packet Inspection Accelerator*, which engulfs the proposed signature matching engines and discusses how they can be integrated with the network processors used by the RGR. Finally, section 5.6 thoroughly evaluates the proposed signature matching engines. The signature matching engines were synthesized on a commercial 28nm library and the synthesis results are discussed first in this section. The functionality of the proposed engines was evaluated on the Cadence Palladium platform by injecting multiple gigabytes of payload bytes into the emulated system. The payload bytes were simultaneously injected into a DFA based signature matching engine which was implemented as a software running on a linux platform. The signature matching results from all the systems were compared and total number of signature matches were identical across all the implementations which further verified the functionality of the BiSME and BOBiSME.

5.2 Bitmask Storage

5.2.1 Requirements

The bitmap and the bitmasks are the major control information which are stored in the on-chip SRAM which help to identify the compressed state transition corresponding to the state character combination. Since the width of the bitmap[1] is constant across all the states in the DFA, it is straightforward to store it in the on-chip memories. On the other hand, the width of the MTB which is generated during the inter-state compression step is not uniform across the signature sets. Moreover, even within a signature set, the width of the MTB varies across different groups. The reason for the same is explained further.

Figure 5.2 shows the average number of compressed state transitions in a state after the intra-state compression step and the average number of unique characters across different signature sets, which have been used for the evaluation purposes. The primary vertical axis (shown in red in the left) represents the average number of compressed transitions generated in a state after the intra-state compression step. The secondary vertical axis (shown in blue) represents the unique number of characters in a signature set. It can be seen from Figure 5.2 that the average number of state transitions generated after the intra-state compression step varies across different signature sets. This observation can be attributed to the variation in the character combinations found in the signature sets. For example, as seen in the secondary vertical axis in Figure 5.2, the more the

[1]Since the ASCII character set is used to describe the signatures, the width of the bitmap is always 256 bits.

number of unique characters in the signature sets, the higher the number of compressed state transitions that are generated after the intra-state compression step. Since the width of the MTB depends on the number of transitions generated after the intra-state compression step, the MTB width varies depending on individual signature sets.

Figure 5.2 also shows the variation in the number of compressed state transitions generated across different groups after the intra-state compression step (shown through error bar). Since different bitmap patterns are generated across different groups in a signature set, the number of state transitions compressed during the intra-state compression depends on the bitmap of the group. Consequently, the width of the MTB also depends on the number of state transitions which are compressed after the intra-state compression step. To summarize, unlike the bitmap which is uniform across the different signature sets, the MTBs are irregular, non-linear structures which have to be carefully stored in the memory. So the following are the requirements which have to be addressed by the MTB storage methodology.

- The MTB storage methodology should be able to support the storage of the MTBs of varying widths. This also includes the theoretical worst-case scenario, in which none of the state transitions may be compressed as part of the intra-state compression step. In such a case, the maximum width of the MTB would be 256 bits. As discussed in Chapter 3, the cumulative sum of state transitions is also stored along with the MTB. Assuming that 16 bits are used to represent the cumulative sum of transitions; in worst case scenarios, a 272 bit bitmask[2] has to be stored in the memory.

[2]As part of the rest of the dissertation, the term bitmask will be used to represent the MTB and the cumulative sum of transitions together.

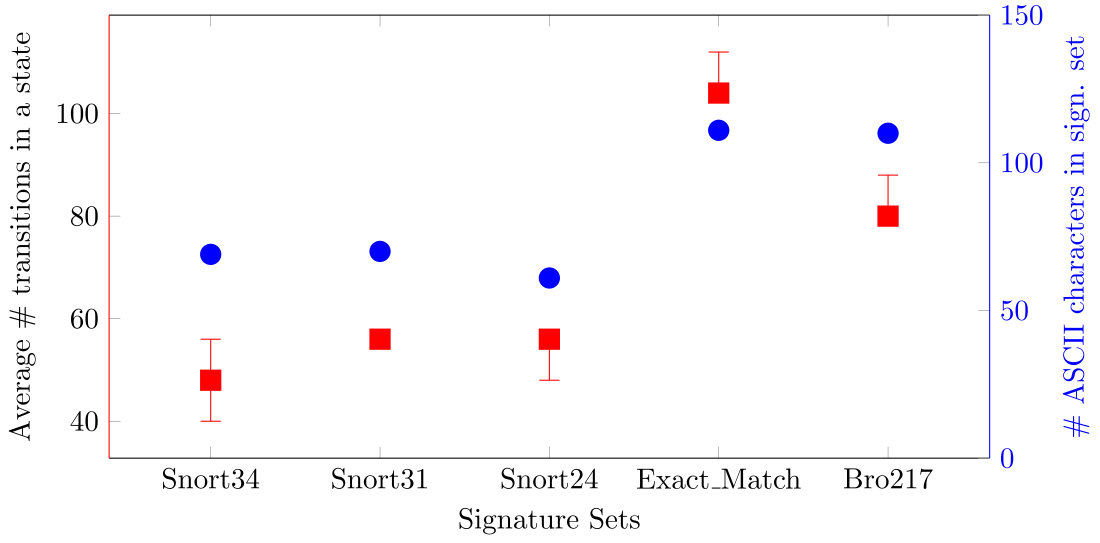


Figure 5.2: MTB width variation across signature sets and within a signature set

- The MTB storage methodology should be able to fetch the MTB from the on-chip memories in a single clock cycle. Fetching the MTB across multiple clock cycles will introduce arbitration in the memory accesses and also complicates the signature matching process across multiple streams. So, the MTB should be stored in the memory in such a way that it can be fetched in a single clock cycle.

5.2.2 Packed Storage Methodology

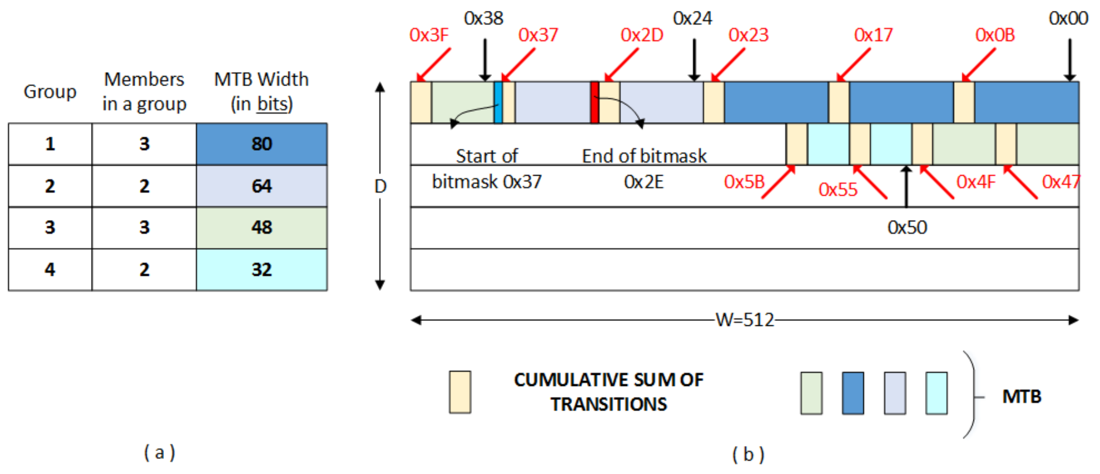
In order to satisfy the requirements discussed above, the *Packed Storage Methodology* (PSM) is proposed to store the bitmasks in the on-chip memory without resulting in any memory wastage. Figure 5.3(a) shows an example of the bitmasks corresponding to 10 member states, split across 4 different groups with different MTB widths. Figure 5.3(b) shows the storage of the bitmasks using the PSM for the bitmasks considered in Figure 5.3(a). As part of the PSM, the width of the physical memory, W is chosen to be of a value which is greater than 272 bits (512 in this example), so that the longest bitmask can be accommodated in a single address location. In the PSM, once a bitmask corresponding to a member state is stored, the bitmask corresponding to the next member state is immediately stored in a contiguous fashion without wasting precious memory resources.

In order to make sure that the bitmask corresponding to the member state is accurately located, the physical memory which stores the bitmasks is made byte addressable. The Most Significant Byte (MSB) address of a bitmask is calculated to identify the physical address location of the bitmask in the memory. The red arrows in Figure 5.3(b) point the location of the most significant byte in each of the bitmasks considered in Figure 5.3(a). Similarly, the black arrows shown in Figure 5.3(b) identify the location of the least significant byte position of the first bitmask (bitmask corresponding to the first member state) in every group and is called as the *bitmask base address*. The most significant byte location of the bitmask is calculated using the bitmask base address and the bitmask length (in bytes). The bitmask length refers to the sum of the width of the MTB and the width of cumulative sum of transitions in bytes.

As discussed in Chapter 3, the compressed state transition is encoded as a combination of the leaderID and the memberID. The location of the most significant byte in the bitmask is calculated as the product of the memberID and the bitmask length which is further added to the base address as shown in (5.1). For example, the location of the most significant byte corresponding to the member state 2 (MemberID: 0x2) in group 2 is calculated as shown in (5.2). Once this address is calculated and the bitmask length is known, the actual MTB and the cumulative sum of transitions can be fetched between address locations 0x37 and 0x2E as shown in Figure 5.3(b). Table 5.1 shows the calculated addresses for all the bitmasks considered in Figure 5.3(a).

$$\text{Bitmask MSB} = (\text{MemberID} \times \text{Bitmask Length}) + \text{Bitmask Base Address} - 1 \quad (5.1)$$

$$\text{Bitmask MSB} = (0\text{x}2 \times 0\text{xA}) + 0\text{x}24 - 1 = 0\text{x}37 \quad (5.2)$$



Group	Members in a group	MTB Width (in bits)
1	3	80
2	2	64
3	3	48
4	2	32

Figure 5.3: Example of how the bitmasks are stored using the "Packed Storage Methodology"

Table 5.1: MTB address calculation example

leaderID	Base Address	Bitmask Length	Member ID	Calculated Address
0x0	0x00	0xC (96 bits)	0x1	0x0B
			0x2	0x17
			0x3	0x23
0x1	0x24	0xA (80 bits)	0x1	0x2D
			0x1	0x37
0x2	0x38	0x8 (64 bits)	0x1	0x3F
			0x2	0x47
			0x3	0x4F
0x3	0x50	0x6 (48 bits)	0x1	0x55
			0x1	0x5B

5.2.2.1 Split Memory Implementation

As shown in Figure 5.4(a), due to the contiguous storage of the bitmasks in the on-chip memory, there are chances that a single bitmask can be stored across multiple physical locations. In such a scenario, multiple clock cycles are required to fetch the bitmask from the memory which is not desirable. In order to support this scenario, the physical memory in which the bitmasks are stored is split width-wise into multiple smaller memories. As shown in Figure 5.4(b), the memory is broken down into 4 blocks with each of width 128 bits. The individual memory blocks are referred as A, B, C and D where memory block A stores the bits 127 to 0 while memory block D stores the bits 511 to 384. The individual address locations from which the data is fetched from each of the memory block depends on which memory block the most significant byte of the bitmask belongs to.

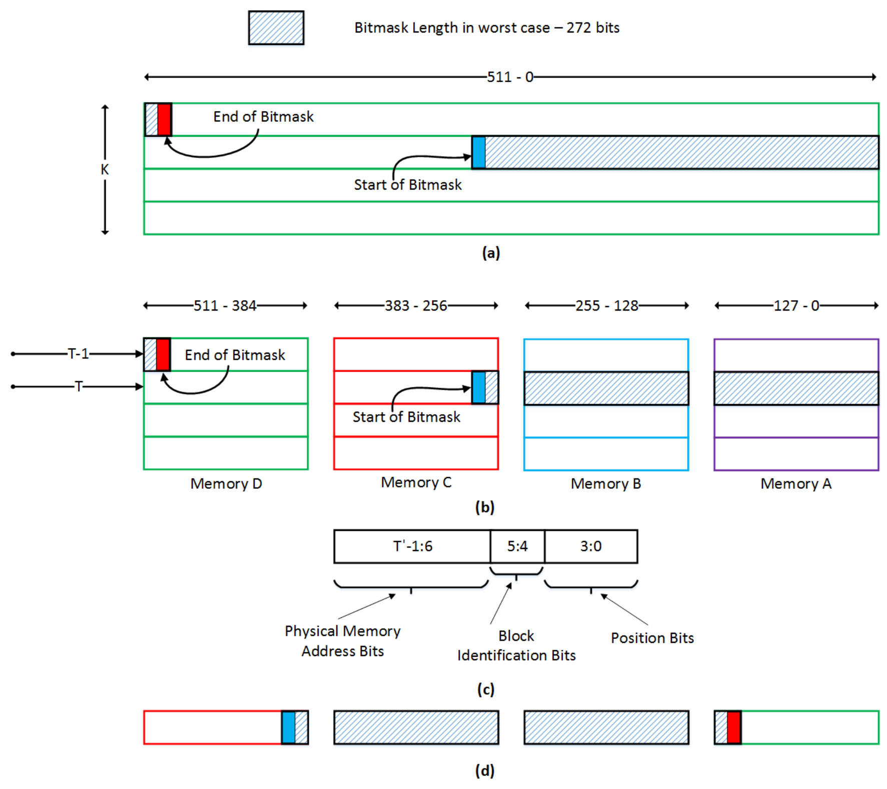


Figure 5.4: Packed Storage Methodology - split memory storage implementation

To generalize this discussion, the longest bitmask of width LB bits can be fetched in a single clock cycle, if it satisfies the condition shown in (5.3). In (5.3), A′ refers to the number of individual memory blocks into which the memory is split into, while BW refers to the width of each of the memory blocks in bits[3]. The worst case scenario with respect to the longest bitmask storage is when a single byte in the bitmask has to be stored in one of the blocks (e.g., block A), while the rest of the bitmask doesn't fit into the remaining blocks (e.g., blocks B,C,D). In this situation, longest bitmask overflows and has to be stored in the block in which the first byte was stored (block A), but in the next address location. This scenario would require two clock cycles to fetch the bitmask. So, the values for BW and A′ have to be chosen in such the way that the condition in (5.3) is satisfied, which makes sure that the above mentioned problem doesn't occur.

$$\left\lceil \frac{\text{LB - 8}}{\text{BW}} \right\rceil < \text{A}', \text{ such that } \{\text{W} > \text{LB}\} \tag{5.3}$$

Figure 5.4(c) shows the organization of the T′ bit address which identifies the location of the most significant byte in the bitmask. The calculated address is split into 3 different portions. The lowermost 4 bits are called as the *Position Bits*, i.e., those bits which

[3]W=BW×A′

identify the location of the most significant byte within a memory block. The next 2 bits, i.e., the bits 5 and 4 are called as the *Block Identification Bits*, which identify the physical block to which the most significant byte address location belongs to. The third portion is called as the *Physical Memory Address Bits* and is used to identify the physical memory address of the memory block in which the most significant byte is stored. So, for example, if T′ is 12 bits wide, the physical memory address bits is 6 bit wide, which implies that each memory block (A/B/C/D) has 2^6 address locations with 128 bits of information stored in each of the blocks. To generalize, $\log_2(BW/8)$ bits are required to represent the position bits and $\log_2(A')$ bits are required to represent the block identification bits.

The physical memory address corresponding to the calculated address location is assumed to be T. As far as the example considered in Figure 5.4(b), since the most significant byte location starts from memory block C, the bitmask is fetched from address location T in memory blocks C, B and A while the data is fetched from the location T-1 for memory D. Once the data is fetched from the memories, the 512 bit data is reconstructed as shown in Figure 5.4(d). Figure 5.5 shows a generic way of how the 512 bit data is reconstructed depending on the block identification bits in T′. The reconstructed data from the physical memory address T is shown using *Dout[T]* while the ones from physical address T-1 is shown with a *Dout[T-1]* in Figure 5.5.

5.2.2.2 Bitmask Extraction

Figure 5.6 details the various steps to be performed to extract the bitmask and to convert it into a usable form. Figure 5.7 shows the associated hardware structures which are required to manage the memory accesses associated with the physical memory blocks and the bitmask extraction, respectively. As shown in Figure 5.7, the circuits are broadly split into the *bitmask address preprocess* and the *bitmask extraction* blocks. The bitmask address preprocess block is responsible to assign the address to each of the physical memory blocks, while the bitmask extraction block is responsible to extract the bitmask from the data fetched from the different memories. As discussed earlier, the data from the physical memory blocks are either fetched from the physical address T′ or T′-1 depending on the block identification bits which is decided using a multiplexer. The select signals for the individual multiplexers are generated based on the block identification bits.

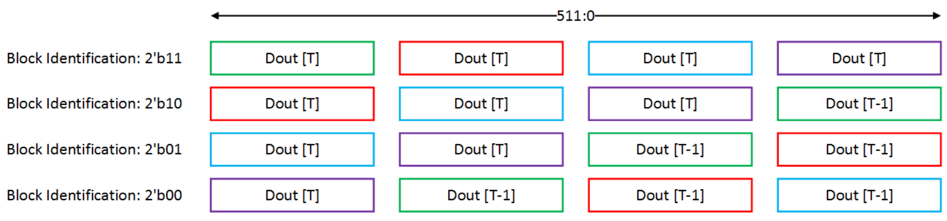


Figure 5.5: Packed Storage Methodology - split memory implementation data reorganization

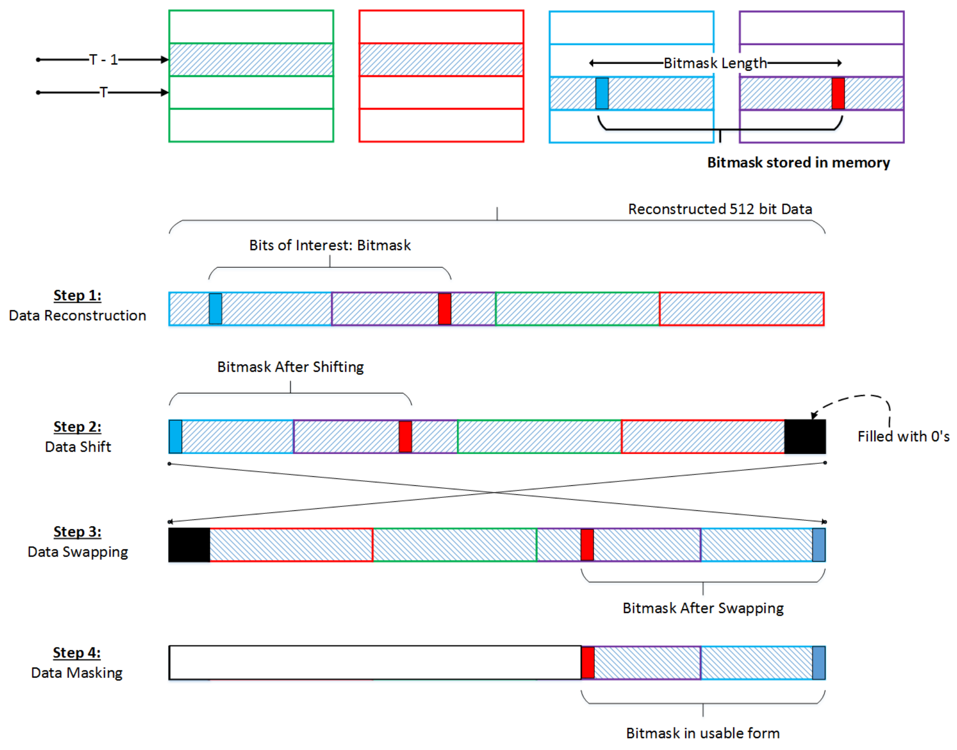


Figure 5.6: Extracting the bitmask from the data fetched from the memory

- **Step 1 - Data Reconstruction** In this step, the data which is fetched from the individual memory blocks are reconstructed based on the block identification bits as explained previously in Figure 5.5. All the combinations of the aggregated data are prepared, amongst which the specific combination is chosen based on the registered block identification bits through a multiplexer as shown in Figure 5.7.

- **Step 2 - Data Shift:** After the data reconstruction step, the most significant byte in the bitmask is shifted to the most significant byte position in the reconstructed data. This is done by left shifting the bitmask by a certain number of byte positions based on the value of the position bits. For example, if the position bits corresponding to the calculated address location T′ is '0', the reconstructed data is shifted by 15 byte positions. To generalize, if the byte position indicated by the position bits is 'P', the reconstructed data is left shifted by '15-P' byte positions. The data shift block is implemented in the hardware through multiplexer circuits, whose select signals are driven by the registered position bits as shown in Figure 5.7. A total of $\log_2(BW/8)$ combinations have to be processed by the multiplexers to effectively shift the data as part of the shift operation.

- **Step 3 - Data Swap:** After the data shift operation, the required bitmask information is available from the most significant byte position in the reconstructed data. However, in order to make the bitmask data usable, the required bitmask data has to start from the least significant byte position. So, in this step, the data is swapped *bit by bit*, between the most significant and the least significant bit positions. For example, the data in bit position 511 is swapped with bit position 0. Similarly, data in bit position 510 is swapped with data in bit position 1 and this process continues until all the 512 bit data is swapped. In order to support this swapping step, the bitmask is swapped in a bit by bit manner before being stored in the memory.

- **Step 4 - Data Masking:** Out of the 512 bit data that is fetched from the memory, only a certain portion of the reconstructed data contains the bitmask. The actual data of interest is defined by the bitmask length for a group and a 512-bit mask is generated remove the non-bitmask bits among the 512 bit data extracted from the memories. A decoder circuit is used to generate the mask from the bitmask length which finally ends the bitmask extraction process as shown in Figure 5.7.

5.2.3 Summary

The bitmasks are highly non-linear data structures, i.e, the width of the bitmasks generated varies depending on the characteristics of the signatures. Moreover, the bitmasks which are stored in the on-chip memories should be fetched in a single clock cycle, so that there are no memory contention issues when multiple streams are inspected by the hardware accelerator. Addressing these requirements, the Packed Storage Methodology was proposed in this section through which the bitmasks can be flexibly stored in the on-chip memories without resulting in memory wastage. Moreover, using the PSM, the bitmasks can also be fetched in a single clock cycle which satisfies all of the requirements pertaining to the bitmask storage.

The next section discusses a flexible storage architecture which is used to store the compressed state transitions in the on-chip memories.

