การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

นายสถาพร สงวนวงษ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2554

RUNTIME DETECTION OF SOFTWARE MODIFICATION

Mr. Sathaporn Sa-ngounwong

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2011

หัวข้อวิทยานิพนธ์	การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน
โดย	นายสถาพร สงวนวงษ์
สาขา	วิทยาศาสตร์คอมพิวเตอร์
อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก	รองศาสตราจารย์ ดร. พรศิริ หมื่นไชยศรี

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

…….. คณบดีคณะวิศวกรรมศาสตร์
(รองศาสตราจารย์ ดร. บุญสม เลิศหิรัญวงศ์)

คณะกรรมการสอบวิทยานิพนธ์

…… ประธานกรรมการ
(รองศาสตราจารย์ ดร. สมชาย ประสิทธิ์จูตระกูล)

……... อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก
(รองศาสตราจารย์ ดร. พรศิริ หมื่นไชยศรี)

…… กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร. เกริก ภิรมย์โสภา)

…… กรรมการภายนอกมหาวิทยาลัย
(รองศาสตราจารย์ ดร. วีระ บุญจริง)

สถาพร สงวนวงษ์ : การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน (RUNTIME DETECTION OF SOFTWARE MODIFICATION) อ. ที่ปรึกษาวิทยานิพนธ์หลัก: รศ.ดร. พรศิริ หมื่นไชยศรี, 70 หน้า.

การเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน มีจุดมุ่งหมายในแง่ลบเป็นส่วนใหญ่ ตัวอย่างเช่น การเปลี่ยนแปลงซอฟต์แวร์เพื่อละเมิดการใช้งานซอฟต์แวร์ หรือ การเปลี่ยนแปลงซอฟต์แวร์เพื่อโจมตีระบบคอมพิวเตอร์ เป็นต้น การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานสามารถกระทำได้หลายวิธี ซึ่งการทำซอฟต์แวร์ปรสิต เป็นวิธีการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่มีประสิทธิภาพวิธีหนึ่ง

งานวิจัยนี้ได้นำเสนอวิธีการที่มีชื่อว่า “การตรวจหาคำสั่งระบบเฉพาะแบบ” เพื่อใช้สำหรับตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่เกิดจากการทำงานของซอฟต์แวร์ปรสิต การทำงานของการตรวจหาที่งานวิจัยนี้นำเสนอ ในขั้นแรกทำการเลือกคำสั่งระบบที่สามารถก่อให้เกิดการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานได้ออกมาหนึ่งกลุ่มเรียกว่า คำสั่งระบบเฉพาะแบบ ซึ่งสามารถแบ่งย่อยได้เป็น 3 ประเภท ได้แก่ คำสั่งระบบที่ก่อให้เกิดการเปลี่ยนแปลงระหว่างโปรเซส คำสั่งระบบที่ก่อให้เกิดการเปลี่ยนแปลงภายในโปรเซส และคำสั่งระบบที่เพิ่มโอกาสให้เกิดการเปลี่ยนแปลงภายในโปรเซส จากนั้นจะทำการตรวจหารูปแบบการโจมตีของซอฟต์แวร์ปรสิต โดยพิจารณาจากรูปแบบการเรียกใช้งาน และความสัมพันธ์ระหว่างคำสั่งระบบเฉพาะแบบดังกล่าว

การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่เกิดจากการทำงานของซอฟต์แวร์ปรสิต โดยใช้การตรวจหาคำสั่งระบบเฉพาะแบบนั้น สามารถทำการตรวจหาได้รวดเร็ว และตรวจหาได้อย่างแม่นยำ ซึ่งรายละเอียดการทำงาน และผลการทำงานของวิธีที่นำเสนอได้ถูกแสดงไว้ในหัวข้อผลการวิเคราะห์ข้อมูล

ภาควิชา วิทยาศาสตร์คอมพิวเตอร์ ลายมือชื่อนิสิต .....
สาขาวิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก .....
ปีการศึกษา 2554 .....

5271456821 : MAJOR COMPUTER SCIENCE

KEYWORDS: RUNTIME DECTECTION / SOFTWARE MODIFICATION / SYSTEM CALLS

SATHAPORN SA-NGOUNWONG : RUNTIME DETECTION OF SOFTWARE MODIFICATION. ADVISOR: PORNSIRI MUENCHAISRI, Ph.D., 70 pp.

Runtime software modification is mostly in negative proposes for software piracy or software attack. Runtime software modification can be introduced in many ways. Software parasites are some effective way of this modification method.

We propose a method called “Restricted System calls Detecting” for detecting the modification caused by software parasite. We select group of system calls that can lead to the Runtime software modification which are called “Restricted system calls”. We categorize the Restricted system calls into three groups, Other process modification system calls, Self-modification system calls and Encouraged modification system calls. We evaluate the software parasite attack by finding the pattern of Restricted system calls calling of running software.

Our proposed method can detect the software modification caused by software parasite with high speed and accuracy that we conclude the good result in data analysis section.

Department. ……Computer Engineering…….. Student’s Signature. ……..……..

Field of Study. ……Computer Science…… Advisor’s Signature. ……...……..

Academic Year ……2011……..……

กิตติกรรมประกาศ

ขอกราบขอบพระคุณ รศ. ดร. พรศิริ หมื่นไชยศรี ที่ทำให้วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงได้ด้วยดี อาจารย์ได้ให้คำแนะนำในการทำงานทุกขั้นตอนไม่เพียงเฉพาะเนื้อหาที่เกี่ยวข้องกับงานวิจัย แต่ยังให้คำแนะนำทางด้านทัศนคติและความรับผิดชอบที่มีต่อการทำงานวิจัยซึ่งเป็นประโยชน์เป็นอย่างมาก นอกจากนี้อาจารย์ยังรับฟังปัญหาและพยายามให้ความช่วยเหลือตลอดระยะเวลาดำเนินงานวิจัย ด้วยดีเสมอมา

ขอกราบขอบพระคุณอาจารย์ทุกท่าน ที่ได้ประสิทธิประสาทวิชาความรู้ ซึ่งทำให้มีโอกาสได้ใช้ความรู้เหล่านั้นเพื่อประโยชน์แก่สังคม และครอบครัว ตลอดช่วงชีวิตที่ผ่านมา

ขอกราบขอบพระคุณ คุณพ่อสว่าง เทพแดง และ คุณแม่ประเทือง เทพแดง ที่เป็นแบบอย่างที่ดีในการดำเนินชีวิตเสมอมา อีกทั้งยังเป็นแรงบันดาลใจในการทำความดี และแสวงหาความรู้เพิ่มเติมให้กับตัวเองมากขึ้นตลอดเวลา

ขอขอบคุณครอบครัว ที่เป็นกำลังใจและให้ความช่วยเหลือทุกสิ่ง ขอขอบคุณเพื่อนๆ ที่ได้ร่วมสุขร่วมทุกข์ด้วยกันตลอดระยะเวลาที่เรียนด้วยกันมา

สารบัญ

หน้า

หน้า

หน้า

สารบัญตาราง

หน้า

สารบัญภาพ

หน้า

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน หรือ RSM (Run-time Software Modification) มีความหมายในแง่ลบเป็นส่วนใหญ่ ซึ่งสามารถถูกนำไปใช้สำหรับการละเมิดใช้งานซอฟต์แวร์ในหลากหลายรูปแบบ เช่นแก้ไขส่วนตรวจสอบลิขสิทธิ์ของซอฟต์แวร์เพื่อทำให้ซอฟต์แวร์สามารถใช้งานได้โดยไม่มีการตรวจสอบความถูกต้องของการใช้งาน หรือ ดัดแปลงแก้ไขบางส่วนของซอฟต์แวร์ เพื่อใช้สำหรับเป็นพาหะในการแพร่กระจายตัวเองของไวรัส หรือมัลแวร์เพื่อโจมตีระบบ เป็นต้น

การเปลี่ยนแปลงของซอฟต์แวร์ขณะทำงานสามารถเกิดขึ้นได้ในหลายๆ ส่วน เช่น ส่วนคำสั่ง ส่วนข้อมูล หรือส่วนโครงสร้างของซอฟต์แวร์ ซึ่งการทำงานโดยปกติทั่วไปของซอฟต์แวร์นั้น ส่วนต่างๆเหล่านี้มีการเปลี่ยนแปลงอยู่ตลอดเวลา เช่น การปรับเปลี่ยนค่าภายในตัวแปรของโปรแกรมย่อมส่งผลต่อข้อมูลภายในโปรเซสเช่นกัน หรือการจองและคืนหน่วยความจำย่อมส่งผลต่อโครงสร้างของซอฟต์แวร์ในขณะทำงานเช่นเดียวกัน ดังนั้นการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานจึงต้องทำการแยกแยะการเปลี่ยนแปลงที่เป็นปกติของซอฟต์แวร์ ออกจากการเปลี่ยนแปลงที่ไม่ปกติให้ได้อย่างถูกต้อง

ซอฟต์แวร์ปรสิต (Software parasite) เป็นหนึ่งในวิธีการโจมตีซอฟต์แวร์ขณะทำงานที่มีประสิทธิภาพ การทำงานของซอฟต์แวร์ปรสิตนั้นจะอาศัยช่องโหว่ของซอฟต์แวร์ในการแทรกตัวเองเข้าไปยังซอฟต์แวร์เป้าหมาย ซึ่งการทำงานของซอฟต์แวร์ปรสิตนั้นจะทำการเปลี่ยนแปลงโครงสร้างบางส่วนของซอฟต์แวร์เป้าหมายในขณะกำลังทำงาน เช่น หน่วยความจำ หรือ Program header เป็นหลัก ซอฟต์แวร์ปรสิตจะไม่พยายามแก้ไขส่วนที่เป็นคำสั่งของโปรเซสโดยตรง แต่จะพยายามแทรกส่วนที่เป็นปรสิตเข้ามาในรูปของการเพิ่มหน่วยความจำ หรือการเพิ่ม memory map ให้กับซอฟต์แวร์ จากนั้นซอฟต์แวร์เป้าหมายจะทำการเรียกใช้งานหน่วยความจำที่เพิ่มเข้ามาโดยอัตโนมัติ เพื่อตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ที่เกิดจากการทำงานของซอฟต์แวร์ปรสิต งานวิจัยนี้จึงให้ความสำคัญกับการตรวจหาการเปลี่ยนแปลงโครงสร้างของซอฟต์แวร์เป็นหลัก

การทำงานของซอฟต์แวร์ปรสิตนั้นอาศัยการทำงานของ Shellcode [1] ซึ่งเป็นการผนวกการใช้งานคำสั่ง Shell ไว้ภายในซอฟต์แวร์ประเภท Executable เพื่อทำงานบางอย่าง เช่นทำให้ระบบทำงานช้าลง หรือเพื่อทำให้ระบบตอบรับการเรียกใช้งานของตัวเอง เป็นต้น ซึ่งมีการทำงานของซอฟต์แวร์ปรสิตบางประเภทที่อาศัยการทำงานของคำสั่ง ptrace ในการโจมตี [19] ซึ่งไม่ได้มีการวิจัยอย่างเป็นทางการ แต่มีงานวิจัยบางงานวิจัยที่ได้นำเสนอวิธีการโจมตีของ Trojans ด้วยช่องโหว่ที่คล้ายคลึงกับซอฟต์แวร์ปรสิต [2] ซึ่งสามารถทำงานได้อย่างมีประสิทธิภาพเช่นเดียวกัน

การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน สามารถแบ่งออกได้ 2 ประเภทได้แก่ การตรวจหาการแก้ไขซอฟต์แวร์ขณะทำงานแบบรวม และการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบแยกส่วน การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบรวมนั้นเป็นวิธีที่ เครื่องมือสำหรับตรวจหาการเปลี่ยนแปลงจะถูกฝังไว้ในตัวซอฟต์แวร์ตั้งแต่ในขั้นตอนการออกแบบ ซึ่งไม่สามารถเปลี่ยนแปลงได้ในขณะทำงาน ตัวอย่างของการตรวจหาการเปลี่ยนแปลงแบบรวม ได้แก่ การใช้รหัสตรวจสอบ (Integrity value checking) [3,4] ซึ่งจะทำการแทรกรหัสตรวจสอบกระจายไว้ทั่วซอฟต์แวร์ เมื่อซอฟต์แวร์ถูกเปลี่ยนแปลง รหัสตรวจสอบจะถูกเปลี่ยนแปลงด้วย จึงทำให้ทราบถึงความเปลี่ยนแปลงที่เกิดขึ้น ตัวอย่างงานวิจัยอีกงานหนึ่งคือการทำซ้ำเพื่อตรวจสอบความเปลี่ยนแปลง[5] ซึ่งจะออกแบบให้ฟังก์ชันมีโครงสร้างที่แตกต่างกันแต่ให้ผลลัพธ์ที่เหมือนกัน เมื่อซอฟต์แวร์ทำงานระบบตรวจสอบจะนำผลลัพธ์ที่ได้มาเปรียบเทียบกันเพื่อตรวจหาว่าซอฟต์แวร์มีการเปลี่ยนแปลงหรือไม่ ข้อดีของการตรวจหาการเปลี่ยนแปลงแบบรวม คือเรียกใช้งานง่ายเพราะส่วนซอฟต์แวร์ และส่วนตรวจสอบอยู่ที่เดียวกัน อีกทั้งสามารถตรวจหาความเปลี่ยนแปลงที่เกิดขึ้นได้ดีพอสมควร แต่ข้อด้อยคือส่วนของการตรวจหาความเปลี่ยนแปลงนั้นแทรกอยู่ภายในซอฟต์แวร์โดยตรง ซึ่งทำให้ซอฟต์แวร์มีขนาดใหญ่ทำให้ใช้เวลาในการทำงานมากขึ้น และส่วนตรวจหาการเปลี่ยนแปลงนั้นสามารถถูกแก้ไขได้ ถ้าผู้โจมตีทราบตำแหน่งของส่วนที่ใช้ตรวจหาการเปลี่ยนแปลงดังกล่าว

ในส่วนการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบแยกส่วน เป็นวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่ส่วนตรวจสอบ กับส่วนซอฟต์แวร์ที่ถูกตรวจสอบแยกออกจากกันอย่างสิ้นเชิง ดังนั้นการทำงานจึงไม่รบกวนการทำงานของซอฟต์แวร์ที่ต้องการตรวจสอบ แต่อาจจะมีผลกระทบต่อการทำงานของระบบปฏิบัติการ ตัวอย่างของการทำงานแบบนี้ ได้แก่ การตรวจหาพฤติกรรมของผู้ใช้และระบบที่ผิดปกติ [6,7,8] ซึ่งกระทำโดย

เปรียบเทียบพฤติกรรมการใช้งานซอฟต์แวร์ของผู้ใช้ในสภาวะปกติ เปรียบเทียบกับพฤติกรรมการใช้งานในสภาวะผิดปกติ หรือวิธีการตรวจหาความผิดปกติภายในซอฟต์แวร์ [9,10,11] ซึ่งกระทำโดยเปรียบเทียบพฤติกรรมของซอฟต์แวร์ในสภาวะปกติเปรียบเทียบกับพฤติกรรมที่ไม่ปกติ ซึ่งพฤติกรรมที่เลือกใช้อาจจะเป็นคำสั่งระบบ หรือ Call stack เป็นต้น ตัวอย่างการตรวจหาการเปลี่ยนแปลงฯ แบบเปลี่ยนแปลงได้ที่น่าสนใจอีกตัวอย่างหนึ่งคือ การวิเคราะห์ลำดับการเรียกใช้งานของคำสั่งระบบ [12,13] ซึ่งกระทำโดยเปรียบเทียบลำดับการเรียกใช้งานคำสั่งระบบ ของระบบปฏิบัติการทั้งหมดเปรียบเทียบกับฐานข้อมูลพฤติกรรมของการโจมตีในรูปแบบต่างๆ โดยภาพรวมของวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบเปลี่ยนแปลงได้นั้นสามารถตรวจหาความเปลี่ยนแปลงที่เกิดขึ้นได้อย่างมีประสิทธิภาพพอสมควร แต่เนื่องจากการตรวจหาการเปลี่ยนแปลงนั้นไม่สามารถกระทำได้ทันทีเพราะจะต้องมีการเก็บข้อมูลในสภาวะปกติของระบบเสียก่อน ซึ่งทำให้การทำงานใช้เวลาค่อนข้างมาก และเนื่องจากระบบที่ทำหน้าที่ตรวจหาความเปลี่ยนแปลงทำงานแยกกับซอฟต์แวร์ที่ถูกตรวจสอบ จึงมีความยุ่งยากในการติดตั้งและปรับแต่งได้อย่างมีประสิทธิภาพ

จากปัญหาของวิธีการตรวจหาความเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน ที่ได้กล่าวไปแล้วข้างต้น งานวิจัยนี้จึงนำเสนอวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานโดยการตรวจหาคำสั่งระบบเฉพาะแบบ หรือ RSD ซึ่งสามารถทำการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ที่เกิดขึ้นจากการทำงานของซอฟต์แวร์ปรสิตได้ทันที โดยไม่ต้องใช้ฐานข้อมูลในการทำงานเพื่อทำให้การทำงานรวดเร็วมากขึ้น นอกจากนี้งานวิจัยนี้ยังให้ความสำคัญกับความสะดวกในการใช้งาน โดยระบบที่นำเสนอจะต้องสามารถติดตั้งได้โดยง่ายและสามารถปรับแต่งการทำงานได้ไม่ยาก ซึ่งรายละเอียดของการออกแบบ และการนำเสนอจะกล่าวโดยละเอียดในส่วนต่อไป

1.2 วัตถุประสงค์ของการวิจัย

เพื่อออกแบบวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่เกิดจากการโจมตีของซอฟต์แวร์ปรสิตที่อาศัยการเรียกใช้งานคำสั่ง ptrace โดยวิธีการตรวจหาที่นำเสนอต้องสามารถทำงานได้รวดเร็ว โดยไม่ต้องอาศัยการเก็บข้อมูลซอฟต์แวร์ในสภาวะปกติก่อนการทำงาน

1.3 ขอบเขตของการวิจัย

1. งานวิจัยนี้นำเสนอวิธีการตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ขณะทำงานที่เกิดขึ้นจากการโจมตีของซอฟต์แวร์ปรสิตที่ทำงานด้วยคำสั่ง ptrace โดยการวิเคราะห์การเรียกใช้งานคำสั่งระบบ

2. การตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ กระทำเฉพาะในระดับโปรเซส ซึ่งไม่รวมถึงการตรวจหาการเปลี่ยนแปลงการทำงานของซอฟต์แวร์ในระดับล่าง อย่างเช่น เคอร์เนล หรือโมดูล

3. การทดสอบการทำงานของแนวคิดที่นำเสนอ กระทำโดยการจำลองการเปลี่ยนแปลงให้เกิดขึ้นกับซอฟต์แวร์ในขณะทำงานในสภาพแวดล้อมที่กำหนด ซึ่งไม่รวมถึงการทดสอบการใช้งานจริง

4. ประเมินผลกระทบของแนวคิดที่นำเสนอ ที่มีต่อระบบปฏิบัติการ ด้วยวิธีการทดสอบภาระ (Load testing) โดยประเมินผลกระทบในแง่ของความเร็วในการทำงานเป็นหลัก

5. ใช้ระบบปฏิบัติการลินุกซ์เป็นระบบปฏิบัติการหลักในการวิจัย และใช้ภาษา C หรือ C++ ในการพัฒนาซอฟต์แวร์ที่ใช้ทดสอบในงานวิจัย

6. ระบบที่นำเสนอจะทดสอบบนระบบปฏิบัติการลินุกซ์ Ubuntu เคอร์เนล 2.6 เท่านั้น

1.4 ประโยชน์ที่คาดว่าจะได้รับ

1. ช่วยลดความเสียหายที่เกิดจากการโจมตีของซอฟต์แวร์ปรสิตที่อาจเกิดขึ้นให้น้อยลง

2. การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่ทำงานได้รวดเร็ว และมีผลกระทบต่อการทำงานโดยรวมของระบบปฏิบัติการน้อยลง

3. วิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่สามารถติดตั้งและปรับแต่งการใช้งานได้ง่ายขึ้น โดยไม่ต้องสร้างซอฟต์แวร์เพื่อใช้งานด้านความปลอดภัยโดยเฉพาะ ซึ่งช่วยลดเวลาในการติดตั้ง และพัฒนาลง

4. ลดค่าใช้จ่ายที่ใช้ในการเพิ่มความปลอดภัยให้กับระบบที่มีอยู่ได้ เนื่องจากสามารถประยุกต์ใช้ซอฟต์แวร์ Opensource ในการทำงานได้ทั้งหมด

5. เป็นแนวทางในการศึกษาพัฒนา เพื่อให้เกิดการต่อยอดความรู้ในการสร้างเครื่องมือสำหรับตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานให้มีประสิทธิภาพสูงขึ้น

1.5 วิธีดำเนินการวิจัย

1. ศึกษางานวิจัยที่เกี่ยวข้องกับการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน และศึกษาวิธีการทำงานของซอฟต์แวร์ปรสิต

2. ออกแบบวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน และค้นคว้าแหล่งข้อมูลสำหรับสร้างซอฟต์แวร์ปรสิต

3. สร้างซอฟต์แวร์ปรสิต เพื่อใช้ทดสอบวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานตามแนวคิดที่นำเสนอ

4. สร้างซอฟต์แวร์สำหรับตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานตามแนวคิดที่ออกแบบ

5. ทดสอบการทำงานของซอฟต์แวร์ที่สร้างขึ้น พร้อมประเมินประสิทธิภาพของการทำงาน และประเมินผลกระทบที่มีต่อระบบปฏิบัติการ

6. สรุปผลการทำงาน

7. เรียบเรียงวิทยานิพนธ์

1.6 คำจำกัดความที่ใช้ในการวิจัย

RSD: Restricted System-calls Detecting หมายถึง วิธีการตรวจหาการเปลี่ยนแปลงที่เกิดขึ้นกับซอฟต์แวร์ในขณะทำงาน ด้วยการตรวจหารูปแบบการเรียกใช้งานคำสั่งระบบเฉพาะแบบ

RSM: Runtime Software Modification หมายถึง การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

