การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี BITMAP INTERSECTION LOOKUP (B'L)

BITMAP INTERSECTION LOOKUP (B'L) : A PACKET CLASSIFICATION'S ALGORITHM WITH RULES UPDATING

นัฐโชติ พรหมฤทธิ์

NUTTACHOT PROMRIT

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิชาเทคโนโลยีสารสนเทศ

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2548

ISBN 974-15-1782-9

การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี BITMAP INTERSECTION LOOKUP (BIL)

BITMAP INTERSECTION LOOKUP (BIL) : A PACKET CLASSIFICATION 'S ALGORITHM WITH RULES UPDATING

ณัฐโชติ พรหมฤทธิ์

NUTTACHOT PROMRIT

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาเทคโนโลยีสารสนเทศ
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2548

ISBN 974-15-1762-9

BITMAP INTERSECTION LOOKUP (BIL) : A PACKET CLASSIFICATION 'S ALGORITHM WITH RULES UPDATING

NUTTACHOT PROMRIT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2005

ISBN 974-15-1762-9

หัวข้อวิทยานิพนธ์	การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL)
นักศึกษา	นายณัฐโชติ พรหมฤทธิ์
รหัสนักศึกษา	43067012
ปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา	เทคโนโลยีสารสนเทศ
พ.ศ.	2548
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ. อัครินทร์ คุณกิตติ

บทคัดย่อ

การจัดประเภทแพ็กเก็ตตามกฎ (Packet Classification) เป็นกลไกที่สำคัญของแอพพลิเคชัน เช่น ไฟร์วอล และอินเตอร์เน็ตเร้าเตอร์ สำหรับการกำหนดกระแส (Flow) ของ Packet บน IP Network ซึ่งต้องปฏิบัติงานบนช่องทางการสื่อสารที่มีความเร็วสูง ด้วยขนาดของกฎที่เพิ่มขึ้น งานวิจัยนี้จึงนำเสนอการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup ที่มีอัลกอริทึมและโครงสร้างของกฎที่ง่ายไม่ซับซ้อน ทำให้ 1. มีความเร็วในการค้นหาสูง 2. มีความเร็วในการปรับปรุงกฎที่ดี และ 3. ใช้เนื้อที่จัดเก็บข้อมูลต่ำ โดยจะจัดโครงสร้างของกฎลงใน BIL tables ซึ่งการปรับปรุงกฎจะนำค่าของกฎที่มีลักษณะเป็นช่วง (Range) หรือมาสก์ (Mask) ทำการแปลงค่าให้อยู่ในรูปของ Prefix และแบ่ง Prefix เป็น Block ขนาดเล็กๆ เพื่อใช้สำหรับ Set Bitvector ในตาราง BIL table สำหรับการค้นหาจะแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block เท่ากับขนาดของ Prefix ที่ถูกแบ่ง นำแต่ละ Block ไป Lookup ใน BIL table และนำผลลัพธ์ที่ได้จากการ Lookup คือ Bitvector จากแต่ละตาราง ทำการ Intersection ด้วยตรรก AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์

นอกจากนั้นงานวิจัยนี้ยังนำเสนอตัววัดประสิทธิภาพของวิธีการของการจัดประเภทแพ็กเก็ต เพื่อใช้เป็นเครื่องมือสำหรับการศึกษารูปแบบการแบ่ง Prefix ที่ทำให้โปรแกรมมีประสิทธิภาพการทำงานดีที่สุด และเปรียบเทียบประสิทธิภาพกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection จากการศึกษาพบว่า 1. การแบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน หรือใกล้เคียงกันจะทำให้โปรแกรมมีประสิทธิภาพในการทำงานดีกว่าการแบ่ง Prefix เป็น Block ที่มีขนาดแตกต่างกัน 2. การแบ่ง Prefix เป็น Block ขนาดเล็กจะมีผลดีในด้านการปรับปรุงกฎสำหรับกฎที่มีช่วง (Range) หรือมาสก์ (Mask) กว้าง และด้านเนื้อที่จัดเก็บข้อมูล และ 3. การแบ่ง Prefix เป็น Block ขนาดใหญ่จะมีผลดีในด้านการค้นหา และการปรับปรุงกฎสำหรับกฎที่มีลักษณะไม่เป็นช่วง โดยเมื่อพิจารณาประสิทธิภาพทั้ง 3 ด้าน พบว่าโปรแกรม BIL จะมีประสิทธิภาพในการทำงานโดยรวมดีกว่าโปรแกรม Linear Search และโปรแกรม Bitmap-intersection

Thesis Title	Bitmap Intersection Lookup (BIL) : A Packet Classification 's Algorithm with Rules Updating
Student	Mr.Nuttachot Promrit
Student ID.	43067012
Degree	Master of Science
Programme	Information Technology
Year	2548
Thesis Advisor	Assist. Prof. Akharin Khunkitti

ABSTRACT

A packet classification is main mechanism, which has important for applications such as Firewall and Internet router. It determine the flow of the packet on IP networks. The problem is performing scalable packet classification at wired speed even as rule database increases size. So this research present "Bitmap Intersection Lookup (BIL): a packet classification 's algorithm with rules updating." This approach uses simple algorithm and rule structure. It makes classifier have 1) high searching speed 2) fast updating and 3) low storage space, by rearrange rule structure into BIL tables. Rule will be defined as range or mask and convert into Prefix then divides Prefix into smaller block. BIL algorithm divides header of packet into smaller block that has size equal to the size of Prefix block, then take the value of each block to lookup in BIL table. The result called Bitvectors from BIL table that intersection with AND logic then the algorithm priority chooses the most important rule.

In addition this research present indicators for study working performance of packet classification 's algorithm. For study, format of Prefix dividing it affects the program's performance. Then we choose the patterns of Prefix, which makes best performance. When comparing performance among Linear Search program, Bitmap Intersection program and purposed BIL program. The results of the study are 1) If Prefix is divided into equal block, it makes more performance than the different size of block 2) If Prefix is divided into small block, it has high speed to update and low storage space for rule database when range or mask are wide 3) If Prefix is divided into large block, it has high speed to search and update for rule database when range or mask are narrow. When consider three balance working performance of programs, BIL program has more performance than Linear Search program and Bitmap-intersection program.

กิตติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษา ผศ. อัครินทร์ คุณกิตติ ที่ให้ความช่วยเหลือ ให้คำชี้แนะช่วยแก้ปัญหา ตลอดจนให้ความรู้และประสบการณ์ที่ดีแก่ข้าพเจ้า

ขอขอบพระคุณครูอาจารย์ทุกท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้

ขอขอบพระคุณครอบครัว และเพื่อนๆ ที่น่ารัก ที่คอยให้กำลังใจเสมอมา

สำหรับคุณงามความดีอันใดที่เกิดจากวิทยานิพนธ์เล่มนี้ ข้าพเจ้าขอมอบให้กับบิดามารดาผู้ให้สติปัญญา การอบรมสั่งสอน คอยส่งเสริมทั้งทางด้านทุนทรัพย์ และกำลังใจในการศึกษาแก่ข้าพเจ้า

ณัฐโชติ พรหมฤทธิ์

สารบัญ

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

สารบัญตาราง

สารบัญตาราง (ต่อ)

สารบัญตาราง (ต่อ)

สารบัญรูป

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ปัจจุบันการจัดประเภทแพ็กเก็ตตามกฎ (Packet Classification) เป็นเครื่องมือที่สำคัญของแอพพลิเคชัน เช่น ไฟร์วอล และอินเตอร์เน็ตเร้าเตอร์ สำหรับการกำหนดกระแส (Flow) ของแพ็กเก็ต โดยการจัดประเภทแพ็กเก็ตของแอพพลิเคชันเหล่านั้นจะทำการตรวจสอบแพ็กเก็ตที่มีลักษณะตรงกับเงื่อนไขที่กำหนดและเลือกกฎที่มีลำดับความสำคัญสูงที่สุด ซึ่งแม้ว่าอินเตอร์เน็ตจะเป็นเครือข่ายแบบแพ็กเก็ตสวิทที่ให้บริการแบบ Best-effort คือไม่ได้มีการรับประกันอัตราการส่งใดๆ เลย เพียงแต่ทำให้ดีที่สุดเท่านั้น แต่ไอพีเน็ตเวิร์คก็สามารถให้บริการแก่ผู้ใช้ที่หลากหลายเช่น Quality of Services, Virtual Private Network (VPN) Services, Distribute Firewall และ IP Security Gateways เป็นต้น [2] ซึ่งการปรับปรุงกฎจะกระทำโดยผู้ดูแลระบบ หรือโดยโปรโตคอลก็ได้ ดังนั้นปัญหาของการจัดประเภทแพ็กเก็ตตามกฎ คือจะต้องปฏิบัติงานบนช่องทางการสื่อสารที่มีความเร็วสูง ด้วยกฎที่มีขนาดใหญ่

จากความสำคัญดังกล่าว จึงมีงานวิจัยทางด้านการจัดประเภทแพ็กเก็ตตามกฎ เช่น [9], [4] ซึ่งมีความเร็วในการค้นหาที่สูง คือ $O\left(D\cdot\log N+\frac{N}{WORD}\right)$ และ O(1) บนฮาร์ดแวร์ (เมื่อ N เป็นจำนวนกฎ) แต่การจัดประเภทแพ็กเก็ตตามกฎเหล่านี้ยังมีปัญหา คือ

1. วิธีการ Bitmap-intersection จะใช้เนื้อที่จัดเก็บข้อมูล คือ $O(D\cdot N^2)$ ซึ่งจะทำให้สิ้นเปลืองเนื้อที่มากสำหรับการจัดประเภทแพ็กเก็ตที่มีกฎจำนวนมาก [9]

2. วิธีการ Ternary CAM ต้องทำการพัฒนาด้วยฮาร์ดแวร์เท่านั้น โดยจะใช้เนื้อที่จัดเก็บข้อมูลเท่ากับ O(N) แต่วิธี Ternary CAM จะใช้หน่วยความจำที่มีจำนวน Transistor มากกว่าหน่วยความจำทั่วๆ ไป จึงทำให้สิ้นเปลืองพลังงานมากกว่า [4]

ดังนั้นในงานวิจัยนี้จึงได้นำเสนอวิธีการการจัดประเภทแพ็กเก็ตตามกฎที่มีความเร็วในการค้นหาที่สูง คือ $O\left(\frac{D\cdot W}{PMX}\cdot\frac{N}{WORD}\right)$ บนซอฟท์แวร์ และ O(1) บนฮาร์ดแวร์ มีเวลาในการปรับปรุงกฎที่ดี และใช้เนื้อที่จัดเก็บข้อมูลเท่ากับ $O\left(\frac{D\cdot W}{PMX}\cdot 2^{PMX}\cdot N\right)$

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

1. เพื่อศึกษาลักษณะของการจัดประเภทแพ็กเก็ตตามกฎ
2. เพื่อศึกษาประสิทธิภาพของการจัดประเภทแพ็กเก็ตตามกฎแต่ละชนิด

3. เพื่อนำเสนอวิธีการของการจัดประเภทแพ็กเก็ตตามกฎที่มีประสิทธิภาพในการค้นหา การปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล

1.3 ขอบเขตของการวิจัย

งานวิจัยในครั้งนี้ผู้วิจัยต้องการนำเสนอวิธีการของการจัดประเภทแพ็กเก็ตตามกฎที่มีประสิทธิภาพทั้งในด้านเวลาการค้นหา เวลาการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล โดยจะต้องสามารถปฏิบัติงานกับฐานข้อมูลกฎขนาดใหญ่ได้ ดังนั้นเพื่อทดสอบแนวคิดที่กล่าวมาข้างต้น จึงกำหนดให้ระบบที่ทำการศึกษาเป็นระบบที่พัฒนาขึ้นด้วยตัวแปรภาษา gcc บนระบบปฏิบัติการ Linux Kernel 2.4 และในการวัดประสิทธิภาพจะวัดจากความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล ซึ่งการทดลองจะประกอบด้วยฐานข้อมูล 2 ชนิด คือ ฐานข้อมูลกฎสำหรับการปรับปรุงกฎ และฐานข้อมูลแพ็กเก็ตสำหรับการจัดประเภทแพ็กเก็ต (การค้นหา) ที่สร้างขึ้นมาเองเพื่อให้เหมาะสมกับจุดประสงค์ของการทดลองแต่ละการทดลอง และจะทำการเปรียบเทียบประสิทธิภาพการทำงานในด้านต่างๆ ดังที่กล่าวมาข้างต้นกับวิธีการ Linear search และวิธีการ Bitmap-intersection ที่จำนวนกฎ 1,000-10,000 ข้อ

1.4 ขั้นตอนของการศึกษา

1. ศึกษาหลักการและทฤษฎีของการจัดประเภทแพ็กเก็ตตามกฎ
2. ออกแบบอัลกอริทึมสำหรับการจัดประเภทแพ็กเก็ตตามกฎ
3. พัฒนาโปรแกรมการจัดประเภทแพ็กเก็ตตามกฎ
4. ทำการทดลอง
5. ทดสอบและแก้ไข
6. ประเมินผล หาข้อสรุป และข้อเสนอแนะ
7. จักทำเอกสาร

1.5 รายละเอียดของแต่ละบท

ในส่วนเนื้อหาของวิทยานิพนธ์ฉบับนี้จะประกอบด้วยบทต่างๆ ดังนี้

บทที่ 1 บทนำ กล่าวถึงความเป็นมาของการจัดประเภทแพ็กเก็ตตามกฎ วัตถุประสงค์ของการศึกษา ขอบเขตของงานวิจัย รวมถึงขั้นตอนของการศึกษา

บทที่ 2 ระบบอินเตอร์เน็ตและการจัดประเภทแพ็กเก็ต จะกล่าวถึงอินเตอร์เน็ตและสถาปัตยกรรมทีซีพี/ไอพี รูปแบบของเฮดเดอร์ในโปรโตคอลทีซีพี/ไอพี และอีเทอร์เน็ตเฟรม

ความหมายของการจัดประเภทแพ็กเก็ตตามกฎ ความต้องการของการจัดประเภทแพ็กเก็ตตามกฎ การกำหนดค่าของกฎให้อยู่ในรูป Prefix และตัวอย่างของวิธีการจัดประเภทแพ็กเก็ตตามกฎ

บทที่ 3 การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) ซึ่งในบทนี้จะกล่าวถึงแนวคิดและสถาปัตยกรรมของ BIL การวิเคราะห์ประสิทธิภาพเบื้องต้น และการพัฒนาโปรแกรม

บทที่ 4 การทดลองและผลการทดลอง เป็นการทดสอบการทำงานของโปรแกรม เพื่อแสดงให้เห็นประสิทธิภาพในด้านต่างๆ เช่น การค้นหา การปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูล ทำการวิเคราะห์ผลลัพธ์ที่ได้จากการทดลอง เพื่อหาขนาดของ Prefix ที่เหมาะสม และเปรียบเทียบประสิทธิภาพกับวิธีการอื่นๆ

บทที่ 5 สรุปผลการวิจัยและข้อเสนอแนะ กล่าวถึงผลสรุปของการวิจัย ข้อเสนอแนะ และแนวทางการทำวิจัยต่อไป

บทที่ 2
ระบบอินเตอร์เน็ตและการจัดประเภทแพ็กเก็ต

2.1 อินเตอร์เน็ตและสถาปัตยกรรมทีซีพี/ไอพี

เครือข่ายอินเตอร์เน็ตเป็นเครือข่ายสาธารณะที่มีการเชื่อมต่อกับอุปกรณ์ประมวลผลเป็นล้านๆ เครื่องทั่วโลกเข้าไว้ด้วยกัน ซึ่งอุปกรณ์ประมวลผลเหล่านี้ส่วนใหญ่เป็นเครื่องคอมพิวเตอร์ทั้งแบบส่วนบุคคลและเครื่องให้บริการ (Server) และอาจรวมทั้งโทรทัศน์ที่เชื่อมต่อเข้าสู่เครือข่ายอินเตอร์เน็ต และโทรศัพท์มือถือ เป็นต้น อุปกรณ์ต่างๆ เหล่านี้ถูกเรียกว่าโฮสท์หรือ End System ซึ่งโปรแกรมที่ใช้งานบนอินเตอร์เน็ต เช่น เว็บเพจ และจดหมายอิเล็กทรอนิกส์นั้นมีการทำงานอยู่บน End System

อุปกรณ์ต่างๆที่อยู่บนเครือข่ายอินเตอร์เน็ตจะติดต่อกันโดยใช้ข้อตกลงในการสื่อสารข้อมูลร่วมกันเพื่อควบคุมการส่งและรับข้อมูลด้วยโปรโตคอลทีซีพี/ไอพี

End System จะถูกเชื่อมต่อเข้าด้วยกันโดยช่องทางการสื่อสาร (Communication Links) ซึ่งประสิทธิภาพของช่องทางการสื่อสารมักเรียกว่า Link Bandwidth ซึ่งมีหน่วยวัดเป็นบิตต่อวินาที แต่ปกติแล้ว End System จะไม่ถูกเชื่อมต่อเข้าด้วยกันโดยตรงผ่านช่องทางการสื่อสารเพียงเส้นเดียว แต่จะถูกเชื่อมต่อเข้าด้วยกันโดยผ่านทางเร้าเตอร์ โดยเร้าเตอร์จะทำการรับข้อมูลที่ส่งเข้ามาผ่านทางช่องทางการสื่อสารขาเข้าแล้วส่งผ่านข้อมูลไปยังช่องทางขาออก ในขณะที่ไอพีโปรโตคอลจะทำการกำหนดรูปแบบของข้อมูล และควบคุมการส่งและรับข้อมูลระหว่างเร้าเตอร์กับ End System ซึ่งเทคนิคที่ใช้ในการสื่อสารระหว่าง End System นั้นเรียกว่าแพ็กเก็ตสวิทชิง ซึ่งเป็นเทคนิคที่ยอมให้ End System ต่างๆ ร่วมกันใช้เส้นทางหรือใช้บางส่วนของเส้นทางได้ในเวลาเดียวกัน

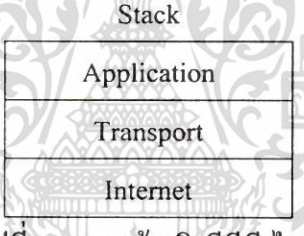
ในเครือข่ายอินเตอร์เน็ตนั้น แอพพลิเคชันที่ทำงานอยู่บน End System สามารถแลกเปลี่ยนข้อมูลระหว่างกัน ได้โดยเลือกใช้บริการที่มีอยู่ 2 แบบด้วยกัน คือแบบ Connection-Oriented Service และแบบ Connectionless Service

ทีซีพี/ไอพีจึงเป็นโปรโตคอลที่ได้รับความนิยมอย่างแพร่หลายเนื่องจากความสามารถในการเชื่อมต่อเครือข่ายที่ประกอบด้วยฮาร์ดแวร์และซอฟต์แวร์ต่างชนิดกันได้อย่างกลมกลืน โดยเครือข่ายคอมพิวเตอร์มีแบบอ้างอิงที่ใช้เป็นมาตรฐานคือ แบบอ้างอิงโอเอสไอ (OSI : Open Systems Interconnection Reference Model) ในขณะที่ทีซีพี/ไอพีเป็นโปรโตคอลที่กำเนิดก่อนโอเอสไอ และมีแบบอ้างอิงเฉพาะตามรูปที่ 2.1 โดยแต่ละชั้นจะมีคุณสมบัติต่างๆ ดังต่อไปนี้

ชั้นแอพพลิเคชัน เป็นระดับชั้นที่มีหน้าที่สนับสนุนแอพพลิเคชันของเครือข่าย โดยในชั้นแอพพลิเคชันจะมีอยู่หลายโปรโตคอล เช่น FTP ทำการสนับสนุนการถ่ายโอนไฟล์ SMTP ทำการสนับสนุนจดหมายอิเล็กทรอนิกส์ หรือ HTTP ทำการสนับสนุนเว็บ เป็นต้น

ชั้นทรานสปอร์ต ทำหน้าที่ส่ง Message ของชั้นแอพพลิเคชันระหว่าง Client และ Server โดยในชั้นทรานสปอร์ตนั้นมีโปรโตคอลที่ใช้งานอยู่ 2 โปรโตคอลด้วยกัน คือ TCP และ UDP โดย TCP เป็น Connection-Oriented Service ที่รับประกันว่า Message ที่ถูกส่งไปยังปลายทางนั้นจะถูกส่งอย่างถูกต้องครบถ้วนและตรงตามลำดับ และมี Flow-control คือจะควบคุมให้อัตราการส่งข้อมูลของฝั่งส่งไม่เกินความสามารถในการรับข้อมูลของฝั่งรับ นอกจากนี้ยังทำการแบ่ง Message ที่ยาวๆ ออกเป็น segment ที่เล็กลง และมีการควบคุมความคับคั่งของข้อมูลด้วย ส่วน UDP นั้นเป็น Connectionless Service ซึ่งเป็นบริการที่ทำการส่ง Message ไปให้ปลายทางอย่างเดียวไม่มีการรับรองว่าข้อมูลจะไปถึงปลายทางจริงหรือไม่

ชั้นอินเตอร์เน็ต ทำหน้าที่เลือกเส้นทางเพื่อส่งเดตาแกรมหรือแพ็กเก็ตจากโฮสท์หนึ่งไปยังอีกโฮสท์หนึ่ง [1]

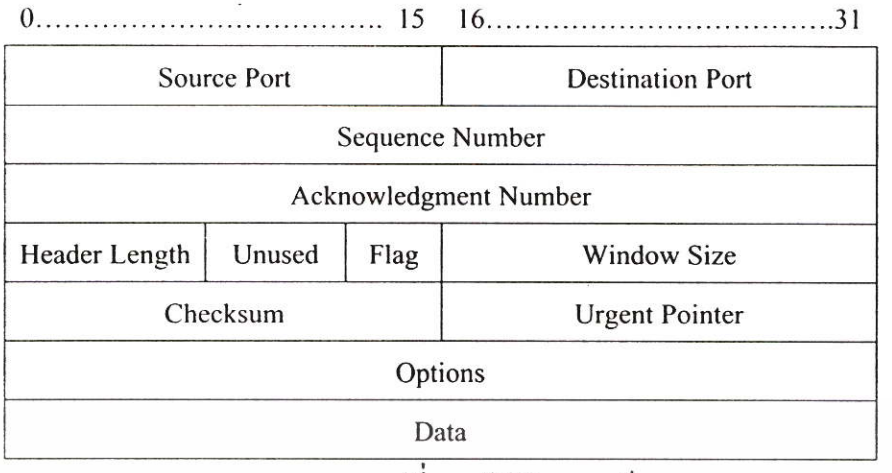


รูปที่ 2.1 แบบอ้างอิงทีซีพี/ไอพี

2.2 รูปแบบของเฮดเดอร์ในโปรโตคอลทีซีพี/ไอพี และอีเทอร์เน็ตเฟรม

2.2.1 ทีซีพีเฮดเดอร์

ประกอบด้วยฟิลด์จำนวนมากทำหน้าที่ให้บริการตามหน้าที่ที่กล่าวข้างต้น รูปที่ 2.2 แสดงเฮดเดอร์ของโปรโตคอลทีซีพี โดยแต่ละฟิลด์มีความหมายดังต่อไปนี้



รูปที่ 2.2 ทีซีพีเฮดเดอร์

Source Port (ขนาด 16 บิต) บอกหมายเลขพอร์ตต้นทาง

Destination Port (ขนาด 16 บิต) บอกหมายเลขพอร์ตปลายทาง

Sequence Number และ Acknowledgment Number (ขนาด 32 บิต) ซึ่งจะถูกใช้โดย TCP Sender และ TCP Receiver เพื่อตรวจสอบความถูกต้องของข้อมูล

Header Length (ขนาด 4 บิต) บอกขนาดของทีซีพีเฮดเดอร์ทั้งหมด

Flag (ขนาด 6 บิต) คือ ACK บอกว่าเป็น segment ตอบรับหรือไม่, RST, SYN และ FIN ใช้สร้างการเชื่อมต่อและยกเลิกการเชื่อมต่อ, PSH เมื่อถูกกำหนดจะเป็นการบอกให้ผู้รับส่งข้อมูลไปยังชั้นบนทันที และ URG ใช้บอกว่าข้อมูลใน Segment นี้ด่วนที่สุด (Urgent) เมื่อทีซีพีฝั่งรับได้รับก็จะบอกชั้นบนทันทีว่ามีขอมูลด่วนเข้ามาและส่งค่า Pointer ที่ชี้ไปยังข้อมูลด่วนนั้นให้ด้วย (16 บิต Urgent Pointer)

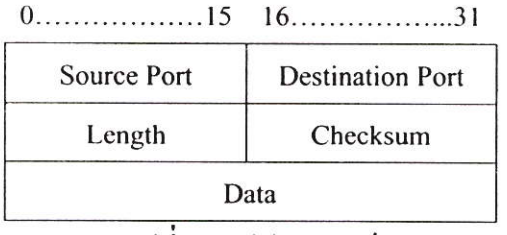
Windows Size (ขนาด 16 บิต) สำหรับกำหนด Flow Control และบอกจำนวนไบต์ ที่ฝั่งรับสามารถรับได้

Checksum (ขนาด 16 บิต) ผลรวมตรวจสอบ คำนวณจากผลรวมของเฮดเดอร์และข้อมูล

Option (ขนาดไม่คงที่) จะถูกใช้เมื่อฝ่ายรับทำการเจรจาขนาดของ Segment มากที่สุด (MSS) และกรณีอื่นๆ

2.2.2 ยูดีพีเฮดเดอร์

ประกอบด้วยฟิลด์ที่ให้บริการตามหน้าที่ที่กล่าวข้างต้น รูปที่ 2.3 แสดงเฮดเดอร์ของโปรโตคอลยูดีพี โดยแต่ละฟิลด์มีความหมายดังต่อไปนี้



รูปที่ 2.3 ยูดีพีเฮดเดอร์

Source Port (ขนาด 16 บิต) บอกหมายเลขพอร์ตต้นทาง

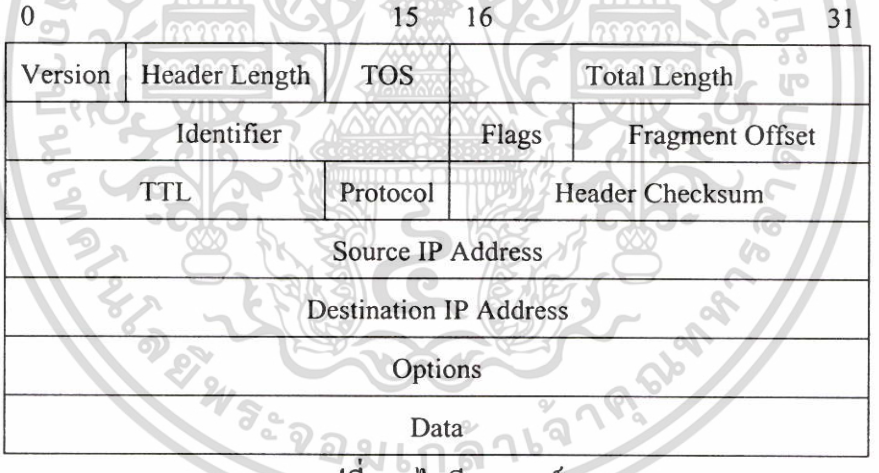
Destination Port (ขนาด 16 บิต) บอกหมายเลขพอร์ตปลายทาง

Length (ขนาด 16 บิต) บอกความยาวทั้งเฮดเดอร์และข้อมูล

Checksum (ขนาด 16 บิต) ผลรวมตรวจสอบ คำนวณจากผลรวมของเฮดเดอร์และข้อมูล

2.2.3 ไอพีเฮดเดอร์

ไอพีเวอร์ชันที่ 4 ประกอบด้วยฟิลด์ต่างๆ ที่แสดงไว้ในรูปที่ 2.4



รูปที่ 2.4 ไอพีเฮดเดอร์

Version (ขนาด 4 บิต) แสดงถึงเวอร์ชันของโปรโตคอลไอพีของเดตาแกรม เพื่อให้เร้าเตอร์สามารถแปรความหมายส่วนที่เหลือของเดตาแกรมได้

Header Length (ขนาด 4 บิต) บอกความยาวเฉพาะเฮดเดอร์ของเดตาแกรมโดยนับจาก Version ไปจนถึงไบต์สุดท้ายก่อนจะถึงข้อมูล หน่วยนับความยาวจะบอกเป็นจำนวนเท่าของ 4 ไบต์ เช่น หาก Header Length มีขนาดเท่ากับ 5 จะหมายถึงส่วนหัวมีขนาดเท่ากับ 20 ไบต์ ซึ่งเป็นค่าที่บอกว่าไม่มี Option อยู่ในเดตาแกรม

TOS (Type of Service มีขนาด 4 บิต) ใช้บอกชนิดของบริการที่เดตาแกรมนั้นต้องการ เช่นระบุว่าเดตาแกรมนี้เป็น Real Time Datagram ซึ่งจะต้องทำงานร่วมกันกับระบบของเร้าเตอร์

Total Length (ขนาด 16 บิต) คือ ความยาวรวมของเดตาแกรม มีขนาดสูงสุด คือ 65,535 ไบต์ แต่โดยส่วนใหญ่แล้วจะมีขนาดไม่เกิน 1,500 ไบต์

Identifier, Flags และ Fragment Offset (ขนาด 16 บิต, 3 บิต และ 16 บิต) ใช้ในระบบ IP Fragmentation เพื่อให้เหมาะสมกับเฟรมในระดับเดตาลิงค์

TTL (Time of Live ขนาด 8 บิต) ใช้กำหนดจำนวนเร้าเตอร์ที่เดตาแกรมจะเดินทางผ่านได้ ซึ่งมีค่าสูงสุดเท่ากับ 255 โดยเร้าเตอร์จะลดค่านี้ลงทีละ 1 ในแต่ละครั้งที่เร้าเตอร์รับเดตาแกรมเข้ามา

Protocol (ขนาด 8 บิต) บอกชนิดของโปรโตคอลในระดับบน โดยฟิลด์นี้จะใช้เมื่อไอพีเดตาแกรมมาถึงปลายทางแล้วเท่านั้น

Header Checksum (ขนาด 16 บิต) ใช้ตรวจสอบความผิดพลาดเฉพาะเฮดเดอร์โดยไม่รวมส่วนข้อมูล โดยเร้าเตอร์จะคำนวณ Checksum และบรรจุกลับเข้าไปใหม่ทุกครั้งที่ได้รับไอพีเดตาแกรม ซึ่งปกติแล้วเร้าเตอร์จะไม่รับเดตาแกรมที่ผิดปกติ

Source IP Address (ขนาด 32 บิต) กำหนดไอพีแอดเดรสต้นทาง

Destination IP Address (ขนาด 32 บิต) กำหนดไอพีแอดเดรสปลายทาง

Options (ขนาดไม่คงที่) ใช้สำหรับกำหนดข่าวสารเพิ่มเติมสำหรับเดตาแกรม ค่าที่ใช้ในปัจจุบันจะเกี่ยวข้องกับการรักษาความปลอดภัย และการบันทึกผลลัพธ์จากการทำงานของคำสั่ง traceroute หรือ ping

Data เป็นข้อมูลของโปรโตคอลในระดับบน เช่น TCP และ UDP หรือข้อมูลในรูปแบบอื่นๆ เช่น ICMP Message

2.2.4 อีเทอร์เน็ตเฟรม

เทคโนโลยีอีเทอร์เน็ตทุกๆ ตัวไม่ว่าจะใช้ Coaxial, Copper หรือจะทำงานที่ 10 Mbps, 100 Mbps หรือ 1 Gbps จะใช้โครงสร้างของเฟรมของอีเทอร์เน็ตเหมือนกัน ดังรูปที่ 2.5

Preamble	Destination Address	Source Address	Type	Data	CRC

รูปที่ 2.5 อีเทอร์เน็ตเฟรม

Preamble (ขนาด 8 ไบต์) อีเทอร์เน็ตเฟรมจะเริ่มด้วยฟิลด์ Preamble โดยที่แต่ละไบต์ของ 7 ไบต์แรกใน Preamble จะมีค่าเป็น 10101010 ส่วนไบต์สุดท้ายจะเป็น 10101011 ซึ่ง 7 ไบต์แรกของ Preamble เป็นการปลุก Adapter ของผู้รับและเป็นการเข้าจังหวะสัญญาณนาฬิการะหว่างผู้รับ

และผู้ส่งด้วย และ 2 บิตสุดท้ายของไบต์ที่ 8 ของ Preamble จะเตือนให้ Adapter ผู้รับรู้ว่าส่วนสำคัญของข้อมูลซึ่งก็คือเฮดเดอร์ และข้อมูล (Data) กำลังจะมา

Destination Address (ขนาด 6 ไบต์) ใช้เก็บ MAC Address ของ Destination Adapter สมมติว่า Adapter ของโฮสท์ B มี MAC Address เป็น BB-BB-BB-BB-BB-BB เมื่อ Adapter B ได้รับอีเทอร์เน็ตเฟรมที่มี Destination Address เป็นอย่างอื่นไม่ใช่ BB-BB-BB-BB-BB-BB หรือไม่ใช่แอดเดรสบรอดแคสท์ มันจะทิ้งเฟรมนั้นไป แต่ถ้าได้รับอีเทอร์เน็ตเฟรมที่มี Destination Address เป็นของมันก็จะส่งผ่าน Data ที่อยู่ในฟิลด์นั้นไปให้ชั้นเน็ตเวิร์คต่อไป

Source Address (ขนาด 6 ไบต์) ใช้เก็บ MAC Address ของ Adapter ที่ส่งเฟรมเข้าไปในแลน

Type (ขนาด 2 ไบต์) บอกว่าให้อีเทอร์เน็ตสามารถใช้โปรโตคอลในชั้นเน็ตเวิร์คหลายๆ ชนิดได้ เพราะว่าโฮสท์สามารถใช้โปรโตคอลในชั้นเน็ตเวิร์คที่ไม่ใช่ไอพีได้

Data (ขนาด 46 – 1500 ไบต์) ใช้เก็บไอพีเดตาแกรม โดยที่ Maximum Transfer Unit (MTU) ของอีเทอร์เน็ตมีขนาดเป็น 1500 ไบต์ นั่นคือถ้าไอพีเดตาแกรมมีขนาดมากกว่า 1500 ไบต์ โฮสท์จะทำการแบ่งเดตาแกรมออกเป็นส่วนย่อยๆ ส่วน Minimum Size ของอีเทอร์เน็ตมีขนาดเป็น 46 ไบต์ นั่นคือถ้าไอพีเดตาแกรมมีขนาดน้อยกว่า 46 ไบต์ Data จะถูกเติมให้เต็ม 46 ไบต์

CRC (ขนาด 4 ไบต์) เพื่อให้ Adapter ของผู้รับตรวจหา Error ที่เกิดขึ้นในเฟรมได้ ถ้าบิตในเฟรมนั้นสลับกัน ซึ่งสาเหตุที่ทำให้บิตเกิด Error ขึ้นนั้นได้แก่สัญญาณที่ส่งมาอ่อน และคลื่นแม่เหล็กไฟฟ้ารบกวน เป็นต้น

2.3 ความหมายของการจัดประเภทแพ็กเก็ตตามกฎ

การจัดประเภทแพ็กเก็ตตามกฎ คือ กลไกในการตรวจตราเน็ตเวิร์คแพ็กเก็ตโดยอาศัยการเปรียบเทียบข้อมูลจากเฮดเดอร์ของแพ็กเก็ต ตั้งแต่ 1 ฟิลด์ขึ้นไป [2] เช่น จาก Source/Destination Internet-layer Address (32 bit), Source/Destination Transport-layer Port Number (16 bit), Type-of-Service (8 bit), Protocol (8 bit) และ Transport-layer Protocol Flag (8 bit) ก่อนที่จะกำหนดกระแส (Flow) ของแพ็กเก็ต โดยแต่ละแพ็กเก็ตในกระแสเดียวกันจะมีแอ็คชันที่เหมือนกัน ซึ่งกระแสของแพ็กเก็ตจะถูกกำหนดด้วยกฎที่ถูกกำหนดไว้ล่วงหน้า

สำหรับงานวิจัยนี้จะเรียกว่า Packet Classification มากกว่า Packet Filtering ซึ่งเป็นคำที่ยอมรับร่วมกัน และเป็นคำจำกัดความที่อธิบายหน้าที่ของมันได้ครอบคลุมมากที่สุด เพราะคำว่า Classify จะหมายถึงการกระทำกับทุกๆ แพ็กเก็ต ส่วนคำว่า Filtering หมายถึงการเลือกกระทำเฉพาะแพ็กเก็ตที่สนใจ [5]

ตัวอย่างของบริการในไอพีเน็ตเวิร์ค เช่น การจัดประเภทแพ็กเก็ตตามกฎสำหรับ Route Lookup จะพิจารณาจาก Destination IP Address และกำหนดช่องทางขาออก (Next-hop) ที่

เหมาะสม การจัดประเภทแพ็กเก็ตตามกฎสำหรับไฟร์วอลจะพิจารณาจาก Source IP Address , Destination IP Address, Source Port, Destination Port หรือฟิลด์อื่นๆ และกำหนดแอ็คชัน เช่น ACCEPT หรือ DROP การจัดประเภทแพ็กเก็ตตามกฎสำหรับ Quality of Service จะพิจารณาจาก Source IP Address และ Source Port และกำหนด Queue ที่เหมาะสม เป็นต้น

2.4 ความต้องการของการจัดประเภทแพ็กเก็ตตามกฎ

การจัดประเภทแพ็กเก็ตตามกฎเป็นเครื่องมือที่สำคัญสำหรับการกำหนดกระแสของแพ็กเก็ตบนเครือข่ายอินเตอร์เน็ต ซึ่งจะต้องปฏิบัติงานบนช่องทางการสื่อสารที่มีความเร็วสูงด้วยขนาดของกฎที่มีขนาดใหญ่ ดังนั้นแอ็พพลิเคชันการจัดประเภทแพ็กเก็ตตามกฎจึงต้องมีประสิทธิภาพสูงสุด โดยในการทำงานจะต้องพิจารณาในด้านต่างๆ ดังต่อไปนี้ [6]

1. มีความเร็วในการค้นหาสูง ปัจจุบันการจัดประเภทแพ็กเก็ตตามกฎจะต้องให้บริการบนช่องทางการสื่อสารที่มีความเร็วสูง ดังนั้นการจัดประเภทแพ็กเก็ตตามกฎจะต้องมีความเร็วในการค้นหาที่สูงตามไปด้วย

2. ใช้เนื้อที่จัดเก็บข้อมูลต่ำ การจัดประเภทแพ็กเก็ตตามกฎที่ดีนั้นต้องสามารถจัดการกับกฎที่มีขนาดใหญ่โดยใช้เนื้อที่ในการจัดเก็บที่ต่ำ

3. มีความเร็วในการปรับปรุงกฎสูง การจัดประเภทแพ็กเก็ตตามกฎจะทำการปรับปรุงโครงสร้างข้อมูลในขณะที่ทำการเพิ่มกฎหรือลบกฎ ซึ่งแอพพลิเคชันที่มีความถี่ในการปรับปรุงกฎสูงจำเป็นต้องมีความเร็วในการค้นหาที่สูงเช่นกัน โดยความเร็วในการปรับปรุงกฎของการจัดประเภทแพ็กเก็ตตามกฎควรจะไม่ต่ำกว่าความเร็วในการค้นหามากนัก

2.5 การกำหนดค่าของกฎให้อยู่ในรูป Prefix

ในการปรับปรุงกฎ วิธีการของการจัดประเภทแพ็กเก็ตตามกฎ หลายๆ วิธีต้องทำการแปลงค่าของกฎไปเป็น Prefix ซึ่งข้อมูลที่ถูกแปลงค่าแล้วจะอยู่ในรูปเซตของ Prefix โดยมีขนาด Prefix เท่ากับขนาดเฮดเดอร์ฟิลด์ (W) ของแพ็กเก็ตซึ่งแต่ละเซตจะมีสมาชิกไม่เกิน 2·W – 2 ตัว [6] เช่น กำหนดให้แต่ละกฎมีขนาด Prefix เท่ากับ 4 บิต กฎ 1 มีค่าเท่ากับ 4-7 ดังนั้นจะประกอบด้วยเซตของ Prefix เป็น {01**}, กฎ 2 มีค่าเท่ากับ 1- 14 ดังนั้นจะประกอบด้วยเซตของ Prefix เป็น {0001, 001*, 01**, 10**, 110*, 1110} เป็นต้น สำหรับตารางที่ 2.1 จะแสดงกฎจำนวน 6 กฎที่ทำการแปลงค่าเป็น Prefix โดยกฎแต่ละข้อประกอบด้วยฟิลด์จำนวน 2 ฟิลด์ คือ ฟิลด์ F1 และ F2

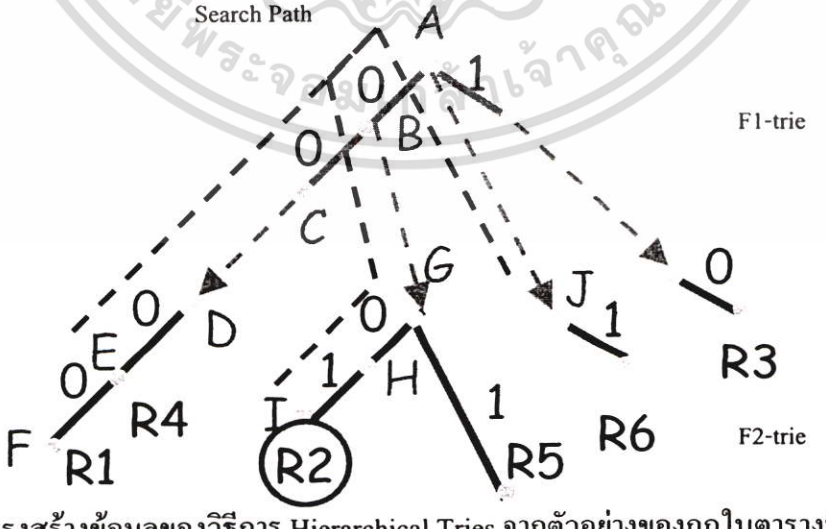
ตารางที่ 2.1 ตัวอย่างของกฎในรูปของ Prefix

Rule	F1	F2
R1	00*	00*
R2	0*	01*
R3	1*	0*
R4	00*	0*
R5	0*	1*
R6	*	1*

2.6 ตัวอย่างของวิธีการจัดประเภทแพ็กเก็ตตามกฎ

1. Linear Search มีลักษณะโครงสร้างข้อมูลของกฎเป็นแบบลิงลิสท์ ซึ่งโดยส่วนใหญ่กฎในลิสท์จะเรียงลำดับความสำคัญจากมากไปน้อย ในการค้นหาจะทำการเปรียบเทียบแพ็กเก็ตกับกฎตามลำดับทีละกฎ จนกระทั่งพบกฎที่ตรงกับข้อมูลในเฮดเดอร์ของแพ็กเก็ต สำหรับ Linear Search มี Search Speed Complexity เท่ากับ O(N) และ Storage Complexity เท่ากับ O(N) [6] ซึ่งแอพพลิเคชันที่ใช้วิธีการของการจัดประเภทแพ็กเก็ตตามกฎแบบ Linear Search ที่พบบ่อยๆ เช่น ไฟร์วอล เป็นต้น

2. Hierarchical Tries มีลักษณะโครงสร้างข้อมูลของกฎเป็นแบบลำดับชั้น โดยจะแบ่งชั้นตามจำนวนฟิลด์ ในแต่ละชั้นจะนำบิต Prefix มาสร้างเป็นทรี แต่ละกิ่งของทรีจะแทนค่าด้วยบิต 0 บิต 1 หรือ * (แทนด้วยลูกศรเป็นเส้นประ) ดังนั้นแต่ละกฎจะถูกแทนด้วยเส้นทางตั้งแต่โหนดแม่ของทรีในชั้นที่ 1 ถึงโหนดลูกของทรีในชั้นสุดท้าย ดังรูปที่ 2.6

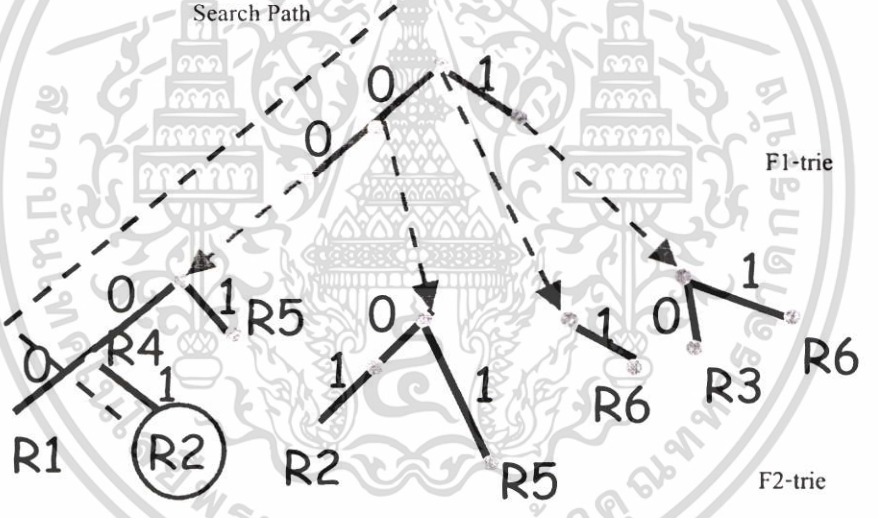


รูปที่ 2.6 โครงสร้างข้อมูลของวิธีการ Hierarchical Tries จากตัวอย่างของกฎในตารางที่ 2.1 ข้อมูลจากเฮดเดอร์ของแพ็กเก็ตเป็น (000,010)

จากรูปที่ 2.6 กฎ R1 ประกอบด้วยเส้นทางเริ่มจากโหนด A โหนด B โหนด C โหนด D โหนด E ถึงโหนด F และ กฎ R2 ประกอบด้วยเส้นทางเริ่มจากโหนด A โหนด B โหนด G โหนด H ถึงโหนด I เป็นต้น

วิธีการ Hierarchical Tries จะเริ่มทำการค้นหาจากชั้นที่ 1 โดยนำแต่ละบิตจากเฮดเดอร์ของแพ็กเก็ตมาเปรียบเทียบกับบิตในเส้นทางของทรีจนถึงชั้นสุดท้าย และเดินทางไปยังทุกๆ เส้นทางที่พบบิตที่ตรงกับบิตจากเฮดเดอร์ของแพ็กเก็ตในลักษณะของ Recursive จากตัวอย่าง วิธีการนี้จะทำการค้นหาทั้งหมด 3 เส้นทางและเลือกกฎ R2 เป็นผลลัพธ์ในการค้นหา ซึ่งวิธีการ Hierarchical Tries มี Search Speed Complexity เท่ากับ $O(W^D)$ และมี Storage Complexity เท่ากับ $O(N \cdot D \cdot W)$ [6]

3. Set - Pruning Tries มีลักษณะโครงสร้างข้อมูลของกฎคล้ายกับวิธีการ Hierarchical Tries แต่ทำการปรับปรุงโครงสร้างให้มีประสิทธิภาพในการค้นหาที่รวดเร็วขึ้น โดยยอมให้มีกฎที่ซ้ำซ้อนกันในแต่ละกิ่งของทรี ทำให้สามารถท่องไปบนทรีเพียงเส้นทางเดียว ดังรูปที่ 2.7

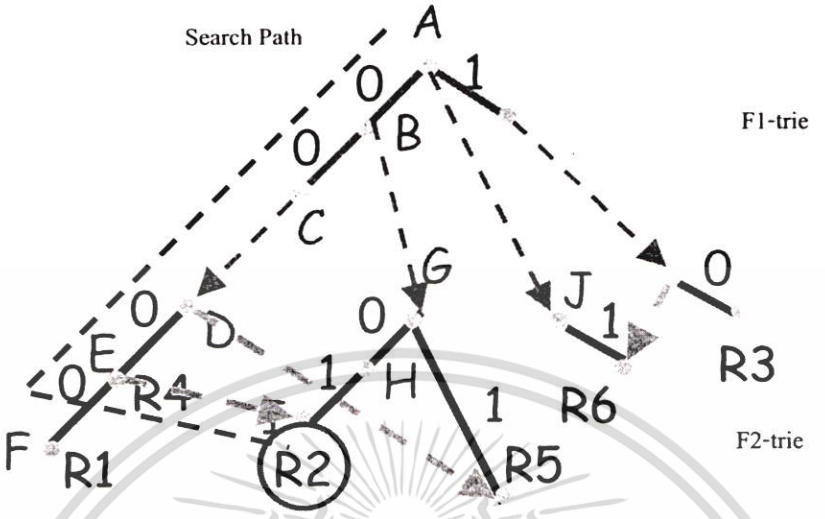


รูปที่ 2.7 โครงสร้างข้อมูลของวิธีการ Set - Pruning Tries จากตัวอย่างของกฎในตารางที่ 2.1 ข้อมูลจากเฮดเดอร์ของแพ็กเก็ตเป็น (000,010)

วิธีการ Set - Pruning Tries จะมีความเร็วในการค้นหาดีกว่าวิธีการ Hierarchical Tries แต่ใช้เนื้อที่จัดเก็บข้อมูลมากกว่าวิธีการ Hierarchical Tries และมีความซับซ้อนในการปรับปรุงกฎมากขึ้น โดยมี Search Speed Complexity เท่ากับ $O(D \cdot W)$ และ Storage Complexity เท่ากับ $O(N^D)$ [8]

4. Grid – of – Tries มีลักษณะโครงสร้างข้อมูลของกฎคล้ายกับวิธีการ Hierarchical Tries เช่นกัน แต่ทำการปรับปรุงโครงสร้างให้มีประสิทธิภาพในการจัดเก็บข้อมูลดีขึ้นกว่าวิธีการ Set - Pruning Tries โดยใช้เทคนิคเพิ่มพอยเตอร์ลงไปในทรีให้ชี้ไปยังโหนดของเส้นทางอื่นๆ เพื่อ

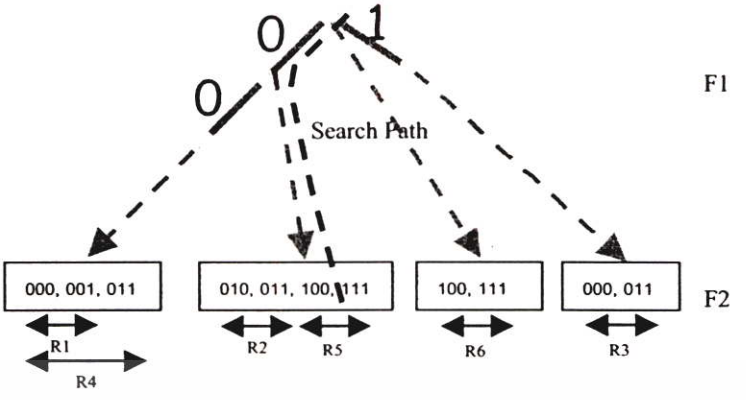
ลดกฎที่ซ้ำซ้อน ดังรูปที่ 2.8 อย่างไรก็ตามด้วยวิธีการนี้จะทำให้มีความซับซ้อนในการปรับปรุงกฎ จึงนิยมใช้กับแอพพลิเคชันการจัดประเภทแพ็กเก็ตตามกฎแบบ 2 ฟิลด์เท่านั้น



รูปที่ 2.8. โครงสร้างข้อมูลของวิธีการ Grid – of – Tries จากตัวอย่างของกฎในตารางที่ 2.1 ข้อมูลจากเฮดเดอร์ของแพ็กเก็ตเป็น (000,010)

วิธีการ Grid – of – Tries มี Search Speed Complexity เท่ากับ $O(W^{D-1})$ และ Storage Complexity เท่ากับ O(N•D•W) [11]

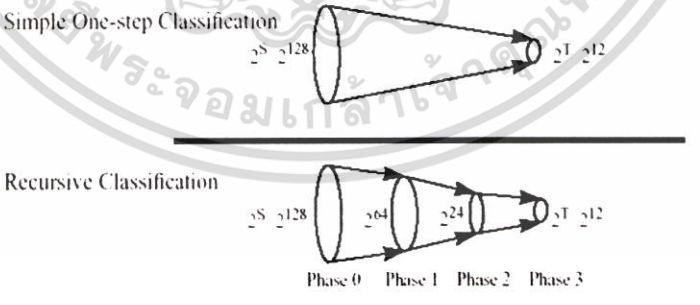
5. A 2D Classification Scheme เป็นวิธีการของการจัดประเภทแพ็กเก็ตตามกฎสำหรับแอพพลิเคชันแบบ 2 ฟิลด์ [9] โดยในฟิลด์แรกมีโครงสร้างข้อมูลของกฎเหมือนกับ Hierarchical Tries ฟิลด์ ที่ 2 จะอนุญาตให้จัดเก็บค่าของกฎในรูปแบบของช่วงลงในโครงสร้างข้อมูล โดยมีวิธีการค้นหาได้ตามความเหมาะสม ดังรูปที่ 2.9 การค้นหาในชั้นแรกจะเหมือนกับวิธีการ Hierarchical Tries แต่ชั้นที่ 2 จะทำการค้นหาแบบ Binary Search Tree โดยจะทำการเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์



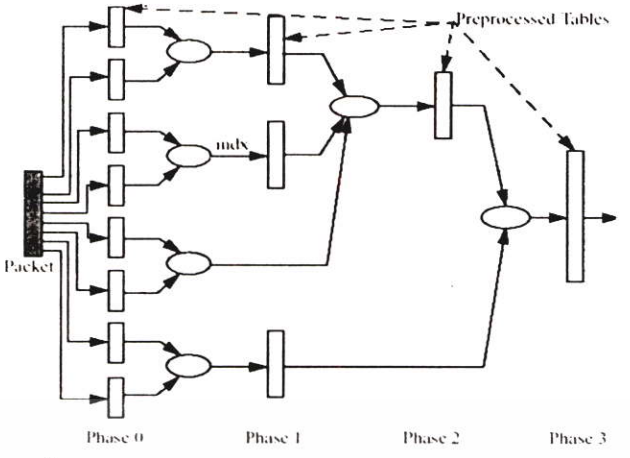
รูปที่ 2.9 โครงสร้างข้อมูลของวิธีการ 2D Classification Scheme จากตัวอย่างของกฎในตารางที่ 2.1 ข้อมูลจากเฮดเดอร์ของแพ็กเก็ตเป็น (011,110)

วิธีการ A 2D Classification Scheme มี Search Speed Complexity เท่ากับ O(W•logN) และ Storage Complexity เท่ากับ O(N•W)

6. Recursive Flow Classification (RFC) เป็นวิธีการของการจัดประเภทแพ็กเก็ตตามกฎแบบฮิวริสติก สำหรับแอพพลิเคชันแบบหลายฟิลด์ มีแนวคิดคือการ Mapping เฮดเดอร์บิตของแพ็กเก็ตขนาด S bit ไปเป็น T bit ซึ่ง T < S ดังรูปที่ 2.10 ก โดยการค้นหาจะทำการแบ่งเฟสจำนวน P เฟส นำเฮดเดอร์ของแพ็กเก็ตมาแบ่งเป็น Block นำค่าที่แบ่งได้มาทำการ Lookup ในตารางของแต่ละเฟส ค่าที่ได้จากตารางจะนำไปคำนวณเป็นค่าอินเด็กในเฟสถัดไป ผลลัพธ์ที่ได้จากเฟสสุดท้ายคือ classID ซึ่งเป็นพ้อยเตอร์ซึ่งชี้ไปยัง Rule ในฐานข้อมูลต่อไป ดังรูปที่ 2.10 ข



รูปที่ 2.10 ก แนวคิดการ Mapping เฮดเดอร์บิตของแพ็กเก็ตไปเป็น T บิต



รูปที่ 2.10 ข กระแสแพ็กเก็ต (Packet Flow) ใน RFC

วิธีการ RFC มี Search Speed Complexity เท่ากับ O(D) และ Storage Complexity เท่ากับ $O(N^D)$ แต่เป็นวิธีการหนึ่งที่มีความซับซ้อนในการปรับปรุงกฎสูง [7]

7. Tuple Space Search เป็นวิธีการของการจัดประเภทแพ็กเก็ตตามกฎสำหรับแอพพลิเคชันแบบหลายฟิลด์ โดยทำการสร้าง Tuple จากตำแหน่งของบิต Prefix ในแต่ละฟิลด์ของกฎ และสร้าง Concatenate Bit จากการนำบิต Prefix ในแต่ละฟิลด์ของกฎมาต่อกันลงในตาราง Hash ซึ่งแต่ละแถวของตาราง Hash จะเท่ากับ 1 Tuple ดังตารางที่ 2.2, Tuple Space Search จะมีวิธีการค้นหาแบบ Linear Search ในตาราง Hash โดยนำเฮดเดอร์บิตของแพ็กเก็ต ตามจำนวนใน Tuple มาต่อกันแล้วทำการเปรียบเทียบกับ Concatenate bit ในตาราง Hash และทำการเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์ [10]

ตารางที่ 2.2 การสร้าง Tuple จากตำแหน่งของบิต Prefix และตาราง Hash จากตัวอย่างของกฎในตารางที่ 2.1

Rule	Specification	Tuple
R1	(00*,00*)	(2,2)
R2	(0**,01*)	(1,2)
R3	(1**,0**)	(1,1)
R4	(00*,0**)	(2,1)
R5	(0**,1**)	(1,1)
R6	(***,1**)	(0,1)

Tuple	Rule	Concatenate Bit
(0,1)	{R6}	{1}
(1,1)	{R3,R5}	{10,01}
(1,2)	{R2}	{001}
(2,1)	{R4}	{000}
(2,2)	{R1}	{0000}

วิธีการ Tuple Space Search มี Search Speed Complexity เท่ากับ O(N) และ Storage Complexity เท่ากับ O(N)

8. Bitmap-intersection เป็นวิธีการของการจัดประเภทแพ็กเก็ตตามกฎสำหรับแอพพลิเคชันแบบหลายฟิลด์ สำหรับการปรับปรุงกฎ ในแต่ละฟิลด์จะแบ่งกฎออกเป็นช่วงโดยโปรเจ็คท์ช่วงของกฎลงบนแกนของกราฟ ทำการสร้าง Bitmap ขนาด N บิต ซึ่งเป็นตัวแทนของแต่ละช่วง ทั้งหมดไม่เกิน 2N+1 Bitmap โดยตำแหน่งแต่ละบิตของ Bitmap จะตรงกับหมายเลขของกฎ กฎใดที่ไม่มีการโปรเจ็คท์ลงในช่วง จะกำหนดบิตของ Bitmap ให้มีค่าเท่ากับ 0 ส่วนกฎใดที่มีการโปรเจ็คท์ลงในช่วง จะกำหนดบิตของ Bitmap ให้มีค่าเท่ากับ 1 วิธี Bitmap-intersection จะเริ่มทำการค้นหาช่วงที่ตรงกับเฮดเดอร์ของแพ็กเก็ตด้วยวิธี Binary Search ผลลัพธ์ที่ได้จากการค้นหาแต่ฟิลด์คือ Bitmap นำ Bitmap ทั้งหมดมาทำการ Intersection ดังตารางที่ 2.3 โดยเฮดเดอร์ของแพ็กเก็ตจะตรงกับกฎที่มีบิตมีค่าเท่ากับ 1 และทำการเลือกกฎที่มีลำดับความสำคัญสูงสุด [9]

ตารางที่ 2.3 Bitmap Table จากตัวอย่างของกฎในตารางที่ 2.1

Field 1

R_1^1	{R1,R2,R4,R5,R6}	110111
R_1^2	{R2,R5,R6}	010011
R_1^3	{R3,R6}	001001

Field 2

R_2^1	{R1,R3,R4}	101100
R_2^2	{R2,R3}	011000
R_2^3	{R5,R6}	000011

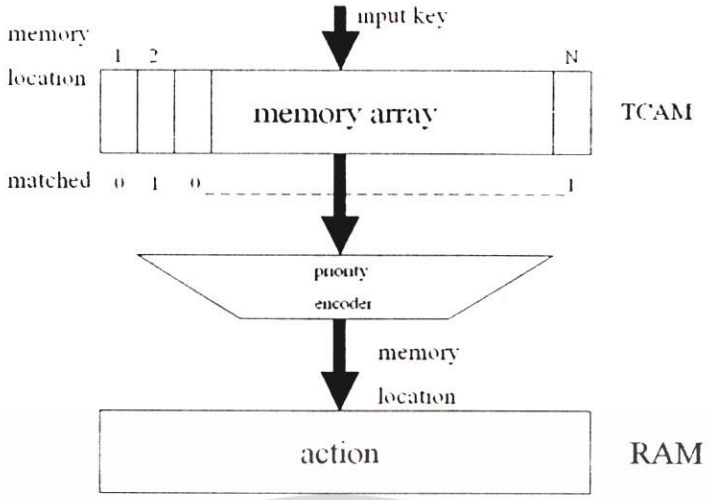
Query on P(011,110): 010011 Field 1 bitmap

000111 Field 2 bitmap

000011 **R5** Best matching rule

วิธีการ Bitmap-intersection มี Search Speed Complexity เท่ากับ $O\left(D \cdot \log N + \frac{N}{WORD}\right)$ บนฮาร์ดแวร์ และมี Storage Complexity เท่ากับ $O(D \cdot N^2)$

9. Ternary CAM เป็นวิธีการของการจัดประเภทแพ็กเก็ตตามกฎที่ทำงานบนฮาร์ดแวร์ และใช้หน่วยความจำที่มีจำนวน Transistor มากกว่าหน่วยความจำทั่วไป เพราะแต่ละบิตของ TCAM จะประกอบด้วยบิต 0 บิต 1 หรือ บิต * (Wildcard) ทำให้สิ้นเปลืองพลังงานมากกว่าหน่วยความจำทั่วไป ซึ่งการค้นหาจะนำเฮดเดอร์บิตของแพ็กเก็ต (Input Key) ทำการ Lookup แบบขนานทุกๆ แอดเดรสของหน่วยความจำ TCAM โดยเฮดเดอร์ของแพ็กเก็ตจะตรงกับกฎที่มีบิตมีค่าเท่ากับ 1 และทำการเลือกกฎที่มีลำดับความสำคัญสูงสุด ดังรูปที่ 2.11.