5.3 Compressed Transition Storage

5.3.1 Requirements

As discussed in Chapter 3, the compressed state transitions generated after the MSBT are segregated into the leader transitions and the member transitions. The total number of leader and member transitions generated for each of the signature sets vary depending on the following factors:

- **Leader Transitions:** The leader transitions generated in a single leader state vary depending on the number of transitions which are compressed during the intra-state compression step. Moreover, as explained in section 5.2, the number of transitions which are compressed in a state after the intra-state compression step depends on the combination of characters which occur in the signature set. Since the leader transitions corresponding to all the leader states are consolidated and

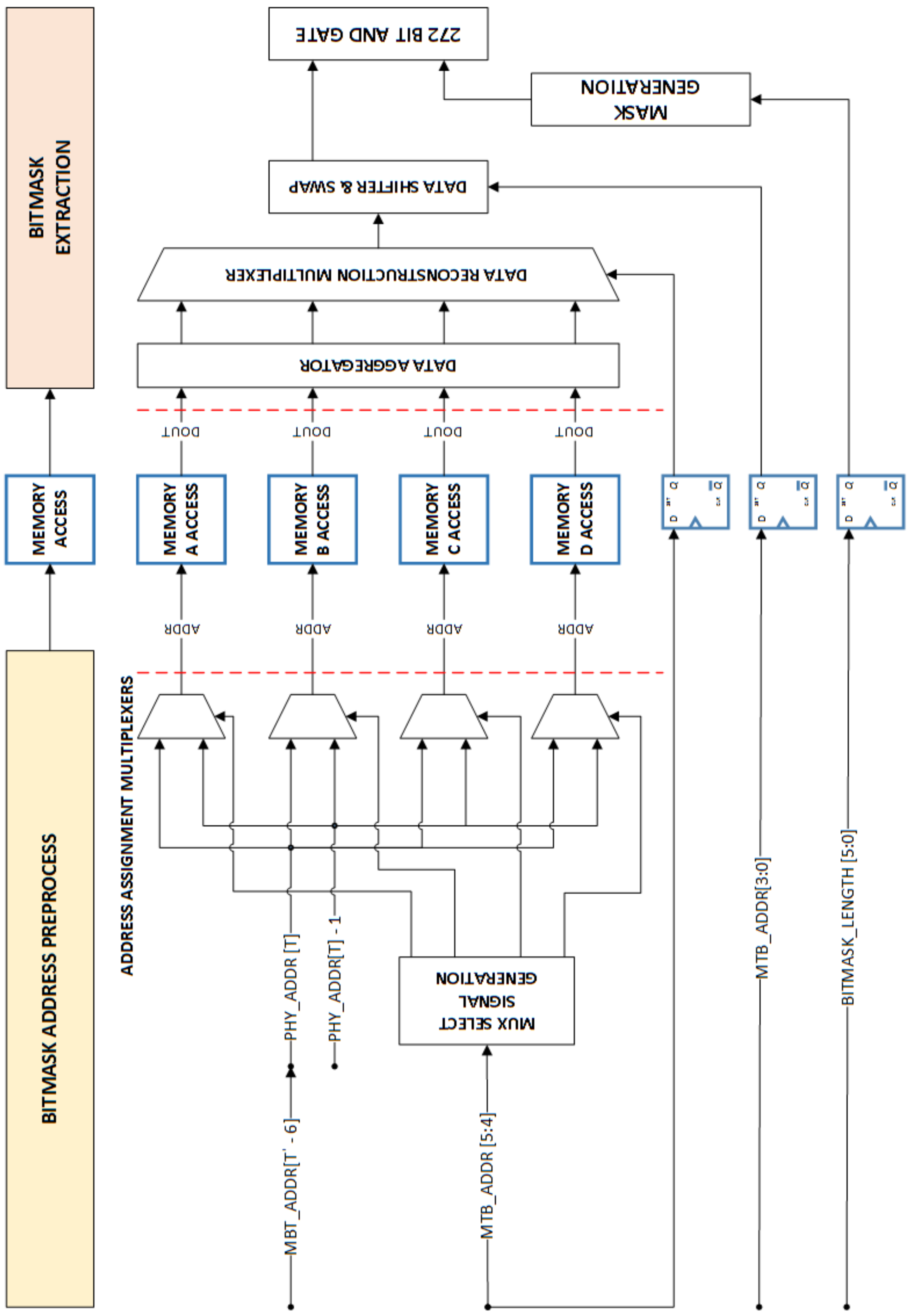


Figure 5.7: Hardware logic used to extract the bitmask from the memory

stored in the LTT, the total number of leader transitions which are stored in the memory also depends on the total number of groups generated during the state grouping step. To summarize, the total number of leader transitions generated after the transition compression varies depending on the characteristics of the signature set and the total number of groups generated as part of the state grouping step.

- **Member Transitions:** As far as the member transitions are concerned, the member transitions across all the groups are consolidated and stored in the MTT. Unlike the leader transitions, the aggregate number of member transitions only depends on the state grouping step. Though, the DC state grouping step proposed in Chapter 4 focuses on minimizing the number of member transitions through various algorithms, the actual number of member transitions generated solely depends on the state grouping.

So, the underlying transition storage architecture should be able to handle varying leader and member transition counts which depends on the characteristics of the signature set.

The simplest way to design the transition storage architecture is to architect the physical memories (LTT & MTT) in such a way that a predefined fixed number of leader and member transitions can be stored in the memories. However, due to the uncertainty in the actual number of transitions generated after the transition compression, designing the LTT and the MTT in this way will result in inefficient usage of on-chip memory space. For example, scenarios could occur in which the number of member transitions generated are higher than the predefined capacity of the MTT, while the LTT is inadequately used or vice versa. In such scenarios, using the same physical memory to store the leader and the member transitions will result in memory contention issues as the LTT and the MTT are accessed in different stages of the transition decompression process[4]. In order to work around the memory contention issues, a memory arbiter will have to be used to manage the memory accesses which will introduce additional latency in the decompression process. So, the storage architecture which is used to store the compressed state transitions should be capable of addressing the following requirements:

- The underlying transition storage architecture should be flexible and programmable so that a varying number of leader and member transitions can be stored in the on-chip memories in a configurable manner.
- The storage architecture should not introduce additional complexity to the decompression engine architecture, so that the latency to fetch the compressed transition increases.

Addressing these requirements, a flexible; run time programmable *Shared Memory Methodology* (SMD) is proposed to store the compressed transitions. The details of the storage architecture is explained further.

[4]It should be remembered from Chapter 3 that the LTT is accessed during the LTBFS, which is the second stage while the MTT is accessed in the MFS, which is the third stage.

5.3.2 Shared Memory Methodology

Figure 5.8 shows the organization of the state transitions which allows a configurable storage of leader and member transitions in the on-chip memories through the SMD. The on-chip memories are logically split into three partitions, i.e., the LTT, the MTT and the Shared Memory (SM). The LTT and the MTT are *dedicated* physical blocks of memory[5], which only store the leader and the member transitions, respectively. On the other hand, the SM consists of multiple blocks of physical memories, where each of the individual memories can be configured to either store the leader or the member transitions. The LTT, the MTT and the memories in the SM store one compressed state transition per address location. The LTT and the MTT are architected to store L leader transitions and M member transitions, respectively. As seen in Figure 5.8, the SM is organized into R physical memory blocks, where each of the blocks can store S compressed state transitions. The individual physical memory blocks belonging to the shared memories can either be allocated to the LTT or the MTT regions, so that the capacity of the dedicated memories can be dynamically increased depending on the number of compressed transitions which are generated after the transition compression. The decision regarding the allocation of the individual memory blocks in the SM is made by the compiler which performs the MSBT. The choices for 'L', 'M', 'R' and 'S' are made during the implementation stages depending on the number of signatures to be supported by the signature matching engine.

Table 5.2 shows the assumptions made to further explain the shared memory methodology. Based on the assumptions described in Table 5.2, the LTT and the MTT can store a total of 8192 transitions while the SM is composed of 4 physical memory blocks each

[5]Though the LTT and the MTT are shown as single memory blocks in the illustration, in order to meet the area and timing requirements, they can be implemented as multiple physical memory blocks.

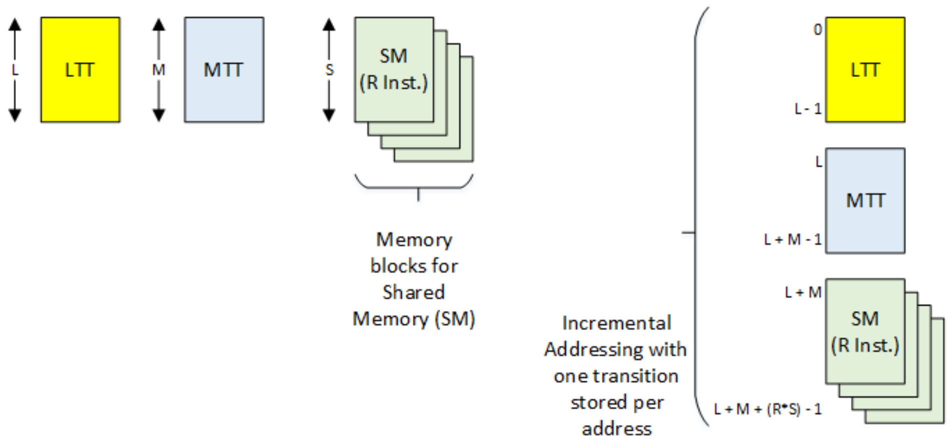

L
LTT
M
MTT
S
SM
(R Inst.)
Memory blocks for Shared Memory (SM)
Incremental Addressing with one transition stored per address
0
L - 1
LTT
L
MTT
L + M - 1
L + M
SM
(R Inst.)
L + M + (R*S) - 1

Figure 5.8: Shared memory architecture with state transitions flexibly stored in the LTT, the MTT and the shared memories

of which can store 1024 transitions. Figure 5.9 describes the overall flow of the shared memory allocation and how it is decided during the compile time. After the transition compression, it is assumed that the compiler generates 9192 leader transitions and 11192 member transitions. Since the maximum capacity of the LTT is only 8192 transitions, the first block from the shared memory is allocated to the LTT. Similarly, since there are 11192 member transitions, the rest of the shared memory blocks are allocated to the MTT as shown in Figure 5.9. The compiler also generates the ownership bits for the shared memories after deciding the allocation of the shared memories to the LTT or the MTT. The ownership bits provide information on whether the LTT or the MTT *own* the physical block of the shared memory, further providing information on whether the physical shared memory block is used to the extend the capacity of the LTT or the MTT. In this way, by introducing the ownership bits, the issue associated with the memory contention in accessing the shared memory by the hardware block is eliminated. As shown in Figure 5.9, the SM ownership bits are set to 0, 1, 1 and 1, respectively which inform the hardware accelerator that the SM block 0 belongs to LTT, while the SM blocks 1, 2 and 3 belong to the MTT.

Table 5.2: Example values used for the parameters to explain the Shared Memory Methodology

LTT	L	8192
MTT	M	8192
SM	R	4
	S	1024

The individual physical blocks of memory are incrementally addressed starting with the LTT followed by the MTT and the shared memory as shown in Figure 5.8. Table 5.3 describes the address ranges for the LTT, the MTT and the SM entries, assuming that the memory parameters are assigned as shown in Table 5.2. This information is essential as the address ranges together with the ownership bits determine the actual physical memory location which is accessed to fetch the compressed state transition.

5.3.2.1 Transition Memory Access

Figure 5.10 shows the organization of the physical memory blocks and how the memories are accessed without any contention in a single clock cycle as part of the SMD. The LTT, the MTT and the SM are organized in the *physical memory blocks* and are individually accessible through a dedicated SRAM interface. The *transition memory control* block connects the memories to the hardware accelerator block, which accesses the memories as part of the decompression process. The transition memory control block provides a dedicated SRAM interface to access the leader transitions and the member transitions alone and is called the leader transition and the member transition access interfaces. Based on the incoming address of the memory transaction, the leader and the member control block can differentiate if the incoming transaction is directed to the LTT, the MTT or the shared memory blocks.

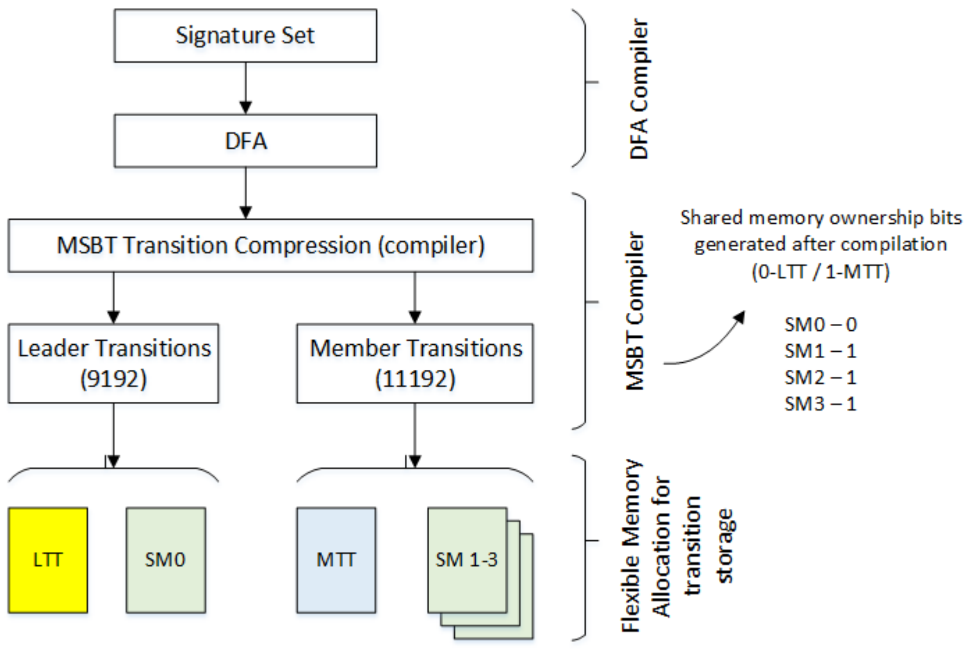

Signature Set
DFA
MSBT Transition Compression (compiler)
Leader Transitions (9192)
Member Transitions (11192)
LTT
SM0
MTT
SM 1-3
DFA Compiler
MSBT Compiler
Flexible Memory Allocation for transition storage
Shared memory ownership bits generated after compilation (0-LTT / 1-MTT)
SM0 – 0
SM1 – 1
SM2 – 1
SM3 – 1

Figure 5.9: Overview of memory block allocation during compile time

Table 5.3: Shared memory architecture - memory addressing

Memory	Start Address	End Address
LTT	0x0000	0x1FFF
MTT	0x2000	0x3FFF
SM0	0x4000	0x43FF
SM1	0x4400	0x47FF
SM2	0x4800	0x4BFF
SM3	0x4C00	0x4FFF

There are 2 levels of demultiplexers which classify a memory transaction. The first demultiplexer (L1 Demux) either directs the memory transaction towards the dedicated memory (LTT/MTT) or the SM blocks. The second level of the demultiplexer (L2 Demux) is only used, if a transaction is directed to the SM blocks and selects the individual block to which the transaction is directed to. The select signal of the demultiplexers are generated based on the incoming address of the transaction falling into a specific memory range as shown in Table 5.3.

Once a transaction is directed to the shared memories through the 2-level demultiplexing process, either the transaction generated by the leader transition or the member transition access interface is selectively directed towards the physical memory. Since a transaction from both the leader and the member transaction access interfaces can be

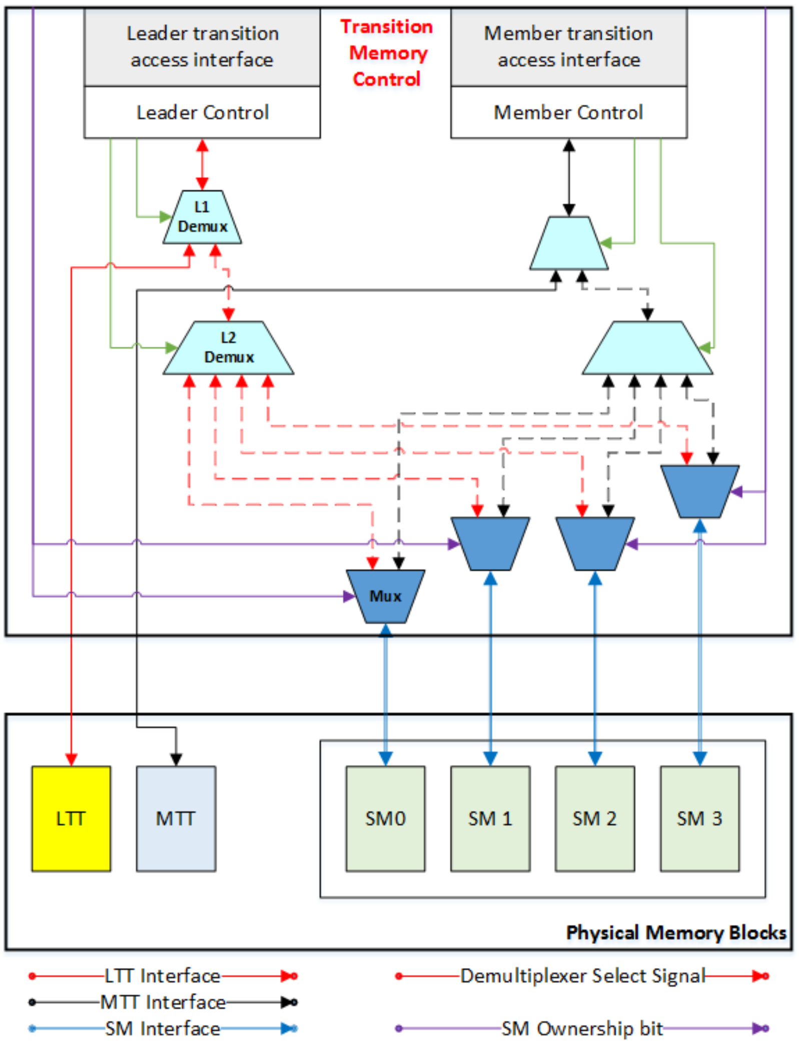


Figure 5.10: Overview of the transition memory access using the shared memory methodology

directed towards the SM blocks, a multiplexer (Mux) is used to direct the request from one of the access interfaces to the SM blocks. The select signal of the multiplexer is controlled through the individual ownership bits which were generated during the compilation process. The ownership bits are stored in the accelerator as internal registers and are appropriately connected to the corresponding multiplexer's select signal. So, the multiplexer logic makes sure that only one of the transition interfaces access the physical memory. So, a scenario can never occur where both the leader and the member transition interfaces access the same physical memory among the shared memory blocks.

5.3.3 Discussion & Summary

The actual number of leader and member transitions which are generated after the transition compression is variable and are highly dependent on the characteristics of the signature set. The Shared Memory Methodology was proposed in this section to effectively store the varying number of compressed state transitions in a flexible and a configurable way. The SMD proposes a flexible and programmable methodology to store the compressed transitions by introducing the concept of shared memories to dynamically increase the capacity of the dedicated memory blocks, which store the leader and member transitions, respectively. Moreover, the proposed architecture prevents any memory contention issues which can potentially introduce additional latency in the transition decompression process.

Though the SMD allows a configurable and flexible storage of the compressed transitions, an additional latency is introduced in the transition fetch due to the demultiplexers and the multiplexers (combinatorial logic) introduced in the memory access path. This additional latency, varies depending on the partitioning of the physical memories. The additional latency which is introduced can be classified into a fixed component and a variable component. The fixed latency component includes the latency of the first level demultiplexing operation and the last level multiplexing operation. On the other hand, the variable component depends on the second level demultiplexing which in turn depends on the number of physical memory blocks in the SM. The second level demultiplexing latency corresponds to $\log_2(R)$ demultiplexer latency and should be considered when designing the memory architecture for the transition memories. The choice for 'R' and 'S' affects the latency introduced in the memory access and the flexibility achieved in the transition storage.

A shared memory region partitioned into small physical blocks results in a higher degree of flexibility in the transition storage. However, this also introduces a higher latency than a shared memory region partitioned into bigger physical blocks with a smaller 'R'. Since the additional latency that is introduced is due to the latency of the combinatorial logic, this only affects the achievable clock frequency making sure that the intended data is fetched in a single clock cycle. With shrinking technology nodes, the additional latency will also be minimal and will not greatly affect the frequency of operation of the signature matching hardware accelerator.

5.4 BiSME - Internal Architecture

After defining the flexible storage architectures to store the bitmasks and the compressed state transitions, this section describes the detailed internal architecture of BiSME, a Bitmap based Signature Matching Engine proposed to accelerate the signature matching operation.

The internal architecture of the BiSME is shown in Figure 5.11(a) and consists of 4 major blocks, namely the Memory Shell (MS), the Address Decoder (AD), the Memory Access Multiplexer (MAM) and the Decompression Engine (DE). The BiSME consists of a combination of various proprietary and non-proprietary interfaces which are used to

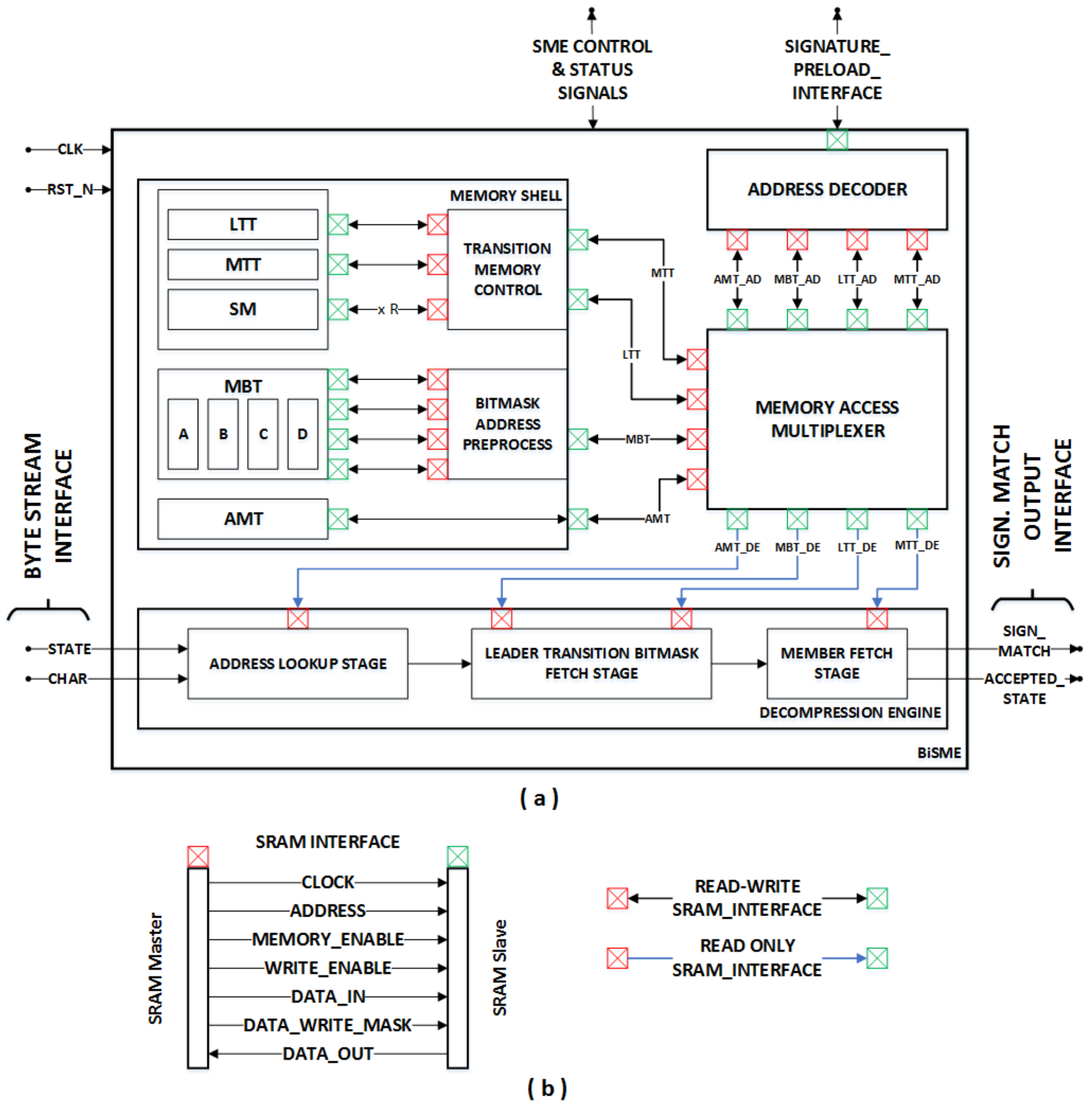


Figure 5.11: (a) Internal Architecture of the Signature Matching Engine (b) SRAM interface description

communicate with the hardware blocks. The *signature preload interface* is used to download the signatures into the memory shell, which consists of multiple on-chip SRAMs that store the compressed DFA. So, the SRAM interface shown in Figure 5.11(b) is used as the interface of communication for the signature preload interface. On the other hand, the *byte stream interface* and the *signature match output interface* are proprietary interfaces, which are used to input the payload bytes for inspection and send appropriate information related to a signature match, respectively. The SME control and status signals are a collection of signals which are fed from the register bits programmed by the network processor. Moreover, it also consists of signals which provide the status information about BiSME to the network processor.