NRSM: Negative Runtime Software Modification หมายถึง การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่อาจส่งผลในแง่ลบต่อการทำงานของซอฟต์แวร์ หรือระบบโดยรวม

Systemtap: หมายถึง ซอฟต์แวร์ที่ทำหน้าที่ในการตรวจดูการทำงานภายในเคอร์เนลของระบบปฏิบัติการ เพื่อประเมินประสิทธิภาพ หรือปัญหาในการทำงานของเคอร์เนล

บทที่ 2
เอกสารและงานวิจัยที่เกี่ยวข้อง

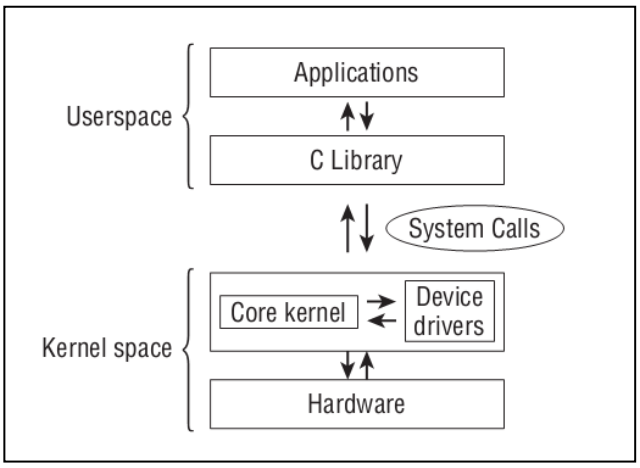
ในบทนี้กล่าวถึงความรู้จำเป็นพื้นฐานที่ใช้ในงานวิจัย ซึ่งประกอบด้วยเครื่องมือและซอฟต์แวร์ที่ใช้ในการวิจัยเป็นสำคัญ อีกทั้งยังกล่าวถึงงานวิจัยที่เกี่ยวข้องกับการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานในแง่มุมต่างๆ ตัวอย่างเช่น การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานวิธีต่างๆ การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานวิธีต่างๆที่ได้มีการวิจัยหรือมีการพัฒนาไปแล้ว เพื่อใช้ความรู้จากงานวิจัยที่เกี่ยวข้องเหล่านี้เป็นตัวกำหนดแนวทางในการวิจัยและเพื่อกำหนดประเด็นปัญหาได้ตรงจุดมากขึ้น รายละเอียดของเอกสารและงานวิจัยที่เกี่ยวข้องมีดังต่อไปนี้

2.1 แนวคิดและทฤษฎี

ในส่วนนี้จะกล่าวถึงโครงสร้าง และการทำงานพื้นฐานของระบบปฏิบัติการลินุกซ์ ซึ่งเป็นตัวอย่างของระบบปฏิบัติการที่มีโครงสร้างการจัดการแบบรวมศูนย์ จึงทำให้เกิดแนวคิดในการตรวจหาความเปลี่ยนแปลงของซอฟต์แวร์โดยการเฝ้าดูจากส่วนกลาง หรือเคอร์เนลเป็นหลัก และเนื่องจากงานวิจัยนี้เป็นการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานในระดับโปรเซสด้วยการวิเคราะห์คำสั่งระบบ ดังนั้นจึงควรกล่าวถึงโครงสร้าง และการทำงานของโปรเซส และคำสั่งระบบโดยสังเขปด้วยเช่นกัน นอกจากนี้ยังได้กล่าวถึงการทำงานโดยสังเขปของ Systemtap ที่ใช้เป็นเครื่องมือในการตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ในขณะทำงาน รายละเอียดของแต่ละส่วนมีดังต่อไปนี้

2.1.1 ระบบปฏิบัติการลินุกซ์ (GNU Linux OS)

ระบบปฏิบัติการลินุกซ์ หรือ GNU Linux เป็นระบบปฏิบัติการที่คล้ายคลึงกับระบบปฏิบัติการยูนิกซ์ (Unix-like Operating System) ซึ่งเป็นระบบปฏิบัติการที่ได้รับความสนใจจากนักพัฒนา และนักวิจัยทั่วไปเป็นอย่างมากด้วยเหตุที่ระบบปฏิบัติการลินุกซ์เป็นซอฟต์แวร์ Opensource ทำให้นักพัฒนาสามารถดัดแปลงแก้ไขเพิ่มเติมความสามารถต่างๆ ได้อย่างอิสระ ซึ่งทำให้เกิดการต่อยอดความรู้ และเกิดเครือข่ายในการพัฒนาปรับปรุงระบบปฏิบัติการลินุกซ์ให้มีประสิทธิภาพสูงขึ้นเป็นลำดับ



ภาพที่ 2.1.1 โครงสร้างโดยรวมของระบบปฏิบัติการลินุกซ์ [14]

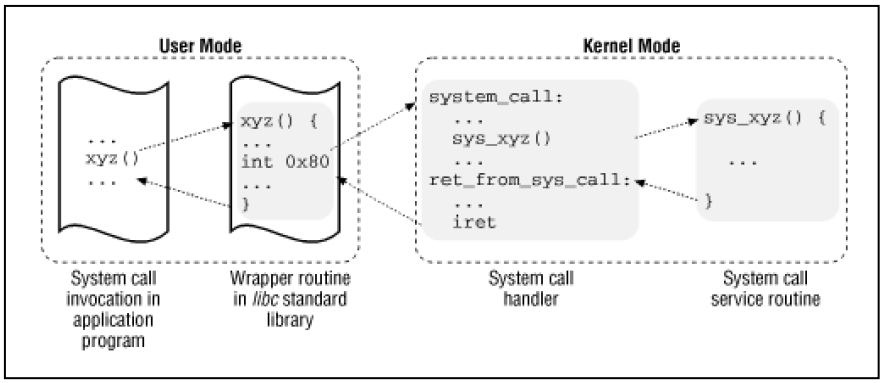
ภาพที่ 2.1.1 แสดงโครงสร้างระดับการทำงานบนระบบปฏิบัติการลินุกซ์ แบ่งได้เป็น 2 ระดับ คือระดับเคอร์เนล (Kernel space) และ ระดับผู้ใช้งาน (User space) ซึ่งความแตกต่างระหว่าง 2 ระดับการทำงานนี้คือ สิทธิในการใช้งานทรัพยากรต่างๆของระบบ โปรแกรมที่ทำงานอยู่ในระดับผู้ใช้งานจะไม่สามารถเรียกใช้งานทรัพยากรต่างๆ ได้โดยตรง แต่จะต้องเรียกใช้งานผ่านคำสั่งระบบเท่านั้น ในทางตรงข้ามถ้าโปรแกรมทำงานอยู่ในระดับเคอร์เนล จะสามารถเรียกใช้งานทรัพยากรต่างๆของระบบได้โดยตรง และซอฟต์แวร์ที่ถูกสร้างขึ้นมาเพื่อทำงานในระดับเคอร์เนลนั้นจะถูกเชื่อมต่อกับเคอร์เนลโดยตรงซึ่งทำให้การทำงานโดยรวมจะมีความเร็วเพิ่มขึ้น ในส่วนของระดับผู้ใช้งานจะเป็นพื้นที่สำหรับทำงาน แอพพลิเคชั่นทั่วไป ซึ่งรวมถึงไลบรารี่ต่างๆที่จำเป็นขณะทำงาน (Runtime library) ในส่วนของระดับเคอร์เนล ประกอบด้วย 2 ส่วนหลัก คือ เคอร์เนล (Core kernel) และ โมดูล (Module) หรือ Device drivers โดยเคอร์เนลเป็นส่วนหลักของระบบปฏิบัติทำหน้าที่จัดการการทำงานของโปรเซสทั้งหมด เคอร์เนลเป็นซอฟต์แวร์ประเภทโมโนลิทิค (Monolithic) ซึ่งช่วยให้ระบบปฏิบัติการทำงานได้เร็วขึ้น โดยเคอร์เนลจะให้บริการการใช้งานพื้นฐานทุกอย่างที่โปรเซสต้องการ เช่น การสร้างโปรเซส การจัดลำดับโปรเซสเพื่อเข้าใช้งานซีพียู การจัดสรรหน่วยความจำ และการติดต่อกับฮาร์ดแวร์ต่างๆ เป็นต้น ส่วนโมดูล ทำหน้าที่เสมือนส่วนเสริมเพื่อเพิ่มความสามารถให้กับเคอร์เนลเพื่อรองรับการทำงานได้หลากหลายมากขึ้น หรือสามารถทำให้เคอร์เนลสามารถรู้จักฮาร์ดแวร์ได้มากขึ้น ตัวอย่างของโมดูลได้แก่ Device drivers ต่างๆ และ Process file system (/proc) เป็นต้น โดยโมดูลเหล่านี้สามารถนำเข้า หรือนำออกจากเคอร์เนล ได้โดยผ่านคำสั่งระบบ

ในปัจจุบันระบบปฏิบัติการลินุกซ์ได้ถูกใช้อย่างแพร่หลายในอุปกรณ์หลายประเภท ตั้งแต่โทรศัพท์มือถือไปจนถึงระบบเซอร์เวอร์ขนาดใหญ่

2.1.2 **คำสั่งระบบ** (System calls)

คำสั่งระบบ เป็นคำสั่งพื้นฐานที่บรรจุไว้ภายในระบบปฏิบัติการเพื่อให้โปรเซสต่างๆ ใช้ติดต่อกับเคอร์เนลภายในของระบบปฏิบัติการ เพื่อร้องขอการทำงานต่างๆ จากเคอร์เนล เช่น การจองหน่วยความจำ การสร้างโปรเซสใหม่ และการติดต่อสื่อสารระหว่างโปรเซส เป็นต้น จากภาพที่ 2.1.1 คำสั่งระบบจะอยู่ระหว่าง ส่วนผู้ใช้ และ ส่วนเคอร์เนล ซึ่งสามารถกล่าวได้ว่าคำสั่งระบบทำหน้าที่เป็นส่วนเชื่อมระหว่าง ส่วนผู้ใช้ กับ ส่วนเคอร์เนล

การทำงานของทุกโปรเซสจะต้องอาศัยความช่วยเหลือของเคอร์เนลในการทำงาน และการเรียกใช้งานเคอร์เนลทุกอย่างต้องกระทำผ่านคำสั่งระบบเท่านั้น ดังนั้นจึงสามารถใช้คำสั่งระบบเป็นเครื่องมือสำหรับตรวจดูพฤติกรรมของแต่ละโปรเซสได้

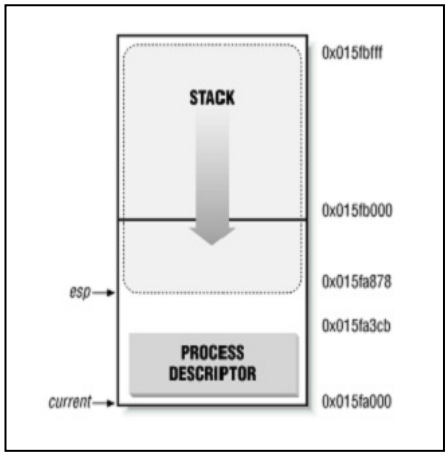


ภาพที่ 2.1.2 การเรียกใช้งานฟังก์ชันในเคอร์เนลผ่านคำสั่งระบบของเคอร์เนล [15]

ภาพที่ 2.1.2 แสดงตัวอย่างการเรียกใช้งานบริการต่างๆ ภายในเคอร์เนลของโปรเซส โดยผ่านทางคำสั่งระบบ จะมีการสลับโหมดการทำงานระหว่างโหมดผู้ใช้งานและโหมดเคอร์เนลของระบบปฏิบัติการ ซึ่งรวมไปถึง อินเตอร์รัพ (Interrupt) ต่างๆ จะอาศัยวิธีการเดียวกันนี้ในการทำงาน

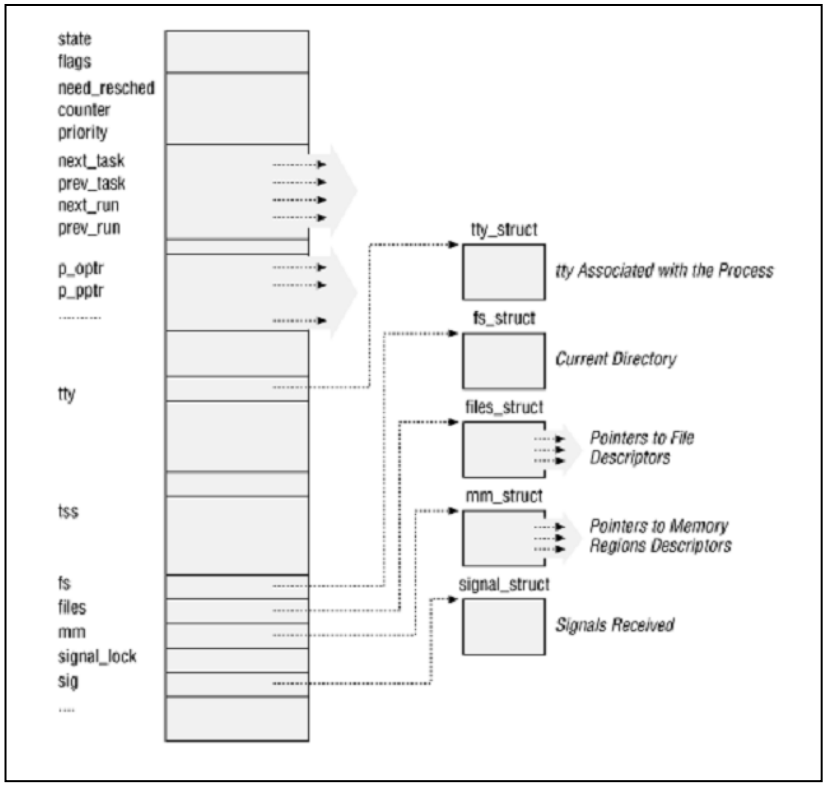
2.1.3 **โปรเซส** (Process)

โปรเซส หรือ ทาส์ก หมายถึงซอฟต์แวร์ที่กำลังทำงานอยู่บนระบบปฏิบัติการ โปรเซสเป็นหน่วยพื้นฐานที่สุดที่ทำให้เกิดงานขึ้นในระบบปฏิบัติการทั้งหมด โปรเซสถูกออกแบบมาให้สามารถทำงานแบบหลายทาส์ก (Multi-tasking) ซึ่งโปรเซสจะต้องถูกสวิตซ์ไปมาในระหว่างรอบการทำงานของ CPU ดังนั้นการทำงานของโปรเซสต้องอาศัยส่วนอธิบายโปรเซส (Process Descriptor) ซึ่งเป็นโครงสร้างข้อมูลที่ใช้อธิบายรายละเอียดต่างๆภายในโปรเซส โดยระบบปฏิบัติการจะใช้ส่วนอธิบายโปรเซสเป็นป้ายบอกสถานะของแต่ละโปรเซส ซึ่งส่วนอธิบายโปรเซสของระบบปฏิบัติการลินุกซ์ ถูกวางไว้ในตำแหน่งหน่วยความจำดังภาพที่ 2.1.3-1



ภาพที่ 2.1.3-1 ตำแหน่งของส่วนอธิบายโปรเซส

รายละเอียดของแต่ละส่วนภายในส่วนอธิบายโปรเซส สามารถศึกษาเพิ่มเติมได้ใน [14,15] โครงสร้างโดยสังเขปของส่วนอธิบายโปรเซสในระบบปฏิบัติการลินุกซ์ สามารถแสดงได้ดังภาพที่ 2.1.3-2



ภาพที่ 2.1.3-2 โครงสร้างส่วนอธิบายโปรเซสของระบบปฏิบัติการลินุกซ์

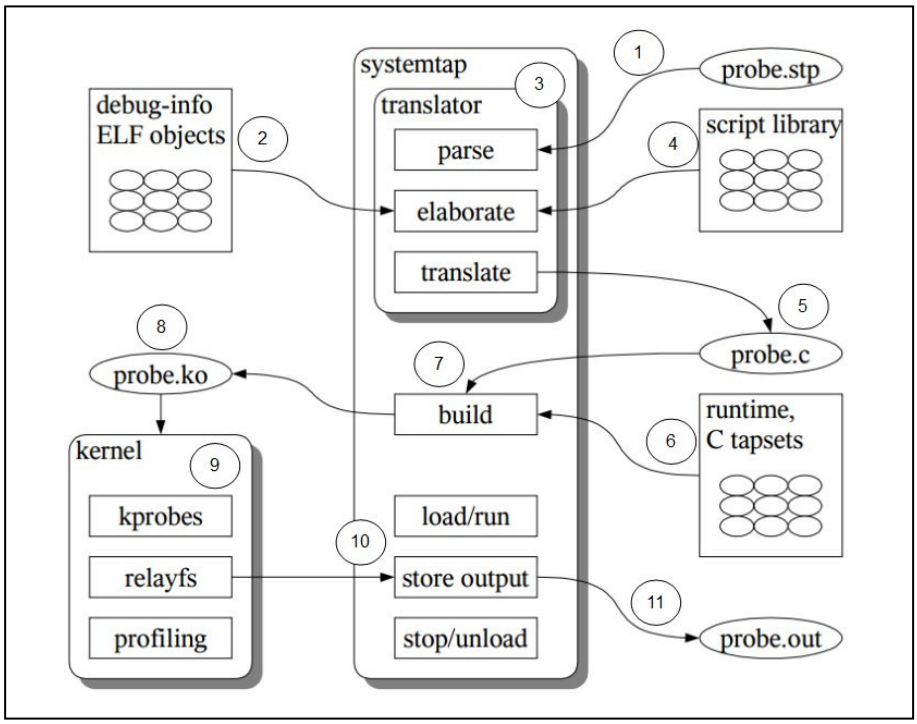
จากภาพที่ 2.1.3-2 ส่วนอธิบายโปรเซส ประกอบด้วยข้อมูลย่อยอีกหลายส่วน เช่น ข้อมูลแสดงสถานะของโปรเซส (state flags) ข้อมูลที่ชี้ไปยังโปรเซสก่อนและหลังโปรเซสตัวเอง (prev_task, next_task) และ ข้อมูลชี้ไปยังหน่วยความจำที่โปรเซสใช้ทั้งหมด (mm) เป็นต้น ข้อมูลชี้ (pointer) ส่วนใหญ่จะชี้ไปยังโครงสร้างข้อมูลอื่นต่อไป เช่น ข้อมูลชี้ไปยังหน่วยความจำจะชี้ไปยังโครงสร้างข้อมูลหน่วยความจำ (mm_struct) ที่มีการแบ่งขนาดเป็นบล๊อก และแต่ละบล็อกจะชี้ไปยังบล็อกต่อไปจนถึงบล็อกสุดท้าย เป็นต้น

2.1.4 Systemtap

Systemtap เป็นซอฟต์แวร์สำหรับตรวจสอบ และติดตามกิจกรรมการทำงานภายในของเคอร์เนลของระบบปฏิบัติการลินุกซ์ เพื่อใช้วิเคราะห์ปัญหาด้านประสิทธิภาพ และปัญหาความผิดปกติในการใช้งานทั่วไป Systemtap จะทำงานร่วมกับ Kprobe[*] ซึ่งจะอาศัยข้อมูล

[*] http://sourceware.org/systemtap/kprobes/

ELF debug-info ภายในโมดูลต่างๆที่ทำงานอยู่ภายในเคอร์เนล ซึ่งจะไม่ถูกผนวกมาในเคอร์เนลเวอร์ชั่นทั่วไป (generic version) ดังนั้นการทำให้ลินุกซ์สามารถใช้งาน Systemtap ได้ ผู้ใช้จะต้องทำการคอมไพล์เคอร์เนลใหม่ โดยกำหนดให้เคอร์เนลสนับสนุนการใช้งานข้อมูล debug-info รายละเอียดของการคอมไพล์เคอร์เนลสามารถศึกษาเพิ่มเติมได้จาก ภาคผนวก ง การทำงานของ Systemtap สามารถแสดงดังภาพที่ 2.1.4-1



ภาพที่ 2.1.4-1 โครงสร้างและการทำงานของ Systemtap [16]

จากภาพที่ 2.1.4-1 หลังจาก Compile เคอร์เนลแล้วจะได้ข้อมูล debug-info (2) ซึ่งเป็นข้อมูลแสดงการทำงานโมดูลต่างๆ ภายในเคอร์เนล จากนั้นผู้ใช้จึงสามารถเริ่มต้นการใช้งาน Systemtap โดยสร้างไฟล์ probe.stp (1) ซึ่งเป็นตัวอย่างของ Systemtap script ที่ทำงานบางอย่างในการตรวจจับการทำงานภายในเคอร์เนล จากนั้นผู้ใช้งานเรียกใช้งาน Systemtap เพื่ออ่าน script ดังกล่าวโดยใช้คำสั่ง

```
sudo stap probe.stp
```

หลังจากนั้น Systemtap จะทำการแปลความหมายคำสั่งใน probe.stp ด้วย Translator (3) ซึ่งจะแปลคำสั่งโดยอาศัยข้อมูล debug-info (2) ร่วมกับข้อมูล script library (4) เพื่อสร้างไฟล์ probe.c ซึ่งเป็นไฟล์ภาษา C และจำเป็นในการสร้างโมดูลสำหรับใช้งานในส่วนต่อไป จากนั้น Systemtap จะอ่าน probe.c ด้วยส่วน build (7) เพื่อสร้างโมดูล probe.ko (8) โดยอาศัยข้อมูล tapsets C runtime (6) ในการสร้างโมดูลดังกล่าว เมื่อได้โมดูลแล้ว Systemtap จะผนวกโมดูลเข้ากับเคอร์เนล (9) ซึ่งภายในเคอร์เนลจะมีการฝัง kprobe ไว้ตามส่วนต่างๆ อยู่แล้วโดยปริยาย ซึ่งข้อมูลการทำงานของเคอร์เนลทุกอย่างจะถูกเก็บไว้ในส่วน relayfs จากนั้น Systemtap จะอ่านข้อมูลจาก relayfs ดังกล่าวด้วยส่วน store output (10) เพื่อนำมาสร้างเป็น probe.out (11) เพื่อใช้ในการแสดงผลความเปลี่ยนแปลงที่ตรวจพบ ซึ่งขั้นตอนทั้งหมดนี้จะเกิดหลังจากเรียกใช้งานคำสั่ง stap เพียงครั้งเดียว รายละเอียดการเขียน Systemtap script ถูกอธิบายไว้ใน [17]