รูปที่ 2.11 โครงสร้างสถาปัตยกรรมของวิธีการ Ternary CAM

วิธีการ Ternary CAM มี Search Speed Complexity เท่ากับ O(1) และมี Storage Complexity เท่ากับ O(N) [4]

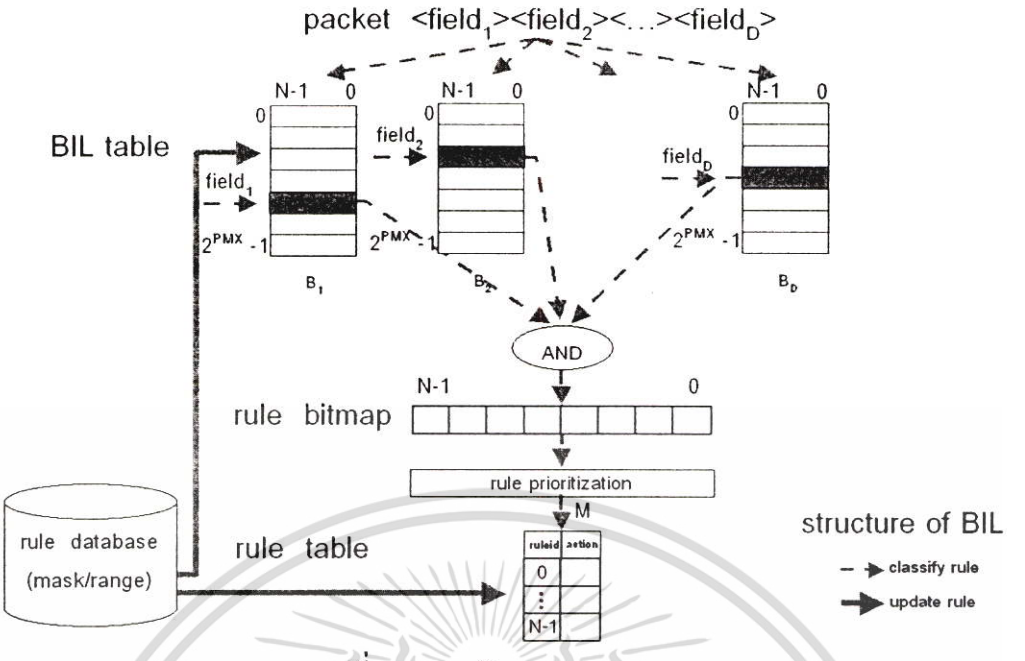
บทที่ 3
การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี
Bitmap Intersection Lookup (BIL)

จากความสำคัญของการจัดประเภทแพ็กเก็ตตามกฎสำหรับแอพพลิเคชันต่างๆ และปัญหาที่เกิดขึ้น งานวิจัยนี้จึงนำเสนอวิธีการของการจัดประเภทแพ็กเก็ตตามกฎที่มีโครงสร้างข้อมูลง่ายและไม่ซับซ้อน สำหรับการค้นหาแบบหลายฟิลด์ โดยสามารถทำงานได้ทั้งบนซอฟท์แวร์ และฮาร์ดแวร์

3.1 แนวคิดและสถาปัตยกรรมของ BIL

จากรูปที่ 3.1 สถาปัตยกรรมของวิธีการ BIL จะประกอบด้วย BIL table จำนวน D ตาราง แต่ละตารางประกอบด้วยแอดเดรสจำนวน 2^{PMX} แอดเดรส ซึ่งแต่ละแอดเดรสประกอบด้วยบิตจำนวน N บิต ตั้งแต่ บิต 0 ถึงบิต N-1 เรียกว่า Bitvector และตำแหน่งแต่ละบิตของ Bitvector จะตรงกับตำแหน่งของกฎใน rule table โดยค่าบิตที่เป็น "1" คือ บิตที่อยู่ในแอดเดรสที่ตรงกับค่าของกฎ และค่าบิตที่เป็น "0" คือ บิตที่ไม่อยู่ในแอดเดรสที่ตรงกับค่าของกฎ การจัดเรียงลำดับความสำคัญของกฎจะจัดเรียงจากมากไปน้อย ในการเพิ่มกฎจะนำค่าของกฎซึ่งถูกกำหนดในรูปของมาสค์ (Mask) หรือช่วง (Range) ทำการแปลงค่าให้อยู่ในรูปของ Prefix เพื่อใช้สำหรับ Set Bitvector ใน BIL table และจัดเก็บแอ็คชันของกฎแต่ละข้อลงใน rule table ต่อไป

สำหรับการค้นหาจะแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block ขนาดเล็กๆ จำนวน D Blocks นำค่าของแต่ละ Block ไป Lookup ใน BIL table และนำผลลัพธ์ที่ได้จากการ Lookup คือ Bitvector จากแต่ละตาราง ทำการ Intersection ด้วยตรรก AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์ และกำหนดแอ็คชันของแพ็กเก็ตจากข้อมูลในตาราง rule table



รูปที่ 3.1 สถาปัตยกรรมของ BIL

โดยวิธีการ BIL สามารถนำไปพัฒนาได้ด้วยวิธีซอฟท์แวร์และฮาร์ดแวร์ ดังต่อไปนี้

3.1.1 การพัฒนา BIL ด้วยซอฟท์แวร์

จากสถาปัตยกรรมของ BIL ในรูปที่ 3.1 สามารถนำไปพัฒนาได้ด้วยวิธีซอฟท์แวร์ โดยประกอบด้วยส่วนต่างๆ ได้แก่

1. BIL Table เป็น Array 3 มิติ มีชนิดของข้อมูลเป็น u_int32_t สำหรับเก็บข้อมูล Bitvector โดยประกาศเป็น

u_int32_t bil_table[DMX][A][Nword];

กำหนดให้

DMX คือ จำนวนตาราง

A คือ จำนวนแอดเดรส

Nword คือ จำนวน WORD ของ Bitvector

2. Rule Bitmap เป็นผลลัพธ์ที่ได้จากการ Intersection Bitvector ด้วยตรรก AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์

3. Rule Database เป็นฐานข้อมูลกฎ ดังตัวอย่างในรูปที่ 3.2 โดยมีคำสั่งดังต่อไปนี้

คำสั่งการเพิ่มกฎแบบ Insert

-I|--insert **<Ruleid>** *-a|--acction* **<Action>**

[*-p|--protocol* **<Protocol Number|Protocol Name>**]

[*-s|--source* **<IP Address|Host Name>**]

[*-d*|*--destination* **<IP Address|Host Name>**]

[*--sport* **<Port Number|Port Name>**]

[*--dport* **<Port Number|Port Name>**]

[*--smac* **<MAC Address>**]

[*--dmac* **<MAC Address>**]

คำสั่งการเพิ่มกฎแบบ Append

-A|*--append -a*|*--acction* **<Action>**

[*-p*|*--protocol* **<Protocol Number|Protocol Name>**]

[*-s*|*--source* **<IP Address|Host Name>**]

[*-d*|*--destination* **<IP Address|Host Name>**]

[*--sport* **<Port Number|Port Name>**]

[*--dport* **<Port Number|Port Name>**]

[*--smac* **<MAC Address>**]

[*--dmac* **<MAC Address>**]

กำหนดให้

Ruleid คือ หมายเลขของกฎ ซึ่งเป็นเลขจำนวนเต็มตั้งแต่ 0 ถึง Nmax-1 โดยที่ Nmax คือ จำนวนกฎสูงสุด

Action คือ แอ็คชันของกฎ ได้แก่ ACCEPT และ DROP

Protocol Number หรือ Protocol Name คือ หมายเลขของโปรโตคอล หรือชื่อของโปรโตคอล เช่น 6, TCP, UDP และ ALL เป็นต้น

IP Address หรือ Host Name คือ หมายเลขไอพีแอดเดรสที่เขียนอยู่ในรูปมาสค์ หรือช่วง หรือชื่อของโฮสท์ตามระบบโดเมนเนม เช่น 192.168.1.0/255.255.255.0, 192.168.1.0/24, 192.168.1.10-192.168.1.15, **www.kmitl.ac.th** และ ALL เป็นต้น

Port Number หรือ Port Name คือ หมายเลขของพอร์ต ที่มีค่าตั้งแต่ 0-65535 หรือชื่อของพอร์ต เช่น 80, 6000-65535, HTTP และ ALL เป็นต้น

MAC Address คือ แลนแอดเดรสของ Adapter เช่น FFFFFFFFFFFF เป็นต้น

```
--insert 0 --action DROP --protocol UDP --source 24.128.0.0/16 --destination 4.0.0.0/8
-A -a ACCEPT -p TCP -s 64.248.128.0/20 -d 8.16.192.0/24
-A -a ACCEPT -p ALL -s 24.128.0.0/16 -d 4.16.128.0/20
-A -a DROP -p TCP -s www.google.com -d 192.168.1.198
-I 4 -a DROP --smac FFFFFFFFFFFF
```

รูปที่ 3.2 ตัวอย่างฐานข้อมูลกฎ

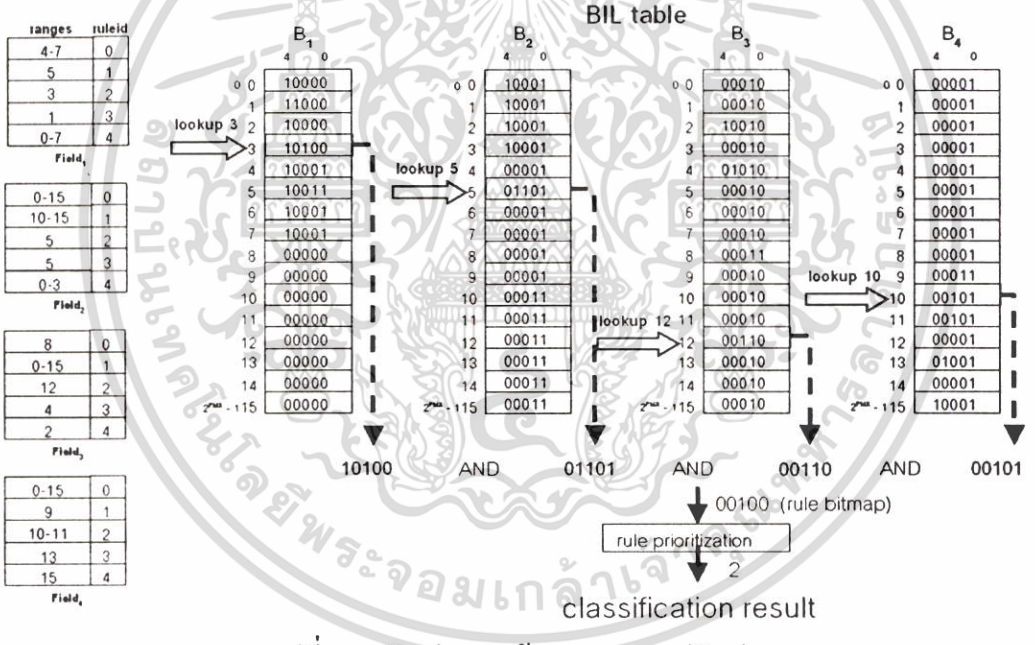
4. Rule Table เป็น Array 1 มิติ มีชนิดของข้อมูลเป็น struct rule สำหรับเก็บข้อมูลแอ็คชันของกฎ และคำสั่งการเพิ่มกฎ (Insert/Append) ที่อ่านมาจากฐานข้อมูล โดยประกาศเป็น

```
struct rule {
    long long int ruleid;
    int action;
    char command[MAX_LEN];
};
struct rule rule_table[Nmax];
```

กำหนดให้

MAX_LEN คือ จำนวนตัวอักษรสูงสุด

Nmax คือ จำนวนกฎสูงสุด



รูปที่ 3.3 ตัวอย่างการค้นหาและการปรับปรุงกฎ

รูปที่ 3.3 จะแสดงตัวอย่างการปรับปรุงกฎและการค้นหาของวิธีการ BIL ดังต่อไปนี้

1. การปรับปรุงกฎ ในตาราง $Field_1$- $Field_4$ แสดงฐานข้อมูลกฎที่กำหนดอยู่ในรูปของช่วงทั้งหมด 5 ข้อ คือ ruleid 0 - ruleid 4 โดยกฎแต่ละข้อจะประกอบด้วยฟิลด์จำนวน 4 ฟิลด์ ซึ่งแต่ละฟิลด์มีขนาด 4 บิต กำหนดให้ PMX คือขนาดของ Block เท่ากับ 4 บิต ดังนั้น BIL table แต่ละตารางจะประกอบด้วยแอดเดรสจำนวน $2^4 = 16$ แอดเดรส คือ หมายเลขแอดเดรส 0 - หมายเลขแอดเดรส 15 ซึ่งแต่ละแอดเดรสจะมีขนาดเท่ากับจำนวนกฎเท่ากับ 5 บิต ตั้งแต่ บิต 0 - บิต 4 ในการ

เพิ่มกฎจะนำค่าของกฎซึ่งถูกกำหนดในรูปของช่วง ทำการแปลงค่าให้อยู่ในรูปของ Prefix เพื่อใช้สำหรับ Set Bitvector ใน BIL table เช่น ที่ $Field_1$ พบว่า ruleid 0 มีค่าเท่ากับ 4-7 (0100-0111) ดังนั้นจะประกอบด้วยเซตของ Prefix เป็น {01**} หรือ {0100/1100} (ในรูปของ value/mask) นำ Prefix ที่ได้ทำการ set Bitvector ในตาราง B_1 ตั้งแต่แอดเดรส 4-แอดเดรส 7 ด้วยอัลกอริทึมในรูปที่ 3.4 (ในกรณีการลบกฎจะนำ Prefix ที่ได้ทำการ reset Bitvector ด้วยอัลกอริทึมในรูปที่ 3.5) โดยจากฐานข้อมูลกฎสามารถสร้างเป็น Bitvector ดังแสดงในตาราง B_1-B_4

```
1. for (j = (value & mask);j <= ((value & mask) + (~mask));j++) {
2.    setbitvector(i,j,ruleid);
3. }
```

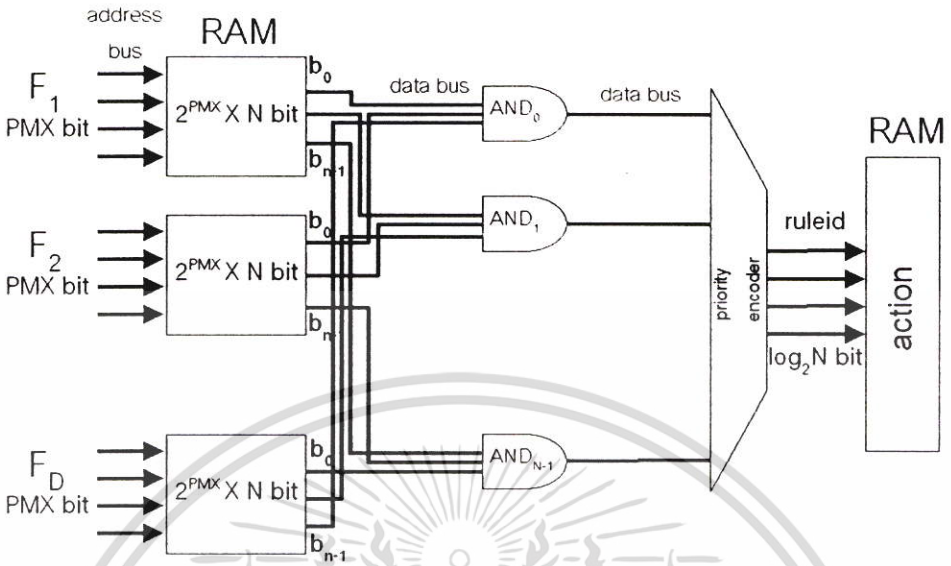
รูปที่ 3.4 อัลกอริทึม set Bitvector

```
1. for (j = (value & mask);j <= ((value & mask) + (~mask));j++) {
2.    resetbitvector(i,j,ruleid);
3. }
```

รูปที่ 3.5 อัลกอริทึม reset Bitvector

2. การค้นหา ทำการแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block ขนาดเท่ากับ PMX เท่ากับ 4 บิต นำค่าของแต่ละ Block ซึ่งมีค่าเป็น 3, 5, 12 และ 10 ไป Lookup ใน BIL table นำ Bitvector ซึ่งได้จากการ Lookup ในตาราง B_1-B_4 ทำการ Intersection ด้วยตรรก AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์ ดังนั้นจากตัวอย่างดังกล่าวแพ็กเก็ตจะถูกจัดประเภทตรงกับกฎข้อที่ 2

3.1.2 การพัฒนา BIL ด้วยฮาร์ดแวร์



รูปที่ 3.6 การพัฒนาวิธีการ BIL ด้วยฮาร์ดแวร์

จากรูปที่ 3.6 BIL table แต่ละตารางจะถูกจัดเก็บอยู่ในหน่วยความจำ RAM แต่ละตัว ซึ่งมีแอดเดรสจำนวน 2^{PMX} แอดเดรส แต่ละแอดเดรสจะมีขนาดเท่ากับ N บิต ตั้งแต่ บิต 0-บิต N-1 ในการค้นหาจะนำเฮดเดอร์ของแพ็กเก็ตไป lookup ผ่านทาง address bus ขนาด PMX บิต และนำผลลัพธ์ที่ได้จากการ lookup (Bitvector) ทำการ Intersection ด้วยตรรก AND บนฮาร์ดแวร์ ผ่านทาง data bus โดย priority encoder จะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์ ซึ่งจะเป็นพ้อยเตอร์ที่ชี้ไปยังแอ็คชันในหน่วยความจำ RAM ต่อไป

3.2 การวิเคราะห์ประสิทธิภาพเบื้องต้น

จากแนวคิดดังที่กล่าวมาข้างต้นของวิธีการ BIL จะแสดงการวิเคราะห์สมการเพื่อหาเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล ดังต่อไปนี้

กำหนดให้

N คือ จำนวนกฎทั้งหมด

D คือ จำนวนเฮดเดอร์ฟิลด์

F_i คือ ขนาดของแต่ละเฮดเดอร์ฟิลด์ โดยที่ $1 <= i <= D$

WORD คือ ขนาดบิตที่ทำการประมวลผลในแต่ละครั้ง

ถ้าให้ F_i มีขนาดเท่ากันทุกฟิลด์ โดยแต่ละฟิลด์มีขนาดเท่ากับ W บิต เพราะฉะนั้นขนาดบิตทั้งหมดจะเท่ากับ $D \cdot W$ บิต

ทำการแบ่ง F_i ออกเป็นขนาด $P_{i,j}$ บิต โดยกำหนดให้

M_i คือ จำนวน Block ที่แบ่งใน F_i โดยที่ $\sum_{j=1}^{M_i} P_{i,j} = F_i$

พิจารณาที่ F_i ใดๆ เพียง 1 ฟิลด์ และที่ $P_{i,j}$ ของ F_i มีขนาดเท่ากัน เท่ากับ PM_i บิต เพราะฉะนั้น $M_i = \frac{F_i}{PM_i}$

กำหนดให้ PM_i มีขนาดเท่ากันทุกฟิลด์ เท่ากับ PMX บิต และ MX คือ จำนวน Block ภายในแต่ละฟิลด์ เพราะฉะนั้น $MX = \frac{W}{PMX}$ และจำนวนตารางจะเท่ากับ $\frac{D \cdot W}{PMX}$ หรือ D•MX

1. เวลาในการค้นหา

```
1.  อ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์ → t_read
2.  for (i=0; i<D; i++ ) {
3.      for (j=0; j<MX; j++ ) {
4.          แบ่งเฮดเดอร์ฟิลด์ที่ i ของแพ็กเก็ตเป็น Block โดยเลื่อนบิตไปทาง
            ขวาครั้งละ PMX บิต และเก็บค่าลงในหน่วยความจำ → t_shiftaverage
5.      }
6.  }
7.  for (i=0; i<n/WORD; i++ ) {
8.      for (j=0; j< D•MX; j++ ) {
9.          นำค่าของเฮดเดอร์ฟิลด์ที่ถูกแบ่งในหน่วยความจำไป Lookup ใน BIL table และ
            นำผลลัพธ์ คือ Bitvector ในตำแหน่ง WORD ที่ i จาก BIL table j มา AND กัน
            → t_look + t_and
10.     }
11.     if (ผลลัพธ์จากการ AND != 0) {
12.         คำนวณหมายเลขของกฎจากตำแหน่งบิตที่มีค่าเป็น 1 บิตแรกใน Bitmap และ
            ส่งค่า Action จาก rule table กลับ
13.     }
14. }
15. ส่งค่า Default Action กลับ
```

t_div: บรรทัดที่ 2-6
t_sproc: บรรทัดที่ 7-14
t_search: บรรทัดที่ 1-15

รูปที่ 3.7 อัลกอริทึมการค้นหา

จากรูปที่ 3.7 อัลกอริทึมการค้นหาจะเริ่มการทำงานโดยอ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์ ดังชุดคำสั่งของบรรทัดที่ 1 และทำการแบ่งเฮดเดอร์ฟิลด์ (ซึ่งมีจำนวนทั้งหมด D ฟิลด์) เป็น Block ขนาดเล็กๆ โดยเลื่อนบิตไปทางขวาครั้งละ PMX บิต และเก็บค่าเหล่านั้นลงในหน่วยความจำ โดยจำนวน Block ทั้งหมดจะเท่ากับ D•MX หรือ $\frac{D \cdot W}{PMX}$ ดังชุดคำสั่งของบรรทัดที่ 2-6 ในลำดับถัดไปจะนำค่าของเฮดเดอร์ฟิลด์ที่ถูกแบ่งในหน่วยความจำไป Lookup ใน BIL table และนำผลลัพธ์ คือ Bitvector มากระทำตรรก AND ครั้งละ

WORD ดังชุดคำสั่งของบรรทัดที่ 8-10 โดยหากพบว่าผลลัพธ์ที่ได้จากการ AND มีค่าไม่เท่ากับ 0 อัลกอริทึมจะทำการคำนวณหมายเลขของกฎจากตำแหน่งบิตที่มีค่าเป็น 1 บิตแรกใน Bitmap และส่งค่า Action จาก rule table กลับไป ดังชุดคำสั่งของบรรทัดที่ 11-13 โดยอัลกอริทึมจะกระทำตรรกะ AND ทั้งหมด $\frac{n}{WORD}$ ครั้งดังชุดคำสั่งของบรรทัดที่ 7-14

กำหนดให้

ชุดคำสั่งของบรรทัดที่ 1-15 ใช้เวลาในการทำงานเท่ากับ t_{search}

ชุดคำสั่งของบรรทัดที่ 1 ใช้เวลาในการทำงานเท่ากับ t_{read}

ชุดคำสั่งของบรรทัดที่ 2-6 ใช้เวลาในการทำงานเท่ากับ t_{div}

ชุดคำสั่งของบรรทัดที่ 4 ใช้เวลาในการทำงานเท่ากับ $t_{shiftaverage}$

ชุดคำสั่งของบรรทัดที่ 7-14 ใช้เวลาในการทำงานเท่ากับ t_{sproc}

ชุดคำสั่งของบรรทัดที่ 9 ใช้เวลาในการทำงานเท่ากับ $t_{look} + t_{and}$

เพราะฉะนั้น $t_{search} = t_{read} + t_{div} + t_{sproc}$

โดยที่ $t_{div} = \sum_{i=1}^{D}\left(\sum_{j=1}^{M_i} t_{shiftaverage}\right)$

พิจารณาที่ขนาดฟิลด์เท่ากัน และขนาด PM_i เท่ากัน ดังนั้น

$t_{div} = MX \cdot D \cdot t_{shiftaverage} = \frac{D \cdot W}{PMX} \cdot t_{shiftaverage}$

และ $t_{sproc} = \frac{D \cdot W}{PMX} \cdot (t_{look} + t_{and}) = \frac{D \cdot W}{PMX} \cdot t_{la}$

$= \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot \left(\frac{D \cdot W}{PMX} \cdot t_{la}\right)$

โดยที่ $\Pr ob_{found}(n)$ คือ ความน่าจะเป็นในการค้นหาว่าข้อมูลนี้ตรงกับกฎที่ n

กรณี Best Case, $\Pr ob_{found}(n) \cdot \frac{n}{WORD} = 1$

กรณี Worst Case, $\Pr ob_{found}(n) \cdot \frac{n}{WORD} = \frac{N}{WORD}$

ดังนั้นจากอัลกอริทึม จะใช้เวลาในการค้นหาต่อแพ็กเก็ตโดยมีสมการทั่วไป คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot \left(\frac{D \cdot W}{PMX} \cdot t_{la}\right)$$

$$= t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la}\right) \quad (1)$$

กรณี Best Case จะใช้เวลาในการค้นหาต่อแพ็กเก็ต โดยมีสมการ คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + t_{la}\right) \quad (2)$$

และ กรณี Worst Case จะใช้เวลาในการค้นหาต่อแพ็กเก็ต โดยมีสมการ คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \frac{N}{WORD} \cdot t_{la} \right) \quad (3)$$

2. เวลาในการปรับปรุงกฎ

```
1.  รับคำสั่งของกฎจากฐานข้อมูลกฎ หรือจากผู้ดูแลระบบ เก็บลงในบัฟเฟอร์
2.  แปลความหมายคำสั่งของกฎที่อ่านมา และเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำ        } t_prep
3.  เก็บค่า ruleid, Action และคำสั่งลงใน rule table, กำหนดค่า tmp = 0 และ d = 0
4.  for (i=0;i<D; i++) {
5.          tmp +=d และ แปลงค่าของกฎให้อยู่ในรูปของ Prefix (value/mask) ➔ t_conv
6.          current = first
7.          While (current != NULL) {
8.                  d=temp
9.                  for (k=0;k < MX; k++) {
10.                         สำหรับ current->Prefix แบ่ง Prefix (value/mask) ของฟิวด์ i เป็น Block โดย
                            เลื่อนบิตไปทางขวาครั้งละ PMX บิต และเก็บค่าลงในตัวแปร value และ
                            mask ➔ t_shiftaverage
11.                         for (l=value AND mask ;l<=( value AND mask)+~mask; l++) {
12.                                 set Bitvector ที่ตำแหน่งบิต ruleid แอดเดรสหมายเลข l
                                    ใน BIL table d ➔ t_set
13.                         }
14.                         d++
15.                 }
16.                 current = current->next
17.         }
18. }
```

t_proc ครอบคลุมบรรทัดที่ 4–18; t_add ครอบคลุมบรรทัดที่ 1–18

รูปที่ 3.8 อัลกอริทึมการเพิ่มกฎ

จากรูปที่ 3.8 อัลกอริทึมการเพิ่มกฎจะเริ่มการทำงานโดยรับคำสั่งของกฎจากฐานข้อมูลกฎ หรือจากผู้ดูแลระบบ เก็บลงในบัฟเฟอร์ แปลความหมายคำสั่งของกฎที่อ่านมา และเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำ และเก็บค่า ruleid, Action และคำสั่งลงใน rule table ดังชุดคำสั่งของบรรทัดที่ 1-3 โดยอัลกอริทึมจะทำการแปลงค่าของกฎให้อยู่ในรูปของ Prefix (value/mask) ซึ่งข้อมูลที่ถูกแปลงค่าแล้วจะอยู่ในรูปเซตของ Prefix ซึ่งแต่ละเซตจะมีสมาชิกไม่เกิน 2·W – 2 ตัว ดังชุดคำสั่งของบรรทัดที่ 5 ในลำดับถัดไปจะทำการแบ่ง Prefix (value/mask) เป็น Block โดยเลื่อนบิตไปทางขวาครั้งละ PMX บิต และเก็บค่าลงในตัวแปร value และ mask ดังชุดคำสั่งของบรรทัดที่ 10 และนำตัวแปร value และ mask มาคำนวณหาหมายเลขแอดเดรสต่ำสุด และหมายเลขแอดเดรสสูง

สุด ซึ่งจะมีจำนวนแอดเดรสไม่เกิน 2^{PMX} แอดเดรส เพื่อทำการ set Bitvector ในตาราง BIL table ต่อไป ดังชุดคำสั่งของบรรทัดที่ 11-13

กำหนดให้

ชุดคำสั่งของบรรทัดที่ 1-18 ใช้เวลาในการทำงานเท่ากับ t_{add}

ชุดคำสั่งของบรรทัดที่ 1-3 ใช้เวลาในการทำงานเท่ากับ t_{prep}

ชุดคำสั่งของบรรทัดที่ 5 ใช้เวลาในการทำงานเท่ากับ t_{conv}

ชุดคำสั่งของบรรทัดที่ 10 ใช้เวลาในการทำงานเท่ากับ $t_{shiftaverage}$

ชุดคำสั่งของบรรทัดที่ 12 ใช้เวลาในการทำงานเท่ากับ t_{set}

ชุดคำสั่งของบรรทัดที่ 4-18 ใช้เวลาในการทำงานเท่ากับ t_{proc}

เพราะฉะนั้น $t_{add} = t_{prep} + t_{proc}$

B_{min} คือ หมายเลขแอดเดรสต่ำสุด

B_{max} คือ หมายเลขแอดเดรสสูงสุด

N_{node} คือ จำนวนสมาชิกของ Prefix (value/mask) ที่ได้จากการแปลงค่าของกฎ

โดยที่ $1 <= N_{node} <= (2 \cdot W-2)$

$N_{address} = B_{max} - B_{max} + 1$

$= 2^{mask_d}$; $mask_d$ คือจำนวนบิตที่จะ net mask, $1 <= d <= \frac{D \cdot W}{PMX}$

ดังนั้น $t_{proc} = D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right]$

กรณี Best Case, $N_{address} = 1$, $N_{node} = 1$

กรณี Worst Case, $N_{address} = 2^{PMX}$, $N_{node} = 2 \cdot W-2$

ดังนั้นจากอัลกอริทึม จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการทั่วไป คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right] \quad (4)$$

กรณี Best Case จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการ คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + \frac{W}{PMX} \cdot \left(t_{shiftaverage} + t_{set} \right) \right] \quad (5)$$

กรณี Worst Case จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการ คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + (2 \cdot W - 2) \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + 2^{PMX} \cdot t_{set} \right) \right] \quad (6)$$

จากอัลกอริทึมการเพิ่มกฎในรูปที่ 3.8 ชุดคำสั่งในบรรทัดที่ 12 จะทำการ set Bitvector ในตาราง BIL table ในขณะที่อัลกอริทึมการลบกฎจะทำการ reset Bitvector ในตาราง BIL table โดยกำหนดให้เวลาในการปรับปรุงกฎคือ t_{update} เพราะฉะนั้น $t_{update} = t_{add} = t_{del}$

3. เนื้อที่จัดเก็บข้อมูล

สำหรับเนื้อที่จัดเก็บข้อมูลจะมีสมการ คือ

$$\text{storage (BIL table)} = \frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot \frac{N}{8} \text{ bytes} \quad (7)$$

จากสมการที่ (1)-(7) พบว่าความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลของวิธีการ BIL จะขึ้นอยู่กับขนาดของ Prefix ที่ถูกกำหนดไว้เป็นสำคัญ ดังนั้นผู้วิจัยจะทำการศึกษารูปแบบการแบ่ง Prefix ที่ทำให้โปรแกรมมีประสิทธิภาพดีที่สุด

โดยจะเสนอตัววัดประสิทธิภาพของวิธีการของการจัดประเภทแพ็กเก็ต จากผลคูณของประสิทธิภาพทั้ง 3 ด้าน จำนวน 4 ผลคูณ เพื่อดูอิทธิพลของตัวคูณแต่ละตัวที่มีผลกระทบต่อความสูงของกราฟ และหาจุดที่ทำให้โปรแกรมมีประสิทธิภาพโดยรวมดีที่สุด โดยพิจารณาจากจุดต่ำสุดของผลคูณ ได้แก่

1. ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล
2. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล
3. ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ
4. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล

ซึ่งจากสมการทั่วไปของวิธีการ BIL ดังที่ได้กล่าวมาข้างต้นสามารถแสดงผลคูณแต่ละผลคูณ ดังต่อไปนี้

1. ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล (4)×(7)

$$= \frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot \frac{N}{8} \cdot \left\{ t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverge} + N_{address} \cdot t_{set} \right) \right] \right\} \quad (8)$$

2. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล (1)×(7)

$$= \frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot \frac{N}{8} \cdot \left\{ t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la} \right) \right\} \quad (9)$$

3. ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ (1)×(4)

$$= \left\{ t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la} \right) \right\} \cdot \left\{ t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right] \right\} \quad (10)$$

4. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล (1)×(4) ×(7)

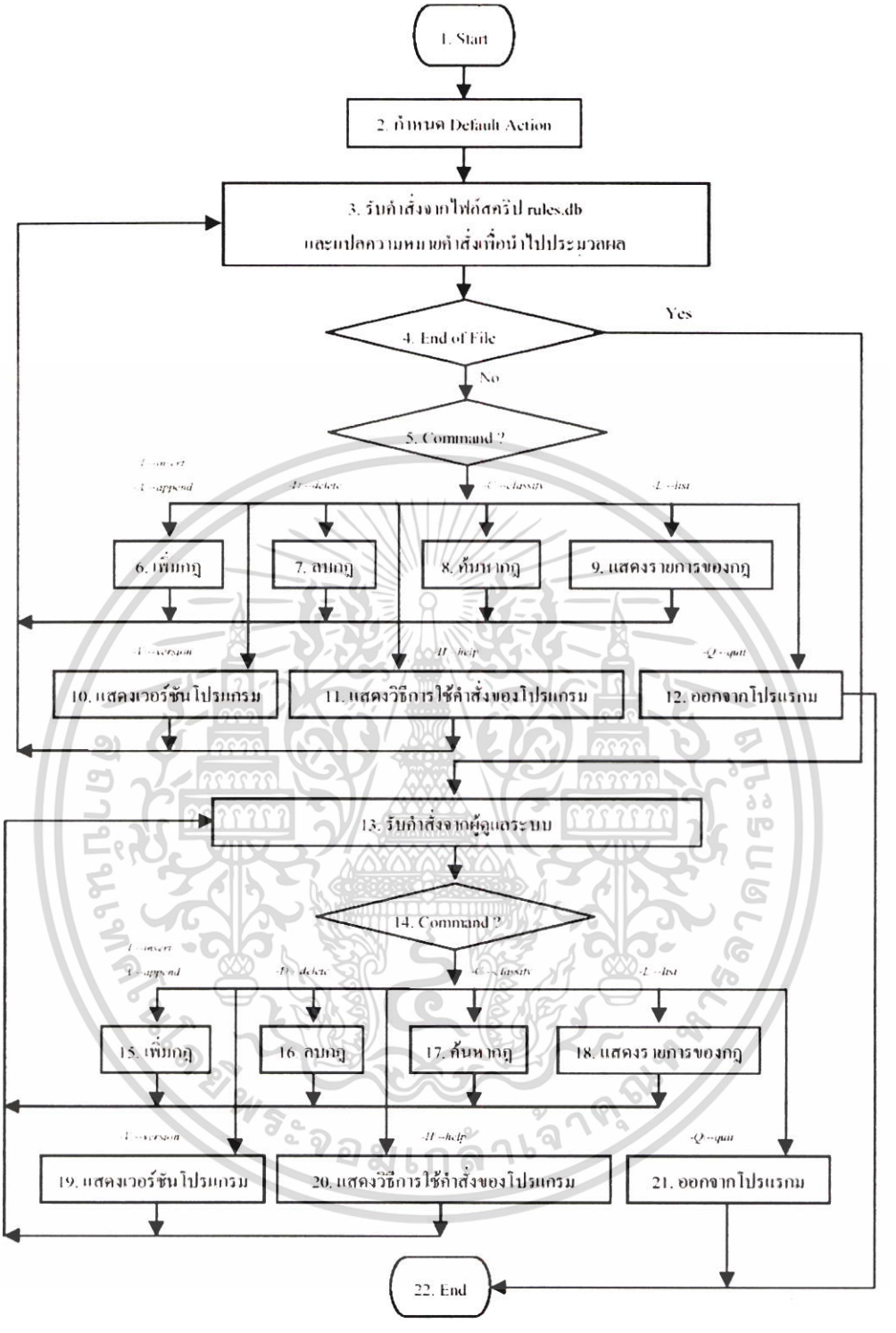
$$= \frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot \frac{N}{8} \cdot \left\{ t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la} \right) \right\} \cdot \left\{ t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right] \right\} \quad (11)$$

3.3 การพัฒนาโปรแกรม

เพื่อทดสอบแนวคิดในงานวิจัยดังที่กล่าวมาข้างต้น ผู้วิจัยจึงทำการพัฒนาโปรแกรม BIL ขึ้นด้วยตัวแปรภาษา gcc บนระบบปฏิบัติการ Linux Kernel 2.4 โดยในการวัดประสิทธิภาพการทำงานของโปรแกรมทั้งในด้านการค้นหา การปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูล ผู้วิจัยเลือกที่จะใช้ฐานข้อมูลกฎ และแพ็กเก็ตสำหรับการจัดประเภทที่สร้างขึ้นมาเอง เพื่อให้เหมาะสมกับจุดประสงค์ของการทดลองแต่ละการทดลอง โดยจัดเก็บข้อมูลเหล่านั้นลงในไฟล์สคริปชื่อ rules.db สำหรับฐานข้อมูลกฎ และไฟล์ไบนารีชื่อ dump.db สำหรับฐานข้อมูลแพ็กเก็ต เนื่องจากต้องการควบคุมตัวแปรแทรกซ้อนที่อาจจะมาจากข้อมูลบนเครือข่าย และอุปกรณ์บนเครือข่าย หรือจากตัว Kernel ของระบบปฏิบัติการเอง ไม่ให้มีผลกระทบต่อการทำงานของโปรแกรม ทำให้สามารถวัดเวลาในการค้นหา และเวลาการปรับปรุงกฎได้อย่างคลาดเคลื่อนน้อยที่สุด และเพื่อทำการทดสอบและเปรียบเทียบประสิทธิภาพการทำงานในด้านต่างๆ กับวิธีการ Linear Search และวิธีการ Bitmap-intersection ผู้วิจัยจึงต้องทำการพัฒนาโปรแกรมเหล่านั้นขึ้นบนสภาพแวดล้อมเดียวกัน ซึ่งจะแสดงแผนผังลำดับการทำงานของโปรแกรมทั้ง 3 ดังในรูปที่ 3.9 และแสดงอัลกอริทึมของโปรแกรม Linear Search และ โปรแกรม Bitmap-intersection ดังภาคผนวก ข.

ในการพัฒนาโปรแกรม BIL โปรแกรม Linear Search และโปรแกรม Bitmap-intersection ทั้ง 3 โปรแกรมประกอบด้วยฟังก์ชันพื้นฐานที่สำคัญทั้งหมด 9 ฟังก์ชัน ได้แก่

1. ฟังก์ชัน do_classify ทำหน้าที่อ่านข้อมูลแพ็กเก็ตจากไฟล์ dump.db และเตรียมข้อมูลเฮดเดอร์ของแพ็กเก็ตส่งไปให้ฟังก์ชัน classify ต่อไป
2. ฟังก์ชัน classify ทำการจัดประเภทแพ็กเก็ตตามกฎ และส่งผลลัพธ์ ซึ่งได้แก่ แอ็คชันและหมายเลขของกฎกลับมา
3. ฟังก์ชัน initial กำหนดค่า Default Action (ในกรณีที่การค้นหาไม่ตรงกับกฎข้อใดเลย โปรแกรมจะกำหนดแอ็คชันเท่ากับ Default Action) อ่านคำสั่งจากไฟล์สคริป rules.db และเตรียมข้อมูลส่งไปให้ฟังก์ชัน do_command ต่อไป
4. ฟังก์ชัน do_command ทำหน้าที่รับคำสั่งการทำงานจากผู้ดูแลระบบ หรือจากไฟล์สคริป และแปลความหมายคำสั่งเหล่านั้นเพื่อนำไปประมวลผลต่อไป
5. ฟังก์ชัน set_policy ทำหน้าที่กำหนด Default Action
6. ฟังก์ชัน append ทำหน้าที่เพิ่มกฎแบบ Append
7. ฟังก์ชัน insert ทำหน้าที่เพิ่มกฎแบบ Insert
8. ฟังก์ชัน list_rule ทำหน้าที่แสดงรายการของกฎทั้งหมดทางหน้าจอ
9. ฟังก์ชัน delete_rule ทำหน้าที่ลบกฎ



รูปที่ 3.9 แผนผังลำดับการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

จากแผนผังลำดับการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL จะเริ่มการทำงาน (ฟังก์ชัน initial) โดย 1. กำหนดค่า Default Action ด้วยฟังก์ชัน set_policy 2. อ่านคำสั่งจากไฟล์สคริป rules.db และเตรียมข้อมูลส่งไปให้

ฟังก์ชัน do_command โดยปกติคำสั่งจากไฟล์สคริป rules.db คือคำสั่งการเพิ่มกฎแบบ Insert (ฟังก์ชัน insert) หรือ Append (ฟังก์ชัน append) ดังตัวอย่างในรูปที่ 3.2 นอกจากนี้หากมีจุดประสงค์เป็นอย่างอื่น ก็สามารถกำหนดคำสั่งอื่นๆ เช่น คำสั่งลบกฎ (ฟังก์ชัน delete_rule) คำสั่งค้นหา (ฟังก์ชัน classify) คำสั่งแสดงรายการกฎ (ฟังก์ชัน list_rule) ในไฟล์สคริป rules.db ได้เช่นกัน โดยฟังก์ชันต่างๆ ของแต่ละโปรแกรมจะมีอัลกอริทึมที่แตกต่างกันตามวิธีการที่ได้กล่าวมาก่อนหน้านี้ และหลังจากอ่านคำสั่งในไฟล์สคริป rules.db โปรแกรมจะรอรับคำสั่งจากผู้ดูแลระบบหรือผู้ใช้งานต่อไป

บทที่ 4

การทดลองและผลการทดลอง

ในบทนี้จะกล่าวถึงการทดลองเพื่อวัดประสิทธิภาพการทำงานของโปรแกรม สำหรับทดสอบแนวคิดในงานวิจัยทั้งในด้านการค้นหา การปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูล

4.1 การออกแบบการทดลองและเครื่องมือที่ใช้ในการทดลอง

4.1.1 การออกแบบการทดลอง

ในการออกแบบการทดลอง ผู้วิจัยได้แบ่งการทดลองออกเป็น 2 ส่วน โดยการทดลองส่วนที่ 1. เป็นการแบ่งขนาด Prefix เป็น Block เล็กๆ (เช่น แบ่ง Source IP Address เป็นขนาด 16+16 บิต) เพื่อหาค่าที่ทำให้โปรแกรม BIL มีประสิทธิภาพในการทำงานดีที่สุด การทดลองส่วนที่ 2. เป็นการเปรียบเทียบประสิทธิภาพการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยในส่วนที่ 1 จะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) และ Source MAC Address (ขนาด 48 บิต) ส่วนที่ 2 จะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) ซึ่งในการศึกษาจะทำการสร้างข้อมูลสำหรับการวัดเวลาการค้นหา และเวลาการปรับปรุงกฎ เป็น 3 กรณี ได้แก่ 1. กรณี Best Case 2. กรณี Average Case และ 3. กรณี Worst Case โดยมีรายละเอียดดังต่อไปนี้

ข้อมูลสำหรับการวัดเวลาการค้นหาของโปรแกรม Linear Search และโปรแกรม Bitmap-intersection

1. กรณี Best Case กำหนด Source IP Address ของแพ็กเก็ตให้ตรงกับกฎข้อแรกในฐานข้อมูลกฎ

2. กรณี Average Case กำหนด Source IP Address ของแพ็กเก็ตให้ตรงกับกฎข้อใดๆ ในฐานข้อมูลกฎ โดยการสุ่ม

3. กรณี Worst Case กำหนด Source IP Address ของแพ็กเก็ตให้ตรงกับกฎข้อสุดท้ายในฐานข้อมูลกฎ

ข้อมูลสำหรับการวัดเวลาการค้นหาของโปรแกรม BIL

1. กรณี Best Case กำหนด Source IP Address และ Source MAC Address ของแพ็กเก็ตให้ตรงกับกฎข้อแรกในฐานข้อมูลกฎ

2. กรณี Average Case กำหนด Source IP Address และ Source MAC Address ของแพ็กเก็ตให้ตรงกับกฎข้อใดๆ ในฐานข้อมูลกฎ โดยการสุ่ม

3. กรณี Worst Case กำหนด Source IP Address และ Source MAC Address ของแพ็กเก็ตให้ตรงกับกฎข้อสุดท้ายในฐานข้อมูลกฎ

ข้อมูลสำหรับการวัดเวลาการเพิ่มกฎของโปรแกรม Linear Search

1. กรณี Best Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยเรียงลำดับจากน้อยไปมาก

2. กรณี Average Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยการสุ่ม

3. กรณี Worst Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยเรียงลำดับจากมากไปน้อย

ข้อมูลสำหรับการวัดเวลาการลบกฎของโปรแกรม Linear Search

1. กรณี Best Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยเรียงลำดับจากมากไปน้อย

2. กรณี Average Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยการสุ่ม

3. กรณี Worst Case กำหนดหมายเลขของกฎในฐานข้อมูลโดยเรียงลำดับจากน้อยไปมาก

ข้อมูลสำหรับการวัดเวลาการปรับปรุงกฎ (การเพิ่มกฎและการลบกฎ) ของโปรแกรม Bitmap-intersection

1. กรณี Best Case กำหนด Source IP Address ของกฎในแต่ละข้อ เป็น 0.0.0.0/0

2. กรณี Average Case กำหนด Source IP Address ของกฎในแต่ละข้อ จำนวน 1 หมายเลขโดยการสุ่ม

3. กรณี Worst Case กำหนด Source IP Address ของกฎในแต่ละข้อเป็นช่วงโดยการสุ่ม

ข้อมูลสำหรับการวัดเวลาการปรับปรุงกฎ (การเพิ่มกฎและการลบกฎ) ของโปรแกรม BIL

1. กรณี Best Case กำหนด Source IP Address และ Source MAC Address ของกฎในแต่ละข้อ จำนวน 1 หมายเลขโดยการสุ่ม

2. กรณี Average Case กำหนด Source IP Address และ Source MAC Address ของกฎในแต่ละข้อเป็นช่วงโดยการสุ่ม

3. กรณี Worst Case กำหนด Source IP Address และ Source MAC Address ของกฎในแต่ละข้อเป็นช่วงโดยให้มีจำนวนสมาชิกของ Prefix มากที่สุด

4.1.2 เครื่องมือที่ใช้ในการทดลอง

สำหรับส่วนฮาร์ดแวร์ ประกอบด้วย

1. เครื่องคอมพิวเตอร์ส่วนบุคคลจำนวน 1 เครื่อง หน่วยประมวลผลกลาง Pentium 4 ความเร็ว 1.60 GHz หน่วยความจำหลัก 256 MB และฮาร์ดดิสก์ขนาด 10 GB

สำหรับส่วนซอฟท์แวร์ ประกอบด้วย

1. ระบบปฏิบัติการ Linux Kernel 2.4
2. ตัวแปรภาษา gcc

4.2 การแบ่งขนาด Prefix เพื่อหาค่าที่ทำให้โปรแกรมมีประสิทธิภาพในการทำงานดีที่สุด

จากแนวคิดในงานวิจัยนี้พบว่าความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลของโปรแกรม BIL จะขึ้นอยู่กับขนาดของ Prefix ที่ถูกกำหนดไว้เป็นสำคัญ ดังนั้นผู้วิจัยจะทำการศึกษาขนาดของ Prefix ที่ทำให้โปรแกรมมีประสิทธิภาพดีที่สุด โดยจัดการทดลองขึ้นจำนวน 3 การทดลองซึ่งทำการแบ่ง Prefix ในลักษณะต่างๆ กัน ได้แก่

1. การทดลองแบ่งขนาด Prefix แบบคงที่ โดยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address และ Source MAC Address
2. การทดลองแบ่งขนาด Prefix แบบปรับเปลี่ยนได้
3. การทดลองวัดประสิทธิภาพการทำงานของโปรแกรม BIL กับฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

4.2.1 การทดลองแบ่งขนาด Prefix แบบคงที่ โดยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address และ Source MAC Address

วัตถุประสงค์การทดลอง เพื่อหาขนาดของ Prefix แบบคงที่ (แต่ละ Block มีขนาดเท่ากัน) ซึ่งทำให้โปรแกรม BIL มีความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลดีที่สุด สำหรับ Source IP Address (ขนาด 32 บิต) และ Source MAC Address (ขนาด 48 บิต) ที่จำนวนกฎ 4,096 ข้อ

วิธีการทดลอง

สำหรับ Source IP Address

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. แบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน โดยกำหนดให้ PMX เท่ากับ 2 บิต 4 บิต 8 บิต และ 16 บิต ตามลำดับ
3. กำหนดจำนวนกฎเท่ากับ $2^{12} = 4,096$ ข้อ (Bitvector ขนาด 512 ไบต์)
4. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Best Case
5. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Average Case
6. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Worst Case
7. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต
8. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case

9. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case

10. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case

สำหรับ Source MAC Address

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source MAC Address (ขนาด 48 บิต)

2. แบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน โดยกำหนดให้ PMX เท่ากับ 2 บิต 3 บิต 4 บิต 6 บิต 8 บิต 12 บิต และ 16 บิต ตามลำดับ

3. กำหนดจำนวนกฎเท่ากับ 2^{12} = 4,096 ข้อ (Bitvector ขนาด 512 ไบต์)

4. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Best Case

5. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Average Case

6. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Worst Case

7. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต

8. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case

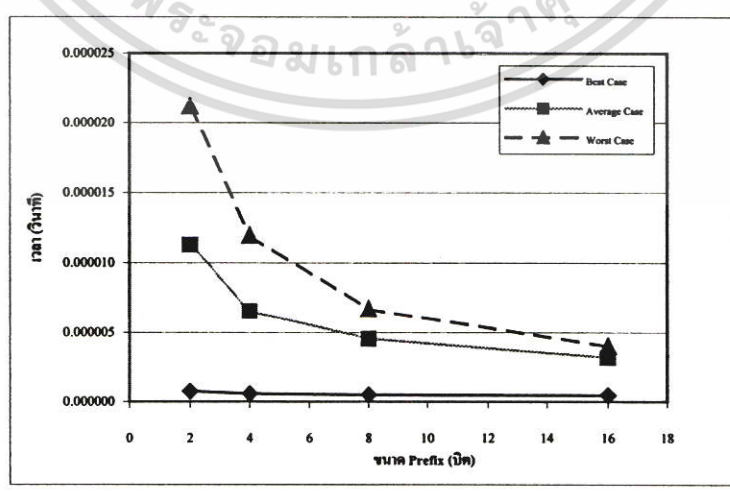
9. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case

10. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case

ผลการทดลอง

ตารางที่ 4.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) สำหรับ Source IP Address

