

KNUST

THE DESIGN AND IMPLEMENTATION OF CONVOLUTED OS KERNEL
ARCHITECTURE FOR SECURED NETWORK INFRASTRUCTURE

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE IN PARTIAL
FULFILMENT OF THE REQUIREMENT FOR THE DEGREE IN DOCTOR OF
PHILOSOPHY (PhD) IN COMPUTER SCIENCE

DECEMBER 2017

EDWARD DANSO ANSONG

(M. Eng. Computer Science & Engineering, BSc. Computer Science)

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma at Kwame Nkrumah University of Science and Technology, Kumasi or any other educational institution, except where due acknowledgement is made in the thesis.

Edward Danso Ansong

PG1547113

.....
(Student name & Index number) Signature Date

Certified by:

Dr. J. B. Hayfron-Acquah

(Supervisor) Signature Date

Certified by:

Dr. Michael Asante

(Supervisor) Signature Date

Certified by:

Dr. Michael Asante

(Head of Department) Signature Date

ACKNOWLEDGEMENTS

To Jehovah God Almighty for my life and all the manifold blessings, thank you my Father. I would like to express my heartfelt appreciation to my supervisor Dr. J.B. Hayfron-Acquah for his

patience, guidance and most importantly, all the useful suggestions he gave me throughout the period of this work. His support has been invaluable to the completion of this work. I wish to also thank Dr. M. Asante for all the technical inputs and assistance and suggestions he offered me.

I would also like to thank Apostle Augustus Arthur for supporting my education after the demise of my late parents over a decade ago and serving as my Father and the spiritual pillar.

I would also wish to mention Engr. Dominic D. Damoah Snr., Dominic Otieko-Asare Damoah Jnr. and Daniel Sarpong-Duah for the immense contribution by setting up the lab used for the simulation and testing of the prototype.

I would also like to appreciate the efforts of Prof. John Criswell of Rochester University who provided the training from the Rochester University, USA.

I wish to acknowledge the immense support of Mr. Benjamin Odoi-Lartey for the immense support provided me. I also appreciate my late parents, who unfortunately after investing so much in my education failed to live to see this day, Mr. Samuel Danso and Christiana Asante. I cannot forget the support of all my siblings who have always been there during the hard-times.

Lastly, I appreciate everyone who in diverse ways has helped in the completion of this work and to my life and education in general. God bless you all.

DEDICATION

I dedicate this thesis to my entire family especially my wife Comfort and wonderful children Emerald, Christabel, and Edward.

ABSTRACT

Research into security enhanced kernel architecture has been on going by computer corporations, research institutions and Kernel development engineers for several decades now. Even though the paradigm-shift from performance enhanced kernel development to application level security operating systems improved the safety of operating systems use, it was apparent that more kernel retrofitting need to be implemented at the kernel level since it is a highly privileged section of the architecture, and therefore a compromise in the kernel could affect the security of the entire systems including the performance of security applications which runs on the kernel. This research introduces the Convolute Kernel Architecture (CKA) which is a security enhanced Linux server based monolithic architecture that re-modifies the original monolithic kernel architecture with additional layer of virtual module within the kernel to improve the security and availability of the Kernel. The system make use of Operating System level virtualization with an integrated security module, which otherwise would have been implemented at the application layer, with a novel authentication module called the Stealth Obfuscation Zero Knowledge Proof algorithm. This research describes how the CKA abstracts the hardware and software layers of conventional servers to implement the operating system all in one security appliance that at the same time, providing High Availability. Finally, this work describes how the CKA protects the core kernel from attacks when upper kernel becomes compromised through vulnerabilities of applications that executes on them or the kernel utilities itself.

TABLE OF CONTENTS

DECLARATION.....	i
ACKNOWLEDGEMENTS.....	i
DEDICATION.....	ii
ABSTRACT	iii
LIST OF ABBREVIATIONS.....	vi
LIST OF FIGURES	ix
LIST OF TABLES.....	xi


LISTINGS	xiii
CHAPTER 1	1
INTRODUCTION	1
1.1 Convolute Kernel Architecture.....	2
1.2 Problem Statement.....	3
1.3 The Goal of the research.....	3
1.4. The Objectives of the Research	3
1.5 Background of the Architecture.....	5
1.6 Motivation	6
1.7 Structure of the Thesis	7
CHAPTER 2	8
LITERATURE REVIEW	8
2.1 Overview of LSM Framework.....	9
2.2 Evolution of OS Security Framework	10
2.3 Generalized Framework for Access Control	11
2.4 Flux Advanced Security Kernel.....	13
2.5 Domain and Type Enforcement.....	15
2.6 Distributed Security Infrastructure	16
2.7 Security Mechanisms.....	17
2.8. OTHER CONCEPTS.....	20
CHAPTER 3	35
RESEARCH METHODOLOGY	35
3.1 Introduction	35
3.2 Research Design Methodologies	35
3.2.1 Structured Analysis and Design Techniques	36
3.2.2 Data Structured Systems Development	38
3.3 OS Host Setup	44
3.4 Secure Kernel Build Step.....	44
CHAPTER 4	51
CONVOLUTED KERNEL ARCHITECTURE.....	51

4.1 Overview of Kernel Architecture	51
4.2 Linux Kernel Security Framework	55
4.3 Drawback of Linux Security Module	56
4.4 The Convolved Kernel Architecture.....	56
4.5 Security Enhanced Framework.....	62
4.6 Overall Privilege Separation of the Design	64
4.7. The implementation of the Convolved Kernel Architecture – Trend-OS.....	65
4.8. Adoption of Chroot Hardening Technique for compatibility	65
4.9. Insertion of Socket restrictions	66
4.10. Summary.....	82
CHAPTER 5	83
STEALTH-OBFUSCATION ZKP AUTHENTICATION PROTOCOL	83
5.1 Introduction	83
5.2 Existing Models	84
5.3 Zero Knowledge Proofs.....	87
5.4 Schnorr's Identification Protocol	87
5.5 Schnorr's Signature Scheme	88
5.6 Signatures Based Proof Knowledge – (SPK)	89
5.7 Stealth Obfuscation ZKP Scheme.....	89
5.8 Implementation of Stealth Obfuscation ZKP.....	90
5.9 Strength of Stealth Obfuscation ZKP	106
5.10 Conclusion	106
CHAPTER 6	108
RESULTS ANALYSIS	108
6.0 Introduction	108
6.1 Evaluation.....	108
6.1.1 Scalability Testing	114
6.1.2 CPU Utilization	116
6.1.3 Memory utilization	117
6.1.4 Operating System Limitations	118


6.1.5 Virtualization Techniques	119
6.1.6 Performance Throughout	120
6.2 Further Performance Analysis	121
6.3 Summary of the Experiments	122
CHAPTER 7	122
DISCUSSION AND CONCLUSION	122
7.0 DISCUSSION.....	122
7.1 CONTRIBUTION TO KNOWLEDGE.....	122
7.2 CONCLUSION.....	123
7.3 Future work.....	124
References	126

LIST OF ABBREVIATIONS

ACI	- Access Control Information
ACR	- Access Control Rules
ACLs	- Access Control Lists
ADF	- Access Decision Facility
AEF	- Access Enforcement Facility
AJAX	- Asynchronous JavaScript and XML
API	- Application Programming Interface
AVC	- Access Vector Cache
CGs	- Control Groups
CKA	- Convolved Kernel Architecture
CPU	- Central Processing Unit
DAC	- Discretionary Access Control
DAC	- Discretionary Access Control
DARPA	- Defense Advanced Research Projects Agency
DDT	- Domain Definition Table



DoS	- Denial of Service
DISEC	- Disarmament and International Security Committee
DIT	- Domain Interaction Table
DSI	- Distributed Security Infrastructure
DTE	- Domain and Type Enforcement
DTOS	- Distributed Trusted Operating System
EVM	- Extended Verification Module
FLASK	- Flux Advanced Security Kernel
GFAC	- Generalized Framework for Access Control
GKA	- Generic Kernel Architecture
HTTPS	– HyperText Transfer Protocols
HVM	- Hypervisor Virtual Machine
IAC	- Integrity Access Class
IM	- Instant Messaging
IPC	- Instruction per Cycle
IPE	- Inheritable, Permitted (P), and Effective (E)
KVM	- Kernel Virtual Machine
LIDS	- Linux Intrusion Detection System
LKMs	- Loadable Kernel Modules
LSM	- Linux Security Module
LXC	- Linux Containers
MAC	- Mandatory Access Control
MLS	- Multilevel Security
NSA	- National Security Agency
NTT	– Nippon Telegraph and Telephone
OS	- Operating System
PAM	- Portable Authentication System
PAM	- Pluggable Authentication Module
POSIX	- Portable Operating System Interface



RBAC	- Role-Based Access Control
RSBAC	- Rule Set Based Access Control
SAC	- Security Access Class
SCC	- Secure Computing Corporation
SELinux	- Security Enhanced Linux
SIDs	- Security Identifiers
SMACK	- Simplified Mandatory Access Control Kernel
SPK	- Signature based Proofs Knowledge
SSID	- Service Set Identifier
STAT I	- Space – Time Authentication Technique
STL1	- Secure Trust Level 1
SVA	- Secure Virtual Architecture
TCPA	- Trusted Computer Protection Alliance
TCSEC	- Trusted Computer System Evaluation Criteria
TE	- Type Enforcement
TLSM	- TrendOS Linux Security Module
TOMOYO	- Task Oriented Management Obviates Your Onus
TPM	- Trusted Platform Module
TREND-OS	- Trust Resilient Enhanced Network Defense Operating System
TSID	- Target Security Identifiers
UIDs	- User Identifiers
UTM	- Unified Threat Management
VMS	- Virtual Memory System
XML	- Extensible Markup Language