2.2 งานวิจัยที่เกี่ยวข้อง

งานวิจัยที่เกี่ยวข้องกับการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานนั้นสามารถแบ่งได้เป็น 2 กลุ่มใหญ่ๆ คือ การเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน และการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน

ซึ่งการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานสำหรับงานวิจัยนี้ถูกใช้เพื่อเป็นเครื่องมือสำหรับทดสอบการตรวจหาการเปลี่ยนแปลงที่ได้นำเสนอ ในส่วนของงานวิจัยที่เกี่ยวข้องกับการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานนั้นเป็นงานวิจัยที่เกี่ยวข้องกับงานวิจัยนี้โดยตรง เพื่อศึกษาถึงการทำงานของวิธีการตรวจหาที่มีอยู่ในปัจจุบัน ว่ามีข้อดีข้อเสียอย่างไร เพื่อจะได้นำมาวิเคราะห์และหาแนวทางปรับปรุง และพัฒนาให้มีประสิทธิภาพมากขึ้น ซึ่งรายละเอียดงานวิจัยที่เกี่ยวข้องมีดังต่อไปนี้

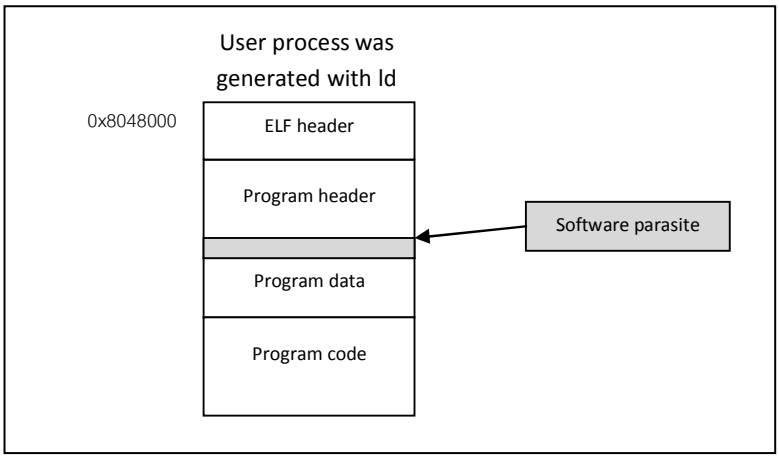
2.2.1 การเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

2.2.1.1 ซอฟต์แวร์ปรสิต (Software parasite)

การทำซอฟต์แวร์ปรสิตเป็นวิธีการแพร่กระจายซอฟต์แวร์ประเภทมัลแวร์ หรือไวรัส ไปยังซอฟต์แวร์เป้าหมายซึ่งมีลักษณะการทำงานคล้ายปรสิต ที่ต้องอาศัยโปรแกรมอื่นเป็นที่อยู่และทำงานบางอย่างของตัวเอง ลักษณะการทำงานของซอฟต์แวร์ปรสิตส่วนใหญ่จะใช้การทำงานแบบ Shellcode [1] ซอฟต์แวร์ปรสิตนั้นมีลักษณะเช่นเดียวกับไวรัส หรือมัลแวร์ที่

ประกอบด้วย 2 ส่วน ส่วนแรกคือ ซอฟต์แวร์ที่ทำหน้าที่เป็นปรสิต ซึ่งจะอยู่ในรูปของไลบรารี่เป็นส่วนใหญ่ ส่วนที่สองคือซอฟต์แวร์ที่ทำหน้าที่แพร่ปรสิตไปยังซอฟต์แวร์เป้าหมาย

ส่วนของซอฟต์แวร์ที่ทำหน้าที่เป็นปรสิต ถูกออกแบบมาเพื่อให้ทำงานอย่างใดอย่างหนึ่งเพื่อบุกรุกหรือขโมยข้อมูล เช่น ค้นหาบัญชีรายชื่อผู้ใช้งานเมล์ หรือ ค้นหาข้อมูลรหัสผ่านของผู้ใช้ เป็นต้น ซอฟต์แวร์ที่เป็นปรสิตนี้จะถูกวางไว้ในตำแหน่งไลบรารี่ที่ใช้ร่วมกันทั้งระบบ (Shared library) เพื่อให้สามารถเรียกใช้งานได้โดยง่าย ในส่วนซอฟต์แวร์ที่ทำหน้าที่แพร่ปรสิตเป็นซอฟต์แวร์ หรือโปรแกรมที่สามารถเรียกใช้งานได้โดยตรง การทำงานของซอฟต์แวร์แพร่ปรสิตนั้นอาศัยช่องโหว่ในการทำงานของคำสั่ง ld (Loader command) ร่วมกับโครงสร้างของไฟล์ ELF (Executable and Linkable Format) ซึ่งเป็นรูปแบบไฟล์มาตราฐานที่ระบบปฏิบัติการลินุกซ์ใช้สำหรับสร้างโปรแกรม ประเด็นสำคัญอยู่ที่เมื่อไฟล์ ELF ถูกสร้างด้วยคำสั่ง ld ในขณะถูกเรียกใช้งานไฟล์นั้นจะถูกวางไว้ที่ตำแหน่งหน่วยความจำเดิมเสมอ [18] ซึ่งทำให้ซอฟต์แวร์ปรสิตสามารถรู้โครงสร้างต่างๆ ของโปรแกรม เช่น ตำแหน่งของ Program header ตำแหน่งที่เป็นข้อมูล รวมถึงตำแหน่งของคำสั่งต่างๆ ภายในโปรเซส จากนั้นตัวแพร่ซอฟต์แวร์ปรสิตจะใช้คำสั่ง ptrace เพื่อดีบักโปรแกรมเป้าหมาย และทำการแทรกหรือแก้ไขบางส่วนของโปรแกรมเพื่อให้ทำการเรียกใช้งานซอฟต์แวร์ปรสิตต่อไป



ภาพที่ 2.2.1.1 การทำงานของซอฟต์แวร์ปรสิต

ภาพที่ 2.2.1.1 แสดงการทำงานของซอฟต์แวร์ปรสิตโดยสังเขป โดยเมื่อไฟล์ ELF ถูกเรียกใช้งานไฟล์จะถูกวางไว้ ณ ตำแหน่ง 0x8048000 ของหน่วยความจำในโปรเซสเสมอ จากนั้นซอฟต์แวร์ปรสิตจึงทำการค้นหาไปยังส่วนต่างๆ ของโปรแกรมเพื่อทำการแทรกส่วนที่เป็นปรสิตเข้าไปยังโปรแกรม ซึ่งในภาพที่ 2.2.1.1 นั้นซอฟต์แวร์ปรสิตทำการแก้ไขส่วนที่เป็น Program

header เพื่อแทรกชื่อของไลบรารี่ที่ต้องการให้โปรแกรมเรียกใช้งาน หลังจากที่ปรสิตสามารถแพร่เข้าไปยังซอฟต์แวร์เรียบร้อยแล้ว

การทำซอฟต์แวร์ปรสิตนี้ไม่มีรูปแบบการวิจัยอย่างเป็นทางการแต่สามารถศึกษารายละเอียดเพิ่มเติมได้จาก [18,19]

2.2.2 การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

2.2.2.1 การตรวจสอบลายเซ็น (Signature based)

เป็นการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานแบบรวม ส่วนซอฟต์แวร์กับส่วนตรวจสอบไว้ด้วยกัน สามารถทำได้หลายวิธี เช่น การแทรกรหัสตรวจสอบ หรือค่าคงที่ตรวจสอบเฉพาะ ไว้ในส่วนต่างๆของซอฟต์แวร์ [3,4] เมื่อซอฟต์แวร์ถูกแก้ไข หรือมีการเปลี่ยนแปลงเกิดขึ้นกับบางส่วนของซอฟต์แวร์ ที่ส่งผลให้ค่าตรวจสอบเหล่านี้เปลี่ยนแปลง ซึ่งทำให้ส่วนตรวจสอบสามารถตรวจพบความเปลี่ยนแปลงที่เกิดขึ้นกับซอฟต์แวร์ได้ทันที หรืออีกตัวอย่างของการตรวจสอบลายเซ็น คือการสร้างฟังก์ชันคู่ [5] โดยออกแบบให้ซอฟต์แวร์มีฟังก์ชันในการทำงาน 2 รูปแบบที่มีโครงสร้าง หรือใช้คำสั่งที่แตกต่างกันแต่ได้ผลลัพธ์ของฟังก์ชันเหมือนกัน ซึ่งถ้ามีการเปลี่ยนแปลงเกิดขึ้นกับฟังก์ชันใดฟังก์ชันหนึ่ง ส่วนตรวจหาการเปลี่ยนแปลงจะสามารถตรวจพบได้โดยเปรียบเทียบผลลัพธ์การทำงานของฟังก์ชันทั้งสองทันที

ข้อดีของการตรวจหาการเปลี่ยนแปลงด้วยวิธีนี้คือส่วนซอฟต์แวร์ และส่วนตรวจหาการเปลี่ยนแปลงอยู่ในส่วนเดียวกันซึ่งทำให้การเรียกใช้งานสามารถทำได้ง่าย แต่ข้อด้อยคือ ซอฟต์แวร์มีส่วนการทำงานที่ไม่จำเป็น และมีขนาดใหญ่ขึ้นเนื่องจากการแทรกรหัสตรวจสอบไว้ตามส่วนต่างๆของซอฟต์แวร์ ซึ่งส่งผลกระทบต่อการทำงานที่ช้าลงของซอฟต์แวร์โดยตรง อีกทั้งการตรวจสอบที่ถูกแทรกไว้เหล่านี้ สามารถถูกแก้ไขได้ถ้าผู้โจมตีทราบตำแหน่งของรหัสตรวจสอบดังกล่าว

2.2.2.3 การวิเคราะห์พฤติกรรมของซอฟต์แวร์ (Anomaly based หรือ Software behavior based)

เป็นการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานโดยอาศัยข้อมูลพฤติกรรมในสภาวะปกติของซอฟต์แวร์ โดยนำไปเปรียบเทียบกับข้อมูลพฤติกรรมในซอฟต์แวร์

ทำงาน ถ้ารูปแบบของข้อมูลมีการเปลี่ยนแปลงไปจากข้อมูลในสภาวะปกติสามารถสันนิษฐานเบื้องต้นได้ว่ามีบางอย่างผิดปกติ ซึ่งจะต้องนำความผิดปกติที่ได้นั้นนำมาวิเคราะห์อีกครั้งว่าเป็นความผิดปกติจริงๆ หรือเป็นฐานข้อมูลใหม่ เป็นต้น ข้อมูลที่ใช้สำหรับอ้างอิงพฤติกรรมของซอฟต์แวร์นี้สามารถใช้ได้หลากหลายประเภท เช่น ข้อมูล Call stack [11] หรือข้อมูลตัวแปรต่างๆ ภายในโปรเซส [6] เพื่อตรวจสอบว่าซอฟต์แวร์ว่ามีการใช้งานทรัพยากรที่ผิดปกติหรือไม่ เป็นต้น

ข้อด้อยของการทำการตรวจหาแบบนี้คือ ไม่สามารถทำการตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ได้ทันที เพราะต้องใช้เวลาในการจัดทำฐานข้อมูลสำหรับซอฟต์แวร์แต่ละชนิด และใช้เวลานานในการประมวลผลการเปลี่ยนแปลงที่เกิดขึ้น อีกทั้งข้อมูลในฐานข้อมูลอาจไม่ครอบคลุมการตรวจหาได้ทุกกรณี เนื่องจากการทำงานของซอฟต์แวร์นั้นไม่สามารถตรวจสอบได้ว่าทำงานครบทุกส่วน ครบทุกกรณี หรือครบทุกสภาพแวดล้อมหรือไม่

2.2.2.4 การวิเคราะห์พฤติกรรมโดยรวมของระบบ (Log analysis หรือ System behavior based)

เป็นการตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ โดยอาศัยการวิเคราะห์ข้อมูลจากประวัติการทำงานต่างๆ ของระบบ (System log) ซึ่งประวัติการทำงานนี้อาจเป็น ข้อมูลประวัติพฤติกรรมการใช้งานของผู้ใช้ [7] ซึ่งจะทำการเก็บประวัติการใช้งานซอฟต์แวร์ ประวัติการติดต่อกับระบบภายในเครือข่ายของผู้ใช้แต่ละคน หรือ ข้อมูลการเขียนไฟล์ของผู้ใช้งานแต่ละคน [8] ลงบนระบบปฏิบัติการ หรือประวัติจำนวนของซอฟต์แวร์ที่ทำงานบนระบบ หรือประวัติข้อมูลเครือข่าย (packet) ที่ติดต่อระหว่างเครื่องคอมพิวเตอร์ เป็นต้น ซึ่งเมื่อมีความผิดปกติเกิดสามารถตรวจสอบการใช้งานที่ผิดปกติต่างๆได้

ข้อด้อยของการตรวจหาแบบนี้คือไม่สามารถตรวจหาการเปลี่ยนแปลงได้ทันที จะต้องมีการเก็บข้อมูลประวัติให้มากพอที่จะสามารถใช้ในการตัดสินใจได้ว่าการเกิดความผิดปกติขึ้นหรือไม่ อีกทั้งการวิเคราะห์รูปแบบของการเปลี่ยนแปลงที่เกิดขึ้นต้องใช้เวลาในการประมวลผลค่อนข้างมาก

บทที่ 3
การออกแบบ RSD

ในบทนี้จะกล่าวถึงภาพรวมในการออกแบบ และวิธีการสร้าง RSD ทั้งหมด โดยนำเสนอตั้งแต่การเลือกใช้เครื่องมือ แนวคิดที่ใช้ในการออกแบบวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ไปจนถึงการนำเสนอวิธีการตรวจหาฯ ดังกล่าว นอกจากนี้ยังได้นำเสนอวิธีการประเมินประสิทธิภาพของวิธีการที่นำเสนอว่ามีผลกระทบต่อการทำงานของระบบปฏิบัติการอย่างไร รายละเอียดการออกแบบทั้งหมดมีดังต่อไปนี้

3.1 เครื่องมือสำหรับใช้ในงานวิจัย

เครื่องมือที่ใช้ในงานวิจัยนี้ประกอบด้วย 3 ส่วน ส่วนแรกได้แก่ระบบปฏิบัติการซึ่งถือเป็นส่วนหลักในการกำหนดทิศทางของงานวิจัย ส่วนต่อมาคือเครื่องมือสำหรับตรวจการทำงานของโปรเซสซึ่งถือเป็นหัวใจหลักของงานวิจัยเพราะประสิทธิภาพในการทำงานของ RSD จะอยู่ที่ส่วนนี้เป็นหลัก ส่วนสุดท้ายคือเครื่องมือประกอบการทำงานอื่นๆ เช่นซอฟต์แวร์สำหรับคอมไพล์เคอร์เนล หรือซอฟต์แวร์สำหรับสร้างซอฟต์แวร์ทดสอบ เป็นต้น โดยปัจจัยที่ใช้ในการเลือกเครื่องมือทั้งหมดตั้งอยู่บนข้อกำหนดเบื้องต้นดังนี้

- ระบบปฏิบัติการที่ใช้จะต้องมีความเสถียรพอสมควร และสามารถทำการปรับแต่งเพื่อตรวจสอบการทำงานภายในของระบบปฏิบัติการ หรือสามารถตรวจสอบการทำงานของโปรเซสที่อยู่บนระบบปฏิบัติการได้
- เครื่องมือที่ใช้สำหรับตรวจหาการเปลี่ยนแปลงของซอฟต์แวร์ในขณะทำงานจะต้องสามารถทำงานได้โดยรบกวนการทำงานของระบบปฏิบัติการให้น้อยที่สุด และต้องปลอดภัยในกรณีที่ระบบ RSD เกิดปัญหาจะต้องไม่ส่งผลกระทบต่อระบบปฏิบัติการ และต้องสามารถปรับแต่งการทำงานได้ง่ายไม่ซับซ้อน
- เครื่องมือประกอบการทำงานอื่นๆ เช่นเครื่องมือพัฒนาโปรแกรม และคอมไพเลอร์จะต้องสามารถทำงานร่วมกับระบบปฏิบัติการได้เป็นอย่างดี

จากข้อกำหนดเบื้องต้นดังกล่าว งานวิจัยนี้เลือกระบบปฏิบัติการลินุกซ์ [15] เป็นระบบปฏิบัติการหลักสำหรับพัฒนา ระบบตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน เนื่องจากลินุกซ์เป็นระบบปฏิบัติการที่มีความเสถียร และเป็นที่ยอมรับในการใช้งานโดยดูได้จากการที่ระบบปฏิบัติการลินุกซ์ถูกนำไปใช้ในงานหลากหลายประเภท ตั้งแต่ระบบขนาดเล็กอย่างเช่น ระบบสมองกลฝังตัว (Embedded System) ไปจนถึงระบบเครือข่ายขนาดใหญ่ เช่น Cloud computing เป็นต้น เนื่องจากระบบปฏิบัติการลินุกซ์เป็นซอฟต์แวร์ประเภท Opensource[†] จึงทำให้ผู้วิจัยสามารถทำการปรับแต่งการใช้งานให้เหมาะสมกับความต้องการได้โดยสะดวก อีกทั้งระบบปฏิบัติการลินุกซ์ยังมีการผนวกเครื่องมือต่างๆ ไว้ภายในตัวเองซึ่งจำเป็นต่องานวิจัย เช่น เครื่องมือพัฒนาซอฟต์แวร์ภาษา C และ C++ รวมถึงเครื่องมือสนับสนุนการทำงานอื่นๆ ที่มีอยู่อย่างครบครัน

ในส่วนของเครื่องมือสำหรับตรวจสอบการทำงานของระบบปฏิบัติการ หรือของโปรเซสนั้น งานวิจัยนี้ได้เลือก Systemtap [16,17] เป็นเครื่องมือหลักสำหรับใช้พัฒนาระบบ RSD ด้วยเหตุที่ Systemtap ถูกใช้เป็นเครื่องมือสำหรับติดตามการทำงานของโปรเซส โดยไม่รบกวนการทำงานของระบบและประสิทธิภาพของระบบปฏิบัติการ และยังสามารถปรับแต่งการทำงานได้โดยง่ายซึ่งแตกต่างกับการใช้งาน dtrace[‡] (Dynamic tracing) ซึ่งถูกใช้ในระบบปฏิบัติการตระกูลยูนิกซ์ อย่างเช่น Oracle Solaris[§], FreeBSD[**] เป็นต้น โดยผู้ใช้ไม่สามารถปรับแต่งการทำงานของ dtrace ได้เพราะเครื่องมือสำหรับตรวจการทำงานของโปรเซสถูกผนวกรวมไว้กับเคอร์เนล อีกทั้ง dtrace ยังทำงานได้ดีบนระบบปฏิบัติการ Solaris เพียงอย่างเดียวในการนำมาใช้งานบนระบบปฏิบัติอื่นอย่างเช่นลินุกซ์ ยังไม่สามารถใช้งานได้ดีเท่าที่ควร ดังนั้นงานวิจัยนี้จึงเลือก Systemtap เป็นเครื่องมือหลักในการพัฒนาระบบตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน หรือ RSD ดังกล่าว

3.2 การออกแบบวิธีการ RSD

จากการทำงานพื้นฐานของระบบปฏิบัติการทั่วไป ที่ทุกโปรเซสต้องอาศัยการทำงานผ่านศูนย์กลางของระบบปฏิบัติการ หรือเคอร์เนล โดยการทำงานดังกล่าวจะต้องกระทำ

[†] http://www.opensource.org/osd.html

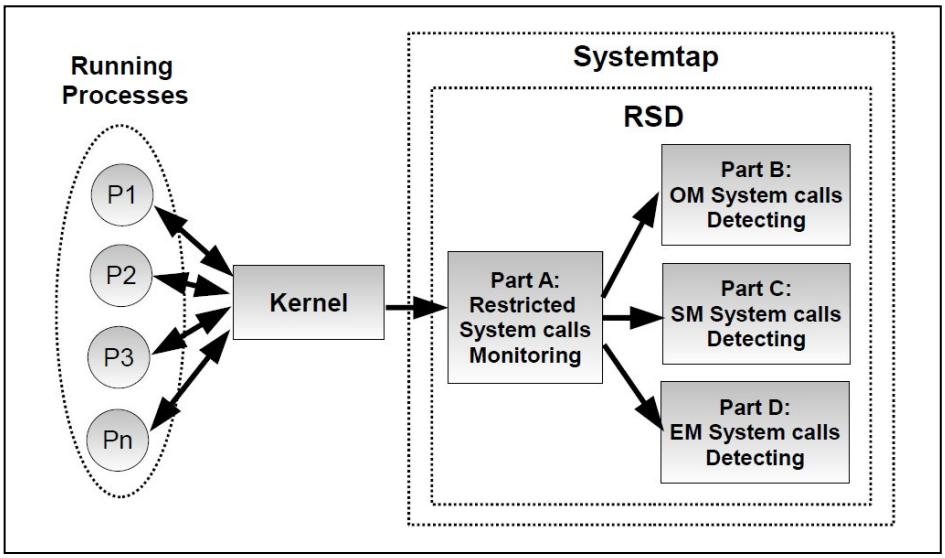
[‡] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal (June 2004). "Dynamic Instrumentation of Production Systems"

[§] www.oracle.com

[**] www.freebsd.com

ผ่านคำสั่งระบบ ซึ่งทำหน้าที่เป็นเสมือนช่องทางติดต่อระหว่างโปรเซสกับเคอร์เนลของระบบ อีกทั้งเป็นการทำงานแบบรวมศูนย์จึงทำให้ง่ายต่อการจัดการ จากคุณสมบัติการทำงานระหว่างโปรเซสและคำสั่งระบบดังกล่าว จึงเกิดแนวคิดในการทำงานว่า ถ้าสามารถตรวจสอบการเรียกใช้งานคำสั่งระบบของทุกโปรเซสได้ ย่อมสามารถทราบถึงพฤติกรรมของโปรเซสต่างๆได้เช่นกัน รายละเอียดในการออกแบบ RSD มีดังต่อไปนี้