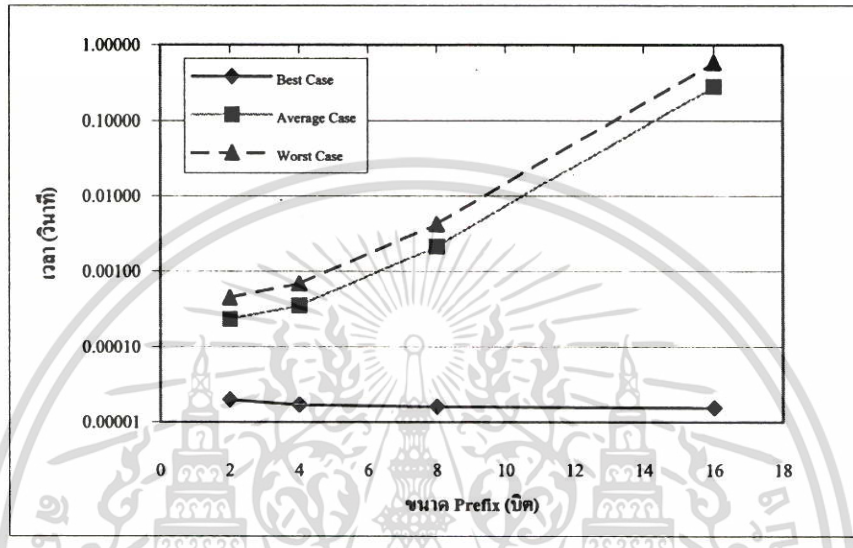
กรณี	ขนาด Prefix (บิต)			
	2	4	8	16
Best Case (วินาที)	0.000000748	0.000000574	0.000000494	0.000000462
Average Case (วินาที)	0.000011299	0.000006555	0.000004548	0.000003200
Worst Case (วินาที)	0.000021176	0.000011942	0.000006686	0.000003997



รูปที่ 4.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) สำหรับ Source IP Address

ตารางที่ 4.2 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) สำหรับ Source IP Address

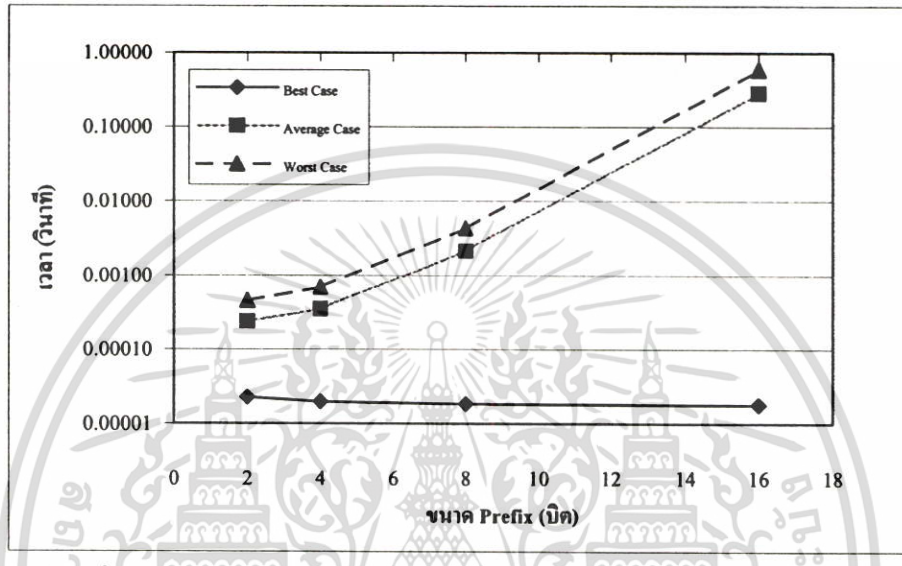
กรณี	ขนาด Prefix (บิต)			
	2	4	8	16
Best Case (วินาที)	0.000019864244200	0.000016981860806	0.000015954417582	0.000015421880342
Average Case (วินาที)	0.000232681169719	0.000350491934066	0.002128886884005	0.284997731843712
Worst Case (วินาที)	0.000448200239316	0.000686301995116	0.004239103406593	0.587700832478632



รูปที่ 4.2 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม

ตารางที่ 4.3 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) สำหรับ Source IP Address

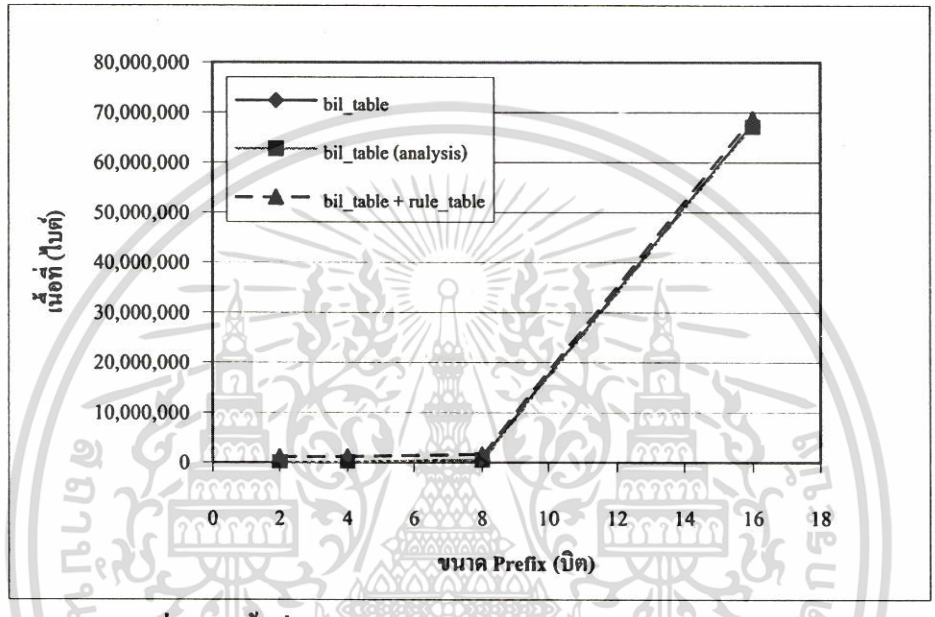
กรณี	ขนาด Prefix (บิต)			
	2	4	8	16
Best Case (วินาที)	0.000022744461538	0.000019780251526	0.000018434349206	0.000017842590965
Average Case (วินาที)	0.000238564170940	0.000354317355311	0.002127146673993	0.282898271848596
Worst Case (วินาที)	0.000457851018315	0.000691261670330	0.004281886424908	0.584254819617420



รูปที่ 4.3 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม

ตารางที่ 4.4 เนื้อที่จัดเก็บข้อมูล (ไบต์) สำหรับ Source IP Address

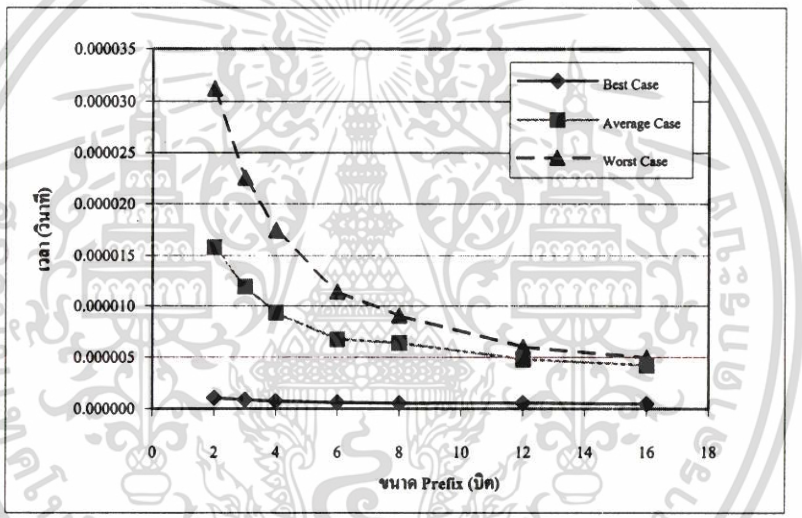
Data Structure	ขนาด Prefix (บิต)			
	2	4	8	16
bil_table (ไบต์)	33,024	66,048	528,384	67,633,152
rule_table (ไบต์)	1,097,728	1,097,728	1,097,728	1,097,728
bil_table analysis (ไบต์)	32,768	65,536	524,288	67,108,864
bil_table + rule_table (ไบต์)	1,130,752	1,163,776	1,626,112	68,730,880



รูปที่ 4.4 เนื้อที่จัดเก็บข้อมูล (ไบต์) สำหรับ Source IP Address

ตารางที่ 4.5 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) สำหรับ Source MAC Address

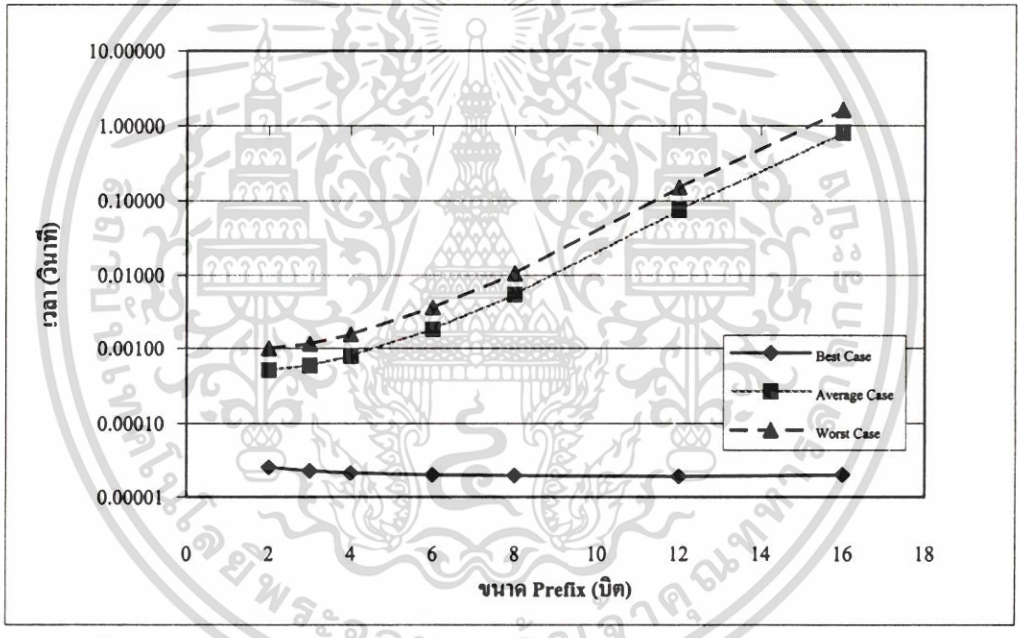
กรณี	ขนาด Prefix (บิต)						
	2	3	4	6	8	12	16
Best Case (วินาที)	0.000001061	0.000000881	0.000000750	0.000000631	0.000000555	0.000000575	0.000000479
Average Case (วินาที)	0.000015853	0.000011923	0.000009328	0.000006718	0.000006372	0.000004799	0.000004235
Worst Case (วินาที)	0.000031217	0.000022526	0.000017483	0.000011410	0.000009051	0.000005994	0.000004956



รูปที่ 4.5 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) สำหรับ Source MAC Address

ตารางที่ 4.6 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) สำหรับ Source MAC Address

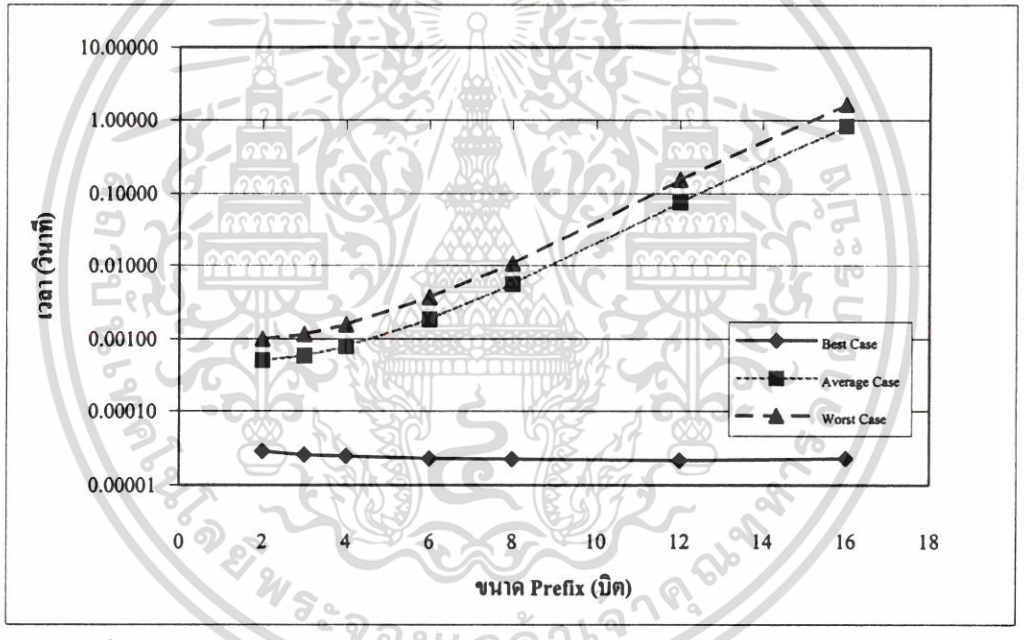
กรณี	ขนาด Prefix (บิต)						
	2	3	4	6	8	12	16
Best Case (วินาที)	0.000025332542125	0.000022616761905	0.000021162568987	0.000019905435897	0.000019593111111	0.000019004036630	0.000019919111111
Average Case (วินาที)	0.000508868605617	0.000587972808303	0.000788225096459	0.001829766945055	0.005478409037851	0.075027683203907	0.816362080229548
Worst Case (วินาที)	0.000997232354090	0.001151983787546	0.001553739135531	0.003639867203907	0.010557661750916	0.151480182647694	1.633706101831500



รูปที่ 4.6 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) สำหรับ Source MAC Address โดยใช้มาตราส่วนลอการิทึม

ตารางที่ 4.7 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) สำหรับ Source MAC Address

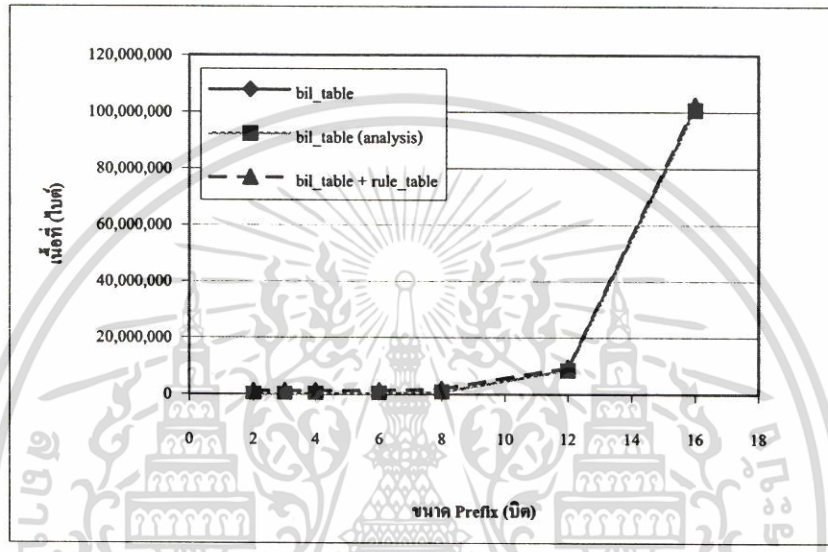
กรณี	ขนาด Prefix (บิต)						
	2	3	4	6	8	12	16
Best Case (วินาที)	0.000028503052503	0.000025502285714	0.000024433555556	0.000022774378510	0.000022390478632	0.000021485457875	0.000022762871795
Average Case (วินาที)	0.000511647255189	0.000590023428571	0.000791075658120	0.001834405728938	0.005539494681319	0.075511994214896	0.825536942925519
Worst Case (วินาที)	0.000999760258852	0.001160799533578	0.001560857118437	0.003649649255189	0.010650957125763	0.153089696900535	1.653285198192920



รูปที่ 4.7 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) สำหรับ Source MAC Address โดยใช้มาตราส่วนลอการิทึม

ตารางที่ 4.8 เนื้อที่จัดเก็บข้อมูล (ไบต์) สำหรับ Source MAC Address

Data Structure	ขนาด Prefix (บิต)						
	2	3	4	6	8	12	16
bil_table (ไบต์)	49,536	66,048	99,072	264,192	792,576	8,454,144	101,449,728
rule_table (ไบต์)	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728
bil_table analysis (ไบต์)	49,152	65,536	98,304	262,144	786,432	8,388,608	100,663,296
bil_table + rule_table (ไบต์)	1,147,264	1,163,776	1,196,800	1,361,920	1,890,304	9,551,872	102,547,456



รูปที่ 4.8 เนื้อที่จัดเก็บข้อมูล (ไบต์) สำหรับ Source MAC Address

วิเคราะห์ผลการทดลอง

1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ โดยกำหนดขนาด PMX เป็น 2 บิต 4 บิต 8 บิต และ 16 บิต สำหรับ Source IP Address (W เท่ากับ 32 บิต) และกำหนดขนาด PMX เป็น 2 บิต 3 บิต 4 บิต 6 บิต 8 บิต 12 บิต และ 16 บิต สำหรับ Source MAC Address (W เท่ากับ 48 บิต) โดยจากการวิเคราะห์ในกรณี Best Case โปรแกรม BIL จะมี Search Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX}\right)$ และในกรณี Worst Case จะมี Search Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}\right)$ ซึ่งจะเห็นว่าที่จำนวน N คงที่ ความเร็วในการค้นหาจะขึ้นอยู่กับขนาดของ PMX โดยยิ่งกำหนดค่า PMX สูงขึ้น โปรแกรม BIL จะใช้เวลาในการค้นหาลดลง จากผลการทดลองในตารางที่ 4.1 และ 4.5 ที่ PMX เท่ากับ 16 บิต โปรแกรม BIL ใช้เวลาในการค้นหาน้อยที่สุดในทุกกรณี ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

2. จากการวิเคราะห์ในกรณี Best Case โปรแกรม BIL จะมี Update Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX}\right)$ ดังนั้นเมื่อ PMX มีค่ามากขึ้น โปรแกรมจะใช้เวลาในการปรับปรุงกฎลดลง

จากผลการทดลองในตารางที่ 4.2, 4.3, 4.6 และ 4.7 ที่ PMX เท่ากับ 16 บิต โปรแกรม BIL จะใช้เวลาในการปรับปรุงกฎน้อยที่สุด ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

3. กรณี Worst Case โปรแกรม BIL จะมี Update Speed Complexity เท่ากับ $O\left((2\cdot W-2)\cdot\frac{D\cdot W}{PMX}\cdot 2^{PMX}\right)$ ดังนั้นเมื่อ PMX มีค่ามากขึ้น โปรแกรมจะใช้เวลาในการปรับปรุงกฎเพิ่มขึ้นเช่นกัน จากผลการทดลองในตารางที่ 4.2, 4.3, 4.6 และ 4.7 ที่ PMX เท่ากับ 16 บิต โปรแกรม BIL จะใช้เวลาในการปรับปรุงกฎมากที่สุด ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

4. จากการวิเคราะห์ โปรแกรม BIL จะมี Storage Complexity เท่ากับ $O\left(\frac{D\cdot W}{PMX}\cdot 2^{PMX}\cdot N\right)$ ดังนั้นเมื่อ PMX มีค่ามากขึ้น โปรแกรม BIL จะใช้เนื้อที่จัดเก็บข้อมูลเพิ่มขึ้นเช่นกัน จากผลการทดลองในตารางที่ 4.4 และ 4.8 ที่ PMX เท่ากับ 16 บิต โปรแกรมจะใช้เนื้อที่จัดเก็บข้อมูลมากที่สุด ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

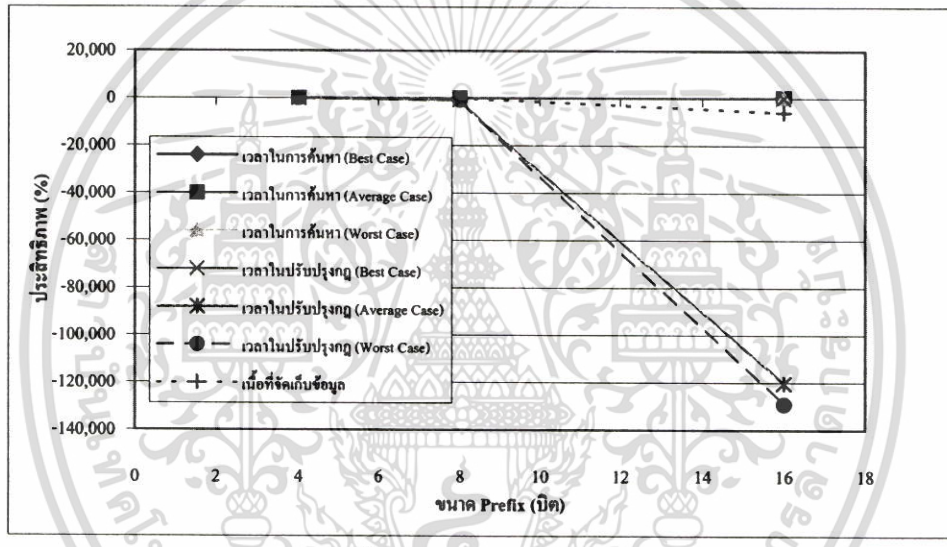
5. จากผลการทดลองในตารางที่ 4.1-4.8 สามารถแสดงประสิทธิภาพของโปรแกรมที่ PMX ขนาดต่างๆ เปรียบเทียบกับ PMX ขนาด 2 บิต (เช่น จากตารางที่ 4.1 ที่ PMX เท่ากับ 4 บิตจะมีเวลาในการค้นหาในกรณี Best Case มากกว่า 2 บิต เท่ากับ $[(\textit{เวลาในการค้นหาที่ 2 บิต - เวลาในการค้นหาที่ 4 บิต})/\textit{เวลาในการค้นหาที่ 4 บิต})] \times 100 = [(0.000000748 - 0.000000574)/0.000000574] \times 100 = 30.31\,\%$) ดังตารางที่ 4.9-4.10 และ รูปที่ 4.9-4.10 ซึ่งจากผลการทดลองและสมมติฐานที่กล่าวมาข้างต้นสามารถแบ่ง Complexity ของโปรแกรม BIL ออกเป็น 2 ประเภท ได้แก่

- ประเภทที่ 1 ให้ประสิทธิภาพที่ดีเมื่อกำหนด PMX สูง คือ Search Speed Complexity สำหรับทุกๆ กรณี และ Update Speed Complexity สำหรับกรณี Best Case

- ประเภทที่ 2 ให้ประสิทธิภาพที่ดีเมื่อกำหนด PMX ต่ำ คือ Update Speed Complexity สำหรับกรณี Average Case กับกรณี Worst Case และ Storage Complexity

ตารางที่ 4.9 ประสิทธิภาพของโปรแกรมที่ PMX เท่ากับ 4 บิต 8 บิต และ 16 บิต เปรียบเทียบกับ 2 บิต สำหรับ Source IP Address

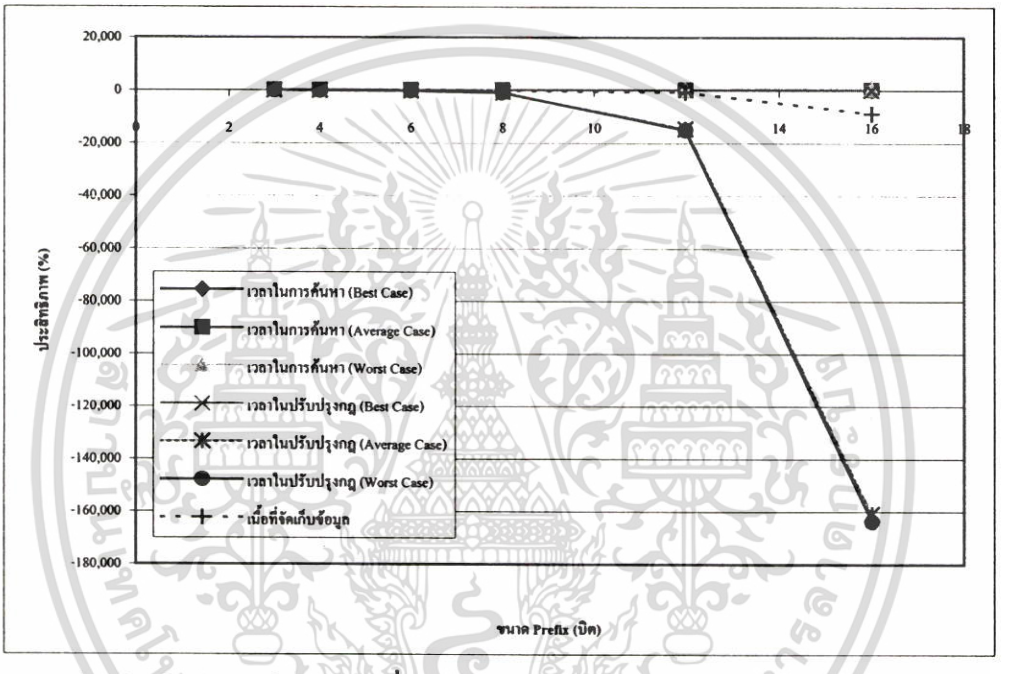
ประสิทธิภาพเปรียบเทียบกับ 2 บิต	ขนาด Prefix (บิต)		
	4	8	16
เวลาในการค้นหา (Best Case)	มากกว่า 30.31 %	มากกว่า 51.42 %	มากกว่า 61.90 %
เวลาในการค้นหา (Average Case)	มากกว่า 72.37 %	มากกว่า 148.44 %	มากกว่า 253.09 %
เวลาในการค้นหา (Worst Case)	มากกว่า 77.32 %	มากกว่า 216.72 %	มากกว่า 429.80 %
เวลาในปรับปรุงกฎ (Best Case)	มากกว่า 15.90 %	มากกว่า 23.90 %	มากกว่า 28.09 %
เวลาในปรับปรุงกฎ (Average Case)	น้อยกว่า 49.56 %	น้อยกว่า 803.15 %	น้อยกว่า 120,409.63 %
เวลาในปรับปรุงกฎ (Worst Case)	น้อยกว่า 52.04 %	น้อยกว่า 840.45 %	น้อยกว่า 129,247.61 %
เนื้อที่จัดเก็บข้อมูล	น้อยกว่า 2.92 %	น้อยกว่า 43.81 %	น้อยกว่า 5,978.33 %



รูปที่ 4.9 ประสิทธิภาพของโปรแกรมที่ PMX เท่ากับ 4 บิต 8 บิต และ 16 บิต เปรียบเทียบกับ 2 บิต สำหรับ Source IP Address

ตารางที่ 4.10 ประสิทธิภาพของโปรแกรมที่ PMX เท่ากับ 3 บิต 4 บิต 6 บิต 8 บิต 12 บิต และ 16 บิต เปรียบเทียบกับ 2 บิต สำหรับ Source MAC Address

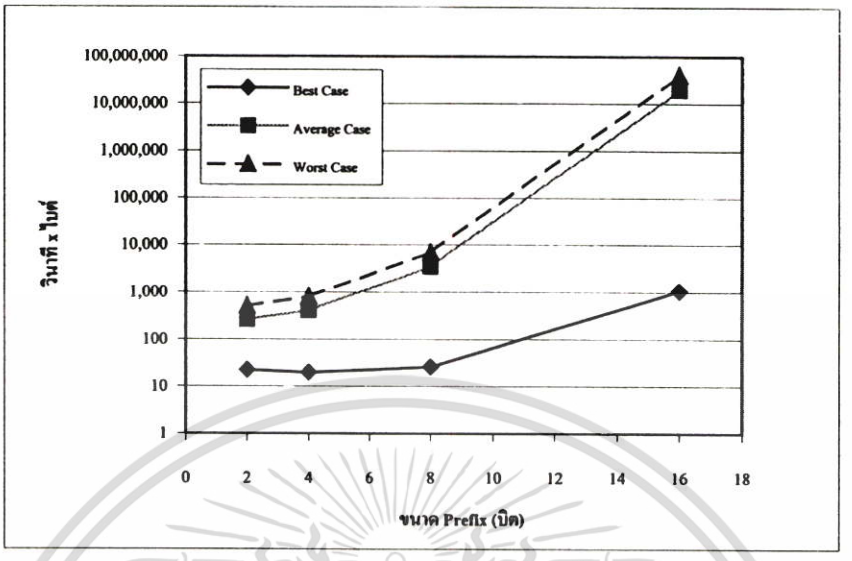
ประสิทธิภาพเปรียบเทียบกับ 2 บิต	ขนาด Prefix (บิต)					
	3	4	6	8	12	16
เวลาในการค้นหา (Best Case)	มากกว่า 20.43 %	มากกว่า 41.47 %	มากกว่า 68.15 %	มากกว่า 91.17 %	มากกว่า 84.52 %	มากกว่า 121.50 %
เวลาในการค้นหา (Average Case)	มากกว่า 32.96 %	มากกว่า 69.95 %	มากกว่า 135.98 %	มากกว่า 148.79 %	มากกว่า 230.34 %	มากกว่า 274.33 %
เวลาในการค้นหา (Worst Case)	มากกว่า 38.58 %	มากกว่า 78.56 %	มากกว่า 173.59 %	มากกว่า 244.90 %	มากกว่า 420.80 %	มากกว่า 529.88 %
เวลาในปรับปรุงกฎ (Best Case)	มากกว่า 11.88 %	มากกว่า 18.07 %	มากกว่า 26.14 %	มากกว่า 28.23 %	มากกว่า 32.96 %	มากกว่า 26.13 %
เวลาในปรับปรุงกฎ (Average Case)	น้อยกว่า 15.43 %	น้อยกว่า 54.76 %	น้อยกว่า 259.05 %	น้อยกว่า 979.64 %	น้อยกว่า 14,651.33 %	น้อยกว่า 160,789.12 %
เวลาในปรับปรุงกฎ (Worst Case)	น้อยกว่า 15.52 %	น้อยกว่า 55.81 %	น้อยกว่า 265.00 %	น้อยกว่า 958.70 %	น้อยกว่า 15,090.06 %	น้อยกว่า 163,724.02 %
เนื้อที่จัดเก็บข้อมูล	น้อยกว่า 1.44 %	น้อยกว่า 4.32 %	น้อยกว่า 18.71 %	น้อยกว่า 64.77 %	น้อยกว่า 732.58 %	น้อยกว่า 8,838.44 %



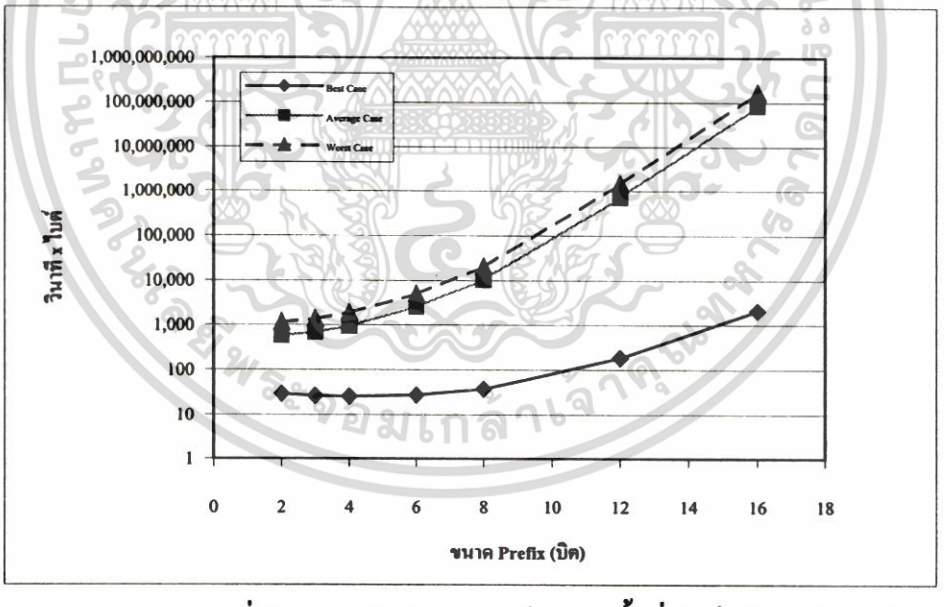
รูปที่ 4.10 ประสิทธิภาพของโปรแกรมที่ PMX เท่ากับ 3 บิต 4 บิต 6 บิต 8 บิต 12 บิต และ 16 บิต เปรียบเทียบกับ 2 บิต สำหรับ Source MAC Address

จากผลการทดลองดังกล่าว เพื่อหาขนาด Prefix ที่เหมาะสม ผู้วิจัยจะทำการศึกษาประสิทธิภาพในการทำงานของโปรแกรมโดยพิจารณาจากผลคูณของประสิทธิภาพการทำงานจำนวน 4 ผลคูณ ตามที่เคยกล่าวในบทก่อนหน้า

1. ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



รูปที่ 4.11 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบคงที่สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม

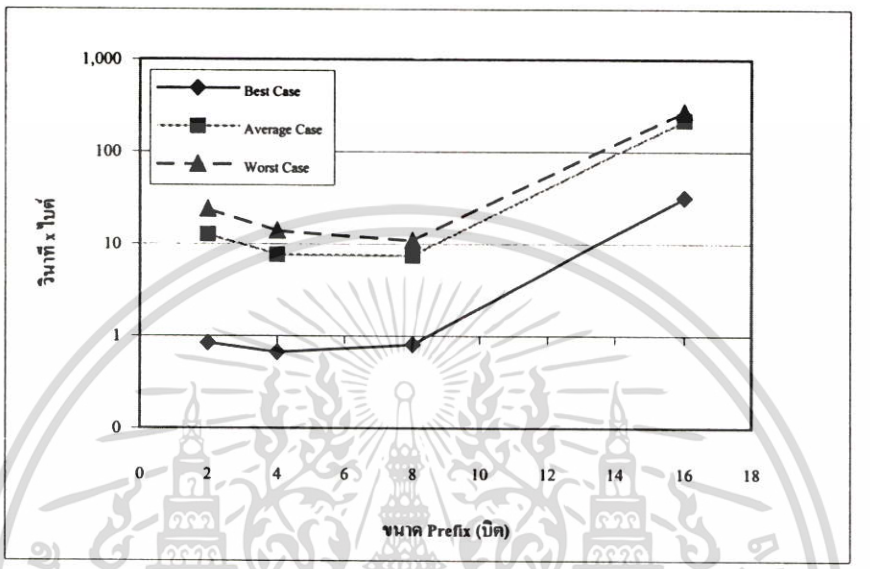


รูปที่ 4.12 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบคงที่สำหรับ Sourec MAC Address โดยใช้มาตราส่วนลอการิทึม

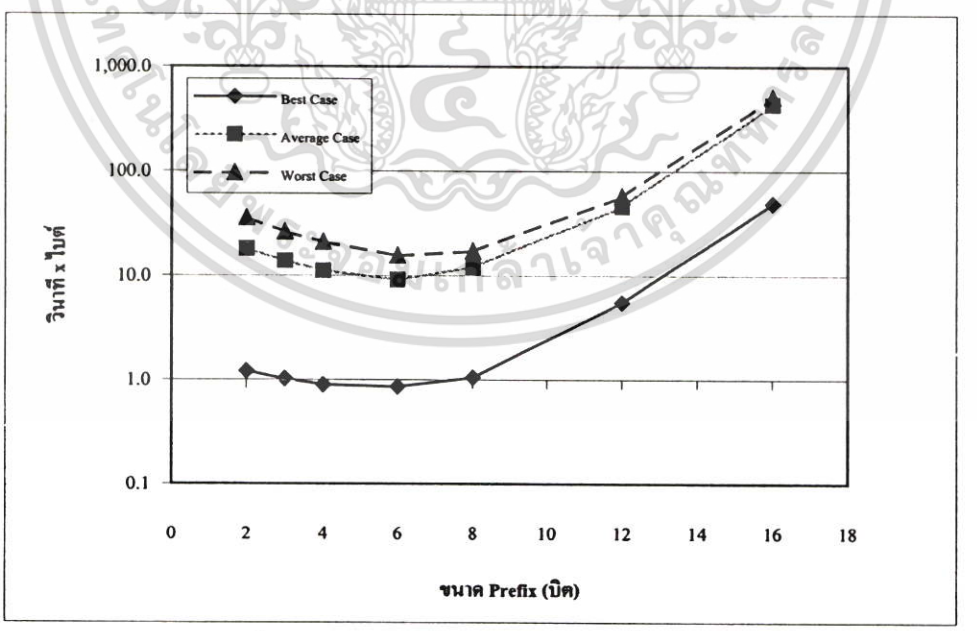
จากกราฟในรูปที่ 4.11 และ 4.12 พบว่าในกรณี Best Case ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำที่สุดที่ขนาด PMX เท่ากับ 4 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการปรับปรุงกฎโดยจะมีประสิทธิภาพสูงเมื่อกำหนด PMX สูงขึ้น ในขณะที่ในกรณี

Average Case และกรณี Worst Case ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำที่สุดที่ขนาด PMX เท่ากับ 2 บิต และมีค่ามากขึ้นเมื่อ PMX มีค่าสูงขึ้นตามลำดับ โดยทั้ง 3 กรณีที่ PMX เท่ากับ 16 บิต เส้นกราฟจะมีค่าสูงขึ้นอย่างเห็นได้

2. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล



รูปที่ 4.13 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบคงที่สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม

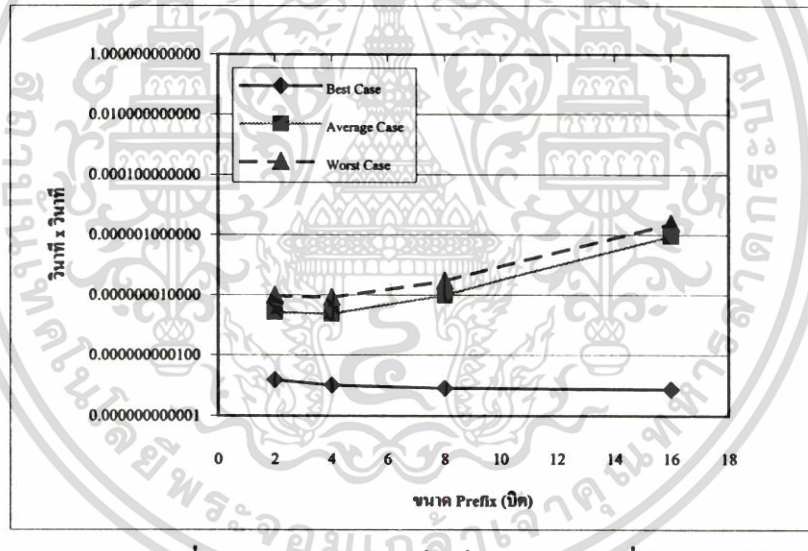


รูปที่ 4.14 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบคงที่สำหรับ Sourec MAC Address โดยใช้มาตราส่วนลอการิทึม

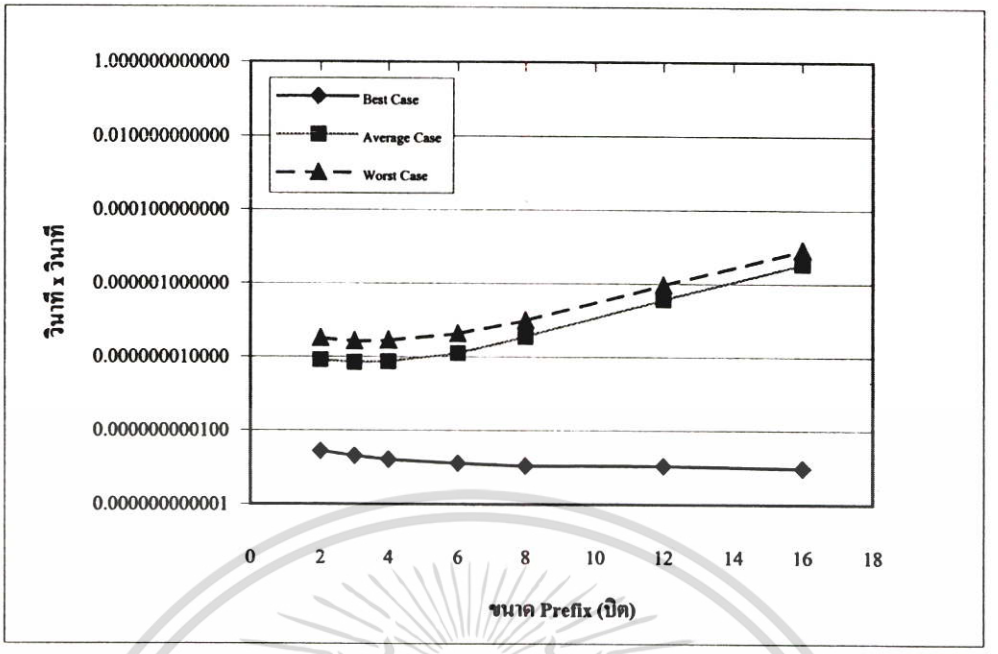
จากกราฟในรูปที่ 4.13 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูลสำหรับ Sourec IP Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 4 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการค้นหาโดยจะมีประสิทธิภาพสูงเมื่อกำหนด PMX สูงขึ้น ในขณะที่กรณี Average Case และกรณี Worst Case ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูลมีค่าต่ำที่สุดที่ PMX เท่ากับ 8 บิต โดยทั้ง 3 กรณีเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเนื้อที่จัดเก็บข้อมูล โดยจะมีประสิทธิภาพที่ต่ำเมื่อกำหนด PMX สูงขึ้น

จากกราฟในรูปที่ 4.14 พบว่าในทุกกรณีผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูลสำหรับ Sourec MAC Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 6 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการค้นหาโดยจะมีประสิทธิภาพสูงเมื่อกำหนด PMX สูงขึ้น และเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเนื้อที่จัดเก็บข้อมูล โดยจะมีประสิทธิภาพที่ต่ำเมื่อกำหนด PMX สูงขึ้น

3. ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ



รูปที่ 4.15 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ โดยแบ่ง Prefix แบบคงที่สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม

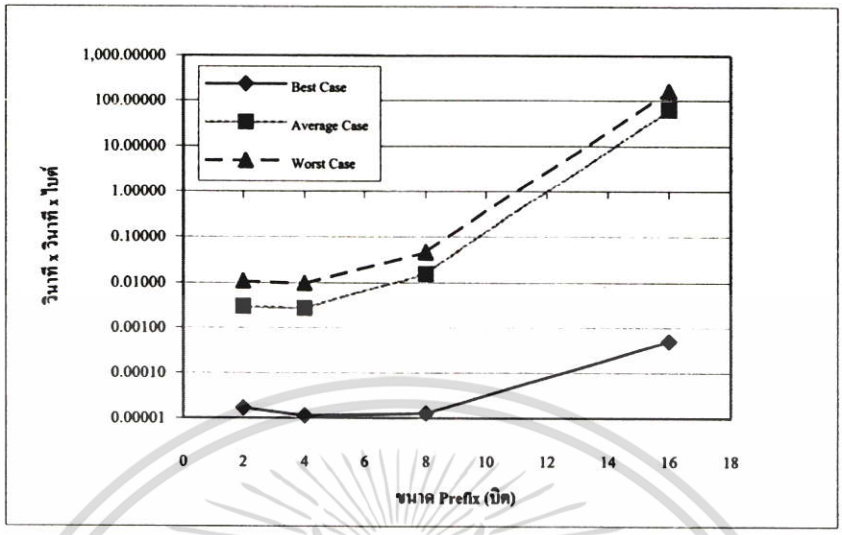


รูปที่ 4.16 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ โดยแบ่ง Prefix แบบคงที่สำหรับ Sourec MAC Address โดยใช้มาตราส่วนลอการิทึม

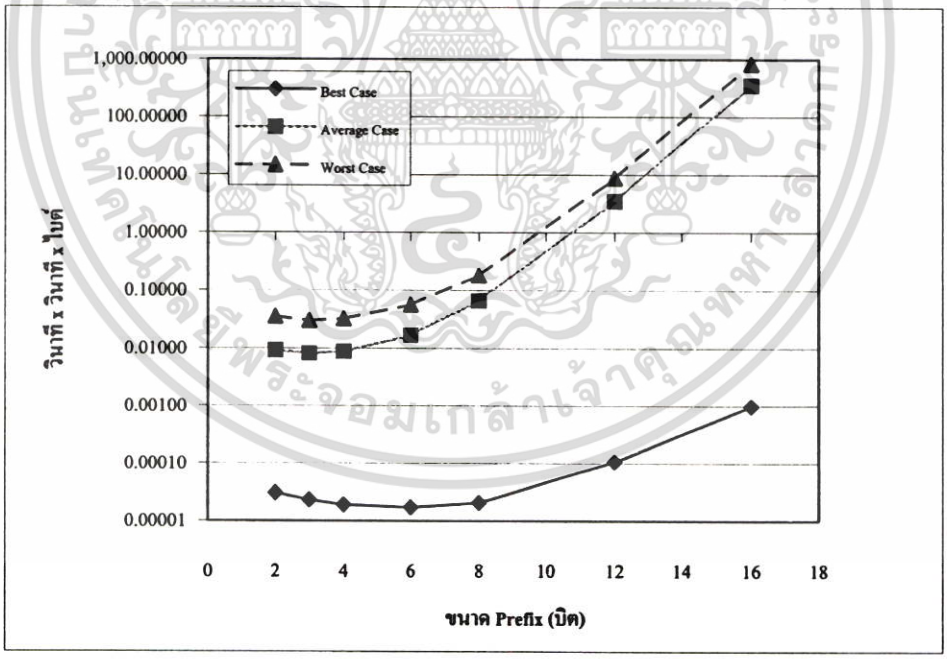
จากกราฟในรูปที่ 4.15 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎต่อข้อสำหรับ Sourec IP Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ (ในกรณี Best Case) โดยทั้งสองจะมีประสิทธิภาพที่สูงเมื่อกำหนด PMX สูงขึ้น ในขณะที่กรณี Average Case และกรณี Worst Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎต่อข้อมีค่าต่ำที่สุดที่ PMX เท่ากับ 4 บิต และเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการปรับปรุงกฎ โดยจะมีประสิทธิภาพที่ต่ำเมื่อกำหนด PMX สูงขึ้น

จากกราฟในรูปที่ 4.16 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎต่อข้อสำหรับ Sourec MAC Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ (ในกรณี Best Case) โดยทั้งสองจะมีประสิทธิภาพที่สูงเมื่อกำหนด PMX สูงขึ้น ในขณะที่กรณี Average Case และกรณี Worst Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎต่อข้อมีค่าต่ำที่สุดที่ PMX เท่ากับ 3 บิต และเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต ซึ่งเป็นผลมาจากอิทธิพลของเวลาในการปรับปรุงกฎ โดยจะมีประสิทธิภาพที่ต่ำเมื่อกำหนด PMX สูงขึ้น

4. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



รูปที่ 4.17 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบคงที่สำหรับ Source IP Address โดยใช้มาตราส่วนลอการิทึม



รูปที่ 4.18 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูล โดยแบ่ง Prefix แบบคงที่สำหรับ Sourec MAC Address โดยใช้มาตราส่วนลอการิทึม

จากกราฟในรูปที่ 4.17 พบว่าในทุกกรณี ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล สำหรับ Sourec IP Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 4 บิต และเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต

จากกราฟในรูปที่ 4.18 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล สำหรับ Sourec MAC Address มีค่าต่ำที่สุดที่ PMX เท่ากับ 6 บิต ในขณะที่กรณี Average Case และกรณี Worst Case ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล มีค่าต่ำที่สุดที่ PMX เท่ากับ 3 บิต โดยทั้ง 3 กรณีเส้นกราฟสูงที่สุดอย่างเห็นได้ชัดที่ PMX เท่ากับ 16 บิต

ตารางที่ 4.11 ขนาด Prefix ที่เหมาะสมสำหรับ Source IP Address โดยพิจารณาจากจุดต่ำสุดของผลคูณ

ผลคูณ	กรณี		
	Best Case	Average Case	Worst Case
1. เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	4 บิต	2 บิต	2 บิต
2. เวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล	4 บิต	8 บิต	8 บิต
3. เวลาในการค้นหาและเวลาในการปรับปรุงกฎ	16 บิต	4 บิต	4 บิต
4. เวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	4 บิต	4 บิต	4 บิต

ตารางที่ 4.12 ขนาด Prefix ที่เหมาะสมสำหรับ Source MAC Address โดยพิจารณาจากจุดต่ำสุดของผลคูณ

ผลคูณ	กรณี		
	Best Case	Average Case	Worst Case
1. เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	4 บิต	2 บิต	2 บิต
2. เวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล	6 บิต	6 บิต	6 บิต
3. เวลาในการค้นหาและเวลาในการปรับปรุงกฎ	16 บิต	3 บิต	3 บิต
4. เวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	6 บิต	3 บิต	3 บิต

จากตารางที่ 4.11 และ 4.12 พบว่าในกรณี Worst Case ที่ PMX เท่ากับ 2 บิต 4 บิต และ 8 บิต เป็นขนาด Prefix ที่เหมาะสมสำหรับ Source IP Address และ ที่ PMX เท่ากับ 2 บิต 3 บิต และ 6 บิต เป็นขนาด Prefix ที่เหมาะสมสำหรับ Source MAC Address โดยการพิจารณาจากผลคูณจำนวน 4 ผลคูณที่ได้นำเสนอในบทที่ 3 ซึ่งจะนำผลลัพธ์ที่ได้เหล่านี้ไปกำหนดรูปแบบ Prefix และทำการทดลองในการทดลองที่ 4.2.2 ต่อไป

4.2.2 การทดลองแบ่งขนาด Prefix แบบปรับเปลี่ยนได้ โดยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address

วัตถุประสงค์การทดลอง เพื่อหารูปแบบของ Prefix แบบปรับเปลี่ยนได้ (แต่ละ Block อาจจะมีขนาดไม่เท่ากัน) ซึ่งทำให้โปรแกรม BIL มีความเร็วในการค้นหา ความเร็วในการ

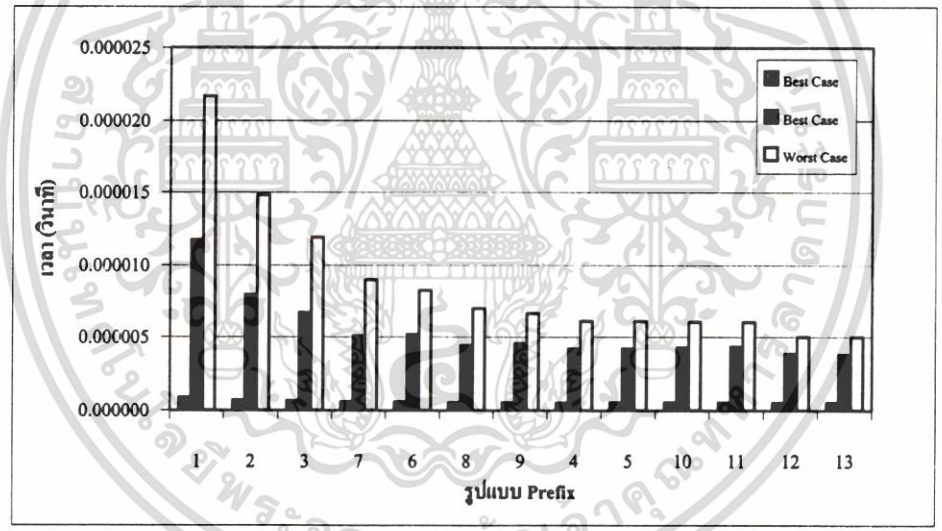
ปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลดีที่สุด สำหรับ Source IP Address (ขนาด 32 บิต) ที่จำนวนกฎ 4,096 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดการแบ่งขนาด Prefix จำนวน 13 รูปแบบ ได้แก่
   - รูปแบบที่ 1 แบ่งขนาด Prefix เป็น 2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต
   - รูปแบบที่ 2 แบ่งขนาด Prefix เป็น 3+3+3+3+3+3+3+3+3+3+3+2 บิต
   - รูปแบบที่ 3 แบ่งขนาด Prefix เป็น 4+4+4+4+4+4+4+4 บิต
   - รูปแบบที่ 4 แบ่งขนาด Prefix เป็น 2+6+10+14 บิต
   - รูปแบบที่ 5 แบ่งขนาด Prefix เป็น 4+8+8+12 บิต
   - รูปแบบที่ 6 แบ่งขนาด Prefix เป็น 2+4+6+8+10+2 บิต
   - รูปแบบที่ 7 แบ่งขนาด Prefix เป็น 6+6+6+6+6+2 บิต
   - รูปแบบที่ 8 แบ่งขนาด Prefix เป็น 7+7+7+7+4 บิต
   - รูปแบบที่ 9 แบ่งขนาด Prefix เป็น 8+8+8+8 บิต
   - รูปแบบที่ 10 แบ่งขนาด Prefix เป็น 9+9+9+5 บิต
   - รูปแบบที่ 11 แบ่งขนาด Prefix เป็น 10+10+10+2 บิต
   - รูปแบบที่ 12 แบ่งขนาด Prefix เป็น 11+11+10 บิต
   - รูปแบบที่ 13 แบ่งขนาด Prefix เป็น 12+12+8 บิต
3. กำหนดจำนวนกฎเท่ากับ 2^{12} = 4,096 ข้อ (Bitvector ขนาด 512 ไบต์)
4. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Best Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13
5. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Average Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13
6. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Worst Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13
7. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต
8. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13
9. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13
10. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case กับ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13

ตารางที่ 4.13 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13

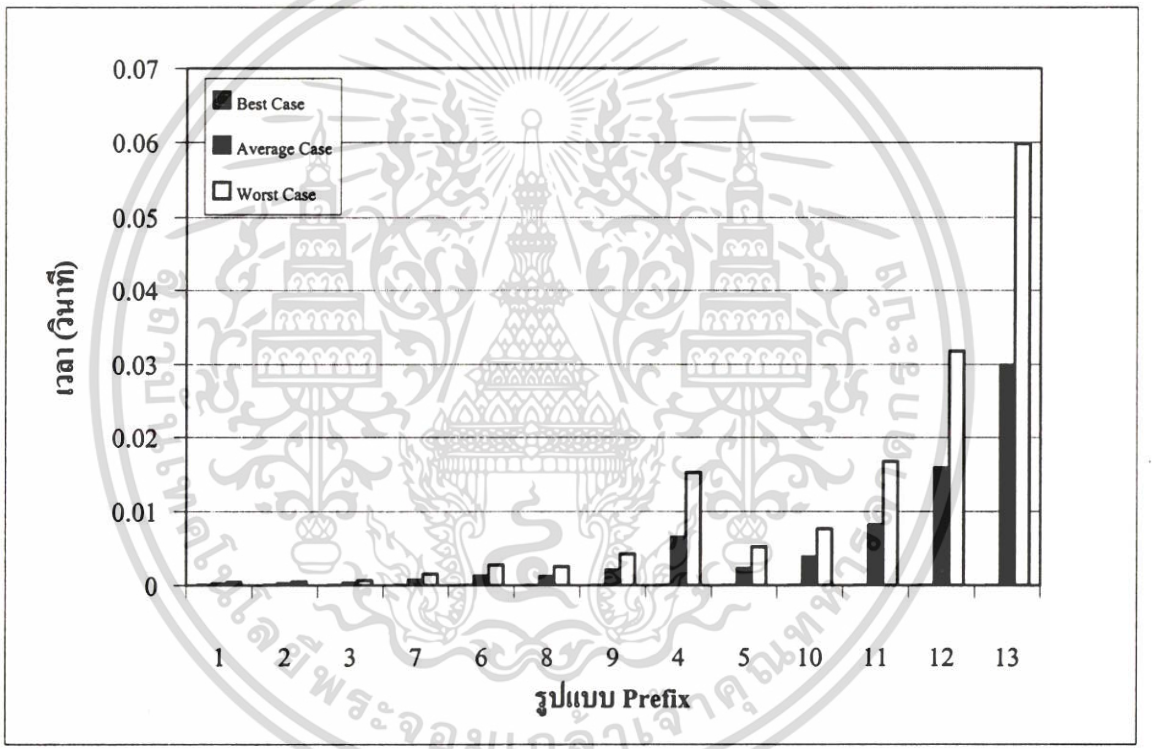
กรณี	รูปแบบ Prefix												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Best Case (วินาที)	0.000000884	0.000000706	0.000000670	0.000000530	0.000000554	0.000000601	0.000000602	0.000000548	0.000000556	0.000000554	0.000000553	0.000000532	0.000000532
Best Case (วินาที)	0.000011771	0.000007947	0.000006681	0.000004235	0.000004241	0.000005184	0.000005095	0.000004444	0.000004576	0.000004304	0.000004364	0.000003902	0.000003842
Worst Case (วินาที)	0.000021682	0.000014804	0.000011928	0.000006082	0.000006081	0.000008238	0.000008980	0.000006963	0.000006624	0.000006035	0.000006023	0.000005014	0.000004998