5.4.1 Memory Shell

Before describing the details with respect to the individual memories in the memory shell, a short description on the state encoding is described first. Since the states in the DFA are organized into the leader and the member states after the MSBT, a state transition is also encoded as a combination of the leaderID and the memberID as shown in Figure 5.12(a). The leaderID identifies the group to which a state belongs to, while the memberID identifies each of the individual states in a group. 8 bits are used to represent the leaderID and the memberID to support a theoretical maximum of 64k states spread across a maximum of 256 groups with each group consisting of a maximum of 256 states. Finally, the most significant bit in the encoded state representation is called as the *'Signature Match Bit'* which recognizes an accepting state.

The exact_match signature set was used as the reference to architect the various memories in the memory shell, so that a minimum of 500 string signatures can be stored in the signature matching engine. The transition and the control memories are architected to store the compressed DFA corresponding to a maximum of 16k states. As seen in Chapter 4, the experimental evaluations on the DC state grouping method identified that the average number of states generated in a group after the state grouping step is always smaller than the allowed predefined maximum number of states, B. Moreover, the individual number of states clustered into a group also depends on the characteristics of the signature set. Considering these issues, even though a maximum of 64k states can be represented using the encoded state representation, the memories are architected to store the state transitions for a maximum of 16k states only, supporting transition compression rates of 98%[6]. However, a compressed DFA with a higher state count can be stored, if it can fit into the predefined memory boundaries. This corresponds to a scenario where the transition compression rate achieved by the MSBT in the signature set of interest is higher than 98%.

The memory shell primarily stores the compressed DFA in various on-chip SRAMs as shown in Figure 5.12(b-d). Based on the shared memory methodology, the transition memories are composed of the leader transition table, the member transition table and the shared memories as shown in Figure 5.12(b). A cumulative total of 96k[7] compressed transitions can be stored in a configurable way within the transition memories. The leader and the member transition tables respectively are architected to store a maximum of 16k and 48k transitions, respectively, while the shared memory consists of 4 blocks, with each block capable of storing 8k state transitions. Since each of the group only has a single leader state, a maximum of 64 leader transitions can be stored in the leader

[6]The choice of 98% for the transition compression rates were made after the experimental results seen in Chapter 4

[7]The total number of state transitions generated from the uncompressed DFA with 16k states is 4096k. 2% of the uncompressed transitions is $\sim$84k transitions. So, the memories are capable of storing slightly over 2% of the overall transitions in the DFA corresponding to 16k states.

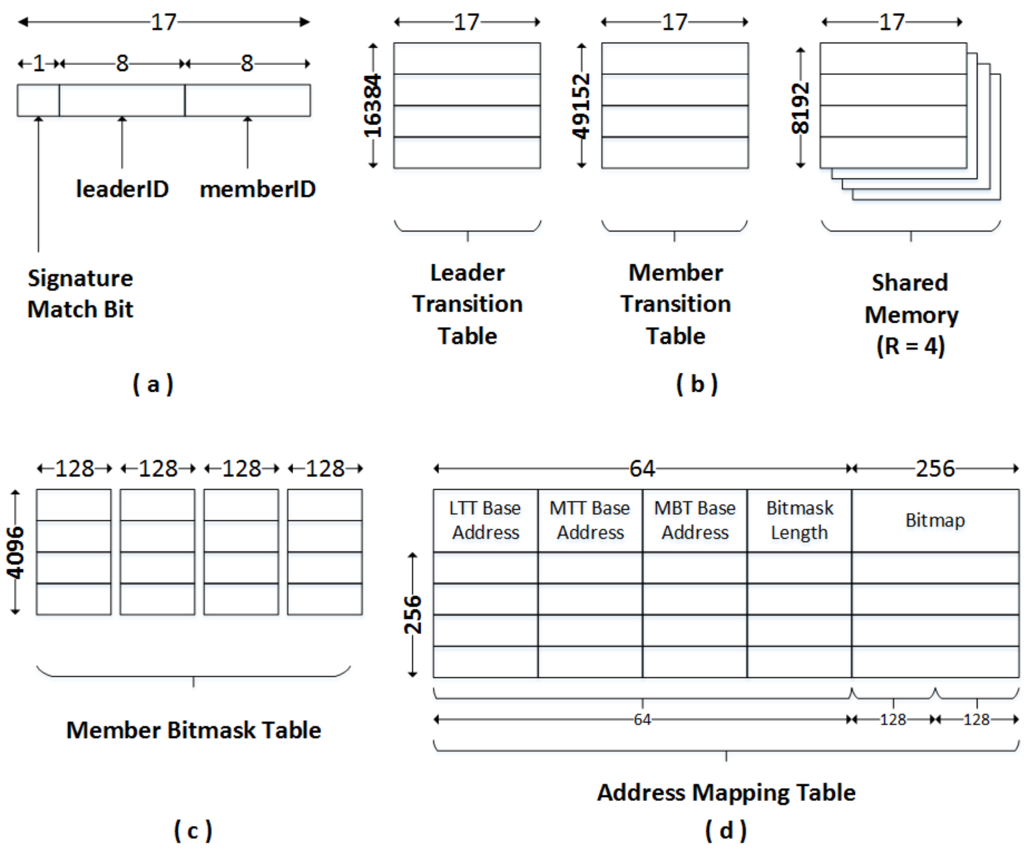


Figure 5.12: (a) DFA state encoding (b) Organization of the compressed transitions in the memory (c) Table to store the MTB and the cumulative sum of transitions (d) Table to store the base addresses and other control information

transition table corresponding to a maximum of 256 leader states[8]. If more leader transitions are generated, the memory blocks in the shared memories can be used to extend the capacity of the leader transition table. The *transition memory control* block, described in the previous section is also a part of the memory shell and performs the logic operations pertaining to address assignment to the individual physical memories.

The MBT is designed using the packed storage methodology and is capable of storing the bitmasks for a maximum of ∼16k states, assuming that each member state has an

[8]The average number of leader state transitions generated across various signatures sets used in the MSBT evaluation, varied between 60-80. So, it was assumed to use an average value of 64 leader transitions per state to define the boundaries of the LTT.

average bitmask width of 128 bits[9]. Furthermore, based on the previous discussions on the composition of the characters in the signature set, it should be remembered that not all characters are always represented in a signature set. Moreover, only a very small portion of the extended ASCII codes (128-255) are found in real signature sets. So, based on these observations and the experimental results seen in Figure 5.2, the average width of the bitmask to be stored for each member state is set to 128 bits. Though, this assumption is just used to define the boundary for the MBT, the bitmasks corresponding to an increased number of member states can be stored if the average bitmask width in the member states is smaller than the original assumption. The MBT is split across 4 physical blocks whose memory width is set to 128 bits to make sure that the 272 bit longest bitmask can be fetched in a single clock cycle. The *bitmask address preprocess* block performs logic operations associated with the address assignment to the individual memories in the PSM and also generates the consolidated 512-bit data from the memories.

Figure 5.12(d) shows the Address Mapping Table (AMT) which stores the control information for the 256 groups. The AMT stores the address location of the first leader transition and the first member transition in the group which are referred to as the LTT and the MTT base address, respectively. Similarly, the MBT base address refers to the least significant byte of the first bitmask in each of the groups (i.e., locations referred by the black arrows in Figure 5.3). Together with the base addresses, the bitmap and the Bitmask Length (BL) in bytes are also stored in the AMT. The bitmap and the BL are common for all the states in the group and are stored only once in the AMT. Since, a total of 320 bits of information is stored per group, the AMT is split width-wise and physically implemented as three separate memories to minimize the SRAM access latencies.

5.4.2 Address Decoder & Memory Access Multiplexer

The address decoder is the gateway to access the memories through an external agent, e.g., the network processor to download the compressed DFA. The address decoder block routes the incoming read or the write transactions to one of the memories, based on the transaction address falling into a specific predefined address range corresponding to the individual memories.

The memory access multiplexer regulates the access to all the individual memories, in such a way that they are accessed by the address decoder when the compressed DFA is downloaded and the decompression engine during the decompression operation. In this way, the signature download process is decoupled from the signature matching operation

[9]The 128 bits chosen for the bitmask, includes the cumulative sum of transitions which utilizes a fixed 16 bits. It should be remembered that the MBT was made byte addressable, which requires the bitmask width to be a multiple of 8 bits. Additional 0's are padded to the MBT to make sure that the MBT is a multiple of 8 bits. Similarly, the cumulative of sum of transitions was allocated 16 bits as part of the bitmask. The number of member transitions generated in certain groups were over 256 in the experimental evaluation of the MSBT and this required the cumulative sum of transitions to be set to 16 bits. This would allow a maximum of 2^{16} transitions to be offset from the base address which is more than enough for the considered configuration.

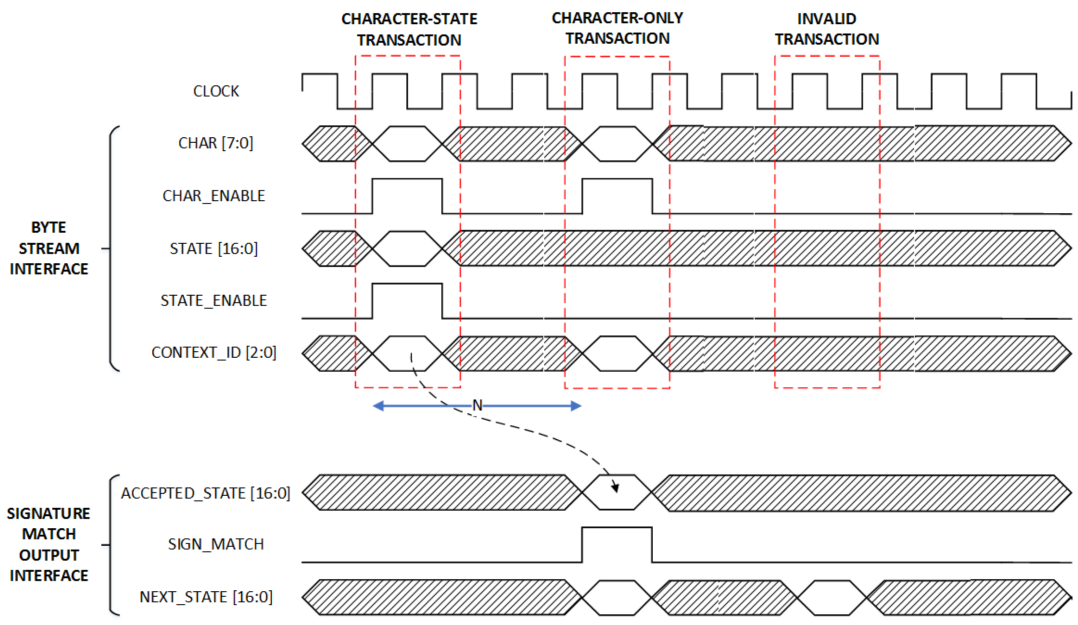


Figure 5.13: Details of the byte stream interface and signature match output interface

and also ensures that the content of the memories are not modified when the signature matching is ongoing. The multiplexers in the memory access multiplexer block and the ownership bits are controlled through the registers which are programmed by the network processor. The *SME control signals* connects these signals from a register bank which is external to BiSME.

5.4.3 Decompression Engine

The Decompression Engine (DE) is the heart of BiSME and performs the compressed transition lookup corresponding to a state-character combination. The DE receives the current state information and the payload byte through the *Byte Stream Interface* and performs various calculations in the hardware to fetch the compressed state transition. After identifying the compressed state transition, the DE checks if there is a signature match corresponding to the processed sequence of payload bytes. If a signature match is detected, the DE generates the required information to initiate the post-processing corresponding to the signature match through the *Signature Match Output Interface.*

The byte stream interface is a custom interface that is used to inject the payload bytes into the BiSME for signature matching. The byte stream interface is a collection of 5 signals as shown in Figure 5.13. The clock signal represented in the byte stream interface is the clock signal which is used to synchronize all the sequential circuits and the memories in BiSME. The byte stream interface consists of the *char*, *state* and the *context_id* signals, which represent the ‘payload byte’, the ‘current_state’ and the ‘context_id’ for which the compressed state transition is fetched. A single bit enable signal is provided for the char and the state signals to assert their validity in a transaction. It is mandatory for the char signal to always have the char_enable signal asserted during a transaction, while it is optional in the case of the state signal. It takes multiple clocks cycles[10] (N) as shown in Figure 5.13 to identify the compressed state transition corresponding to a state character combination. In order to keep the hardware pipeline occupied, the payload bytes corresponding to different 5-tuple streams are interleaved and processed by the decompression engine. So, in a given time, the payload bytes corresponding to 'N' different streams are processed in parallel by the hardware. Each of these 'N' streams are individually identified using a specific context_id. So, the context_id is also provided as an input along with the character-state combination. Since, it takes 'N' clock cycles to identify the compressed transition, it is mandatory to input the subsequent character corresponding to a stream after 'N' clock cycles to enable the correct operation of the signature matching function.

As part of the byte stream interface, only a certain valid signal combinations are accepted by the decompression engine and are described below:

- *Character-state transaction:* In this transaction, both the payload byte and the current state information are input to the engine for which the corresponding compressed state transition is identified. This transaction is used to input the root state into the decompression engine when the signature matching corresponding to a new 5-tuple stream starts. Similarly, this transaction is also used to communicate the state information to the decompression engine when the signature matching function is resumed corresponding to a specific 5-tuple network stream. In this specific transaction type, the char_enable and the state_enable signals must be asserted to inform the validity of the incoming character state combination.

- *Character-only transaction:* If the stream of bytes corresponding to the context_id are continuously input to the engine every 'N' clock cycles, the state information is not mandatory to be input to the decompression engine. The current_state information in this case is directly used from the next_state that was generated in the previous comparison cycle. In this case, the enable signal associated with the state is set to 0. However, the char_enable signal should be asserted to inform the validity of the incoming character in this transaction.

- *Invalid transaction:* This is the third scenario where there is no valid character that is input to the decompression engine. So, in this scenario, the enable signal corresponding to both the state and the payload byte are set to logic low to identify an invalid transaction. The decompression engine doesn't consume a byte during an invalid transaction.

Once the compressed state transition is identified, the identified state transition is assigned as the next_state. Then, the relevant information corresponding to the sig-

[10]As described in Chapter 3, the hardware accelerator requires a minimum of 3 clock cycles to identify the compressed state transition. However, the value of N depends on the additional number of pipeline stages added to improve the achievable clock frequency in the design and will be discussed further in this chapter.

nature match are output through the signature match output interface as shown in Figure 5.13. The signature match output interface consists of a total of three signals which are described further. The next_state signal contains the compressed state transition corresponding to the state-character combination. If the next_state is an accepting state, the corresponding next_state information is driven in the accepted_state signal along with a sign_match signal set to logic high for a single clock cycle.

The processing associated with the transition decompression is very simple and is performed across 3 hardware blocks as described below.

Address Lookup Stage: The character and the state information are received through the byte stream interface in the address lookup stage. Then the current state information is first decoded into its leaderID and the memberID. Then, the leaderID is used as the address to fetch the control data from the address mapping table. The data is fetched from the address mapping table through a dedicated SRAM interface which connects the address lookup stage to the memory shell.

The base addresses corresponding to the transition memories, bitmask memory together with the bitmap and the bitmask length is fetched from the address mapping table. After the data is fetched from the memories, the address of the leader transition and the most significant byte address of the bitmask are calculated through separate hardware blocks simultaneously. (5.4) shows the computation performed to calculate the address of the leader state transition (LTT_Addr.) corresponding to the incoming state character combination. The base address (LTT_BA) which is fetched from the address mapping table is added to an offset address which is calculated by performing the population count operation on the bitmap. According to previous bitmap based implementations [4, 5], an efficient implementation of the population count function in the hardware enables to achieve the decompression in low latencies. In order to achieve this task, the accumulative parallel adder [91] circuit is used to perform the population count operation[11]. Though, various implementations have been proposed to perform the population count operation [91, 92, 93, 94], it has been shown that the usage of accumulative parallel adder is the most efficient method to implement the population count operation [95].

The calculation performed to generate the MBT address (MBT_Addr.) is shown in (5.5). The bitmask base address (MBT_BA) fetched from the address mapping table is added to the product of the memberID and the bitmask length. The multiplication in the address generation process is implemented through the wallace tree multiplier circuits [96] to enable low latency processing. As discussed during the packed storage methodology description, the base address refers to the location of the least significant byte of the first bitmask in the group. Once the addresses are calculated, they are passed as internal signals to the next stage, i.e., the leader transition and the bitmask fetch stage.

$$\text{LTT\_Addr.} = \text{LTT\_BA} + \text{popcount (bitmap)} \tag{5.4}$$

$$\text{MBT\_Addr.} = \text{MBT\_BA} + (\text{memberID} \times \text{BL}) - 1 \tag{5.5}$$

[11]Appendix A describes the architecture of the accumulative parallel adders

$$\text{MTT\_Addr.} = \text{MTT\_BA} + \text{Cum.\_Sum} + \text{Popcount(MTB)} \tag{5.6}$$

Leader Transition Bitmask Fetch Stage: In this stage, the leader transition and the bitmask are fetched simultaneously from the respective memories using the dedicated SRAM interfaces. The address locations which were calculated in the previous stage are used to fetch the data from the memories. After the memory access, the data returned from the leader transition table directly corresponds to the leader transition. On the other hand, the *bitmask extraction* is performed in this stage to extract the MTB and the cumulative sum of transitions. The MTB bit corresponding to the incoming payload byte is fetched by using a multiplexer circuit in this stage. Simultaneously, the location of the transition to be fetched from the member transition table is calculated as shown in (5.6). The member transition base address (MTT_BA) fetched previously is added to two different offsets. The first offset, i.e., the cumulative sum of transitions provides information about the number of member transitions stored in the member transition table prior to the current member state of interest. The population count operation performed on the MTB provides the member offset. The population count operation in this stage is also performed using the accumulative parallel adder circuitry.

Member Fetch Stage: This is the last stage of calculations performed in the decompression engine. In this stage, the next_state is assigned either from the registered leader transition or the member transition fetched from the MTT. If the current_state is a leader state, then the registered leader transition is assigned as the next_state. If the current_state is a member state, the next_state is either assigned from the leader transition or the member transition depending on the MTB bit corresponding to the character. After identifying the next_state, the 'signature match bit' corresponding to the next state is checked and the results are sent to the signature match output interface accordingly.

5.4.4 BOBiSME - Modified BiSME to support Bitmask Compression

This section describes the architectural modifications which are made in BiSME to support the bitmask compression. The signature matching engine which is capable of supporting bitmask compression is called as the Bitmask Optimized BiSME (BOBiSME). Various modifications are made to the internal architecture of BiSME to accommodate bitmask compression. The modifications are made to the internal organization of the physical memories in the memory shell, the address decoder and the decompression engine. However, there are no modifications required for the external interfaces which communicate with the signature matching engine. The block level architecture of the BiSME and the BOBiSME are identical and due to this reason, the block level description of BOBiSME is not discussed in detail in this section.