3.2.1 ภาพรวมของวิธีการ RSD



ภาพที่ 3.2.1 ภาพรวมของการทำ RSD

จากการที่ RSD ต้องมีส่วนที่ทำหน้าที่ตรวจสอบการทำงานระหว่าง โปรเซส และเคอร์เนล เพื่อตรวจสอบการทำงานของโปรเซสว่ามีความต้องการเปลี่ยนแปลง ข้อมูลหรือโครงสร้างภายในตัวเองหรือไม่ ดังภาพที่ 3.2.1 ซึ่งแสดงโครงสร้างพื้นฐานของการทำ RSD ที่นำเสนอ ซึ่งสามารถอธิบายการโดยสังเขปได้ดังนี้

P1 – Pn เป็นตัวแทนของโปรเซสที่ทำงานอยู่บนระบบปฏิบัติการทั้งหมด โดยในขณะโปรเซสทำงานอาจจะมีการร้องขอบริการบางอย่างจากเคอร์เนลโดยผ่านคำสั่งระบบ หรือมีการสั่งงานบางอย่างจากเคอร์เนลมายังโปรเซสที่ทำงานอยู่ โดยคำสั่งระบบจะแทนด้วยลูกศร 2 ทิศทาง ในขณะที่การติดต่อระหว่างโปรเซสและเคอร์เนลดำเนินอยู่นั้น ข้อมูลการใช้งานคำสั่งระบบดังกล่าวจะถูกดักจับได้โดย Kprobes ซึ่งเป็นเสมือนอุปกรณ์ที่วางไว้ภายในส่วนต่างๆ ของเคอร์

เนลเพื่อตรวจจับการทำงานภายในเคอร์เนล ซึ่งการเรียกใช้งานข้อมูลจาก Kprobes โดยตรงนั้นสามารถทำได้ แต่มีความยุ่งยากในการใช้งาน และอาจก่อให้เกิดผลกระทบต่อการทำงานของระบบปฏิบัติการได้โดยตรง

ดังนั้น RSD จึงอาศัยการทำงานของ Systemtap ในการนำข้อมูลจาก Kprobes มานำเสนอในระดับผู้ใช้งาน โดย Systemtap จะทำหน้าที่เป็นส่วนเฝ้าดูและเก็บข้อมูลการทำงานของโปรเซสจากภายนอก ซึ่งทำให้รบกวนการทำงานโดยรวมของระบบปฏิบัติการเพียงเล็กน้อย จากแนวคิดหลักของ RSD ในการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ด้วยการตรวจหาคำสั่งระบบเฉพาะแบบ ซึ่งมีการแบ่งคำสั่งระบบเฉพาะออกเป็น 3 กลุ่ม ซึ่งจะกล่าวถึงรายละเอียดของคำสั่งระบบเฉพาะแบบในหัวข้อต่อไป ในส่วนนี้จะกล่าวถึงส่วนประกอบหลักของการทำ RSD ซึ่งประกอบด้วย 4 ส่วนดังนี้

3.2.1.1 **ส่วนตรวจจับการเรียกใช้งานคำสั่งระบบทั้งหมด** (Part A)

ส่วนนี้เป็นส่วนสำหรับกรองการเรียกใช้งานคำสั่งระบบทั้งหมดของโปรเซสที่ทำงานอยู่บนระบบปฏิบัติการ ซึ่งการทำงานโดยปกติของ Systemtap จะทำการตรวจจับการเรียกใช้งานคำสั่งระบบทั้งหมดอยู่แล้ว เพียงแต่ว่าการตรวจจับเหล่านั้นไม่ถูกแสดงผล หรือไม่ถูกนำเสนอในระดับผู้ใช้ การที่จะสามารถแสดงผลการเรียกใช้งานคำสั่งระบบสามารถทำได้โดยใช้รูปแบบคำสั่งของ Systemtap ดังนี้

```
probe kernel.function("[ชื่อคำสั่งระบบ]")
```

คำสั่ง probe เป็นการระบุว่าให้ Systemtap อ่านข้อมูลจาก probe ซึ่งกำหนดในส่วนต่อไป คำสั่ง kernel.function() เป็นการระบุว่า probe ที่ทำการอ่านค่านั้นเป็นฟังก์ชันหนึ่งของเคอร์เนล ซึ่งจะถูกระบุชื่อของฟังก์ชันคือชื่อของคำสั่งระบบ

3.2.1.2 **ส่วนตรวจหาคำสั่งระบบที่ก่อให้เกิดการแก้ไขข้อมูลระหว่างโปรเซส** (Part B)

ส่วนนี้เป็นส่วนสำหรับกรองและตรวจหาเฉพาะคำสั่งระบบที่สามารถทำให้เกิดการแก้ไขข้อมูลระหว่างโปรเซสได้ ซึ่งคำสั่งระบบประเภทนี้จะเรียกโดยย่อว่า OM (Other process Modification system calls) ซึ่งรูปแบบในการเรียกใช้งานคำสั่งระบบ OM นั้นจะประกอบด้วย 2

ส่วนสำคัญคือ โปรเซสผู้เรียกใช้งาน และโปรเซสผู้ถูกเรียกใช้งาน ในขณะที่คำสั่งระบบ OM ถูกเรียกใช้งานนั้น RSD จะทำการบันทึก หมายเลขโปรเซสของผู้เรียกใช้งาน และหมายเลขโปรเซสของผู้ถูกเรียกใช้งาน โดย RSD จะนำหมายเลขทั้งสอง เก็บลงในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย (TargetProcessMap) เพื่อใช้สำหรับเป็นข้อมูลในการตรวจสอบการแก้ไขโปรเซสในส่วนต่อไป โครงสร้างของตัวแปรเชื่อมโยงโปรเซสเป้าหมายสามารถแสดงด้วยโครงสร้างพื้นฐานดังนี้

```
TargetProcessMap [PID of called process] = PID of caller process
```

รายละเอียดของส่วน B (Part B) สามารถแสดงได้ดังภาพที่ 3.2.1.2

```
บรรทัด 1: probe kernel.function("sys_ptrace") {
บรรทัด 2:          TargetProcessMap[$pid] = pid()
}
```

ภาพที่ 3.2.1.2 ตัวอย่างสคริป RSD ของส่วน B

จากภาพที่ 3.2.1.2 ซึ่งเป็นส่วนหนึ่งของสคริป RSD ตามภาคผนวก ข สามารถอธิบายการทำงานได้ดังนี้

บรรทัด 1 เป็นการกำหนดให้ Systemtap ตรวจจับการเรียกใช้งานคำสั่งระบบ “sys_ptrace” ซึ่งเป็นคำสั่งระบบที่โปรเซสบางโปรเซสเรียกใช้งานเพื่อแก้ไขข้อมูลบางอย่างในโปรเซสอื่น

บรรทัด 2 เป็นการจับคู่หมายเลขโปรเซสผู้เรียกใช้งาน กับหมายเลขโปรเซสผู้ถูกเรียกใช้งานลงในตัวแปรชื่อ “TargetProcessMap” โดยหมายเลขโปรเซสผู้เรียกใช้งานนั้นจะตรวจสอบได้โดยการเรียกฟังก์ชัน pid() ส่วนหมายเลขโปรเซสของผู้ถูกเรียกใช้งานนั้นจะตรวจสอบได้โดยการอ่านค่าในตัวแปร $pid

3.2.1.3 **ส่วนตรวจหาคำสั่งระบบที่ทำให้เกิดการแก้ไขข้อมูลในโปรเซส** (Part C)

ส่วนนี้เป็นส่วนสำหรับกรองและตรวจหาการเรียกใช้งานคำสั่งระบบที่ทำให้โปรเซสเกิดการแก้ไขตัวเอง ซึ่งคำสั่งระบบประเภทนี้จะเรียกโดยย่อว่า SM (Self Modification system calls) รูปแบบการเรียกใช้งานคำสั่งระบบ SM นี้จะแตกต่างกับคำสั่งระบบประเภท OM คือจะสามารถตรวจสอบได้เฉพาะหมายเลขโปรเซสผู้เรียกใช้งานเท่านั้น

การทำงานของส่วนนี้คือทันทีที่มีการเรียกใช้งานคำสั่งระบบประเภท SM ระบบ RSD จะทำการตรวจสอบหมายเลขโปรเซสของผู้เรียกว่าอยู่ในตัวแปรเชื่อมโยงโปรเซสเป้าหมายหรือไม่ เพื่อตรวจสอบว่าโปรเซสที่เรียกใช้งานนั้นมีโอกาสที่จะถูกแก้ไขโดยโปรเซสอื่นหรือไม่ ถ้าหมายเลขโปรเซสผู้เรียกใช้งานไม่อยู่ในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย แสดงว่าโปรเซสที่เรียกคำสั่งระบบ SM นั้นทำการแก้ไขตัวเองซึ่ง RSD จะไม่สนใจ แต่ถ้าหมายเลขโปรเซสผู้เรียกใช้งานคำสั่งระบบ SM ปรากฏอยู่ในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย นั่นหมายถึงการแก้ไขตัวเองที่เกิดขึ้นมีความเป็นไปได้ว่าเกิดจากการแก้ไขโดยโปรเซสอื่น RSD จะทำการรายงานว่ามี NRSM เกิดขึ้น รายละเอียดของส่วนนี้สามารถอธิบายได้ดังภาพที่ 3.2.1.3

```
บรรทัด 1: probe kernel.function("sys_brk") {
บรรทัด 2:     if(TargetProcessMap [pid()] != 0)
              {
บรรทัด 4:     printf("%s(%d) NRSM by %d\n",execname(),pid(),TargetProcessMap[pid()])
บรรทัด 5:     NRSM_list[pid(),"sys_brk"] <<< 1
              }
}
```

ภาพที่ 3.2.1.3 ตัวอย่างสคริป RSD ของส่วน C

จากภาพที่ 3.2.1.3 ซึ่งเป็นส่วนหนึ่งของสคริป RSD ตามภาคผนวก ข สามารถอธิบายการทำงานได้ดังนี้

บรรทัด 1 เป็นการกำหนดให้ Systemtap ตรวจจับการเรียกใช้งานคำสั่งระบบประเภท SM ซึ่งในตัวอย่างเป็นการตรวจจับการเรียกใช้งานคำสั่งระบบชื่อ “sys_brk” ซึ่งเป็นคำสั่งระบบที่ใช้สำหรับปรับเปลี่ยนขนาดของหน่วยความจำของโปรเซสที่เรียกใช้งาน

บรรทัด 2 เป็นการตรวจสอบว่าหมายเลขโปรเซสผู้เรียกใช้งาน อยู่ในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย (TargetProcessMap) หรือไม่ ถ้าอยู่จะทำงานในบรรทัดที่ 4 แต่ถ้าไม่อยู่จะจบการทำงานในส่วนนี้

บรรทัด 4 เป็นการรายงานว่าตรวจพบ NRSM หรือการเปลี่ยนแปลงข้อมูลของโปรเซสที่ไม่ปกติ โดยหมายเลขของโปรเซสที่ถูกเปลี่ยนแปลงจะอ่านได้จากฟังก์ชัน pid() และหมายเลขโปรเซสของผู้แก้ไขจะสามารถอ่านได้จากการอ่านค่าของตัวแปรเชื่อมโยงโปรเซสเป้าหมาย TargetProcessMap[pid()]

บรรทัด 5 เป็นการเก็บสถิติว่าเกิด NRSM ขึ้นกับโปรเซสที่เรียกใช้งานทั้งหมดกี่ครั้ง ซึ่งถูกใช้สำหรับประเมินรูปแบบของการเปลี่ยนแปลงซอฟต์แวร์ว่าเกิดขึ้นกับคำสั่งระบบประเภทใดมากที่สุด

3.2.1.4 **ส่วนวิเคราะห์คำสั่งระบบที่สนับสนุนให้เกิดการแก้ไขข้อมูลโปรเซส** (Part D)

ส่วนนี้เป็นส่วนสำหรับกรอง และตรวจหาเฉพาะคำสั่งระบบที่สนับสนุนให้เกิดการแก้ไขข้อมูลภายในโปรเซส ซึ่งคำสั่งระบบประเภทนี้จะเรียกโดยย่อว่า EM (Encourage Modification system calls) หรืออีกนัยหนึ่งเป็นคำสั่งระบบที่อาจก่อให้เกิดความเสี่ยงในการแก้ไขข้อมูลหรือโครงสร้างภายในโปรเซส การทำงานของส่วนนี้ไม่ได้เป็นการชี้ชัดว่าข้อมูลภายในโปรเซสถูกแก้ไขหรือไม่ ดังนั้น RSD จึงทำเพียงรายงานว่ามีความเสี่ยงในการแก้ไขข้อมูลภายในโปรเซสเกิดขึ้นหรือไม่ รายละเอียดของส่วนนี้สามารถอธิบายได้ดังภาพที่ 3.2.1.4

```
บรรทัด 1: probe kernel.function("sys_mprotect") {
บรรทัด 2: if($prot & 0x2){
บรรทัด 3:         printf("%s(%d) has a chance to modification\n",execname(),pid())
บรรทัด 4:         CHANCE_list[pid(),"sys_mprotect"] <<< 1
        }
}
```

ภาพที่ 3.2.1.4 ตัวอย่างสคริป RSD ของส่วน D

จากภาพที่ 3.2.1.4 ซึ่งเป็นส่วนหนึ่งของสคริป RSD ตามภาคผนวก ข สามารถอธิบายการทำงานได้ดังนี้

บรรทัด 1 เป็นการกำหนดให้ Systemtap ตรวจจับการเรียกใช้งานคำสั่งระบบประเภท EM ซึ่งในตัวอย่างเป็นการตรวจจับการเรียกใช้งานคำสั่งระบบชื่อ “sys_mprotect” ซึ่งเป็นคำสั่งระบบที่ใช้สำหรับกำหนดการป้องกันหน่วยความจำที่เก็บโปรแกรมของโปรเซส ว่าอนุญาตให้เขียนได้หรือไม่

บรรทัด 2 เป็นการตรวจสอบค่าพารามิเตอร์ของคำสั่งระบบว่าถูกกำหนดให้เขียนข้อมูลในหน่วยความจำของโปรเซสได้หรือไม่ โดยการเขียนข้อมูลลงหน่วยความจำของโปรเซสสามารถตรวจสอบได้โดยอ่านข้อมูลในบิตที่ 2 ของตัวแปรชื่อ $prot ถ้ามีการกำหนดให้เขียนข้อมูลได้จะไปทำยังบรรทัดที่ 3

บรรทัด 3 เป็นการรายงานว่ามีโอกาสที่โปรเซสจะถูกเปลี่ยนแปลงในขณะทำงานโดยแสดงหมายเลขโปรเซสด้วยการเรียกใช้งานฟังก์ชัน pid()

บรรทัด 4 เป็นการเก็บสถิติว่ามีโอกาสที่โปรเซสจะถูกเปลี่ยนแปลงขณะทำงานทั้งหมดกี่ครั้ง ซึ่งใช้สำหรับประเมินรูปแบบของโอกาสการเปลี่ยนแปลงซอฟต์แวร์ว่าเกิดขึ้นกับคำสั่งระบบประเภทใดมากที่สุด

3.2.2 **คำสั่งระบบเฉพาะแบบ**

คำสั่งระบบเฉพาะแบบ หมายถึงคำสั่งระบบที่สามารถทำให้เกิดการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ซึ่งอาจจะเป็นผลจากการทำงานของคำสั่งโดยตรง หรือเป็นการสนับสนุนโดยทางอ้อม จากตารางที่ 3.2.2 เห็นได้ว่าคำสั่งระบบทั้งหมดที่งานวิจัยนี้ให้ความสำคัญมีทั้งหมด 8 คำสั่ง ซึ่งคัดเลือกมาจากคำสั่งระบบในระบบปฏิบัติการลินุกซ์ทั้งหมด ซึ่งหลักการในการเลือกคำสั่งระบบเหล่านี้ผู้วิจัยพิจารณาที่ผลการทำงานของคำสั่งระบบแต่ละคำสั่งเป็นหลัก เช่น คำสั่ง ptrace ซึ่งเป็นคำสั่งที่โปรเซสหนึ่ง สามารถควบคุมการทำงานของอีกโปรเซสหนึ่งได้ ซึ่งเห็นได้ชัดว่ามีความเป็นไปได้ที่โปรเซสเป้าหมายจะถูกแก้ไขในระหว่างที่มีการเรียกใช้งานคำสั่งระบบดังกล่าว หรือ อีกตัวอย่างคือคำสั่ง munmap ซึ่งเป็นคำสั่งที่โปรเซสทำการคืนหน่วยความจำให้กับระบบปฏิบัติการซึ่งเห็นได้ชัดว่ามีการเปลี่ยนแปลงเกิดขึ้นกับโครงสร้างภายในของโปรเซสอย่างชัดเจน ดังนั้นคำสั่งระบบนี้จึงถูกจัดอยู่ในกลุ่มคำสั่งระบบเฉพาะแบบ ดังกล่าว

คำสั่งระบบเฉพาะแบบสามารถแบ่งได้เป็น 3 ประเภทดังนี้

1. คำสั่งระบบเฉพาะแบบที่ทำให้เกิดการแก้ไขข้อมูลระหว่างโปรเซส หรือเรียกโดยย่อว่า OM คำสั่งระบบในกลุ่มนี้เป็นคำสั่งระบบที่ยินยอมให้มีการติดต่อสื่อสารควบคุมการทำงานระหว่างโปรเซส โดยไม่สนใจว่าการติดต่อหรือการควบคุมนั้นจะกระทำด้วยความถูกต้องหรือไม่ เพราะการเปิดโอกาสให้มีการควบคุมระหว่างโปรเซสนั้นอาจเป็นช่องทางที่ซอฟต์แวร์ปรสิตใช้สำหรับโจมตีซอฟต์แวร์เป้าหมายได้ ซึ่งคำสั่งระบบในกลุ่มนี้มีเพียง คำสั่ง ptrace ที่ถูกใช้เพื่อทำการดีบักซอฟต์แวร์ ซึ่งซอฟต์แวร์ปรสิตส่วนใหญ่ ใช้ในการทำงานด้วยเช่นกัน รายละเอียดของคำสั่งระบบประเภทนี้ได้ถูกกล่าวไปแล้วในหัวข้อ 3.2.1

2. คำสั่งระบบเฉพาะแบบในกลุ่มที่ทำให้เกิดการแก้ไขข้อมูลภายในโปรเซสตัวเอง หรือเรียกโดยย่อว่า SM คำสั่งระบบในกลุ่มนี้เป็นคำสั่งระบบที่ทำให้เกิดการเปลี่ยนแปลงโครงสร้างภายในโปรเซสเองอย่างชัดเจน เช่นการเพิ่ม หรือลดหน่วยความจำที่ใช้ในโปรเซส ซึ่งอาจไม่ส่งผลต่อการทำงานของโปรเซสโดยตรง คำสั่งระบบในกลุ่มนี้ประกอบด้วย คำสั่งระบบต่อไปนี้

    - mmap เป็นคำสั่งระบบ ที่ทำหน้าที่ผนวกหน่วยความจำเข้ากับโปรเซสที่เรียกใช้งานทำให้จำนวนเพจของหน่วยความจำและข้อมูลในหน่วยความจำเปลี่ยนแปลง

    - munmap เป็นคำสั่งระบบ ที่ทำหน้าที่ยกเลิกการผนวกหน่วยความจำออกจากโปรเซสที่เรียกใช้งาน ส่งผลให้จำนวนเพจของหน่วยความจำ และ ข้อมูลภายในโปรเซสเปลี่ยนแปลง

    - mremap เป็นคำสั่งระบบ ที่ทำหน้าที่ปรับเปลี่ยนเพจของหน่วยความจำที่ถูกผนวกเข้ากับโปรเซส ซึ่งการปรับเปลี่ยนนี้อาจจะเป็นการลดหรือเพิ่มหน่วยความจำได้ทั้ง 2 กรณี ส่งผลให้จำนวนเพจของหน่วยความจำและข้อมูลโปรเซสเปลี่ยนแปลง

- brk เป็นคำสั่งระบบ ที่ทำหน้าที่ปรับเปลี่ยนขนาดของหน่วยความจำสำหรับเก็บส่วนที่เป็นข้อมูลของโปรเซส ส่งผลให้ขนาดของหน่วยความจำและข้อมูลในโปรเซสเปลี่ยนแปลง

3. คำสั่งระบบเฉพาะแบบที่สนับสนุนให้เกิดการแก้ไขข้อมูลภายในโปรเซส หรือเรียกโดยย่อว่า EM คำสั่งระบบประเภทนี้เมื่อถูกเรียกใช้งานจะไม่สามารถระบุได้อย่างชัดเจนว่าเกิดการเปลี่ยนแปลงขึ้นภายในโปรเซสหรือไม่ แต่สามารถระบุได้ถึงความเสี่ยงและแนวโน้มที่โปรเซสอาจจะถูกแก้ไข หรือเปลี่ยนแปลง เช่นคำสั่ง mlock ซึ่งเป็นการกำหนดให้คงหน่วยความจำของโปรเซสไว้บนหน่วยความจำหลักแม้โปรเซสนั้นถูก suspend ไปแล้ว ซึ่งมีความเป็นไปได้ที่ข้อมูลบนหน่วยความจำจะถูกแก้ไขในช่วงเวลาดังกล่าว คำสั่งระบบกลุ่มนี้ประกอบด้วยคำสั่งระบบต่อไปนี้

   - mprotect เป็นคำสั่งระบบ ที่สามารถใช้สำหรับปลดการป้องกันการเขียนข้อมูลลงหน่วยความจำ ส่วนที่เป็นคำสั่งหรือโปรแกรมของโปรเซส (เฉพาะการอนุญาตให้เขียน) เนื่องจากโดยปกติส่วนที่เป็นคำสั่งของโปรเซสจะถูกป้องกันไม่ให้สามารถเขียนได้ ดังนั้นถ้ามีการเรียกใช้งานคำสั่ง mprotect ย่อมมีความเสี่ยงที่จะก่อให้เกิดการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานได้

   - mlock เป็นคำสั่งระบบ ที่สั่งให้ระบบคงหน่วยความจำของโปรเซสไว้บนหน่วยความจำหลักของระบบเพื่อทำงานบางอย่าง ซึ่งโดยปกติหน่วยความจำของโปรเซสจะถูกนำเข้าและออกจากหน่วยความจำหลักตลอดเวลา ส่งผลให้อาจมีการแก้ไขข้อมูลบนหน่วยความเกิดขึ้นในระหว่างช่วงเวลาที่คงข้อมูลไว้บนหน่วยความจำดังกล่าว