รูปที่ 4.19 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13 เรียงลำดับตามความเร็วในการค้นหาจากน้อยไปหามากตามกรณี Worst Case

ตารางที่ 4.14 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13

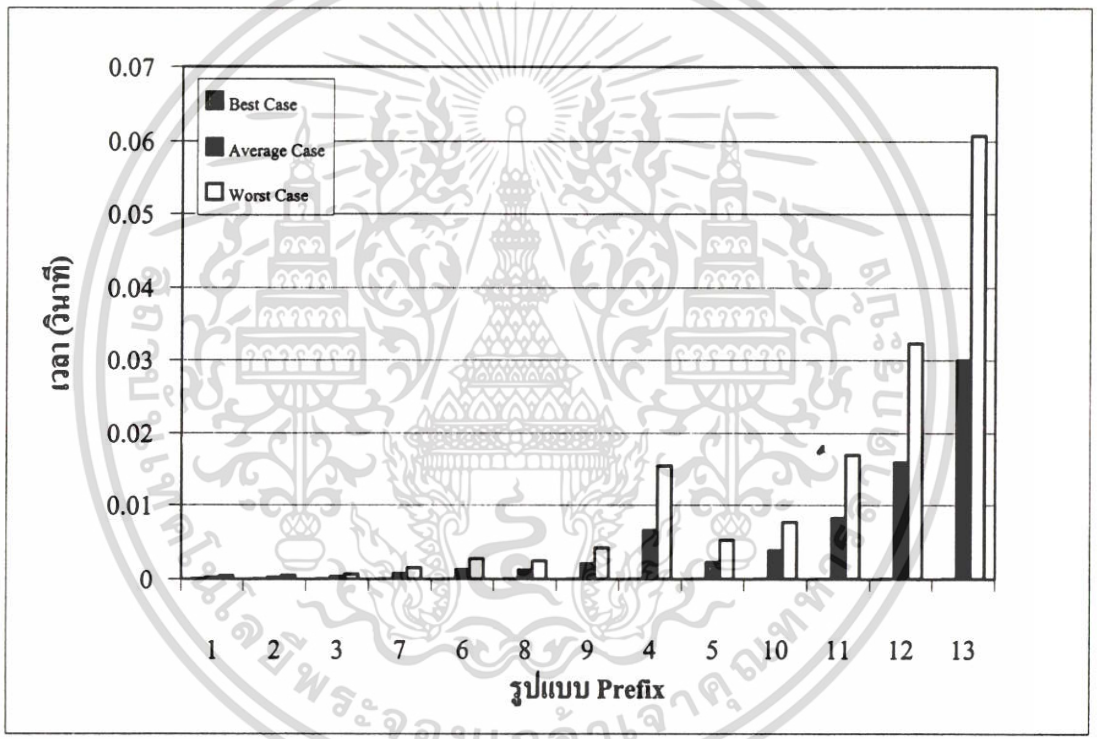
กรณี	รูปแบบ Prefix												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Best Case (วินาที)	0.000020532461538	0.000018391553114	0.000017397841270	0.000016196061050	0.000016141267399	0.000016897147741	0.000016830534799	0.000016200398046	0.000016063882784	0.000016113658120	0.000016218442002	0.000015920373626	0.000015818236874
Best Case (วินาที)	0.000241508930403	0.000272868512821	0.000353498947497	0.006580663387057	0.002316693550672	0.001330447306471	0.000785509125763	0.001258357623932	0.002111631028083	0.003877471924298	0.008242263514042	0.015890491501832	0.029917196952381
Worst Case (วินาที)	0.000464145750916	0.000528357987790	0.000689529413919	0.015271177457876	0.005223326163614	0.002772122647131	0.001565708556777	0.002548089194139	0.004246677975580	0.007670278945055	0.016725706879121	0.031825749689866	0.059826524699110



รูปที่ 4.20 เวลาเฉลี่ยในการเพิ่มกฎต่อข้อ (วินาที) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13 เรียงลำดับตามความเร็วในการค้นหาจากน้อยไปหามากตามกรณี Worst Case

ตารางที่ 4.15 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) ของ Prefix แบบที่ 1 ถึงรูปแบบที่ 13

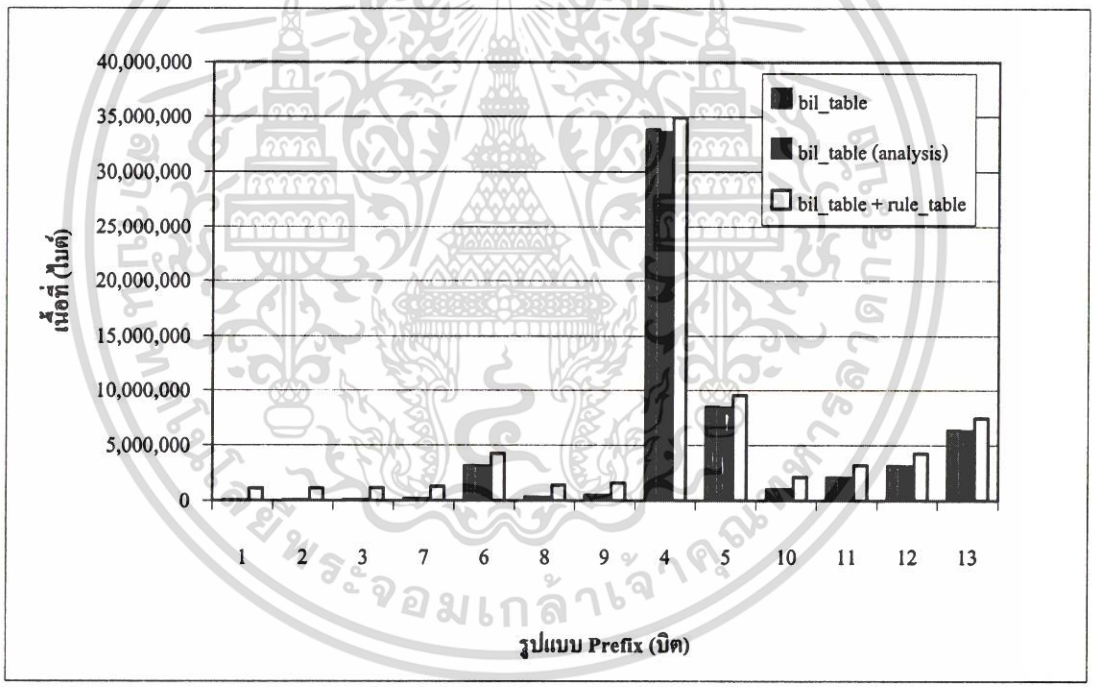
กรณี	รูปแบบ Prefix												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Best Case (วินาที)	0.000022686383394	0.000020799965812	0.000019655726496	0.000018540776557	0.000018532432234	0.000019251689866	0.000019235687424	0.000018784813187	0.000018505802198	0.000018513738706	0.000018620561661	0.000018262273504	0.000018176085470
Best Case (วินาที)	0.000243508459096	0.000274750312576	0.000355711240537	0.006632610913309	0.002327189936508	0.001336226879121	0.000791966630037	0.001255813626374	0.002131259565324	0.003925644803419	0.008299768888889	0.015949218730159	0.030033273181929
Worst Case (วินาที)	0.000461456669109	0.000527092334554	0.000692323255189	0.015436761865690	0.005305973428571	0.002787807560440	0.001575143225885	0.002504708031746	0.004283104227106	0.007705938815629	0.016982747069597	0.032330055789988	0.060778803558346



รูปที่ 4.21 เวลาเฉลี่ยในการลบกฎต่อข้อ (วินาที) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13 เรียงลำดับตามความเร็วในการค้นหาจากน้อยไปหามากตามกรณี Worst Case

ตารางที่ 4.16 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13

Data Structure	รูปแบบ Prefix												
	1	2	3	4	5	6	7	8	9	10	11	12	13
bil_table (ไบต์)	33,024	45,408	66,048	33,816,576	8,454,144	3,170,304	198,144	330,240	528,384	1,056,768	2,113,536	3,170,304	6,340,608
rule_table (ไบต์)	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728	1,097,728
bil_table analysis (ไบต์)	32,768	45,056	65,536	33,554,432	8,388,608	3,145,728	196,608	327,680	524,288	1,048,576	2,097,152	3,145,728	6,291,456
bil_table + rule_table (ไบต์)	1,130,752	1,143,136	1,163,776	34,914,304	9,551,872	4,268,032	1,295,872	1,427,968	1,626,112	2,154,496	3,211,264	4,268,032	7,438,336



รูปที่ 4.22 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 1 ถึงรูปแบบที่ 13 เรียงลำดับตามความเร็วในการค้นหาจากน้อยไปหามากตามกรณี Worst Case

วิเคราะห์ผลการทดลอง

1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) และกำหนดการแบ่งขนาด Prefix จำนวน 13 รูปแบบ จากผลการทดลองในตารางที่ 4.13 พบว่าในกรณี Worst Case รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) จะใช้เวลาในการค้นหามากที่สุด และรูปแบบที่ 13 (12+12+8 บิต) จะใช้เวลาในการค้นหาน้อยที่สุด โดยจะแสดงเวลาในการค้นหาต่อแพ็กเก็ตเรียงลำดับจากมากไปน้อยได้แก่ รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 3 รูปแบบที่ 7 รูปแบบที่ 6 รูปแบบที่ 8 รูปแบบที่ 9 รูปแบบที่ 4 รูปแบบที่ 5 รูปแบบที่ 10 รูปแบบที่ 11 รูปแบบที่ 12 รูปแบบที่ 13 ตามลำดับ โดยสามารถจัดกลุ่มรูปแบบ Prefix ตามจำนวนของ Prefix เป็น 7 กลุ่ม และเรียงลำดับเวลาในการค้นหาจากน้อยไปมาก ดังตารางที่ 4.17

ตารางที่ 4.17 จำนวน Prefix และรูปแบบ Prefix สำหรับ Source IP Address เรียงลำดับตามเวลาในการค้นหาจากน้อยไปมาก

ลำดับที่	จำนวน Prefix (Block)	รูปแบบ Prefix
1	3	13,12
2	4	11,10,5,4,9
3	5	8
4	6	6,7
5	8	3
6	11	2
7	16	1

จากการสังเกตพบว่ารูปแบบ Prefix ในกลุ่มเดียวกันจะมีความเร็วในการค้นหาในทุกกรณี และความเร็วในการปรับปรุงกฎในกรณี Best Case ที่ใกล้เคียงกัน ในขณะที่มีความเร็วในการปรับปรุงกฎในกรณี Average Case และ Worst Case แตกต่างกันอย่างเห็นได้ชัด อันเนื่องมาจากขนาดของ Prefix เป็นสำคัญ

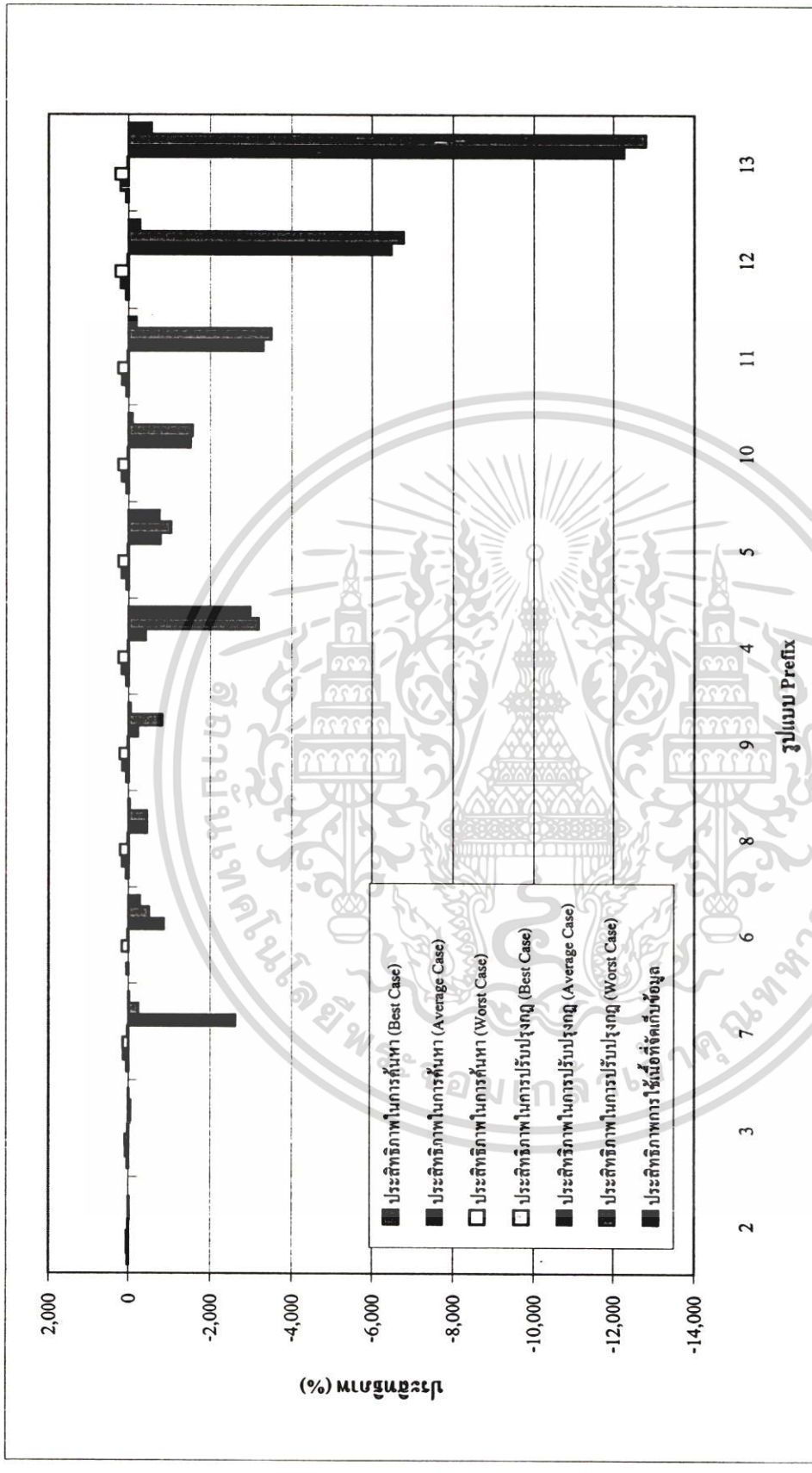
2. จากผลการทดลองในตารางที่ 4.14 และ 4.15 พบว่าในกรณี Best Case รูปแบบที่ 13 (12+12+8 บิต) จะใช้เวลาในการปรับปรุงกฎน้อยที่สุด และรูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) จะใช้เวลาในการปรับปรุงกฎมากที่สุด ในขณะที่ในกรณี Average Case และ Worst Case ที่ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) จะใช้เวลาในการปรับปรุงกฎน้อยที่สุด และรูปแบบที่ 13 (12+12+8 บิต) จะใช้เวลาในการปรับปรุงกฎมากที่สุด

3. จากผลการทดลองในตารางที่ 4.16 พบว่ารูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) จะใช้เนื้อที่จัดเก็บข้อมูลต่ำที่สุด และรูปแบบที่ 4 (2+6+10+14 บิต) โปรแกรมจะใช้เนื้อที่จัดเก็บข้อมูลสูงที่สุด

4. จากผลการทดลองในตารางที่ 4.13-4.16 สามารถแสดงประสิทธิภาพของโปรแกรมที่ Prefix รูปแบบต่างๆ เปรียบเทียบกับรูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) ดังตารางที่ 4.18 และ รูปที่ 4.23

ตารางที่ 4.18 ประสิทธิภาพของโปรแกรมที่ Prefix รูปแบบที่ 2 รูปแบบที่ 3 รูปแบบที่ 7 รูปแบบที่ 6 รูปแบบที่ 8 รูปแบบที่ 9 รูปแบบที่ 4 รูปแบบที่ 5 รูปแบบที่ 10 รูปแบบที่ 11 รูปแบบที่ 12 และรูปแบบที่ 13 เปรียบเทียบกับรูปแบบที่ 1

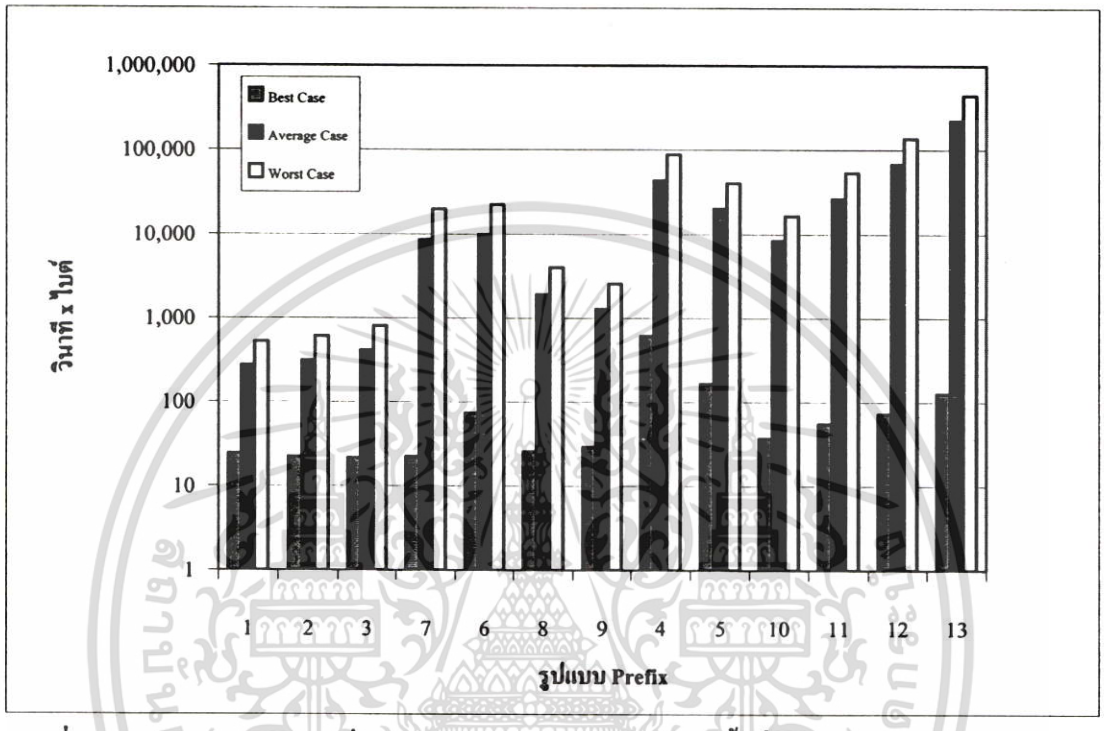
ประสิทธิภาพเปรียบเทียบกับแบบที่ 1	รูปแบบ Prefix											
	2	3	7	6	8	9	4	5	10	11	12	13
เวลาในการค้นหา (Best Case)	มากกว่า 25.21 %	มากกว่า 31.94 %	มากกว่า 46.84 %	มากกว่า 47.09 %	มากกว่า 61.31 %	มากกว่า 58.99 %	มากกว่า 66.79 %	มากกว่า 59.57 %	มากกว่า 59.57 %	มากกว่า 59.86 %	มากกว่า 66.17 %	มากกว่า 66.17 %
เวลาในการค้นหา (Average Case)	มากกว่า 48.12 %	มากกว่า 76.19 %	มากกว่า 131.03 %	มากกว่า 127.06 %	มากกว่า 164.87 %	มากกว่า 157.23 %	มากกว่า 177.95 %	มากกว่า 177.55 %	มากกว่า 173.49 %	มากกว่า 169.73 %	มากกว่า 201.67 %	มากกว่า 206.38 %
เวลาในการค้นหา (Worst Case)	มากกว่า 46.46 %	มากกว่า 81.77 %	มากกว่า 141.45 %	มากกว่า 163.19 %	มากกว่า 211.39 %	มากกว่า 227.32 %	มากกว่า 256.49 %	มากกว่า 256.55 %	มากกว่า 259.27 %	มากกว่า 259.99 %	มากกว่า 332.43 %	มากกว่า 333.81 %
เวลาในการปรับปรุงกฎ (Best Case)	มากกว่า 10.28 %	มากกว่า 16.64 %	มากกว่า 24.42 %	มากกว่า 24.64 %	มากกว่า 19.56 %	มากกว่า 19.83 %	มากกว่า 23.53 %	มากกว่า 25.02 %	มากกว่า 24.81 %	มากกว่า 24.05 %	มากกว่า 26.44 %	มากกว่า 27.14 %
เวลาในการปรับปรุงกฎ (Average Case)	น้อยกว่า 12.91 %	น้อยกว่า 46.22 %	น้อยกว่า 2,624.29 %	น้อยกว่า 857.47 %	น้อยกว่า 449.81 %	น้อยกว่า 225.24 %	น้อยกว่า 418.37 %	น้อยกว่า 774.79 %	น้อยกว่า 1,508.83 %	น้อยกว่า 3,310.61 %	น้อยกว่า 6,464.65 %	น้อยกว่า 12,260.48 %
เวลาในการปรับปรุงกฎ (Worst Case)	น้อยกว่า 13.83 %	น้อยกว่า 48.56 %	น้อยกว่า 237.33 %	น้อยกว่า 497.25 %	น้อยกว่า 448.98 %	น้อยกว่า 814.94 %	น้อยกว่า 3,190.17 %	น้อยกว่า 1,025.36 %	น้อยกว่า 1,552.56 %	น้อยกว่า 3,503.55 %	น้อยกว่า 6,756.84 %	น้อยกว่า 12,789.60 %
เนื้อที่จัดเก็บข้อมูล	น้อยกว่า 1.10 %	น้อยกว่า 2.92 %	น้อยกว่า 14.60 %	น้อยกว่า 277.45 %	น้อยกว่า 26.28 %	น้อยกว่า 43.81 %	น้อยกว่า 2,987.71 %	น้อยกว่า 744.74 %	น้อยกว่า 90.54 %	น้อยกว่า 183.99 %	น้อยกว่า 277.45 %	น้อยกว่า 557.82 %



รูปที่ 4.23 ประสิทธิภาพของโปรแกรมที่ Prefix รูปแบบที่ 2 รูปแบบที่ 3 รูปแบบที่ 7 รูปแบบที่ 6 รูปแบบที่ 8 รูปแบบที่ 9 รูปแบบที่ 4 รูปแบบที่ 5 รูปแบบที่ 10 รูปแบบที่ 11 รูปแบบที่ 12 และรูปแบบที่ 13 เปรียบเทียบกับรูปแบบที่ 1

5. เพื่อหารูปแบบ Prefix ที่เหมาะสมผู้วิจัยจะทำการศึกษาประสิทธิภาพในการทำงานของโปรแกรม โดยพิจารณาจากผลคูณของประสิทธิภาพการทำงานจำนวน 4 ผลคูณ ตามที่เคยกล่าวในบทก่อนหน้า

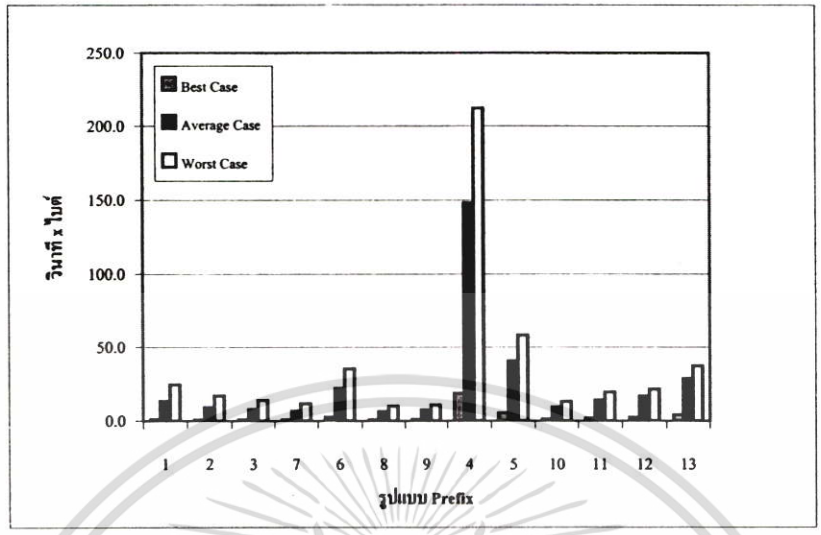
1. ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



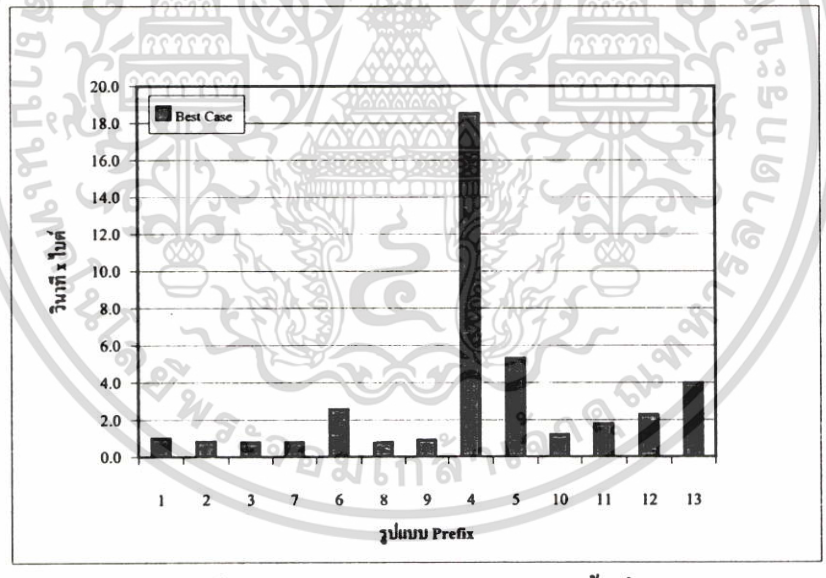
รูปที่ 4.24 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address โดยใช้มาตราส่วนลอการิทึม

จากกราฟในรูปที่ 4.24 พบว่าในกรณี Best Case ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 3 (4+4+4+4+4+4+4+4 บิต) ในขณะที่ในกรณี Average Case และ Worst Case ผลคูณของเวลาในการเพิ่มกฎและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

2. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล



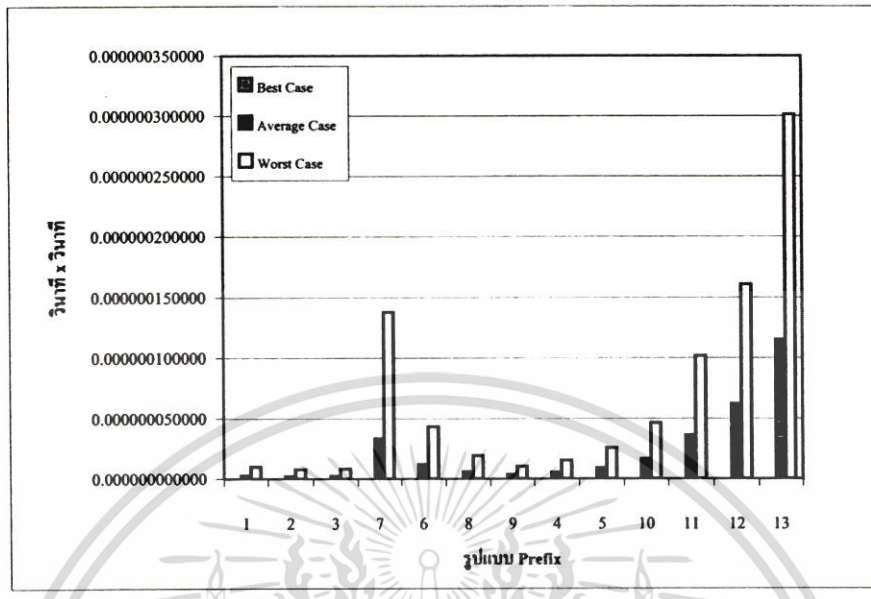
รูปที่ 4.25 ก ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address



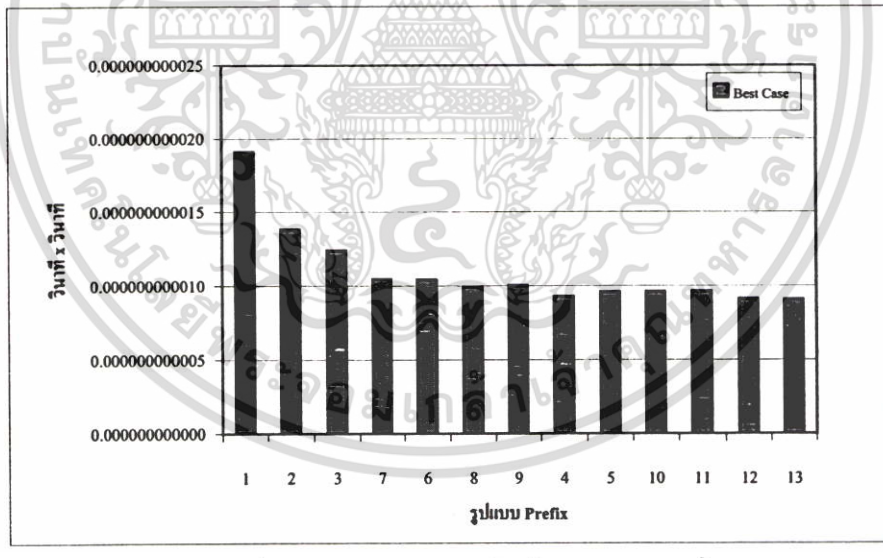
รูปที่ 4.25 ข ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเนื้อที่จัดเก็บข้อมูล ในกรณี Best Case โดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address

จากกราฟในรูปที่ 4.25 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 3 (4+4+4+4+4+4+4+4 บิต) ในขณะที่ในกรณี Average Case และ Worst Case ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 8 (7+7+7+7+4 บิต)

3. ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ



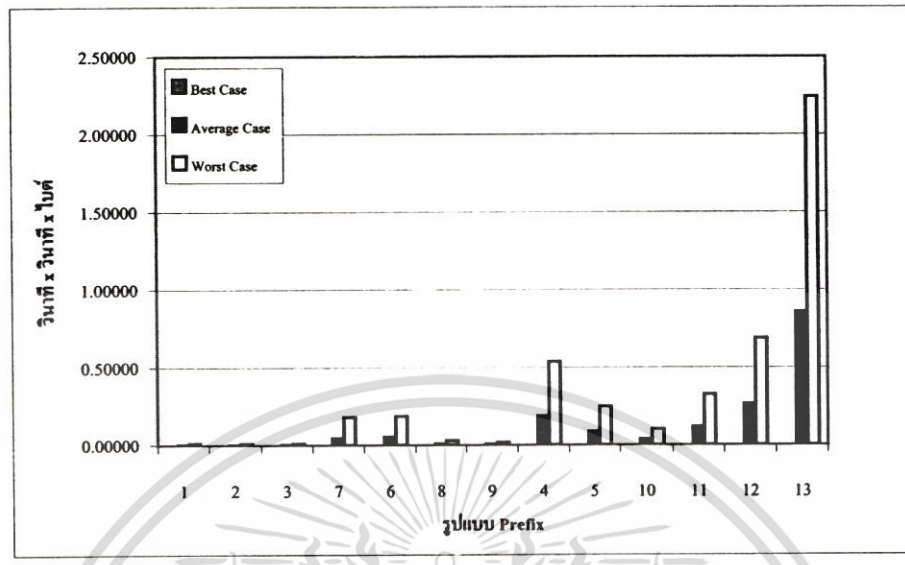
รูปที่ 4.26 ก ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ โดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address



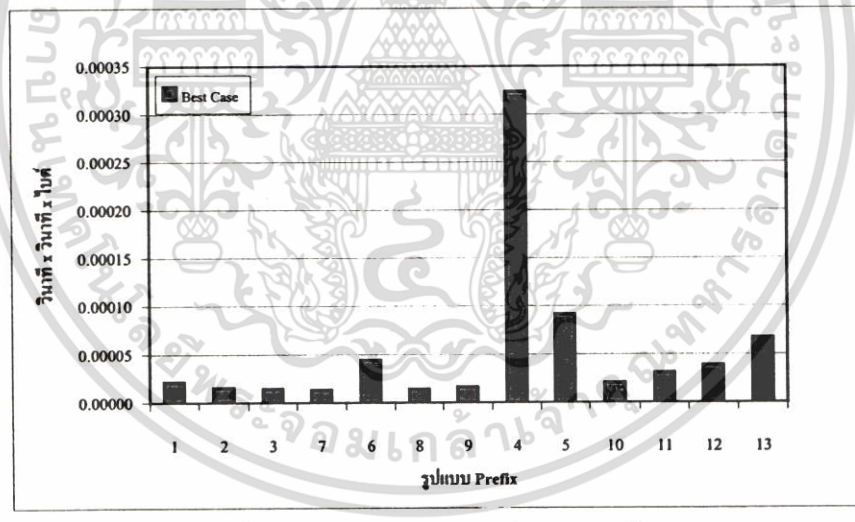
รูปที่ 4.26 ข ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต และเวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ ในกรณี Best Case โดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address

จากกราฟในรูปที่ 4.26 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎจะมีค่าต่ำสุดที่รูปแบบที่ 13 (12+12+8 บิต) ในขณะที่ในกรณี Average Case และ Worst Case ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎจะมีค่าต่ำสุดที่รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

4. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



รูปที่ 4.27 ก ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูลโดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address



รูปที่ 4.27 ข ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎต่อข้อ และเนื้อที่จัดเก็บข้อมูล ในกรณี Best Case โดยแบ่ง Prefix แบบปรับเปลี่ยนได้ สำหรับ Sourec IP Address

จากกราฟในรูปที่ 4.27 พบว่าในกรณี Best Case ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 7 (6+6+6+6+6+2 บิต) ในขณะที่ในกรณี Average Case และ Worst Case ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลจะมีค่าต่ำสุดที่รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

ตารางที่ 4.19 รูปแบบ Prefix ที่เหมาะสมสำหรับ Source IP Address โดยพิจารณาจากจุดต่ำสุดของผลคูณ

ผลคูณ	กรณี		
	Best Case	Average Case	Worst Case
1. เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	รูปแบบที่ 3	รูปแบบที่ 1	รูปแบบที่ 1
2. เวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล	รูปแบบที่ 3	รูปแบบที่ 8	รูปแบบที่ 8
3. เวลาในการค้นหาและเวลาในการปรับปรุงกฎ	รูปแบบที่ 13	รูปแบบที่ 2	รูปแบบที่ 2
4. เวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	รูปแบบที่ 7	รูปแบบที่ 2	รูปแบบที่ 2

จากตารางที่ 4.19 พบว่าในกรณี Worst Case รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+3+2 บิต) และรูปแบบที่ 8 (7+7+7+7+4 บิต) เป็นรูปแบบ Prefix ที่เหมาะสมสำหรับ Source IP Address โดยพิจารณาจากผลคูณจำนวน 4 ผลคูณที่ได้นำเสนอในบทที่ 3 ซึ่งผู้วิจัยจะนำผลลัพธ์ที่ได้เหล่านี้ไปทำการทดลองในการทดลองที่ 4.2.3 ต่อไป นอกจากนี้ยังพบว่าการแบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน หรือใกล้เคียงกันจะทำให้โปรแกรมมีประสิทธิภาพในการทำงานดีกว่าการแบ่ง Prefix เป็น Block ที่มีขนาดแตกต่างกัน

4.2.3 การทดลองวัดประสิทธิภาพการทำงานของโปรแกรม BIL กับฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

วัตถุประสงค์การทดลอง เพื่อเปรียบเทียบประสิทธิภาพในแต่ละด้านสำหรับ Prefix ในรูปแบบต่างๆ โดยใช้ฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดการแบ่งขนาด Prefix จำนวน 4 รูปแบบจาก 10 รูปแบบ ในการทดลองที่ 4.2.2 ได้แก่
   - รูปแบบที่ 1 แบ่งขนาด Prefix เป็น 2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต
   - รูปแบบที่ 2 แบ่งขนาด Prefix เป็น 3+3+3+3+3+3+3+3+3+3+3+2 บิต
   - รูปแบบที่ 8 แบ่งขนาด Prefix เป็น 7+7+7+7+4 บิต
   - รูปแบบที่ 9 แบ่งขนาด Prefix เป็น 8+8+8+8 บิต
3. กำหนดจำนวนกฎเท่ากับ 1,000, 2,000, 3,000, ..., 10,000 ข้อ
4. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Best Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9
5. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Average Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

6. สร้างฐานข้อมูลกฎสำหรับวัดเวลาในการปรับปรุงกฎในกรณี Worst Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

7. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต

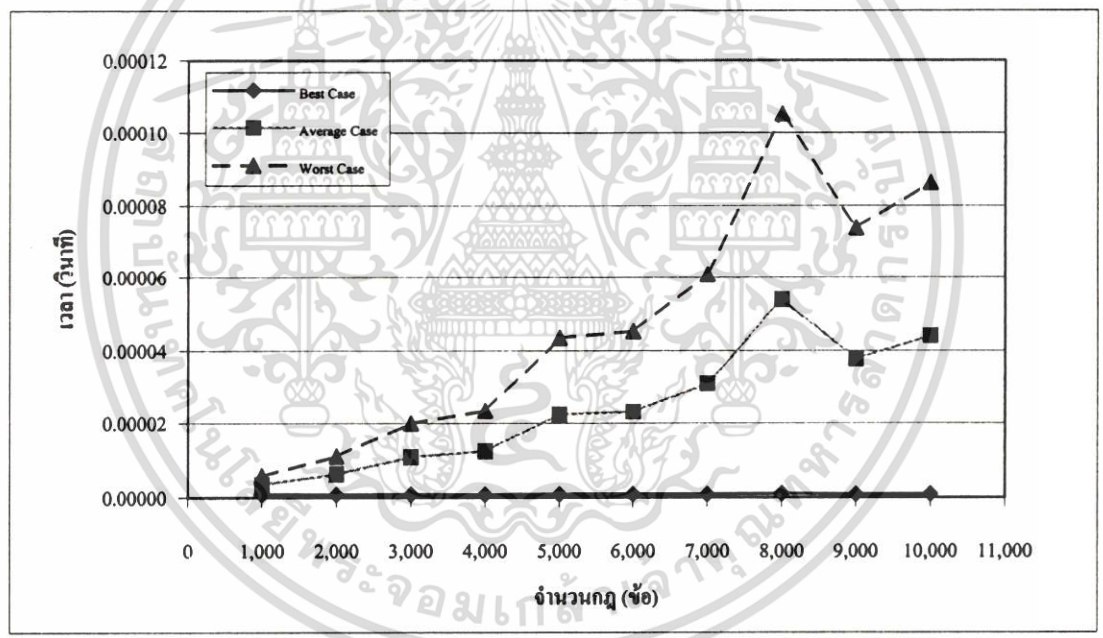
8. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

9. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

10. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case กับ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

ตารางที่ 4.20 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

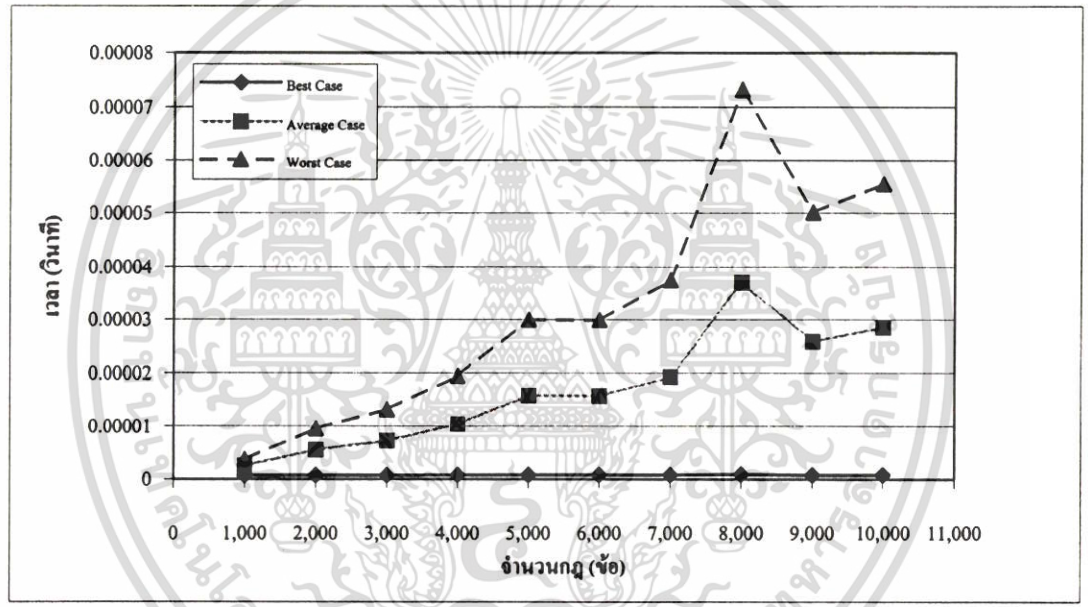
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000000908437300	0.000000868438900	0.000000934728200	0.000000884875100	0.000000967921800	0.000000959066100	0.000000983974700	0.000001150647200	0.000000973379000	0.000000970199800
Average Case (วินาที)	0.000003546333900	0.000006312813500	0.000010926054700	0.000012427085300	0.000022443510700	0.000023153536800	0.000031004483500	0.000053851597500	0.000037739164300	0.000044008891600
Worst Case (วินาที)	0.000005896787500	0.000011123832800	0.000020024048000	0.000023467098500	0.000043494560100	0.000045224967400	0.000060812931500	0.000105309952000	0.000073964566100	0.000086260740500



รูปที่ 4.28 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

ตารางที่ 4.21 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

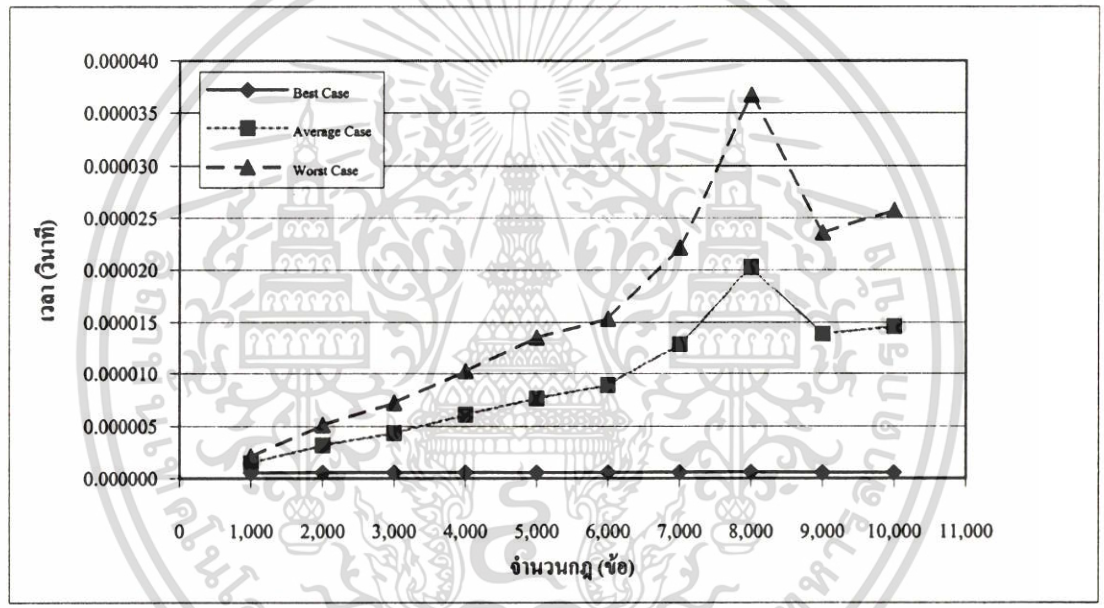
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000000702837000	0.000000736930800	0.000000732350800	0.000000746400300	0.000000773743400	0.000000770883300	0.000000786747300	0.000000891121300	0.000000768673200	0.000000784432800
Average Case (วินาที)	0.000002481870100	0.000005476491300	0.000007229665100	0.000010360857900	0.000015747821500	0.000015663588000	0.000019226494100	0.000037066874300	0.000025831555800	0.000028488583400
Worst Case (วินาที)	0.000003667296300	0.000009534942500	0.000013151176900	0.000019447490900	0.000029951932600	0.000029886736900	0.000037555438100	0.000073196596500	0.000050130497700	0.000055459043700



รูปที่ 4.29 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

ตารางที่ 4.22 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4บิต)

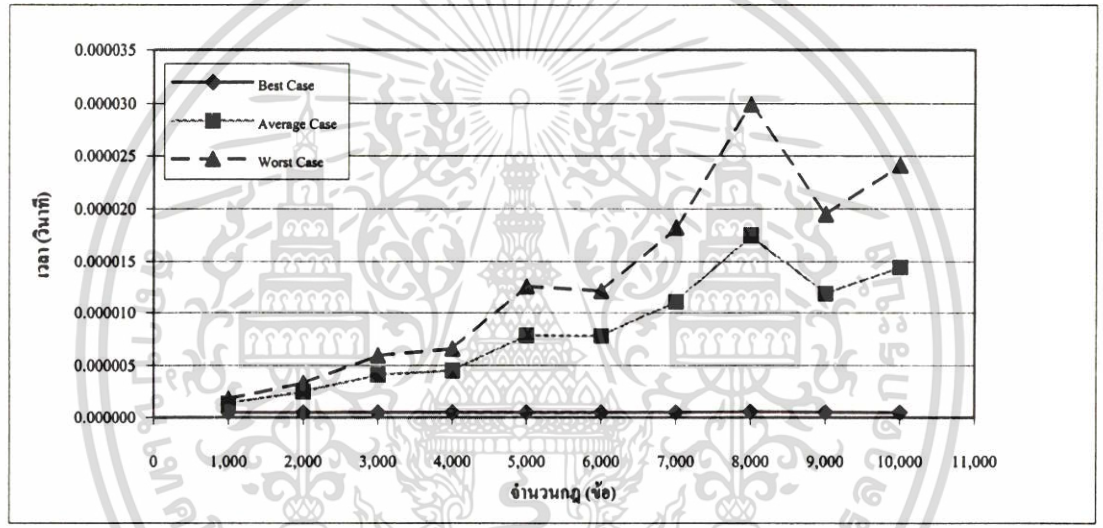
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000000555011100	0.000000583579800	0.000000586031800	0.000000581886000	0.000000574043900	0.000000579540600	0.000000593608800	0.000000636919100	0.000000577519600	0.000000572088500
Average Case (วินาที)	0.000001513003200	0.000003195053300	0.000004349892900	0.000006069322600	0.000007608033900	0.000008897943800	0.000012889123700	0.000020251561300	0.000013890813500	0.000014597138800
Worst Case (วินาที)	0.000002126940500	0.000005109050500	0.000007194264200	0.000010235270200	0.000013535858400	0.000015350727200	0.000022128944300	0.000036729284000	0.000023585236800	0.000025765014700



รูปที่ 4.30 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4บิต)

ตารางที่ 4.23 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

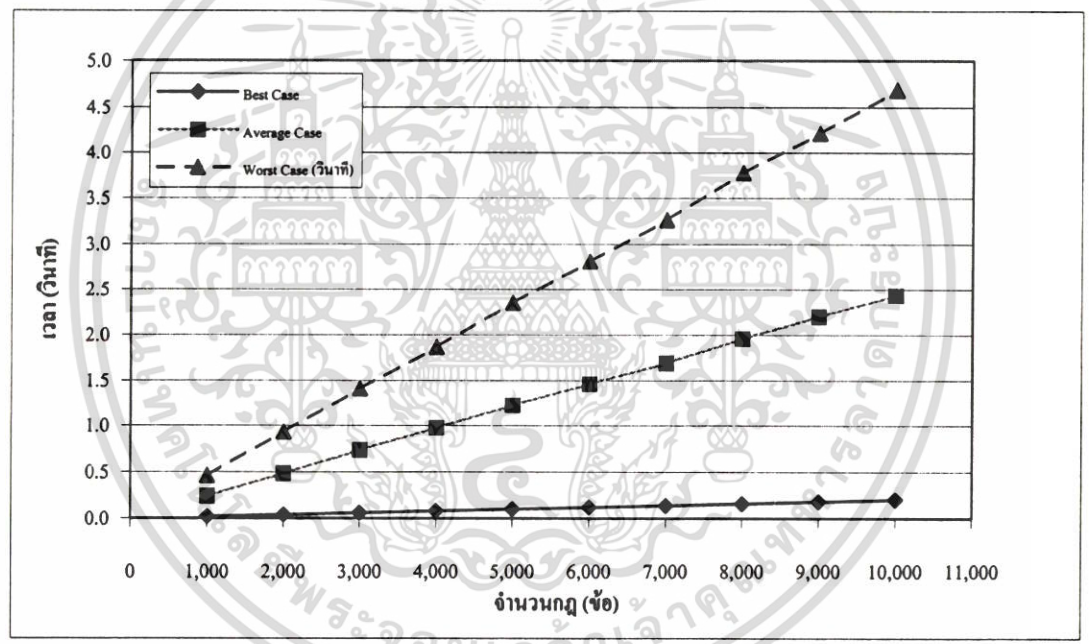
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000000495296900	0.000000493740800	0.000000510685100	0.000000498820300	0.000000510790000	0.000000511846500	0.000000514933900	0.000000568141700	0.000000514578000	0.000000510410100
Average Case (วินาที)	0.000001402734600	0.000002498186000	0.000004117437800	0.000004523089800	0.000007843125900	0.000007780956900	0.000011077597200	0.000017497247600	0.000011848972600	0.000014430385700
Worst Case (วินาที)	0.000001817826200	0.000003337243500	0.000005943245000	0.000006574957500	0.000012579565500	0.000012123821600	0.000018258340800	0.000029925070700	0.000019501263100	0.000024115579600



รูปที่ 4.31 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

ตารางที่ 4.24 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

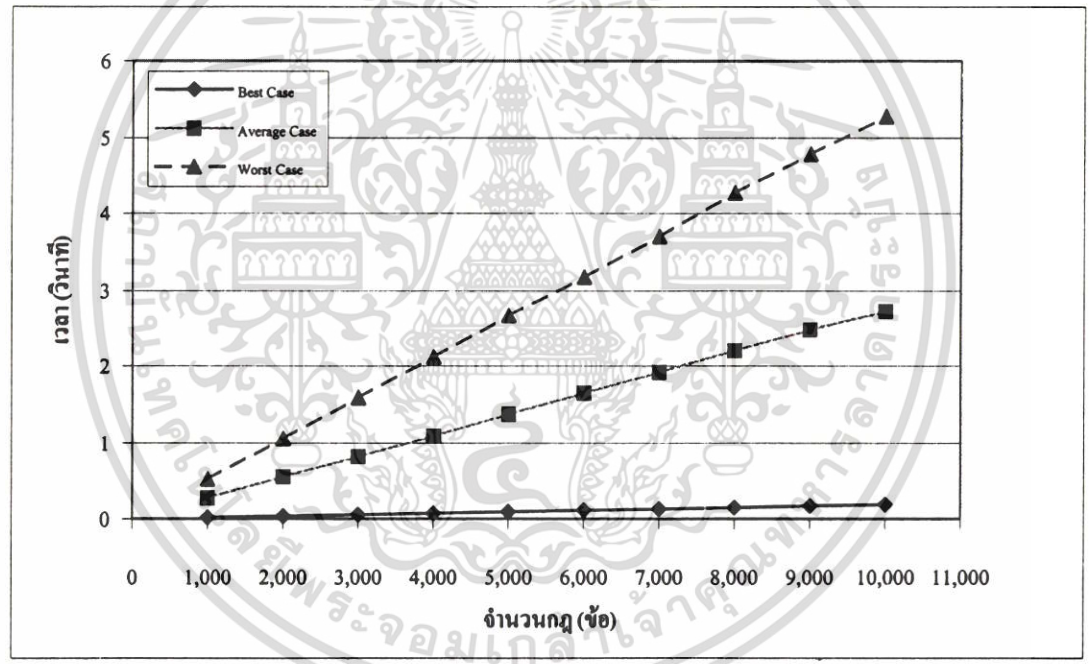
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.02054999	0.0411706	0.06083672	0.08156862	0.10161201	0.1208476	0.13998209	0.16163308	0.18169461	0.20315023
Average Case (วินาที)	0.23951801	0.4820435	0.73038161	0.97180217	1.22245801	1.4617978	1.69234845	1.95881206	2.19813785	2.43321977
Worst Case (วินาที)	0.46006972	0.9301634	1.41094971	1.86763506	2.35029462	2.81094261	3.26757927	3.77640584	4.21531431	4.69061428



รูปที่ 4.32 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

ตารางที่ 4.25 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

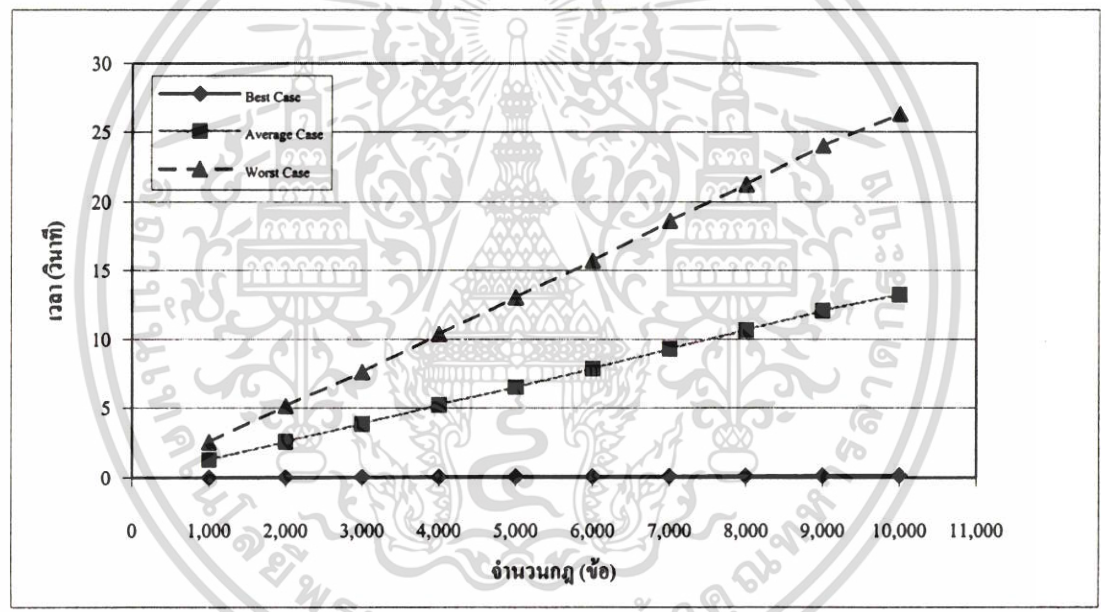
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01863881	0.03680328	0.0550382	0.07362969	0.09219449	0.10981332	0.12742779	0.14599372	0.16719462	0.18432149
Average Case (วินาที)	0.2683127	0.54902696	0.81934816	1.09426954	1.37502906	1.64532279	1.91410884	2.20039825	2.4776695	2.72437878
Worst Case (วินาที)	0.51968104	1.06092947	1.59082686	2.11566673	2.67048639	3.17954119	3.70256929	4.2724447	4.7882148	5.2792106