Figure 5.14 shows the modified organization of the memory shell. As seen in Figure 5.14, there are two modifications which are made to the memory shell to support bitmask compression. Firstly, the unique_bitmask is stored in the address mapping table in addition to the other data which are originally stored in it. Since a maximum of 256 member states are permitted per group, a corresponding 256-bit unique_bitmask is required for each of the groups. This brings the information stored in the address mapping

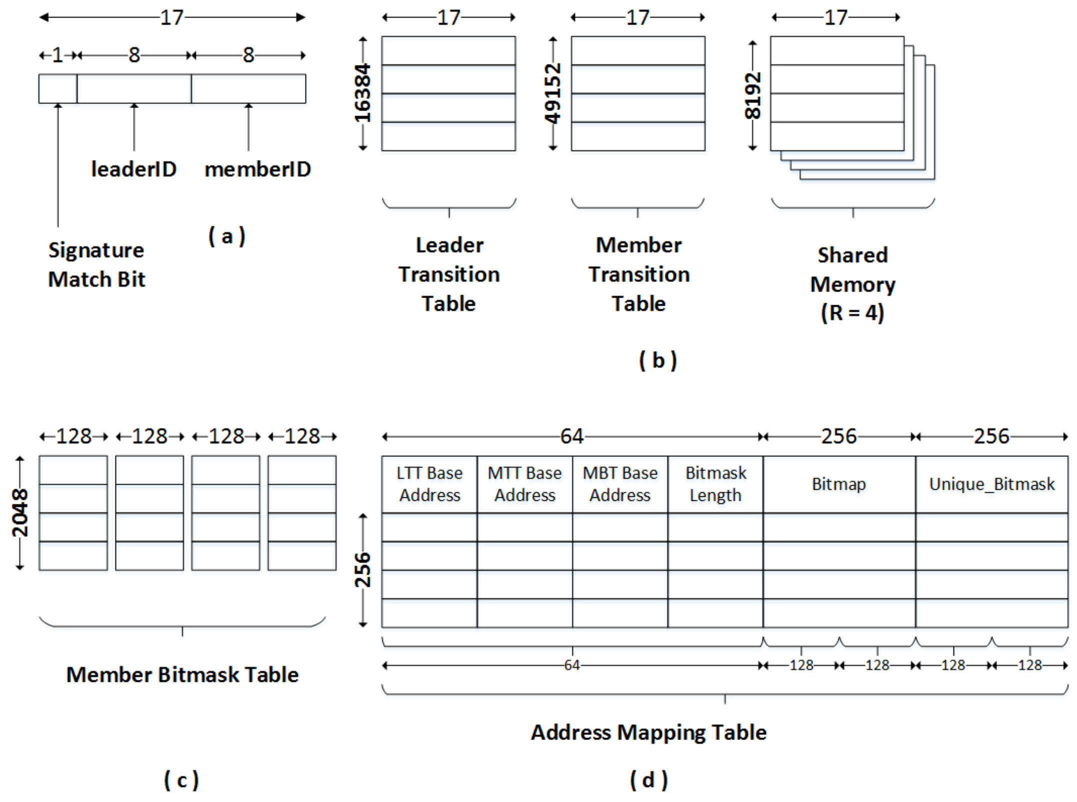


Figure 5.14: (a) DFA state encoding (b) Organization of the compressed transitions in the memory (c) Member Bitmask Table to store the MTB and the cumulative sum of transitions (d) Table to store the base addresses and other control information

table to 576 bits per group. As in the case of the BiSME, the AMT is split width-wise and stored across five different physical memories to reduce the latency associated with the memory fetch. Secondly, the member bitmask table which stores the bitmasks for the member states is reduced by half. Based on the experimental evaluation performed in Chapter 4, it was identified that about ∼50-70% of the bitmasks can be compressed after the bitmask compression. Based on the experimental results, the MBT is reduced to half of its original size in BOBiSME. Apart from these two modifications, no further changes are made to the memory shell in BOBiSME. The organization of the transition memories in BOBiSME is the same as in the case of BiSME, as the bitmask compression primarily focuses on compressing the bitmasks and not the state transitions.

In addition to the changes implemented to the memory shell, additional hardware logic is added to the decompression engine in the BOBiSME. Firstly, the bitmask offset is computed in the ‘address lookup stage’, which is used to compute the location of the most significant byte of the bitmask. In the case of BiSME, the memberID was multiplied with the bitmask length and added to the bitmask base address to compute the location of the most significant byte of the bitmask. In the case of BOBiSME, the bitmask offset is used instead of the memberID to compute the same. The bitmask offset is computed by performing the population count operation on the unique_bitmask. Secondly, the additional offset[12] is computed in the 'leader transition bitmask fetch stage' which is required to compute the location of the compressed state transition in the member state corresponding to the state character combination. To summarize, the introduction of bitmask compression introduces additional logic circuits in the decompression engine in the first and second stages of the decompression engine to reduce the memory used to store the compressed DFA in BOBiSME.

5.4.5 Throughput

If T, F and B represent the overall throughput achieved by the signature matching engine, the frequency of operation and the number of bytes inspected per clock cycle, respectively; then their relationship is represented through (5.7).

$$\mathrm{T} = \mathrm{F} \times \mathrm{B} \times 8 \text{ bits/s} \tag{5.7}$$

Assuming that all the memories and the sequential logic is clocked using the same clock signal in BiSME and BOBiSME, the transition decompression process requires a minimum of three clock cycles to identify the compressed state transition corresponding to a state character combination. The various required internal signals in the decompression engine are registered to maintain the integrity of the hardware pipeline. Since it takes three (N=3) clock cycles to identify the compressed state transition, it would take a minimum of three clock cycles for BiSME/BOBiSME to consume the subsequent byte in the sequence of the bytes corresponding to a single 5-tuple stream. In such a scenario, in order to fully utilize the hardware pipeline, the payload bytes from multiple (N) contexts are input to BiSME in an interleaved fashion, as proposed in [97]. This allows to initiate a compressed transition fetch for a single byte every clock cycle and keep the hardware pipeline busy. Fundamentally, a context refers to a unique 5-tuple network stream whose payload is being inspected.

The maximum throughput achieved by each individual context, $\mathrm{T_c}$ is the ratio of the overall throughput T and N as shown in (5.8), where N represents the number of clock cycles required to identify the compressed state transition corresponding to a single byte.

$$\mathrm{T_c} = \frac{\mathrm{T}}{\mathrm{N}} \tag{5.8}$$

The signature matching throughput, T can either be increased by increasing the frequency of operation, F or by increasing the number of bytes inspected per clock cycle, B. In order to increase the number of bytes inspected per clock cycle, the signature matching operation has to be performed against a corresponding multi-stride DFA[13].

[12]The details related to the additional offset computation was explained in Chapter 4 and is not explained here again.

[13]For example, if 2 bytes are inspected per clock cycle, the corresponding 2-stride DFA will have state transitions corresponding to $2^{2 \times 8}$ characters per state.

The process of converting a DFA to a multi-stride DFA results in an exponential explosion in the number of state transitions generated and is not a feasible option to improve the throughput [98]. So, the only mechanism to improve the throughput is to scale the operating frequency F, by effectively pipelining the decompression engine. In order to achieve signature matching rates of 10 Gbps, the BiSME and BOBiSME engines have to be clocked at 1.25 GHz. In order to achieve such high frequencies, additional registers were added to the processing stages to achieve higher frequencies.

Figure 5.15 shows an overview of the context based pipelining process. Figure 5.15(a) and (b) respectively show the total number of pipeline stages in BiSME and BOBiSME. In the case of BiSME, in order to achieve the intended signature matching throughput, the processing associated with the address lookup stage is split into two pipeline stages and the processing associated with the leader transition bitmask fetch stage is split into three pipeline stages. Thus, it would require a total of 6 clock cycles (N=6) to identify the compressed state transition in the case of BiSME. In the case of BOBiSME, due to the introduction of the processing associated with bitmask compression, an additional pipeline stage is required for both the address lookup stage and the leader transition bitmask fetch stage. This brings the total number of pipeline stages to 8 in BOBiSME. Figure 5.15(c) shows the context based payload interleaving to keep the hardware pipeline busy in the case of BiSME. As shown in the figure, the payload bytes across 6 different streams are injected into the signature matching engine every successive clock cycle.

However, the maximum clock frequency achieved also depends on the memory access latencies which in turn depends on the choice of the implemented technology node. The increase in 'N' resulting from the deeper pipeline impacts the per context throughput T_c, but improves the overall throughput achieved by BiSME and BOBiSME. The increase in the pipeline depth, also requires a corresponding increase in the number of contexts supported to keep the pipeline busy.

5.5 Deep Packet Inspection Accelerator

The signature matching engines which were described in the previous sections primarily focused on accelerating the decompression function to perform signature matching at line rates. Though, the signature preload interface in BiSME / BOBiSME can be directly used to communicate with the signature matching engines, additional hardware logic, external to the signature matching engine is required to extract the payload bytes from the network packets and sequentially inject into the engine. So, in order to complement the functionality and to improve the ease-of-use of BiSME & BOBiSME, a *Deep Packet Inspection Accelerator* (DPIA) is proposed in this section. The proposed DPIA is an integrated entity which consists of the signature matching engines and the additional hardware accelerators which manage the network payload bytes which are inspected by the engine. The following subsection details the internal blocks and the interfaces which are used to communicate with the DPIA.

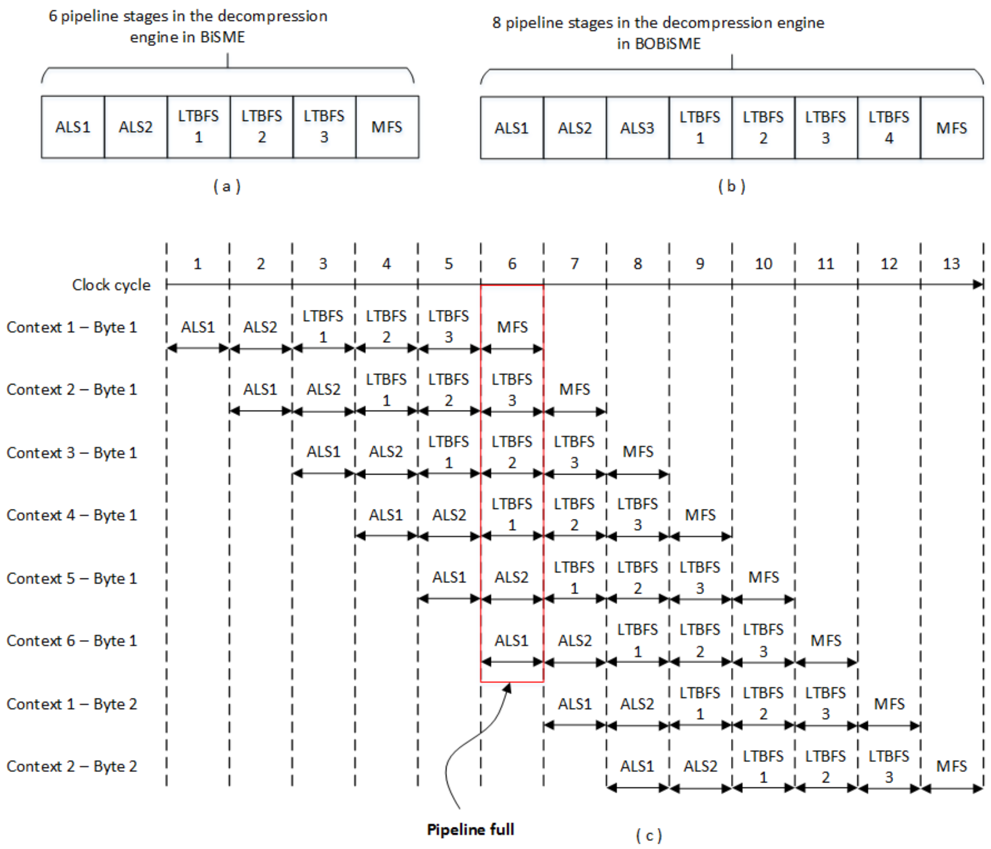


Figure 5.15: (a) ALS & LTBFS split into multiple pipeline stages in BiSME (b) ALS and LTBFS split into multiple pipeline stages in BOBiSME (c) Pipelined operation of BiSME with context based byte interleaving

5.5.1 DPIA Interfaces

Figure 5.16 shows the block level internal architecture of the DPIA. The proposed accelerator consists of two major interfaces to communicate with the signature matching engine and are called as the control and the datapath interfaces, respectively. The control and datapath interfaces will be compliant with standard on-chip network communication interfaces such as the AXI [99] or the OCP[100], to enable easy integration of the DPIA with the network-on-chip fabric in a network processor.

As shown in Figure 5.16, the control interface is used to access control and status registers which configure the functionality of the DPIA. Additionally this interface is also used to write and read back the internal memories which store the compressed signatures.

The datapath interface is used by the host processor or the DMA engines in the SoC, to send the network packets for signature matching. The post-processing associated with a signature match is proposed to be performed as a software function in the host processor.

Performing the post-processing in the software provides the flexibility to define the post-processing functions corresponding to the specific signature matches. Furthermore, this also introduces the critical software interaction with the hardware accelerator to enable efficient hardware software coordination.

5.5.2 DPIA Internal Architecture

As seen in Figure 5.16, the DPIA consists of four different internal blocks, i.e., the *Bus Control Unit* (BCU), the *Register Bank* (RB), the signature matching engine (BiSME/BOBiSME) and the *Network Data Management Engine* (NDME).

The signature matching engine in the DPIA performs the signature matching function and can either be composed of BiSME or BOBiSME.

The bus control unit receives a transaction through the control interface and either forwards the transaction to the register bank[14] or the signature_preload_interface in the signature matching engine depending on the incoming address.

The register bank is a collection of various control information which are programmed by the host processor to configure the functionality of the hardware accelerators in the DPIA. For example, the ownership bits and the memory access multiplexer configuration bits are stored in the register bank. Moreover, the register bank also stores various status information from the accelerators in the DPIA which can be accessed by the host processor to assess the status of the hardware blocks in the DPIA.

[14]SRAM interface is used for the register_program_interface.

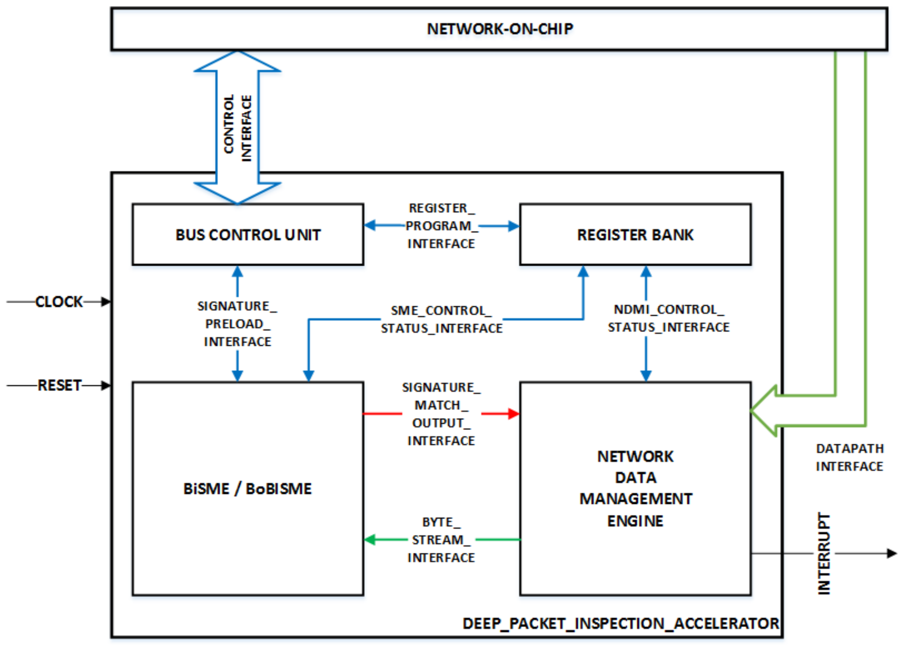

NETWORK-ON-CHIP
CONTROL INTERFACE
BUS CONTROL UNIT
REGISTER_ PROGRAM_ INTERFACE
REGISTER BANK
CLOCK
RESET
SIGNATURE_ PRELOAD_ INTERFACE
SME_CONTROL_ STATUS_INTERFACE
NDMI_CONTROL_ STATUS_INTERFACE
SIGNATURE_ MATCH_ OUTPUT_ INTERFACE
BiSME / BoBISME
NETWORK DATA MANAGEMENT ENGINE
BYTE_ STREAM_ INTERFACE
DATAPATH INTERFACE
INTERRUPT
DEEP_PACKET_INSPECTION_ACCELERATOR

Figure 5.16: Block level description of the Deep Packet Inspection Accelerator

The Network data management engine receives the network packets for DPI and extracts the payload bytes and sends them to the signature matching engines through the byte stream interface. The next subsection provides a detailed overview of the NDME.

5.5.3 Network Data Management Engine

A network processor processes a multitude of network streams in parallel and the signature matching engines should also be capable of inspecting these network streams. The signature matching engines proposed in this work (BiSME/BOBiSME) only inspect a maximum of 'N' contexts in parallel. So, multiple streams have to be mapped to the available contexts to efficiently process the network streams. Additionally the signature matching engines proposed in this work also require the extracted payload bytes to be input to them in a specific interleaved format. In order to satisfy these requirements, the network data management engine is proposed to perform the following functions:

- **Map streams to contexts:** The network streams have to be efficiently mapped into contexts in a programmable manner such that the streams are equally divided and allocated to the contexts.
- **Payload Extraction & Buffering:** The payload from the network packets have to be extracted and input to the signature matching engine's contexts in a sequential manner. In order to do this function efficiently, the NDMI should be capable of buffering the packets. However, the NDMI assumes that the packets corresponding to specific network streams are reordered appropriately before the payload extraction and buffering function.

5.5.3.1 Configurable Stream Mapping

Certain network streams should be processed with a higher priority than the other streams. On the other hand, certain streams may have strict requirements with respect to the packet processing latency. In order to make sure that all the network scenarios are addressed by the NDME, a programmable, rule based decision making engine is proposed for the stream to context mapping process. The main focus of this proposal is to not detail the rule organization or the algorithms involved in handling the packets associated with the streams, but to propose a flexible methodology to complement the functionality of the signature matching engines.

Figure 5.17 shows the description of the stream to context mapping process in which 'S' streams are mapped into 'N' contexts by the NDME. In order to configurably map the streams into contexts, the NDME maintains two tables called the stream table and the context table. The stream table consists of information about packets corresponding to different streams which have been provided to the DPIA for signature matching. The stream table also consists of information such as the pointers to the location of the packets and the various additional metadata which are extracted after the initial packet classification. Additionally, each entry in the stream table also has the state information from which the signature matching is started (or resumed). On the other hand, the

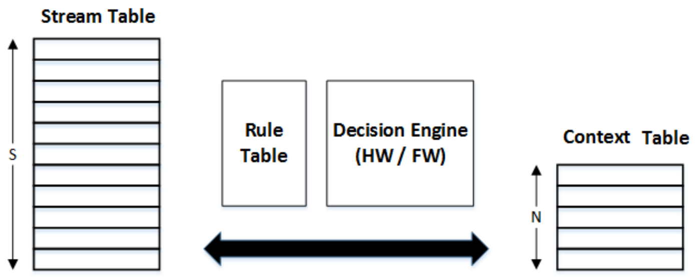


Figure 5.17: Functional description of the programmable rule based stream to context mapping

context table only maintains information about the contexts which are being inspected by the signature matching engines.

The rule table defines the set of rules through which a stream is mapped from the stream table into the context table. The additional data which are extracted after packet classification is generally used by the decision engine to decide the specific streams which are mapped to the context table. In order to make the stream to context mapping configurable, the rule table is programmed by the network processor. The decision engine makes decisions on the stream to context mapping process based on the programmed rules. The decision engine also keeps track of the number of bytes to be sent to the signature matching engine for inspection and further initiates a stream to context transfer or vice versa. The decision engine further seamlessly transfers the data between the context and the stream table to not affect the integrity of the signature matching function. Finally, the decision engine also monitors the signature matching results and accordingly initiates the specific post-processing functions corresponding to a signature match.

5.5.3.2 Postprocessing: Software-Hardware Interaction

In addition to the configurable stream mapping, the NDME is also responsible to inform the higher layer software by rising an interrupt after a signature match is found in one of the streams. After the interrupt is raised by the engine, the software takes over to identify the specific signature match after analyzing the accepted state information. As discussed in Chapter 2, of all the states in the DFA, only a small subset of states identify a signature and are called the accepting states. Once an accepting state is reached, it uniquely identifies the signature which is matched by the sequence of payload bytes. The actual number of accepting states in the DFA varies depending on the total number of signatures represented by the DFA. So, the accepting states and the actions corresponding to them are stored in the off-chip memories using a hash table as shown in Figure 5.18. The state identifier is used as the hash key to identify the actions corresponding to a signature match. Similarly, if there are hash collisions corresponding to a hash index, the accepting state information can be chained using a linked list implementation as shown in Figure 5.18.

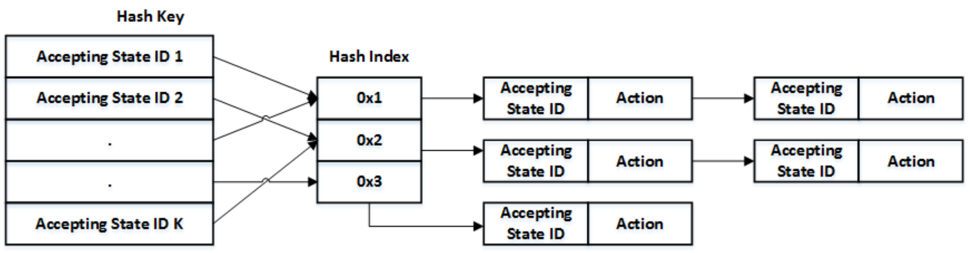


Figure 5.18: The accepting states used as the hash index to identify the postprocessing function associated with a signature match

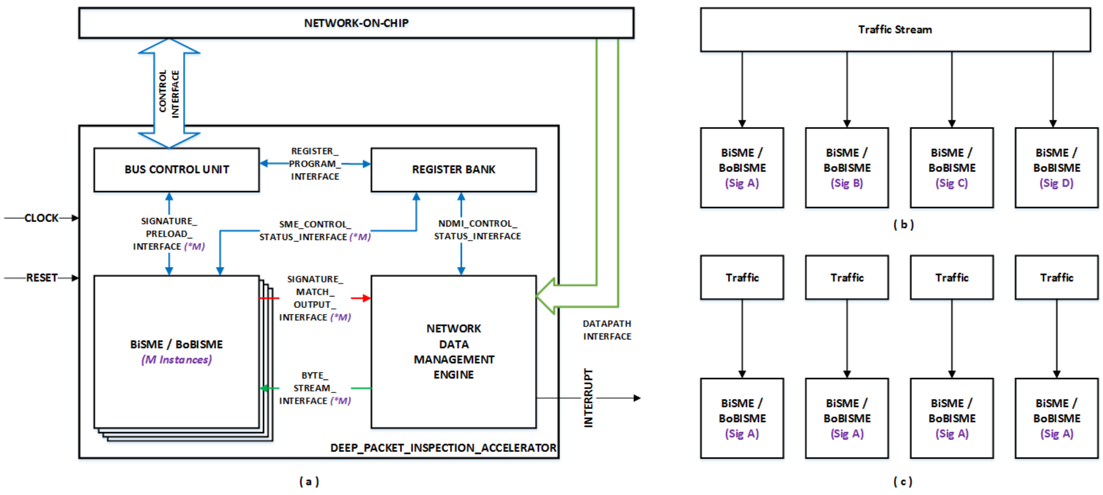

NETWORK-ON-CHIP
CONTROL INTERFACE
BUS CONTROL UNIT
REGISTER_ PROGRAM_ INTERFACE
REGISTER BANK
CLOCK
RESET
SIGNATURE_ PRELOAD_ INTERFACE (*M)
SME_CONTROL_ STATUS_INTERFACE (*M)
NDMI_CONTROL_ STATUS_INTERFACE
BiSME / BoBISME (M Instances)
SIGNATURE_ MATCH_ OUTPUT_ INTERFACE (*M)
BYTE_ STREAM_ INTERFACE (*M)
NETWORK DATA MANAGEMENT ENGINE
DATAPATH INTERFACE
INTERRUPT
DEEP_PACKET_INSPECTION_ACCELERATOR
(a)
Traffic Stream
BiSME / BoBISME (Sig A)
BiSME / BoBISME (Sig B)
BiSME / BoBISME (Sig C)
BiSME / BoBISME (Sig D)
(b)
Traffic
(c)

Figure 5.19: (a) DPIA with multiple instances of signature matching engine (b) Signature count scalability (c) Signature matching throughput scalability

5.5.4 Scalability

The growing network bandwidth and requirement to support increasing signature counts makes the scalability of the DPIA a very important topic to be addressed. As shown in Figure 5.19(a), multiple instances of the signature matching engines can be used to scale the number of signatures supported as well as to scale the achievable signature matching throughput. The associated interfaces of communication with the signature matching engines also linearly scale to support the communication with the individual signature matching engines.