KNUST

LIST OF FIGURES

Figure 2.1 Linux security module architecture.....	10
Figure 2.2. Overview of GFAC Architecture.....	12
Figure 2.3. Overview of FLASK architecture.....	13
Figure 2.4 Domain and Type Enforcement.....	15
Figure 2.5 Role Based Access Control.....	19
Figure 2.6 Overview of SELinux.....	28
Figure 3.1 Design Methodology Flow Chart	37
Figure 3.2 Data Structures.....	38
Figure 3.3 Kernel File System Data Structure.....	39
Figure 3.4 The Process Table.....	40
Figure 3.5 The File Table.....	41
Figure 3.6 The v-node and i-node.....	42
Figure 3.7 Standard File Descriptors.....	43
Figure 3.8 Kernel Configuration.....	48
Figure 3.9. Configuration Environment.....	50

Figure 4.1. – Breakdown of an Operating System into four major Subsystem.....	53
Figure 4.2. Monolithic Design.....	54
Figure 4.3. Convoluted Kernel Architectural Design.....	58
Figure 4.4. Heartbeat architecture of the of the Convoluted Architecture.....	59
Figure 4.5 Simplified High Availability (HA).....	60
Figure 4.6 Expanded Unified Threat Management.....	61
Figure 4.7 The LXC design in the Architecture.....	63
Figure 4.8 Privilege Separation of the architecture.....	64
Figure 4.9 Kernel deployment into the architecture.....	68
Figure 4.10 Loading of the various security and driver modules into the kernel.....	70
Figure 4.11 Modifying the generic kernel to suit the convoluted kernel Architecture.....	71
Figure 4.12 Integrating redundant security modules to protect the kernel against itself.....	72
Figure 4.13 Building the various files and integrating into inbuilt protocols for compatibility....	73
Figure 4.14 Good user friendly installation experience.....	74
Figure 4.15 Step by step system installation guide.....	75
Figure 4.16 Effective Disk space utilization	76
Figure 4.17 Minimum load system resource utilization.....	77
Figure 4.18 Support for multiple server instances running on same system - Linux containers...	78
Figure 4.19 Live backup system for High Availability.....	79
Figure 4.20 Minimum CPU and Memory utilization.....	80
Figure 5.1 Stealth ZKP User Registration (Client Side).	89
Figure 5.2 Stealth ZKP User Authentication (Client Side).	91

Figure 5.3 Stealth User Authentication (Server Side).	98
Figure 6.1 Security Testing of the User Interface.....	107
Figure 6.2 Tamper Popup.....	108
Figure 6.3 Tamper with Request.....	108
Figure 6.4 Tamper Popup with encrypted Password.....	109
Figure 6.5 Code Snippet.....	110
Figure 6.6 CPU System Benchmark.....	111
Figure 6.7 DD Benchmark.....	112
Figure 6.8 DD CPU.....	112
Figure 6.9 IPERF CPU 2	113
Figure 6.10 Scalability Testing between CKA and GKA.....	114
Figure 6.11 CPU Utilization performance of CKA and GKA.....	115
Figure 6.12. Memory Utilization performance between CKA and GKA.....	116
Figure 6.13 Operating System Limitation between CKA and GKA.....	117
Figure 6.14 Forms of Virtualization Techniques available against the method adopted.....	118
Figure 6.15 . Performance Throughput of CKA against GKA.....	119

LIST OF TABLES

Table 2.1: Policy result combination in RSBAC's ADF.....	26
Table 2.2: Available RSBAC ACI modules.....	27
Table 2.3: Predefine SMACK labels.....	30
Table 3.1: Dedicated host properties.....	44

KNUST



LISTINGS

Listing 2.1	Example of an AppArmor profile for /bin/ping.....	23
Listing 2.2	Example of a GRsecurity ACL.....	25
Listing 2.3	Example of a LIDS ruleset.....	26
Listing 2.4	Snippet of a SELinux Policy.....	29
Listing 2.5	Example of a TOMOYO policy.....	33
Listing 4.1	Grsecurity modules	66
Listing 4.2	Kernel integration with Grsecurity.....	66
Listing 4.3	General Configuration.....	69
Listing 5.1	Client Side Login Sample Code.....	93
Listing 5.2	Random Module Sample Code.....	95
Listing 5.3	Signup Server Side Sample Code.....	96
Listing 5.4	Login Server Side Sample Code.....	97
Listing 5.5	Zero Knowledge Proof Sample Code.....	100
Listing 5.6	SHA256 Hashing Code Sample.....	102

CHAPTER 1

INTRODUCTION

This thesis addresses a critical research question of how *security* of current Server Operating Systems using *Virtual Private Servers (VPS)* be ameliorated, and yet, address the challenges associated with *High Availability* via improved throughput in monolithic Operating System. Since the last decade, Universities, Research Institutions, Computer Corporations and Operating System Development Engineers have been involved in the development of multiple isolated user-space instance operating system architecture. This transformation has been met by commensurate improvement of super hardware performance enhancement and device miniaturization, allowing operating system kernels originally designed for high specification servers to be executed on a single server unit.

Server operating system security has undergone a tremendous evolution due to increased complexity of attacks. Network communications on the other hand (since the early 90's) have also witnessed a remarkable revolution from a hitherto, 'mundane' store-and-forward applications like electronic mails, to present real-time and 'resource-hungry' collaboration tools such as instant messaging (IM), media streaming among others, with each presenting its own channels for potential attack. The vulnerability to Kernel exploits and rootkits can be partially ascribed to lack of focus on kernel level system architectural security in modern operating systems (Loscocco et al, 1998). The traditional security mechanism in UNIX, DAC, is limited in scope since it only covers filesystem access. To guarantee the enforcement of a system-wide policy, MAC was adopted. MAC does not necessarily extend the security domain of DAC, but allows system administrators to impose rules which cannot be circumvented by users. However, MAC is not solution to everything: It provides protection for the Kernel before an attack but not after the incident has eventually occurred and it is not an extension to the Kernel architecture

The complexities, which associates current server vulnerabilities, have necessitated the need for an auxiliary system, referred to as Unified Threat Management (UTM), to augment the security of server operating systems. This however comes with its overhead cost against performance of the server operating systems such as security, efficiency, responsiveness among others. The other side

is the budgetary cost of the component which allows only well established businesses and institutions to afford.

1.1 Convolutd Kernel Architecture

The thesis presents an implementation of a modified novel algorithm which implements an advanced authentication system for remote server verification that utilizes Signature Proof Knowledge model based on Zero Knowledge Proofs. The algorithm is integrated into the frontend-interface of the cluster servers via the Portable Authentication Module (PAM) in Linux. As a result of this model, another level of security is added to address the challenges associated with authentication vulnerabilities that could lead to terrible security breach.

Beyond the above situations, one of the major challenges faced by most cluster servers, which this investigation also touches on, is that of high availability. Until recently, the only technique that was via host-per-host redundancy. Even though this approach is considered as the best and easy way of achieving high availability, it is also met with high human and hardware costs. This therefore provides the opportunity for only big corporations to engineer and afford such an infrastructure. However, in this thesis, we shall present how operating system level virtualization can be used to construct on a pre-compiled kernel thereby reducing cost.

The other security protection issue faced with most of the kernel engineers has to do with how multiple kernel security modules could be combined to enhance the Discretionary Access Control policies of the kernel. This became a major challenge because most of the experiments conducted in the lab could not allow two or more different modules e.g. AppArmor and SELinux to coexist on the same kernel platform. It is however one of the cardinal highlights of proposed framework since security assurance is the main objective for its development. GRsecurity module is therefore embedded into the framework to serve that purpose for such challenges aforementioned. Details of which will be captured in the subsequent chapter.

In view of all these challenges, today a new operating system development model - code named – the “Integrated Approach” which is an all-inclusive model in ensuring that, modules required for operating system ‘hardening’ is presented and their associated developments are encapsulated as a single unit rather than separate independent application layer compartments. This framework is

ostensibly aimed at developing a novel server operating system architecture with strong resilience against attacks and that could address the challenges of current operating system level vulnerabilities.

1.2 Problem Statement

Due to the strict-intransigent and inflexible architecture of the Monolithic operating system, adopting a dynamic modular integration has become almost impossible without recompiling the entire kernel (Lindskog, 2000). Moreover, Kernels written in C and C++ and therefore suffer the same security vulnerabilities that applications suffer including buffer overflow and integer overflow attacks. (Piromsopa, 2011). Meanwhile, Legacy authentication problems for UNIX Kernels (old protocols and compatibility) has contributed to Authentication bypass and vulnerabilities in the Linux PAM. With these stated kernel vulnerabilities, a lot of research interest have been generated into the development of a more secured and efficient commodity operating system kernels by various universities and computer science research labs for over two decades now using various kernel development techniques (Criswell et al, 2014).

1.3 The Goal of the research

The goal of this research is the design and implementation of the “Convolutd-OS Kernel Architecture”. This architecture is intended to make the development of the operating system kernel complex as a result of the multifaceted nature of the core utilities integrated into the kernel, yet, easy to operate with high availability and enhanced security to frustrate the efforts of the attacker. Due to the emphasis on security in its architecture, a novel security authentication module is also designed as part of achieving the overall objective or goal of the system in order to enhance the security of this architecture.