   - mlockall เป็นคำสั่งระบบ ที่ทำหน้าที่คล้ายคลึงกับคำสั่ง mlock เพียงแต่ mlockall จะทำการคงข้อมูลของทุกโปรเซสไว้บนหน่วยความจำหลักซึ่งส่งผลเช่นเดียวกับ mlock คืออาจก่อให้เกิดการแก้ไขข้อมูลในหน่วยความจำในระหว่างทำคำสั่งดังกล่าว

ตารางที่ 3.2.2 ตารางสรุปคำสั่งระบบเฉพาะแบบ

No.	System calls	Category	Descriptions
1	*ptrace*	OM	To allow a parent process controls the execution of a child process
2	*mmap*	SM	To create a new memory mapping in the virtual address space of calling process
3	*munmap*	SM	To delete the memory mapping for specified address range
4	*mremap*	SM	To extend or shrink the process's memory mapping
5	*brk*	SM	To change the amount of space allocated for calling data segment
6	*mprotect*	EM	To set protection of memory mapping (focus only write flag)
7	*mlock*	EM	To lock part of process's virtual address space into memory
8	*mlockall*	EM	To lock all processes's virtual address spaces into memory

ตารางที่ 3.2.2 แสดงคำสั่งระบบเฉพาะแบบที่งานวิจัยนี้นำมาพิจารณาในการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานทั้งหมด ในส่วนของรายละเอียดของแต่ละคำสั่งสามารถศึกษาเพิ่มเติมได้ในภาคผนวก ก

3.2.3 **หลักการทำงานของวิธีการ** RSD

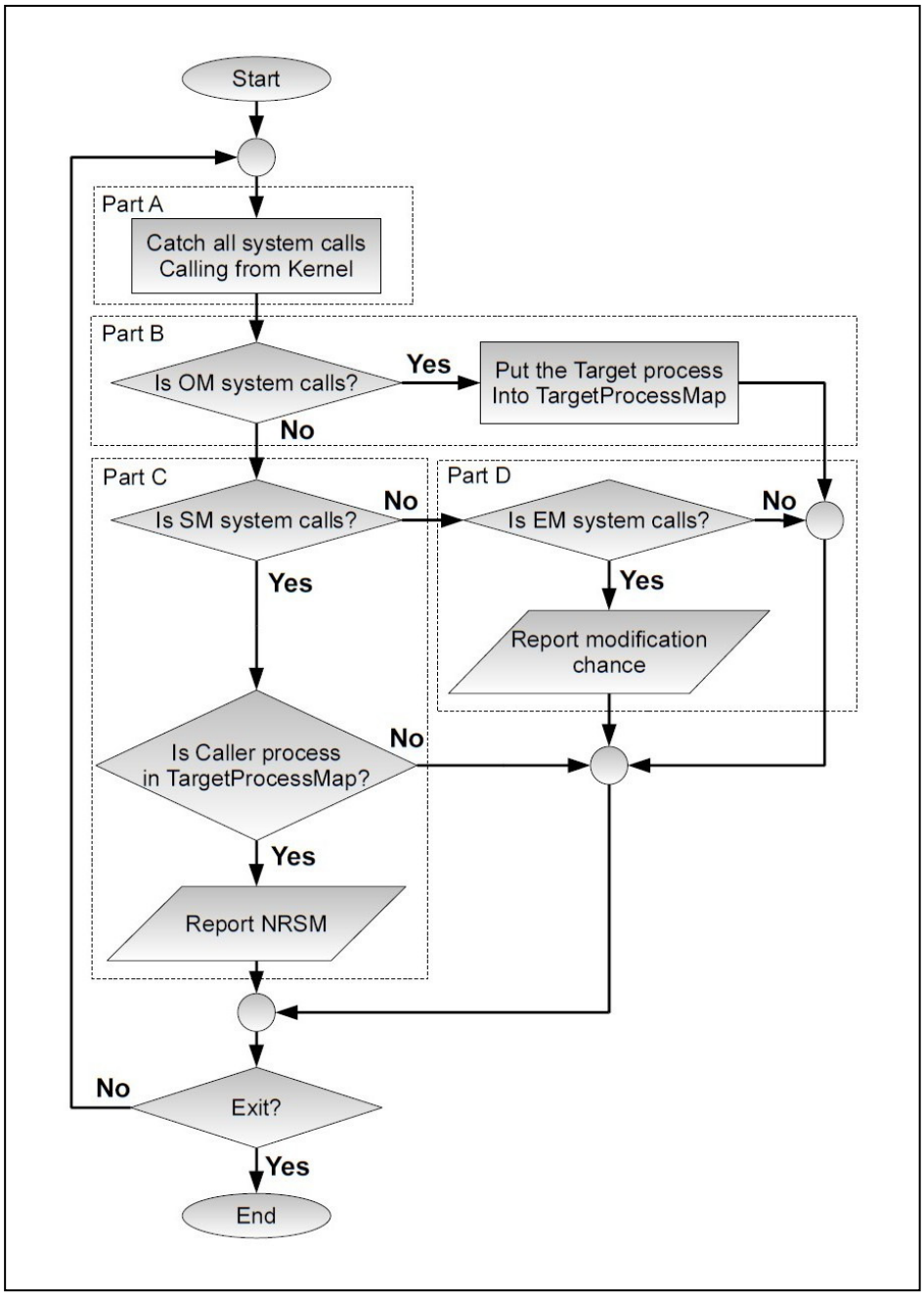
ในส่วนนี้เป็นอธิบายการทำงานของ RSD ในรูปแบบของผังดำเนินงาน และ pseudo code เพื่อให้ง่ายต่อความเข้าใจ และสามารถนำไปพัฒนาเป็นซอฟต์แวร์ที่ใช้งานได้

จากภาพที่ 3.2.3-1 เป็นการอธิบายการทำงานของ RSD ในรูปแบบผังดำเนินงาน ซึ่งแบ่งได้ 4 ส่วนตามภาพรวมการทำ RSD ในภาพที่ 3.2.2 โดย ในส่วนแรก (Part A) เป็นการตรวจสอบการเรียกใช้งานคำสั่งระบบเฉพาะแบบทั้งหมดของโปรเซสที่เรียกเข้ามายังเคอร์เนล ในส่วนที่สอง (Part B) เป็นการตัดสินใจว่าคำสั่งระบบที่ถูกเรียกนั้นเป็น คำสั่งระบบที่แก้ไขโปรเซสอื่นหรือไม่ ถ้าไม่ใช่การทำงานจะถูกส่งต่อไปยังส่วนต่อไป แต่ถ้าใช่คำสั่งระบบที่สามารถแก้ไขโปรเซสอื่นได้ RSD จะทำการบันทึกหมายเลขโปรเซส (PID) ของโปรเซสปลายทางที่ถูกเรียก ลงในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย (TargetProcessMap) ซึ่งจะใช้ในการวิเคราะห์หาการเปลี่ยนแปลงโปรเซสที่ผิดปกติต่อไป

ในส่วนที่สาม (Part C) เป็นการตัดสินใจว่าคำสั่งระบบที่ถูกเรียกเป็นคำสั่งระบบที่โปรเซสแก้ไขตัวเองหรือไม่ ถ้าไม่ใช่จะถูกส่งไปยังส่วนต่อไป แต่ถ้าใช่ RSD จะทำการตรวจสอบว่าหมายเลขโปรเซสของผู้เรียกคำสั่งระบบอยู่ในรายการเป้าหมายหรือไม่ ถ้าไม่อยู่แสดงว่าโปรเซสนั้นทำการแก้ไขตัวเองปกติ RSD จะไม่สนใจต่อการเรียกใช้งานคำสั่งระบบดังกล่าว แต่ถ้าหมายเลขโปรเซสของผู้เรียกคำสั่งระบบอยู่ในตัวแปรเชื่อมโยงโปรเซสเป้าหมาย นั่นแสดงว่าโปรเซสนั้นถูกสั่งให้ทำการแก้ไขโดยโปรเซสอื่น ซึ่งจากการตรวจสอบพฤติกรรมของ ซอฟต์แวร์ปรสิต จะมีการเรียกใช้งานคำสั่งระบบที่แก้ไขโปรเซสอื่นอย่างเช่น ptrace และจะตามด้วยคำสั่งระบบที่โปรเซสทำการแก้ไขตัวเองเสมอ เมื่อ RSD ตรวจพบรูปแบบการเรียกใช้งานคำสั่งดังกล่าว จะทำรายงานว่าตรวจพบ NRSM หรือการแก้ไขโปรเซสที่ไม่ปกติเกิดขึ้นทันที

ในส่วนสุดท้ายส่วนที่สี่ (Part D) เป็นส่วนตัดสินใจว่าคำสั่งระบบที่ถูกเรียกใช้งานนั้นเป็นคำสั่งที่ก่อให้เกิดความเสี่ยงต่อการแก้ไขข้อมูลภายในโปรเซสหรือไม่ ถ้าไม่ใช่ RSD จะไม่ดำเนินการใดๆ แต่ถ้าใช่ RSD จะทำการรายงานว่าตรวจพบการใช้งานคำสั่งที่มีความเสี่ยงต่อการเปลี่ยนแปลงของโปรเซส ซึ่งเป็นเพียงการเตือนและให้เฝ้าระวังโปรเซสที่เรียกใช้งานคำสั่งดังกล่าว

จากนั้น RSD จะวนลูปทำการตรวจสอบการเรียกใช้งานคำสั่งระบบเช่นนี้จนกว่าจะถูกสั่งให้หยุดทำงานโดยผู้ใช้งานหรือโดยระบบปฏิบัติการ



ภาพที่ 3.2.3-1 ผังงานของ RSD

นอกจากผังงานแล้วการอธิบายการทำงานของ RSD ยังสามารถอธิบายได้ด้วยรหัส pseudo ดังภาพที่ 3.2.3-2 เพื่อง่ายต่อการนำไปพัฒนาซอฟต์แวร์ ซึ่งโปรแกรมสำหรับสร้าง RSD นั้นถูกเขียนขึ้นในรูปแบบคำสั่งของ SystemTap โดยมีรายละเอียดของโปรแกรมตามภาคผนวก ข

```
WHILE NOT Exit DO

    ;====================================
    ; Part A: All system calls capturing
    ;====================================
    ; all system calls are captured by default

    ;====================================
    ; Part B: OM System calls analysing
    ;====================================
    IF ptrace IS CALLED THEN
        IF TargetProcessMap[Target PID] IS NULL THEN
            TargetProcessMap [Target PID] = Caller PID
        ENDIF
    ENDIF

    ;====================================
    ; Part C: SM System calls analysing
    ;====================================
    IF      brk OR mmap OR
        munmap OR mremap IS CALLED THEN
        IF TargetProcessMap [Caller PID] IS NOT NULL THEN
            SEND “PID”;Caller PID;”is NRSM”
        ENDIF
    ENDIF

    ;====================================
    ; Part D: EM System calls analysing
    ;====================================
    IF      mprotect(write mode) OR mlock OR
        mlockall IS CALLED THEN

        SEND “PID”; Caller PID; “ has a modification chance”

    ENDIF

ENDWHILE
```

ภาพที่ 3.2.3-2 pseudo ของ RSD

3.3 ขั้นตอนการดำเนินงานวิจัย

ในหัวข้อนี้เป็นการกล่าวถึงขั้นตอนในการนำแนวคิด และการออกแบบที่ได้นำเสนอไปในส่วนที่แล้ว มาประยุกต์ และสร้างให้เกิดซอฟต์แวร์ RSD ที่สามารถทำงานได้ตามที่ได้ออกแบบไว้ โดยในส่วนนี้จะประกอบด้วย ขั้นตอนการเตรียมระบบปฏิบัติการ ไปจนถึงการเตรียมซอฟต์แวร์เพื่อทดสอบการทำงาน รายละเอียดแต่ละขั้นตอนมีดังนี้

3.3.1 เตรียมระบบเพื่อใช้ในงานวิจัย

ก่อนทำการทดสอบการทำงานของ RSD จะต้องเตรียมระบบปฏิบัติการลินุกซ์และเครื่องมือต่างๆ ให้พร้อมสำหรับการทำงาน ในหัวข้อนี้จะเป็นการอธิบายถึงขั้นตอนการปรับแต่งระบบปฏิบัติการลินุกซ์เพื่อให้รองรับการทำงานของ Systemtap โดยมีรายละเอียดการทำงานดังนี้

- ปรับแต่งให้ระบบปฏิบัติการลินุกซ์สามารถคอมไพล์ซอร์ซโค้ด ภาษาซี และ สามารถคอมไพล์ลินุกซ์เคอร์เนลได้
- ปรับแต่งให้ระบบปฏิบัติการลินุกซ์สามารถรองรับการทำงานของ Systemtap

รายละเอียดขั้นตอนการเตรียมระบบทั้งหมด ถูกอธิบายไว้ใน ภาคผนวก ง

3.3.2 จำลองการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

ในส่วนนี้จะเป็นการกล่าวถึงการสร้างซอฟต์แวร์เพื่อใช้สำหรับเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน เพื่อใช้สำหรับทดสอบการทำงานของ RSD ว่าสามารถตรวจหาการเปลี่ยนแปลงที่เกิดขึ้นได้หรือไม่ งานวิจัยนี้ได้เลือกใช้วิธีการทำซอฟต์แวร์ปรสิตในการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ซึ่งวิธีการทำซอฟต์แวร์ parasite นั้นแบ่งเป็นขั้นตอนโดยสังเขปได้ดังนี้

- เตรียมซอร์สโค้ดของซอฟต์แวร์ปรสิต โดยสร้างไฟล์ชื่อ libtest.c และ dlh.c ในเครื่องที่ติดตั้งระบบปฏิบัติการลินุกซ์เรียบร้อยแล้ว

- คอมไพล์ ซอฟต์แวร์ปรสิต ซึ่งประกอบด้วย 2 ส่วนคือ
    - ส่วนที่เป็นปรสิต (libtest.c) ซึ่งส่วนนี้จะเป็นส่วนที่ทำหน้าที่หลักของปรสิต ต้องคอมไพล์ให้อยู่ในรูปของเคอร์เนลโมดูลซึ่งวิธีการคอมไพล์โมดูลสามารถทำได้โดยใช้คำสั่ง

```
gcc –fPIC –c libtest.c –nostdlib [enter]
sudo ld –shared –o libtest.so.1.0 libtest.o [enter]
sudo cp libtest.so.1.0 /lib [enter]
```

หลังจากขั้นตอนนี้จะได้ไลบรารี่ของซอฟต์แวร์ปรสิต (libtest.so.1.0) โดยต้องนำไปวางไว้ยังตำแหน่ง /lib ซึ่งเป็นโฟลเดอร์สำหรับเก็บไลบรารี่ของระบบทั้งหมด

    - ส่วนแพร่กระจาย (dlh.c) ส่วนนี้จะทำหน้าที่นำปรสิตไลบรารี่เข้าไปแทรกยังซอฟต์แวร์เป้าหมาย ซึ่งคอมไพล์ได้ด้วยคำสั่ง

```
gcc dlh.c –o dlh [enter]
```

หลังจากขั้นตอนนี้จะได้ซอฟต์แวร์ชื่อ dlh ซึ่งจะถูกใช้สำหรับเรียกใช้งานเพื่อแพร่ซอฟต์แวร์ปาราสิตไปยังซอฟต์แวร์เป้าหมาย

- ทดสอบการแพร่กระจายปรสิตไปยังโปรแกรมทดสอบอย่างง่าย เช่นโปรแกรมที่ทำงานแบบวนรอบเพื่อแสดงข้อความบางอย่าง เป็นต้น จากนั้นตรวจสอบหมายเลขโปรเซสของโปรแกรมทดสอบ ซึ่งหมายเลขโปรเซสนี้จำเป็นสำหรับการแพร่กระจายซอฟต์แวร์ปรสิต ด้วยคำสั่ง

```
ps ax |grep [ชื่อโปรแกรมทดสอบ] [enter]
```

จะปรากฏหมายเลขโปรเซสของโปรแกรมที่ใช้ทดสอบ จากนั้นทำการแพร่ซอฟต์แวร์ปรสิตไปยังโปรแกรมทดสอบด้วยคำสั่ง

```
./dlh [หมายเลขโปรเซส] puts [enter]
```

- ตรวจสอบว่าปรสิตถูกแพร่ไปยังซอฟต์แวร์เป้าหมายหรือไม่ด้วยคำสั่ง

```
cat /proc/[หมายเลขโปรเซส]/maps [enter]
```

จะปรากฏรายละเอียดโครงสร้างของหน่วยความจำที่ใช้ในโปรแกรมทดสอบทั้งหมด จากนั้นตรวจหาไลบรารี่ชื่อ “libtest.so.1.0” ถ้าพบแสดงว่าการแพร่กระจายซอฟต์แวร์ปรสิตสามารถกระทำได้อย่างสมบูรณ์

3.3.3 การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

หลังจากทดสอบการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ด้วยซอฟต์แวร์ปรสิตซึ่งได้กล่าวถึงในหัวข้อ 3.3.2 เป็นที่เรียบร้อยแล้ว ในส่วนนี้จะกล่าวถึงวิธีการสร้าง และทดสอบการทำงานของ RSD เพื่อใช้สำหรับตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานที่เกิดขึ้นในสภาพแวดล้อมที่กำหนด[††] ขั้นตอนการสร้างและทดสอบการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานด้วยการใช้ RSD สามารถแสดงได้ดังนี้

- เตรียมสคริป RSD โดยสร้างไฟล์ชื่อ “RSD_Script.stp” บนระบบปฏิบัติการลินุกซ์ที่เตรียมไว้ จากนั้นนำข้อมูลในภาคผนวก ข มาใส่ลงในสคริป RSD ทำการบันทึก

- เปิดหน้าต่างคำสั่งใหม่ จากนั้นสั่งให้ RSD เริ่มทำงานทำงานด้วยคำสั่ง

```
sudo stap RSD_Script.stp [enter]
```

จะปรากฏข้อความ “RSD Started.” ซึ่งแสดงว่า RSD ทำงานเรียบร้อย

[††] สภาพแวดล้อมที่กำหนดคือ ติดตั้งเฉพาะซอฟต์แวร์มาตรฐานของระบบปฏิบัติการ Ubuntu Linux และไม่เชื่อมต่อเครือข่าย

ถ้าต้องการให้ RSD แสดงผลเฉพาะการตรวจหา NRSM สามารถทำได้โดยใช้คำสั่ง

```
sudo stap RSD_Script.stp | grep NRSM [enter]
```

หรือถ้าต้องการให้ RSD บันทึกรายงานผลการตรวจหาไปยังไฟล์ สามารถทำได้โดย

```
sudo stap RSD_script.stp > [ชื่อไฟล์ที่ต้องการบันทึก] [enter]
```

- ตรวจสอบหมายเลขโปรเซสของซอฟต์แวร์เป้าหมาย เพื่อใช้หมายเลขโปรเซสนี้ในการแพร่ปรสิต ซึ่งสามารถตรวจสอบหมายเลขของโปรเซสที่ทำงานอยู่บนระบบปฏิบัติการทั้งหมดได้โดยเปิดหน้าต่างคำสั่งใหม่จากนั้นพิมพ์คำสั่ง

```
ps ax [enter]
```

หรือตรวจสอบเฉพาะหมายเลขของโปรเซสที่ทราบชื่อด้วยคำสั่ง

```
ps ax | grep [ชื่อโปรเซส] [enter]
```

รายการโปรเซสและหมายเลขโปรเซสจะถูกแสดงออกสู่หน้าจอแสดงผล

- ทำการแพร่ซอฟต์แวร์ปรสิต ไปยังซอฟต์แวร์เป้าหมายด้วยคำสั่ง

```
./dlh [หมายเลขโปรเซส] puts [enter]
```

จากนั้นตรวจสอบว่าปรสิตแพร่ไปยังซอฟต์แวร์เป้าหมายได้หรือไม่ ด้วยการตรวจสอบผังหน่วยความจำของโปรเซสเป้าหมายโดยใช้คำสั่ง

```
cat /proc/[หมายเลขโปรเซส]/maps [enter]
```

จะปรากฏรายการผังหน่วยความจำที่โปรเซสเป้าหมายใช้งานอยู่ทั้งหมด ซึ่งไลบรารี่ของซอฟต์แวร์ปรสิตที่ใช้ทดสอบมีชื่อว่า “libtest.so.1.0” ถ้าตรวจพบชื่อไลบรารี่ดังกล่าวนั่นหมายถึงซอฟต์แวร์ปรสิตทำงานได้อย่างสมบูรณ์

- ตรวจสอบการรายงานผลของ RSD ว่าสามารถตรวจจับการแก้ไขที่เกิดขึ้นได้หรือไม่ โดยเปลี่ยนกลับไปยังหน้าต่างคำสั่งที่สั่งให้ RSD_Script.stp ทำงาน ซึ่งจะปรากฏการรายงานผลการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานได้ 3 กรณีดังนี้

    - กรณีที่มีการตรวจพบการเปลี่ยนแปลงระหว่างโปรเซสการรายงานผลจะเป็นดังนี้

      [ชื่อโปรเซสที่ถูกเปลี่ยนแปลง] ([หมายเลขโปรเซสที่ถูกเปลี่ยนแปลง]) NRSM by [หมายเลขโปรเซสที่เป็นผู้เปลี่ยนแปลง]