รูปที่ 4.33 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

ตารางที่ 4.26 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

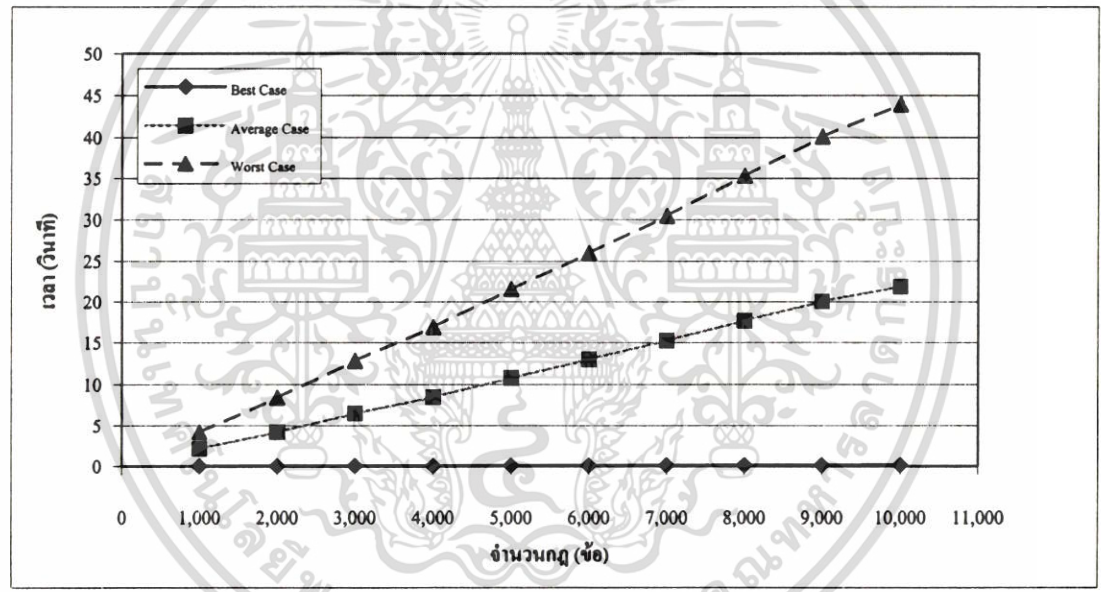
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.0163778	0.03281976	0.0482946	0.06474425	0.08090921	0.09716805	0.11416817	0.12956269	0.14647907	0.16270239
Average Case (วินาที)	1.27618369	2.57319468	3.8364119	5.20304541	6.49950438	7.88037102	9.32550324	10.6606921	12.0546653	13.1987355
Worst Case (วินาที)	2.5400075	5.13846383	7.63614587	10.3705423	13.0034952	15.720685	18.6683832	21.2231717	23.9783667	26.3305477



รูปที่ 4.34 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

ตารางที่ 4.27 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

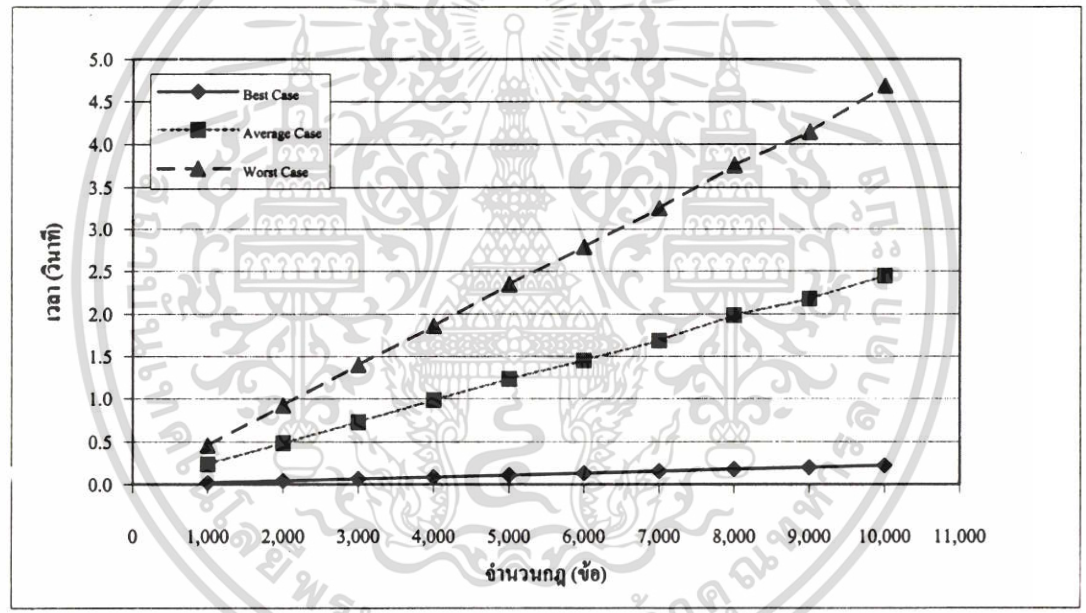
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01615715	0.03216772	0.04833122	0.06401482	0.08017437	0.09566002	0.11137505	0.129892	0.14399003	0.15948871
Average Case (วินาที)	2.12337707	4.18001028	6.48688211	8.49473512	10.7817607	12.9873302	15.2353907	17.6569753	20.0745386	21.888649
Worst Case (วินาที)	4.1818102	8.43977366	12.8296252	16.8866677	21.5852947	26.0059122	30.447958	35.3295042	40.1042642	43.9433329



รูปที่ 4.35 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

ตารางที่ 4.28 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

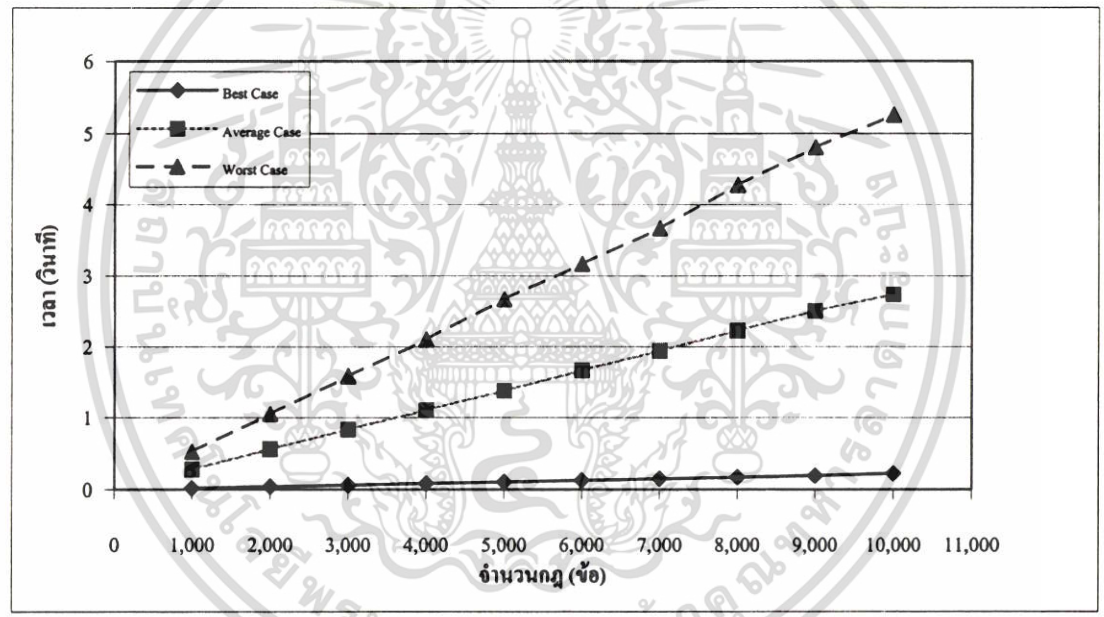
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.02275616	0.04552411	0.06878545	0.09106937	0.11367743	0.1362526	0.158233	0.18337504	0.20339128	0.22586284
Average Case (วินาที)	0.24134278	0.48701612	0.72748042	0.98238004	1.23083496	1.45117483	1.68809659	1.98610062	2.17708727	2.44541625
Worst Case (วินาที)	0.45929894	0.9214927	1.39891194	1.86288304	2.34655206	2.78556248	3.25094068	3.76120124	4.14306563	4.68545137



รูปที่ 4.36 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

ตารางที่ 4.29 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

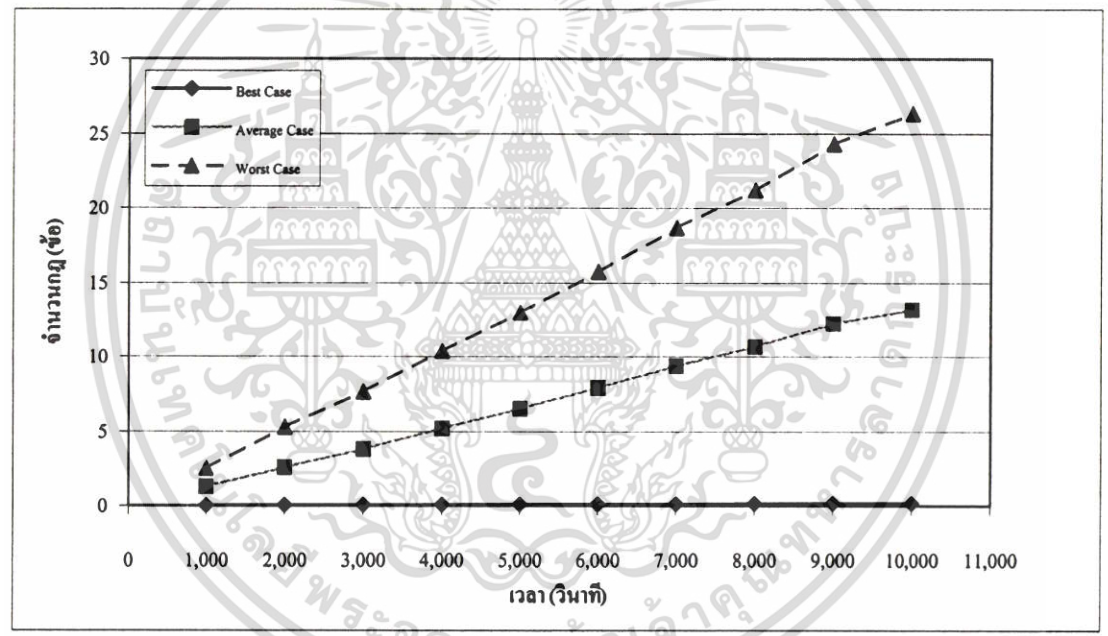
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.02089837	0.04219545	0.06302386	0.08499619	0.10466213	0.12678062	0.14759395	0.16676522	0.19054605	0.21874017
Average Case (วินาที)	0.27265933	0.55701903	0.83137788	1.10470956	1.38105702	1.668752	1.94341323	2.2194247	2.49844129	2.73329057
Worst Case (วินาที)	0.52064089	1.05255486	1.59009681	2.10157903	2.66602289	3.17053947	3.67402274	4.27402634	4.80647279	5.26502045



รูปที่ 4.37 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

ตารางที่ 4.30 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

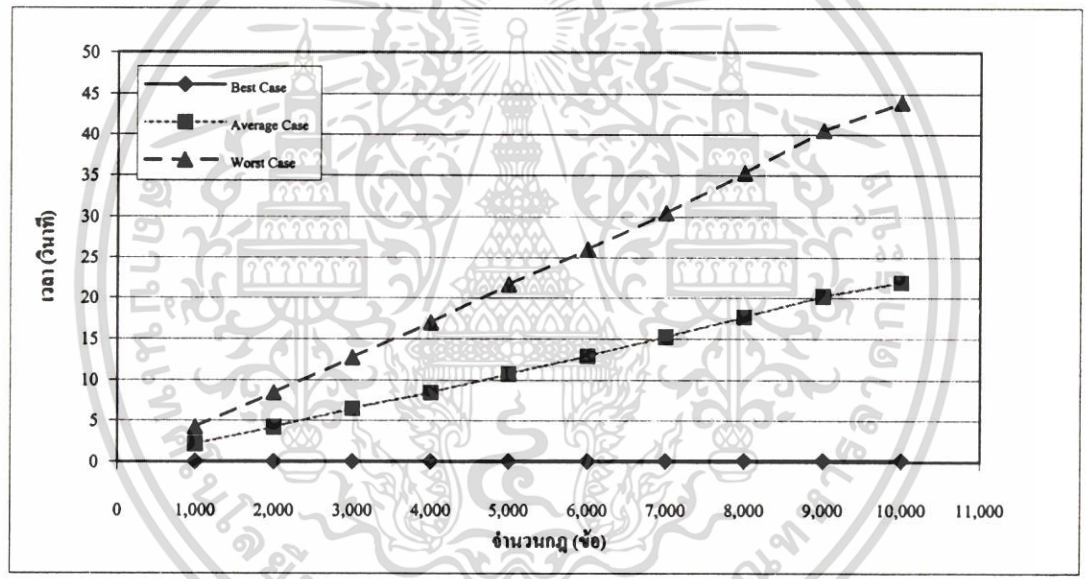
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01886398	0.03809366	0.0569725	0.07670057	0.09389416	0.11246063	0.13329525	0.1487154	0.17064357	0.18777029
Average Case (วินาที)	1.27893106	2.59886827	3.84287066	5.21256014	6.512212	7.89293398	9.35767183	10.6840394	12.2596356	13.2179888
Worst Case (วินาที)	2.54287028	5.30959548	7.6369264	10.3787658	13.0215241	15.75825	18.6772931	21.2327598	24.3391743	26.3419495



รูปที่ 4.38 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

ตารางที่ 4.31 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

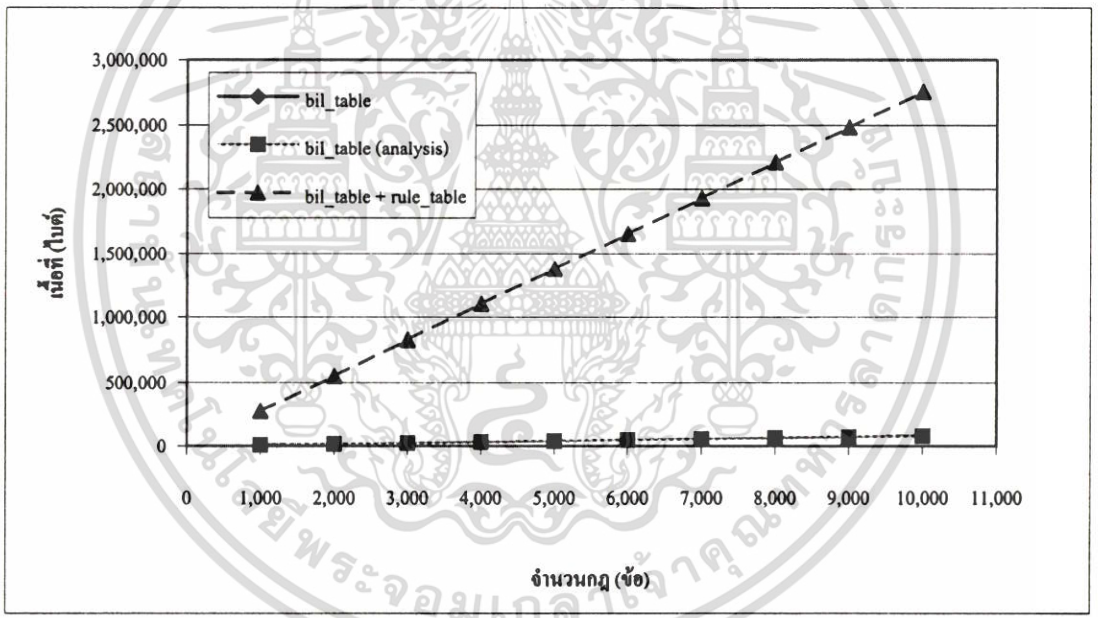
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01858516	0.03784375	0.05595893	0.0744499	0.09223149	0.11107135	0.12796161	0.15015855	0.16568282	0.18446746
Average Case (วินาที)	2.12301113	4.17818671	6.48240025	8.45355502	10.761849	12.9530557	15.2477228	17.6419291	20.195339	21.8682144
Worst Case (วินาที)	4.18348598	8.44792598	12.7749202	16.9788547	21.6127823	26.037913	30.4742588	35.2910205	40.523237	43.985987



รูปที่ 4.39 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

ตารางที่ 4.32 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

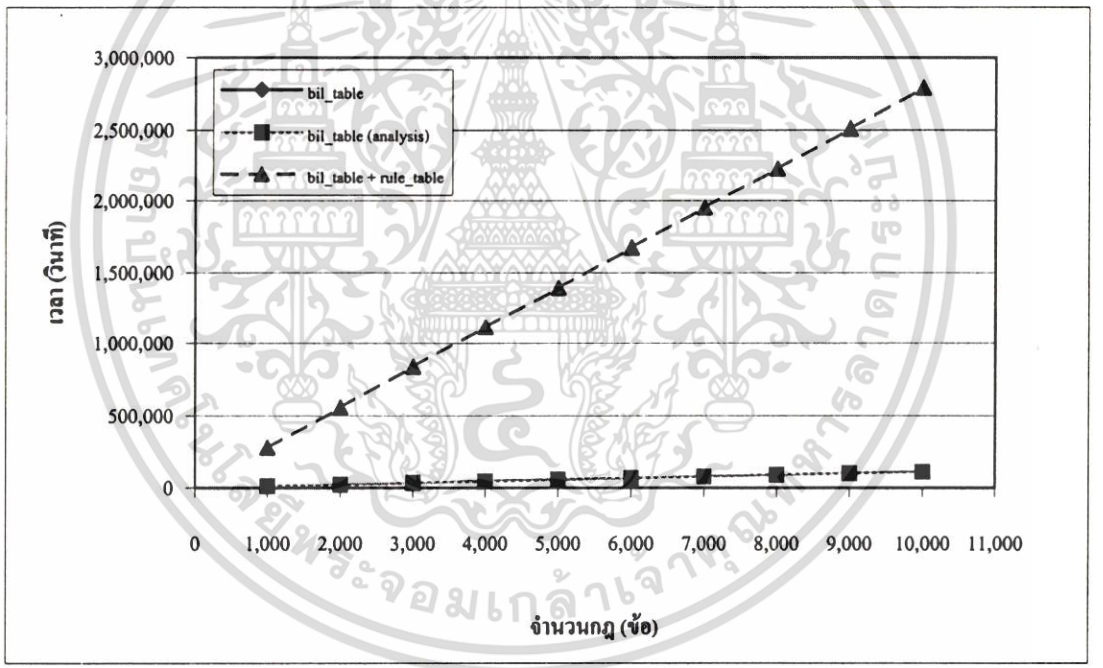
Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
bil_table (ไบต์)	8,192	16,128	24,064	32,256	40,192	48,128	56,064	64,256	72,192	80,128
rule_table (ไบต์)	268,000	536,000	804,000	1,072,000	1,340,000	1,608,000	1,876,000	2,144,000	2,412,000	2,680,000
bil_table analysis (ไบต์)	8,000	16,000	24,000	32,000	40,000	48,000	56,000	64,000	72,000	80,000
bil_table + rule_table (ไบต์)	276,192	552,128	828,064	1,104,256	1,380,192	1,656,128	1,932,064	2,208,256	2,484,192	2,760,128



รูปที่ 4.40 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต)

ตารางที่ 4.33 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

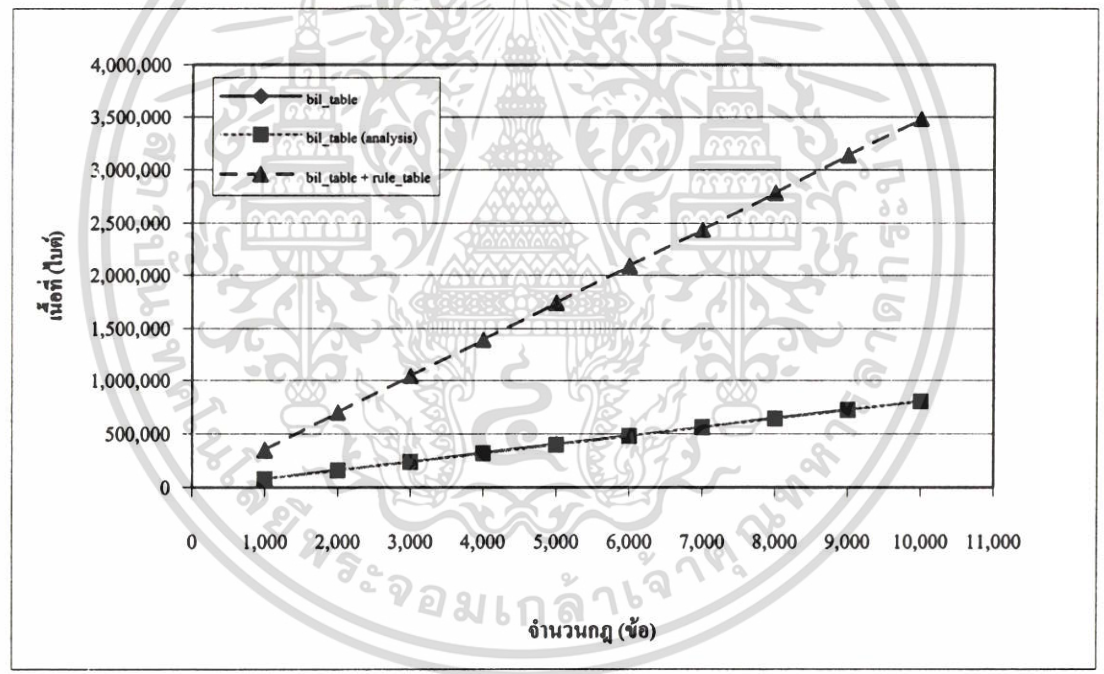
Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
bil_table (ไบต์)	11,264	22,176	33,088	44,352	55,264	66,176	77,088	88,352	99,264	110,176
rule_table (ไบต์)	268,000	536,000	804,000	1,072,000	1,340,000	1,608,000	1,876,000	2,144,000	2,412,000	2,680,000
bil_table analysis (ไบต์)	11,000	22,000	33,000	44,000	55,000	66,000	77,000	88,000	99,000	110,000
bil_table + rule_table (ไบต์)	279,264	558,176	837,088	1,116,352	1,395,264	1,674,176	1,953,088	2,232,352	2,511,264	2,790,176



รูปที่ 4.41 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต)

ตารางที่ 4.34 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

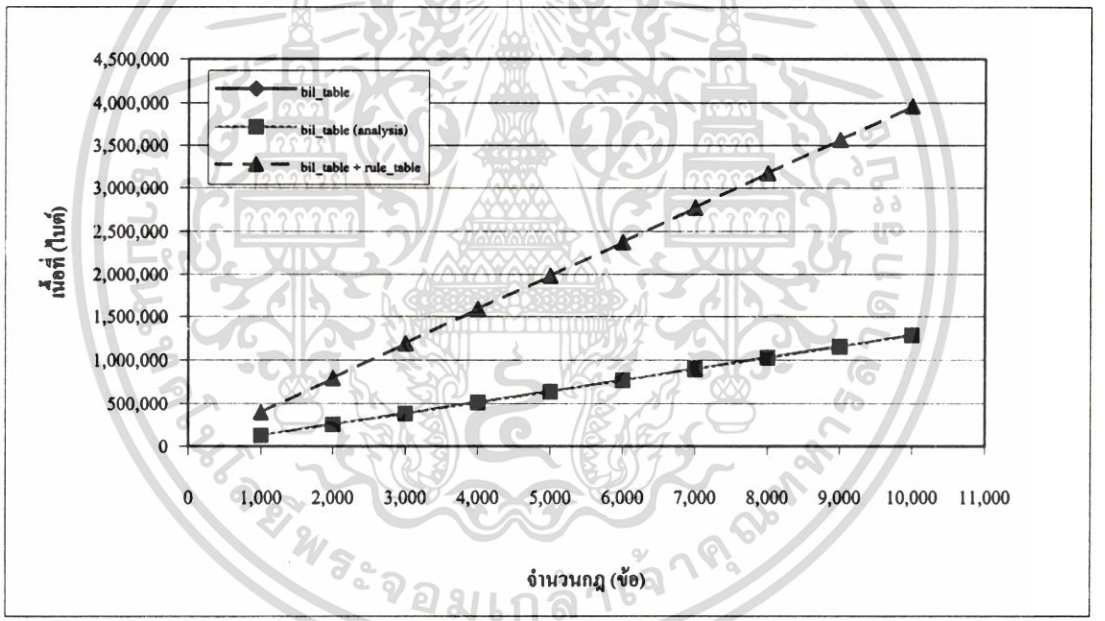
Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
bil_table (ไบต์)	81,920	161,280	240,640	322,560	401,920	481,280	560,640	642,560	721,920	801,280
rule_table (ไบต์)	268,000	536,000	804,000	1,072,000	1,340,000	1,608,000	1,876,000	2,144,000	2,412,000	2,680,000
bil_table analysis (ไบต์)	80,000	160,000	240,000	320,000	400,000	480,000	560,000	640,000	720,000	800,000
bil_table + rule_table (ไบต์)	349,920	697,280	1,044,640	1,394,560	1,741,920	2,089,280	2,436,640	2,786,560	3,133,920	3,481,280



รูปที่ 4.42 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 8 (7+7+7+7+4 บิต)

ตารางที่ 4.35 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
bil_table (ไบต์)	131,072	258,048	385,024	516,096	643,072	770,048	897,024	1,028,096	1,155,072	1,282,048
rule_table (ไบต์)	268,000	536,000	804,000	1,072,000	1,340,000	1,608,000	1,876,000	2,144,000	2,412,000	2,680,000
bil_table analysis (ไบต์)	128,000	256,000	384,000	512,000	640,000	768,000	896,000	1,024,000	1,152,000	1,280,000
bil_table + rule_table (ไบต์)	399,072	794,048	1,189,024	1,588,096	1,983,072	2,378,048	2,773,024	3,172,096	3,567,072	3,962,048



รูปที่ 4.43 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต)

วิเคราะห์ผลการทดลอง

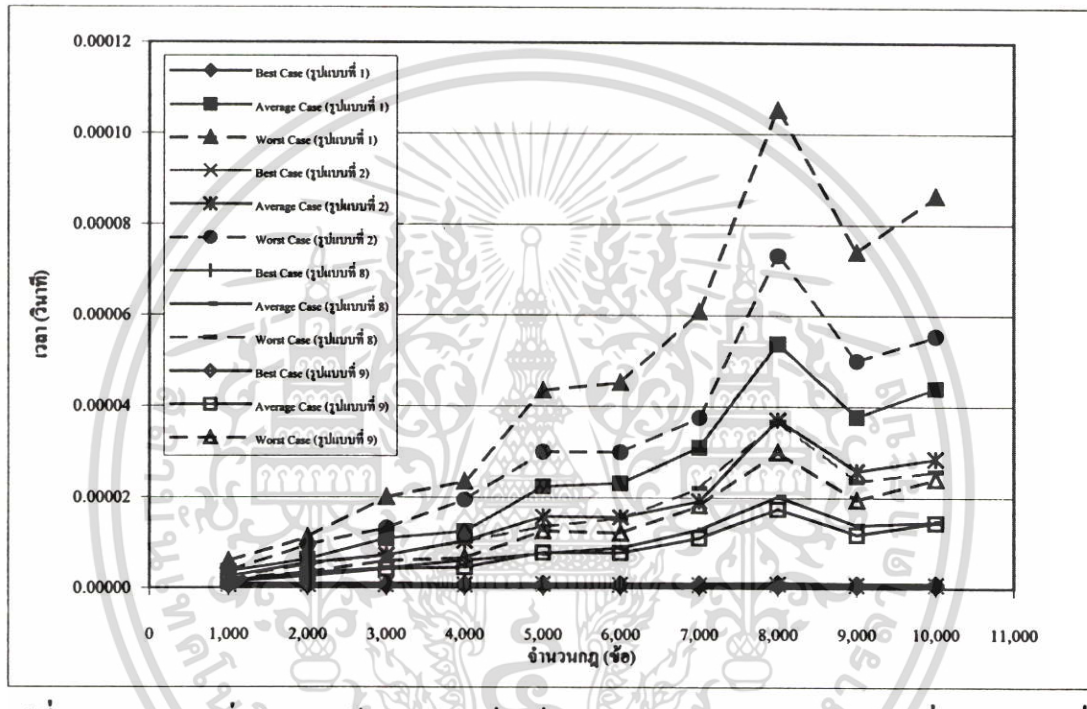
1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎขนาด 1 ฟิลด์ เป็น Source IP Address (W เท่ากับ 32 บิต) โดยกำหนดรูปแบบ Prefix 4 รูปแบบ เปรียบเทียบกัน คือ รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) รูปแบบที่ 8 (7+7+7+7+4 บิต) และรูปแบบที่ 9 (8+8+8+8 บิต) ตามลำดับ จากการวิเคราะห์พบว่าในกรณี Best Case โปรแกรมจะมี Search Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX}\right)$ ซึ่งจะเห็นว่าเวลาในการค้นหาของโปรแกรมจะไม่ขึ้นอยู่กับค่าของ N จากผลการทดลองในตารางที่ 4.20-4.23 พบว่าที่จำนวนกฎตั้งแต่ 1,000 – 10,000 ข้อ โปรแกรมจะใช้เวลาในการค้นหาใกล้เคียงกัน ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา โดยรูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000096 วินาที รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000077 วินาที รูปแบบที่ 8 (7+7+7+7+4 บิต) มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000058 วินาที และ รูปแบบที่ 9 (8+8+8+8 บิต) มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000051 วินาที

2. จากการวิเคราะห์พบว่าในกรณี Worst Case โปรแกรมจะมี Search Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}\right)$ ซึ่งจะเห็นว่าเมื่อ N มีค่าสูงขึ้น ความเร็วในการค้นหาจะมีค่าสูงขึ้นในลักษณะเชิงเส้น จากกราฟในรูปที่ 4.28-4.31 พบว่าเมื่อจำนวนกฎสูงขึ้นโปแกรมจะมีแนวโน้มในการใช้เวลาการค้นหาสูงขึ้น แต่เนื่องจากเส้นกราฟจะมีความแกว่งอย่างเห็นได้ชัด โดยเฉพาะที่จำนวนกฎเท่ากับ 9,000-10,000 ข้อ โปรแกรมจะมีเวลาในการค้นหาต่ำกว่าที่จำนวนกฎเท่ากับ 8,000 ข้อ สำหรับ Prefix ทุกๆ รูปแบบ โดยคาดว่าสาเหตุประการหนึ่งของการแกว่งของเส้นกราฟจะเกิดจากการประกาศตัวแปรเป็นชนิด register ในฟังก์ชัน classify ของโปรแกรม BIL ดังนั้นเพื่อพิสูจน์ว่าที่ N ใดๆ เส้นกราฟจะมีรูปแบบความสัมพันธ์ของเวลาในการค้นหาต่อแพ็กเก็ตเป็นแบบเชิงเส้น และสาเหตุประการหนึ่งของการแกว่งของเส้นกราฟเกิดจากการประกาศตัวแปรเป็นชนิด register ผู้วิจัยจึงทำการทดลองวัดเวลาในการค้นหาของโปรแกรม ดังในภาคผนวก ค. และผนวก ง. ซึ่งจากผลการทดลองพบว่าเป็นไปตามสมมติฐานดังที่กล่าวมา

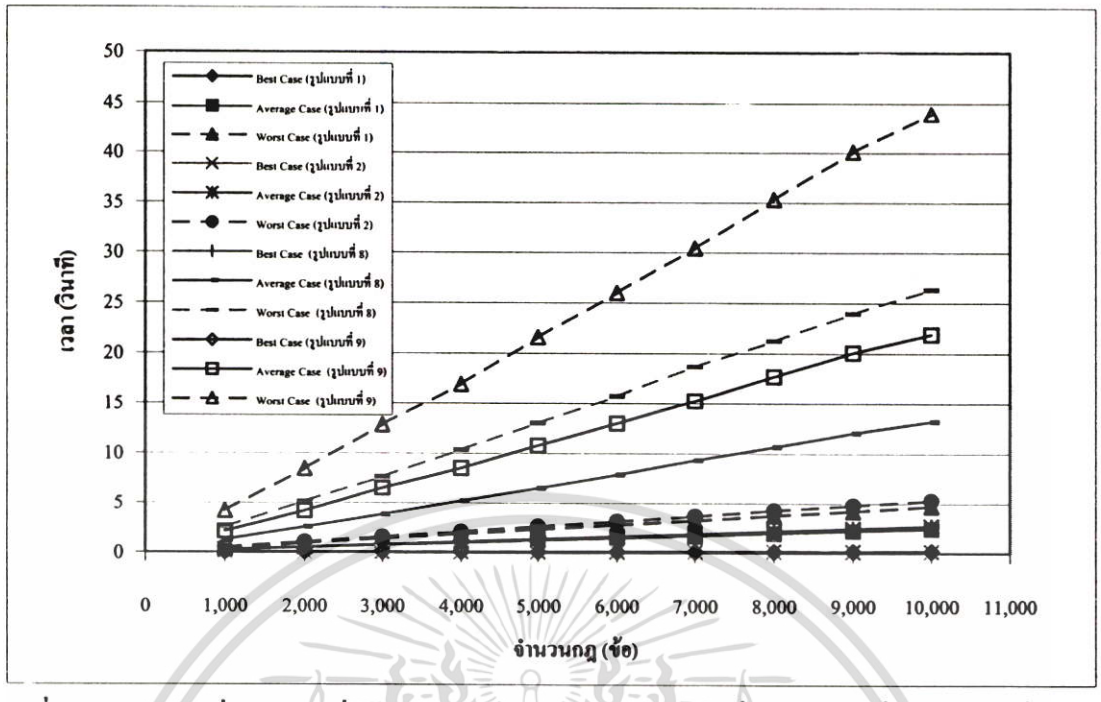
3. จากการวิเคราะห์ (เวลาในการปรับปรุงกฎที่จำนวนกฎ N ข้อ) พบว่าในกรณี Best Case โปรแกรมจะมี Update Speed Complexity เท่ากับ $O\left(N \cdot \frac{D \cdot W}{PMX}\right)$ และในกรณี Worst Case โปรแกรมจะมี Update Speed Complexity เท่ากับ $O\left(N \cdot (2 \cdot W - 2) \cdot \frac{D \cdot W}{PMX} \cdot 2^{PMX}\right)$ ซึ่งจะเห็นว่าเมื่อ N มีค่าสูงขึ้น เวลาในการปรับปรุงกฎจะมีค่าสูงขึ้นเช่นกัน จากกราฟในรูปที่ 4.32-4.39 พบว่า

เมื่อจำนวนกฎสูงขึ้นโปรแกรมจะมีแนวโน้มในการใช้เวลาการปรับปรุงกฎสูงขึ้นในทุกกรณี ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

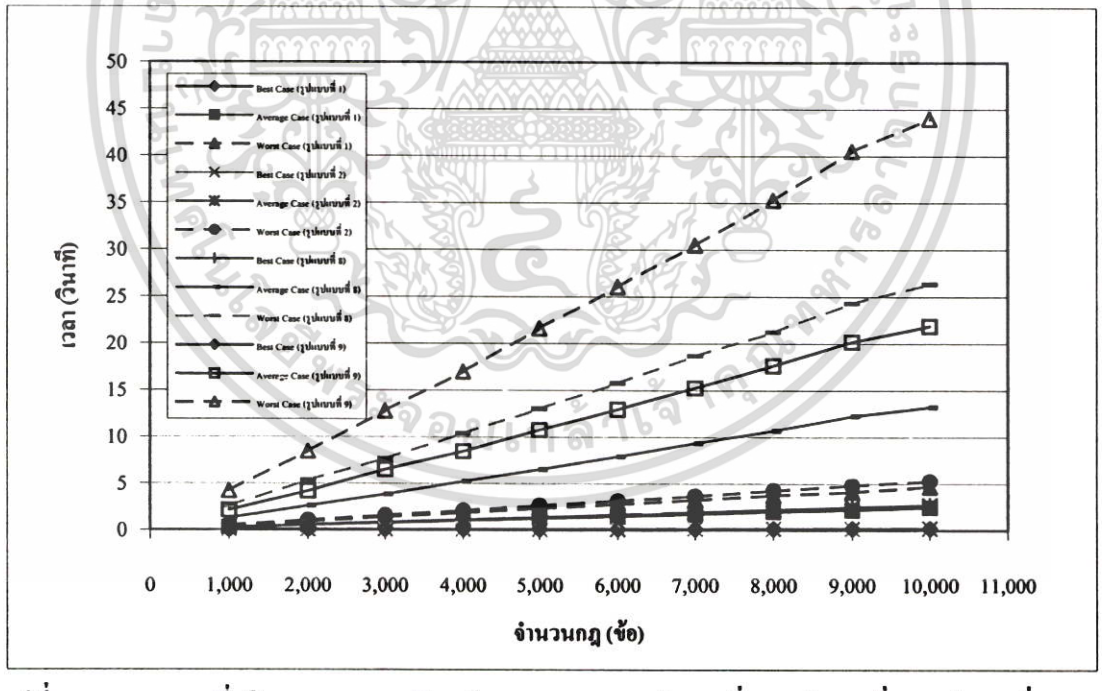
4. จากการวิเคราะห์พบว่าโปรแกรมจะมี Storage Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot N\right)$ ซึ่งจะเห็นว่าเมื่อ N มีค่าสูงขึ้น เนื้อที่จัดเก็บข้อมูลจะสูงขึ้นเช่นกัน จากกราฟในรูปที่ 4.40-4.43 พบว่าเมื่อจำนวนกฎสูงขึ้นโปรแกรมจะมีแนวโน้มในการใช้เนื้อที่จัดเก็บข้อมูลสูงขึ้น ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา



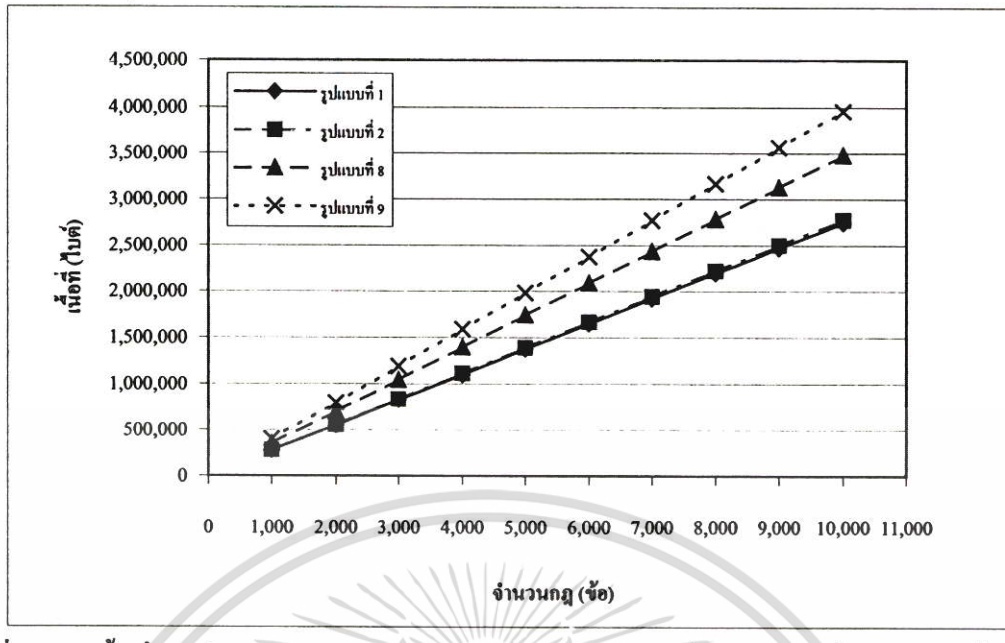
รูปที่ 4.44 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9



รูปที่ 4.45 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9



รูปที่ 4.46 เวลาเฉลี่ยในการลบกฎ (วินาที) ของ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9



รูปที่ 4.47 เนื้อที่จัดเก็บข้อมูลรวม (ไบต์) ของ Prefix รูปแบบที่ 1 รูปแบบที่ 2 รูปแบบที่ 8 และรูปแบบที่ 9

5. จากการเปรียบเทียบเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตจากกราฟในรูปที่ 4.44 พบว่าที่จำนวนกฎจำนวน 1,000-10,000 ข้อ ในทุกกรณี มีความเร็วในการค้นหาเรียงลำดับจากมากไปน้อย ได้แก่ รูปแบบที่ 9 (8+8+8+8 บิต) รูปแบบที่ 8 (7+7+7+7+4 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) และรูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) ตามลำดับ

6. จากการเปรียบเทียบเวลาเฉลี่ยในการปรับปรุงกฎจากกราฟในรูปที่ 4.45 และ 4.46 พบว่าที่จำนวนกฎจำนวน 1,000-10,000 ข้อ ในกรณี Best Case มีความเร็วในการปรับปรุงกฎเรียงลำดับจากมากไปน้อย ได้แก่ รูปแบบที่ 9 (8+8+8+8 บิต) รูปแบบที่ 8 (7+7+7+7+4 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) และรูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) ตามลำดับ ในขณะที่ในกรณี Average Case และกรณี Worst Case มีความเร็วในการปรับปรุงกฎเรียงลำดับจากมากไปน้อย ได้แก่ รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) รูปแบบที่ 8 (7+7+7+7+4 บิต) และรูปแบบที่ 9 (8+8+8+8 บิต) ตามลำดับ

7. จากการเปรียบเทียบการใช้เนื้อที่จัดเก็บข้อมูลจากกราฟในรูปที่ 4.47 พบว่าที่จำนวนกฎจำนวน 1,000-10,000 ข้อ มีการใช้เนื้อที่จัดเก็บข้อมูลเรียงลำดับจากน้อยไปมาก ได้แก่ รูปแบบที่ 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) รูปแบบที่ 8 (7+7+7+7+4 บิต) และรูปแบบที่ 9 (8+8+8+8 บิต) ตามลำดับ โดยที่รูปแบบที่ 1

(2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 บิต) และรูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+3+2 บิต) จะมีการใช้เนื้อที่ใกล้เคียงกัน

4.3 การเปรียบเทียบประสิทธิภาพการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

ในการทดลองส่วนที่ 2 จะจัดการทดลองขึ้นจำนวน 3 การทดลอง ได้แก่

1. การเปรียบเทียบเวลาในการค้นหาของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

2. การเปรียบเทียบเวลาในการปรับปรุงกฎของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

3. การเปรียบเทียบเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

โดยทั้ง 3 การทดลองจะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) ในการวัดเวลาในการค้นหาและเวลาในการปรับปรุงกฎจะทำการสร้างข้อมูลเป็น 3 กรณี ได้แก่ 1. กรณี Best Case 2. กรณี Average Case และ 3. กรณี Worst Case ดังที่ได้อธิบายในหัวข้อ 4.1.1 โดยผู้วิจัยจะเลือก Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+3+2 บิต) จากการทดลองที่ 4.2.3 ในการเปรียบเทียบประสิทธิภาพการทำงานกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection ซึ่งมีอัลกอริทึมดังภาคผนวก ข. เนื่องจากเป็นรูปแบบที่มีประสิทธิภาพที่ดีโดยทั้ง 3 ด้านมีความสมดุลกัน

4.3.1 การเปรียบเทียบเวลาในการค้นหาของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

วัตถุประสงค์การทดลอง เพื่อเปรียบเทียบเวลาในการค้นหาของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยใช้ฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)

2. กำหนดจำนวนกฎเท่ากับ 1,000, 2,000, 3,000, ..., 10,000 ข้อ และสร้างฐานข้อมูลกฎสำหรับวัดเวลาการค้นหา

3. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต

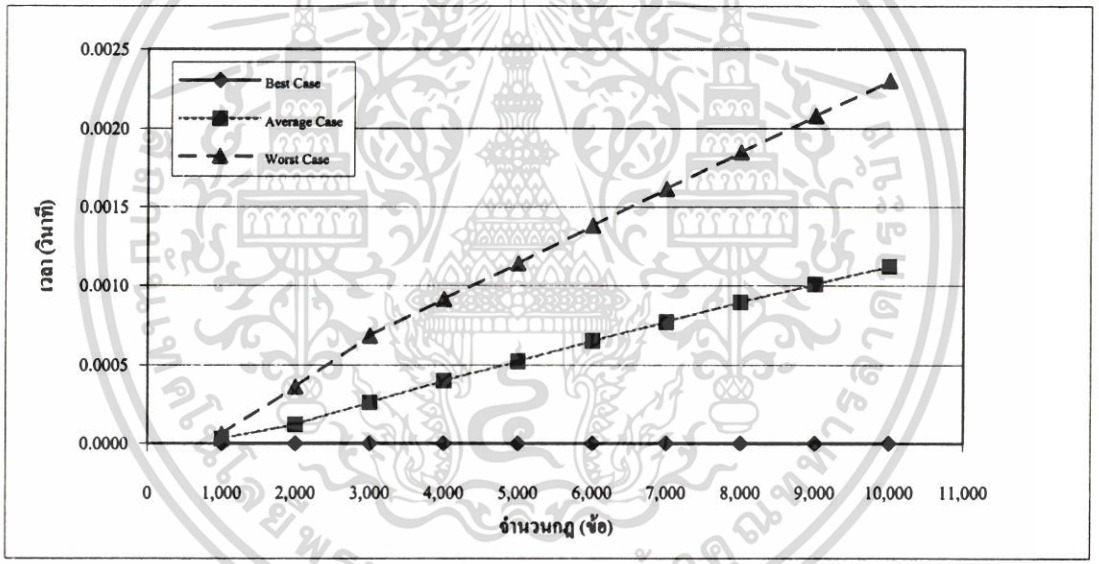
4. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

5. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

6. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

ตารางที่ 4.36 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Linear Search

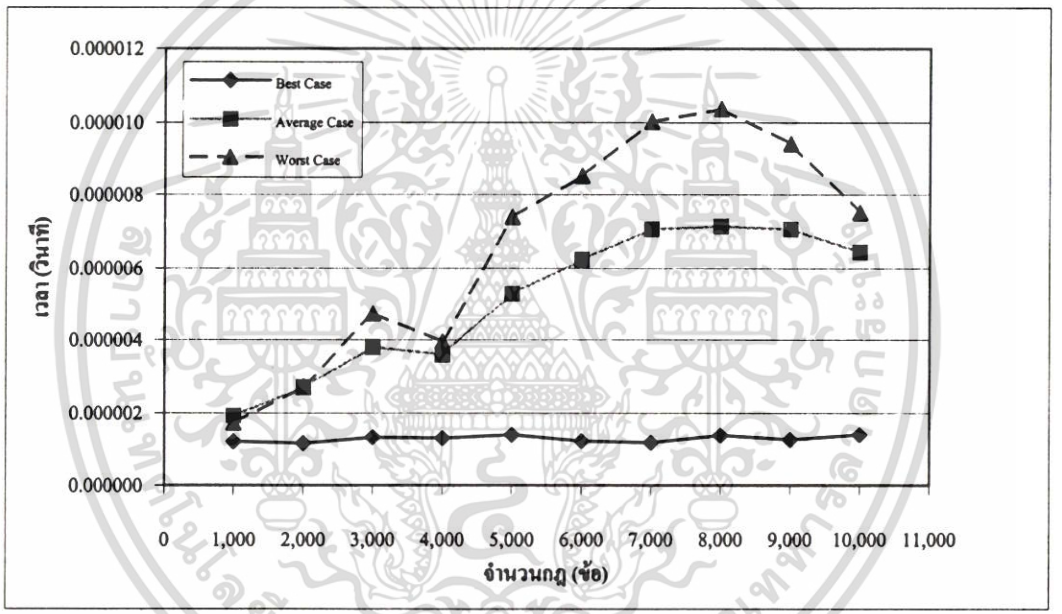
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000000219241000	0.000000219779000	0.000000352989000	0.000000219810000	0.000000219725000	0.000000219815000	0.000000219738000	0.000000219882000	0.000000218167000	0.000000218022000
Average Case (วินาที)	0.000029708088000	0.000120042531000	0.000262070082000	0.000401971487000	0.000521931021000	0.000650210987000	0.000768350386000	0.000891421428000	0.001007041467000	0.001122755030000
Worst Case (วินาที)	0.000059495154000	0.000363887789000	0.000682547921000	0.000912685649000	0.001142073231000	0.001383398256000	0.001614333255000	0.001847635135000	0.002081440197000	0.002303818840000



รูปที่ 4.48 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Linear Search

ตารางที่ 4.37 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Bitmap-intersection

กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.000001210873600	0.000001159453300	0.000001325511700	0.000001305046000	0.000001394850500	0.000001218475600	0.000001179282700	0.000001381370300	0.000001264424700	0.000001402583600
Average Case (วินาที)	0.000001925517400	0.000002702050900	0.000003800378800	0.000003594157500	0.000005280228800	0.000006245130100	0.000007071371600	0.000007142301500	0.000007069760700	0.000006441874900
Worst Case (วินาที)	0.000001723022500	0.000002726447000	0.000004721465200	0.000003944437100	0.000007398797700	0.000008525021500	0.000010023626400	0.000010370529400	0.000009408770200	0.000007515166900

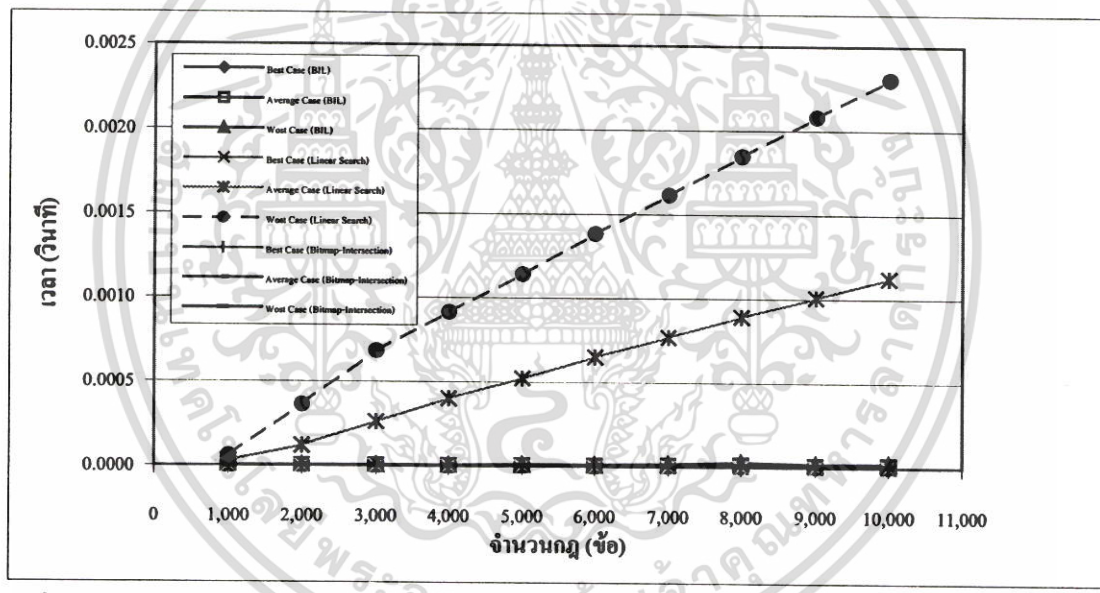


รูปที่ 4.49 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Bitmap-intersection

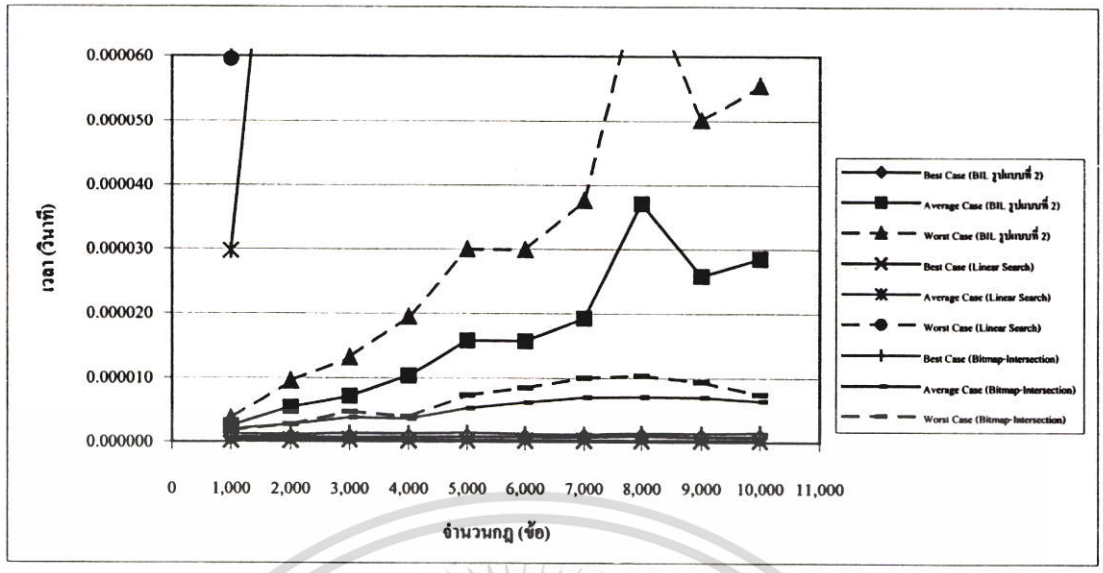
วิเคราะห์ผลการทดลอง

1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (W เท่ากับ 32 บิต) โดยเลือก Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) สำหรับเปรียบเทียบเวลาในการค้นหากับโปรแกรมอื่นๆ จากการทดลองในตารางที่ 4.36, 4.37 และ 4.21 ในกรณี Best Case พบว่าโปรแกรมจะใช้เวลาในการค้นหาใกล้เคียงกันที่จำนวนกฎตั้งแต่ 1,000 – 10,000 ข้อ โดยโปรแกรม Linear Search มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000023 วินาที โปรแกรม Bitmap-intersection มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000128 วินาที และโปรแกรม BIL มีเวลาในการค้นหาเฉลี่ยเท่ากับ 0.00000077 วินาที

2. จากกราฟในรูปที่ 4.48, 4.49 และ 4.29 พบว่าในกรณี Average Case และ กรณี Worst Case โปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL จะมีแนวโน้มในการใช้เวลาการค้นหาสูงขึ้นเมื่อจำนวนกฎสูงขึ้น



รูปที่ 4.50 ก เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL



รูปที่ 4.50 ข เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยกำหนดค่าแกน y มากที่สุดเท่ากับ 0.00006 วินาที

3. จากการเปรียบเทียบเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตดังกราฟในรูปที่ 4.50 ก และ 4.50 ข พบว่าที่จำนวนกฎ 1,000-10,000 ข้อ ในกรณี Best Case มีความเร็วในการค้นหาเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในขณะที่ในกรณี Average Case และกรณี Worst Case มีความเร็วในการค้นหาเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Bitmap-intersection 2. โปรแกรม BIL และ 3. โปรแกรม Linear Search ตามลำดับ

4.3.2 การเปรียบเทียบเวลาในการปรับปรุงกฎของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

วัตถุประสงค์การทดลอง เพื่อเปรียบเทียบเวลาในการปรับปรุงกฎของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยใช้ฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

วิธีการทดลอง

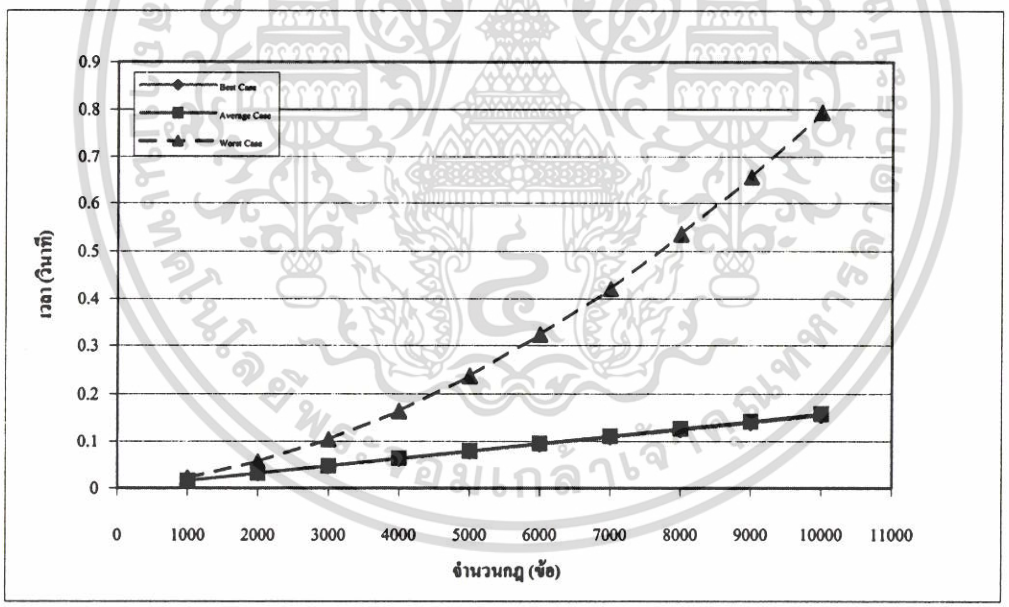
1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดจำนวนกฎเท่ากับ 1,000, 2,000, 3,000, ..., 10,000 ข้อ
3. สร้างฐานข้อมูลกฎสำหรับวัดเวลาการปรับปรุงกฎในกรณี Best Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

4. สร้างฐานข้อมูลกฎสำหรับวัดเวลาการปรับปรุงกฎในกรณี Average Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

5. สร้างฐานข้อมูลกฎสำหรับวัดเวลาการปรับปรุงกฎในกรณี Worst Case กับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