1.4. The Objectives of the Research

The objectives of this research seek to provide the following:

1. To design a novel OS Kernel Architecture to improve and protect the security of the core kernel against internal threats and attacks.

2. To analyze the extent to which the kernel integrated UTM, could improve performance of the overall operating system.
3. To identify the extent to which the performance of monolithic kernels could be improved via the use of High Availability (HA) techniques and OS level virtualization.
4. To develop an authentication module in the OS Kernel to improve the legacy authentication algorithm from UNIX in Linux, PAM.
5. To Design a prototype Operating System to integrate the Kernel Architecture to simulate the model.

The subject of enhancing the security in monolithic operating system is the focal point in this thesis, and key concern to most computer corporations and operating system development engineers around the globe for several decades now. This however, cannot be resolved in isolation as it inter-connects to features such as high availability, increased throughput and resilience. The work therefore described in this thesis focuses on the design and implementation of the convoluted operating system architecture with its TrendOS prototype as novel server operating system in monolithic kernels to improve security, availability and reliability of the monolithic kernel architecture.

The continuous reported incidents of computer security breaches globally, and their associated operating system vulnerabilities has inspired fresh investigations of the current operating system kernel architecture. This has therefore renewed the interest of operating system engineers, research institutions and computer corporations into the domain of emerging security in operating systems design.

Finally, a unique authentication cryptographic algorithm is introduced in which it falls under the taxonomy of zero-knowledge authentication, to improve the server level authentication via remote service connection. This design is code-named ‘Stealth-Obfuscation-0K Authentication’ and its goal is to hide the existence of authentication and to improve on the existing zero knowledge authentication algorithms. It is an all in one security enhanced authentication model using a public

key cryptosystem that involves the activity of the client side with an exchange of a token (random key value) to make the channel of the cryptanalyst more secured.

1.5 Background of the Architecture

Growing from 2008, after the successful design of a new algorithm for firewall implementation using a FreeBSD prototype operating system, the desire for further study in the development of a more secured operating system architecture, to mitigate the extent to which operating system vulnerabilities are exposed to malicious programs has been the key motivation.

Even though several attempts have been made to provide safe and trusted commodity operating systems kernels for user protection, it is still susceptible to various forms of attacks because of the fact that, these kernels are written in C and C++ which therefore carry the same baggage of vulnerabilities such as buffer overflows, code injection attacks, string manipulation, memory corruption vulnerabilities among others (Piromsopa, 2011)

Research into a more secured commodity operating system kernels have currently generated an active field of research in various universities and research labs due to the increase in kernel exploits (Wang et al, 2009), (Criswell et al, 2014).

With the setting up of a private virtual laboratory to serve as a test-bed for this new operating system architectural prototype, and support from research scientists and engineers in commodity operating systems in institutions such as Rochester University-US, Kwame Nkrumah University of Science and Technology and Fredan Computers Inc. (a Research and Development Lab) - have made significant strides in the success of this novel development.

The past five (5) years have however witnessed a consistent and coherent computer laboratory experiments in operating system vulnerabilities and penetration testing across most operating system platforms. This thesis analyses the performance of monolithic kernel architecture when subjected to various kernel development approaches such as retrofitting techniques, integrated approach after a complete transfiguration and the effectiveness of a kernel retrofitted security integrated modules instead of the user space model of the kernel architecture which has been the

practiced in kernel development. This prior methodological approach has contributed to the several reported security breaches in operating systems. The current trend of increased application level security has been necessitated because, security was not a major issue when operating systems were first introduced. Moreover, the subsequent deployment of policies such as “Mandatory Access Control” (MAC) and the “Discretionary Access Control” (DAC) in operating systems at the application layer was considered as pragmatic enough to provide security.

1.6 Motivation

Four factors contributed to the exploration in this research. These are: the *increase* in the number of reported incidents of operating system vulnerabilities (National Vulnerabilities Database, 2016) the weakness of the strict-intransigent-structured *monolithic* operating system architecture, the challenges that characterizes the various kernel development approaches and the opportunity to exploit the performance of operating system extensibility using the integrated approach.

First of all, vulnerabilities related to operating systems could be largely associated with the fact that the advent of early operating systems were developed in a friendly, collaborative environment without much considerations to predefined persistence. Several decades later, as the environments became less friendly in addition to increased functional complexity, coupled with their associated user sophistication and technical competence, vulnerabilities began to expose the cracks in the ‘robustness’ and security of the operating systems in use (Bishop, 2009).

Secondly, the traditional monolithic architecture of Linux operating system, inherited from UNIX, has always managed large part of the systems functions in a strict-intransigent-structure at the kernel mode while only limited part is managed at the user mode. Legacy authentication problems for UNIX Kernels (old protocols and compatibility) have contributed to Authentication bypass and vulnerabilities in the Linux PAM (Lindskog, 2000). This therefore requires that any core system configuration necessitates a re-compilation of the entire kernel in order to update the kernel. Therefore, such architecture becomes inflexible and intractable to manage, as the slenderest modification obligates a re-compilation of the kernel.

The third motivation is centered on the numerous approaches adopted by operating system engineers in optimizing and enhancing the security of latest kernels in ongoing development. Even though several research has been carried out and still ongoing, there is the need for additional reengineering be made to exploit new opportunities to make the operating systems more resilient against attacks.

Finally, to present an enhanced implementation of a zero knowledge authentication algorithm called Stealth-OK that improves the current state of zero knowledge by correcting an anomaly in the original zero knowledge algorithm with the stealth-OK algorithm. Furthermore, a vulnerability identified in the original algorithm is also revised to further enhance the security of the algorithm against man-in-the-middle attack and cryptanalyst.

1.7 Structure of the Thesis

Chapter one provides the overview of the Linux architecture and their impact is also discussed in this chapter. The first section introduces the architecture and its implementation. The subsequent section also looks at the design goal of the architecture and an introduction to the key part of the algorithm which provides security and protection to the PAM which shields the entire authentication systems of the architecture from various forms of attacks. The third section looks at the background study of the architectures including the various efforts made by commodity Operating System developers to develop systems which are resilient to all forms of attacks. Finally, the concluding section of the chapter looks at the motivation for the development of a new architecture for the kernel.

In the second chapter, various literature on operating system kernel architecture were reviewed. A description of the Linux Security Module (LSM) framework is carried out and the function of each layer explained. A thorough discussion on the several security modules are also explained. Their strengths and vulnerabilities captured in literature are also expatiated.

In the third chapter, various research design methods are introduced. Three of the methodologies used in the research design are discussed; Structured Analysis & Design Technique, Data Structured Systems Development and Integrated Design Approach. The rationale for the selection of the three research design methodologies and the roles they play explained.

In the fourth chapter, the various kernel architecture and framework that enhances security of Linux kernel is discussed. Several of these frameworks were discussed as well as the various modules that constitute their implementation. The framework of the Convolutional Kernel Architecture is introduced as well as the goal and objectives of the entire design. Detailed analysis of the key parts of the architecture was also discussed with their functions.

In the fifth chapter, the thesis looked at the Stealth-Obfuscation Zero Knowledge Authentication Protocol and its performance in the architecture. It brings out the extent to which this protocol could address the several security vulnerabilities in the authentication process of PAM.

In the sixth chapter, the various reports generated from the experiments and testing of the architecture is analyzed. Several reports of the comparative study between the Convolutional Kernel Architecture with the Generic Kernel Architecture are analyzed and the performance trend simulated on a graph.

Finally, the seventh chapter summarizes it all with the conclusion and future work of the proposed architecture.

CHAPTER 2

LITERATURE REVIEW

Advanced data structures and kernel exploits are implemented on commodity OS Kernel on a daily basis. However, over the years, most security organizations, government agencies, universities and computer corporations around the world have been involved in study to further strengthen the native security policy built in commodity operating system kernel. One of such successful projects is the Security Enhanced Linux (SELinux) which was developed by America's National Security Agency for their own internal use for Linux patches and later became an official security module.

Later, several computer kernel development scientists and engineers also began pushing for their own security module to be integrated into the mainstream kernel until Linus Torvalds proposed the idea of a project dubbed, the Linux Security Module (LSM) to allow dynamic loadable security modules to protect the kernel (Edge, 2012).

In this chapter, we will examine the various Linux security Modules and other integrated kernel retrofit techniques which have been developed to enhance the protection of commodity operating system kernels from exploits. The various strengths and weaknesses of these technologies shall also be addressed.

2.1 Overview of LSM Framework

LSM Framework creates an Application Programming Interface (API) to permit the Linux kernels to provide support to the several kernel security models that has implemented the framework's standard. The motivation behind its creation is to allow uninhibited selection of kernel security model of choice while avoiding a fixed hard-wired module.

The LSM project was developed to effectively implement Mandatory Access Control (MAC) without altering the base kernel yet augmenting the traditional Unix Discretionary Access Control (DAC) service already provided by the Linux kernel. The rationale behind the development of an additional security mechanism (MAC) to enhance the kernel security is because, the extent to which DAC is implemented relies on user discretion on access constraints. While DAC provides restrictions for file system access, the need for security mechanism to enforce defense against threats of secured objects in systems such as Network Sockets, IPC among others which cannot be circumvented by users became inevitable.

A more intelligent implementation of the Linux Security Module architecture is the preventive access control mechanism. The architecture of the LSM is as shown in figure 2.1. It uses a technique of hooks by adding a security field to the Linux structure. In so doing it loads the credentials of the program in order to know the module to load and whether it meets the policy requirements (Wright et al, 2003).