      ตัวอย่าง

      gnome-screensaver(2178) NRSM by 3856

    - กรณีที่มีการตรวจพบโอกาสในการเปลี่ยนแปลงข้อมูลในโปรเซสการรายงานผลจะเป็นดังนี้

      [ชื่อโปรเซส] ([หมายเลขโปรเซส]) has a chance to modification

      ตัวอย่าง

      metacity(2075) has a chance to modification

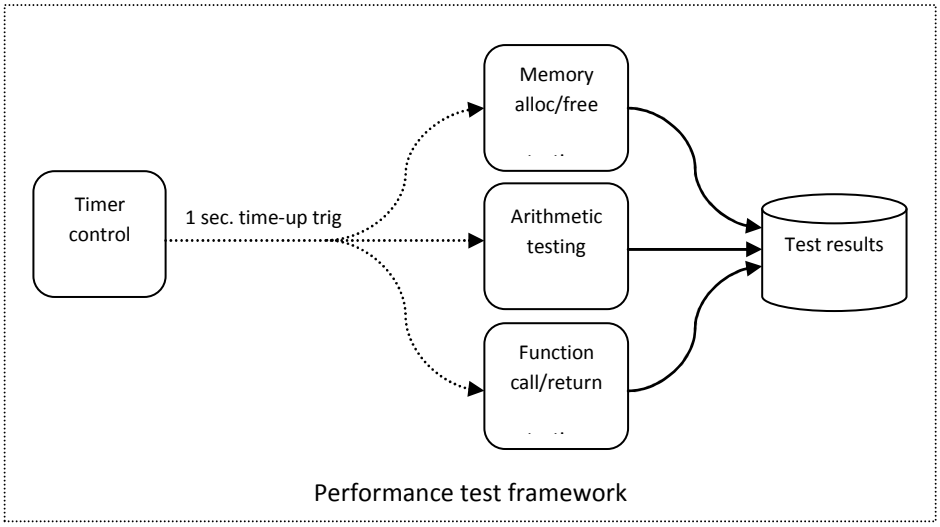
การทดสอบการทำงานของ RSD จะกระทำโดย แพร่ซอฟต์แวร์ปรสิตไปยังซอฟต์แวร์ที่ใช้งานจริงบนระบบปฏิบัติการลินุกซ์ จากนั้นจึงทำการตรวจสอบผลการทำงานของ RSD ว่าสามารถตรวจหาความเปลี่ยนแปลงที่เกิดขึ้นได้หรือไม่ โดยผลการตรวจหาฯ ได้นำเสนอในบทที่ 4

3.3.4 การประเมินผลกระทบของ RSD ที่มีต่อระบบปฏิบัติการ

ในส่วนนี้จะกล่าวถึงการออกแบบการประเมินผลกระทบของ RSD ที่มีต่อระบบปฏิบัติการที่ทำงานอยู่ ซึ่งเมื่อกล่าวถึงการทำงานของซอฟต์แวร์ที่เกี่ยวข้องกับความปลอดภัยของซอฟต์แวร์ หรือกล่าวถึงซอฟต์แวร์ที่ทำงานเกี่ยวกับความปลอดภัยของระบบคอมพิวเตอร์ นอกจากความสามารถและความแม่นยำในการทำงานของซอฟต์แวร์นั้นๆ แล้ว ผลกระทบต่อระบบปฏิบัติการในขณะที่ซอฟต์แวร์ทำงาน เป็นอีกปัญหาที่มีความสำคัญเช่นกัน ตัวอย่างเช่น ถ้าสามารถสร้างระบบตรวจหาฯ ที่มีความสามารถสูง โดยสามารถตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ได้ทุกรูปแบบ แต่ถ้าการทำงานของซอฟต์แวร์ดังกล่าวนั้น ทำให้การทำงานโดยรวมของระบบปฏิบัติการลดประสิทธิภาพลง 50% หรือมากกว่า นั่นย่อมแสดงว่าวิธีการทำงานของซอฟต์แวร์นั้นไม่มีประสิทธิภาพ หรือรบกวนระบบปฏิบัติการอย่างมากนั่นเอง ดังนั้นการประเมินผลการทำงานของการตรวจหาฯ ย่อมมีความสำคัญเช่นกัน

วิธีการทดสอบผลกระทบต่อระบบปฏิบัติการในขณะที่ RSD ทำงานนั้น ผู้วิจัยได้เลือกวิธีการทดสอบด้วยการทดสอบภาระ [20] (Load testing) ซึ่งเน้นทดสอบความเร็วที่ลดลงของระบบโดยรวม การทดสอบนี้กระทำโดยสร้างโปรแกรมประเมินผลกระทบของ RSD ที่มีต่อระบบปฏิบัติการ โดยโปรแกรมประเมินฯ จะทดสอบความเร็วในการทำงานพื้นของซอฟต์แวร์ 3 ประการดังนี้

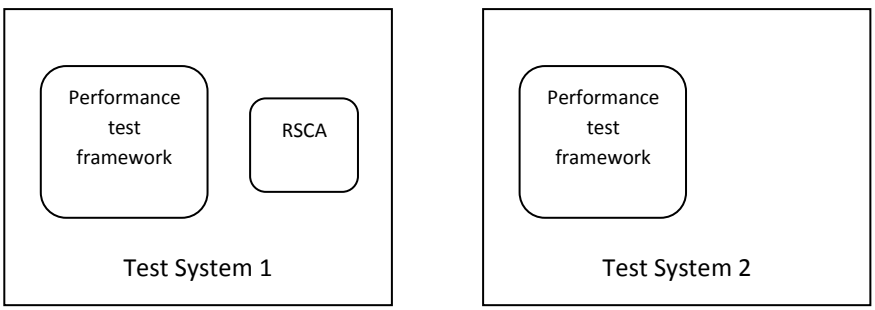
- ความเร็วในการจองและคืนหน่วยความจำ (Memory allocate/deallocate)
- ความเร็วในการคำนวณทางคณิตศาสตร์ (Arithmetic calculation)
- ความเร็วในการเรียกและกลับจากฟังก์ชัน (Function call/return)



ภาพที่ 3.3.4-1 การทำงานของโปรแกรมประเมินผลกระทบของ RSD

ภาพที่ 3.3.4-1 แสดงการทำงานของโปรแกรมประเมินฯ โดยจะนับจำนวนครั้งการทำงานที่แต่ละส่วนทดสอบสามารถทำได้ใน 1 วินาที โดยจะมีส่วนควบคุมเวลาเป็นตัวกำกับการทำงานของทุกส่วน เมื่อครบทุก 1 วินาที แต่ละส่วนทดสอบจะทำการบันทึกผลลงไฟล์ เพื่อใช้ในการการประเมินฯ ต่อไป รายละเอียดของซอฟต์แวร์ทดสอบแสดงไว้ใน ภาคผนวก ค

การประเมินผลกระทบของ RSD ที่มีต่อระบบ กระทำโดยเปรียบเทียบระบบปฏิบัติการ 2 ระบบ โดยทั้ง 2 ระบบ จะมีโปรแกรมประเมิน ทำงานอยู่ แต่มีเพียง 1 ระบบที่มี RSD ทำงานอยู่ โดยมีข้อกำหนดว่าสภาพแวดล้อม เช่น ฮาร์ดแวร์ และ โปรแกรมอื่นๆ ที่ทำงานบนระบบ ต้องเหมือนกันทุกประการ การเปรียบเทียบผลการประเมินฯ แสดงได้ดังภาพที่ 3.3.4-2



ภาพที่ 3.3.4-2 การเปรียบเทียบผลกระทบของ RSD ที่มีต่อระบบปฏิบัติการ

แต่ละหัวข้อของการทดสอบจะกำหนดให้โปรแกรมประเมินฯ ทำงานทั้งหมด 1000 ครั้ง จากนั้นนำผลการทำงานที่ได้ทั้งหมดมาคำนวณหาค่าเฉลี่ยใน 1 วินาทีว่าแต่ละส่วนสามารถทำงานได้จำนวนกี่รอบ ผลการประเมินการทำงาน สามารถดูรายละเอียดได้ในบทที่ 4

บทที่ 4
ผลการวิเคราะห์ข้อมูล

ในส่วนนี้เป็นการรายงานผลการทดสอบการทำงานของ RSD ที่ได้นำเสนอไปในบทที่ 3 โดยผลการทดสอบจะประกอบด้วย 2 ส่วนคือ ผลการทำงานในการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน ด้วยการจำลองการเปลี่ยนแปลงโดยใช้ซอฟต์แวรปรสิต และผลการทดสอบอีกส่วนหนึ่งคือ ผลของการประเมินผลกระทบในการทำงานของ RSD ที่มีต่อระบบปฏิบัติการ ซึ่งรายละเอียดของทั้ง 2 ส่วนมีดังนี้

4.1 ผลการทำงานของ RSD

จากวิธีการทดสอบการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงาน ด้วยการแพร่ซอฟต์แวร์ปรสิต ไปยังซอฟต์แวร์ที่ใช้งานจริงบนระบบปฏิบัติการลินุกซ์ 8 รายการ รายชื่อซอฟต์แวร์ที่ทำการทดสอบ และผลการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ที่ RSD สามารถตรวจหาได้แสดงในตารางที่ 4.1

ตารางที่ 4.1 ผลการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน

Target Software	NRSM	Fault negative
OpenOffice	Yes	No
Nautilus	Yes	No
Gnome-terminal	Yes	No
Update-manager	Yes	No
Qcalctool	Yes	No
Gnome-nettool	Yes	No
Mozilla-firefox	N/A	N/A
Google chrome	N/A	N/A

รายละเอียดวิธีการอ่านผลการทดสอบในตารางที่ 4.1 สามารถอธิบายได้ดังนี้

ในคอลัมน์แรกเป็นรายชื่อของซอฟต์แวร์ที่ทำการทดสอบการแพร่กระจายของซอฟต์แวร์ปรสิตทั้งหมด คอลัมน์ที่สอง NRSM เป็นการรายงานว่า RSD ตรวจพบว่ามีการ

เปลี่ยนแปลงซอฟต์แวร์ระหว่างโปรเซสหรือไม่ ซึ่งในที่นี้หมายถึง RSD ตรวจสอบได้ว่ามีการแพร่ปรสิตไปยังซอฟต์แวร์ทดสอบหรือไม่ "Yes" หมายถึง RSD สามารถตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานที่เกิดขึ้นได้ ซึ่งตามผลการทดสอบพบว่า RSD สามารถตรวจพบการเปลี่ยนแปลงซอฟต์แวร์ของซอฟต์แวร์ที่เกิดจากปรสิตได้เกือบทั้งหมด ยกเว้นเพียง Mozilla Firefox และ Google Chrome นั้นไม่สามารถสรุปการตรวจหาได้ เนื่องจากซอฟต์แวร์ปรสิตไม่สามารถแพร่ไปยังซอฟต์แวร์เป้าหมายได้ จากสมมุติฐานที่ว่าซอฟต์แวร์ปรสิตจะอาศัยช่องโหว่ของไฟล์แบบ ELF ร่วมกับการทำงานของคำสั่ง "ld" ซึ่ง Mozilla Firefox และ Google chrome ไม่ได้มีการใช้ไลบรารี่ที่ไม่ปลอดภัยร่วมกับโปรเซสอื่น ดังนั้นผลการทดสอบจึงรายงาน "N/A" (Not Applicable) ซึ่งจะต้องหาวิธีอื่นในการแก้ไข Mozilla Firefox และ Google Chrome ซึ่งไม่อยู่ในขอบเขตของงานวิจัยนี้

ในช่อง Fault Negative เป็นการแสดงว่า RSD รายงานผลที่มีความผิดพลาดด้านลบหรือไม่ ตัวอย่างของการรายงานผลผิดพลาดด้านลบเช่น มีการแพร่กระจายของซอฟต์แวร์ปรสิตไปยังซอฟต์แวร์เป้าหมายเป็นที่เรียบร้อย แต่ RSD รายงานผลการตรวจหาว่าเป็นปกติซึ่งเป็นการรายงานที่ผิดพลาดร้ายแรง เป็นต้น "No" แสดงว่าไม่มีการรายงานผลที่ผิดพลาดด้านลบเกิดขึ้น

จากการทดสอบการทำงานของ RSD มีการตรวจพบการรายงานผลที่เป็น Fault Positive ในกรณีที่มีการดีบักโปรแกรม (Program debugging) ของการใช้งานโปรแกรมประเภท IDE (Integrated Development Environment) เช่น QtDesigner[‡‡] และ Netbean[§§] ซึ่งเมื่อมีการดีบักโปรแกรม ไม่ว่าจะเป็นการหยุดการทำงานของโปรแกรมชั่วคราว (break) หรือ แก้ไขตัวแปรภายในโปรแกรมในระหว่างทำการดีบักโปรแกรม RSD จะรายงานว่าตรวจพบ NRSM ซึ่งเป็นการรายงานที่ผิดพลาดในแง่บวก ซึ่งเป็นการผิดพลาดที่ไม่เกิดผลเสียแต่อย่างใด

[‡‡] http://qt.nokia.com

[§§] http://netbeans.org

4.2 ผลกระทบของ RSD ต่อระบบปฏิบัติการ

จากการทดสอบผลกระทบการทำงานของ RSD ที่มีต่อระบบปฏิบัติการด้วยการทดสอบภาระ ในหัวข้อที่ 3.3.4 สามารถสรุปผลการทดสอบได้ดังตารางที่ 4.2

ตารางที่ 4.2 แสดงผลการทดสอบผลกระทบต่อระบบปฏิบัติการของ RSD

Load test cases	**Number of loop in 1 sec. without RSD running (average of 1000 samples)**	**Number of loop in 1 sec. with RSD running (average of 1000 samples)**	**Performance comparison (Slow down ratio)**
Memory allocation/free	292254	268272	0.918
Arithmetic calculation	1518488	1431902	0.943
Function call/return	1502089	1403504	0.934

ข้อมูลตารางที่ 4.2 แสดงให้เห็นถึงผลการทดสอบภาระการทำงานของระบบปฏิบัติการ ในขณะที่มี RSD และไม่มี RSD ทำงานอยู่ โดยทดสอบการทำงานพื้นฐานของซอฟต์แวร์ 3 ประการได้แก่ การทดสอบการจองและคืนหน่วยความจำ (Memory allocation/free) การทดสอบการคำนวณทางคณิตศาสตร์ (Arithmetic calculation) และการทดสอบการไปและกลับจากการเรียกฟังก์ชัน ซึ่งหัวข้อของการทดสอบจะแสดงในคอลัมน์แรกตามลำดับ ในคอลัมน์ที่ 2 แสดงจำนวนรอบการทำงานเฉลี่ย 1000 ครั้ง ของแต่ละหัวข้อการทดสอบภายในเวลา 1 วินาที ในขณะที่ไม่มี RSD ทำงานอยู่ ในคอลัมน์ที่ 3 เป็นจำนวนรอบการทำงานโดยเฉลี่ย 1000 ครั้ง ของแต่ละหัวข้อการทดสอบภายในเวลา 1 วินาทีในขณะที่มี RSD ทำงานอยู่ และคอลัมน์สุดท้ายเป็นการเปรียบเทียบความเร็วในการทำงาน ระหว่างระบบที่มี RSD และระบบที่ไม่มี RSD โดยประยุกต์ใช้สมการ Speed up/ slow down [21] ดังสมการ 4.2

$$rs = \frac{t1}{t2}$$ สมการที่ 4.2

เมื่อ rs คือ อัตราส่วนความเร็วของระบบที่เปลี่ยนไป

$t1$ คือ เวลาที่ใช้ในการทำงานของระบบที่ไม่มี RSD ทำงานอยู่

$t2$ คือ เวลาที่ใช้ในการทำงานของระบบที่มี RSD ทำงานอยู่

จากผลการทดสอบดังกล่าวสามารถสรุปได้ว่า ในขณะ RSD ทำงานนั้น ส่งผลกระทบต่อการทำงานของระบบปฏิบัติการ โดยทำให้ระบบปฏิบัติการทำงานช้าลงโดยเฉลี่ยประมาณ 7% รายละเอียดฮาร์ดแวร์ที่ใช้ในการทดสอบแสดงไว้ใน ภาคผนวก จ

บทที่ 5
สรุปผลการวิจัย และข้อเสนอแนะ

ในส่วนนี้เป็นการอธิบายถึงข้อสรุปที่ได้จากงานวิจัยทั้งหมด ซึ่งรวมถึงประสิทธิภาพ และการทำงานโดยรวมเป็นไปตามความคาดหวังหรือไม่ นอกจากนี้ส่วนนี้ยังได้กล่าวถึงข้อจำกัด และข้อเสนอแนะที่สำคัญต่างๆ เพื่อเป็นแนวทางสำหรับงานวิจัยที่จะนำวิธีการทำ RSD ไปประยุกต์ให้เกิดประโยชน์ต่อไปในอนาคต

5.1 สรุปผลการวิจัย

จากที่งานวิจัยนี้ได้นำเสนอวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ในขณะทำงานด้วยการตรวจหาการเรียกใช้งานคำสั่งระบบของโปรเซส โดยอาศัยการทำงานของ Systemtap โดยคาดหวังว่าวิธี RSD ที่นำเสนอนั้นจะสามารถทำงานได้รวดเร็วกว่าวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบอื่นๆ และคาดหวังว่าการทำงานของ RSD นั้นรบกวนการทำงานของระบบปฏิบัติการน้อยกว่าวิธีอื่นนั้น

ในภาพรวมสามารถสรุปได้ว่าวิธีการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานแบบ RSD ที่นำเสนอนั้น สามารถทำงานได้เร็วกว่าวิธีอื่น เพราะสามารถตรวจหาการเปลี่ยนแปลงที่เกิดขึ้นได้ในทันทีโดยไม่ต้องอาศัยข้อมูลของระบบในสภาวะปกติ และจากการทดสอบประสิทธิภาพในการทำงานสามารถสรุปได้ว่า RSD รบกวนการทำงานของระบบปฏิบัติการโดยรวมประมาณ 7% ซึ่งถือได้ว่าเป็นผลกระทบที่ไม่มาก นอกจากนี้ RSD ยังสามารถเรียกใช้งานได้ง่ายเพราะสามารถทำงานได้โดยเรียกใช้งานสคริปเพียงครั้งเดียว อีกทั้งมีความปลอดภัยในการทำงานเพราะใช้เครื่องมือที่ถูกออกแบบมาเพื่อประเมินประสิทธิภาพในการทำงานของระบบปฏิบัติการโดยเฉพาะ ซึ่งทำให้โดยภาพรวมการทำงานของ RSD อยู่ในเกณฑ์ค่อนข้างดี

5.2 ข้อจำกัด

การทำงานของ RSD ที่ได้นำเสนอไปแล้วนั้น เป็นเพียงวิธีการต้นแบบสำหรับทดสอบการทำงานของการตรวจสอบการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน จึงทำให้มีข้อจำกัดในการทำงานอยู่ในหลายจุด ซึ่งสามารถสรุปได้ดังนี้

1. สามารถตรวจจับการเปลี่ยนแปลงที่เกิดขึ้นจากการทำงานของซอฟต์แวร์ปรสิตเท่านั้น ซึ่งยังมีการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานอีกหลายประเภทที่การตรวจหายังไม่ครอบคลุม

2. ยังไม่สามารถทำการกำหนดได้ว่า การเปลี่ยนแปลงประเภทใดเป็นการเปลี่ยนแปลงที่ไม่สร้างความเสียหายต่อ ระบบเพื่อเป็นการลดการทำงานของ RSD และเพิ่มความเร็วในการตรวจหาให้สูงขึ้น

3. เนื่องจาก RSD อาศัยการทำงานของ Systemtap เป็นหลัก จึงทำให้สามารถตรวจจับการทำงานของซอฟต์แวร์ได้ในระดับผู้ใช้งานเท่านั้น ซึ่งในระบบปฏิบัติการลินุกซ์นั้นยังมีซอฟต์แวร์ที่ทำงานอยู่ในระดับล่าง อย่างเช่น โมดูล หรือ ดีไวซ์ไดรเวอร์ ซึ่งทำงานในระดับที่ต่ำกว่าการทำงานของ Systemtap จึงทำให้ RSD ไม่สามารถตรวจหาการเปลี่ยนแปลงที่เกิดขึ้นกับซอฟต์แวร์ดังกล่าวได้

4. การทดสอบการทำงานของ RSD นั้นกระทำเฉพาะบนลินุกซ์เคอร์เนล 2.6 แบบ 32 บิต เท่านั้น ซึ่งยังไม่ครอบคลุมเคอร์เนลที่ใหม่กว่า อย่างเช่น เคอร์เนล 3.0 และยังไม่ครอบคลุมการทำงานบนระบบปฏิบัติการ 64 บิต ด้วยเช่นกัน

5. ผลการทดสอบการทำงานกระทำกับซอฟต์แวร์ในจำนวนที่จำกัด ซึ่งยังไม่ครอบคลุมซอฟต์แวร์ทุกตัว ที่ทำงานอยู่บนระบบปฏิบัติการ หรือซอฟต์แวร์ที่ผู้ใช้งานอาจจะติดตั้งเพิ่มเติมในอนาคต

5.3 ข้อเสนอแนะ

สำหรับงานวิจัยที่ต้องการศึกษาเกี่ยวกับการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงานโดยการวิเคราะห์การเรียกใช้งานคำสั่งระบบในอนาคต ควรพิจารณาข้อเสนอแนะที่ได้จากข้อจำกัดของงานวิจัย ซึ่งสามารถสรุปได้ดังนี้

1. ควรเพิ่มรูปแบบในการตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ขณะทำงาน ให้ครอบคลุมการเปลี่ยนแปลงซอฟต์แวร์ที่มีอยู่ให้มากขึ้น ตัวอย่างเช่น การตรวจหา Buffer overflow หรือ การตรวจหาการเปลี่ยนแปลงซอฟต์แวร์ของไวรัส หรือมัลแวร์ในรูปแบบใหม่ๆ เป็นต้น

2. ควรเพิ่มเติมรายการเปลี่ยนแปลงที่ไม่ก่อให้เกิดความเสียหาย หรือ White List เพื่อลดการทำงานของ RSD และช่วยเพิ่มความเร็วในการทำงานของระบบโดยรวม

3. ควรศึกษาพัฒนาเครื่องมือประเภทอื่น ที่สามารถตรวจสอบการทำงานภายในเคอร์เนล โดยไม่ยึดติดกับ Systemtap เพื่อทำให้ RSD สามารถตรวจหาการเปลี่ยนแปลงที่เกิดขึ้นกับซอฟต์แวร์ในระดับเคอร์เนลได้

4. ขยายการทดสอบการทำงานของ RSD ไปยังระบบปฏิบัติการประเภทอื่น ตัวอย่างเช่น ลินุกซ์ตระกูลอื่น ยูนิกซ์ หรือ Windows เพื่อเปรียบเทียบประสิทธิภาพในการทำงานว่าแตกต่างกันอย่างไร

5. ขยายการทดสอบการทำงานของ RSD ไปยังซอฟต์แวร์ที่ทำงานอยู่บนระบบปฏิบัติการลินุกซ์ให้มากขึ้น เพื่อทราบถึงข้อจำกัด และความสามารถของ RSD ว่าสามารถนำไปใช้งานจริงได้มากน้อยเพียงใด

รายการอ้างอิง

[1] Foster, James C.; and Price, Mike. Sockets, Shellcode, Porting, & Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals. Elsevier Science & Technology Books. 2005.