ตารางที่ 4.38 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Linear Search

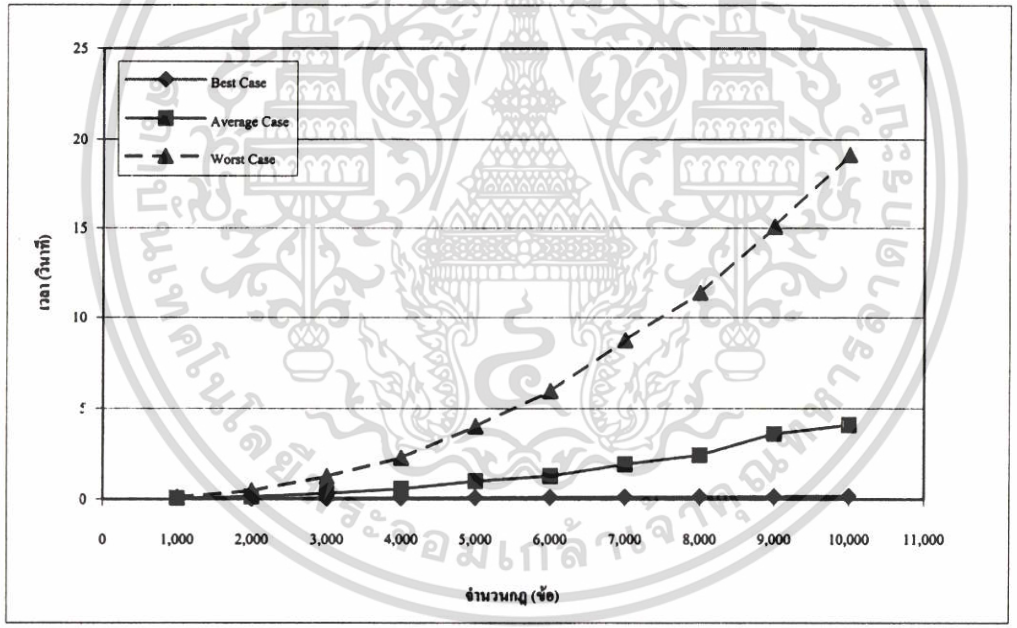
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01552583	0.03103093	0.04646674	0.06213695	0.07779185	0.0935079	0.10881675	0.1243516	0.13969717	0.15583927
Average Case (วินาที)	0.01576908	0.03171288	0.04741614	0.0635825	0.07926956	0.09542625	0.11102308	0.12738304	0.14273167	0.15893497
Worst Case (วินาที)	0.02188436	0.0564333	0.10388594	0.1641829	0.23731836	0.32282245	0.42117539	0.53555461	0.65602373	0.7943561



รูปที่ 4.51 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Linear Search

ตารางที่ 4.39 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Bitmap-intersection

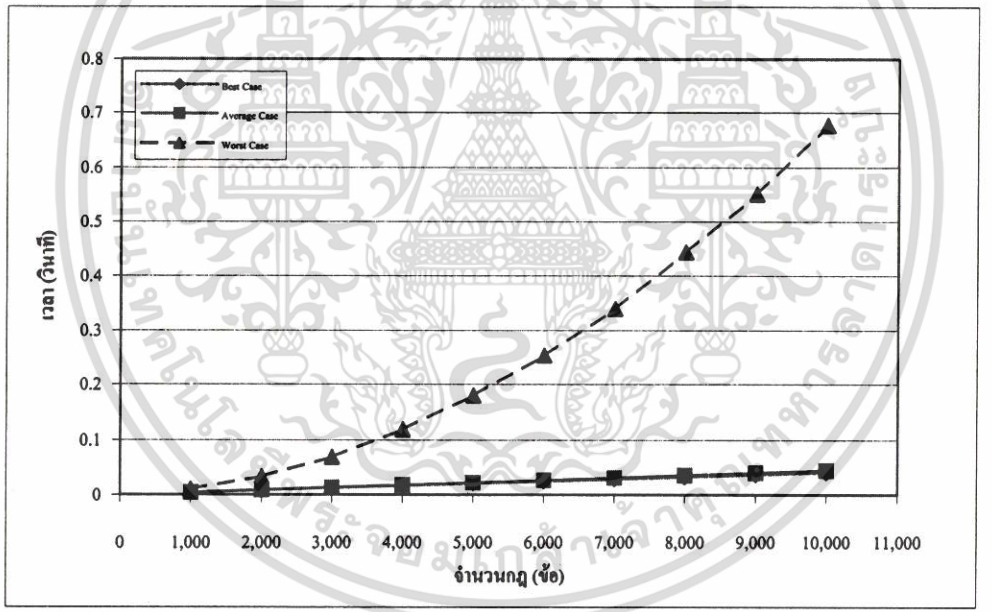
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.01792819	0.03592554	0.05401067	0.07178224	0.08958129	0.10287454	0.12308899	0.13968686	0.15308929	0.17491483
Average Case (วินาที)	0.04566952	0.15382656	0.32454345	0.55743214	0.97847693	1.25090027	1.89169945	2.40544425	3.58299967	4.08655952
Worst Case (วินาที)	0.11741049	0.46257578	1.23300944	2.26406283	4.00614419	5.95918145	8.74125825	11.4144461	15.0929982	19.1319466



รูปที่ 4.52 เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Bitmap-intersection

ตารางที่ 4.40 เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Linear Search

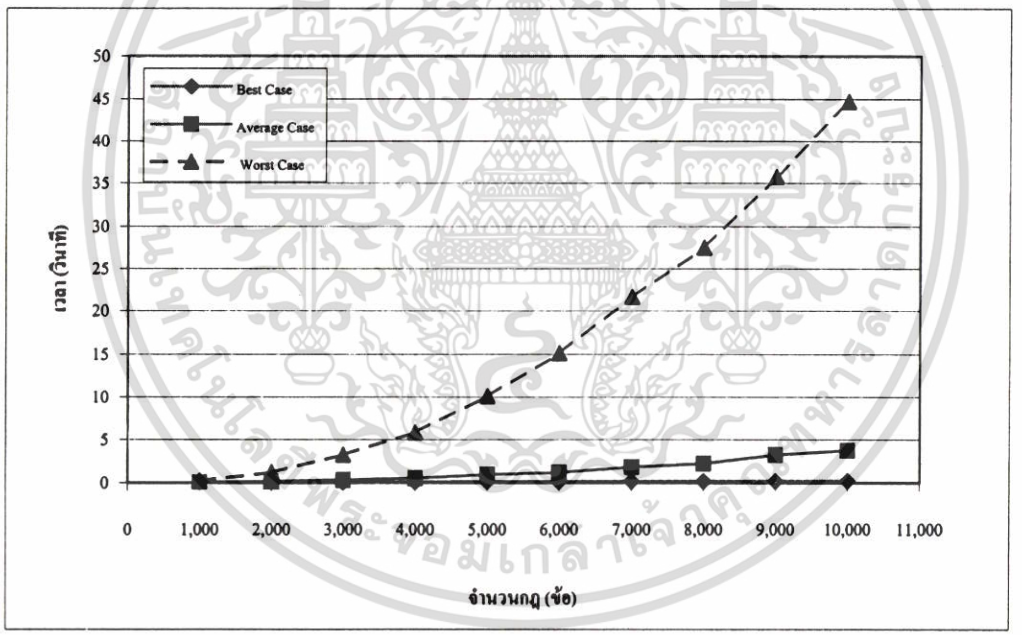
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.004127	0.008263	0.012598	0.016828	0.020908	0.025088	0.029196	0.033373	0.037296	0.041644
Average Case (วินาที)	0.004302	0.008786	0.013456	0.017955	0.022491	0.026885	0.031478	0.036232	0.040572	0.045326
Worst Case (วินาที)	0.010543	0.033807	0.069547	0.119025	0.179618	0.253506	0.340122	0.443663	0.551362	0.676888



รูปที่ 4.53 เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Linear Search

ตารางที่ 4.41 เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Bitmap-intersection

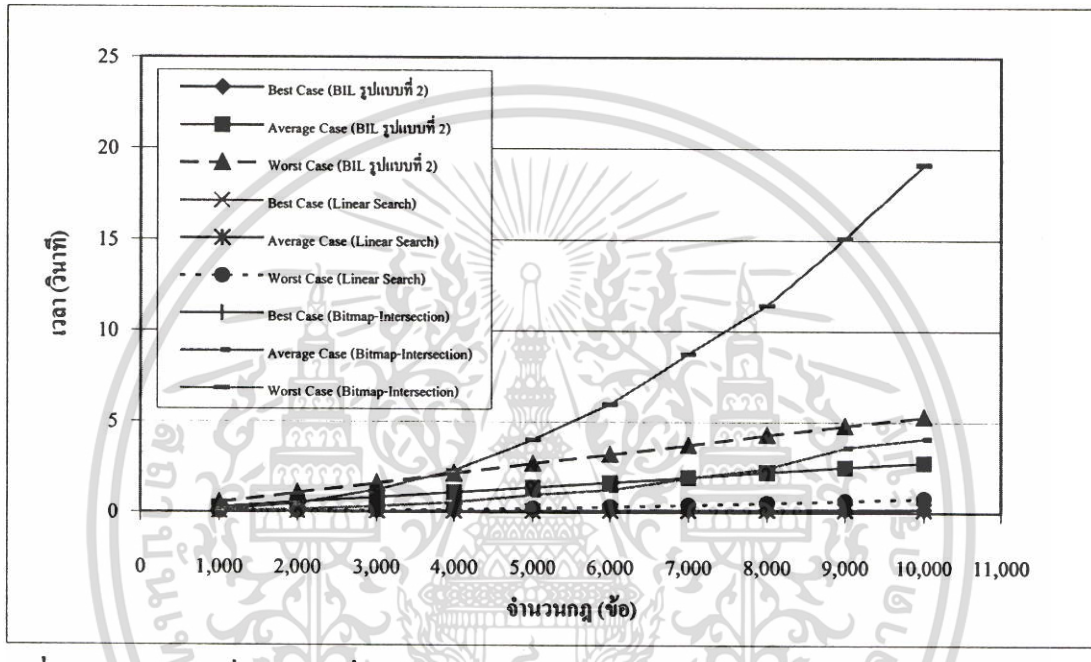
กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (วินาที)	0.02058498	0.04307411	0.06465114	0.08542679	0.11317384	0.1354922	0.1660025	0.18293233	0.20888575	0.22244945
Average Case (วินาที)	0.04693288	0.1516784	0.31418322	0.53077262	0.93536968	1.19325691	1.8027124	2.23247709	3.23371121	3.74894549
Worst Case (วินาที)	0.22049838	1.18381194	3.23247242	5.84201839	10.1171741	15.1295976	21.6526115	27.5161123	35.7819068	44.7316725



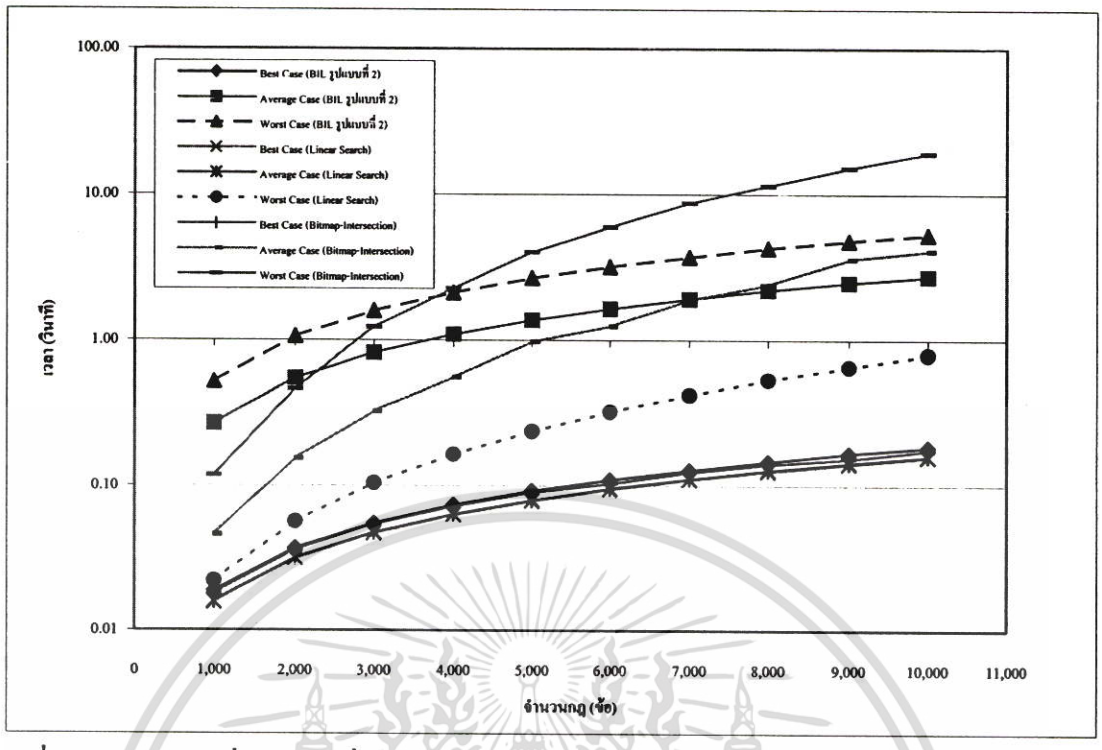
รูปที่ 4.54 เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Bitmap-intersection

วิเคราะห์ผลการทดลอง

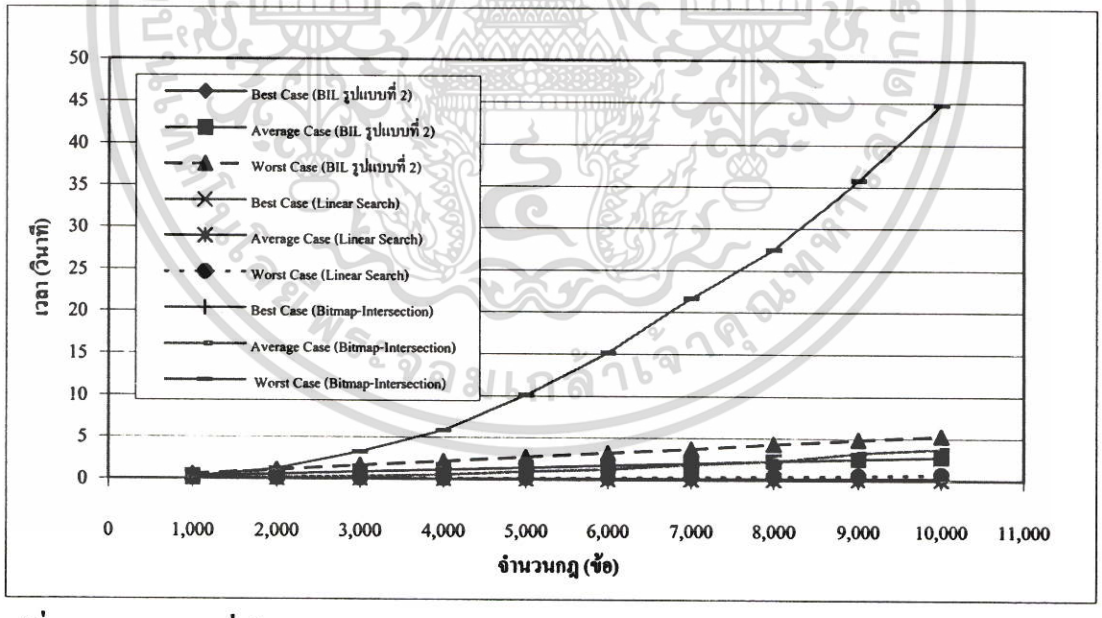
1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (W เท่ากับ 32 บิต) โดยเลือก Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) สำหรับเปรียบเทียบเวลาในการปรับปรุงกฎกับโปรแกรมอื่นๆ จากกราฟในรูปที่ 4.51, 4.52, 4.53, 4.54, 4.25 และ 4.37 พบว่าในทุกกรณี โปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL จะมีแนวโน้มในการใช้เวลาการปรับปรุงกฎสูงขึ้นเมื่อจำนวนกฎสูงขึ้น



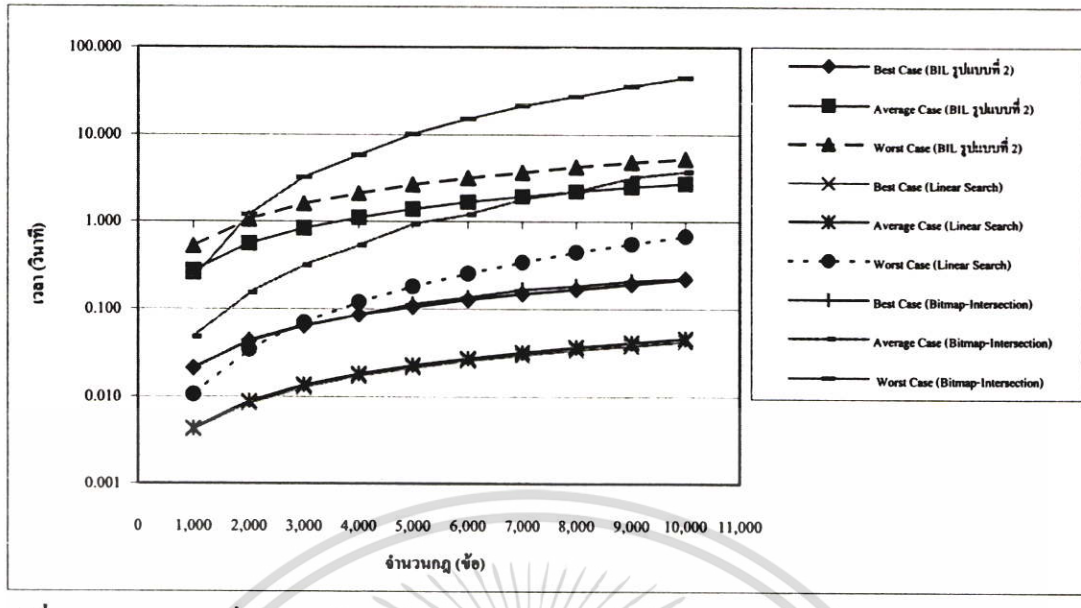
รูปที่ 4.55 ก เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL



รูปที่ 4.55 ข เวลาเฉลี่ยในการเพิ่มกฎ (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยใช้มาตราส่วนลอการิทึม



รูปที่ 4.56 ก เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL



รูปที่ 4.56 ข เวลาเฉลี่ยในการลบกฎ (วินาที) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยใช้มาตราส่วนลอการิทึม

2. จากผลการทดลองวัดเวลาในการปรับปรุงกฎ ดังกราฟในรูปที่ 4.55-4.56 ซึ่งข้อมูลที่ใช้ในการทดลองในกรณี Worst Case ของโปรแกรม Bitmap-intersection จะเป็นข้อมูลที่มีลักษณะเดียวกับข้อมูลในกรณี Average Case ของโปรแกรม BIL และข้อมูลในกรณี Average Case ของโปรแกรม Bitmap-intersection จะเป็นข้อมูลที่มีลักษณะเดียวกับข้อมูลในกรณี Best Case ของโปรแกรม BIL ดังที่กำหนดไว้ในหัวข้อ 4.1.1 โดยจากการทดลองพบว่าโปรแกรม BIL จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรม Bitmap-intersection อย่างเห็นได้ชัด สำหรับกฎที่มีลักษณะเป็นช่วง (range) หรือเน็ตมาสค์ (net mask) แคบๆ

3. จากการเปรียบเทียบเวลาเฉลี่ยในการเพิ่มกฎดังกราฟในรูปที่ 4.55 ก และ 4.55 ข พบว่าในกรณี Worst Case โปรแกรม Bitmap-intersection จะมีแนวโน้มที่จะมีความเร็วในการเพิ่มกฎต่ำกว่าโปรแกรมอื่นๆอย่างเห็นได้ชัด โดยที่จำนวนกฎ 1,000-10,000 ข้อ ในกรณี Best Case มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ในกรณี Average Case ที่จำนวนกฎ 1,000-7,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ที่จำนวนกฎ 8,000-10,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Worst Case ที่จำนวนกฎ 1,000-4,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ

และที่จำนวนกฎ 5,000-10,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

4. จากการเปรียบเทียบเวลาเฉลี่ยในการลบกฎดังกราฟในรูปที่ 4.56 ก และ 4.56 ข พบว่า ในกรณี Worst Case โปรแกรม Bitmap-intersection จะมีแนวโน้มที่จะมีความเร็วในการลบกฎต่ำกว่าโปรแกรมอื่นๆ อย่างเห็นได้ชัด โดยที่จำนวนกฎจำนวน 1,000-10,000 ข้อ ในกรณี Best Case มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Average Case ที่จำนวนกฎ 1,000-8,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ที่จำนวนกฎ 9,000-10,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Worst Case ที่จำนวนกฎ 1,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ และที่จำนวนกฎ 2,000-10,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

4.3.3 การเปรียบเทียบเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

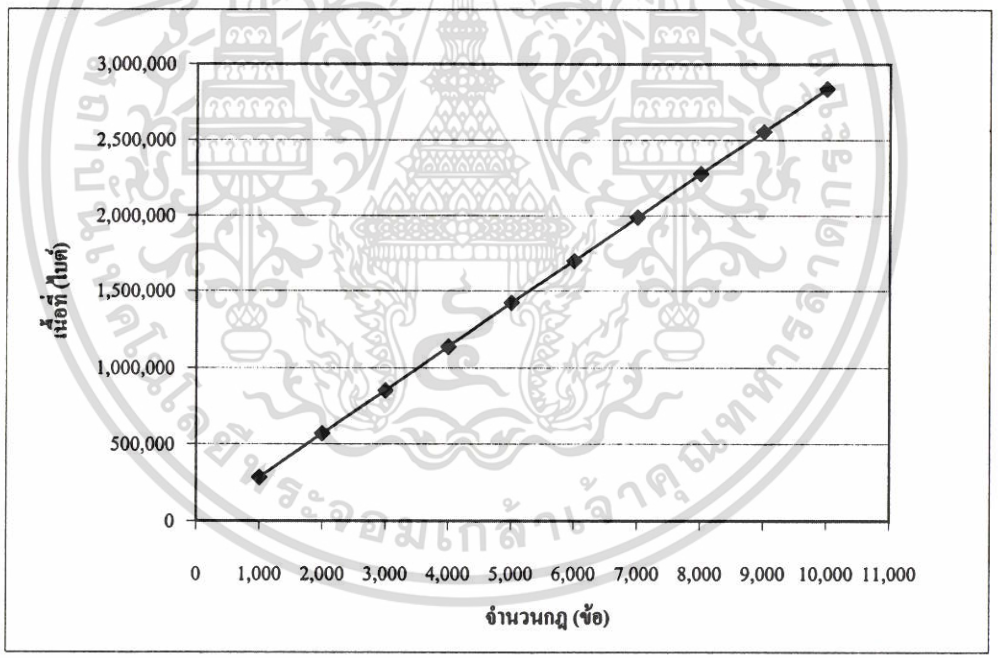
วัตถุประสงค์การทดลอง เพื่อเปรียบเทียบเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยใช้ฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดจำนวนกฎเท่ากับ 1,000, 2,000, 3,000, ..., 10,000 ข้อ
3. สร้างฐานข้อมูลกฎสำหรับวัดการใช้เนื้อที่จัดเก็บข้อมูลกับโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

ตารางที่ 4.42 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของโปรแกรม Linear Search

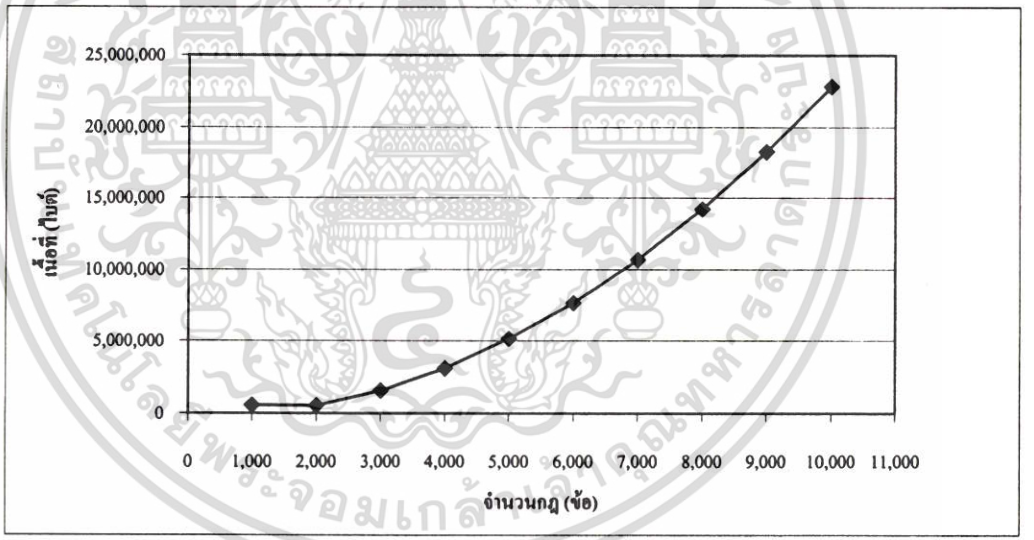
Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
fw_rule (ไบต์)	280,000	560,000	840,000	1,120,000	1,400,000	1,680,000	1,960,000	2,240,000	2,520,000	2,800,000
map_table (ไบต์)	4,000	8,000	12,000	16,000	20,000	24,000	28,000	32,000	36,000	40,000
รวม (ไบต์)	284,000	568,000	852,000	1,136,000	1,420,000	1,704,000	1,988,000	2,272,000	2,556,000	2,840,000



รูปที่ 4.57 เนื้อที่จัดเก็บข้อมูลรวม (ไบต์) ของโปรแกรม Linear Search

ตารางที่ 4.43 เนื้อที่จัดเก็บข้อมูล (ไบต์) ของโปรแกรม Bitmap-intersection

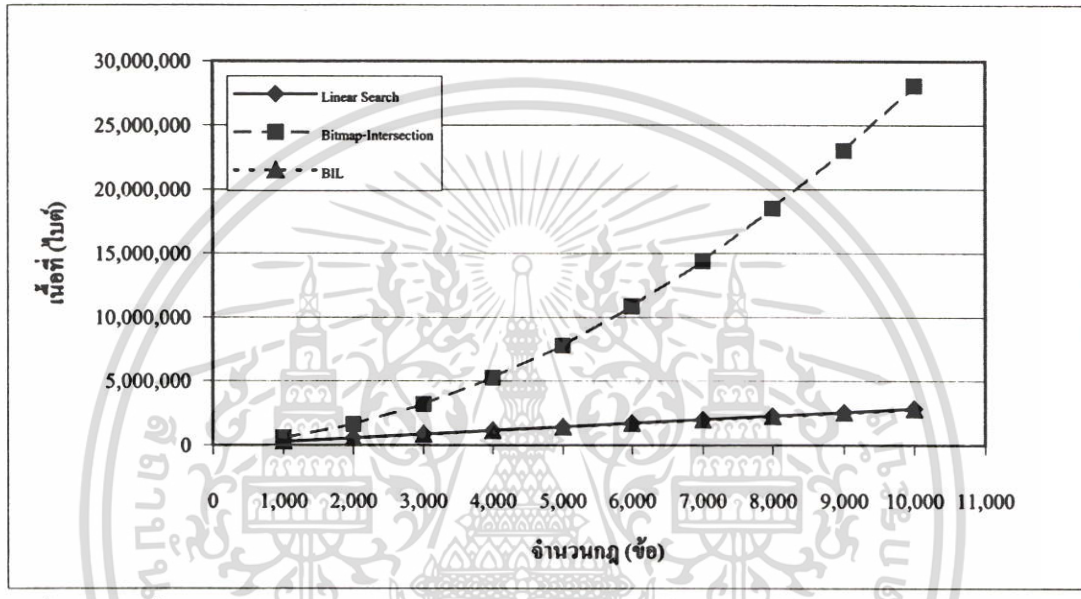
Data Structure	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
bitmap_table (ไบต์)	256,128	1,008,252	2,256,376	4,032,504	6,280,628	9,024,752	12,264,876	16,065,004	20,305,128	25,041,252
rule_table (ไบต์)	268,000	536,000	804,000	1,072,000	1,340,000	1,608,000	1,876,000	2,144,000	2,412,000	2,680,000
search_table (ไบต์)	8,012	16,012	24,012	32,012	40,012	48,012	56,012	64,012	72,012	80,012
allocate_table (ไบต์)	8,012	16,012	24,012	32,012	40,012	48,012	56,012	64,012	72,012	80,012
range record (ไบต์)	19,980	39,980	59,980	79,980	99,980	119,980	139,980	159,980	179,980	199,980
รวม (ไบต์)	560,132	1,616,256	3,168,380	5,248,508	7,800,632	10,848,756	14,392,880	18,497,008	23,041,132	28,081,256



รูปที่ 4.58 เนื้อที่จัดเก็บข้อมูลรวม (ไบต์) ของโปรแกรม Bitmap-intersection

วิเคราะห์ผลการทดลอง

1. ในการทดลองผู้วิจัยใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (W เท่ากับ 32 บิต) โดยเลือก Prefix รูปแบบที่ 2 (3+3+3+3+3+3+3+3+3+3+2 บิต) สำหรับเปรียบเทียบเนื้อที่จัดเก็บข้อมูลกับโปรแกรมอื่นๆ จากกราฟในรูปที่ 4.57, 4.58 และ 4.41 พบว่าโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL จะมีแนวโน้มในการเนื้อที่จัดเก็บข้อมูลสูงขึ้นเมื่อจำนวนกฎสูงขึ้น



รูปที่ 4.59 เนื้อที่จัดเก็บข้อมูลรวม (ไบต์) ของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

2. จากการเปรียบเทียบเนื้อที่จัดเก็บข้อมูลดังกราฟในรูปที่ 4.59 พบว่าที่จำนวนกฎ 1,000-10,000 ข้อ โปรแกรม BIL และโปรแกรม Linear Search มีเนื้อที่จัดเก็บข้อมูลใกล้เคียงกัน ในขณะที่โปรแกรม Bitmap-intersection มีเนื้อที่จัดเก็บข้อมูลสูงสุด โดยที่จำนวนกฎจำนวน 1,000-10,000 ข้อ มีเนื้อที่จัดเก็บข้อมูลเรียงลำดับจากน้อยไปมาก ได้แก่ 1. โปรแกรม BIL 2. โปรแกรม Linear Search และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

4.4 วิเคราะห์ผลการทดลองทั้งหมด

การทดลองที่จัดขึ้นเป็นการวัดประสิทธิภาพการทำงานในด้านความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลของโปรแกรมการจัดประเภทแพ็กเก็ตตามกฎ เพื่อทดสอบแนวคิดในงานวิจัยจึงแบ่งการทดลองออกเป็น 2 ส่วน คือ 1. การแบ่งขนาด Prefix เพื่อหาค่าที่ทำให้โปรแกรม BIL มีประสิทธิภาพในการทำงานดีที่สุด และ 2. การ

เปรียบเทียบประสิทธิภาพการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL

จากการศึกษาประสิทธิภาพการทำงานของโปรแกรม BIL ในการทดลองที่ 4.2.1-4.2.3 สามารถสรุปรูปแบบการแบ่ง Prefix ที่เหมาะสมสำหรับแต่ละด้านดังต่อไปนี้

1. ด้านความเร็วในการค้นหา ควรแบ่ง Prefix ให้มีจำนวน Block น้อยที่สุดเพื่อให้มีประสิทธิภาพในการค้นหาสูงสุด

2. ด้านความเร็วในการปรับปรุงกฎ จะขึ้นอยู่กับลักษณะของกฎเป็นสำคัญ โดยสำหรับกฎที่มีลักษณะเป็นช่วงควรแบ่ง Prefix เป็น Block ขนาดเล็กที่สุดเพื่อให้มีประสิทธิภาพในการปรับปรุงกฎสูงสุด สำหรับกฎที่มีลักษณะไม่เป็นช่วง ควรแบ่ง Prefix ให้มีจำนวน Block น้อยที่สุดเพื่อให้มีประสิทธิภาพในการปรับปรุงกฎสูงสุด

3. ด้านเนื้อที่จัดเก็บข้อมูล ควรแบ่ง Prefix เป็น Block ขนาดเล็กที่สุดเพื่อให้มีประสิทธิภาพในการใช้เนื้อที่จัดเก็บข้อมูลสูงสุด

นอกจากนี้ยังพบว่าการแบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน หรือใกล้เคียงกันจะทำให้โปรแกรมมีประสิทธิภาพในการทำงานดีกว่าการแบ่ง Prefix เป็น Block ที่มีขนาดแตกต่างกัน

สำหรับการเปรียบเทียบประสิทธิภาพการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ สามารถสรุปได้ดังต่อไปนี้

1. ในกรณี Best Case มีความเร็วในการค้นหาเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในขณะที่ในกรณี Average Case และกรณี Worst Case มีความเร็วในการค้นหาเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Bitmap-intersection 2. โปรแกรม BIL และ 3. โปรแกรม Linear Search ตามลำดับ

2. ในกรณี Best Case มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ในกรณี Average Case ที่จำนวนกฎ 1,000-7,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ที่จำนวนกฎ 8,000-10,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Worst Case ที่จำนวนกฎ 1,000-4,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ และที่จำนวนกฎ 5,000-10,000 ข้อ มีความเร็วในการเพิ่มกฎเรียงลำดับ

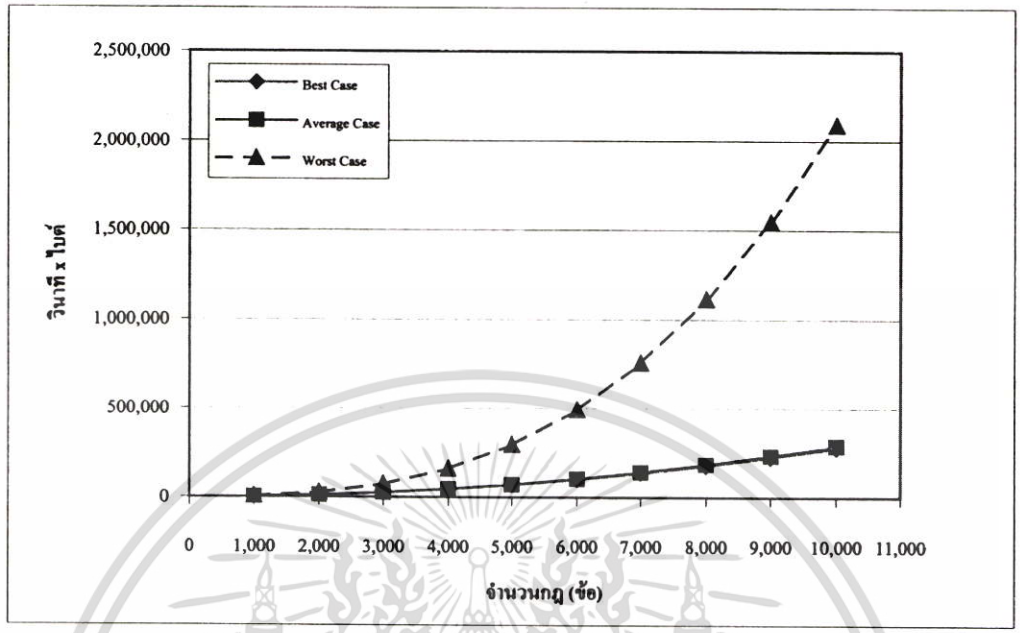
จากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

3. ในกรณี Best Case มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Average Case ที่จำนวนกฎ 1,000-8,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ ที่จำนวนกฎ 9,000-10,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ ในกรณี Worst Case ที่จำนวนกฎ 1,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม Bitmap-intersection และ 3. โปรแกรม BIL ตามลำดับ และที่จำนวนกฎ 2,000-10,000 ข้อ มีความเร็วในการลบกฎเรียงลำดับจากมากไปน้อย ได้แก่ 1. โปรแกรม Linear Search 2. โปรแกรม BIL และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

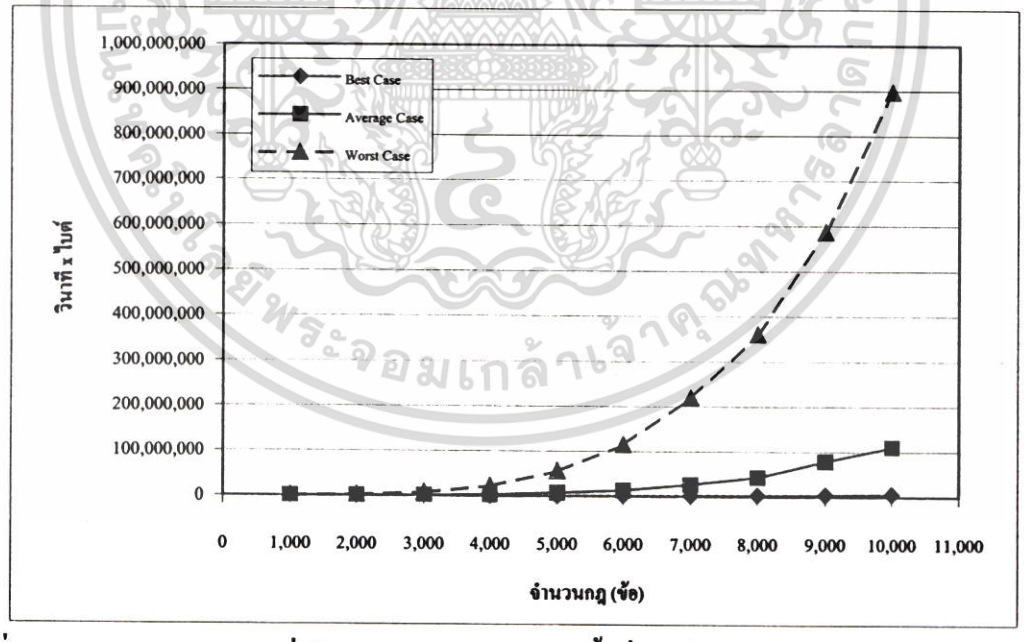
4. สำหรับเนื้อที่จัดเก็บข้อมูลเรียงลำดับจากน้อยไปมาก ได้แก่ 1. โปรแกรม BIL 2. โปรแกรม Linear Search และ 3. โปรแกรม Bitmap-intersection ตามลำดับ

5. จากผลการทดลองที่ 4.3.1-4.3.3 ผู้วิจัยจะทำการศึกษาประสิทธิภาพในการทำงานของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยพิจารณาจากผลคูณของประสิทธิภาพการทำงานจำนวน 4 ผลคูณตามที่เคยกล่าวในบทก่อนหน้า ดังต่อไปนี้

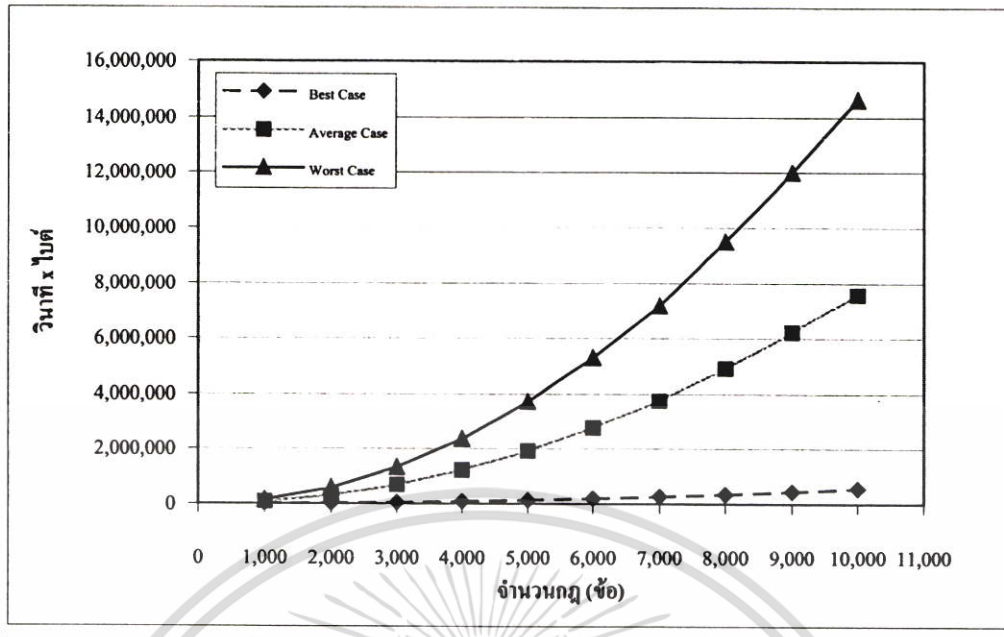
1. ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



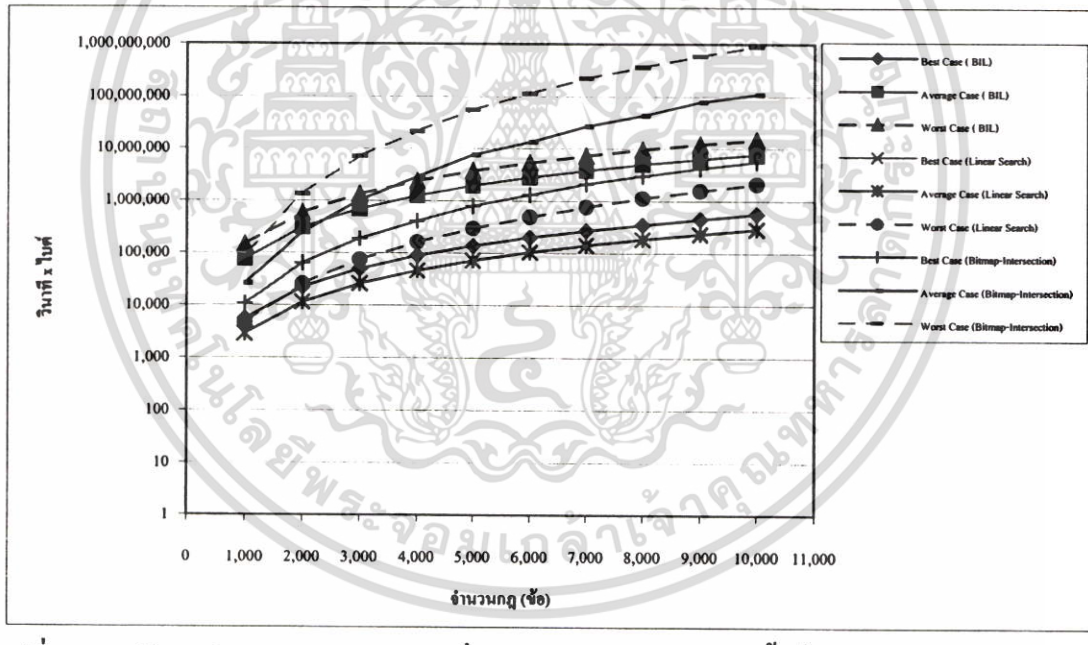
รูปที่ 4.60 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.61 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Bitmap-intersection ที่จำนวนกฎ 1,000-10,000 ข้อ

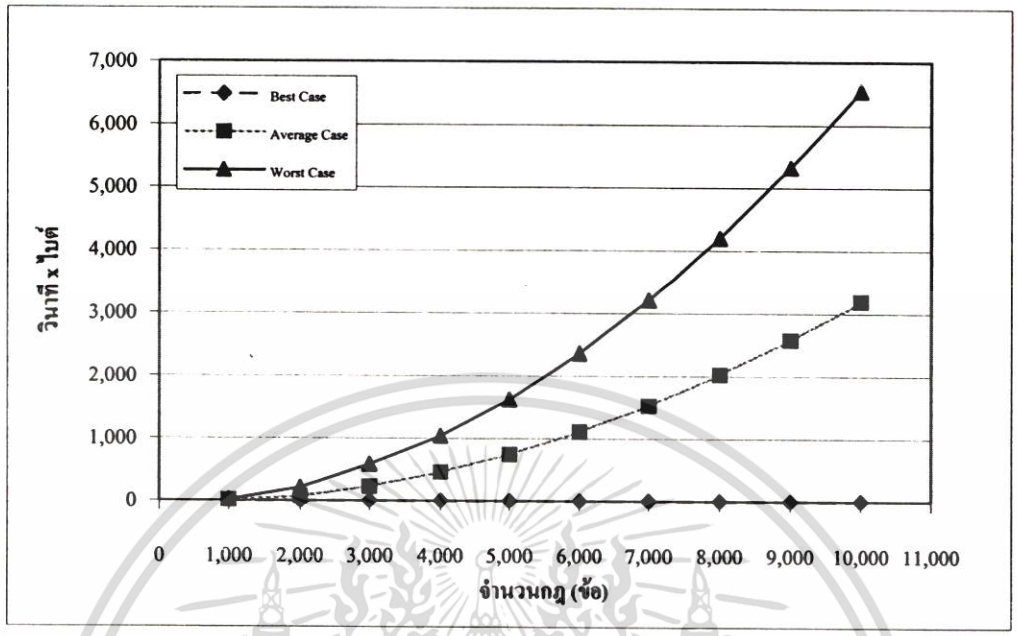


รูปที่ 4.62 ผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลของโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ

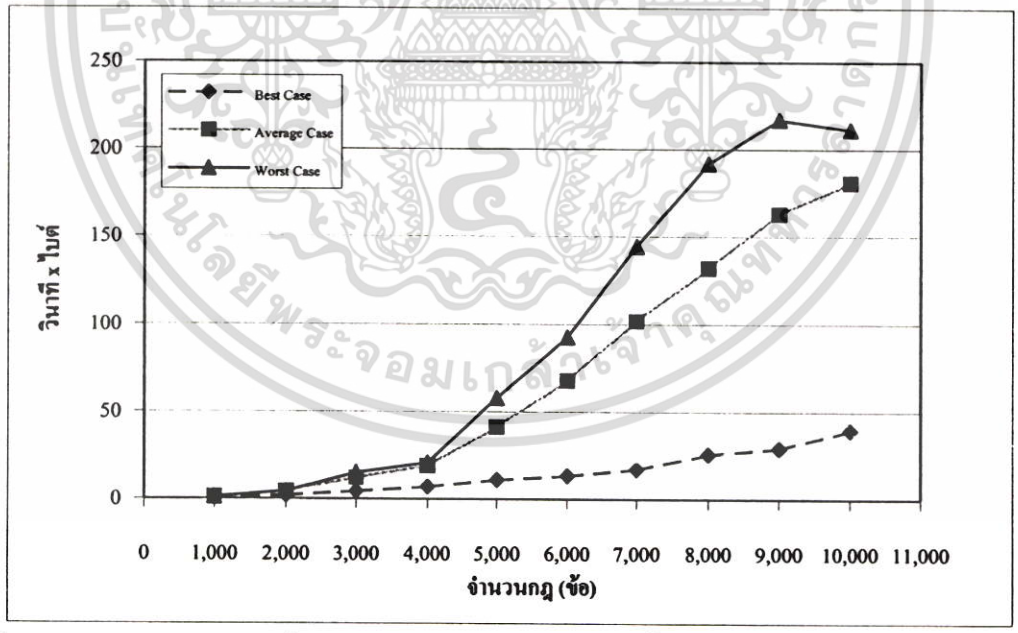


รูปที่ 4.63 เปรียบเทียบผลคูณของเวลาเฉลี่ยในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ โดยใช้มาตราส่วนลอการิทึม

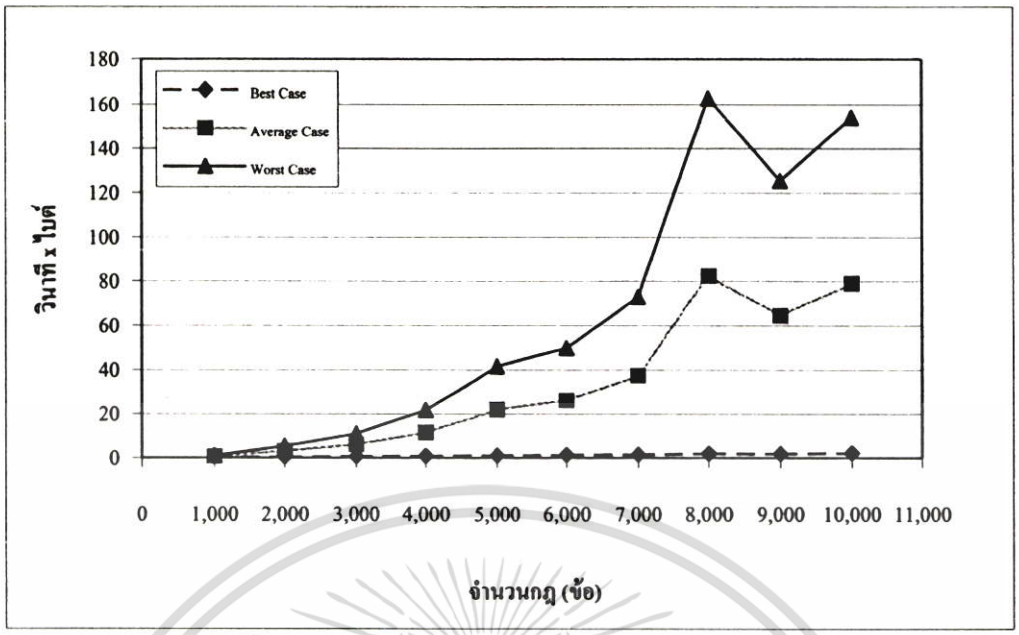
2. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล



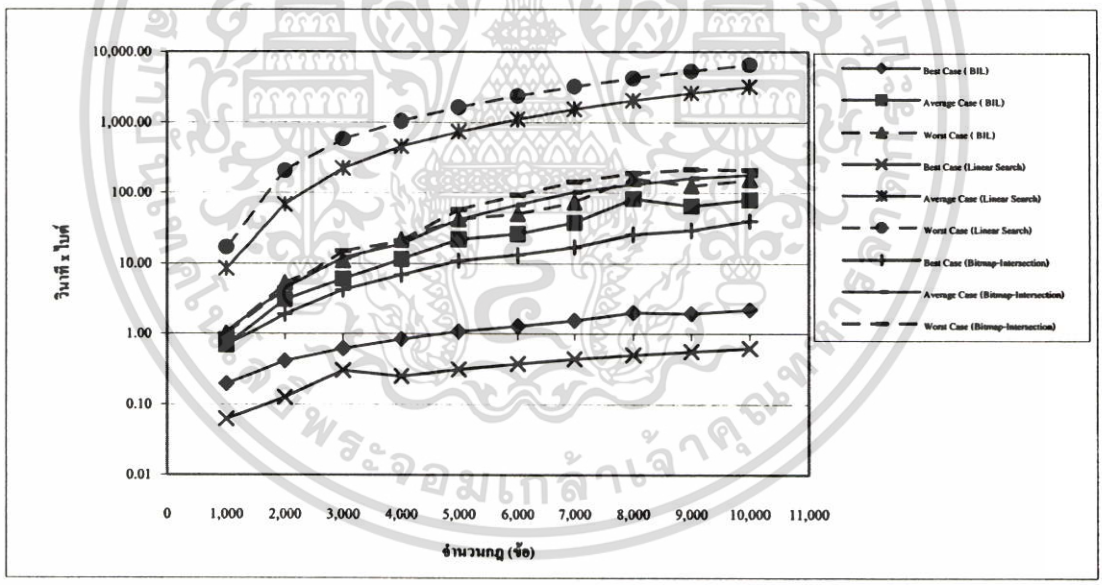
รูปที่ 4.64 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.65 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Bitmap-intersection ที่จำนวนกฎ 1,000-10,000 ข้อ

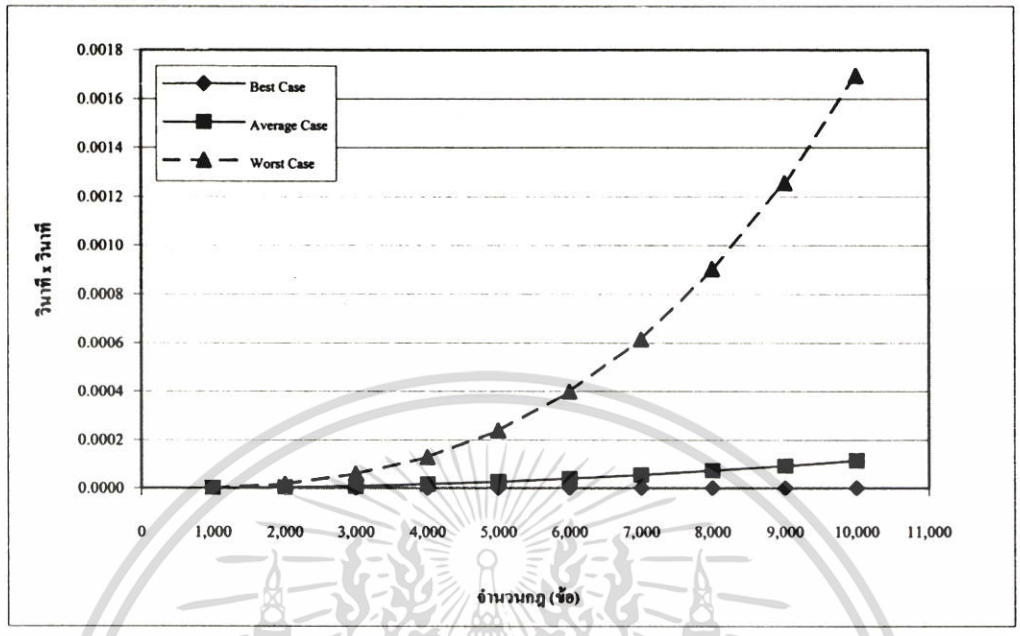


รูปที่ 4.66 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเนื้อที่จัดเก็บข้อมูลของโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ

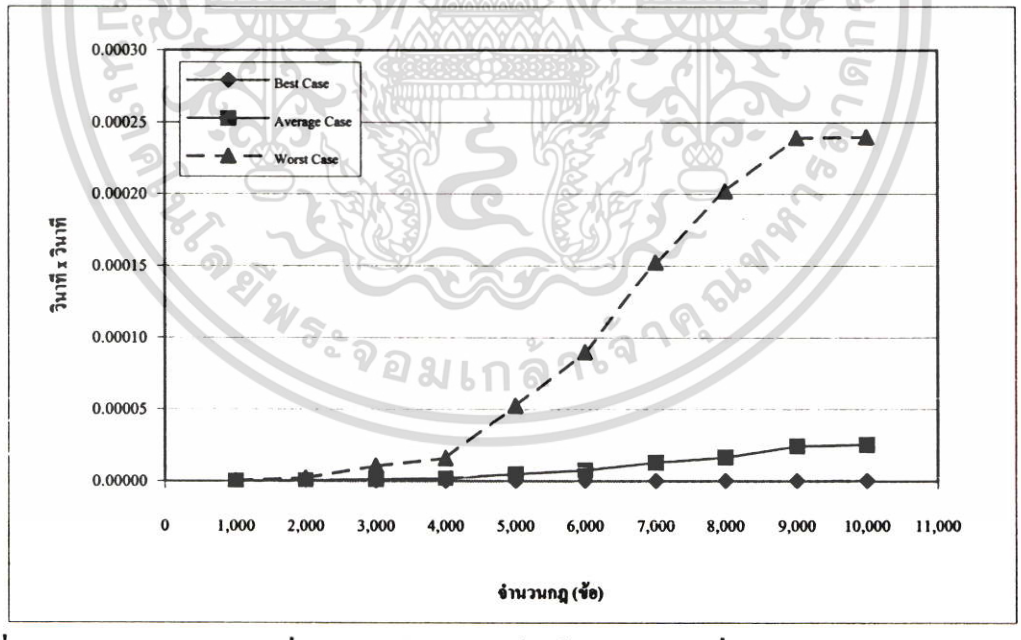


รูปที่ 4.67 เปรียบเทียบผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ โดยใช้มาตราส่วนลอการิทึม