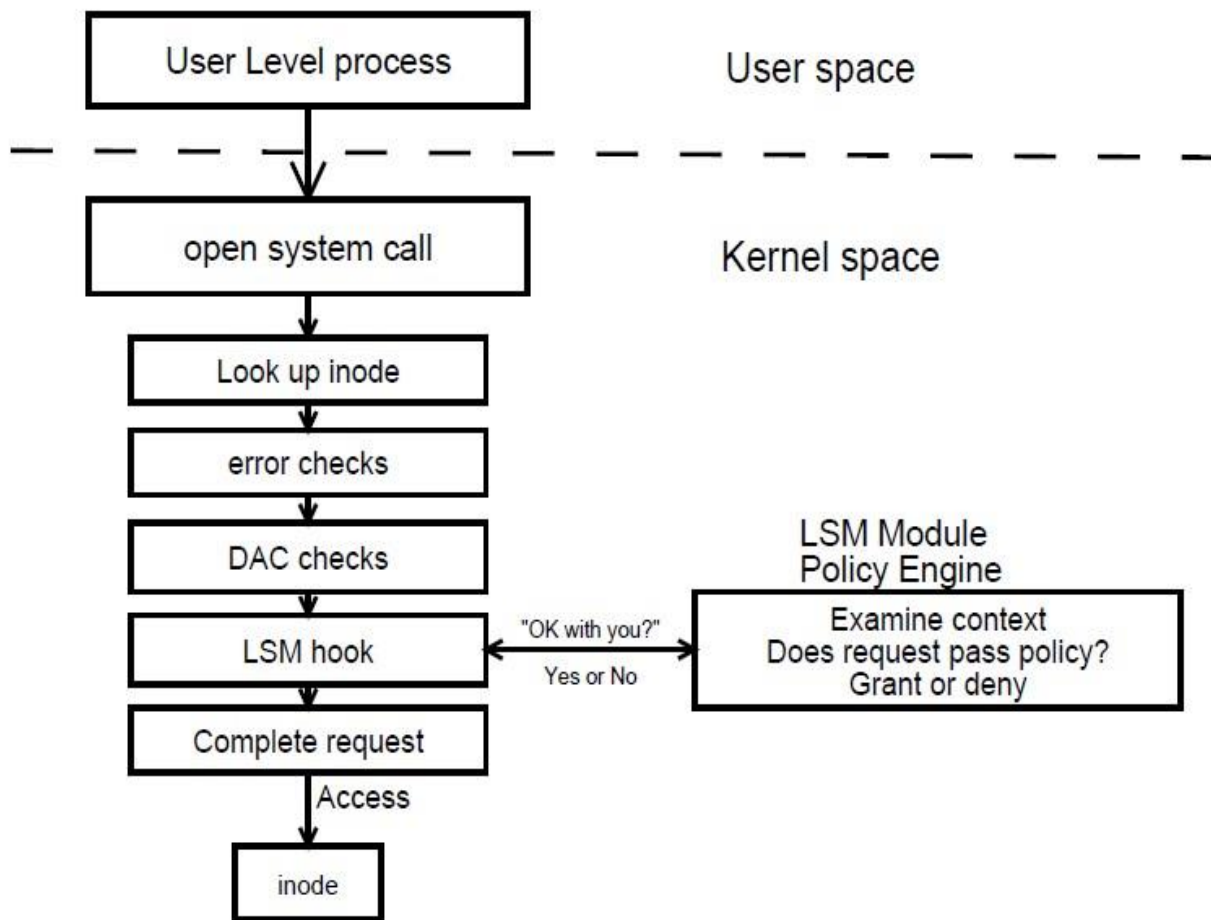


Figure 2.1 Linux security module architecture (Source: Wright et al, 2003)

2.2 Evolution of OS Security Framework

Traditional UNIX (the mother of Linux) architecture was created without adequate emphasis to security (Alm, 2006).

This was not however considered as a weakness to the architecture since data protection and operating system level vulnerability did not exist at the time as a threat. In the early 90's, the first worm attack across the globe exposed the vulnerabilities of operating system and the debate of data protection. The only form of protection at the time was the Discretionary Access Control

(DAC) which is user defined to protect user files and directories and are subject to the discretion of the user.

The weaknesses of DAC became so highly evident when during the over reliance on networking and therefore led to the development of other frameworks. The MAC framework where the mandatory access control is managed was introduced to protect the kernel. Subsequently, other framework such as the, SESBD, Flask and SELinux frameworks were introduced to further enhance the architecture of the monolithic kernel.

2.3 Generalized Framework for Access Control

Generalized Framework for Access Control (GFAC) model divides the method of access control into two parts which falls under Access Decision Facility (ADF) and Access Enforcement Facility (AEF). When an instance of a task is meant to be performed the AEF queries the ADF for a policy decision. The AEF presents the access control list required of ADF to reference in order to match the policy. The architecture shown in figure 2.2 gives further details of the design.

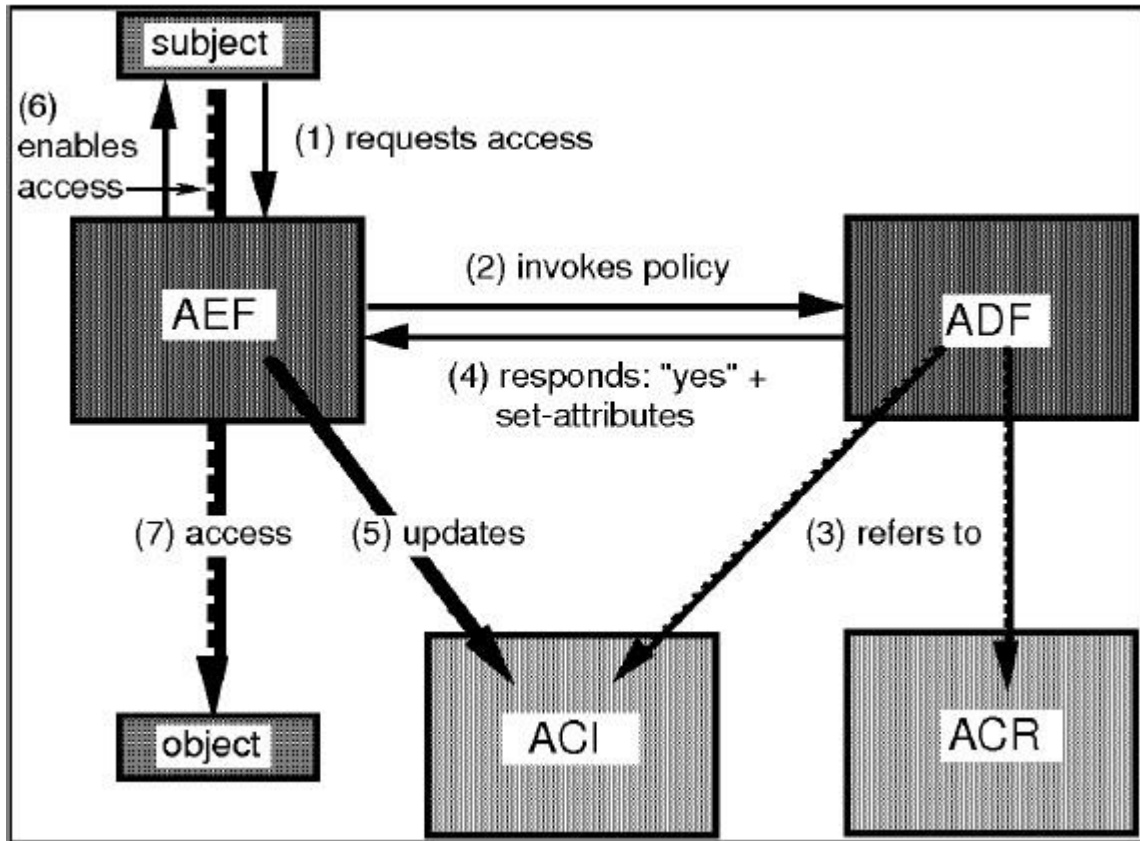


Figure 2.2. Overview of GFAC Architecture (Source: Karlsson, 2010)

Figure 2.2 depicts the GFAC access control process containing numbered units referenced in the following manner of the access control process. Anytime a process say (1), tries executing an operation, the AEF queries, (2), the ADF for decision on a policy. Being part of the query, the AEF provides the information for the ADF to make a decision, denoted Access Control Information (ACI). The ADF then utilizes the gained information to search for relevant access rules, (3), in the Access Control Rules (ACR) database. Afterwards there is returns thus, (4), to the AEF. The AEF then enforces the ADF's decisions about the needed control requests to deny or allow access.

The enforcement facility, AEF, is described as a state machine where each system call is represented by a transition rule, to accurately model the operations of the target system. The rules can also be chosen as generalizations of several operations, using a mapping between system calls and rules (Karlsson, 2010).

2.4 Flux Advanced Security Kernel

Flux Advanced Security Kernel (FLASK) model was created with the goal of allowing random policy flexibility to protect any part of the system flagged that need to be protected. It also has two main parts namely the object managers and the security servers. The object manager enforce security decisions made while the security server maintains the policy.