The scalability associated with the signature matching throughput can be achieved by downloading the same signature set into multiple instances of the signature matching engines. In such a scenario, network traffic corresponding to different streams can be injected into different engines to linearly scale the signature matching throughput as shown in Figure 5.19(c). Similarly, as shown in Figure 5.19(b), in order to scale the number of signature sets supported by the DPIA, the compressed DFA corresponding to different signature sets are stored in the signature matching engines, to which the same sequence of payload bytes are injected. So, through the proposed DPIA, multiple signature matching engine instances can be used to configurably scale the achievable signature matching throughput as well as the supported signature counts.

One of the major disadvantages with the signature matching engines such as RegEx [37] and UAP [81] is the deterioration in the signature matching throughput as the signatures are stored in a combination of on-chip and off-chip memories. Since the signature matching is a highly memory intensive function, fetching the compressed DFA at low latencies is crucial to perform signature matching at multi gigabit rates. So, as part of the DPIA, the scalability aspect is addressed by using multiple instances of the signature matching engines. Since each of the signature matching engine has its own dedicated memories to store the compressed DFA, the memory bandwidth will not be a bottleneck to support scalable throughput and signature count. Similarly, since the signatures are effectively compressed and stored in the on-chip memories, the compressed signatures can be accessed at low latencies to support multi gigabit line rate signature matching.

5.6 Experimental Evaluation & Discussion

5.6.1 Synthesis Results

The BiSME and the BOBiSME were implemented using Verilog and were synthesized using the Synopsys Design Compiler [101] on a commercial 28nm technology library operating at 0.81V. The BiSME design was pipelined with N set to 6, to achieve an operating frequency of F=1.165 GHz. So, each BiSME instance is capable of performing signature matching at 9.3 Gbps, while the per-context throughput (T_c) that is achieved is 1.55 Gbps. On the other hand, the BOBiSME design was pipelined with N set to 8, to achieve an operating frequency of F=1.325 GHz. Consequently, each BOBiSME intance is capable of performing signature matching at 10.6 Gbps (T) and the per context throughput, T_c that is achieved is 1.325 Gbps. As discussed in section 5.4, the effect of increasing clock frequencies, resulting in an increased signature matching throughput can be observed in the case of BOBiSME in comparison to BiSME. Similarly, the increased N in the case of BOBiSME, resulted in ∼15% reduction in the per context throughput T_c in comparison to BiSME.

Table 5.4 shows a summary of the logic and the memory area corresponding to a single BiSME and BOBiSME instances. Columns 3 and 4 in Table 5.4 shows a summary of the logic and memory area corresponding to a single BiSME instance. It can be seen from Table 5.4 that a significant portion of the BiSME (∼97.7%) is composed of the on-chip SRAMs which store the compressed DFA. The logic (both combinatorial and sequential) portion in BiSME consisted of 110k gates and consumed only ∼2.3% of the overall area. The memories used for BiSME were provisioned with the related test logic to yield realistic area numbers during synthesis. Power analysis simulations performed during the design synthesis resulted in 155 mW *(115.7 mW dynamic power + 35.3 mW static power)* of power consumed by each BiSME instance.

Table 5.4: Synthesis Area Results

		BiSME		BOBiSME	
Component		Area (in μmm^2)	Percentage	Area (in μmm^2)	Percentage
Memory	MS	1401641	97.7 %	1143535	96.5 %
Logic	MS	6915	2.3 %	7449	3.5 %
	DE	16439		21915	
	MAM	2082		2546	
	AD	7431		8983	
Total Area		1434508	100 %	1184428	100 %

On the other hand, columns 5 and 6 show the logic and the memory area corresponding to a single BOBiSME instance. Similar to BiSME, a significant portion of BOBiSME is composed of the on-chip SRAMs which constitutes ∼96.5% of the overall area. In comparison to BiSME, the reduction in the MBT memories, resulting from the bitmask compression in BOBiSME reduces the overall memory area by a factor of ∼20%. On the other hand, only an additional 22k gates are required to perform the calculations associated with the bitmask decompression and the member transition fetch. This overall increase in the logic just corresponds to 0.01 mm^2 of logic area which is very minor in comparison to the 0.26mm^2 reduction in the memory area due to bitmask compression. Similarly, the power analysis simulations performed during the design synthesis resulted in 167 mW of power consumed by each BOBiSME instance.

Table 5.5 compares the throughput, area and power consumption of BiSME and BOBiSME against various other signature matching engines proposed in the literature. The technology node in which each of the design was implemented is represented in brackets in Table 5.5. The signature matching throughput achieved by these systems is differentiated into peak and achievable throughput. The peak throughput refers to the theoretical maximum throughput for which the system is designed, while the achievable throughput represents the throughput that is achieved when the signature matching operation is performed across realistic workloads in these systems. Since the various signature matching engines which have been considered for comparison are implemented across various technology nodes, a simple comparison of the achievable signature matching throughput is not possible. Moreover, since these are hardware implementations, there is neither a standard platform and nor standard datasets through which the signature matching throughput can be compared across all these systems.

The signature matching engines proposed by Tuck et al. [63] and FEACAN are bitmap based engines and are capable of performing signature matching at 8 Gbps. Similarly, each instance of HAWK is also capable of performing signature matching at 8 Gbps[15]. In comparison, each instance of BiSME performs signature matching at 9.3 Gbps which is about ∼1.15 times higher than the solutions described above. On the other hand, due to a

[15]Though there are variations in HAWK which can process multiple bytes per clock cycle, the HAWK version which consumes a single byte per clock cycle is considered for comparison purposes.

Table 5.5: Comparison of BiSME against other Hardware Engines

	Throughput (in Gbps)		Area (in mm^2)		Power
	Peak	Achievable	Logic	Mem.	
BiSME (28nm)	9.3	9.3	0.03	1.4	.155W
BOBiSME (28nm)	10.6	10.6	0.04	1.14	.167W
FEACAN	8	8	N.A		
Tuck et al.[63]	∼8	∼8	N.A.		
HAWK (45nm)	8	8	1.4	5.7	2.6W
RegX (45nm)	73.6	15-40	15.4		N.A.
UAP (32nm)	9.6/lane	∼5/lane	2.2	3.5	1W

deeper hardware pipeline in the decompression engine, the BOBiSME performs signature matching at 10.6 Gbps, which is ∼1.325 times higher than the bitmap based solutions [63, 4] described above. Since all the engines mentioned above store the signatures in the on-chip memories alone, the achievable throughput is identical to the peak throughput for which the system is designed. With respect to area and power consumption, there are no results available for FEACAN and the one proposed by Tuck et al. in the respective publications. On the other hand, each instance of HAWK consumes 7.1 mm^2 silicon area and 2.6W power. Since the technology nodes in which these systems are evaluated are completely different, a direct comparison of the area and the power consumption is not discussed further in this dissertation.

Unlike the other implementations discussed above, both the UAP and RegX engines store the automata in a combination of on-chip and off-chip memories. So, the achievable throughput varies depending on the number of signatures compiled into the system and the sequence of payload bytes inspected by the system[16]. The throughput results corresponding to RegX (4 physical lanes) seen in Table 5.5 shows the variation in the achievable throughput which is roughly between 20-50% of the peak throughput for which the system is designed for. Similar results are also seen in the case of a single instance of UAP, in which the achievable throughput is only about 5 Gbps/lane (∼50% efficiency) when it is used to compare the payload bytes against the compressed DFA representations. The RegX engine also employs a cache controller to fetch the data from the off-chip memories which consumes a considerable portion of the engine which is seen from the area usage of the accelerator. While in the case of UAP, no specific results associated with such controllers are discussed, but will inevitably be required to fetch the data from the off-chip memories. In comparison, since BiSME and BOBiSME completely store the compressed DFA in the on-chip memories, the signature matching throughput and the number of signatures supported can be linearly scaled by using multiple instances of these engines. The specific number of BiSME/BOBiSME instances

[16]The RegX implementation only uses a sequence of 1000 bytes to evaluate the throughput of the system. The implications of increasing the length of the byte sequence is never discussed by the authors. Similarly the implications of varying the number of signatures inserted in the injected byte sequence is also not compared in the publication.

to be part of an overall system depends on the requirements pertaining to number of signatures and the throughput to be supported.

5.6.2 Signature Capacity

5.6.2.1 BiSME

BiSME's capacity was evaluated using the five different signature sets which have been used over the dissertation for evaluation purposes. A compiler was developed to perform the MSBT transition compression and to convert the compressed DFA into the required memory formats defined in BiSME. The exact_match and the bro signature sets were compiled into one BiSME instance. On the other hand, since the total number of groups generated during the state grouping step exceeded 256[17], the snort signatures were partitioned into 2 DFAs and were compressed separately[18].

Columns 3, 4 and 5 in Table 5.6 show the memory utilization in BiSME. Since the transition compression rates achieved by MSBT are much higher than 98%, only a small portion of the transition memories were used in BiSME. On the other hand, the control memory utilization varied depending on the signature sets. For example, due to the long average bitmask width, the exact_match signature set resulted in 92% of MBT utilized. In the case of other signatures, the MBT usage was not as high as the exact_match signature set. However, it can be seen from Table 5.6 that each instance of BiSME is capable of storing over 500 string signatures with a possibility to use multiple instances to scale higher signature counts.

5.6.2.2 BOBiSME

Similar to the evaluation of BiSME, the signature sets are compressed and converted into the memory formats supported by BOBiSME. Table 5.7 shows the usage of various memories in BOBiSME when the considered signatures are converted into BOBiSME

[17]It should be remembered that Snort24, Snort31 and Snort34 signature sets generated 428, 420 and 518 groups, respectively as discussed in Chapter 4.

[18]The current version of the compiler developed to convert the DFA into the memory formats in BiSME did not have the capability to optimize the transition compression to store the signature sets in the predefined memories.

Table 5.6: Signature sets compiled into BiSME

Signature Set	#BiSME Instances	Memory Utilization (%)		
		AMT	MBT	LTT/MTT/SM
exact_match	1	43	92	37
bro217	1	38	29	27
snort24	2	64	7	11
snort31	2	46	15	16
snort34	2	38	4	6

specific memory formats through a compiler. The trends which were observed in BiSME with respect to transition storage are also observed in the case of BOBiSME. Since the bitmask compression only focuses on compressing the redundant MTBs, the percentage of the transition memories being used to store the signatures remain the same between BiSME and BOBiSME. On the other hand, due to the introduction of bitmask compression, even in the case of the exact_match signature set, only ∼50% of the MBT is used after being converted into BOBiSME formats, which allows more string signatures to be stored in a single BOBiSME instance.

In order to identify the maximum capacity of BOBiSME, additional synthetic string signatures were generated using the regex tool [86]. The average length of the signatures were set to 50 bytes and two additional signature sets were generated with 750 and 1000 signatures, respectively. Table 5.8 shows the transition compression rates and the memory utilization for the newly created signature sets. It can be seen from Table 5.8 that a maximum of 1000 string signatures can be comfortably stored in BOBiSME and the additional number of signatures which are stored is due to the increased memory availability due to bitmask compression. It can also be seen from Table 5.8 that a compressed DFA whose state count is more than 16k can be comfortably stored in the BOBiSME if higher transition compression rates are achieved and the bitmasks are effectively compressed.

5.6.3 Hardware Implementation Validation

The BiSME and the BOBiSME were evaluated on the Cadence Palladium platform [102]. Palladium is a processor based compute engine which is used to emulate and verify the functionality of the hardware design. The palladium platform allows the emulated design

Table 5.7: Signature sets compiled into BOBiSME

Signature Set	#BiSME Instances	Memory Utilization (%)		
		AMT	MBT	LTT/MTT/SM
exact_match	1	43	49	37
bro217	1	38	28	27
snort24	2	64	8	11
snort31	2	46	8	16
snort34	2	38	4	6

Table 5.8: Evaluation BOBiSME string signature capacity

Signature Set	# Signatures	# States	Transition Compression Rate	Memory Utilization (%)		
				AMT	MBT	LTT/MTT/SM
exact_match_500	500	15149	99.07	43	49	37
exact_match_750	750	24034	99.18	67	73	51
exact_match_1000	1000	31633	99.17	88	93	68

to interact with various real high speed and low speed interfaces which allow to verify the design with real traffic patterns, but at emulated speeds.

Figure 5.20(a) shows the pictorial representation of the evaluation setup which is used to verify the functionality of the signature matching engines. A compiler running on the Palladium Host Computer (a dedicated computer which communicates with the Palladium hardware) compiles the hardware design and the testbench components into Palladium specific hardware primitive blocks. The packets to be injected into the system were either generated through a software based Ethernet packet generation tool called Bittwist [103] or using the Spirent Test Center platform [104]. Since the emulated design frequency was much lesser than the original operating frequency of the design, the Ethernet Speedbridge [105] was used to regulate the communication between the traffic generation system (which operates at line rates) and the emulated design. Furthermore, the Palladium was connected to a reference board which enables to communicate with various serial control interfaces which are used to download various control information into the emulated design. A computer connected to the reference board through the RS-232 port was used to download the signatures and the control information into the emulated design.

Figure 5.20(b) shows an overview of the evaluation setup from the system perspective, to show the various interfaces which were used for communication. The evaluation setup consisted of 2 Ethernet ports terminating at the Gigabit Media Independent Interface (GMII) and a Universal Asynchronous Transmitter Receiver (UART) interface. The UART interface was used to program the various register signals in the signature matching engine and to download the compiled signatures into the emulated design. The first GMII interface (port 0) was used to inject the traffic into the signature matching engine while the other interface (port 1) was used to monitor the packets in which a signature match is identified. A short overview of the traffic generation process is explained first before detailing the overview of the design and the testbench components implemented in Cadence Palladium.

5.6.3.1 Traffic Generation

The network traffic traces which are used to evaluate the signature matching engines should comprise varying number of signatures in it. This allows to verify the functionality of the engines across a multitude of scenarios. To verify the signature matching engines, the traces can either be extracted from real network traffic captures or can be generated synthetically. In the case of the former, it is very difficult to obtain traces which contain the signatures corresponding to each and every signature set. On the other hand, synthetic traffic generation provides the flexibility to generate traffic traces with varying number of signatures in it. Due to this advantage, various other automata based signature matching engines proposed in the literature [90, 89, 36, 62] have used the synthetic traffic generation mechanism to evaluate their signature matching engine implementations. So, the synthetic traffic generation mechanism is used to verify the proposed signature matching engines.

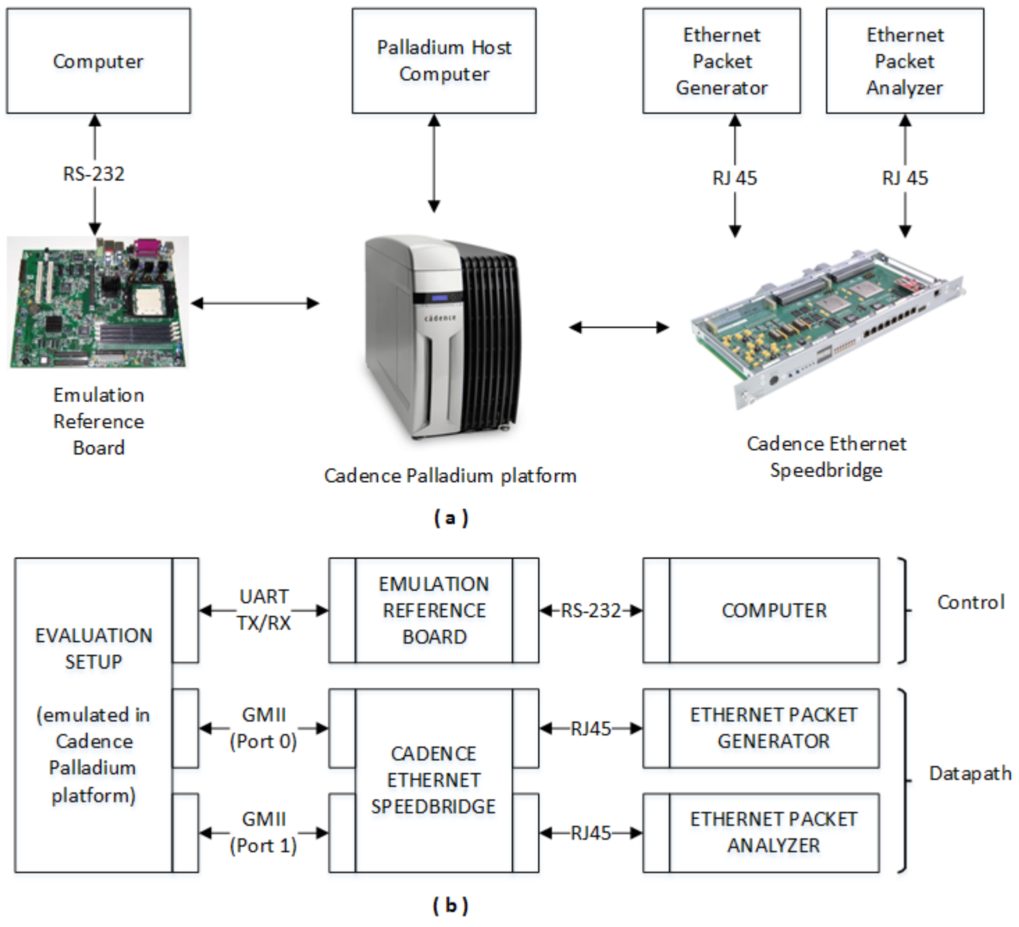

Computer
Palladium Host Computer
Ethernet Packet Generator
Ethernet Packet Analyzer
RS-232
RJ 45
RJ 45
Emulation Reference Board
Cadence Palladium platform
Cadence Ethernet Speedbridge
(a)
EVALUATION SETUP
(emulated in Cadence Palladium platform)
UART TX/RX
EMULATION REFERENCE BOARD
RS-232
COMPUTER
Control
GMII (Port 0)
GMII (Port 1)
CADENCE ETHERNET SPEEDBRIDGE
RJ45
RJ45
ETHERNET PACKET GENERATOR
ETHERNET PACKET ANALYZER
Datapath
(b)

Figure 5.20: A pictorial representation of the evaluation setup used to verify the SME hardware

The byte sequences to be injected into the system are generated through two different methods and are evaluated separately. In the first method, i.e., *controlled sequence*, the payload bytes are generated in a controlled manner. In this method, the number of signatures which are introduced in the trace is made configurable. The trace generation mechanism proposed in [90] which is part of the regex tool [86] is used in this method to generate the payload byte sequences. In this method, each byte in the sequence is generated based on a fixed predefined probability of maliciousness P_M, by simultaneously traversing the automaton (either the NFA/DFA) corresponding to the individual signature sets. The P_M defines the probability with which a byte (among the 256 ASCII characters) is chosen for the payload byte sequence such that, the chosen byte leads the byte sequence towards plausible signature match[19]. In order to cover a range of scenarios, the P_M values 0.35, 0.55, 0.75 and 0.95 are used to generate the traffic traces. The P_M value of 0.35 represents the average network traffic scenario while 0.95 represents

[19]Appendix A provides the details of P_M based trace generation.

the worst case network traffic scenario [90]. In this mechanism, the raw payload byte sequence to be inspected is generated first which is further split into multiple Ethernet frames. Figure 5.21(a) shows an example of how a payload byte sequence of 4500 bytes in a context is split across multiple Ethernet frames each consisting of a maximum of 1500 bytes[20].

Since BiSME and BOBiSME are only capable of processing 6 and 8 contexts in parallel, the payload byte sequences are generated separately for 6 and 8 streams, respectively. In this way, the hardware pipeline is always kept busy to effectively evaluate the proposed signature matching engines. The Ethernet frame corresponding to each of the contexts is encapsulated with a unique combination of source and destination Media Access Control (MAC) addresses as shown in Table 5.9 for easy packet classification. Figure 5.21(b) shows the order in which the Ethernet frames sent in the physical link. The packets from each of the respective streams are arbitrated and sent in a round-robin fashion, so that the payload bytes are distributed efficiently. A UNIX bash script was written to completely automate the above mentioned process.

In the second scenario, i.e., *random sequence*, the payload bytes in the Ethernet frames are generated in a random fashion. Since the payload bytes are generated in a random fashion, the number of signatures generated in the payload byte sequence cannot be determined prior to the generation. The Spirent Test Center [104] is used to generate the Ethernet frames with the random payload byte sequence. The Spirent Test Center has an inbuilt feature to generate the Ethernet frames corresponding to multiple streams

[20]Ethernet packets are generated with 1500 bytes as this would be the worst case scenario corresponding to signature matching for a given packet.