[2] James Oakley and Sergey Bratus., Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. In Proceedings of the 5th USENIX conference on Offensive technologies (WOOT'11). USENIX Association, Berkeley, CA, USA, 11-11, 2011.

[3] Myles, G. (n.d.). The Use of Software-Based Integrity Checks in Software Tamper Resistance Techniques. IBM. IBM Almaden Research Center, 2005

[4] Chen, Venkatesan, Cary, Pang, Sinha, and Jakubowski, Oblivious Hashing: A Stealthy Software Integrity Verification Primitive, Proc. of 5th International Workshop on Information Hiding, 2002.

[5] Wagner, D., and Dean, R. (n.d.). Intrusion detection via static analysis. Proceedings 2001 IEEE Symposium on Security and Privacy, pp.156-168, 2001.

[6] Shahzad, F., Bhatti, S., Shahzad, M., and Farooq, M. (n.d.). In-Execution Malware Detection using Task Structures of Linux Processes. IEEE International Conference on Communications (ICC), 2011.

[7] Nguyen, N., Reiher, P., and Kuenning, G. H., Detecting Insider Threats by Monitoring System Call Activity. IEEE Workshop on information assurance, 2003.

[8] Das, S., Chattopadhayay, A., Kalyani, D. K., and Saha, M., File-system Intrusion Detection by preserving MAC DTS: A Loadable Kernel Module based approach for LINUX Kernel. CSIIRW, 2009.

[9] Andrew P. Kosoresow, Steven A. Hofmeyr, "Intrusion Detection via System Call Traces," IEEE Software, vol. 14, no. 5, pp.35-42, 1997.

[10] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P., A fast automaton-based method for detecting anomalous program behaviors. Proceedings 2001 IEEE Symposium on Security and Privacy, 9(C), 144-155, 2001.

[11] Feng, H. H., Kolesnikov, O. M., Fogla, P., and Lee, W. (n.d.). Anomaly detection using call stack information. Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), pp.62-75.

[12] Federico Maggi, Stefano Zanero, and Vincenzo Iozzo., Seeing the invisible: forensic uses of anomaly detection and machine learning. SIGOPS Oper. Syst. Rev. 42, pp. 51-58, 2008.

[13] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel., Anomalous system call detection, ACM Trans. Inf. Syst. Secur. 9, pp.61-93, 2006.

[14] Wolfgang Mauerer, Professional Linux® Kernel Architecture, Wiley Publishing, 2008.

[15] Bovet, D. P., and Cesati, M., Understanding the Linux Kernel. (First.). O'Reilly, 2000.

[16] Eigler, F. C., Architecture of systemtap: a Linux trace / probe tool Systemtap processing steps, 2005.

[17] Robb, Romans, SystemTap Language Reference, Boston, USA.: Red Hat, IBM Corp., Intel Corp. pp.1-46, 2010.

[18] Bartolich A., "The ELF Virus Writing HOWTO" ,http://www.ouah.org/virus-writing-HOWTO/index.html, 2002.

[19] O'Neill R., Modern Day ELF Runtime infection via GOT poisoning, 2009. Available from: http://www.vxheavens.com/lib/vrn00.html

[20] Segue Software, Choosing A Load Testing Strategy, 2005. Available from: http://www.iiquality.com/articles/load_testing.pdf

[21] Rodgers, David P., Improvements in multiprocessor system design. ACM SIGARCH Computer Architecture News archive (New York, NY, USA: ACM) 13 (3): pp.225–231, 1985.

ภาคผนวก

ภาคผนวก ก
รายละเอียดของคำสั่งระบบเฉพาะแบบ[1]

No.	System calls	Descriptions
1	*ptrace*	NAME ptrace - process trace SYNOPSIS #include <sys/ptrace.h> long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data); DESCRIPTION The ptrace system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing. The parent can initiate a trace by calling fork(2) and having the resulting child do a PTRACE_TRACEME, followed (typically) by an exec(3). Alternatively, the parent may commence trace of an existing process using PTRACE_ATTACH. While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. (The exception is SIGKILL, which has its usual effect.) The parent will be notified at its next wait(2) and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead). When the parent is finished tracing, it can terminate the child with PTRACE_KILL or cause it to continue executing in a normal, untraced mode via PTRACE_DETACH.

2	*mmap*	NAME
		
		mmap, munmap - map or unmap files or devices into memory
		SYNOPSIS
		
		#include <sys/mman.h>
		#ifdef _POSIX_MAPPED_FILES
		
		void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
		
		int munmap(void *start, size_t length);
		
		#endif
		
		DESCRIPTION
		
		The mmap function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap, and is never 0.
		The prot argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either PROT_NONE or is the bitwise OR of one or more of the other PROT_* flags.

3	*munmap*	NAME mmap, munmap - map or unmap files or devices into memory SYNOPSIS #include <sys/mman.h>

[1] http://linux.about.com/library/cmd

		#ifdef _POSIX_MAPPED_FILES
		
		void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
		
		int munmap(void *start, size_t length);
		
		#endif
		
		DESCRIPTION
		
		The mmap function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap, and is never 0.
		The prot argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either PROT_NONE or is the bitwise OR of one or more of the other PROT_* flags.

4	*mremap*	NAME mremap - re-map a virtual memory address SYNOPSIS #include <unistd.h> #include <sys/mman.h> void * mremap(void *old_address, size_t old_size , size_t new_size, unsigned long flags); DESCRIPTION mremap expands (or shrinks) an existing memory mapping, potentially moving it at the same time (controlled by the flags argument and the available virtual address space). old_address is the old address of the virtual memory block that you want to expand (or shrink). Note that old_address has to be page aligned. old_size is the old size of the virtual memory block. new_size is the requested size of the virtual memory block after the resize. The flags argument is a bitmap of flags. In Linux the memory is divided into pages. A user process has (one or) several linear virtual memory segments. Each virtual memory segment has one or more mappings to real memory pages (in the page table). Each virtual memory segment has its own protection (access rights), which may cause a segmentation violation if the memory is accessed incorrectly (e.g., writing to a read-only segment). Accessing virtual memory outside of the segments will also cause a segmentation violation. mremap uses the Linux page table scheme. mremap changes the mapping between virtual addresses and memory pages. This can be used to implement a very efficient realloc.

5	*brk*	NAME
		
		brk, sbrk - change data segment size
		SYNOPSIS
		
		#include <unistd.h>
		int brk(void *end_data_segment);
		
		void *sbrk(ptrdiff_t increment);
		
		DESCRIPTION
		
		brk sets the end of the data segment to the value specified by end_data_segment, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size (see setrlimit(2)).
		sbrk increments the program's data space by increment bytes. sbrk isn't a system call, it is just a C library wrapper. Calling sbrk with an increment of 0 can be used to find the current location of the program break.

6	*mprotect*	NAME mprotect - control allowable accesses to a region of memory

		SYNOPSIS #include <sys/mman.h> int mprotect(const void *addr, size_t len, int prot); DESCRIPTION mprotect controls how a section of memory may be accessed. If an access is disallowed by the protection given it, the program receives a SIGSEGV. prot is a bitwise-or of the following values: PROT_NONE The memory cannot be accessed at all. PROT_READ The memory can be read. PROT_WRITE The memory can be written to. PROT_EXEC The memory can contain executing code. The new protection replaces any existing protection. For example, if the memory had previously been marked PROT_READ, and mprotect is then called with prot PROT_WRITE, it will no longer be readable.
7	*mlock*	NAME mlock - disable paging for some parts of memory SYNOPSIS #include <sys/mman.h> int mlock(const void *addr, size_t len); DESCRIPTION mlock disables paging for the memory in the range starting at addr with length len bytes. All pages which contain a part of the specified memory range are guaranteed be resident in RAM when the mlock system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked by munlock or munlockall, until the pages are unmapped via munmap, or until the process terminates or starts another program with exec. Child processes do not inherit page locks across a fork. Memory locking has two main applications: real-time algorithms and high-security data processing. Real-time applications require deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with sched_setscheduler. Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets could be transfered onto a persistent swap store medium, where they might be accessible to the enemy long after the security software has erased the secrets in RAM and terminated. Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlock for the corresponding range or by munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process. On POSIX systems on which mlock and munlock are available, _POSIX_MEMLOCK_RANGE is defined in <unistd.h> and the value PAGESIZE from <limits.h> indicates the number of bytes per page.
8	*mlockall*	NAME mlockall - disable paging for calling process SYNOPSIS #include <sys/mman.h> int mlockall(int flags); DESCRIPTION

		mlockall disables paging for all pages mapped into the address space of the calling process. This includes the pages of the code, data and stack segment, as well as shared libraries, user space kernel data, shared memory and memory mapped files. All mapped pages are guaranteed to be resident in RAM when the mlockall system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked again by munlock or munlockall or until the process terminates or starts another program with exec. Child processes do not inherit page locks across a fork. Memory locking has two main applications: real-time algorithms and high-security data processing. Real-time applications require deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with sched_setscheduler. Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets could be transfered onto a persistent swap store medium, where they might be accessible to the enemy long after the security software has erased the secrets in RAM and terminated. For security applications, only small parts of memory have to be locked, for which mlock is available.

ภาคผนวก ข

RSD SCRIPT

```
/*
        RSD_script.stp

        Used for detect run-time software modification via restricted system calls calling.
        This script can run on Linux OS with SystemTap and Kprobes installed.
        This script was tested with Ubuntu 10.10.

        Created by :     Sathaporn Sa-ngounwong
                         Chulalongkorn University, 2011.
*/

global TargetProcessMap
global NRSM_list
global CHANCE_list


/* other process modify system call */
probe kernel.function("sys_ptrace") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_ptrace\n",execname(),pid())

        TargetProcessMap[$pid] = pid()
}


/* self modifier system call */
probe kernel.function("sys_brk") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_brk\n",execname(),pid())

        if(target_list[pid()] != 0)
        {
                printf("%s(%d) NRSM by %d\n",execname(),pid(),TargetProcessMap[pid()])
                NRSM_list[pid(),"sys_brk"] <<< 1
        }

}


probe kernel.function("sys_old_mmap") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_old_mmap\n",execname(),pid())

        if(target_list[pid()] != 0)
        {
                printf("%s(%d) NRSM by %d\n",execname(),pid(),TargetProcessMap[pid()])
                NRSM_list[pid(),"sys_old_mmap"] <<< 1
        }

}

probe kernel.function("sys_munmap") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_munmap\n",execname(),pid())

        if(target_list[pid()] != 0)
        {
                printf("%s(%d) NRSM by %d\n",execname(),pid(),TargetProcessMap[pid()])
                NRSM_list[pid(),"sys_munmap"] <<< 1
        }


}
```

```
probe kernel.function("sys_mremap") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_mremap\n",execname(),pid())

        if(target_list[pid()] != 0)
        {
                printf("%s(%d) NRSM by %d\n",execname(),pid(),TargetProcessMap[pid()])
                NRSM_list[pid(),"sys_mremap"] <<< 1
        }

}



/* memory content modify system call */
probe kernel.function("sys_mprotect") {
        if($prot & 0x2){
                //!!!! uncomment below code for debugging !!!!
                //printf("%s(%d) call sys_mprotect for write\n",execname(),pid())

                printf("%s(%d) has a chance to modification\n",execname(),pid())
                CHANCE_list[pid(),"sys_mprotect"] <<< 1
        }
}


probe kernel.function("sys_mlock") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_mlock\n",execname(),pid())

        printf("%s(%d) has a chance to modification\n",execname(),pid())
        CHANCE_list[pid(),"sys_mlock"] <<< 1
}


probe kernel.function("sys_mlockall") {
        //!!!! uncomment below code for debugging !!!!
        //printf("%s(%d) call sys_mlockall\n",execname(),pid())

        printf("%s(%d) has a chance to modification\n",execname(),pid())
        CHANCE_list[pid(),"sys_mlockall"] <<< 1
}
```

ภาคผนวก ค

โปรแกรมสำหรับประเมินผลกระทบของ RSD ต่อระบบปฏิบัติการ

```
/*****************************************************************************
* perf_test.c
*
* Created by Sathaporn S. Jul 2011
*
*****************************************************************************
*
* this program used for exercise system by test following criterias.
*       1. allocate/deallocate memory
*       2. large number arithemetic calculation
*       3. call/return function
*
* This test will create 4 processes,
*       First is         main process for controling all sub processes.
*       Second is        memory allocation/free testing process.
*       Third is arithmatic testing process.
*       Fourth is        function call/return testing process.
*
* USAGE:
*       command format :
*                ./perf_test [wait time] [loop number]
*
*                - [wait time] is interval time that first process let
*                  the second process to collect the test data
*
*                - [loop number] is number of time to test
*
*       program will automatically terminate after reach the loop number
*
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <fstream>
#include <string.h>


#define WAIT_TIME1        /* default wait time */
#define START_SIGNAL      0        /* start signal */
#define STOP_SIGNAL       1        /* stop signal */
#define BUF_SIZE 100      /* memory allocation buffer size */

using namespace std;


void test_function(int i)
{
        long a = i*0xFFFFFFFF;
        long b = i*0xFFFFFFFF;
        long c = i*0xFFFFFFFF;
        long d = i*0xFFFFFFFF;
        long e = i*0xFFFFFFFF;
        long f = i*0xFFFFFFFF;
        long g = i*0xFFFFFFFF;
}
```

```
int main(int argc,char** argv)
{
        int wait_time=WAIT_TIME;
        int loop=1;

        /* get inline parameters */
        /*  */
        if(argc > 1)
        {
                /* get wait time value */
                wait_time = atoi(argv[1]);
                /* get loop number */
                if(argc > 2) loop = atoi(argv[2]);
                else loop = 1;

        }


        /* pipe file descripter */
        int pipefd[2];

        /* signal data buffer */
        char sigbuf=0;

        /* create pipe   */
        if(pipe2(pipefd,O_NONBLOCK)==-1)
        {
                printf("create pipe error\n");
                exit(-1);
        }

        /* first process create */
        /* the second process */
        pid_t pid = fork();

        /* for second process do */
        if(pid==0)
        {
                /* create result output file */
                ofstream fp_out;
                fp_out.open("malloc_result.txt",ios::out);

                /* close unuse write end of pipe */
                close(pipefd[1]);

                /* loop until reach loop number */
                while(loop > 0)
                {
                        /* clear counter */
                        long count=0;

                        /* loop until got stop signal from
                        *  first process */
                        while(sigbuf != STOP_SIGNAL)
                        {
                                /* increment counter */
                                count++;

                                /* do memory allocation test */
                                int i;
                                char * pch = new char[BUF_SIZE];
                                for(i=0;i<BUF_SIZE;i++)
                                {
                                        *(pch+i) = i;
                                }

                                /* free memory */
                                delete[] pch;

                                /* read signal from first process */
                                read(pipefd[0],&sigbuf,1);
                        }
```

```
                        /* display counter value after receive stop signal */
                        printf("%d:(2)%lu\n",loop,count);

                        /* update result to file */
                        fp_out << count << endl;

                        /* decrement loop number */
                        loop--;

                        /* wait for first process start signal */
                        while(sigbuf == STOP_SIGNAL) read(pipefd[0],&sigbuf,1);
                }

                /* close result output file */
                fp_out.close();

                _exit(0);
        }
        /* for first process do */
        else
        {
                /* pipe file descripter */
                int pipefd2[2];

                /* signal data buffer */
                char sigbuf2=0;

                /* create pipe   */
                if(pipe2(pipefd2,O_NONBLOCK)==-1)
                {
                        printf("create pipe error\n");
                        exit(-1);
                }
                pid_t pid2 = fork();
                if(pid2 == 0){

                        /* create result output file */
                        ofstream fp_out2;
                        fp_out2.open("arith_result.txt",ios::out);

                        close(pipefd2[1]);
                        int c=0;
                        /* loop until reach loop number */
                        while(loop > 0)
                        {
                                long count=0;    /* clear counter */

                                /* loop until got stop signal from main process */
                                while(sigbuf2 != STOP_SIGNAL)
                                {
                                        /* do calculate testing */
                                        unsigned long arithresult0 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult1 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult2 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult3 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult4 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult5 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult6 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult7 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult8 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        unsigned long arithresult9 =
                                                0xAAAAAAAA/0x55555555*0x33333333;
                                        count ++;
                                        /* read signal from first process */
                                        read(pipefd2[0],&sigbuf2,1);

                                }
```

```
                        /* display counter value after receive stop signal */
                        printf("%d:(3)%lu\n",loop,count);

                        /* update result to file */
                        fp_out2 << count << endl;

                        /* decrement loop number */
                        loop--;

                        /* wait for first process start signal */
                        while(sigbuf2 == STOP_SIGNAL) read(pipefd2[0],&sigbuf2,1);
                }
                /* close result output file */
                fp_out2.close();

                _exit(0);
        }
        else
        {
                /* pipe file descripter */
                int pipefd3[2];

                /* signal data buffer */
                char sigbuf3=0;

                /* create pipe   */
                if(pipe2(pipefd3,O_NONBLOCK)==-1)
                {
                        printf("create pipe error\n");
                        exit(-1);
                }
                pid_t pid3 = fork();

                if(pid3 == 0)
                {
                        /* create result output file */
                        ofstream fp_out3;
                        fp_out3.open("call_result.txt",ios::out);

                        close(pipefd3[1]);
                        int c=0;
                        /* loop until reach loop number */
                        while(loop > 0)
                        {
                                /* clear counter */
                                long count=0;

                                /* loop until got stop signal from
                                *  first process */
                                while(sigbuf3 != STOP_SIGNAL)
                                {
                                        /* do calculate testing */
                                        test_function(0xFFFFFFFF);
                                        count ++;

                                        /* read signal from first process */
                                        read(pipefd3[0],&sigbuf3,1);

                                }

                                /* display counter value after
                                        receive stop signal */

                                printf("%d:(4)%lu\n",loop,count);

                                /* update result to file */
                                fp_out3 << count << endl;

                                /* decrement loop number */
                                loop--;

                                /* wait for first process start signal */
                                while(sigbuf3 == STOP_SIGNAL)
                                        read(pipefd3[0],&sigbuf3,1);
                        }
```

```
                                /* close result output file */
                                fp_out3.close();

                                _exit(0);

                        }
                        else
                        {

                                /* close unuse read end of pipe */
                                close(pipefd[0]);
                                close(pipefd2[0]);
                                close(pipefd3[0]);

                                printf("performance testing start...\n");
                                printf("first process waiting for second process(%d)
                                        %d secs \n",pid,wait_time);

                                /* loop until reach loop number */
                                while(loop > 0)
                                {
                                        /* sleep parent for 1 and 2 secs */
                                        sleep(wait_time);

                                        /* set stop signal value */
                                        sigbuf = STOP_SIGNAL;
                                        sigbuf2 = STOP_SIGNAL;
                                        sigbuf3 = STOP_SIGNAL;

                                        /* send stop signal to second process */
                                        write(pipefd[1],&sigbuf,1);
                                        write(pipefd2[1],&sigbuf2,1);
                                        write(pipefd3[1],&sigbuf3,1);

                                        /* decrement loop number */
                                        loop--;

                                        /* waiting for second process */
                                        sleep(1);

                                        /* send start signal to second process */
                                        sigbuf = START_SIGNAL;
                                        sigbuf2 = START_SIGNAL;
                                        sigbuf3 = START_SIGNAL;

                                        write(pipefd[1],&sigbuf,1);
                                        write(pipefd2[1],&sigbuf2,1);
                                        write(pipefd3[1],&sigbuf3,1);
                                }
                                exit(0);
                        }
                }
        }
        return 0;
}
```

ภาคผนวก ง

ขั้นตอนการเตรียมระบบสำหรับการทำงานของ RSD

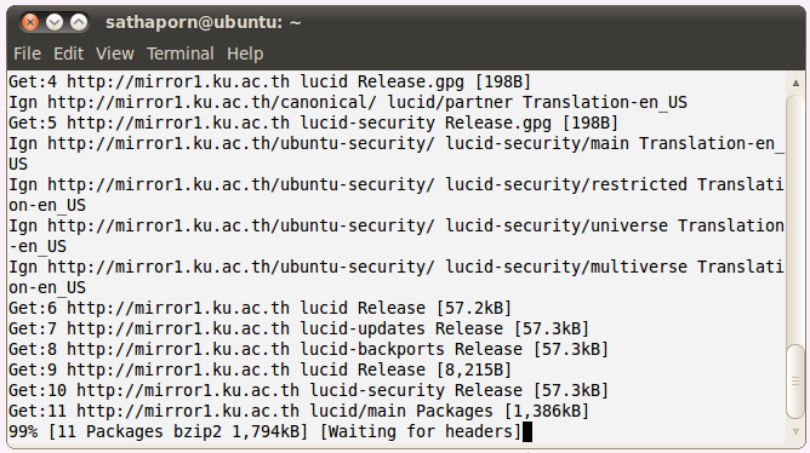
1. ระบบปฏิบัติการที่ใช้ในงานวิจัยนี้คือ Ubuntu Linux เวอร์ชั่นที่ใช้ทดสอบการทำงานของ RSD คือ Ubuntu Linux 10.04 x86 ซึ่งสามารถดาวน์โหลดได้จาก http://www.ubuntu.com/download/ubuntu/download

2. ติดตั้งระบบปฏิบัติการลินุกซ์บนเครื่องคอมพิวเตอร์ หรือ เครื่องคอมพิวเตอร์เสมือน (Virtual Machine) ซึ่งจะไม่กล่าวถึงในรายละเอียด สามารถศึกษาเพิ่มเติมได้จาก http://www.ubuntu.com