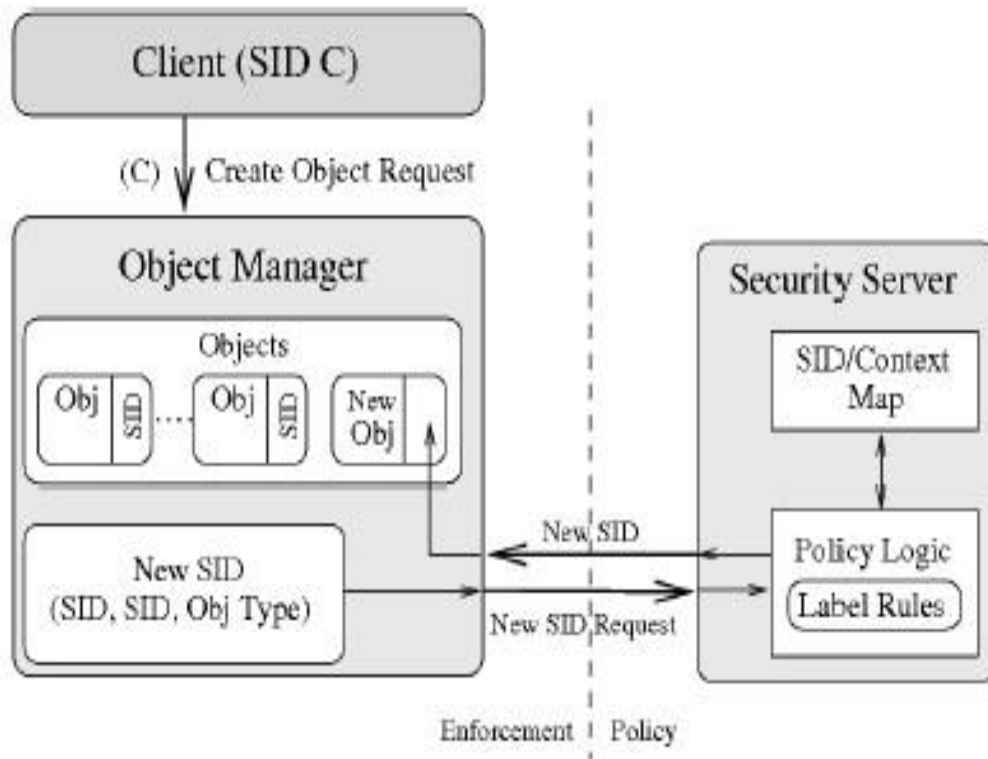


Figure 2.3. Overview of FLASK architecture (Source: Karlsson, 2010)

Smalley et al as part of a Defense Advanced Research Projects Agency (DARPA) funded project are credited with the invention of the Flux Advanced Security Kernel (FLASK) model. The model is based on the prototype system Distributed Trusted Operating System (DTOS), similar design, although lacks mechanism flexibility. Superficially, FLASK is consistent with the GFAC model, but attempts to secure non-atomic policy operations (Spencer et al., 1999).

FLASK is basically meant for allowing arbitrary policy flexibility in order to cater for as many security interests as possible. This is achieved by allowing the access controls refined grained and propagation of access rights conformity to policy. Moreover, in order to fully support dynamic policies and policy changes, the model must support revocation of previously granted access rights. Secondary goals of the model include application transparency, ease of assurance and minimal performance impact (Bugiel et al, 2013).

Considering a computer system as a state machine is one of the surest ways of achieving total policy flexibility whose state transition and operations and where the policy then can mediate any operation atomically. The complexity of a real system makes it difficult to model this way, and in FLASK a compromise is sought by subjecting a security relevant part of the system to such policy control.

Figure 2.3 shown above depicts an overview of the architecture of the FLASK model whose architecture is similar to that of GFAC, and the policy enforcement is sub divided into two parts namely: the Object Managers and Security Contexts (Liaropoulos & Tsihrintzis, 2014).

Object managers which enforce security decisions made by security servers based on the security policy. The object managers implement a control policy which allows the security policy to govern the object managers' subjugated objects, by defining the actions taken for security decisions made by security servers. They are also required to define a mechanism for assigning labels to their objects (Manual, 2011).

Security contexts or labels are strings which can be interpreted by applications or users, but are opaque to object managers. Security contexts are passed onto the security servers, which maps them to security identifiers (SIDs) which is a unique identifier used to reference a specific security context, within the security server, where it represents a policy permission statement (Haines, n.d.).

Access Vector refers to permissions returned by a security server in return of an access request. Access requests are made using the SIDs of the requesting subject and the target object as parameters, signified source SID (SSID) and target SID (TSID) respectively. Since access requests are often made multiple times for the same object, the model defines an Access Vector Cache (AVC) which allows object managers to cache access vectors made by the security servers (Melorose et al, 2016).

2.5 Domain and Type Enforcement

Domain and Type Enforcement (DTE) designed by Badger and Sterne as an alternative to pervasive military-style security frameworks with focus on confidentiality. The greatest motive for this work was to ease adoption of MAC by providing low system administration overhead, application compatibility, as well as unobtrusive operation (Badger et al, 1995).

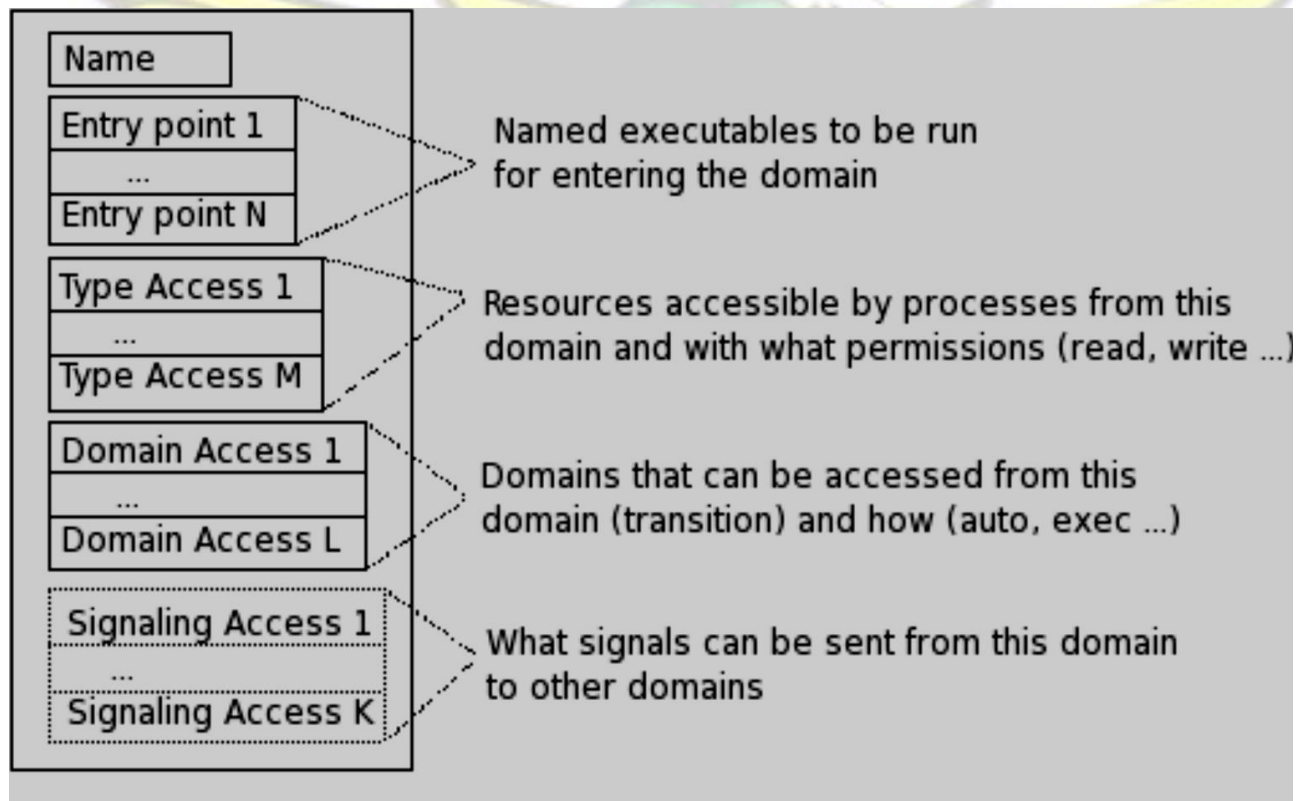


Figure 2.4. Domain and Type Enforcement (Badger et al, 1995)

In order to satisfy these requirements, DTE extends on Type Enforcement (TE), a table-oriented MAC mechanism. TE also assigns attribute domain to each subject and type to each object. The allowed interactions between domains and types are then kept in a universal or global table, the Domain Definition Table (DDT), which is an ACM with domains and types in place of subjects and objects. Similarly, interactions between subjects are mediated using the Domain Interaction Table (DIT). (Watson, 2013)

On the other hand, the strictly table-oriented approach in TE has several limitations addressed in DTE by expressing policy in a high level language through implicit typing of managed objects. The Domain and Type Enforcement (DTE) policy language looks at how express the information of the DDT and the DIT appears in a human-friendly or readable form. It further adds features such as macros and automatic domain transition rules. Implicit typing means that the security attributes of subjects and objects are kept in memory instead of being stored on disk. The database for memory is created upon system initialization from log files updated on file system changes such as file creation or renaming. Attributes are kept or stored sparsely, applying recursively to directory structures if possible.

2.6 Distributed Security Infrastructure

The Ericsson Research in Canada developed the Distributed Security Infrastructure (DSI) for use in Linux Clusters with intentions of moving the scope of MAC from unique nodes to an entire cluster, enforcing a common, distrusted, security policy. The DSI mode is divided into management as well as services components. The management components basically includes a security server, security managers and a security communication channel and the Security server is designed to take care of the distributed security policy. The security manager enforces the

security policy at each node and the security communication channel provides encrypted and authenticated communication between the management components (Pourzandi et al, 2002).