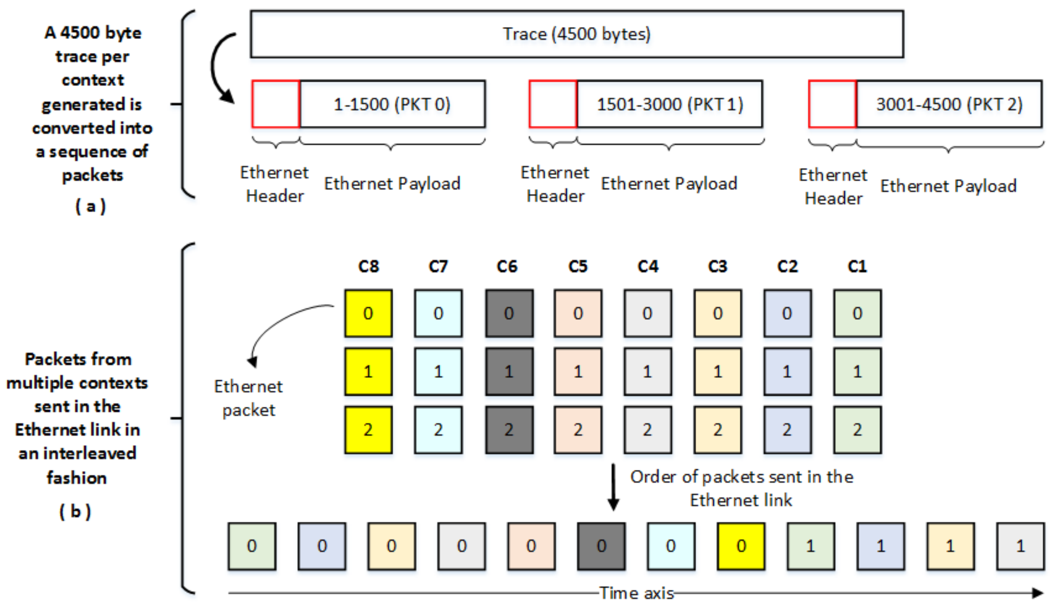


Figure 5.21: Functional overview of Traffic generation

Table 5.9: Source and Destination MAC address combinations to identify a context

Context	Source MAC address	Destination MAC Address
1	00:00:00:00:00:01	00:00:00:00:00:02
2	01:00:00:00:00:03	01:00:00:00:00:04
3	02:00:00:00:00:05	02:00:00:00:00:06
4	03:00:00:00:00:07	03:00:00:00:00:08
5	04:00:00:00:00:09	04:00:00:00:00:0A
6	05:00:00:00:00:0B	05:00:00:00:00:0C
7	06:00:00:00:00:0D	06:00:00:00:00:0E
8	07:00:00:00:00:0F	07:00:00:00:00:10

and send them in a round-robin fashion in the physical link. So this feature is directly used to generate the Ethernet frames.

The next section describes the detailed internal architecture of the evaluation setup and how the testbench components extract the payload bytes from the packets and send them to the signature matching engines.

5.6.3.2 Internal Architecture of the Evaluation Setup

Figure 5.22(a) shows the block level internal architecture of the evaluation setup. The evaluation setup primarily consists of the signature matching engine and various additional testbench components to interface the signature matching engines with the external interfaces shown in Figure 5.20(b). Since the interface of communication to BiSME and BOBiSME are identical, both the engines are verified using the same evaluation setup. The testbench components designed to evaluate the signature matching engines are split into the control and the datapath subsystems.

The control subsystem assists in downloading the compressed signatures and the various other control information required in the evaluation setup and the signature matching engines. The control subsystem consists of the UART controller and the control status block. The UART controller converts the serial data transmitted through the UART interface into the parallel PDI 16-bit interface as shown in Figure 5.22(d). Once the data is received in the PDI interface, the incoming address is used to identify if the transaction is directed towards the signature preload interface or to program one of the control signals defined in the evaluation setup. The other blocks seen in Figure 5.22(a) constitutes the datapath subsystem. The datapath subsystem consists of various blocks to receive, prepare and process the Ethernet frames for signature matching. Figure 5.23 shows an example of how the Ethernet frames are processed in the datapath subsystem. The functions performed by various blocks within the datapath subsystem is described below:

- The datapath block consists of a Gigabit Media Access Control (GMAC) block that is capable of sending and receiving the Ethernet frames. The GMAC block

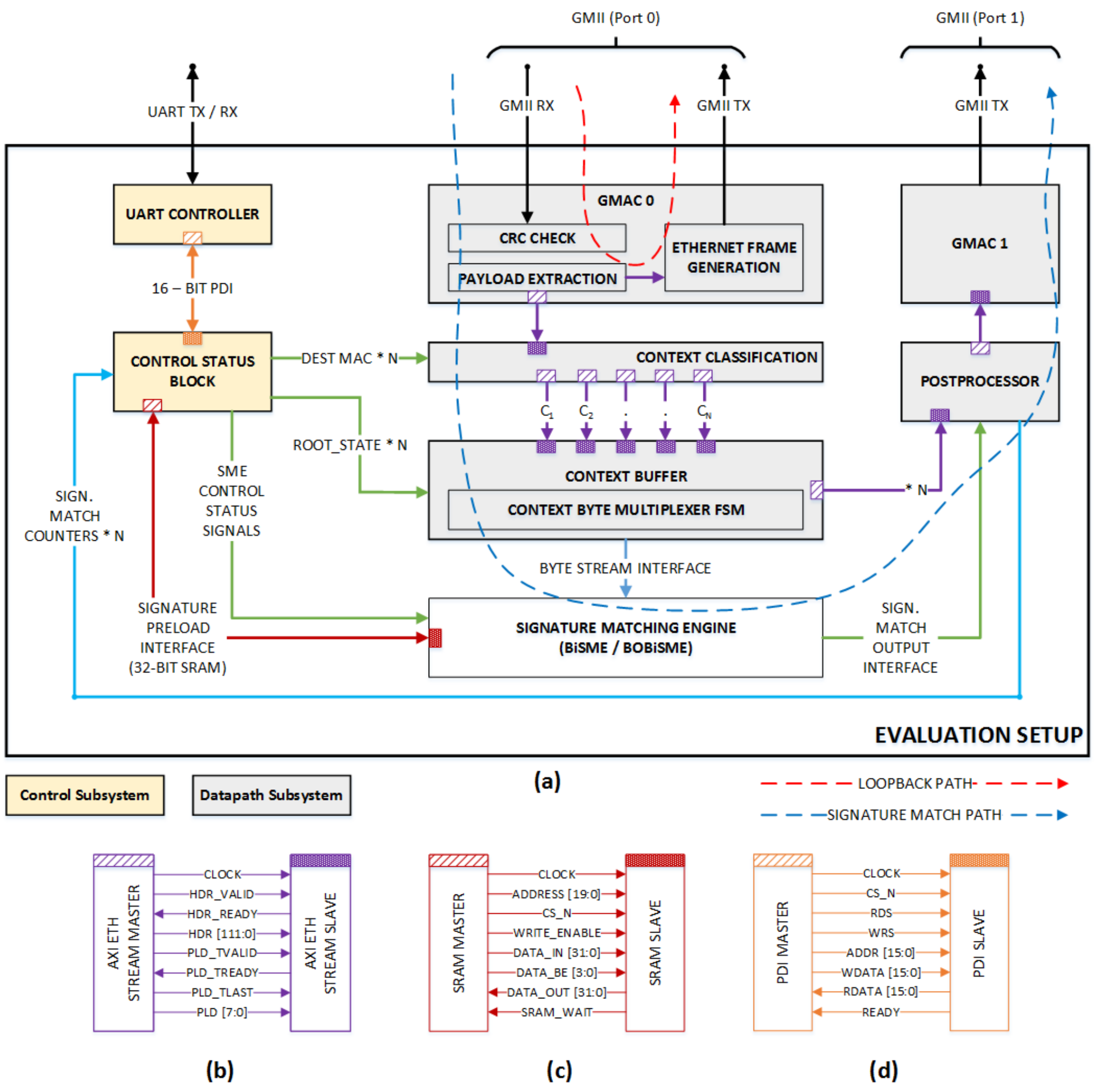


Figure 5.22: Detailed block level internal architecture of the evaluation setup

receives the Ethernet frames and performs the cyclic redundancy check to identify the validity of the frame and further regenerates the valid frames to loop them back to the source. The frames are looped back to the source to verify the signature matching results through an alternate source, if required. The incoming Ethernet frame is split into its header and payload in this block and is sent to the context classification block for further processing. A modified version of the AXI4 stream interface [106], shown in Figure 5.22(b) is used to communicate between the various blocks after the payload extraction.

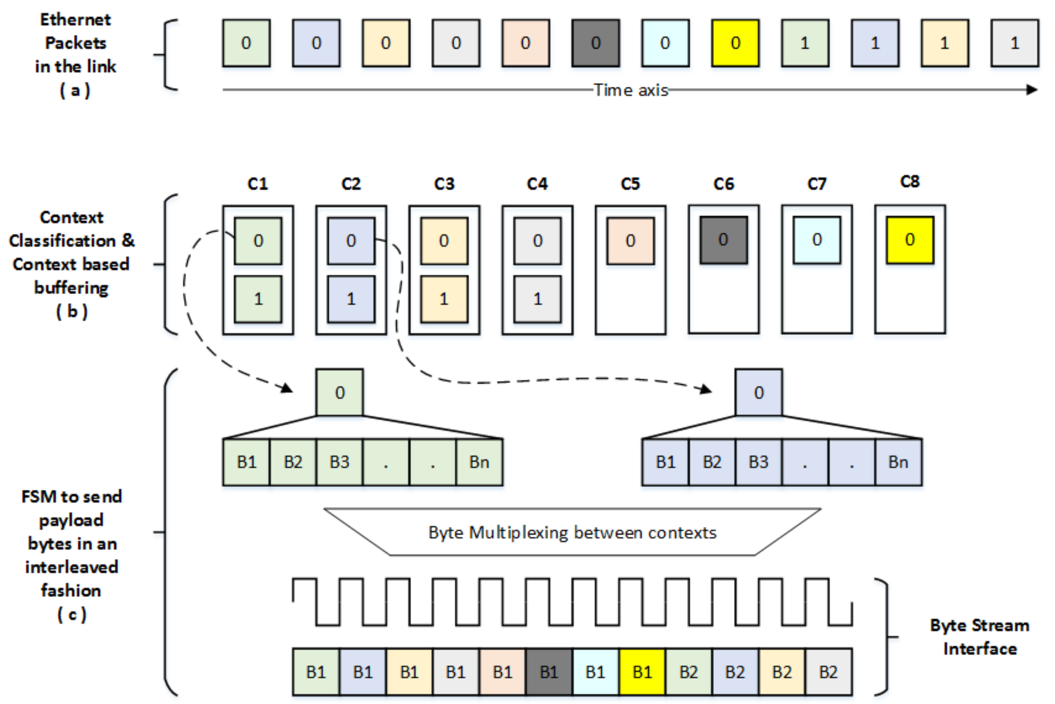

Ethernet Packets in the link (a)
0
0
0
0
0
0
0
0
1
1
1
1
Time axis
Context Classification & Context based buffering (b)
C1
C2
C3
C4
C5
C6
C7
C8
FSM to send payload bytes in an interleaved fashion (c)
B1 B2 B3 . . Bn
Byte Multiplexing between contexts
B1 B1 B1 B1 B1 B1 B1 B1 B2 B2 B2 B2
Byte Stream Interface

Figure 5.23: Extraction of the payload bytes and injection into the signature matching engine

- After the frame is split into its header and payload, it is further classified into a stream context. The destination Media Access Control (MAC) address is used to classify the frames into contexts. A *context definition table* in the context classification block stores the MAC addresses of the contexts, which is used to classify the incoming frames into their respective contexts. The context classification block compares the destination MAC address of the incoming frame against the MAC addresses stored in the context definition table. The content of the context definition table is configured through the UART interface so that the content of the table can be modified, if required. After classifying the frames into their respective contexts, the frames corresponding to each of the contexts are sent to the context buffer block through a dedicated AXI4 interface for each of contexts as shown in Figure 5.23(b).

- As the frames are being filled into the context buffers, a finite state machine in the context buffer block fetches the payload bytes across all the contexts and sends them to the signature matching engine in an interleaved fashion. This is pictorially represented in Figure 5.23(c). The finite state machine logic streams all the payload bytes of a frame into the signature matching engine before the arrival of the next frame across all the streams. The root state of the DFA is programmed through the UART interface and is stored in the context buffer block. So, the root state is also sent to the signature matching engine when the first payload byte corresponding to the stream is sent across to the signature matching engine.

- As the payload bytes are sent to the signature matching engine, they are simultaneously sent to the postprocessor block. If a signature match is identified by the signature matching engine, the postprocessor is notified through the signature match output interface. The postprocessor keeps track of the number of signature matches identified by the hardware on a per context basis. The signature match counters in the postprocessor block are made accessible through the UART interface. Additionally, the accepted states are also tracked for each of the context in the postprocessor block to keep track of the unique signatures which matched among the payload bytes. The postprocessor block also sends those frames in which a signature match is identified through the GMAC 1. If the signature match is identified in the first half of the frame[21], the frame in which the signature match is found and its predecessor are sent to the GMAC. If not, the frame with the signature match and its successor are sent to the GMAC. Sending two frames corresponding to a signature match enables to identify the signature match by an external agent even if the signature is spread across multiple frames.

In order to simplify the evaluation setup implementation, all the blocks are supplied with the same clock signal. A clock signal with frequency of 125 MHz is chosen to enable the GMAC to virtually process the packets at 1 Gbps[22]. The signature matching engine is also clocked at 125 MHz, resulting in a slowdown factor of ∼9-10x. The reduction in the clock frequency doesn't affect the functionality of the implementation but simplifies the overall evaluation setup. Even if the signature matching engine was clocked at the target clock frequency at which it was synthesized, the actual achievable clock frequency in Palladium doesn't vary. This is because of the fact that the achievable frequency in palladium is restricted by the physical implementation of the design in the platform and is typically in the range of 0-4 MHz [102].

Two different emulation databases[23] were separately created to verify BiSME and BOBiSME, respectively. The actual frequency corresponding to the 125 MHz clock in both the databases were 1.1 MHz. So, the signature matching throughput achieved by the emulated design is 8.8 Mbps. This is roughly a scaledown factor of 1000x in the evaluation setup in comparison to the original target frequency of operation.

5.6.3.3 Results & Discussion

BiSME and BOBiSME were thoroughly evaluated using the bro217 and the exact_match signature sets described in Chapter 3. The compressed signatures were downloaded

[21]For example, if there are 128 payload bytes in a frame, bytes 1-64 are considered the first half while bytes 65-128 are considered the second half.

[22]Since the Palladium is a cycle accurate system, the reduction in the clock frequency doesn't affect the functionality of the implementation

[23]The term database is used to represent the hardware design which is compiled and emulated in Palladium

into the emulation platform through the UART interface along with the other control information required to configure the signature matching engines and the evaluation setup. As described earlier, both the controlled and the random sequence of payload bytes were injected to verify the signature matching engines. In both the cases, the generated traffic was also compared against the DFA based signature matching engine implemented as software through the Regex tool [86].

The payload bytes for the controlled sequence was generated from the DFA corresponding to the signature set using the regex tool. In order to minimize the storage of the traffic traces, the payload bytes for the controlled traffic test was split across multiple passes. In each pass, ~384 KB[24] of payload bytes per context was generated, which resulted in 2.3 MB of data injected per pass in the case of BiSME and 3 MB of payload bytes injected per pass in the case of BOBiSME. The controlled traffic injection test was automated using a UNIX bash script which performed the following functions.

- Step 1: 384KB Payload byte sequence generation per context.
- Step 2: Compare the payload byte sequence with the DFA to collect the signature matching statistics per context.
- Step 3: Convert the payload byte sequence into Ethernet frames for each of the contexts.
- Step 4: Interleave the Ethernet frames across multiple contexts and inject them into the emulated design through the Ethernet link in the computer.

The above mentioned steps were performed on a computer which was running on an Intel processor. The processor was running at 3 GHz and consisted of 3 MB of on-chip cache memories. The tests were first performed on BiSME and then on BOBiSME. The corresponding test results are discussed separately.

5.6.3.3.1 BiSME

As a first test, the Ethernet frames were injected into the evaluation setup only for 10 passes. This test was basically performed to verify the traffic generation methodology[25]. Since BiSME is only capable of processing 6 contexts in parallel, the Ethernet frames were injected corresponding to 6 unique streams. In order to verify the signature matching engine across different traffic characteristics, the payload byte sequence for the streams were generated with different P_M values. P_M values of 0.35, 0.55, 0.75, 0.95, 0.35, 0.55

[24]The amount of payload bytes injected in every pass was carefully optimized to not affect the rate of traffic injection by bittwist as injecting a bigger amount of data will add additional processing latency to fetch the packets from off-chip memories.

[25]Prior to this test, cycle accurate RTL simulations were performed in the evaluation setup to verify the complete evaluation setup. In the RTL simulations, traffic traces of 4KB corresponding to different P_M values were injected into the system. The location of the signature matches observed in the Ethernet frames were identical to that of the uncompressed DFA which verified the implementation of the overall evaluation setup.

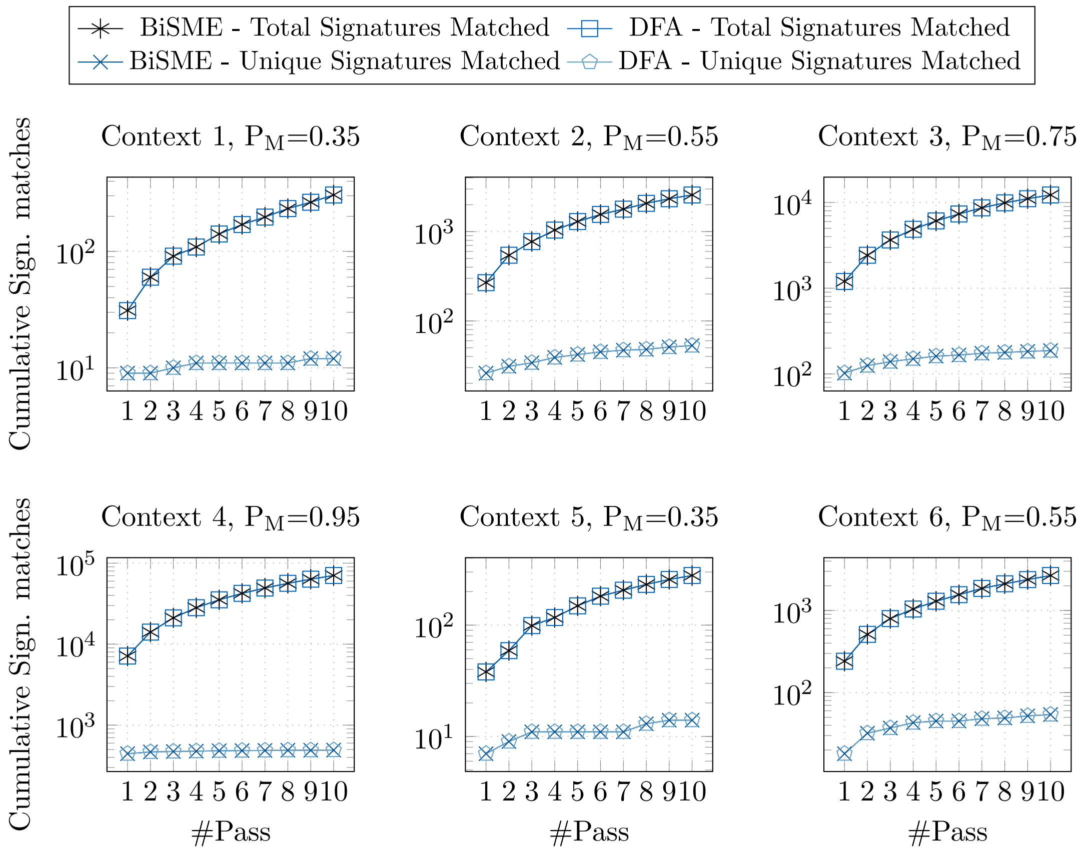


Figure 5.24: Signature Matching results on the exact_match signature set in BiSME for 10 passes

were chosen to generate the payload bytes corresponding to contexts 1, 2, 3, 4, 5 and 6, respectively.

Figure 5.24 shows the signature matching results corresponding to the exact_match signature set for each of the individual passes across the 6 contexts. It can be seen from Figure 5.24 that the cumulative number of signatures matched across all the contexts, for each of the incremental passes is identical between BiSME and the DFA. Certain signatures in the traffic trace do occur multiple times. So, the unique number of signatures matched is also tracked which represents the total number of non-repetitive signature matches observed in the traffic traces. It can also be seen from Figure 5.24 that the unique number of signatures matched across all the contexts with every incremental pass is also identical between BiSME and the DFA. The identical signature matches in this test verifies the functional correctness of the evaluation setup.

As a next step, BiSME is stress tested by injecting multiple gigabytes of payload bytes. In this test, a total of 2.3 GB of payload bytes (∼366 MB/context) were in-

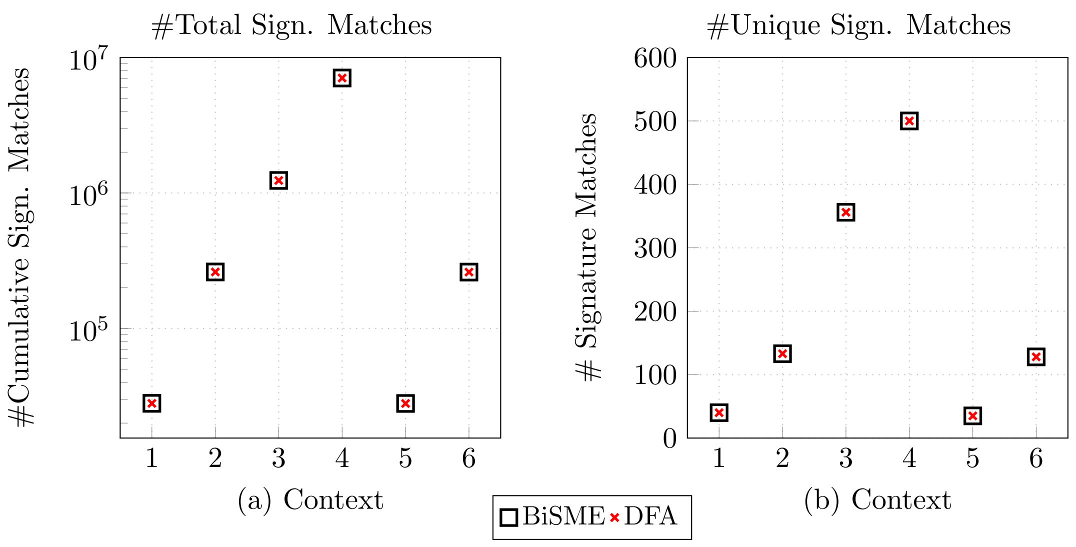


Figure 5.25: Signature matching results after 2.3 GB of controlled traffic inspection on the exact_match signature set

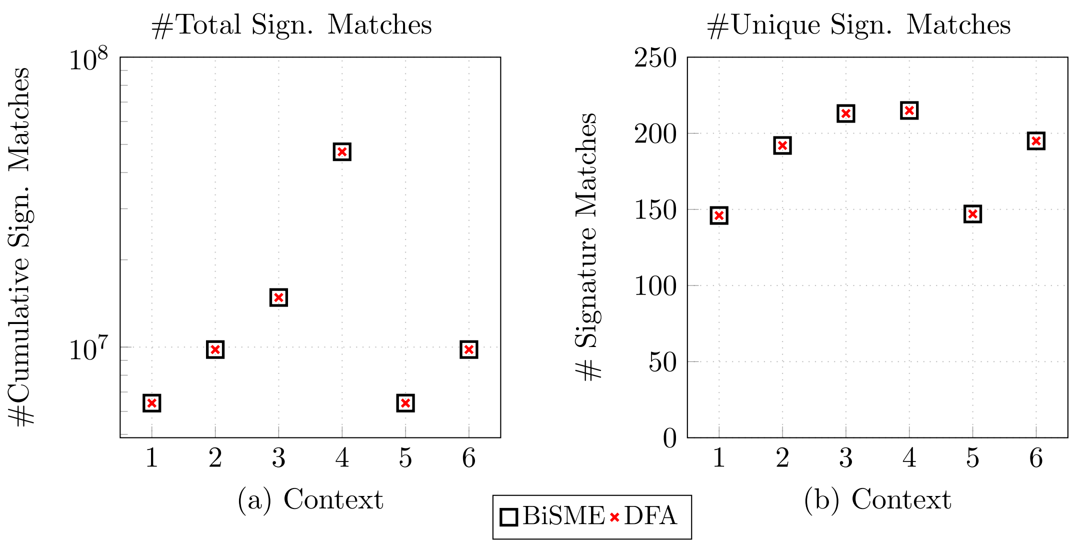


Figure 5.26: Signature matching results after 2.3 GB of controlled traffic inspection on the bro217 signature set

jected into the system over a period of 12 hours[26] across 1000 passes. Figure 5.25 and Figure 5.26 show the signature matching results for the exact_match and the bro217 signature sets, respectively. Figure 5.25(a) and Figure 5.26(a), respectively show the cumulative total number of signature matches observed after injecting 2.3 GB of pay-

[26]Since the Palladium is a shared emulation platform, the maximum time for running the tests was set to 12 hours to share the resource with other projects.

load bytes, while Figure 5.25(b) and Figure 5.26(b), respectively show the total number of unique signature matches observed. It can be seen from both the figures that the signature matching results across all the contexts are identical between BiSME and the DFA. Thus, the identical signature matching results in both the signature sets further verifies the functionality of BiSME.