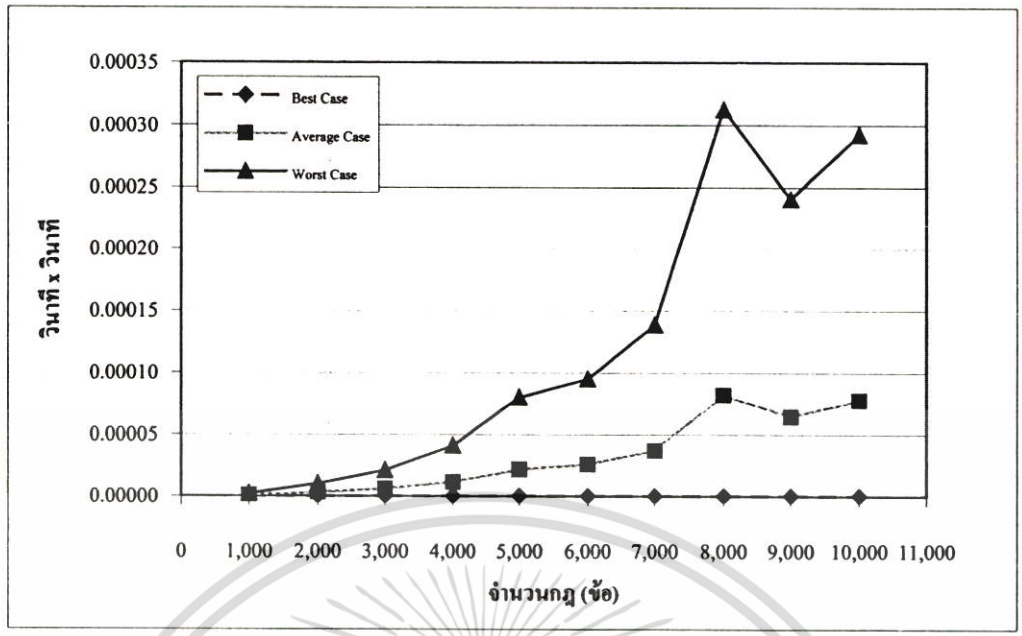
3. ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ



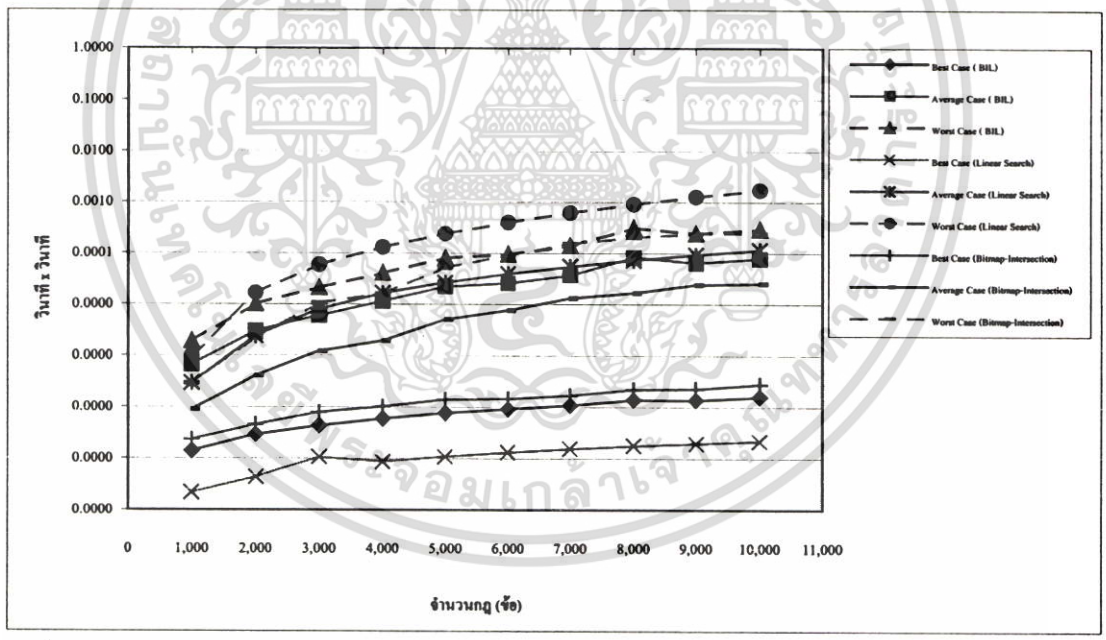
รูปที่ 4.68 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเวลาเฉลี่ยในการปรับปรุงกฎของโปรแกรม Linear Search ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.69 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเวลาเฉลี่ยในการปรับปรุงกฎของโปรแกรม Bitmap-intersection ที่จำนวนกฎ 1,000-10,000 ข้อ

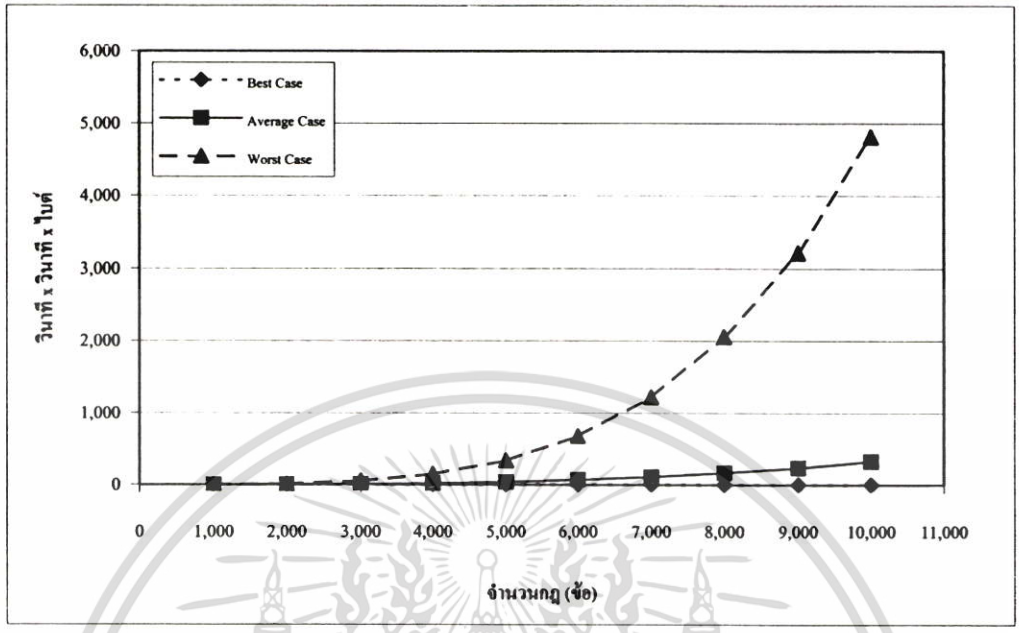


รูปที่ 4.70 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเวลาเฉลี่ยในการปรับปรุงกฎของโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ

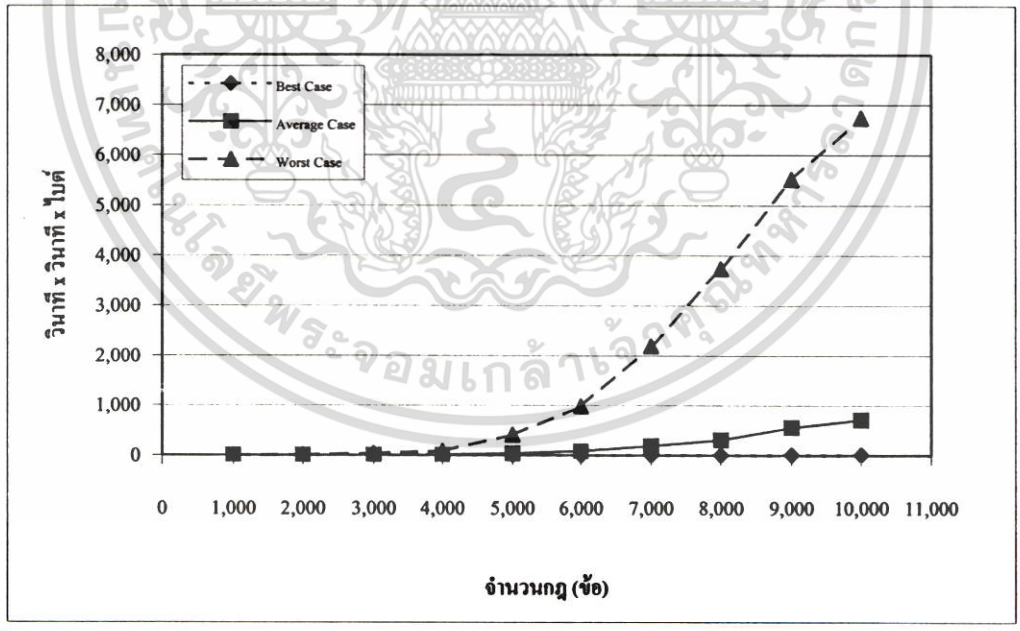


รูปที่ 4.71 เปรียบเทียบผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ตและเวลาเฉลี่ยในการปรับปรุงกฎของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ โดยใช้มาตราส่วนลอการิทึม

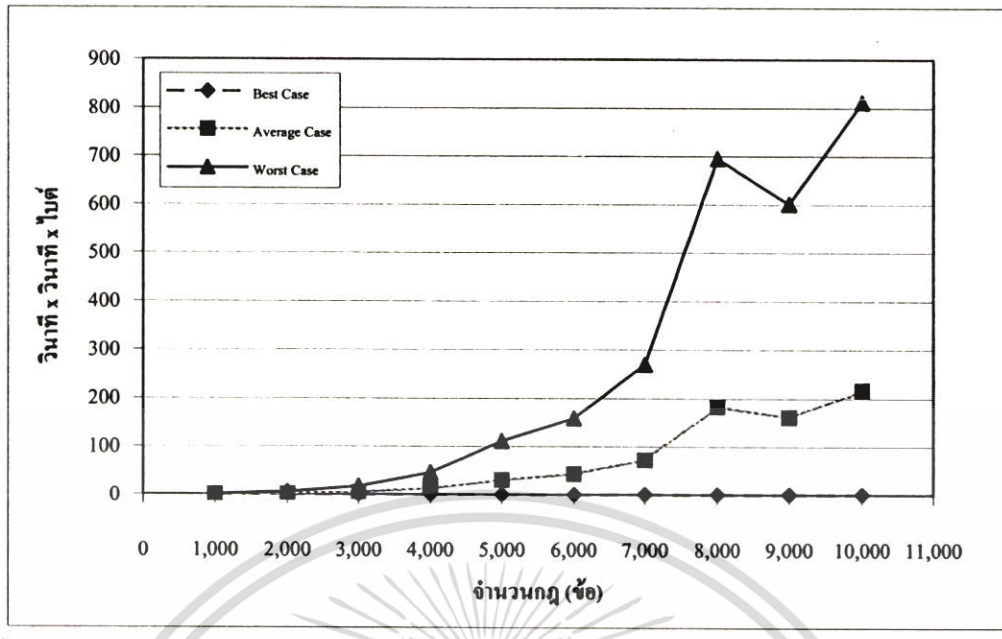
4. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล



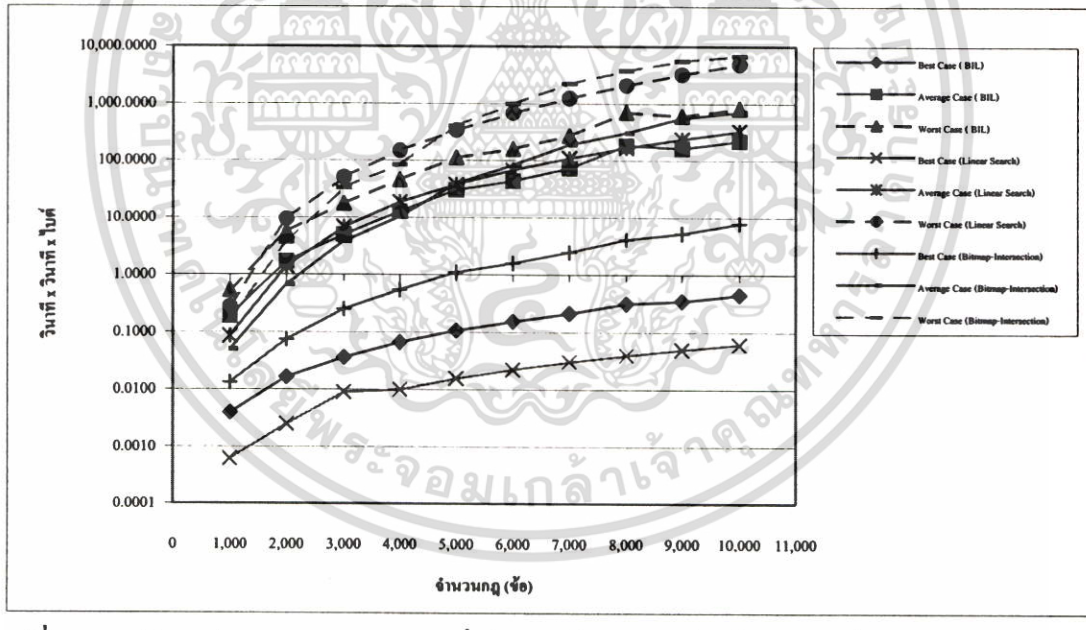
รูปที่ 4.72 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.73 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูลของโปรแกรม Bitmap-Intetsection ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.74 ผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูลของโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ



รูปที่ 4.75 เปรียบเทียบผลคูณของเวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต เวลาเฉลี่ยในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูลของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL ที่จำนวนกฎ 1,000-10,000 ข้อ โดยใช้มาตราส่วนลอการิทึม

จากกราฟของผลคูณในรูปที่ 4.60–4.75 สามารถเปรียบเทียบประสิทธิภาพของโปรแกรมต่างๆ ดังตารางที่ 4.44

ตารางที่ 4.44 เปรียบเทียบประสิทธิภาพของโปรแกรม Linear Search โปรแกรม Bitmap-intersection และโปรแกรม BIL เรียงลำดับประสิทธิภาพจากมากไปน้อย โดยพิจารณาจากแนวโน้มของเส้นกราฟแต่ละผลคูณ

กรณี	ผลคูณ			
	เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล	เวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล	เวลาในการค้นหาและเวลาในการปรับปรุงกฎ	เวลาในการค้นหา เวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล
Best Case	1.Linear Search 2.BIL 3.Bitmap-intersection	1.Linear Search 2.BIL 3.Bitmap-intersection	1.Linear Search 2.BIL 3.Bitmap-intersection	1.Linear Search 2.BIL 3.Bitmap-intersection
Average Case	1.Linear Search 2.BIL 3.Bitmap-intersection	1.BIL 2.Bitmap-intersection 3.Linear Search	1.Bitmap-intersection 2.BIL 3.Linear Search	1.BIL 2.Linear Search 3.Bitmap-intersection
Worst Case	1.Linear Search 2.BIL 3.Bitmap-intersection	1.BIL 2.Bitmap-intersection 3.Linear Search	1.Bitmap-intersection 2.BIL 3.Linear Search	1.BIL 2.Linear Search 3.Bitmap-intersection

จากตารางที่ 4.44 พบว่าในกรณี Worst Case โปรแกรม BIL จะมีประสิทธิภาพสูงกว่าโปรแกรม Linear Search และโปรแกรม Bitmap-intersection จากผลคูณจำนวน 2 ผลคูณ ได้แก่ 1. ผลคูณของเวลาในการค้นหาและเนื้อที่จัดเก็บข้อมูล 2. ผลคูณของเวลาในการค้นหา เวลาในการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล ในขณะที่โปรแกรม Linear Search มีประสิทธิภาพสูงกว่าโปรแกรมอื่นๆ ที่ผลคูณของเวลาในการปรับปรุงกฎและเนื้อที่จัดเก็บข้อมูล และโปรแกรม Bitmap-intersection มีประสิทธิภาพสูงกว่าโปรแกรมอื่นๆ ที่ผลคูณของเวลาในการค้นหาและเวลาในการปรับปรุงกฎ

จากการเปรียบเทียบประสิทธิภาพการทำงานกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection พบว่าโปรแกรม BIL มีข้อดี/ข้อด้อย ดังต่อไปนี้

1. โปรแกรม BIL จะมีแนวโน้มที่จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรม Bitmap-intersection เมื่อมีจำนวนกฎสูงขึ้น ในขณะที่โปรแกรม Linear Search จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรมอื่นๆ ในทุกกรณี

2. โปรแกรม BIL จะมีประสิทธิภาพในการใช้เนื้อที่จัดเก็บข้อมูลสูงกว่าโปรแกรม Linear Search และโปรแกรม Bitmap-intersection

3. โปรแกรม BIL จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรม Bitmap-intersection สำหรับกฎที่มีลักษณะเป็นช่วง (range) หรือเน็ตมาสก์ (net mask) แคบๆ

4. เมื่อพิจารณาประสิทธิภาพทั้ง 3 ด้าน พบว่าโปรแกรม BIL จะมีประสิทธิภาพในการทำงานโดยรวมดีที่สุด

ดังนั้นจึงสามารถสรุปได้ว่าการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) มีประสิทธิภาพในการทำงานที่ดี เมื่อเปรียบเทียบกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection โดยเป็นไปตามสมมติฐานดังที่กล่าวมาในงานวิจัยนี้

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

การจัดประเภทแพ็กเก็ตตามกฎเป็นเครื่องมือที่สำคัญสำหรับการกำหนดกระแสของแพ็กเก็ตบนเครือข่ายอินเตอร์เน็ต ซึ่งจะต้องปฏิบัติงานบนช่องทางการสื่อสารที่มีความเร็วสูงด้วยขนาดของกฎที่มีขนาดใหญ่ ดังนั้นแอ็พพลิเคชันการจัดประเภทแพ็กเก็ตจึงต้องมีวิธีการค้นหาที่มีประสิทธิภาพสูงสุด เพื่อไม่ให้เกิดผลกระทบต่อการจราจรบนเครือข่าย โดยยังคงมีประสิทธิภาพในการปรับปรุงกฎ และการใช้เนื้อที่จัดเก็บข้อมูลที่ดีด้วย

งานวิจัยนี้ได้นำเสนอวิธีการของการจัดประเภทแพ็กเก็ตตามกฎที่มีประสิทธิภาพสูง โดยจัดโครงสร้างของกฎให้อยู่ในรูปของตาราง Bitmap Intersection Lookup (BIL) สำหรับการค้นหาจะแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block ขนาดเล็กๆ นำค่าของแต่ละ Block ไป Lookup ในตาราง BIL table และนำผลลัพธ์ที่ได้ทำการ Intersection ด้วยตรรก AND ดังนั้นโปรแกรมจึงมีวิธีการทำงานและโครงสร้างของกฎที่ง่ายไม่ซับซ้อน ทำให้มีความเร็วในการค้นหาที่สูง มีเวลาในการปรับปรุงกฎที่ดี และใช้เนื้อที่จัดเก็บข้อมูลต่ำ จากการวิเคราะห์อัลกอริทึมการค้นหา อัลกอริทึมการปรับปรุงกฎ และเนื้อที่จัดเก็บข้อมูล สามารถแสดงสมการทั่วไปในแต่ละด้านดังต่อไปนี้

1. เวลาในการค้นหาต่อแพ็กเก็ต

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la} \right)$$

2. เวลาในการเพิ่มกฎต่อข้อ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right]$$

3. เนื้อที่จัดเก็บข้อมูล

$$storage = \frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot \frac{N}{8} \text{ bytes}$$

ในการออกแบบการทดลองเพื่อทดสอบแนวคิดดังกล่าว ได้แบ่งการทดลองออกเป็น 2 ส่วน โดยการทดลองส่วนที่ 1 เป็นการแบ่ง Prefix เป็น Block เล็กๆ เพื่อหาค่าที่ทำให้โปรแกรม BIL มีประสิทธิภาพในการทำงานดีที่สุด และการทดลองส่วนที่ 2 เป็นการเปรียบเทียบประสิทธิภาพการทำงานของโปรแกรม Linear Seasrch โปรแกรม Bitmap-intersection และโปรแกรม BIL โดยในส่วนที่ 1 จะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) และ Source MAC Address (ขนาด 48 บิต) ส่วนที่ 2 จะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต) โดยผู้วิจัยเลือกที่จะสร้างฐานข้อมูลกฎ และฐานข้อมูลแพ็กเก็ตสำหรับการจัดประเภทขึ้นมาเอง เพื่อให้เหมาะสมกับจุดประสงค์ของการทดลองแต่ละการทดลอง

จากการศึกษาประสิทธิภาพการทำงานของโปรแกรม BIL สามารถสรุปรูปแบบการแบ่ง Prefix ได้ดังต่อไปนี้

1. ควรแบ่ง Prefix เป็น Block ขนาดเท่าๆ กัน

2. การแบ่ง Prefix เป็น Block ขนาดเล็กจะมีผลดีในด้านการปรับปรุงกฎ สำหรับกฎที่มีช่วง (range) หรือ เน็ตมาสค์ (net mask) กว้าง และด้านเนื้อที่จัดเก็บข้อมูล

3. การแบ่ง Prefix เป็น Block ขนาดใหญ่จะมีผลดีในด้านการค้นหา และการปรับปรุงกฎ สำหรับกฎที่มีลักษณะไม่เป็นช่วง

จากการเปรียบเทียบประสิทธิภาพการทำงานกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection พบว่าโปรแกรม BIL มีข้อดี/ข้อด้อย ดังต่อไปนี้

1. โปรแกรม BIL จะมีแนวโน้มที่จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรม Bitmap-intersection เมื่อมีจำนวนกฎสูงขึ้น ในขณะที่โปรแกรม Linear Search จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรมอื่นๆ ในทุกกรณี

2. โปรแกรม BIL จะมีประสิทธิภาพในการใช้เนื้อที่จัดเก็บข้อมูลสูงกว่าโปรแกรม Linear Search และโปรแกรม Bitmap-intersection

3. โปรแกรม BIL จะมีความเร็วในการปรับปรุงกฎสูงกว่าโปรแกรม Bitmap-intersection สำหรับกฎที่มีลักษณะเป็นช่วง (range) หรือเน็ตมาสค์ (net mask) แคบๆ

4. เมื่อพิจารณาประสิทธิภาพทั้ง 3 ด้าน พบว่าโปรแกรม BIL จะมีประสิทธิภาพในการทำงานโดยรวมดีที่สุด

โดยจากการศึกษา แต่ละวิธีจะมี Complexity ดังตารางที่ 5.1

ตารางที่ 5.1 เปรียบเทียบประสิทธิภาพการค้นหา การปรับปรุงกฎ และการจัดเก็บข้อมูล ของวิธีการ Linear Search วิธีการ Bitmap-intersection และวิธีการ BIL

Algorithm	Search Speed Complexity	Update Speed Complexity	Storage Complexity
Linear Search	N	-	N
Software Bitmap-intersection	$D \cdot \left(\log N + \frac{N}{WORD} \right)$	-	$D \cdot N^2$
Hardware Bitmap-intersection	$D \cdot \log N + \frac{N}{WORD}$	-	$D \cdot N^2$
Software BIL	$\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}$	$\left(2 \cdot W - 2\right) \cdot \frac{D \cdot W}{PMX} \cdot 2^{PMX}$	$\frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot N$
Hardware BIL	1	$\left(2 \cdot W - 2\right) \cdot \frac{D \cdot W}{PMX} \cdot 2^{PMX}$	$\frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot N$

โดยจากตารางที่ 5.1 พบว่าวิธีการ BIL จะมี Search Speed Complexity บนฮาร์ดแวร์เท่ากับ O(1) ซึ่งมีประสิทธิภาพสูงว่าวิธีการอื่นๆ แต่มี Search Speed Complexity บนซอฟท์แวร์

เท่ากับ $O\left(\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}\right)$ ซึ่งมีประสิทธิภาพต่ำกว่าวิธีการ Bitmap-intersection ทั้งบนซอฟท์แวร์และบนฮาร์ดแวร์ โดยวิธีการ BIL จะมี Update Speed Complexity ทั้งบนซอฟท์แวร์และฮาร์ดแวร์เท่ากับ $O\left((2 \cdot W - 2) \cdot \frac{D \cdot W}{PMX} \cdot 2^{PMX}\right)$ เมื่อพิจารณาถึงการใช้เนื้อที่จัดเก็บข้อมูลพบว่าวิธีการ Linear Search มี Storage Complexity เท่ากับ O(N) ซึ่งมีประสิทธิภาพสูงที่สุด และวิธีการ Bitmap-intersection จะมี Storage Complexity เท่ากับ $O(D \cdot N^2)$ ซึ่งมีประสิทธิภาพต่ำว่าวิธีการอื่นๆ อย่างไรก็ตามเมื่อทำการแบ่งขนาด Prefix ให้เหมาะสม โปรแกรม BIL ก็จะมีประสิทธิภาพในการใช้เนื้อที่จัดเก็บข้อมูลสูงกว่าโปรแกรม Linear Search ดังผลการทดลองในบทก่อนหน้า

ดังนั้นจึงสามารถสรุปได้ว่าการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) มีประสิทธิภาพในการทำงานที่ดี เมื่อเปรียบเทียบกับโปรแกรม Linear Search และโปรแกรม Bitmap-intersection โดยเป็นไปตามสมมติฐานดังที่กล่าวมาในงานวิจัยนี้

จากแนวคิดของโปรแกรม BIL สามารถนำไปประยุกต์ใช้กับแอพพลิเคชัน เช่น ไฟร์วอล และอินเตอร์เน็ตเร้าเตอร์ที่ให้บริการต่างๆ เช่น Quality of Services, Virtual Private Network (VPN) Services, Distribute Firewall และ IP Security Gateways เป็นต้น นอกจากนั้นแนวคิดในงานวิจัยดังกล่าวสามารถนำไปพัฒนาเป็นแอพพลิเคชันด้วยฮาร์ดแวร์ ซึ่งจะมีประสิทธิภาพในการค้นหาเพิ่มขึ้นโดยการค้นหาแบบขนานบนฮาร์ดแวร์

สำหรับแนวทางการทำวิจัยต่อไปในอนาคตจะทำการทดลองเพื่อเปรียบเทียบประสิทธิภาพในด้านต่างๆ บนอินเตอร์เน็ตแอพพลิเคชันที่เป็นฮาร์ดแวร์ เช่น ไฟร์วอล หรือเร้าเตอร์ต่อไป

เอกสารอ้างอิง

[1] สุรศักดิ์ สงวนพงษ์. 2543. สถาปัตยกรรมและโปรโตคอลทีซีพี/ไอพี. กรุงเทพฯ : ซีเอ็ดยูเคชั่น.

[2] A. Feldman and S. Muthukrishnan. 2000. "Tradeoffs for Packet Classification." **In Proc. INFOCOM.** (1) : 397-413.

[3] F. Baboescu, G. Varghese. 2001. "Scalable Packet Classification." **In Proc. ACM Sigcomm '01.**

[4] J. van Lunteren and A.P.J. Engbersen. 2002. "Multi-field Packet Classification Using Ternary CAM." **IEE Electronics Letters.** (38) : 1.

[5] M.L. Bailey, B.Gopal, M.Pagels, L.L.Peterson, and P.Sarker. 1994. "PATHFINDER: A Pattern-Based Packet Classifier." **In Proc. of the First Symposium on Operating Systems Design and Implement.**

[6] P. Gupta and N. McKeown. 2001. "Algorithms for Packet Classification." **IEEE Netw.**

[7] P. Gupta and N. McKeown. 1999. "Packet Classification on Multiple Fields." **In Proc. ACM Sigcomm '99.**

[8] P. Tsuchiya. "A search algorithm for table entries with non-contiguous wildcarding." **unpublished report, Bellcore.**

[9] T. V. Lakshman and D. Stiliadis. 1998. "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching". **In Proc. ACM Sigcomm '98.**

[10] V. Srinivasan, S. Suri, and G. Varghese. 1999. "Packet Classification using Tuple Space Search." **In Proc. ACM Sigcomm '99.**

[11] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. 1998. "Fast and Scalable Layer four Switching." **In Proc. ACM Sigcomm '98.**

ภาคผนวก

ภาคผนวก ก.

คำสั่งการใช้งานโปรแกรม BIL โปรแกรม Linear Search และ โปรแกรม Bitmap-intersection

1. คำสั่งการเพิ่มกฎแบบ Insert

   -I|--insert **<Ruleid>** *-a|--acction* **<Action>**

   [*-p|--protocol* **<Protocol Number|Protocol Name>**]

   [*-s|--source* **<IP Address|Host Name>**]

   [*-d|--destination* **<IP Address|Host Name>**]

   [*--sport* **<Port Number|Port Name>**]

   [*--dport* **<Port Number|Port Name>**]

   [*--smac* **<MAC Address>**]

   [*--dmac* **<MAC Address>**]

2. คำสั่งการเพิ่มกฎแบบ Append

   -A|--append -a|--acction **<Action>**

   [*-p|--protocol* **<Protocol Number|Protocol Name>**]

   [*-s|--source* **<IP Address|Host Name>**]

   [*-d|--destination* **<IP Address|Host Name>**]

   [*--sport* **<Port Number|Port Name>**]

   [*--dport* **<Port Number|Port Name>**]

   [*--smac* **<MAC Address>**]

   [*--dmac* **<MAC Address>**]

3. คำสั่งลบกฎ

   -D|--delete **<Ruleid>**

4. คำสั่งการค้นหา

   -C|--classify **<File Name>**

5. คำสั่งแสดงรายการกฎ

   -L|--list

6. คำสั่งแสดงเวอร์ชันโปรแกรม

-V|--version -

7. คำสั่งแสดงวิธีการใช้งานคำสั่งโปรแกรม

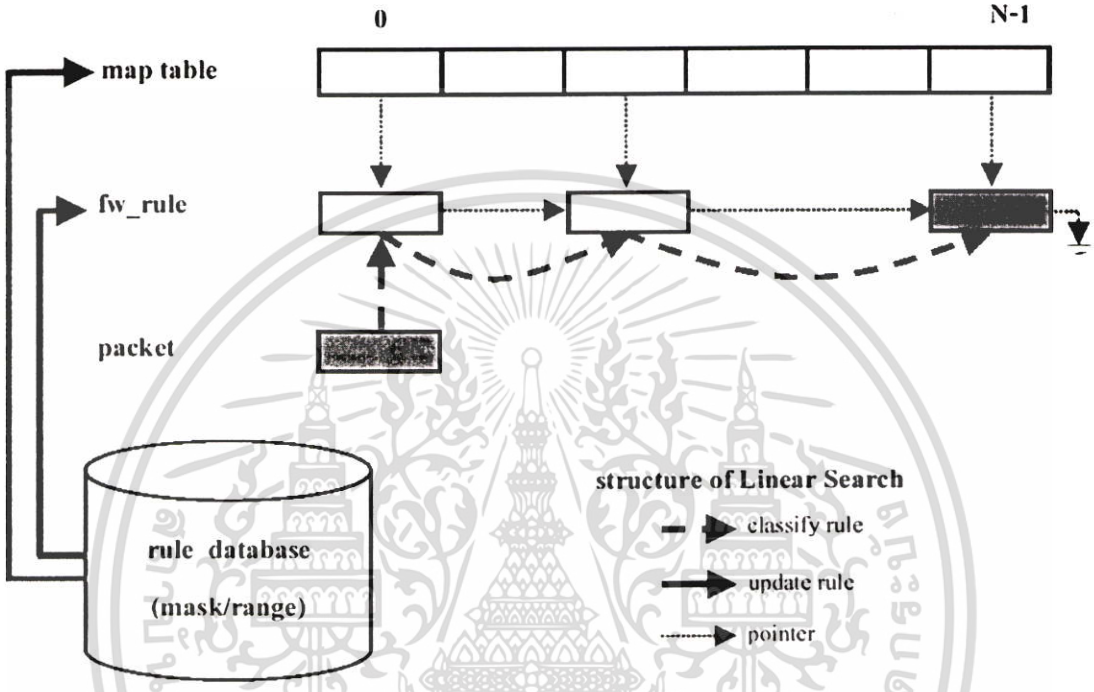
-H|--help

8. ออกจากโปรแกรม

-Q|--quit

ภาคผนวก ข.

สถาปัตยกรรมของโปรแกรม Linear Search และโปรแกรม Bitmap-intersection



รูปที่ ข.1 สถาปัตยกรรมของโปรแกรม Linear Search

สถาปัตยกรรมของโปรแกรม Linear Search ประกอบด้วยส่วนต่างๆ ดังต่อไปนี้

1. Map Table

เป็น Array 1 มิติ มีชนิดของข้อมูลเป็น struct fw_rule* สำหรับเก็บเลขที่อยู่ของ fw_rule ซึ่งใช้ในการปรับปรุงกฎ โดยประกาศเป็น

```
struct fw_rule* map_table[Nmax];
```

กำหนดให้

Nmax คือ จำนวนกฎสูงสุด

2. fw_rule

เป็นข้อมูลชนิด struct สำหรับเก็บค่าของกฎที่อ่านมาจากฐานข้อมูล โดยประกาศเป็น

```
struct ranges {
    u_int32_t lower;
    u_int32_t upper;
};
```

```
struct fw_rule {
        long long ruleid;
        int action;
        char command[MAX_LEN];
        struct ranges sip;
        struct fw_rule *next;
};
```

กำหนดให้

MAX_LEN คือ จำนวนตัวอักษรสูงสุด

3. Rule Database

เป็นฐานข้อมูลกฎ โดยมีรูปแบบคำสั่งดังภาคผนวก ก.

```
1.  อ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์
2.  current = first
3.  action = DEFAULT_RULE
4.  While ((current !=NULL) && (found == FALSE)){
5.      if (current->filed1= field1) {
6.          action = current->action
7.          found = TRUE
8.      }
9.      current = current->next;
10. }
11. ส่งค่า Action กลับ
```

รูปที่ ข 2 อัลกอริทึมการค้นหาของโปรแกรม Linear Search

```
1.  รับคำสั่งของกฎจากฐานข้อมูลกฎหรือจากผู้ดูแลระบบ เก็บลงในบัฟเฟอร์
2.  แปลความหมายคำสั่งของกฎที่อ่านมา และเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำ
3.  if (search_rule(ruleid,&pre) == FALSE) {
4.          newnode = (struct fw_rule *)malloc(sizeof(struct fw_rule))
5.          newnode->ruleid = ruleid
6.          newnode->action = action
7.          newnode->field_i = field_i
8.          map_table[ruleid] = newnode
9.          if (first == NULL) {
10.                 first = newnode
11.                 newnode->next = NULL
12.         } else {
13.                 if (pre == NULL) {
14.                         newnode->next = first
15.                         first = newnode
16.                 } else {
17.                         newnode->next = pre->next
18.                         pre->next = newnode
19.                 }
20.         }
21. }
```

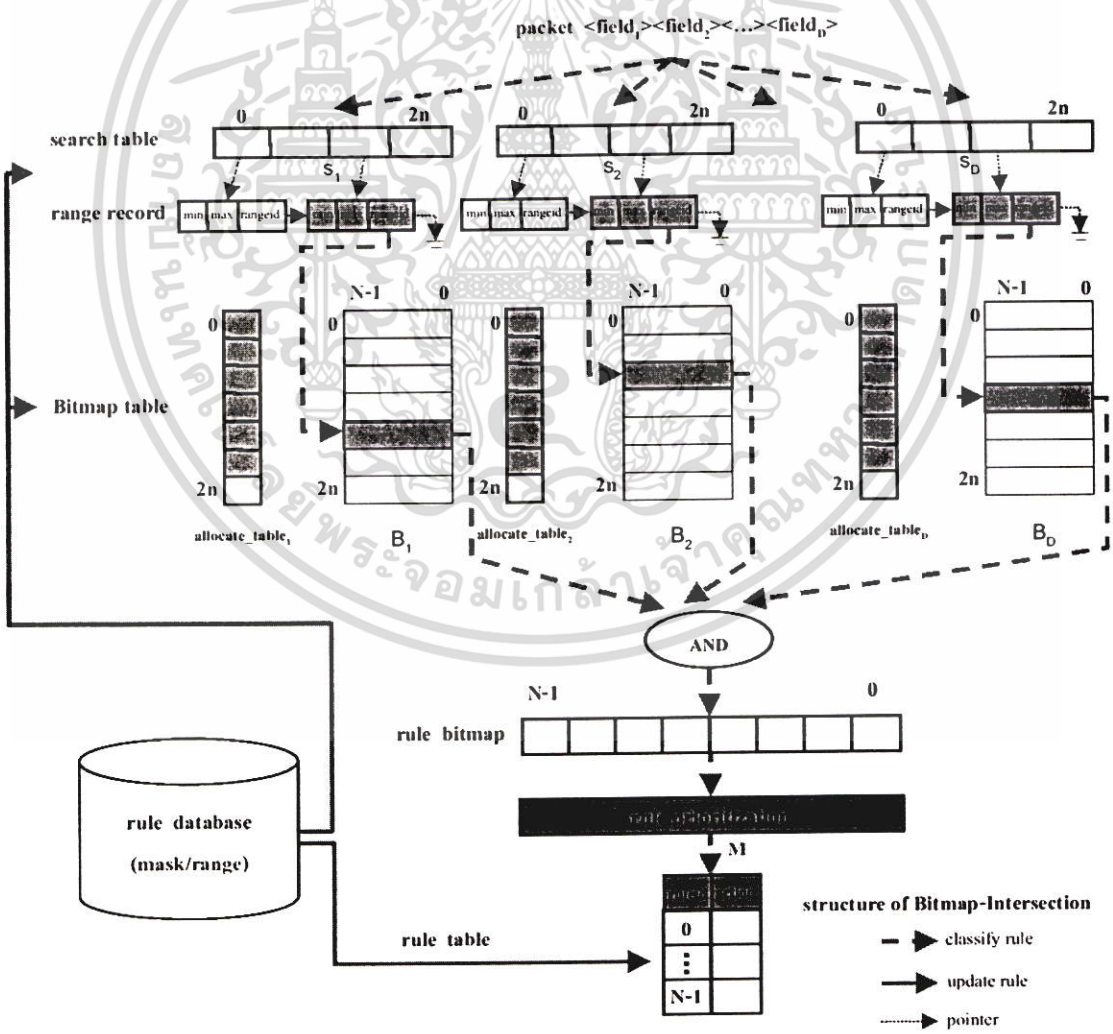
รูปที่ ข.3 อัลกอริทึมการเพิ่มกฎของโปรแกรม Linear Search

```
1.  รับคำสั่งลบกฎจากไฟล์สคริปหรือผู้ดูแลระบบ เก็บลงในบัฟเฟอร์
2.  แปลความหมายคำสั่ง
3.  if (search_rule(ruleid,&pre)== TRUE) {
4.          map_table[ruleid] = NULL
5.          if (pre == NULL) {
6.                  delnode = first
7.                  first = first->next
8.          } else {
9.                  delnode = pre->next
10.                 pre->next = delnode->next
11.         }
12.         free(delnode)
13. }
```

รูปที่ ข.4 อัลกอริทึมการลบกฎของโปรแกรม Linear Search

```
1.  int search_rule(long long ruleid, struct fw_rule **pre)
2.  {
3.          found = FALSE
4.          *pre = NULL
5.          if (map_table[ruleid] != NULL) {
6.                  found = TRUE
7.          }
8.          r = ruleid -1
9.          while ((map_table[r] == NULL) && (r > 0)){
10.                 r –
11.         }
12.         *pre = map_table[r]
13.         return found
14. }
```

รูปที่ ข.5 ฟังก์ชัน search_rule ของโปรแกรม Linear Search



รูปที่ ข.6 สถาปัตยกรรมของโปรแกรม Bitmap-intersection

D คือ จำนวนเฮดเดอร์ฟิลด์

4. Bitmap Table

เป็น Array 3 มิติ มีชนิดข้อมูลเป็น u_int32_t สำหรับเก็บ Bitmap โดยประกาศเป็น

u_int32_t bitmap_table[D][R][B];

กำหนดให้

D คือ จำนวนเฮดเดอร์ฟิลด์

R คือ จำนวนช่วงสูงสุดเท่ากับ 2•N + 1 โดยที่ N คือ จำนวนกฎทั้งหมด

B คือ จำนวน WORD ของ Bitmap

5. Rule Bitmap

เป็นผลลัพธ์ที่ได้จากการ Intersection Bitvector ด้วยตรรก AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์

6. Rule Database

เป็นฐานข้อมูลกฎ โดยมีคำสั่งดังภาคผนวก ก

7. Rule Table

เป็น Array 1 มิติ มีชนิดข้อมูลเป็น struct rule สำหรับเก็บค่าแอ็คชันของกฎ และคำสั่งการเพิ่มกฎที่อ่านมาจากฐานข้อมูล โดยประกาศเป็น

```
struct rule {
    long long int ruleid;
    int action;
    char command[MAX_LEN];
};
struct rule rule_table[Nmax];
```

กำหนดให้

MAX_LEN คือ จำนวนตัวอักษรสูงสุด

Nmax คือ จำนวนกฎสูงสุด

```
1.    อ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์
2.    for (i=0; i<D; i++ ) {
3.          id = binary_search(i,field_i)
4.          if (id != (-1)) {
5.                rangeid[i] = id
6.          } else {
7.                ส่งค่า Default Action กลับ
8.          }
9.    }
10.   for (i=0; i<N/WORD; i++ ) {
11.         for (j=0; j< D; j++ ) {
12.               นำ Bitmap ในตำแหน่ง WORD ที่ i จาก Bitmap table j และ
                  แอดเดรสที่ rangeid[j] มา AND กัน
13.         }
14.         if (ผลลัพธ์จากการ AND != 0) {
15.               คำนวณหมายเลขของกฎจากตำแหน่งบิตที่มีค่าเป็น 1 บิตแรกใน Bitmap และ
                  ส่งค่า Action จาก rule table กลับ
16.         }
17.   }
18.   ส่งค่า Default Action กลับ
```

รูปที่ ข.7 อัลกอริทึมการค้นหาของโปรแกรม Bitmap-intersection

```
1.  รับคำสั่งของกฎจากฐานข้อมูลกฎหรือจากผู้ดูแลระบบ เก็บลงในบัฟเฟอร์
2.  แปลความหมายคำสั่งของกฎที่อ่านมา และเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำ
3.  เก็บค่า ruleid, Action และคำสั่งลงใน rule table
4.  for (i=0;i<D; i++) {
5.      แบ่งกฎสำหรับ field_i ออกเป็นช่วง
6.      for (แต่ละช่วงที่ไม่ตรงกับช่วงเดิมใน range record) {
7.          rangeid = search_range(i)
8.          allocate_table[i].range[rangeid] = TRUE
9.          newnode = (struct range_record *)malloc(sizeof(struct range_record))
10.         newnode->min = lowerrange
11.         newnode->max = upperrange
12.         newnode->rangeid = rangeid
13.         if (ช่วงใหม่เป็นซับเซ็ตของช่วงเก่า) {
14.             for (j = 0; j <= Rword ; j++ ) {
15.                 bitmap_table[i][rangeid][j] = bitmap_table[i][old_rangeid][j]
16.             }
17.         }
18.         setbitmap(i,rangeid,ruleid)
19.         เพิ่ม newnode ใน range record สำหรับ field_i
20.     }
21.     for (แต่ละช่วงที่ตรงกับช่วงเดิมใน range record) {
22.         setbitmap(i,old_rangeid,ruleid)
23.     }
24. }
```

รูปที่ ข.8 อัลกอริทึมการเพิ่มกฎของโปรแกรม Bitmap-intersection

ภาคผนวก ค.

การทดลองวัดเวลาในการค้นหาของโปรแกรม BIL กับฐานข้อมูลกฎจำนวน 1,000-32,000 ข้อ

จากการวิเคราะห์ประสิทธิภาพเบื้องต้นของอัลกอริทึมตามที่เคยกล่าวในบทที่ 3 ในกรณี Worst Case โปรแกรม BIL จะมี Search Speed Complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}\right)$ ซึ่งจะเห็นว่าเมื่อ N มีค่าสูงขึ้น ความเร็วในการค้นหาจะมีค่าสูงขึ้นในลักษณะเชิงเส้น และจากการทดลองที่ 4.2.3 พบว่าเมื่อจำนวนกฎสูงขึ้นโปรแกรมจะมีแนวโน้มในการใช้เวลาการค้นหาสูงขึ้นเช่นเดียวกัน แต่เส้นกราฟจะมีความแกว่งอย่างเห็นได้ชัด โดยเฉพาะที่จำนวนกฎเท่ากับ 9,000-10,000 ข้อ โปรแกรมจะมีเวลาในการค้นหาต่ำกว่าที่จำนวนกฎเท่ากับ 8,000 ข้อ สำหรับ Prefix ทุกๆ รูปแบบ ดังนั้นเพื่อพิสูจน์ว่าที่ N ใดๆ เส้นกราฟจะมีรูปแบบความสัมพันธ์ของเวลาในการค้นหาต่อแพ็กเก็ตเป็นแบบเชิงเส้นเสมอ ผู้วิจัยจึงทำการทดลองโดยเพิ่มจำนวนกฎสูงสุดเป็น 32,000 ข้อ ดังต่อไปนี้

วัตถุประสงค์การทดลอง เพื่อศึกษารูปแบบความสัมพันธ์ของเวลาในการค้นหา กับฐานข้อมูลกฎจำนวนกฎ 1,000-32,000 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดการแบ่ง Prefix เป็น 8+8+8+8 บิต
3. กำหนดจำนวนกฎเท่ากับ 1,000, 1,024, 1,536, 2,000, 2,048, 2,560, 3,000, 3,072, 3,584, 4,000, 4,096, 4,608, 5,000, 5,120, 5,632, 6,000, 6,144, 6,656, 7,000, 7,168, 7,680, 8,000, 8,192, 8,704, 9,000, 9,216, 9,728, 10,000, 11,000, 12,000, 13,000, 14,000, 15,000, 16,000, 17,000, 18,000, 19,000, 20,000, 21,000, 22,000, 23,000, 24,000, 25,000, 26,000, 27,000, 28,000, 29,000, 30,000, 31,000, 32,000 ข้อ และสร้างฐานข้อมูลกฎสำหรับวัดเวลาการค้นหา
4. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต
5. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case
6. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case
7. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case

ตารางที่ ค.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต) จำนวนกฎ 1,000-32,000 ข้อ

กรณี	จำนวนกฎ (ข้อ)									
	1,000	1,024	1,536	2,000	2,048	2,560	3,000	3,072	3,584	4,000
Best Case (วินาที)	0.000000685552000	0.000000533303000	0.000000546460000	0.000000541936000	0.000000534179000	0.000000538804000	0.000000562462000	0.000000546850000	0.000000544226000	0.000000534569000
Average Case (วินาที)	0.000002660008000	0.000001377252000	0.000001927899000	0.000002092505000	0.000002109474000	0.000003210319000	0.000003471290000	0.000003182661000	0.000003620554000	0.000003759868000
Worst Case (วินาที)	0.000004304974000	0.000002115682000	0.000003113409000	0.000003221227000	0.000003420011000	0.000005477965000	0.000005929252000	0.000005351579000	0.000006129224000	0.000006602022000

ตารางที่ ค.1 (ต่อ)

กรณี	จำนวนกฎ (ข้อ)									
	4,096	4,608	5,000	5,120	5,632	6,000	6,144	6,656	7,000	7,168
Best Case (วินาที)	0.000000530943000	0.000000541339000	0.000000548364000	0.000000542329000	0.000000575366000	0.000000547956000	0.000000538623000	0.000000547462000	0.000000554000000	0.000000543949000
Average Case (วินาที)	0.000003828268000	0.000004411202000	0.000005823716000	0.000004859363000	0.000006404395000	0.000006773546000	0.000005565813000	0.000007461326000	0.000009803634000	0.000006977223000
Worst Case (วินาที)	0.000006607133000	0.000007648644000	0.000010144678000	0.000008386949000	0.000011499545000	0.000012175796000	0.000009938101000	0.000013484317000	0.000018258045000	0.000012592112000

ตารางที่ ค.1 (ต่อ)

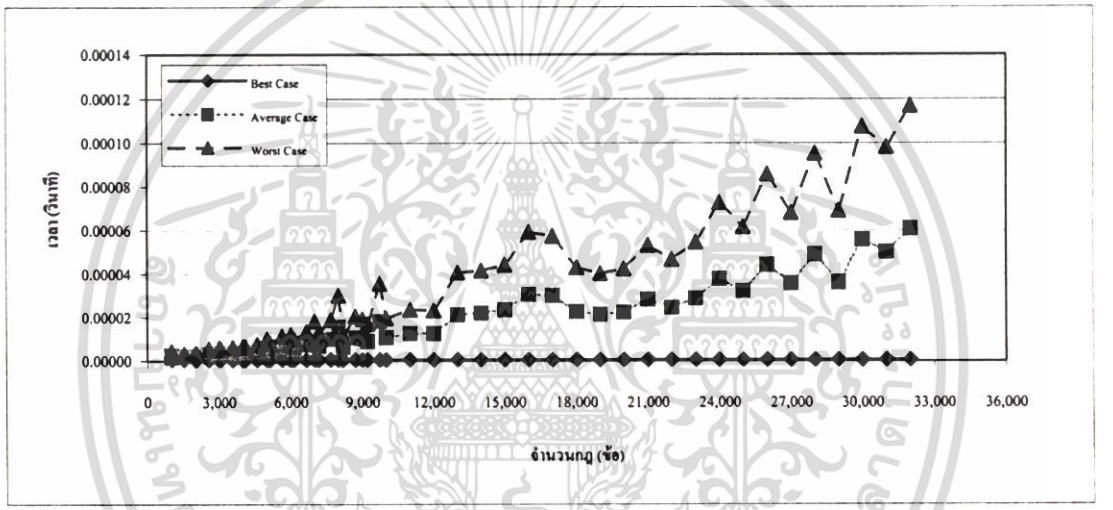
กรณี	จำนวนกฎ (ข้อ)									
	7,680	8,000	8,192	8,704	9,000	9,216	9,728	10,000	11,000	12,000
Best Case (วินาที)	0.000000562851000	0.000000613154000	0.000000544827000	0.000000547210000	0.000000553210000	0.000000545652000	0.000000581986000	0.000000547367000	0.000000551854000	0.000000550369000
Average Case (วินาที)	0.000010034537000	0.000015728090000	0.000006215779000	0.000010769730000	0.000010482461000	0.000008935993000	0.000018245745000	0.000010774649000	0.000012694648000	0.000012634432000
Worst Case (วินาที)	0.000018969285000	0.000030280931000	0.000011130737000	0.000020567216000	0.000019302492000	0.000017127415000	0.000035720951000	0.000019659012000	0.000023721602000	0.000023288746000

ตารางที่ ค.1 (ต่อ)

กรณี	จำนวนกฎ (ข้อ)									
	13,000	14,000	15,000	16,000	17,000	18,000	19,000	20,000	21,000	22,000
Best Case (วินาที)	0.000000566808000	0.000000559544000	0.000000580949000	0.000000613847000	0.000000591591000	0.000000557446000	0.000000552578000	0.000000550944000	0.000000550512000	0.000000551052000
Average Case (วินาที)	0.000021358113000	0.000022044943000	0.000023728664000	0.000030683942000	0.000030199267000	0.000022814005000	0.000021397859000	0.000022402255000	0.000028409594000	0.000024663752000
Worst Case (วินาที)	0.000040827261000	0.000041691734000	0.000044322188000	0.000059287423000	0.000057448998000	0.000043160772000	0.000040395364000	0.000042482368000	0.000053268606000	0.000046863355000

ตารางที่ ค.1 (ต่อ)

กรณี	จำนวนกฎ (ข้อ)									
	23,000	24,000	25,000	26,000	27,000	28,000	29,000	30,000	31,000	32,000
Best Case (วินาที)	0.000000559215000	0.000000561046000	0.000000556918000	0.000000596239000	0.000000589488000	0.000000595788000	0.000000560290000	0.000000574800000	0.000000567075000	0.000000612384000
Average Case (วินาที)	0.000028877588000	0.000037863782000	0.000032225234000	0.000044365253000	0.000035771876000	0.000049136858000	0.000036222822000	0.000055885475000	0.000050244026000	0.000060873755000
Worst Case (วินาที)	0.000054786579000	0.000072807712000	0.000061739885000	0.000085810970000	0.000068161563000	0.000095282978000	0.000069177316000	0.000107584688000	0.000098194331000	0.000117002479000



รูปที่ ค.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต) จำนวนกฎ 1,000-32,000 ข้อ

วิเคราะห์ผลการทดลอง

จากตารางที่ ค.1 และรูปที่ ค.1 พบว่าเมื่อจำนวนกฎสูงขึ้นโปรแกรมจะมีแนวโน้มในการใช้เวลาการค้นหาสูงขึ้น โดยที่จำนวนกฎเท่ากับ 1,000-32,000 ข้อ เส้นกราฟจะมีรูปแบบความสัมพันธ์ของเวลาในการค้นหาต่อแพ็กเก็ตเป็นแบบเชิงเส้น ซึ่งสามารถแสดงสมการความสัมพันธ์ระหว่างจำนวนกฎ และเวลาในการค้นหา คือ t_{search} = 0.000000002848N, R^2 = 0.908279593734 ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

ภาคผนวก ง.