The service components provide security mechanisms for the security managers and consists of two types; security services and security service providers. The latter which provides services for the former, such as secure key management. For access control, the DSI model divides access types into four categories; local, remote, outside, and no cluster access modes (Klemm, 2001, Zakrzewski, 2002).

Only the first two are considered by the model. During local accesses, in which the subject and object are on the same node, access permissions are based on the subject's SSID and the target's TSID, much like in the FLASK model. For remote accesses, where the subject and object are on different nodes in the same cluster, the access queries are extended to include node identifiers for both the subject and the object (Dallas, n.d.).

2.7 Security Mechanisms

Security mechanisms, also known as protection mechanisms, are operations of security concepts which basically protects a subset of an entire system. This concept is quite wide, as one may choose to encompass an entire system implementation. Mechanisms are usually considered to lie in one of the following security domains; authentication, authorization, auditing, or encryption (Brinkley & Schell, n.d.).

Authentication is the process of verifying the identity of a user to a system.

Authorization is the act of determining which resources of a system to make available to a, usually authenticated, user.

Auditing is the gathering of security information, such as access violations, for use in security assessments.

Encryption, or **cryptography**, is the practice of hiding information, usually in plain sight, by scrambling it using a reversible mathematical process.

- Discretionary Access Control

Discretionary Access Control (DAC) is defined in the Trusted Computer System Evaluation Criteria. (TCSEC) to control access between users and objects, and to allow users to specify and control sharing of those objects to other users or groups of users. Most DAC implementations further restricts this definition in that the objects have an owner, who will have primary control over their owned objects' permissions. It is common to make use of ACLs for specifying the policy used in systems employing DAC (Department of Defense, 1985).

- Mandatory Access Control

Mandatory Access Control (MAC) is also defined in the TCSEC to be mainly associated with Multilevel Security (MLS), primarily used in military systems, which makes it too narrow for public or general use.

Generally, MAC can be said to enforce policy decisions on access control made by a central authority, for example a system administrator, as opposed to the owner of an object in DAC. The enforcement must be performed reliably, for instance by the operating system's kernel to see to users inability to override the policy (Publications & Publications, 2008)

- Role-Based Access Control

Role-Based Access Control (RBAC) enforces access decisions based on the roles of the subjects. With RBAC, subjects are authorized to pick roles, which represents a set of permissions through assuming a role, the subject will also gain use of the permissions of that role (Finin et al., 2008).

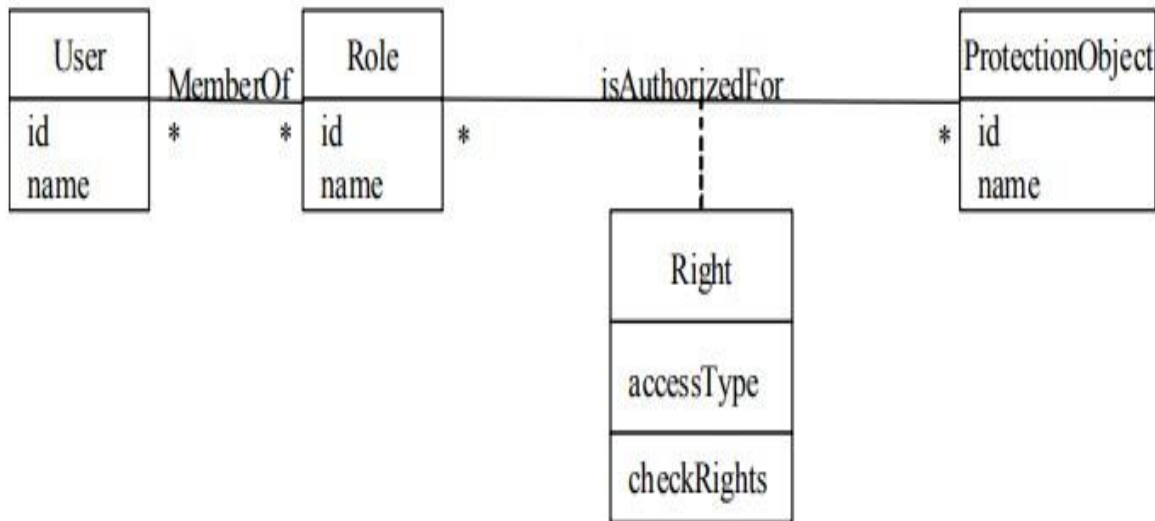


Figure 2.5. Role Based Access Control Source: (Fernandez, Pernul, & Larrondo-Petrie, 2008)

- Reference Monitor

The reference monitor describes the necessary and sufficient properties for achieving secure access controls in a system. It is an abstract model which does not require a specific implementation or policy. In any case, any specific implementation will enforce a specific policy set. The model describes three properties of a reference monitor; it must be invoked for every access request, be tamperproof, and be small enough to be tested for completeness (Control, 2007)

The first property of been invoked for every access request makes sure that the security policy is always invoked when a policy decision is needed, as it would otherwise be possible to circumvent the policy. The mechanism must be tamperproof to be safe from attacks directly against the mechanism, causing a Denial of Service (DoS) for example (Hu et al., 2014).

Lastly, the mechanism must be verifiably secure, as it could not otherwise guarantee enforcement of the policy. Reference monitors are often used as the most basic entities in secure systems, and are required to implement some security models, such as FLASK.

2.8. OTHER CONCEPTS

Roles are usually used as a way of describing responsibility, where multiple subjects can share the same role, or responsibility. This enables roles to be updated without updating the permissions of individual users. Role hierarchies can be created to more closely reflect the natural structure of an organization, than other access control mechanisms (Reilly, 1998).

In modern Linux distribution, netfilter is responsible for caring for network security as compared to security in traditional Linux kernel consisting of file system, or Discretionary Access Control (DAC). Netfilter as a framework for manipulating network packets in the Linux network stack is commonly associated with designing firewalls purposes. Other tools for network security as well, such as OpenSWAN for IPSec (Red & Enterprise, n.d.) (Krátký et al., 2016) These are however beyond the scope of this thesis

- **Linux Security Modules**

The Linux Kernel did not initially permit Loadable Kernel Modules (LKMs) facilitating access to kernel objects out of their scope. Thus, several previous attempts at creating security modules have used system call interposition which is not suitable for access control, due to the ease with which it can be circumvented. There is the requirement for these wrappers to request frequent patches to the kernel or even re-implement certain kernel functionality because of the limited LKMs scope (Wang, 2012).

In the 2001 Summit of the National Security Agency (NSA) a presentation was made on their new security framework Security-Enhanced Linux (SELinux). They argued for the need for a flexible access control architecture in the Linux kernel, but Linus Torvalds, the kernel maintainer, took heed by instead suggesting a new infrastructure for implementing security using LKMs.

Through the suggestions by Linux Torvalds, there was creation of a lighter weight access control framework that allows multitude of security models implemented as LKMs bringing about the Linux Security Modules (LSM) framework. The design goal of LSM was to allow modules to

answer the question “May a subject S perform kernel operation OP on the internal kernel object OBJ?” Moreover, the design had to be generic, simple, minimally invasive, efficient, and be able to support the existing POSIX.1e capabilities logic.

The Linux Security Modules places hooks in the kernel right before accesses to internal objects are granted, where these accesses may be rescinded by the policy implemented in an LSM module. Due to the effortlessness design constraint, LSM hooks are mainly restrictive, in that they are only used when the kernel is about to grant access, not when about to deny access. The comprehensive solution, authoritative hooks, would have been significantly more complex. A few permissive hooks exist, to allow the implementation of POSIX.1e capabilities as a LSM module (Zakrzewski, 2002).

In order to allow modules to keep track of special security properties of objects, LSM attaches security fields to mediated objects. The modules are responsible for managing the data in these fields themselves (Leggieri et al, 2014).

The LSM implementation has been comprehensively tested automatically, regarding the placement of hooks. While most studies found exploitable bugs in the placement of hooks in early implementations, they also agreed with the already manually chosen hook placements (Jaeger et al, 2004, Ganapathy et al, 2005).

- Available Security Frameworks

A number of projects has designed and brought about security modules following the inclusion of LSM in the Linux Kernel. There are also frameworks which does not employ LSM, because of technical reasons. The examples of policy language presented with some of the following frameworks are intended to highlight the configuration syntax complexity and readability (Wright et al., n.d.) (Corbet, 2008).

- POSIX.1e Capabilities

Currently, the POSIX draft 1003.1e bids to standardize UNIX security measures by introduction to the Portable Operating System Interface for UNIX (POSIX) standard. Linux includes an implementation of a part of this draft, the capabilities framework (Howard et al, 2010).

This framework in Linux work by partitioning the all-encompassing root privilege into separate privileges, called capabilities which can be assigned to processes allowing these processes to run under other user identifiers (UIDs) other than root, but with some of its privileges (Works, 2000).

With this capability-enabled system, the processes has three sets of bitmaps each describing which capabilities are available to it: inheritable (I), permitted (P), and effective (E) respectively. These effective set contains the capabilities a process can use, the permitted set contains the capabilities a process is permitted to put in its effective set, and the inheritable set contains the capabilities which are allowed to be inherited by a program executed by the process (Linux & Works, n.d.).

- AppArmor

This is an application-centric MAC framework built using LSM first developed by the company Immunix, which used it in its GNU/Linux offering between 1998 and 2003. When Immunix was acquired by Novell in 2005 and maintained AppArmor until 2007, when the AppArmor team was laid off (Cisco, 2013).