An important observation can also be made from the figures shown above. The signature matches (both cumulative total and unique signatures) seen in each of the contexts varies as it depends on the characteristics (P_M value) of the payload bytes being inspected. However, the number of signature matches (both cumulative total and unique signatures) observed in BiSME and the DFA are identical across all the contexts. This clearly shows that the per context signature matching throughput in BiSME is always constant and is independent of the characteristics of the payload bytes being inspected.

By performing the tests over a longer period of time, a signature coverage of 100% was achieved for the exact_match signature set. This can be seen from Figure 5.25(b), where all the 500 signatures are matched at least once in the case of context 4 (P_M=0.95). On the other hand, a signature coverage of 99% (215/217) was achieved for the bro217 signature set as seen in Figure 5.26(b).

After performing the tests based on the controlled sequence traffic, BiSME is further verified through the random sequence traffic. The spirent test center[27] is configured to inject Ethernet frames corresponding to the contexts in an interleaved manner as shown in Figure 5.21(b). In the case of the controlled sequence tests described previously, the generated traces were simultaneously compared against the DFA to double confirm the results. Since, this is not possible in the case of the random sequence traffic tests, the generated Ethernet frames were captured using wireshark [108] and processed offline. After the Ethernet frames were captured and saved, the payload bytes corresponding to the individual contexts were extracted and further compared against the DFA separately. In this way, the signature matching results extracted from BiSME was cross-verified with that of the DFA in the case of the random sequence tests.

Similar to the controlled sequence test, the random sequence test was performed on the exact_match and the bro217 signature sets separately. Unlike the controlled sequence test, a total of 1 GB ($\sim 172MB/context$) of data was injected into the evaluation setup. Since the signature matching on the DFA was performed offline, the traffic injected was kept to 1GB. Figure 5.27(a) and (b), respectively show the cumulative total number of signature matches and the unique number of signature matches on the bro signature set across all the contexts. Figure 5.27(c) shows the unique number of signature matches on the exact_match signature set. Similar to the results seen in the controlled sequence, the signature matching results are identical across the signature sets in the random sequence as well. However, the total number of signatures matched in the case of the random sequence is much lesser than the number of signatures matched in the case of the controlled sequence observed previously. Since the payload bytes are generated in a random fashion, the chances of a signature being part of the random payload

[27]The Spirent Test Center uses $x^{23} + x^{18} + 1$ as the polynomial to generate the random sequence of bytes [107].

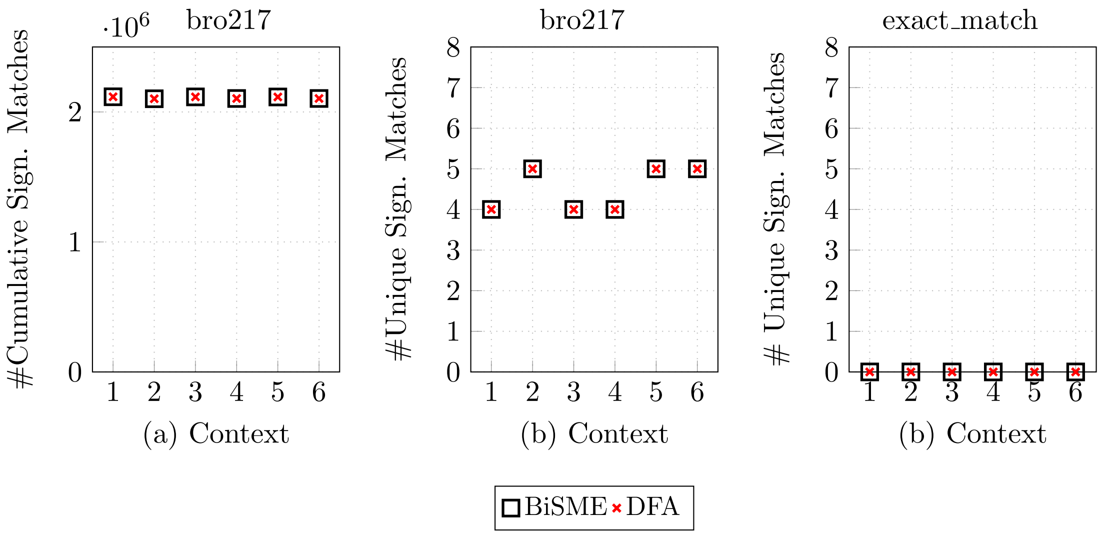


Figure 5.27: Signature matching results after 1 GB of random traffic inspection on the bro217 signature set

byte sequence is very minimal. Since the bro217 signature set consisted of a few short signatures, they could be matched through the random sequence test. On the other hand, in the case of the exact_match signature set, the average length of the signatures was about 50 bytes and none of the signatures could be matched by the random sequence of traffic generated as seen in Figure 5.27(c). However, the identical signature matching results seen in the random sequence tests also verify the functionality of BiSME.

5.6.3.3.2 BOBiSME

This section describes the signature matching test results performed on BOBiSME based on the controlled and the random sequence traffic injection mechanisms proposed earlier. The verification methodologies and the signature sets used to verify BOBiSME are similar to the ones which were used to verify BiSME. However, the only difference in the verification methodology is the number of unique streams for which the traffic is injected into the system. Since BOBiSME is capable of inspecting 8 contexts in parallel, the traffic is generated for a maximum of 8 streams for evaluating BOBiSME. For the controlled sequence tests, the payload byte sequence for contexts 1, 2, 3, 4, 5, 6, 7 and 8 were generated with P_M values set to 0.35, 0.55, 0.75, 0.95, 0.35, 0.55, 0.75 and 0.95, respectively.

As in the case of BiSME evaluation, a total of 2.3 GB of traffic was injected over 12 hours, with each context being injected with ~275 MB of payload bytes as part of the controlled sequence tests. Figure 5.28(a) and (b), respectively show the cumulative sum of signature matches and the total number of unique signature matched in the exact_match signature set across the various contexts. Figure 5.29(a) and (b), respectively show the signature matching results for the bro217 signature set. Figure 5.30 shows the

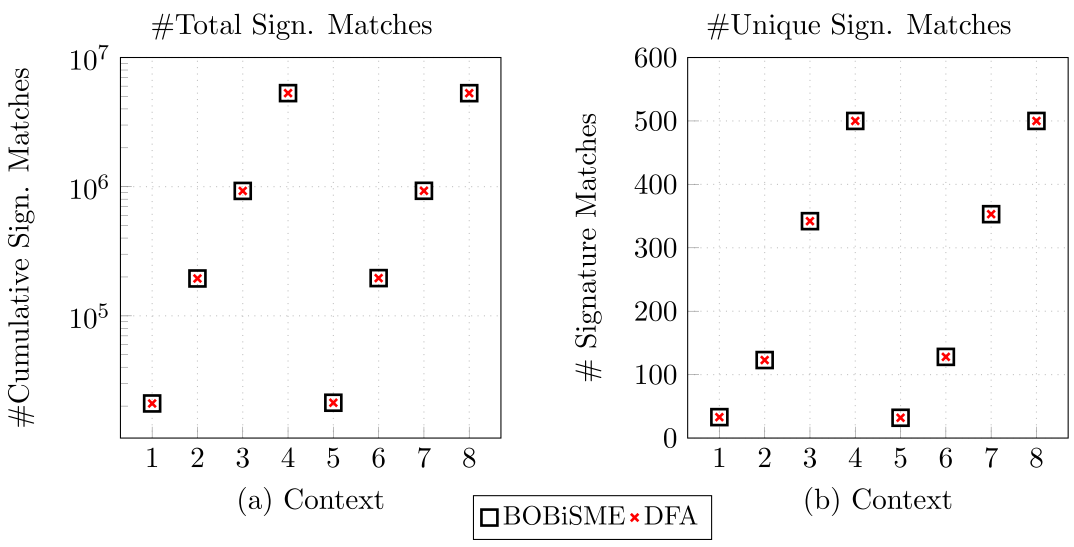


Figure 5.28: Signature matching results after 2.3 GB of controlled traffic inspection on the exact_match signature set

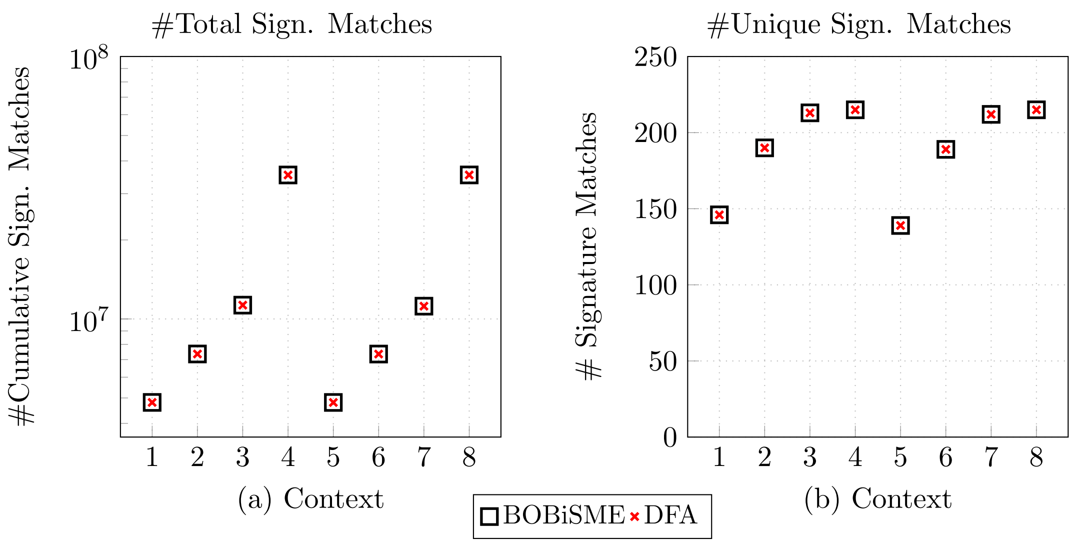


Figure 5.29: Signature matching results after 2.3 GB of controlled traffic inspection on the bro217 signature set

signature matching results on the BOBiSME after the random sequence tests. Similar to BiSME, 1 GB (128 MB/context) of payload bytes across multiple contexts were injected into BOBiSME during the random sequence tests. Various trends which were observed in BiSME evaluation are also observed in BOBiSME and are described below.

- It can be seen from Figures 5.28, 5.29 and 5.30 that the signature matching results are identical between BOBiSME and DFA in both the controlled and the random

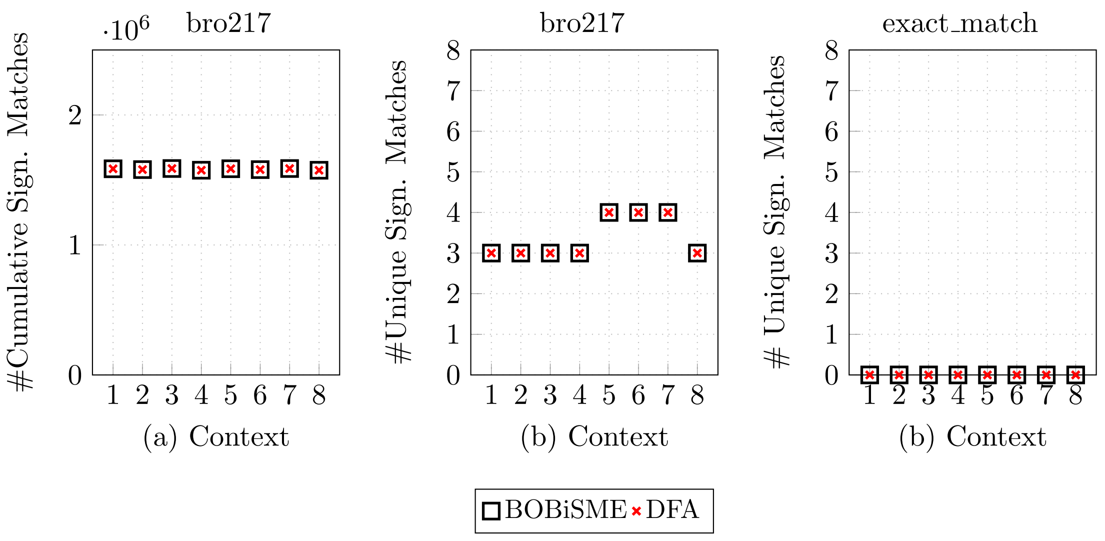


Figure 5.30: Signature matching results after 1 GB of random traffic inspection on the bro217 signature set

sequence tests. In both the test methodologies, the cumulative total number of signatures matched across all the contexts and the individual number of signatures matched are identical, further validating the correctness of BOBiSME hardware implementation. Similarly, the per context signature matching throughput achieved in BOBiSME is also independent of the traffic characteristics.

- As part of the controlled traffic injection tests, all the signatures among the exact_match signature set were matched at least once resulting in 100% signature coverage while 99% of the signatures were covered in the bro217 signature set.
- With respect to random traffic injection tests, none of the signatures among the exact_match signature set could be matched by the traffic generated, while a very small number of signatures in the bro217 were matched.

5.7 Summary

Hardware acceleration of signature matching is essential to achieve deep packet inspection at line rates. Addressing this topic, two hardware acceleration engines called BiSME and BOBiSME were proposed in this chapter which perform the signature matching function at 9.3 Gbps and 10.6 Gbps, respectively.

The various data which were generated after compressing the DFA using the MSBT and the BOMSBT are highly non-linear and signature dependent. Storing these data in the on-chip memories required flexible and efficient storage methodologies. The Packed Storage Methodology and the Shared Memory Methodology were first proposed in this chapter to effectively store the bitmasks and the compressed state transitions, respectively. The proposed architectures effectively allowed to store the compressed DFA in a flexible and programmable manner in the on-chip memories. In this way, the compressed DFA was completely stored in the on-chip memories which enabled to fetch the compressed state transitions at low latencies.

The BiSME and BOBiSME were implemented using Verilog and were synthesized on a commercial 28nm standard library operating at 0.81V. The proposed signature matching engines were effectively pipelined to achieve clock frequencies of 1.165 GHz and 1.325 GHz, respectively. This enabled them to achieve an overall signature matching throughput of 9.3 Gbps and 10.6 Gbps, respectively. Since the compressed DFA is completely stored in the on-chip memories, the achievable signature matching throughput is identical to the peak throughput for which the system is designed. The BiSME and BOBiSME take multiple clock cycles to process a single payload byte. So, in order to effectively utilize the hardware, payload bytes across multiple streams (contexts) are interleaved and injected into the signature matching engines. However, the signature matching throughput corresponding to a single context is always constant and it doesnt depend on the characteristics of the network traffic. The synthesis results showed that BiSME and BOBiSME only consumed 1.43 mm^2 and 1.18 mm^2 of on-chip area and only consumed .155W and .167W of power, respectively.

The BiSME and the BOBiSME signature matching engines are engulfed by the Deep Packet Inspection Accelerator. The DPIA is proposed to consist of the Network Data Management Engine which can effectively inject the payload byte stream for inspection into the signature matching engines. In this way, the processing associated with the payload injection and the postprocessing is proposed to be decoupled from the actual signature matching engine. Multiple instances of the signature matching engines can be instantiated within the DPIA to scale the supported signature matching throughput as well as the supported signature counts.

The functionality of the proposed signature matching engines were thoroughly verified on the Cadence Palladium platform by injecting $\sim$2.3 GB of payload bytes over more than 12 hours. The extended verification performed in the Palladium platform enabled to achieve 100% signature coverage to further verify that all the signatures within a signature set can be identified by the proposed signature matching engine. The traffic which was generated for the signature matching was simultaneously injected into a DFA based signature matching engine. The identical signature matching results between the hardware engines and the DFA, further verified the functional correctness of the hardware implementation.

6 Conclusion & Outlook

The signature matching is the most time critical function which has to be accelerated to perform deep packet inspection at line rates. The signatures which are represented through the strings and the regular expressions are either converted into the NFA or the DFA against which the payload bytes are compared. Since each payload byte can be compared against the DFA in constant time, the DFA is preferred over the NFA for line rate signature matching implementations. However, due to the transition redundancy and the state explosion problem, the DFA is highly storage inefficient. So, the DFA is generally compressed through the transition and the state compression algorithms and the payload bytes are compared against the compressed DFA as part of signature matching. Thus, the compression algorithms not only plays a role in effectively compressing the DFA, but also plays an important role in defining the achievable signature matching throughput.

The transition compression algorithms which have been proposed in the literature suffer from a variety of problems. Those solutions which use the default transitions to compress the redundant state transitions achieve very high transition compression rates, typically of the order of over 95%. However, since the transition compression is performed at the cost of increased memory bandwidth, the decompression cannot be performed in a dedicated hardware accelerator. On the other hand, the bitmap based approaches compress the DFA in such a way that the signature matching function can be performed in a dedicated hardware accelerator. However, these approaches only achieve transition compression rates of about 90-95% as they compromise the compression rates to minimize the number of bitmaps stored in the memory.

Addressing the inefficient transition compression rates in the bitmap based approaches, two bitmap based transition compression methods, the MSBT and the LSCT were proposed and evaluated in this dissertation. The MSBT and the LSCT primarily use the bitmap to eliminate intra-state transition redundancy and introduce the bitmasks to eliminate the inter-state transition redundancy. The bitmasks form the secondary layer of indexing which efficiently identify the redundant state transitions between multiple states within the DFA and generates an effectively compressed DFA representation. Additionally, a compression-aware divide and conquer state grouping method was proposed to effectively group the states through which the states are coherently clustered to effectively perform bitmask based inter-state transition compression. Experimental evaluation of the proposed methods showed that the MSBT and the LSCT in combination with the proposed state grouping method can effectively compress the DFA and achieve transition compression rates of the order of 98-99%, a 4-5% increase in comparison to the state-of-the-art. The cost paid to achieve such high transition compression rates is the necessity to store the bitmask together with the compressed state transitions. However, the linearity in the organization of the characters in the signature sets were further leveraged to compress the redundant bitmasks. The evaluation on the estimated memory used to store the compressed DFA further showed that the overall memory footprint of the compressed DFA reduced by 70% through the MSBT and the LSCT in combination with the bitmask compression.

Compressing the redundant state transitions through the bitmaps and the bitmasks effectively allows the decompression to be performed in a dedicated hardware accelerator. Two signature matching engines, BiSME and BOBiSME were proposed and implemented to perform the signature matching function after compressing the DFA through the MSBT. The BiSME and the BOBiSME store the compressed DFA in the on-chip memories, which allows them to be accessed at low latencies to perform the signature matching function at lines rates. The proposed signature matching engines implement two flexible and programmable storage methodologies to effectively store the compressed state transitions and the control data in the on-chip memories, while the decompression is performed through dedicated hardware circuits. The BiSME and the BOBiSME were effectively pipelined to achieve signature matching throughput of 9.3 Gbps and 10.6 Gbps, respectively. Since the compressed signatures are completely stored in the on-chip memories, the achievable signature matching throughput is equivalent to the peak throughput for which the engines are designed for. The proposed signature matching engines were implemented using Verilog and synthesized on the TSMC 28nm technology node operating at 0.81V. The BiSME and the BOBiSME only consume 1.43 mm^2 and 1.18 mm^2 of silicon area and 155 mW and 170 mW power, respectively which make them area-efficient and power efficient architectures. The proposed hardware accelerators were verified using cycle-accurate RTL simulations and further on the Cadence Palladium platform by injecting over 2GB of synthetic traces over a period of 12 hours. The generated traces were simultaneously injected into a DFA based signature matching engine implemented as a software to further verify the signature matching results from the hardware implementation. The identical signature matching results between the hardware implemented in Palladium and the DFA based signature matching engines further verified the correctness of the hardware implementation.

The Deep Packet Inspection Accelerator was further proposed which engulfs the proposed signature matching engines and effectively enables them to be integrated with standard network-on-chip architectures. Multiple instances of the signatures matching engines can be implemented to effectively scale the signature matching throughput, the number of supported signatures or a combination of both. The DPIA also consists of the network data management engine which decouples the functionality associated with stream related data processing from the signature matching engines.

Though the key concept behind the network data management engine was proposed, they were not implemented as part of the dissertation. The programmable methodology to map the network streams into contexts will be an exciting future research area to complement the signature matching engines. Through the programmable stream mapping methodology, the network streams can effectively be mapped to the contexts to enable payload inspection across a multitude of network streams. Moreover, intelligent algorithms to self-optimize the stream to context mapping process will add additional intelligence to the network data management engine and would be an exciting research area.