การทดลองวัดเวลาในการค้นหาของโปรแกรม BIL โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify กับฐานข้อมูลกฎจำนวน 1,000-10,000 ข้อ

จากการทดลองดังแสดงในภาคผนวก ค. แม้ว่ารูปแบบความสัมพันธ์ของเวลาในการค้นหาจะเป็นแบบเชิงเส้น โดยเมื่อจำนวนกฎสูงขึ้นโปรแกรมจะมีแนวโน้มในการใช้เวลาการค้นหาสูงขึ้น อย่างไรก็ตามจากรูปที่ ค.1 พบว่าที่จำนวนกฎเท่ากับ 1,000-32,000 ข้อ เส้นกราฟยังมีความแกว่งอย่างเห็นได้ชัด ซึ่งคาดว่าสาเหตุประการหนึ่งจะเกิดจากการประกาศตัวแปรเป็นชนิด register ในฟังก์ชัน classify ของโปรแกรม BIL ดังนั้นเพื่อทดสอบสมมติฐานดังกล่าว ผู้วิจัยจึงทำการทดลองเปรียบเทียบเวลาในการค้นหาของโปรแกรม BIL โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify ดังต่อไปนี้

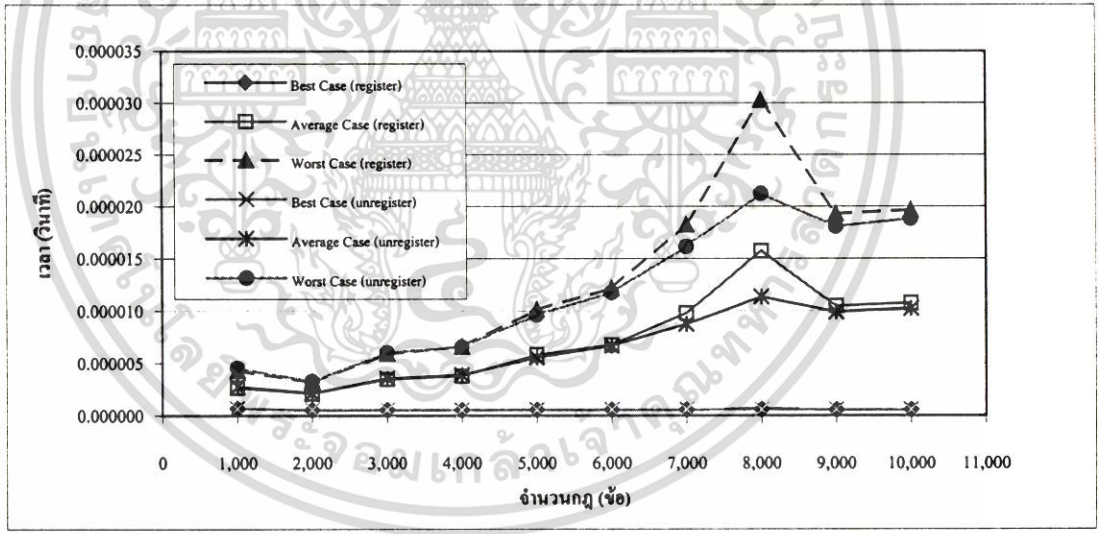
วัตถุประสงค์การทดลอง เพื่อเปรียบเทียบเวลาในการค้นหา โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify กับฐานข้อมูลกฎจำนวนกฎ 1,000-10,000 ข้อ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. กำหนดการแบ่ง Prefix เป็น 8+8+8+8 บิต
3. กำหนดจำนวนกฎเท่ากับ 1,000, 2,000, 3,000, ..., 10,000 ข้อ และสร้างฐานข้อมูลกฎสำหรับวัดเวลาการค้นหา
4. กำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต
5. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Best Case โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify
6. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Average Case โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify
7. สร้างฐานข้อมูลแพ็กเก็ตสำหรับวัดเวลาในการค้นหาในกรณี Worst Case โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify

ตารางที่ ง.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต) จำนวนกฎ 1,000-10,000 ข้อ โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify

กรณี	จำนวนกฎ (ข้อ)									
	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000
Best Case (register)	0.000000685552000	0.000000541936000	0.000000562462000	0.000000534569000	0.000000548364000	0.000000547956000	0.000000554000000	0.000000613154000	0.000000553210000	0.000000547367000
Average Case (register)	0.000002660008000	0.000002092505000	0.000003471290000	0.000003759868000	0.000005823716000	0.000006773546000	0.000009803634000	0.000015728090000	0.000010482461000	0.000010774649000
Worst Case (register)	0.000004304974000	0.000003221227000	0.000005929252000	0.000006602022000	0.000010144678000	0.000012175796000	0.000018258045000	0.000030280931000	0.000019302492000	0.000019659012000
Best Case (unregister)	0.000000674285200	0.000000554632400	0.000000565476400	0.000000542478800	0.000000568270200	0.000000568083200	0.000000582647200	0.000000575775600	0.000000588082000	0.000000570306600
Average Case (unregister)	0.000002748194000	0.000002137815400	0.000003545821200	0.000003882865600	0.000005555344400	0.000006689870200	0.000008762070400	0.000011355123800	0.000009930012600	0.000010250988400
Worst Case (unregister)	0.000004531818800	0.000003259682800	0.000006035725800	0.000006554774800	0.000009630370200	0.000011767716600	0.000016147233200	0.000021228037200	0.000018070783200	0.000018833118600



รูปที่ ง.1 เวลาเฉลี่ยในการค้นหาต่อแพ็กเก็ต (วินาที) ของ Prefix รูปแบบที่ 9 (8+8+8+8 บิต) จำนวนกฎ 1,000-10,000 ข้อ โดยประกาศตัวแปรชนิด register และไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify

วิเคราะห์ผลการทดลอง

จากผลการทดลองในตารางที่ ง.1 และรูปที่ ง.1 พบว่าเมื่อไม่ประกาศตัวแปรชนิด register ในฟังก์ชัน classify ที่จำนวนกฎเท่ากับ 1,000-32,000 ข้อ เส้นกราฟจะมีความแกว่งน้อยกว่าเมื่อประกาศตัวแปรชนิด register ในฟังก์ชัน classify ซึ่งเป็นไปตามสมมติฐานดังที่กล่าวมา

ภาคผนวก จ.
บทความที่ตีพิมพ์ในการประชุมวิชาการ ทางวิศวกรรมไฟฟ้า ครั้งที่ 27 (EECON 27)

การเพิ่มประสิทธิภาพการ Filter Packet ของ Firewall โดยกระบวนการทำ Hashing Function ด้วย Multi-binary Tree

Algorithm for Improving the Filtering Packet of Firewall with Hashing Function by Multi-binary Tree

ณัฐโชติ พรหมฉุทธิ์ และ อัครินทร์ คุณกิตติ

คณะเทคโนโลยีสารสนเทศ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ถนนฉลองกรุง เขตลาดกระบัง กรุงเทพฯ 10520

E-mail : pnuttachot@yahoo.com , akharin@it.kmitl.ac.th

CP12

บทคัดย่อ

Packet Filtering Firewall เป็นเครื่องมือหลักตัวหนึ่งที่มีบทบาทสำคัญในการควบคุมการจราจรบนเครือข่าย โดยอาศัยการเปรียบเทียบคุณสมบัติของ IP packet header กับ filter rule ทีละ rule อย่างไรก็ตามเมื่อ rule มีขนาดใหญ่ขึ้นจะทำให้การดูแลรักษาและตรวจสอบความถูกต้องยากขึ้น ประสิทธิภาพในการ filter packet จะลดลง ในบทความนี้จะนำเสนอวิธีการ filter packet เพื่อเพิ่มประสิทธิภาพในการทำงานของ Firewall ให้สามารถทำงานกับ rule ที่มีขนาดใหญ่ได้ดี โดยจัดโครงสร้างของ filter rule ขึ้นมาใหม่จาก rule list ให้อยู่ในรูปของ multi-binary tree และ hashing table ทำให้สามารถค้นหา rule ได้รวดเร็วขึ้น ลดคำสั่งในการเปรียบเทียบให้น้อยลง นอกจากนั้นในกระบวนการสร้าง binary tree เรายังสามารถลดรูปและตัด rule ที่ผิดปกติออกไปได้

คำสำคัญ : Packet Filtering Firewall, Filter Rule, Multi-binary Tree, Hashing Function

Abstract

Packet filtering firewall was main equipment which has important function for network traffic control. This function compared one by one between property of IP packet header and filter rule. However when the rule was expanded that will decrease capability of maintenance and verify. That result was reducing efficiency of filtering packet. So this article will present filtering packet method that increase performance of firewall when working with enlarge rules. The main idea was rearrange new filter rule structure from rule list transform to multi-binary tree and hashing table. That process will search rule faster and minimize comparing condition in firewall process. Anyway in create binary tree process we could reduce and cut off anomaly rules.

Keywords : Packet Filtering Firewall, Filter Rule, Multi-binary Tree, Hashing Function

1. คำนำ

Packet Filtering Firewall เป็นเครื่องมือหลักตัวหนึ่งที่มีบทบาทสำคัญในการควบคุมการจราจรบนเครือข่าย จากเครือข่ายหนึ่งไปยังอีกเครือข่ายหนึ่ง โดยอาศัยการเปรียบเทียบคุณสมบัติของ IP packet header field กับ filter rule ที่ได้กำหนดไว้จากนโยบายความปลอดภัย ก่อนที่จะตัดสินใจอนุญาตหรือไม่อนุญาตให้ packet นั้นผ่านไป โดยลำดับของ rule นั้นมีนัยสำคัญ Firewall จะทำการตรวจสอบ packet กับ rule ทีละ rule ตามลำดับ โดยเริ่มจาก rule แรก จนกระทั่งตรงกับเงื่อนไขที่กำหนด (matching packet)

เมื่อรายการของกฎ หรือ database มีขนาดใหญ่และซับซ้อนมากขึ้น Firewall จะต้องใช้ทรัพยากรในการประมวลผลมากขึ้น ทำให้มีผลกระทบต่อประสิทธิภาพในการสื่อสารบนเครือข่าย หากจะกำหนดกฎในการรักษาความปลอดภัยเพิ่มขึ้นก็จะต้องแลกด้วยประสิทธิภาพของเครือข่ายที่ลดลง นอกจากนั้นการกำหนด rule ที่ไม่ถูกต้อง เช่น กฎที่มีความขัดแย้งกัน กฎที่มีความซ้ำซ้อน และการเรียงลำดับของกฎที่ไม่ดีก็จะมีผลกระทบต่อการทำงานของ Firewall ได้เช่นกัน [1]

ในบทความนี้จะนำเสนอวิธีการ filter packet แบบใหม่เพื่อเพิ่มประสิทธิภาพในการทำงานของ Firewall ให้มากขึ้น โดยการจัดเรียงโครงสร้างของ rule ขึ้นมาใหม่ จากแบบ list ให้อยู่ในรูปของ multi-binary tree และ hashing table ทำให้สามารถค้นหา rule ได้รวดเร็วขึ้น และลดคำสั่งการเปรียบเทียบให้น้อยลง แม้ว่า Firewall จะมี database ขนาดใหญ่ก็ตาม

นอกจากนั้น ในกระบวนการสร้าง binary tree เรายังสามารถมองเห็นความผิดปกติของ rule ที่กำหนดไว้ ทำให้สามารถลดรูป และกำจัด rule ที่ผิดปกติออกไปได้ง่ายขึ้น ซึ่งจะแสดงให้เห็นในบทต่อไป

2. การสร้าง Multi-binary Tree

ในการบริหารจัดการ Firewall ผู้ดูแลระบบสามารถเพิ่ม rule เข้าไปใน database เพื่อให้ Firewall นำไปใช้งานได้โดยตรง แต่สำหรับกระบวนการที่นำเสนอนี้ จะต้องทำการ mapping จาก rule list แบบปกติให้อยู่ในรูปของ multi-binary tree และ hashing table ก่อนที่จะนำไปประมวลผล ในกระบวนการ mapping จะใช้แนวคิดของเซตและกราฟ ดังต่อไปนี้

2.1 มอง Rule ให้อยู่ในรูปของเซต

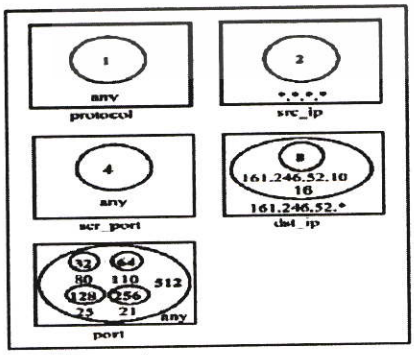
การ filter packet จะอาศัยการเปรียบเทียบคุณสมบัติของ IP packet header field เช่น protocol, source IP address, source port, destination IP address และ flag ต่างๆ เป็นเงื่อนไขในการตรวจสอบกับ rule โดยนำมาสร้างเป็น rule list ในรูปโดยทั่วไปดังต่อไปนี้

<order><protocol><src_ip><src_port><dst_ip>< dst_port><action>

ตารางที่ 1 ตัวอย่างของ rule list

order	protocol	src_ip	src_port	dst_ip	dst_port	action
1	any	*.*.*.*	any	161.246.52.*	80,110,25	accept
2	any	*.*.*.*	any	161.246.52.10	80,110,25,21	deny
3	any	*.*.*.*	any	161.246.52.*	any	deny

จากตารางที่ 1 จะแสดง rule list จำนวน 3 ข้อ ประกอบด้วย header field จำนวน 5 field ได้แก่ protocol, src_ip, src_port, dst_ip และ dst_port แต่ละ field จะมี object ต่างๆที่จะประกอบกันเป็นเงื่อนไขของหนึ่ง rule จาก rule list ให้ทำการ mapping object ให้อยู่ในรูปของเซต หากมี object ใดที่ทับซ้อนกัน เช่น object A = 161.246.52.25 – 161.246.52.100, object B = 161.246.52.90 – 161.246.52.120 ให้ทำการแตกส่วนที่ทับซ้อนกันเป็น object ใหม่ และตัดส่วนที่ซ้ำกันของ object ที่มีขอบเขตบนมากกว่าออก จะได้ object ทั้งหมด 3 object ดังต่อไปนี้ object A = 161.246.52.25 – 161.246.52.100, object B = 161.246.52.101 – 161.246.52.120, object C = 161.246.52.90 – 161.246.52.100 จากตารางที่ 1 จะได้เซตทั้งหมด 5 เซตดังรูปที่ 1



รูปที่ 1 เซตของ filter rule

นอกจากนั้นเราจะกำหนดค่าให้กับ object ทุกตัว โดยกำหนดให้ object แรกมีค่าเป็น 1 และ object ถัดไปมีค่าเพิ่มขึ้นเป็นสองเท่าของ object ก่อนหน้า ซึ่งแต่ละ object จะมีค่าไม่ซ้ำกัน

2.2 Mapping จากเซตให้อยู่ในรูปของกราฟ

นำเซตที่ได้จากข้อ 2.1 มาทำการ mapping ให้อยู่ในรูปของกราฟจะได้กราฟทั้งหมด 3 กราฟ ตามจำนวนของ rule โดยมี layer ทั้งหมด 5 layer คือ layer protocol, layer src_ip, layer scr_port, layer dst_ip และ layer dst_port โดยทำการ mapping rule ทีละข้อตามลำดับ เริ่มจาก rule แรกจนครบทุกข้อ มีขั้นตอนดังต่อไปนี้

1. สำหรับแต่ละ layer เลือก object ในเซตที่ตรงกับ rule นั้น มาสร้างเป็น node ถ้าใน object นั้นมี object อื่นเป็นสมาชิก ให้เลือก object ย่อยนั้นด้วย

2. ลากเส้นจาก node ใน layer แรกไปยัง node ใน layer ถัดไปทุกเส้นที่เป็นไปได้จนครบทุก layer

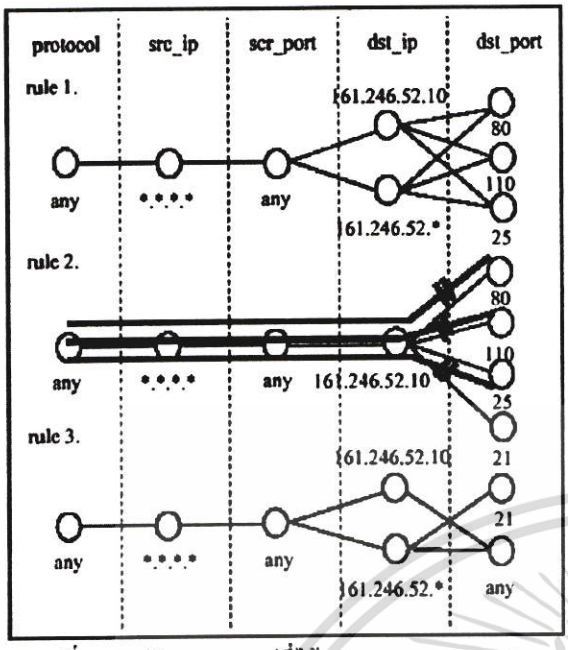
3. ถ้ากราฟมีเส้นทางที่ซ้ำซ้อนกับเส้นทางของกราฟก่อนหน้า (rule ก่อนหน้า) ให้ตัดเส้นทางนั้นทิ้ง

4. ทำการพิจารณาทีละ rule ตามเงื่อนไขข้อ 1 ถึง 3 เมื่อได้กราฟครบทั้งหมดแล้วจะทำการพิจารณาแต่ละ layer ของกราฟทุกกราฟพร้อมกัน ให้ตัด object ในเซตที่ไม่พบใน layer นั้นๆออกไป

5. บวกค่าของ node ทุกๆ node ในแต่ละเส้นทาง ค่าที่ได้จะเป็น hashing key และกำหนด action ของ rule ใน hashing table เพื่อใช้สำหรับกระบวนการ filter packet ของ Firewall ต่อไป โดยกำหนดให้มีฟังก์ชันดังต่อไปนี้

set_action(hashing_key) = action เป็นฟังก์ชันสำหรับกำหนด action ของ rule

action = get_action(hashing_key) เป็นฟังก์ชันตรวจสอบ action ของ rule



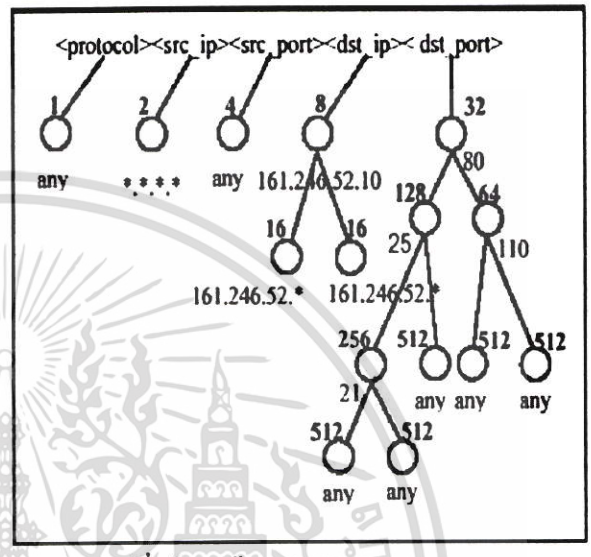
รูปที่ 2 กราฟจำนวน 3 กราฟที่ได้จากกระบวนการ mapping

จากรูปที่ 2 แสดงผลลัพธ์ที่ได้จากการ mapping rule ให้อยู่ในรูปของกราฟ จะเห็นว่ากราฟของ rule ที่ 2 มีเส้นทางที่ซ้ำซ้อนกันกับกราฟแรก 3 เส้นทางจึงทำการตัดเส้นทางเหล่านั้นออกไป เมื่อกลับไปพิจารณาอย่างละเอียดในตารางที่ 1 จะพบว่า rule ที่ 1 กับ rule 2 มีการทับซ้อนกัน ซึ่ง rule ตัวที่ 2 มีจุดประสงค์จะควบคุม packet จากที่ใดๆ ไม่ให้สามารถขอใช้บริการ HTTP, POP3, SMTP และ FTP จาก host 161.246.52.10 ได้ อย่างไรก็ตามก่อนที่จะถึง rule ที่ 2 นั้น rule ที่ 1 ยังคงอนุญาตให้ traffics ที่ไปยัง host 161.246.52.0 – 161.246.52.255 สามารถขอใช้บริการ HTTP, POP3 และ SMTP ได้ จากการใช้วิธีการตัดเส้นทางของกราฟที่ซ้ำซ้อนกัน ทำให้สามารถลดรูป rule ที่ 2 ให้มี dst_port เพียง port FTP เท่านั้น

ดังนั้น rule แรกจะมีเส้นทางทั้งหมด 6 เส้นทาง action เท่ากับ accept ส่วน rule ที่ 2 มีเส้นทางทั้งหมด 1 เส้นทาง action เท่ากับ deny และ rule ที่ 3 มีเส้นทางทั้งหมด 3 เส้นทาง action เท่ากับ deny นำค่าที่บวกได้ของแต่ละเส้นทางไปเป็น hashing key สำหรับกำหนด action ใน hashing table จากตัวอย่าง rule ที่ 2 มีค่า hashing_key เท่ากับ 1 + 2 + 4 + 8 + 256 = 271, action เท่ากับ deny ทำการกำหนดค่าใน hashing tale ดังต่อไปนี้ set_action(271) = deny

2.3 Mapping เซตของ rule ให้อยู่ในรูปของ Multi-binary Tree

Multi-binary Tree เป็น database ที่ Firewall จะใช้ในกระบวนการ filter packet ซึ่ง การ mapping เซตของ rule จะได้จำนวน binary tree เท่ากับจำนวน layer ของกราฟ จากข้อ 2.1 และข้อ 2.2 นำเซตที่ได้ปรับปรุงแล้วมาทำการสร้างเป็น binary tree จำนวน 5 tree โดยนำ object ที่มีขนาดเล็กมาทำการ insert ลงใน binary tree ก่อนด้วยวิธีการปกติ แต่หากเป็น object ที่เป็นซุปเปอร์เซตให้ทำการ insert object ลงไปทุก leaf node ของ tree ที่เป็นซับเซตของ object นั้นๆ โดยค่าที่กำหนดให้แต่ละ object ในข้อ 2.1 จะนำมาเป็น key สำหรับสร้าง hashing key ต่อไป ดังรูปที่ 3



รูปที่ 3 โครงสร้างของ rule แบบ binary tree

3. ขั้นตอนการ Filter Packet ด้วย Multi-binary Tree Rule และ Hashing Function

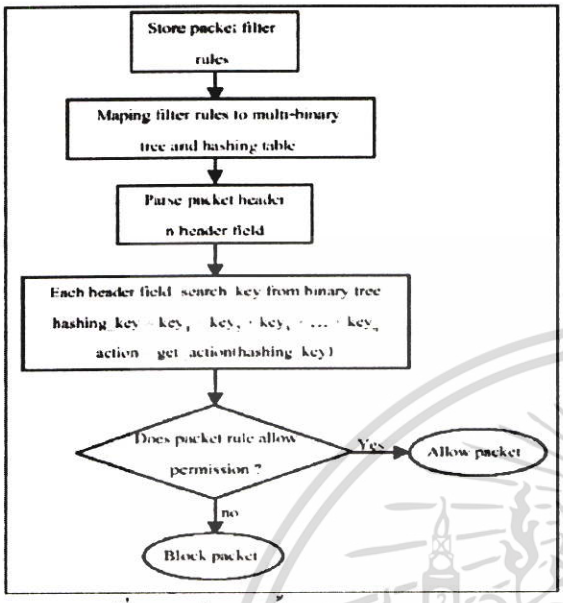
โดยทั่วไป Packet Filtering Firewall จะทำการ filter packet โดยเปรียบเทียบคุณสมบัติของ IP packet header กับ Firewall rule ทีละข้อตามลำดับจนกระทั่งตรงกับเงื่อนไขที่กำหนด แต่สำหรับกระบวนการที่นำเสนอจะทำการเปรียบเทียบคุณสมบัติของ IP packet header field ใน binary tree สำหรับ field นั้นๆ และนำค่า key ที่ได้มาบวกกันซึ่งจะได้เป็นค่า hashing key นำ hashing key ที่ได้ไปตรวจสอบ action ของ packet ใน hashing table ด้วยฟังก์ชัน action = get_action(hashing_key) ดังรูปที่ 4 โดยกำหนดให้มีฟังก์ชันดังต่อไปนี้

key = search_key(binary_tree,header_field) เป็นฟังก์ชันสำหรับค้นหา key จาก binary tree ด้วยการเปรียบเทียบกับข้อมูล packet header field

ตัวอย่าง ให้ packet ที่เข้ามามี IP packet header คือ protocol = tcp, src_ip = 216.239.59.104, src_port = 2064, dst_ip = 162.246.52.10 และ dst_port = 21 จากการค้นหาค่า key ของแต่ละ tree ในรูปที่ 3 จะได้

hashing_key = search_key(protocol,tcp) + search_key(src_ip,216.239.59.104) + search_key(src_port,2064) + search_key(dst_ip,162.246.52.10) + search_key(dst_port,21) = 1 + 2 +

4 + 8 + 256 = 271 ฟังก์ชัน get_action(271) จะมีค่าเท่ากับ deny ดังนั้น Firewall จะไม่อนุญาตให้ packet นี้ผ่านไปได้



รูปที่ 4 แผนผังแสดงขั้นตอนการ filter packet

4. วิเคราะห์ประสิทธิภาพเบื้องต้นของอัลกอริทึม

โดยทั่วไป Firewall จะตรวจสอบ IP packet header field กับ rule ทีละข้อตามลำดับ ดังนั้นประสิทธิภาพของอัลกอริทึมที่ได้คือ O(n) เมื่อ n คือ จำนวน rule ทั้งหมด สำหรับกระบวนการที่นำเสนอ Firewall จะตรวจสอบ IP packet header field จำนวน d field โดยแต่ละ field จะทำการค้นหาค่า key จาก binary tree ถ้า binary tree แต่ละตัวมีความสมดุลกัน (balance tree) ประสิทธิภาพของอัลกอริทึมที่ได้คือ $O(d\log_2 n)$

จากข้อ 2.3 การสร้าง binary tree จะต้องทำการ insert object ที่มีขนาดเล็กที่ด้านบนของ tree และ object ที่เป็นซุปเปอร์เซตที่ด้านล่างของ tree เรียงลำดับกันไป ดังนั้นการเพิ่มหรือลบ rule ทุกครั้งจึงต้องทำการปรับโครงสร้างของ binary tree ใหม่ ทำให้ Firewall ต้องใช้เวลาในการปรับปรุง rule ที่สูง

5. สรุป

Packet Filtering Firewall เป็นเครื่องมือหลักตัวหนึ่งที่มีบทบาทสำคัญในการควบคุมการจราจรบนเครือข่าย การเพิ่ม rule ที่มีความซับซ้อนและมีขนาดใหญ่จะลดประสิทธิภาพในการ filter packet ของ Firewall และมีผลกระทบต่อการติดต่อสื่อสารบนเครือข่าย

บทความนี้ได้นำเสนอวิธีการเพิ่มประสิทธิภาพการ filter packet ของ Firewall โดยการจัดโครงสร้างของ filter rule ขึ้นมาใหม่ โดยใช้แนวคิดของเซตและกราฟ ทำการ mapping rule list ให้อยู่ในรูปของ multi-binary tree และ hashing table ทำให้สามารถค้นหา rule ได้รวดเร็วขึ้น นอกจากนั้นยังสามารถลด rule ที่ผิดปกติออกไปได้ โดยผู้ดูแลระบบยังคงสามารถสร้าง rule ได้ด้วยวิธีการเดิม

จากการวิเคราะห์เบื้องต้นพบว่าอัลกอริทึมของเราให้ประสิทธิภาพการค้นหามากกว่าอัลกอริทึมเดิม ลดภาระงานในการ filter packet ของ Firewall ลง โดยประสิทธิภาพของกระบวนการนี้จะขึ้นอยู่กับ 1. ความสมดุลของ binary tree ที่สร้างขึ้น 2. ลักษณะของ traffic 3. ความถี่ในการปรับปรุง rule

สำหรับ static Firewall การปรับปรุง rule มักกระทำโดยผู้ดูแลระบบซึ่งอาจจะต้องการความเร็วที่ต่ำกว่า dynamic Firewall ซึ่งการปรับปรุง rule จะกระทำโดยผู้ดูแลระบบหรือ protocol ก็ได้ โดยความเร็วในการปรับปรุง rule ควรไม่ต่ำกว่าความเร็วในการค้นหามากนัก จากอัลกอริทึมที่นำเสนอเมื่อทำการเพิ่มหรือลบ rule, Firewall จะต้องทำการปรับโครงสร้างของ binary tree ใหม่ จึงทำให้เหมาะสมสำหรับ static Firewall ซึ่งในอนาคตจะทำการปรับปรุงอัลกอริทึมให้สามารถรองรับการปรับปรุง rule แบบ dynamic และนำผลที่ได้จากการใช้งานจริงไปพัฒนาให้มีประสิทธิภาพยิ่งขึ้นต่อไป

เอกสารอ้างอิง

[1] Ehab S. Al-Shaer and Hazem H. Hamed, "Firewall Policy Advisor for Anomaly Discovery and Rule Editing", IEEE Computer Society.

[2] T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu and N. Dulay, "Compiling Policy Descriptions into Reconfigurable Firewall Processors", IEEE Computer Society, 2003.

[3] Chris Hare and Karanjit Siyan, "Internet firewall and network Security", 2nd Edition, Newriders, 1996.

อัครินทร์ คุณกิตติ : อาจารย์ประจำคณะเทคโนโลยีสารสนเทศ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

งานวิจัย : - Computer Systems
- Distributed Systems
- Data Communication and Network

ณัฐโชติ พรหมฤทธิ์ : จบปริญญาตรีสาขา วิทยาการคอมพิวเตอร์ มหาวิทยาลัยทักษิณ

ปัจจุบัน นักศึกษาระดับปริญญาโท คณะเทคโนโลยีสารสนเทศ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

งานวิจัย : - Packet Filtering Firewall

ภาคผนวก ฉ.

บทความที่ตีพิมพ์ใน International Conference on Control, Automation and System 2005 (ICCAS 2005)

Bitmap Intersection Lookup (BIL) : A Packet Classification 's Algorithm with Rules Updating

Akharin Khunkitti*, Nuttachot Promrit**

*Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang, Thailand
(Tel : +66-2-737-2551; E-mail: akharin@it.kmitl.ac.th)

**Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang, Thailand
(Tel : +66-2-737-2551; E-mail: pnuttachot@yahoo.com)

Abstract: The Internet is a packet switched network which offers best-effort service, but current IP network provide enhanced services such Quality of Services, Virtual Private Network (VPN) services, Distribute Firewall and IP Security Gateways. All such services need packet classification for determining the flow. The problem is performing scalable packet classification at wire speeds even as rule databases increase in size. Therefore, this research offer packet classification algorithm that increase classifier performance when working with enlarge rules database by rearrange rule structure into Bitmap Intersection Lookup (BIL) tables. It will use packet 's header field for looking up BIL tables and take the result with intersection operation by logical AND. This approach will use simple algorithm and rule structure, it make classifier have high search speed and fast updates.

Keywords: Packet Classification, Packet Flow, Firewall

1. INTRODUCTION

The packet classification nowadays becomes the important tool of many applications, such as firewall and internet router. These applications will initially search by classifying packets, detecting packets accorded to the determined condition and high priority choosing the rule. Even though the internet is a packet switched network which offers the best - effort service (the sent-received rate never guaranteed), IP network can provide many different services responding to users likes: Quality of Service, Virtual Private Network (VPN) Services, Distribute Firewall and IP Security Gateways. Rules updating can be maintained by both network administrators and Protocol, thus, the problem about packet classification is it must operate on high-speed link bandwidth along with the larger number of rules.

Many researches focus on solving packet classification algorithms problems such as [1], [2]. These researches proposed the method using High - Search Speed Complexity that defined as $O\left(D \cdot \log_2 \cdot N + \frac{N}{WORD}\right)$ and O(1) on hardware but they still encounter such problems as:

1. Bitmap-Intersection method: Storage Complexity of this method is $O(D \cdot N^2)$ which waste too much space if the packet classification algorithm have to work with many rules [1].

2. Ternary CAM method: Storage Complexity of this method is O(N) and must be developed only on hardware; nevertheless, this method use memory that has the amount of Transistor more than the other, so it wastes more power [2].

Thus, this paper introduces packet classification algorithms with high – speed searching rules: searching Speed Complexity defined as $O\left(\frac{D \cdot W}{PMX} \cdot \frac{N}{WORD}\right)$ on the appropriate software, spending appropriate time to updating rules while Storage Complexity is defined $O\left(\frac{D \cdot W}{PMX} \cdot 2^{PMX} \cdot N\right)$

2. RELATED WORKS

2.1 Meaning of Packet Classification

Packet Classification is mechanical method being used to inspect network packets. This method works by comparing data in header of packet more than 1 field. For example, compare from these fields: Source/Destination Network-layer Address (32 bit), Source/Destination Transport-layer Port Number (16 bit), Type-of-Service (8 bit), Protocol (8 bit) ,and Transport-layer Protocol Flag (8 bit) then determine packet action, let each flow packet be determined by the rule. Every each packet in the same current will perform the same action. Instead of defining this process as "Packet Filtering", this paper prefers to define this process as "Packet Classification" because the words "Packet Classification" is overall admitted and ultimately define its process, moreover the word "classify" denotes the action of every packet while the word "filtering" denotes the action of only preferred packet [3].

IP Network services examples are: 1) Packet Classification according to the rules method applied with Route Lookup considering from Destination IP Address and determine the suitable Next-hop, 2) Packet Classification according to the rules method applied with firewall considering from Source IP Address , Destination IP Address, Source Port, Destination Port or any other field and determine action likes, ACCEPT or DROP ,and 3) Classification according to the rules method applied with Quality of Service considering from Source IP Address and Source Port then determine Queue further.

2.2 Packet Classification Requirements

Packet Classification according to the rules method is the significant tool that used to determine the flow of packets on Internet that must operate on the high - speed link bandwidth with the large rule. So Packet Classification applications must show the best perform. The working procedure is then determined by following conditions:

1. High search speed: Packet Classification according to the rules method must operate on the high – speed link bandwidth.

2. Low storage space: Good Packet Classification according to the rules method must be capable to cooperate with larger rule but use the low hard disk space.

3. Less updating time Packet Classification according to the rules method: While augmenting or decreasing rule, the application that has high frequency of rule updating must spend the high search speed also. The quickness of rule updating shouldn't be lower than the quickness of searching.

2.3 Convert rules to Prefix

Many Packet Classification according to the rules methods must convert rule to Prefix Sets, determine Prefix size equal to header field (W) of packet. Each set number is not more than 2W – 2 [4]. For examples, determining each Prefix size equal to 4 bits, rule no. 1 is 4-7, thus, set of Prefix is {01**}, or rule no. 2 equals to 1- 14, thus, set of Prefix is {0001, 001*, 01**, 10**, 110*, 1110}

3. BITMAP INTERSECTION LOOKUP (BIL)

From the problems stated in the previous section, we realize the importance of packet classification algorithms. Then our approach offers simple packet classification algorithms which can be used in multi - field searching, moreover, this method can also work well on software and hardware.

Figure 1 below shows an architecture of BIL method which consists of following components: 1) BIL table with D tables, 2) each table includes 2^{PMX} addresses, each address includes N bits (0 bits ups to N-1 bits called Bitvector), 3) each position on Bivector is identical with rule contained in rule table, 4) arrange rule by priority (descending). 5) to augment rules, rule will be defined as Mask or Range, and 6) convert rule into Prefix in order to use for Set Bitvector in BIL table and store every action of each rule into rule table further.

BIL method divides header of packets into smaller block then take the value of each block to look up in BIL table. The result from looking up called Bitvector, take that Bivector to be intersected with AND logic, and then the algorithm priority chooses the most important rule as the arrows stated in Fig. 1. Classify Algorithm and Update Algorithm will show in Fig. 2 and 3, and example of Classify in Fig.4

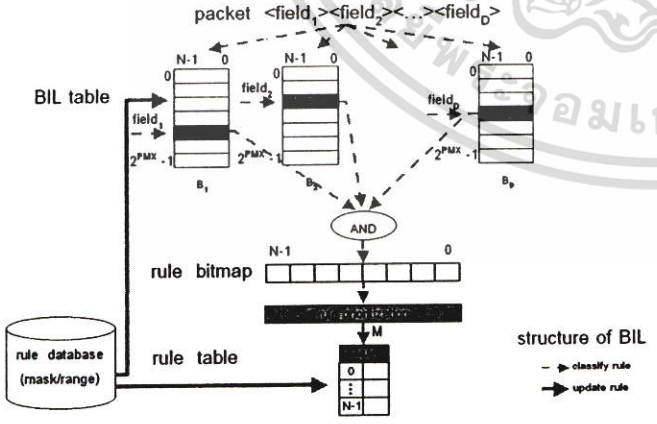


Fig. 1 An architecture of BIL

In defining this algorithm, we determine:

N is number of rules

D is number of header field

F_i is number of bits in each divided header filed: $1 <= i <= D$

WORD is bits size used in each processing

If F_i has the same size in every field, each field is W bits, since all bit size equal to D•W bit

Divide F_i into $P_{i,j}$ bit, determine

M_i is Block divided in F_i where $\sum_{j=1}^{M_i} P_{i,j} = F_i$

Consider any F_i in only 1 field, where $P_{i,j}$ of F_i equal to PM_i bits, thence $M_i = \frac{F_i}{PM_i}$

Define PM_i as PMX bit in every field and MX is Block amount in each field, thence $MX = \frac{W}{PMX}$ and the number of the table is D•MX

```
1. Read header field of packet and keep into buffer
2. for (i=0; i<D; i++ ) {
3.     for (j=0; j<MX; j++ ) {
4.         Divide header field of packet i into small block,
           shift bit right had side bit by PMX bit,
           record into memory
5.     }
6. }
7. for (i=0; i<N/WORD; i++ ) {
8.     for (j=0; j< D•MX; j++ ) {
9.         take the header field to Lookup in BIL table and
           take the Bitvector at position WORD i from BIL table j
           to intersect by logic AND
10.    }
11.    if (readout intersected by logic AND != 0) {
12.        calculate rule number of the first 1 bit of
           Bitvector and Return Action value from rule table
13.    }
14. }
15. Return Default Action
```

Fig. 2 Classify Algorithm

```
1. Read header field of packet and keep into buffer
2. translate the instruction of rule and record value of
   each field into memory
3. record ruleid, Action and instruction into rule table,
   define tmp = 0 and d = 0
4. for (i=0;i<D; i++) {
5.     tmp +=d, then convert rule value to Prefix (value/mask)
6.     While (current != NULL) {
7.         d=temp
8.         for (k=0;k < MX; k++) {
9.             let dividing Prefix value (value/mask) be j member
               of field i by shift bit right had side bit by PMX bit,
               record the readout as value and mask
10.            for (l=value AND mask ;l<=( value AND mask)+~mask; l++) {
11.                set Bitvector at address no.l, bit no. ruleid in BIL table d
12.            }
13.            d++
14.        }
15.    }
16.}
```

Fig. 3 Update Algorithm

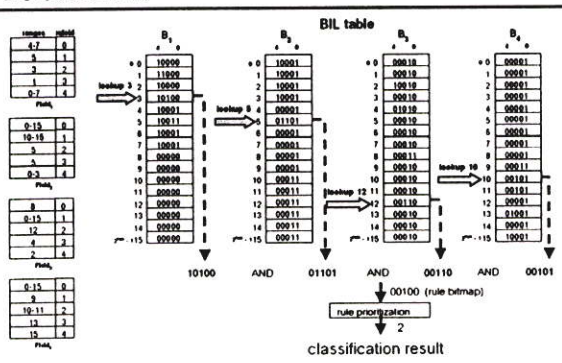


Fig. 4 example of Classify

Our research shows that the quickness of searching, the quickness of rule updating and managing storage space of BIL program depend on specific Prefix size, so we concentrate on the study of Prefix size that allow the program work best, considering from the product of working performance which are: 1) Product of updating time and storage space 2) Product of searching time and storage space 3) Product of searching time and updating time, and 4) Product of searching time, updating time and storage space. The objective is to study the influence each product on the highest and the lowest of the graph that well affected towards the program's working performance.

4. EXPERIMENTAL RESULTS

According to the hypothesis mentioned earlier, we had developed BIL program with gcc complier on Linux Kernel 2.4 operating system and had created rules' database and classifying packets which are suitable for evaluating the program performances. These rule databases stored in script file named "rules.db" and packet databases stored in file name "dump.db" because it's necessary to control an interfere, which could happen from network traffic and network application or even from operating system's Kernel, not to affect the working performance of program. This procedure allow us not to mistakenly measure searching time and updating rule besides that we can test and compare working performance of Linear Search program [4] and Bitmap-Intersection program [1], we need to develop these programs in the same environment.

We had designed the experiments into 2 sections which are: Section 1 is to divide Prefix size into small Block (such as dividing Source IP Address into 16+16 bits) to calculate the precise value that allow BIL program work with the best performance. Section 2 is to compare working performance of Linear search program, Bitmap Intersection program and BIL program.

The database used in section 1 is 1 field rule database and let it be Source IP Address (32 bits) and Source MAC Address (48 bits), The database used in section 2 is 1 field rule database and let it be Source IP Address (32 bits). We specially create database cases for measuring searching time and updating time are: 1) Best Case 2) Average Case ,and 3) Worst Case

Experiment 4.1: Dividing Prefix sizes to calculate the most effective value

Objective: To calculate the stable prefix size and changeable prefix size which allow BIL program spend less time in searching and updating process, moreover it effectively use storage space for Source MAC Address (48 bits) where number of rules are 4, 096 .

Experiment 4.1.1: Dividing stable Prefix size by using 1 field rule database as Source MAC Address

The result shows:

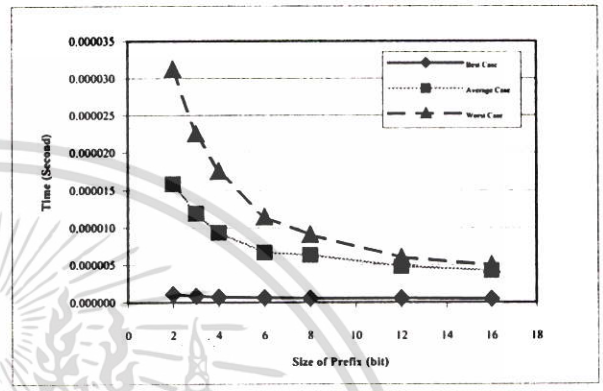


Fig. 5 Average searching time per packet (second of time) for Source MAC Address

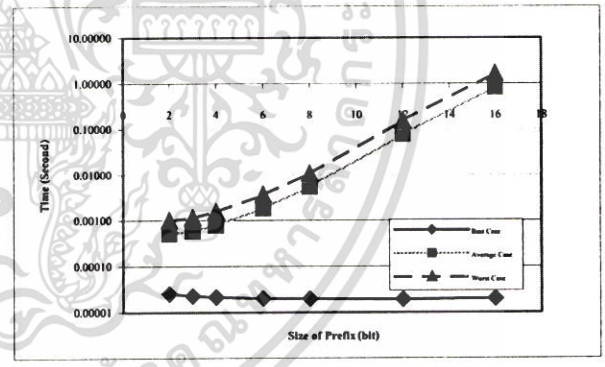


Fig. 6 Average inserting time per rule (second of time) for Source MAC Address with logarithm scale

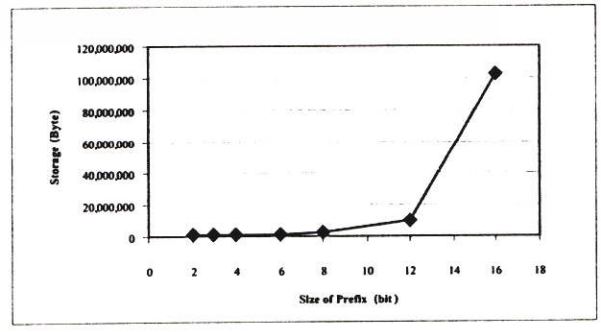


Fig. 7 Storage space (bytes) for Source MAC Address

It appears in Fig.5 that the quickness of searching time relates to PMX size. The bigger PMX is determined; BIL program spends lesser searching time in every case. When PMX is increased to 16 bits, BIL program spends the less searching time.

From Best Case studied in Fig.6, the bigger PMX is determined; BIL program spends lesser updating time. When PMX is increased to 16 bits, BIL program spends the less updating time. Moreover, when PMX is increased in Average and Worst Case, BIL program spends more updating time as well. When PMX is increased up to 16 bits, BIL program spends the most inserting time.

Fig.7 shows that BIL program uses more storage space when PMX extend to 16 bits

Table 1 Shows Prefix sizes suitable for Source MAC Address considering from the lowest point of products

Products	Cases		
	Best Case	Average Case	Worst Case
1. updating time x storage space	4 Bits	2 Bits	2 Bits
2. searching time x storage space	6 Bits	6 Bits	6 Bits
3. searching time x updating time	16 Bits	3 Bits	3 Bits
4. searching time x updating time x storage space	6 Bits	3 Bits	3 Bits

Considering from 4 products presented in section 3, the result concluded from Table 1 shows that Worst Case that let PMX is 2 bits, 3 bits, and 6 bits: is the most suitable Prefix size for Source MAC Address. We had brought these products to use in experiment 4.1.2 further.

Experiment 4.1.2 : Dividing changeable Prefix patterns by using 1 field rule database as Source IP Address

Determined 13 patterns of Prefix size:

- Pattern 1: let Prefix size be 2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 bits
- Pattern 2 let Prefix size be 3+3+3+3+3+3+3+3+3+3+2 bits
- Pattern 3: let Prefix size be 4+4+4+4+4+4+4+4 bits
- Pattern 4: let Prefix size be 2+6+10+14 bits
- Pattern 5: let Prefix size be 4+8+8+12 bits
- Pattern 6: let Prefix size be 2+4+6+8+10+2 bits
- Pattern 7: let Prefix size be 6+6+6+6+6+2 bits
- Pattern 8: let Prefix size be 7+7+7+7+4 bits
- Pattern 9: let Prefix size be 8+8+8+8 bits
- Pattern 10: let Prefix size be 9+9+9+5 bits
- Pattern 11: let Prefix size be 10+10+10+2 bits
- Pattern 12: let Prefix size be 11+11+10 bits
- Pattern 13: let Prefix size be 12+12+8 bits

The result shows:

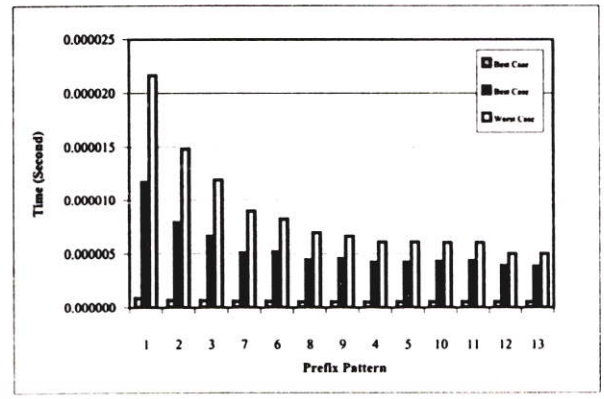


Fig. 8 Average searching time per packet (second of time) of Prefix pattern 1 to 13, ordered by the quickness of searching ascending according to Worst Case

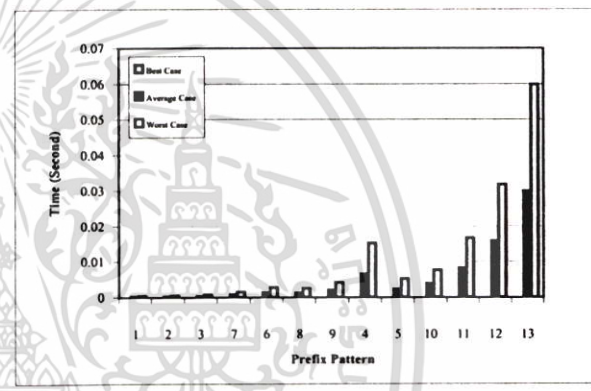


Fig. 9 Average inserting time per rule (second of time) of Prefix pattern 1 to 13, ordered by the quickness of searching ascending according to Worst Case

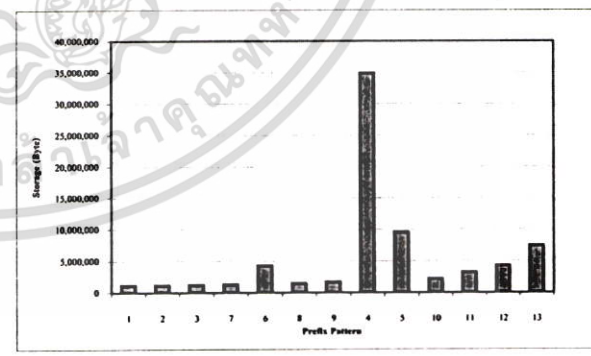


Fig. 10 Storage space (bytes) of Prefix pattern 1 to 13, ordered by the quickness of searching ascending according to Worst Case

From the results are presented in Fig. 8, we find that Pattern 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 bits) in Worst Case spends the most searching time and Pattern 13 (12+12+8 bits) spends the less searching time.

The results are presented in Fig. 9 appear that pattern 13 (12+12+8 bits) in Best Case spends the less time in inserting

rule and pattern 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 bits) spends the most time in inserting rule. While Prefix pattern 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2bits) in Average and Worst Case spends the less time in inserting rule and Pattern 13 (12+12+8 bits) spends the most time in inserting rule.

It appears in Fig. 10 that pattern 1 (2+2+2+2+2+2+2+2+2+ 2+2+2+2+2+2+2 bits) uses the smallest storage space while pattern 4 (2+6+10+14 bits) uses the largest storage space.

Table 2 The suitable Prefix of Prefix pattern 1 to pattern 13 considered by the lowest point of the product.

Products	Case		
	Best Case	Average Case	Worst Case
1. updating time x storage space	Pattern 3	Pattern 1	Pattern 1
2. searching time x storage space	Pattern 3	Pattern 8	Pattern 8
3. searching time x updating time	Pattern 13	Pattern 2	Pattern 2
4. searching time x updating time x storage space	Pattern 8	Pattern 2	Pattern 2

Table.2 shows that Worst Cases at Prefix pattern 1 (2+2+2+2+2+2+2+2+2+2+2+2+2+2+2 bits), pattern 2 (3+3+3+3+3+3+3+3+3+3+2 bits), and pattern 8 (7+7+7+7+4 bits) are the most suitable pattern considering from all 4 products presented in section 3. In order to compare working performance of another program in the 4.2.1~4.2.3 experiments, we choose Prefix pattern 2 (3+3+3+3+3+3+3+3+3+3+2 bits) because pattern 2 is utmost well balance.

Experiment 4.2: The comparison of working performance of Linear Search program, Bitmap Intersection program , and BIL program

Objective: To compare working performance between Linear Search program, Bitmap Intersection program and BIL program using 1,000-10,000 rules for Source IP Address

Experiment 4.2.1: The comparison of time spent in searching rules of Linear Search program, Bitmap-intersection program and BIL program

The result shows:

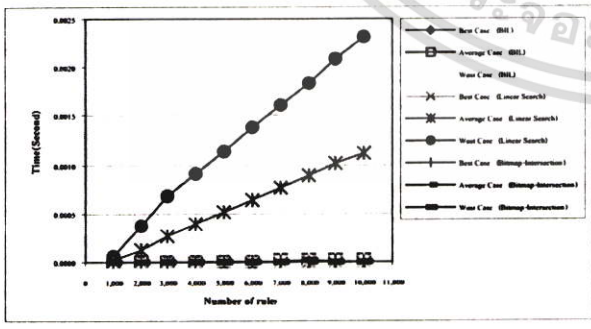


Fig. 11-A Average searching time per packet (second of time) of Linear Search program, Bitmap-intersection program, and BIL program.

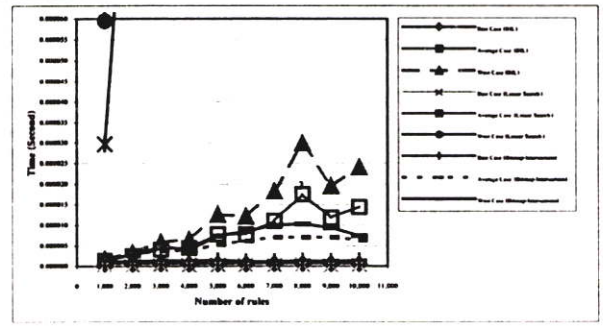


Fig. 11-B Average searching time per packet (second of time) of Linear Search program, Bitmap-intersection program, and BIL program operated with zooming in.

Fig. 11-A and 1-B prove that these programs identically spend searching time in Best Case where algorithms are 1,000-10,000 rules. Linear Search program spends 0.00000023 seconds, Bitmap-Intersection program spends 0.00000128 seconds, and BIL program spends 0.00000077 seconds.

Both figures also show that each program tends to spend more searching time when N is increased in Average and Worst Case. In both case, these programs can be ascending arranged as follow: Bitmap-intersection program, BIL program, and Linear Search program.

Experiment 4.2.2: The comparison of time spent in inserting rules of Linear Search program, Bitmap-intersection program and BIL program

The result shows:

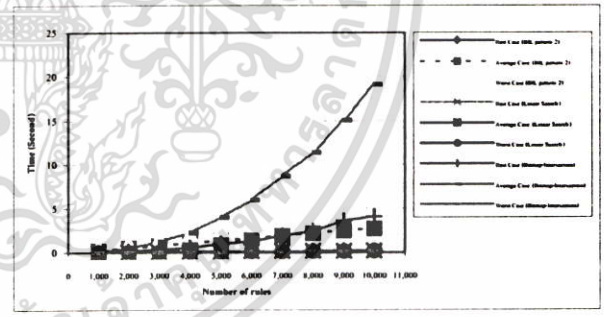


Fig. 12-A Average inserting time (second of time) of Linear Search program, Bitmap-intersection program, and BIL program.

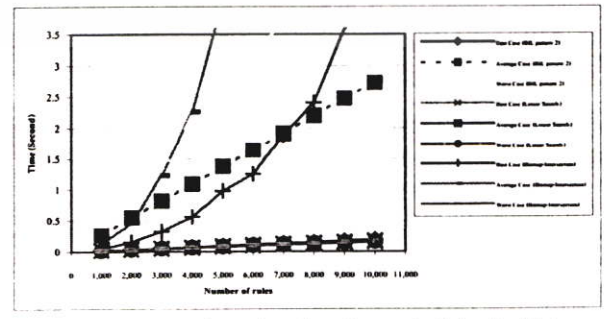


Fig. 12-B Average inserting time (second of time) of Linear Search program, Bitmap-intersection program, and BIL program operated with zooming in

It appear in Fig. 12-A and 12-B that when N is increased in Best and Average Case, each program tends to spend more time in inserting rule at the number of rules is 1,000-10,000 which can be ascending arranged as follow: Linear Search program, Bitmap-intersection program, and BIL program.

Experiment 4.2.3: The Comparison of storage space of Linear Search program, Bitmap-intersection program and BIL program

The result shows:

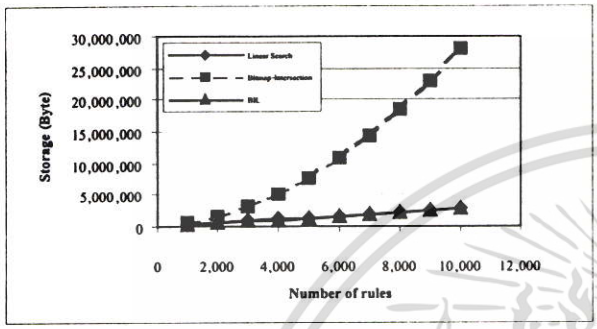


Fig. 13 Storage space (byte) of Linear Search program, Bitmap-intersection program ,and BIL program

The experiment in Fig. 13 shows that each program tends to spend more time in inserting rule when N is increased and the number of rules is 1,000-10,000. BIL program and Linear Search use the identical storage space but Bitmap-intersection uses the largest storage space.

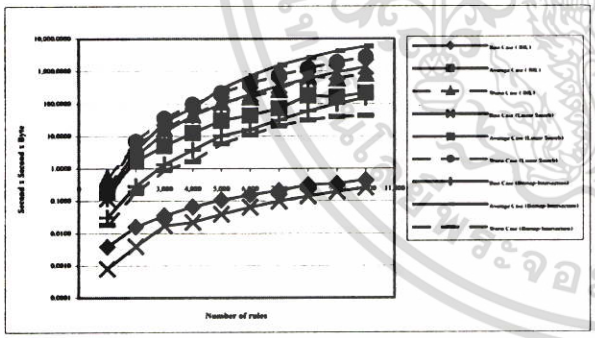


Fig. 14 products of searching time, updating time and storage space of Linear Search program, Bitmap-intersection program, and BIL program operated with logarithm scale

Figure. 11-A and 11-B shows that Linear Search program tends to spend most searching time. Figure. 12-A and 12-B shows that Bitmap-intersection program tends to spend most time for inserting rule, and Figure. 13 shows that Bitmap-intersection tends to use the largest storage space.

Table 3 The ascending results from the comparison of each product of working performance between Linear Search program, Bitmap-Intersection program, and BIL program for Source IP Address

Case	Products			
	updating time x storage space	searching time x storage space	searching time x updating time	searching time x updating time x storage space
Best Case	1.BIL 2.Linear Search 3.Bitmap-Intersection	1.Linear Search 2.BIL 3.Bitmap-Intersection	1.Linear Search 2.BIL 3.Bitmap-Intersection	1.Linear Search 2.BIL 3.Bitmap-Intersection
Average Case	1.Linear Search 2.BIL 3.Bitmap-Intersection	1.BIL 2.Bitmap-Intersection 3.Linear Search	1.BIL 2.Bitmap-Intersection 3.Linear Search	1.BIL 2.Linear Search 3.Bitmap-Intersection
Worst Case	1.Linear Search 2.Bitmap-Intersection 3.BIL	1.BIL 2.Bitmap-Intersection 3.Linear Search	1.Bitmap-Intersection 2.BIL 3.Linear Search	1.Bitmap-Intersection 2.BIL 3.Linear Search

The results from Table 3 appears that BIL program in Average Case proves more effectual than products of others which are: Product of searching time and storage space, Produce of searching time and updating time and Product of searching time, updating time and storage space.

5. CONCLUSIONS

This paper introduces the increasing efficiency of packet classification methods which are: 1) rearranging rules structures into Bitmap Intersection Lookup tables (BIL), 2) in searching method, dividing the header of packets into Block ,then takes each Block to look up in BIL, and 3) intersecting the result by AND logic. These methods not only simplify the working procedure of the program but also rules' structure. As a result, the program spend lesser searching time and has appropriate interval to adjust rules.

Our experiments show that BIL program can work efficiently where Prefix is 3 bits. And when compared BIL program with Linear Search program and Bitmap-intersection program, we consider that the program efficiency is related to complexity as stated previously. Furthermore, BIL program discernibly shows the best performance in Average Case between any other programs.

Results may also pave the way to the further experiment such as comparing the efficiency of other hardware internet applications like firewall or router, etc.

REFERENCES

[1] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching", *In Proc. ACM Sigcomm '98*, Sept. 1998.

[2] J. van Lunteren and A.P.J. Engbersen, "Multi-field packet classification using ternary CAM", *IEE Electronics Letters*, Vol. 38, No. 1, Jan. 2002.

[3] M.L. Bailey, B.Gopal, M.Pagels, L.L.Peterson, and P.Sarker, "PATHFINDER: A Pattern-Based Packet Classifier", *In Proc. Operating Systems Design and Implement*, Nov. 1994.

[4] P. Gupta and N. McKeown, "Algorithms for Packet Classification", *IEEE Netw.*, 2001.

ประวัติผู้เขียน

ชื่อ-นามสกุล	นายณัฐโชติ พรหมฤทธิ์
วัน เดือน ปีเกิด	19 มีนาคม พ.ศ. 2521
ที่อยู่	95 ม.5 ต.น้ำจืดน้อย อ.กระบุรี จ.ระนอง 85110
ประวัติการศึกษา	พ.ศ. 2542 วิทยาศาสตรบัณฑิต สาขาวิชาวิทยาการคอมพิวเตอร์ มหาวิทยาลัยทักษิณ
ผลงานทางวิชาการที่ได้รับการตีพิมพ์	- การเพิ่มประสิทธิภาพการ Filter Packet ของ Firewall โดยกระบวนการทำ Hashing Function ด้วย Multi-binary Tree - Bitmap Intersection Lookup (BIL) : A Packet Classification 's Algorithm with Rules Updating