The framework has the following design goals: It needs to be secure to keep applications in confinement, fast to avoid overhead, and transparent to allow normal operation of software. Security is achieved by using LSM which is more secure than system call interposition, as described in Section 3.1. Speed is achieved by a simple security model with fast lookup which does not need caching. Transparency is achieved by reusing UNIX security semantics on applications instead of users (Universit, 2010).

AppArmor policies, called profiles, consists of sets of POSIX.1e capabilities and file permissions attached to programs. The identical of rules to programs is done via path lookup or search, with spawning discussion with primarily SELinux proponents on the relative security merits of this method and other design aspects of AppArmor (“SUSE Linux Enterprise Server 10 SP1 EAL4 High-Level Design,” n.d.).

Listing 2.1 shows an example of an AppArmor profile with use of capabilities, files and network controls.

Initial execution of AppArmor automatically generates profiles in a so called “learning mode”, where all rule violations are logged, and then analyzing these logs interactively. This method can also be used to incrementally improve a profile.

Listing 2.1: Example of an AppArmor profile for /bin/ping

```
#include <tunables/global> /bin/ping
{
    #include <abstractions /base>
    #include <abstractions /consoles>
    #include <abstractions /nameservice>

    capability net raw , capability setuid ,
    network inet raw ,

    /bin/ping mixr ,
    /etc/modules . conf r , }
```

- DISEC

DISEC developed by Ericsson Research in Canada between 2002 and 2004 is the collective name for a set of utilities implementing the DSI security model but it is now outdated.

DSI's access control enforcement is implemented using LSM, and the distributed security policy is specified with XML (DISEC, 2015).

Though the model does not point out the mediated object types, DISEC implements access control with process level granularity. This means that it only concerns itself with the spawning of processes, the execution of programs, and the communication between processes on the cluster network (Pourzandi et al, 2005).

- GRsecurity

GRSecurity birthed by Brad Spengler in 2001 as a port of the Openwall kernel patch from Linux 2.2 to 2.4 is a patch-set for the Linux Kernel which emphasizes the security-in-depth principle and enforces MAC through RBAC. It has since become a project of its own. Included in the GRsecurity patch-set is a broader security which is allowed through the memory protection patch PaX (Spengler & Security, 2012).

Custom hooks in targeted systems calls and kernel functions are used in implementation of GRsecurity, as opposed to system call interposition or use of LSM. Among the reasons for not using LSM are the greater functionality requirement of GRsecurity than available through LSM, and that LSM could further enable root-kits rather than hamper them (Grattafiori, 2016).

The goals of GRsecurity are

- Simplicity via configuration free operation
- Protection against address space modification bugs
- Feature-rich ACL and Auditing
- Support for multiple processor architectures

ACLs and the user space tool gradm are basically used in managing the configuration of GRsecurity. The ACLs consists of roles and their subjects, which in turn contains file, capability,

socket and computational resource controls. Roles can define transitions into other roles and subject properties can be inherited from parents in the file system (Spengler, 2003).

Listing 2.2 shows an example of an ACLs with a role and a subject with file and socket controls for the sshd Unix daemon.

Listing 2.2 Example of a GRsecurity ACL

```
role sshd u subject /  
    / h  
    /var/run/sshd r -CAP ALL  
    bind disabled  
    connect disabled
```

ACLs can be automatically created with gradm for either the entire system, a specific subject, or a specific role. This learning mode will attempt to create a minimal ACL for the task presented and can take hints on how expressively to generate rules for certain subjects (Spengler & Security, 2012).

• Linux Intrusion Detection Systems

The Linux Intrusion Detection System (LIDS) developed by Xie Huagang and Philippe Biondi is contrary to its name, a MAC framework but also includes a port scan detector and process protection facilities. The patch-set has been maintained since version 2.2 of the Linux Kernel and has since been ported to the 2.6-series, and is now using LSM (Biondi, 2002).

Mediated objects are files, sockets, directories and POSIX.1e capabilities. Configuration of access rights is handled by the command-line program lidsconf which is stored in text files and can be edited by hand, but the command-line tool is the preferred way of changing the security policy.

Listing 2.3 shows an example of lidsconf usage which will generate a configuration based on the current file system information. This means that the configuration command will have to be re-run when, for instance, a file is overwritten and its inode2 number changes (Phones, 2008).

Listing 2.3: Example of a LIDS ruleset

```
/sbin/ lidsconf -A -R -o /etc/log -j APPEND
/sbin/ lidsconf -A -o /var/log/wtmp -j WRITE
/sbin/ lidsconf -A -s /bin/ login -o /etc/shadow -j READONLY /sbin/ lidsconf -A -s /bin/su -o
/etc/shadow -j READONLY /sbin/ lidsconf -A -s /bin/ login -o CAP SETUID
```

- Rule Set Based Access Control

Rule Set Based Access Control (RSBAC) is a security framework for the Linux Kernel designed by Amon Ott, with similar goals as LSM, but with some design difference and level of abstraction. It is based on the GFAC model (Ott & Fischer-hübner, 1995).

The framework uses system call interception as its system call mediation method. It avoids LSM because of, for example, limited scope of mediation, lack of concurrent capability support, and stateless calls (Provos, 2003). In RSBAC, the GFAC model has been modified to include a specific metapolicy for the ADF policy decision process, breaking up the ACI into loadable modules, and further abstract the AEF by not allowing it to update ACI after a successful access control request. The metapolicy is the boolean AND operator, with concessions for the different return types of the policy modules. The ACI policy modules have four different return values: granted, not granted, don't care, and undefined. Table 2.1 illustrates the results of the metapolicy on two operands (Ott & Fischer-hübner, 1995).

Table 2.1: Policy result combination in RSBAC's ADF

Result 1	Result 2	Final Result
Granted	Granted	Granted

Granted	Don't care	Granted
Not granted	Granted	Not granted
Not granted	Don't care	Not granted
Undefined	*	Undefined

The currently available modules are shown in Table 2.2.

Table 2.2: Available RSBAC ACI modules

Name	Code name	Description
Authenticated User	AUTH	Advanced user authentication
Role Compatibility	RC	Role-Based Access Control
Jail	JAIL	Encapsulation of processes
Linux Capabilities	CAP	Manages Linux capabilities
Pageexec	PAX	Prevents against unwanted code execution
Dazuko	DAZ	On-access anti-virus scanner
File Flags	FF	Per file access control flags
Linux Resources	RES	Manages Linux resources
User Management	UM	In-kernel user management
Access Control Lists	ACL	Extensive Access Control Lists
Privacy Model	PM	Controls data privacy
Mandatory Access Control	MAC	Bell-La Padula with modifications

Overview of SELinux

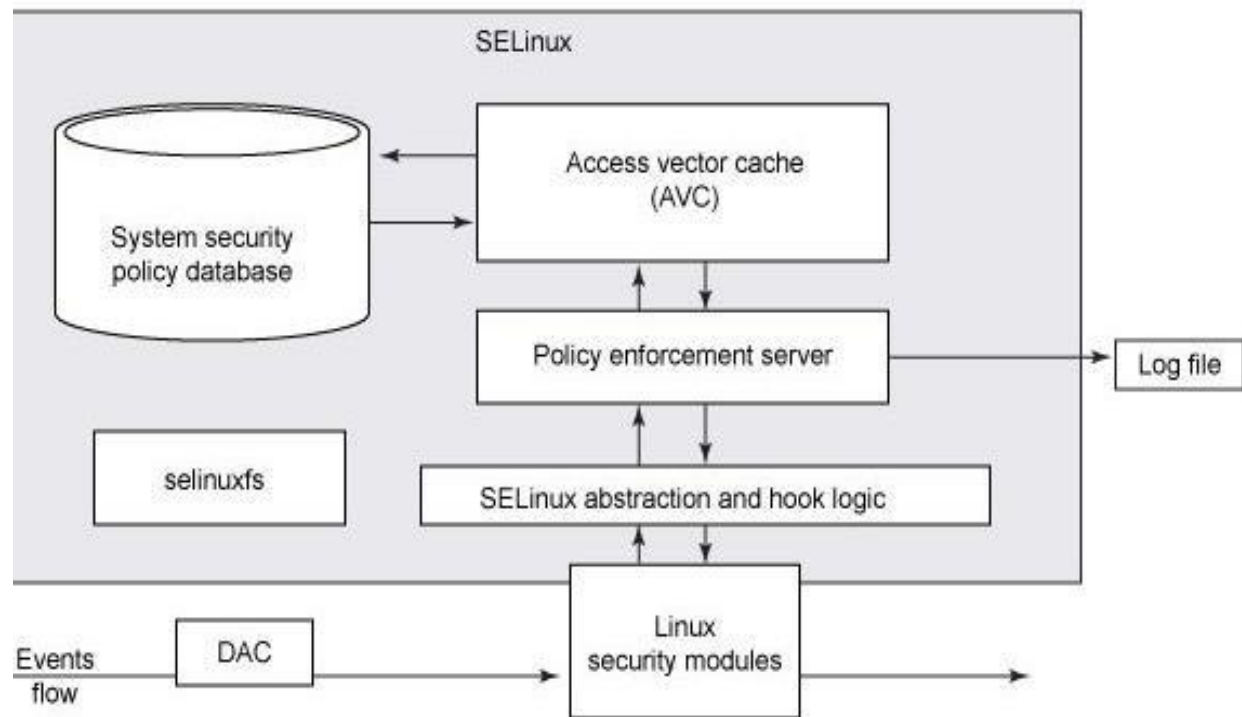


Figure 2.6. Overview of SELinux (Source: Karlsson, 2010)

The operating system subject (process) attempts to perform a certain action on a particular object (file, process, socket), which is permitted within the Linux standard discretionary security system (DAC). This launches a stream of requests to the object. Every request to perform the action with the object is intercepted by the Linux Security Modules and is transferred, along with the subject's and object's security context, to the SELinux Abstraction & Hook Logic subsystem, which is responsible for interaction with LSM. The information received from the SELinux Abstraction and Hook Logic subsystem is forwarded to the basic Policy Enforcement Server module, which is directly responsible for making the decision about allowing the subject to access the object. To receive the decision as to whether the action is allowed or prohibited, the policy enforcement server contacts the special Access Vector Cache subsystem, which most often caches the rules being used.