The population count function forms the most important part of the transition decompression when the redundant transitions are compressed through the bitmaps. The population count is available as an instruction in major processor architectures including the Intel x86, ARM etc. Though the main intention of the bitmap based transition compression algorithms is to perform the decompression in a dedicated hardware accelerator, the population count instruction in general purpose processor architectures can be leveraged for software implementations. Though various bitmap based compression algorithms have proposed the usage of the population count instructions, there have been no specific implementation results which have been published. So, it would be an exiting next step to implement a software centric decompression engine after performing the compression through the MSBT and the LSCT leveraging the population count instruction sets. It would be an interesting work to further ponder the performance and the storage trade-offs when the decompression function is implemented as software based implementation. These implementations should be targeted towards general purpose processor architectures used by server or client systems which implement host based DPI. Moreover, majority of the processor architectures used in the server or client platforms consists of multiple processor cores with a deep memory hierarchy. It would be an interesting future extension to implement software based decompression engines to target these platforms.

Bibliography

[1] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.*, 36(4):339–350, August 2006.

[2] M. Becchi and P. Crowley. A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim.*, 10(1):1–26, April 2013.

[3] D. Ficara, A. D. Pietro, S. Giordano, G. Procissi, F. Vitucci, and G. Antichi. Differential encoding of dfas for fast regular expression matching. *IEEE/ACM Transactions on Networking*, 19(3):683–694, June 2011.

[4] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang, and V. Prasanna. Feacan: Front-end acceleration for content-aware network processing. In *2011 Proceedings IEEE INFOCOM*, pages 2114–2122, April 2011.

[5] K. Wang, Y. Qi, Y. Xue, and J. Li. Reorganized and compact dfa for efficient regular expression matching. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5, June 2011.

[6] URL: `https://medium.com/iotforall/who-is-buying-into-iot-8f65c701b1ef`.

[7] URL: `https://www.accenture.com/t00010101T000000Z__w__/nz-en/_acnmedia/PDF-49/Accenture-Internet-Of-Things.pdf`.

[8] URL: `https://www.internetsociety.org/wp-content/uploads/2017/08/ISOC-IoT-Overview-20151221-en.pdf`.

[9] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017. `doi:10.1109/MC.2017.201`.

[10] URL: `https://www.infoworld.com/article/3176673/internet-of-things/your-smart-fridge-may-kill-you-the-dark-side-of-iot.html`.

[11] URL: `https://www.bleepingcomputer.com/news/security/about-90-percent-of-smart-tvs-vulnerable-to-remote-hacking-via-rogue-tv-signals/`.

[12] URL: `https://www.cisco.com/c/dam/m/hu_hu/campaigns/security-hub/pdf/acr-2018.pdf`.

[13] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980. doi:10.1109/TCOM.1980.1094702.

[14] URL: https://www.asus.com/my/Networking/RT-AC5300/.

[15] URL: https://www.netgear.com/home/products/networking/wifi-routers/R8500.aspx.

[16] URL: http://us.dlink.com/products/connect/ax6000-ultra-wi-fi-router-dirx6060/.

[17] URL: http://www.ieee802.org/3/cfi/0715_2/CFI_02_0715.pdf.

[18] M. S. Afaqui, E. Garcia-Villegas, and E. Lopez-Aguilera. Ieee 802.11ax: Challenges and requirements for future high efficiency wifi. *IEEE Wireless Communications*, 24(3):130–137, June 2017. doi:10.1109/MWC.2016.1600089WC.

[19] J. Park, G. Y. Kim, H. J. Park, and J. H. Kim. Ftth deployment status amp; strategy in korea: Gw-pon based ftth field trial and reach extension strategy of ftth in korea. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*, pages 1–3, Nov 2008. doi:10.1109/GLOCOM.2008.ECP.1074.

[20] W. Bo. China telecom ftth deployment - lessons learnt and future plans. In *2012 Asia Communications and Photonics Conference (ACP)*, pages 1–3, Nov 2012. doi:10.1364/ACP.2012.ATh3C.1.

[21] URL: http://www.arris.com/products/nvg578-gpon-gateways/.

[22] URL: http://www.sscqueens.org/sites/default/files/WP_Deep_Packet_Inspection_Parsons_Jan_2008.pdf.

[23] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. Gódor, G. Szabó, and T. Westholm. Deep packet inspection tools and techniques in commodity platforms: Challenges and trends. *Journal of Network and Computer Applications*, 35(6):1863 – 1878, 2012. doi:https://doi.org/10.1016/j.jnca.2012.07.010.

[24] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251327.1251336.

[25] T. H. Cheng, Y. D. Lin, Y. C. Lai, and P. C. Lin. Evasion techniques: Sneaking through your intrusion detection/prevention systems. *IEEE Communications Surveys Tutorials*, 14(4):1011–1020, Fourth 2012. doi:10.1109/SURV.2011.092311.00082.

[26] X. Yu, W. Feng, D. Yao, and M. Becchi. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–11, March 2016. `doi:10.1145/2881025.2881034`.

[27] N. F. Huang, H. W. Hung, and W. Y. Tsai. A unique-pattern based pre-filtering method for rule matching of network security. In *2012 18th Asia-Pacific Conference on Communications (APCC)*, pages 744–748, Oct 2012. `doi:10.1109/APCC.2012.6388294`.

[28] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis. Packet pre-filtering for network intrusion detection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, pages 183–192, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1185347.1185372`, `doi:10.1145/1185347.1185372`.

[29] N. Weng, L. Vespa, and B. Soewito. Deep packet pre-filtering and finite state encoding for adaptive intrusion detection system. *Comput. Netw.*, 55(8):1648–1661, June 2011. URL: `http://dx.doi.org/10.1016/j.comnet.2010.12.007`, `doi:10.1016/j.comnet.2010.12.007`.

[30] M. Nourani and P. Katta. Bloom filter accelerator for string matching. In *2007 16th International Conference on Computer Communications and Networks*, pages 185–190, Aug 2007.

[31] S. S. Shankar, L. PinXing, and A. Herkersdorf. Deep packet inspection in residential gateways and routers: Issues and challenges. In *2014 International Symposium on Integrated Circuits (ISIC)*, pages 560–563, Dec 2014.

[32] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 145–154. ACM, 2007.

[33] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, pages 93–102, 2006.

[34] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys Tutorials*, 18(4):2991–3029, Fourthquarter 2016.

[35] X. Yu, B. Lin, and M. Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10):1822–1833, Oct 2014.

[36] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabó. Design and optimizations for efficient regular expression matching in dpi systems. *Comput. Commun.*, 61(C):103–120, May 2015.

[37] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 461–472, Washington, DC, USA, 2012. IEEE Computer Society.

[38] URL: `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-sep-2018.pdf`.

[39] S. S. Shankar, P. Lin, A. Herkersdorf, and T. Wild. Hardware acceleration of signature matching through multi-layer transition bit masking. In *2016 26th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 217–224, Dec 2016.

[40] S. S. Subramanian, P. Lin, A. Herkersdorf, and T. Wild. Bitmaps & bitmasks: Efficient tools to compress deterministic automata. *Australian Journal of Telecommunications and the Digital Economy*, 6(3):41–75, sep 2018. URL: `https://doi.org/10.18080/ajtde.v6n3.125`, `doi:10.18080/ajtde.v6n3.125`.

[41] S. S. Shankar, P. Lin, A. Herkersdorf, and T. Wild. A divide and conquer state grouping method for bitmap based transition compression. In *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 400–406, Dec 2017.

[42] S. S. Shankar, P. Lin, A. Herkersdorf, and T. Wild. Bisme: A hardware coprocessor to perform signature matching at multi-gigabit rates. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 227–235, July 2018.

[43] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 262–271, 2003.

[44] R. Smith, C. Estan, and S. Jha. Xfa: Faster signature matching with extended automata. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 187–201, May 2008.

[45] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

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

[47] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. URL: `http://doi.acm.org/10.1145/359842.359859`, `doi:10.1145/359842.359859`.

[48] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, January 2004.

[49] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. URL: `http://doi.acm.org/10.1145/362686.362692`, `doi:10.1145/362686.362692`.

[50] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 1–12. ACM, 2007.

[51] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.

[52] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[53] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. URL: `http://doi.acm.org/10.1145/363347.363387`, `doi:10.1145/363347.363387`.

[54] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960. `doi:10.1109/TEC.1960.5221603`.

[55] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[56] T. Liu, A. X. Liu, J. Shi, Y. Sun, and L. Guo. Towards fast and optimal grouping of regular expressions via dfa size estimation. *IEEE Journal on Selected Areas in Communications*, 32(10):1797–1809, Oct 2014. `doi:10.1109/JSAC.2014.2358839`.

[57] Y. Xu, J. Jiang, R. Wei, Y. Song, and H. J. Chao. Tfa: A tunable finite automaton for pattern matching in network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 32(10):1810–1821, Oct 2014.

[58] C. Liu, A. Chen, D. Wu, and J. Wu. A dfa with extended character-set for fast deep packet inspection. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 1–10. IEEE Computer Society, 2011.

[59] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 25:1–25:12, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1544012.1544037`, `doi:10.1145/1544012.1544037`.

[60] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[61] URL: `https://ark.intel.com/products/52213/Intel-Core-i7-2600-Processor-8M-Cache-up-to-3-80-GHz-`.

[62] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo. An efficient regular expressions compression algorithm from a new perspective. In *2011 Proceedings IEEE INFOCOM*, pages 2129–2137, April 2011. `doi:10.1109/INFCOM.2011.5935024`.

[63] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, volume 4, pages 2628–2639 vol.4, March 2004. `doi:10.1109/INFCOM.2004.1354682`.

[64] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 191–202, 2006.

[65] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 1:1–1:10, New York, NY, USA, 2008. ACM.

[66] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 175–184, April 2009. `doi:10.1109/ISPASS.2009.4919649`.

[67] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society. URL: `https://doi.org/10.1109/FCCM.2001.22`, `doi:10.1109/FCCM.2001.22`.

[68] C. H. Lin, C. T. Huang, C. P. Jiang, and S. C. Chang. Optimization of pattern matching circuits for regular expression on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(12):1303–1310, Dec 2007. `doi:10.1109/TVLSI.2007.909801`.

[69] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, pages 249–257, Washington, DC, USA, 2004. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=1025123.1025834`.

[70] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag. URL: `http://dx.doi.org/10.1007/978-3-540-87403-4_7`, `doi:10.1007/978-3-540-87403-4_7`.

[71] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010. URL: `http://doi.acm.org/10.1145/1880153.1880157`, `doi:10.1145/1880153.1880157`.

[72] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 129–140, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2145816.2145833`, `doi:10.1145/2145816.2145833`.

[73] X. Yu and M. Becchi. Exploring different automata representations for efficient regular expression matching on gpus. *SIGPLAN Not.*, 48(8):287–288, February 2013. URL: `http://doi.acm.org/10.1145/2517327.2442548`, `doi:10.1145/2517327.2442548`.

[74] X. Yu and M. Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2482767.2482791`, `doi:10.1145/2482767.2482791`.

[75] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pages 174–183, Oct 2004. `doi:10.1109/ICNP.2004.1348108`.

[76] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In *GLOBECOM '05. IEEE Global Telecommunications Conference, 2005.*, volume 1, pages 5 pp.–, Dec 2005. `doi:10.1109/GLOCOM.2005.1577667`.

[77] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network ids/ips. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 187–196, Nov 2006. `doi:10.1109/ICNP.2006.320212`.

[78] K. Peng, S. Tang, M. Chen, and Q. Dong. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. In *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 24–35, Oct 2011. `doi:10.1109/ANCS.2011.13`.

[79] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1929820.1929831`.

[80] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng. Fast regular expression matching using small tcam. *IEEE/ACM Transactions on Networking*, 22(1):94–109, Feb 2014. `doi:10.1109/TNET.2013.2256466`.

[81] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 533–545, New York, NY, USA, 2015. ACM.

[82] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch. Hawk: Hardware support for unstructured log processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 469–480, May 2016. `doi:10.1109/ICDE.2016.7498263`.

[83] J. van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13, April 2006. `doi:10.1109/INFOCOM.2006.204`.

[84] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 112–122, Washington, DC, USA, 2005. IEEE Computer Society.

[85] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[86] URL: `http://regex.wustl.edu/`.

[87] URL: `https://www.bro.org/`.

[88] URL: `https://www.snort.org/`.

[89] X. Chen, B. Jones, M. Becchi, and T. Wolf. Picking pesky parameters: Optimizing regular expression matching in practice. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1430–1442, May 2016. `doi:10.1109/TPDS.2015.2453986`.

[90] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *2008 IEEE International Symposium on Workload Characterization*, pages 79–89, Sept 2008.

[91] B. Parhami and C.-H. Yeh. Accumulative parallel counters. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 966–970 vol.2, Oct 1995.

[92] V. Sklyarov and I. Skliarova. Design and implementation of counting networks. *Computing*, 97(6):557–577, Jun 2015. URL: `https://doi.org/10.1007/s00607-013-0360-y`, `doi:10.1007/s00607-013-0360-y`.

[93] S. J. Piestrak. Efficient hamming weight comparators of binary vectors. *Electronics Letters*, 43(11):611–612, May 2007. `doi:10.1049/el:20070141`.

[94] V. A. Pedroni. Compact hamming-comparator-based rank order filter for digital vlsi and fpga implementations. In *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, volume 2, pages II–585–8 Vol.2, May 2004. `doi:10.1109/ISCAS.2004.1329339`.

[95] B. Parhami. Efficient hamming weight comparators for binary vectors based on accumulative and up/down parallel counters. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(2):167–171, Feb 2009.

[96] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, Feb 1964.

[97] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. *IEEE/ACM Trans. Netw.*, 13(3):690–703, June 2005.

[98] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 50–59, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1477942.1477950`, `doi:10.1145/1477942.1477950`.

[99] URL: `http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf`.

[100] W. D. Schwaderer. *Introduction to Open Core Protocol: Fastpath to System-on-Chip Design.* Springer, 2012.

[101] URL: `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`.

[102] URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-xp.html.

[103] URL: http://bittwist.sourceforge.net/.

[104] URL: https://www.spirent.com/Products/TestCenter/Platforms.

[105] URL: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/speedbridge-adapter-multi-ethernet-ds.pdf.

[106] URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[107] URL: https://support-kb.spirent.com/infocenter/index?page=content&id=FAQ11176&cat=SOFTWARE&actp=LIST.

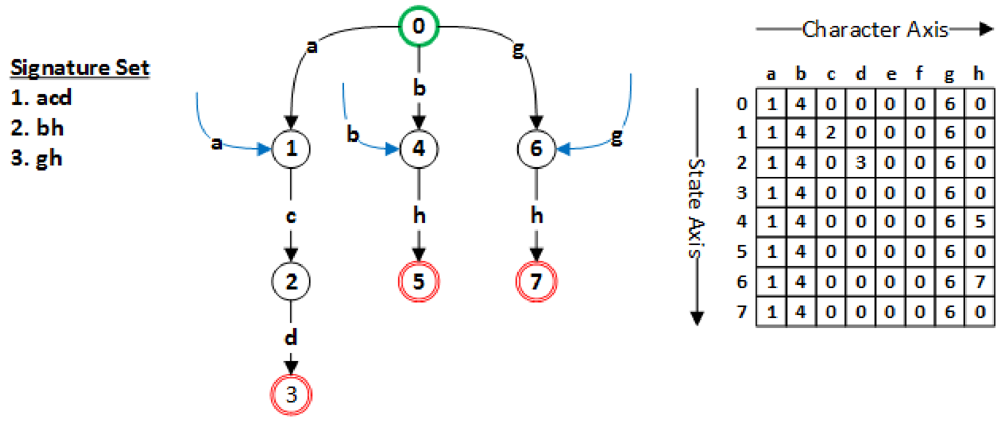
[108] URL: https://www.wireshark.org/.

A Trace Generation

This section provides a short description of the pseudorandom trace generation methodology proposed in [90]. In this method, the sequence of payload bytes are generated by traversing through an automata as described further below. Figure A.1 shows the DFA corresponding to the signature set acd, bh and gh. The characters used in the signature set belongs to the alphabet $\Sigma=\{a,b,c,d,e,f,g,h\}$.

The trace generation methodology takes the DFA (or the NFA) as an input and generates a sequence of characters based on the probability of maliciousness P_M, which is chosen between 0 and 1. The lower the value chosen for the P_M, the lower the number of signatures resulting in the generated sequence of payload bytes. Table A.1 describes an example of the trace generation process corresponding to the DFA shown in Figure A.1.The P_M value chosen for this exercise is 0.35.

The trace generation process starts with the root state being set as the current state. As discussed earlier in Chapter 2, the state transitions in the DFA are split into forward and the failure transitions. The forward transitions are those state transitions which lead the state machine towards an accepting state, while the failure transitions are those which don't. As part of the trace generation process, a character is chosen among all the characters in the alphabet in such a way that the probability of the chosen character resulting in a forward transition is P_M. For example, as seen in Table A.1, when the current state is '0', among all the characters in the alphabet, the characters a, b and g alone lead the state machine towards a plausible signature match. So, the probability of one of these characters being chosen in the trace generation process is P_M=0.35. In the



	a	b	c	d	e	f	g	h
0	1	4	0	0	0	0	6	0
1	1	4	2	0	0	0	6	0
2	1	4	0	3	0	0	6	0
3	1	4	0	0	0	0	6	0
4	1	4	0	0	0	0	6	5
5	1	4	0	0	0	0	6	0
6	1	4	0	0	0	0	6	7
7	1	4	0	0	0	0	6	0

Figure A.1: Example DFA to explain the trace generation mechanism

Table A.1: Sequence of characters generated with P_M=0.35

Current_State	0	0	4	0	1	2
Characters which generate a forward transition	a,b,g	a,b,g	h	a,b,g	c	d
Characters which generate a failure transition	c,d,e, f,h	c,d,e, f,h	a,b,c,d, e,f,g	c,d,e f,h	a,b,d,e, f,g,h	a,b,c,e, f,g,h
Generated character	c	b	d	a	c	a
Next_State	0	4	0	1	2	0

specific example being considered, the character which is generated is ‘c’ which results in the next state ‘0’. Subsequently, the next state is assigned as the current state and the next character in the trace is generated in a similar fashion as explained above. Following this process, the sequence of characters generated specific to this example are *cbdaca*. In this way, by generating the sequence of characters in a pseudorandom fashion, it is made sure that the traces which are generated for payload inspection do contain a certain number of signatures and the number of signatures generated is controlled through the P_M value. A higher value set to P_M will result in traces with higher signature count in comparison to a lower value.

B Accumulative Parallel Adder

As discussed in Chapter 5, the population count is a critical operation which is performed as part of the decompression process. So, it is essential to implement an efficient hardware block to perform the population count function. Existing literature on population count implementations have identified that the Accumulative Parallel Adder (APA) [91] is the most effective way to implement the population count function [95]. A short description of the APA is described in this section.

The population count is the process of counting the total number of set bits in an N-bit vector. The APA consists of a tree of increasing wider ripple carry adders. The first level consists of $\log_2 N$ full adders while the last level consists of a single $\log_2\mathrm{N}$ wide ripple carry adder. Through each successive level, the width of the ripple carry adders increases and the number of ripple carry adders reduces in a logarithmic fashion. The worst case latency of the parallel adder circuit is $2*\log_2\mathrm{N}-1$ full adder delays and is also the critical path in the combinatorial computation. The APA is also capable of adding a $log_2\mathrm{N}$ wide initial value which is added to the output of the APA. In the context of the decompression engine, the initial value can be the base address to be added to the output of the population count operation performed on the bitmap or the bitmask.

Figure B.1 shows an example of the parallel adder circuit for N set to 16 together with a 4 bit initial value. As seen in Figure B.1, the first level of the adder tree consists of 4 full adders. For every subsequent level, the width of the ripple carry adders increases with the last level consisting of a 4 bit ripple carry adder. The circuit finally outputs a 4 bit sum with a single bit carry which detects an overflow in the addition operation. Thus, the proposed circuit performs the population count operation and simultaneously adds the output to an initial value. In the case of BiSME and BOBiSME implementations, a 256 bit APA is used which results in a worst case processing latency of 15 full adder delays.

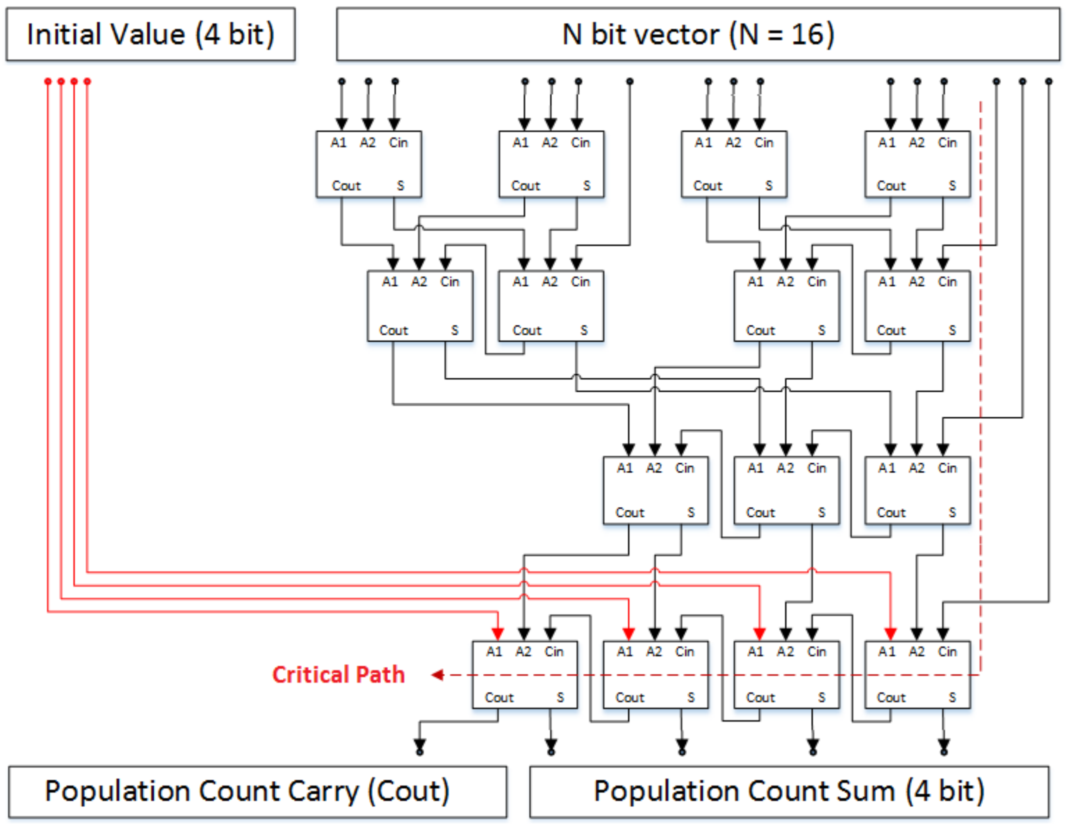


Figure B.1: Example of the Accumulative Parallel Adder circuitry to perform the population count function