3. หลังจากติดตั้งระบบปฏิบัติการลินุกซ์เรียบร้อย ทำการปรับแต่งให้ Ubuntu Linux สามารถติดตั้งซอฟต์แวร์ผ่านอินเตอร์เน็ตได้โดย ดาวน์โหลดไฟล์ source.list จาก http://mirror1.ku.ac.th/apt-ubuntu/10.04 จากนั้นทำการ double click ที่ไฟล์ source.list ลินุกซ์ Ubuntu จะทำการเพิ่มแหล่งติดตั้งซอฟต์แวร์เข้าไปยังระบบโดยอัตโนมัติ ทำการสั่งปรับปรุงฐานข้อมูลของระบบการติดตั้งด้วยคำสั่ง

   sudo apt-get update [enter]

   จะปรากฏรายการปรับปรุงข้อมูล ดังภาพที่ ง-1 รอจนการทำงานเสร็จสมบูรณ์



ภาพที่ ง-1 การปรับปรุงแหล่งติดตั้งซอฟต์แวร์

4. ดาวน์โหลดเคอร์เนลซอร์ซโค้ดได้จาก www.kernel.org/pub/linux/kernel/v2.6/ โดยเลือกเวอร์ชั่นที่ใหม่กว่าเคอร์เนลของระบบที่กำลังทำงานอยู่ประมาณ 1-2 เวอร์ชั่น สามารถตรวจสอบเวอร์ชั่นของเคอร์เนลด้วยคำสั่ง

   uname –r

   ไฟล์ที่ดาวน์โหลดเสร็จจะถูกเก็บอยู่ใน /home/[user name]/Downloads ซึ่งชื่อของไฟล์จะอยู่ในรูป linux-[kernel version].tar.bz2 จากนั้นคัดลอกไฟล์ที่ดาวน์โหลดเรียบร้อยแล้วไปยังโฟลเดอร์ /usr/src/ ด้วยคำสั่ง

   sudo cp ~/Downloads/linux-[linux version].tar.bz2 /usr/src/

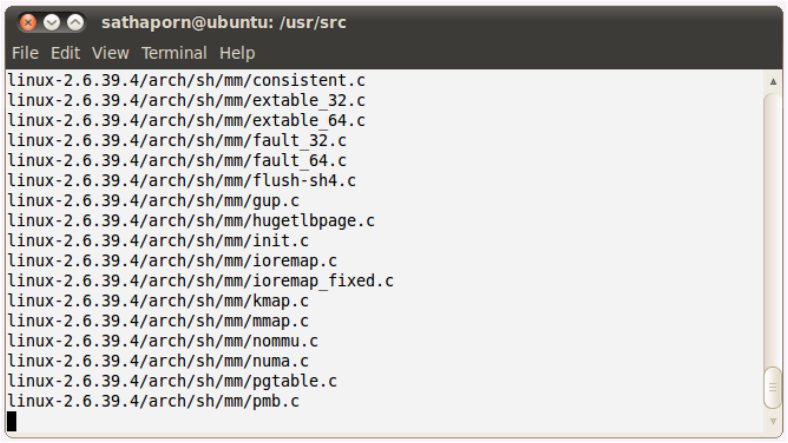
   ตัวอย่าง

   sudo cp ~/Downloads/linux-2.6.39.4.tar.bz2 /usr/src

5. ทำการแตกเคอร์เนลซอร์ซโค้ดภายใต้โฟลเดอร์ /usr/src ด้วยคำสั่งด้านล่าง ซึ่งผลของการคำสั่งดังกล่าวแสดงได้ดังภาพที่ ง-2

   cd /usr/src

   tar jxvf linux-[linux version].src.tar.bz2



ภาพที่ ง-2 การแตกซอร์สโค้ดของเคอร์เนล

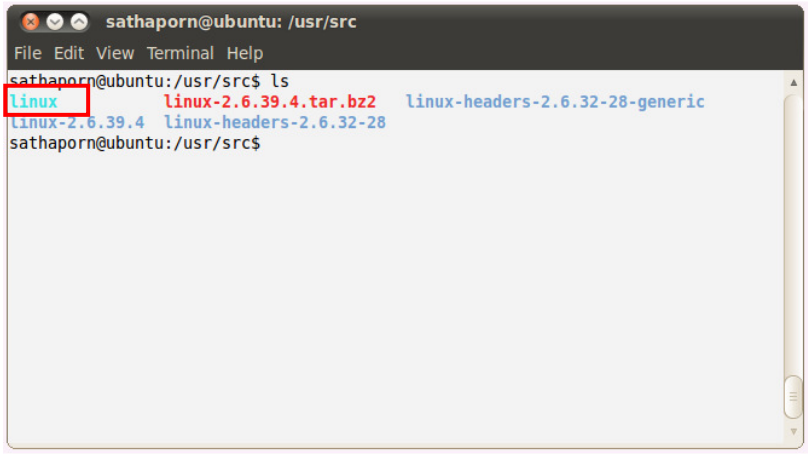
จากนั้นภายใต้ folder /usr/src จะมี folder linux-[kernel version] ปรากฏขึ้นมา จากนั้นทำการสร้าง symbolic link มายังโฟลเดอร์ที่สร้างขึ้นเพื่อให้ง่ายต่อการเรียกใช้งานด้วยคำสั่ง

sudo ln –s linux-[kernel version] linux [enter]

<u>ตัวอย่าง</u>

sudo ln –s linux-2.6.39.4 linux [enter]

จากนั้นจะปรากฏ symbolic link ชื่อ linux ขึ้นมาดังภาพที่ ง-3



ภาพที่ ง-3 การทำ Symbolic link

6. โดยปกติระบบปฏิบัติการลินุกซ์จะมีไฟล์สำหรับปรับแต่งเคอร์เนลเพื่อใช้สำหรับคอมไพล์เคอร์เนล ซึ่งจะถูกเก็บอยู่ที่โฟลเดอร์ /boot ให้ทำการคัดลอกไฟล์ปรับแต่งเคอร์เนลไปยังโฟลเดอร์ /usr/src/linux ด้วยคำสั่งต่อไปนี้

   sudo cp /boot/config-[kernel version]-generic /usr/src/linux/.config [enter]

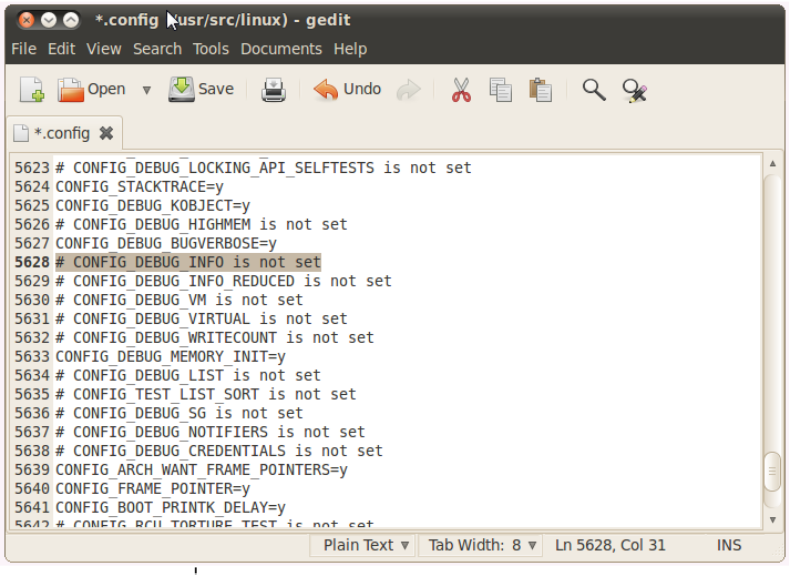
   ตัวอย่าง

   sudo cp /boot/config-2.6.32-generic /usr/src/linux/.config [enter]

7. ทำการปรับแต่งไฟล์ config สำหรับเคอร์เนลเพื่อให้รองรับข้อมูล debug-info โดยเปิดไฟล์ด้วยโปรแกรม gedit ดังนี้

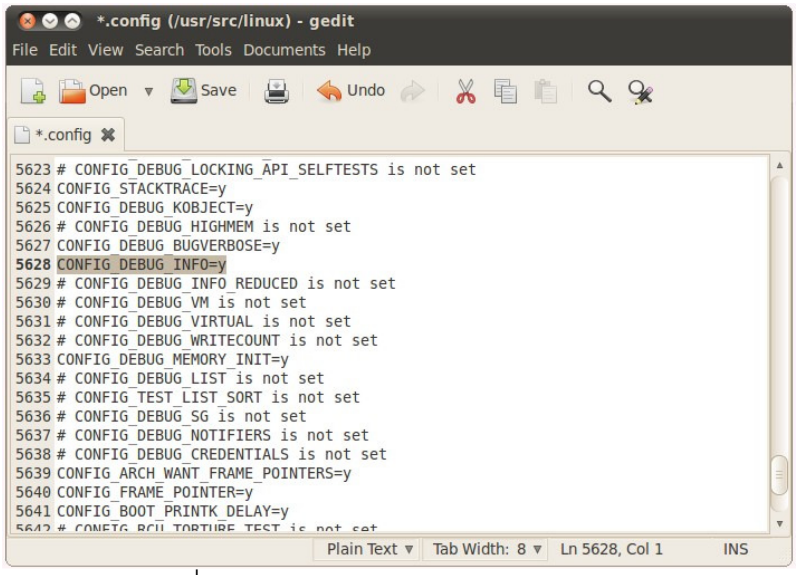
   sudo gedit /usr/src/linux/.config [enter]

   จากนั้นค้นหาคำว่า CONFIG_DEBUG_INFO ดังภาพที่ ง-4



ภาพที่ ง-4 ตำแหน่ง CONFIG_DEBUG_INFO

แก้ไขโดยลบเครื่องหมาย “#” ออกและเพิ่มเครื่องหมาย “=y” ผลลัพท์ดังภาพที่ จ-7



ภาพที่ ง-5 การแก้ไข CONFIG_DEBUG_INFO

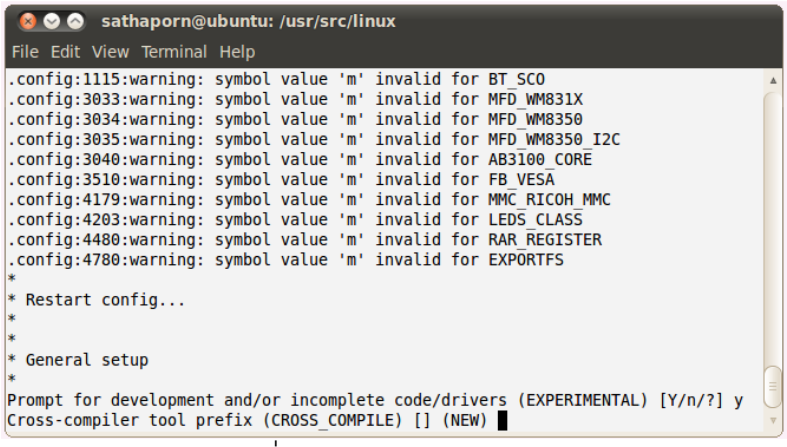
ทำการบันทึกไฟล์ และปิดโปรแกรม gedit

8. ทำการ config ค่าต่างๆที่ทำการปรับแต่งแล้วให้กับเคอร์เนลซอร์ซโค้ดด้วยคำสั่งต่อไปนี้

cd /usr/src/linux [enter]

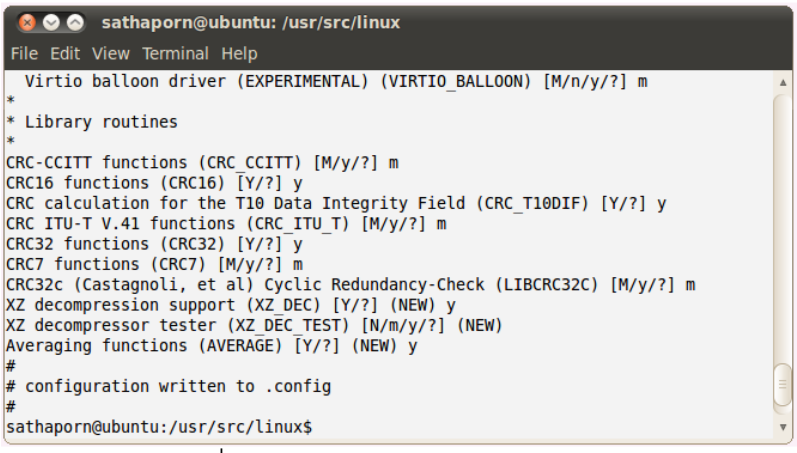
sudo make oldconfig [enter]

จะปรากฏหน้าจอแสดงคำถามดังภาพที่ ง-6 ให้ตอบโดยใช้ค่าปริยายทั้งหมดซึ่งสามารถกระทำได้โดยกด Enter key ค้างไว้จนหมดคำถาม



ภาพที่ ง-6 การปรับแต่งเคอร์เนล

จนกระทั่งการปรับแต่งเสร็จสมบูรณ์ดังภาพที่ ง-7

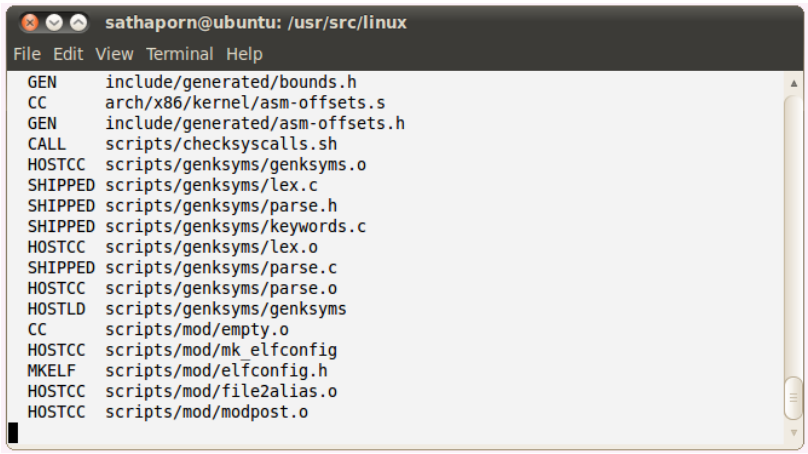


ภาพที่ ง-7 การบันทึกไฟล์ปรับแต่งเคอร์เนล

9. จากนั้นเริ่มทำการคอมไพล์เคอร์เนลด้วยคำสั่งต่อไปนี้

sudo make [enter]

จะปรากฏรายละเอียดการคอมไพล์ดังภาพที่ ง-8 รอจนการคอมไพล์เคอร์เนลเสร็จสมบูรณ์ ซึ่งระยะเวลาในการคอมไพล์ขึ้นอยู่กับประสิทธิภาพของฮาร์ดแวร์ที่ระบบปฏิบัติการทำงาน



ภาพที่ ง-8 การคอมไพล์เคอร์เนล

10. ทำการติดตั้งโมดูลต่างๆ ที่คอมไพล์เรียบร้อยแล้วไปยังระบบไฟล์ที่กำลังทำงานอยู่โดย

    sudo make modules_install [enter]

    หลังจากทำคำสั่งด้านบนเสร็จสิ้น จะปรากฏโฟลเดอร์ที่มีชื่อเดียวกับเวอร์ชั่นของเคอร์เนลภายใต้โฟลเดอร์ /lib/modules

11. จากนั้นทำการสร้างไฟล์หน่วยความจำเสมือน (Ram disk) เพื่อระบุตำแหน่งไฟล์ต่างๆ ที่ใช้ในการเริ่มระบบ สามารถกระทำโดยใช้คำสั่ง

    cd /lib/modules [enter]

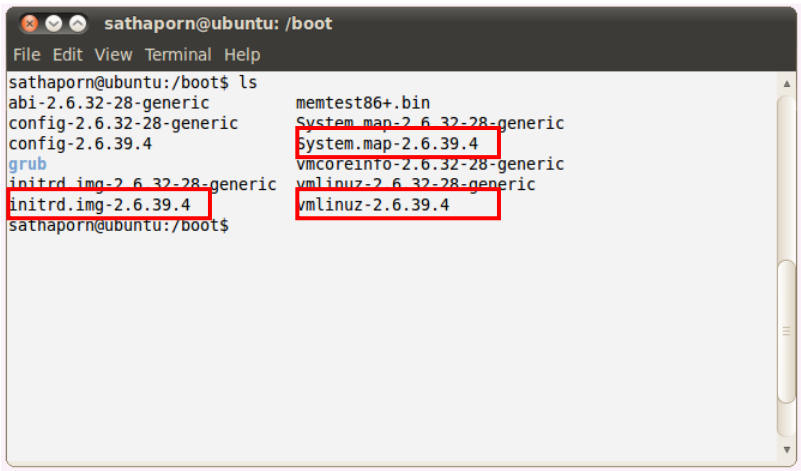
    sudo mkinitramfs –o /boot/initrd.img-[kernel version] [kernel version] [enter]

12. ทำการติดตั้งข้อมูลที่จำเป็นสำหรับการเริ่มระบบใหม่โดย

    cd /usr/src/linux [enter]

    sudo make install [enter]

    ในขั้นตอนนี้จะมีข้อมูลที่ใช้สำหรับการเริ่มระบบเพิ่มเติมขึ้นมาภายใต้โฟลเดอร์ /boot ดังแสดงใน ภาพที่ ง-9

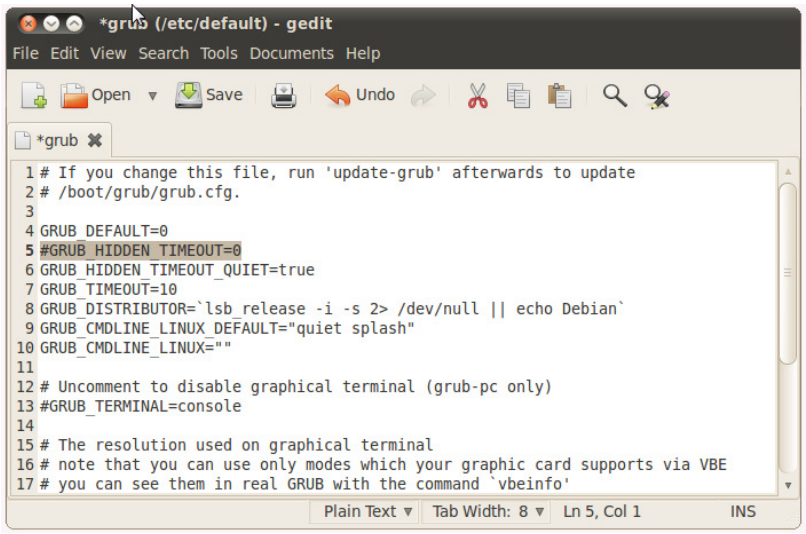


ภาพที่ ง-9 ไฟล์ที่เพิ่มขึ้นหลังจากติดตั้งเคอร์เนล

13. ปรับแต่ง grub menu เพื่อให้สามารถเรียกใช้งานเคอร์เนลใหม่ได้โดยทำคำสั่ง

sudo gedit /etc/default/grub [enter]

ทำการคอมเมนท์บรรทัดที่มีข้อความ GRUB_HIDDEN_TIMEOUT ดังภาพที่ ง-10



ภาพที่ ง-10 การปรับแต่งการเริ่มระบบใหม่

บันทึกไฟล์ grub และปรับปรุงเมนูสำหรับเริ่มระบบด้วยคำสั่ง

sudo update-grub [enter]

สั่งให้ระบบเริ่มทำงานใหม่ด้วยคำสั่ง

sudo reboot [enter]

14. รอจนเมนู grub แสดงรายการเคอร์เนลสำหรับเริ่มการทำงาน เลือกเมนูของเคอร์เนลเวอร์ชั่นที่ทำการคอมไพล์ไว้ (ตัวอย่าง คอมไพล์ด้วยเวอร์ชั่น linux- 2.6.39.4) รอจนลินุกซ์เริ่มการทำงานอย่างสมบูรณ์

15. ทำการติดตั้ง Systemtap ด้วยคำสั่งต่อไปนี้

sudo apt-get install systemtap [enter]

รอจนการติดตั้ง Systemtapเสร็จสมบูรณ์ ทดสอบการทำงานของ Systemtap ด้วยคำสั่ง

sudo stap –e ‘probe begin { printf(“Test\n”) exit() }’ [enter]

จะต้องปรากฏข้อความ “Test” แสดงออกหน้าจอแสดงผล หมายถึงการเตรียมระบบเสร็จสมบูรณ์

ภาคผนวก จ

ฮาร์ดแวร์ที่ใช้ทดสอบการทำงานของ RSD

- เครื่องคอมพิวเตอร์จำลอง (Virtual machine) ที่ติดตั้ง RSD

  VM Product : VMWare Workstation, Version 7.1.4 build-385536

  VM OS: Ubuntu Linux 10.04 x86

  VM RAM: 1 GB

  VM CPU: Default generic CPU with 2 cores

  VM Hard disk: 40 GB

  VM Network: NAT

  VM Resolution: 1366 x 768

- เครื่องคอมพิวเตอร์หลักที่ติดตั้งคอมพิวเตอร์จำลอง

  OS: Windows 7 Enterprise, 64 bit,
  build 6.1.7601, Service Pack 1

  RAM: 4 GB

  CPU: Intel Core i5

  Resolution: 1366 x 768

ประวัติผู้เขียนวิทยานิพนธ์

นายสถาพร สงวนวงษ์ เกิดเมื่อวันที่ 3 กรกฎาคม 2515 ณ กรุงเทพมหานครฯ สำเร็จการศึกษาระดับ ประกาศนียบัตรวิชาชีพชั้นสูง จากสถาบันเทคโนโลยีราชมงคลวิทยาเขตเทคนิคกรุงเทพฯ ในปีการศึกษา 2535 หลังจากนั้นได้มีโอกาสทำงานในตำแหน่งผู้ดูแลระบบควบคุมการผลิต บริษัทในกลุ่มปิโตรเคมี ของเครือซีเมนต์ไทย จากนั้นได้สำเร็จการศึกษาในหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต คณะวิศวกรรมศาสตร์คอมพิวเตอร์ จากสถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ในปีการศึกษา 2548 และได้เข้าศึกษาต่อในหลักสูตรวิทยาศาสตรมหาบัณฑิต ภาคนอกเวลาราชการ จุฬาลงกรณ์มหาวิทยาลัยในปีการศึกษา 2552 ปัจจุบัน ดำรงตำแหน่งนักพัฒนาซอฟต์แวร์อาวุโสในบริษัทซอฟต์แวร์เอกชนแห่งหนึ่ง