If AVC does not contain the cached decision for the relevant policy, the request for the necessary security policy is forwarded again to the security policy database.

When the security policy has been found, it is transferred to the policy server receiving the decision. If the requested action complies with the policy that has been found, the operation is permitted. Otherwise, it is prohibited and all the decision-making information is written to the SELinux log file.

Security-Enhanced Linux (SELinux) is an implementation of the FLASK security model in the Linux kernel, initially developed by the NSA and the Secure Computing Corporation (SCC) (Kim et al, 2008).

The Initial prototype of SELinux was built with custom hooks in Linux 2.4, however, it was triggered for the creation of the LSM framework. It was the first implementation of LSM to be accepted into the Linux kernel, aside the POSIX.1e capabilities module. File security contexts are stored in Linux' file system extended attributes, for performance reasons (Ben-Cohen et al, 2008).

Similar to its mother project, FLASK, SELinux's main priority is policy flexibility. As an example, the security server implements a security policy which is a combination of TE, RBAC and MLS. The security context is defined as a string containing a user identity, a role, and optionally a MLS level. Roles are only used for processes, so the role object *r* is used for all file security contexts (Papachristodoulou & Antakis, 2008).

The policy configuration language is based on TE and contains definitions of domains, types, roles, and their respective interactions. These include domain and role transitions, which specifies which changes in security context may occur at and during program execution. Listing 2.4 shows a snippet of a policy configuration for the *dhcpcd* daemon, where the *dhcpcd t* type, or domain, is given access to network domains.

Policy configurations can become quite complex, and a reference policy containing rules for

Listing 2.4: Snippet of a SELinux policy

allow dhcpdt netif type : netif { tcp send udp send rawip send }; allow
dhcpdt node type : node { tcp recv udp recv rawip recv };

- SMACK

Simplified MAC Kernel (SMACK) is a MAC framework written for the Linux kernel by Casey Schauher and implemented using LSM and is included in the kernel since 2.6.24.

The security model in SMACK is a mix of DTE and Bell-La Padula. The security attributes of objects are case sensitive strings between 1 and 23 characters in length, and are called labels. There are four predefined labels listed in Table 3.3. Subjects, or processes, are called tasks and system tasks are given the oor label. Access permissions are enforced by the following rules, in order:

- Any access requested by a task labeled “*” is denied
- A read or execute requested by a task labeled “^” is permitted
- A read or execute requested on an object labeled “_” is permitted
- Any access requested on an object labeled “*” is permitted
- Any access requested by a task on an object with the same label is permitted
- Any access requested that is explicitly defined in the loaded rule set is permitted
- Any other access is denied

Table 2.3: Predefine SMACK lables

Label	Pronunciation
	Floor
^	Hat
*	Star
?	Huh

The policy configuration language consists of one statement: subject-label object-label accessmode

Permissions for labels are given using the traditional access modes of Unix - read, write, and execute - with the addition of append for some object types.

- T CPA

The T CPA framework is not technically a MAC framework, but includes MAC functionality and is therefore included in this thesis. It was developed by IBM but has been abandoned. The framework consists of the modules Extended Verification Module (EVM) and Simple Linux Integrity Module (SLIM), and provides a chain of trust for applications with the goal of minimizing security impacts of application vulnerabilities. The modules are implemented using LSM and are a rare example of LSM stacking. The EVM module is used for validating executables before execution, using a Trusted Platform Module (TPM) (Safford et al, 2005).

The cryptographic verification key is stored in the file system's extended attributes, much like SELinux. The SLIM module is used to constrain privileges for untrusted applications, via MAC. It uses a combination of the Caernarvon and the Low Water-mark models for managing information integrity. All files are given two labels – an Integrity Access Class (IAC) label which contains the classes SYSTEM, USER, UNTRUSTED, and EXEMPT; and a Security Access Class (SAC) label which contains the classes SYSTEM, USER, PUBLIC, and EXEMPT. Process are given the labels IRAC, IWXAC, SWAC, and SRXAC, where the letter R stands for read, W for write and X for execute (Thion, 2008).

SLIM defines the following access control rules, based on the previously given classes:

Read:

$$\begin{aligned} \text{IRAC}(\text{process}) &< \text{IAC}(\text{obje} \\ &= \text{ct}) \end{aligned}$$

SRXAC(proce > SAC(obje
ss) Write: = ct)

IWXAC(proce > IAC(obje
ss) = ct)

SWAC(proce < SAC(obje
ss) Execute: = ct)

IWXAC(proce < IAC(obje
ss) = ct)

SRXAC(proce > SAC(obje
ss) = ct)

Additionally, the module demotes high-classed processes which attempts to read or execute low-classed objects, and vice versa.

- TOMOYO Linux

Task Oriented Management Obviates Your Onus (TOMOYO) Linux is a MAC framework for the Linux kernel developed by NTT Corporation. Version 2.2 of the framework is included in the Linux kernel since version 2.6.30. It is implemented using LSM and features an automatic policy generation system. This system is based on process execution history (Harada et al, 2007).

In Unix, the only way to create new processes is using the scheme “fork-exec”, where a process is divided into two identical processes and then differentiated by replacing the running program in one of them. TOMOYO assigns a domain to each process, and when a new process is started, the security context transitions into this new domain from its parent process' domain. Since the domain transition rules considers the entire process lineage, the same program can belong to several domains, depending on the parent processes (Harada, 2007).

The automatic policy generation simply monitors the process invocations of the system and automatically divides them into domains dependent on their lineage. Other access requests are then

transformed into accepting access rules within each domain. In addition to MAC controls, TOMOYO provides an API which allows applications to relinquish access rights granted by the kernel, to allow for further increase in security (Masumoto et al, 2007).

The policy configuration language is modelled after a user-space view of the kernel, instead of exposing the available internal kernel objects. Permissions are attached to domains, which are string-concatenated process lineage, for example:

```
<kernel> /sbin/init /sbin/getty
```

However, the lineage may be abbreviated to match more instances of a program, like so: <kernel> /sbin/getty

This will match more instances since it is less specific, whereas the previous example would require that init is a parent of getty.

Permissions are given via absolute paths prefixed by keywords. Wildcards are allowed in all paths except for domain transition specifications, where they could lead to ambiguous transitions. An example configuration snippet is shown in Listing 2.5.

Listing 2.5 Example of a TOMOYO policy

```
<kernel> /usr/sbin/smbd use profile  
-  
3   . allow read /etc/samba/smb.conf allow  
    write /var/log/samba/*.log allow create  
    /var/tmp/smbd.+
```

The version of TOMOYO currently in the mainline kernel only supports mediating access to files, directories, and domain transitions; while the older, non-LSM version, supports wider variety of kernel objects, such as network access.

2.9 Summary of Strengths and Limitations of the related work.

Over the past two decades, efforts have been made by researchers and kernel development engineers in enhancing the existing Security of Operating System Kernels. Even though these approaches (as discussed in the literature review) such as the design of numerous security frameworks, development of various kernel level algorithms, and the creation of several kernel design techniques did achieve significant impact in improving the security of commodity operating system kernels, there still is the absence of a single archetypical framework that addresses the seaming challenges of current commodity operating system kernels.

The use of an all integrated kernel level integrated virtualization alongside Highly Available Heartbeat technique via improved throughput in monolithic kernels with an enhanced PAM authentication security module have not been completely implemented by these related work. Moreover, the security framework of the existing commodity Operating Systems is built to be implemented on only a single security model that makes it subject to the vulnerabilities of that model. The solution to this challenge is to design a security framework that could harness the multiple security modules in addition to the novel architectural design to control the various multiple instance virtualization which is non-existent in related works.

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Introduction

Every research is grounded on a core theoretical postulations that establishes 'valid' research. The appropriateness of the approach in the development of knowledge is centred on the research method. This chapter examines the theoretical assumptions and also the design strategies underpinning this research study. Traditional philosophical assumptions in operating system Kernel architecture development which provided a more effective and enhanced prototype was adopted. Furthermore, the chapter examines the research methodologies, and design used in the study including strategies, while explaining the stages and processes involved in the study.

The type of methodology adopted in this research is the Design Methodology which is referred to as a development of a system or method through research philosophies, principles, processes and techniques for a unique situation which is mainly applied in technological fields (Simplicable, 2016). Various conventional methods for investigating the methodology through conceptualizing, verifying, proving, designing implementing and testing operating system Kernels Architecture were implemented during the research. The outcome of the methodology resulted in the design of the Convolved Kernel Architecture and the Stealth Obfuscation Zero Knowledge Proof Authentication which also occasioned the development of the TrendOS prototype to enable testing of the various segments of the designs for testing of the complete system. The techniques used in the actual programming of the kernel prototype involves the Compiler Instrumentation Techniques and Retrofitting Techniques.

3.2 Research Design Methodologies

Three of the research design methodologies were implemented in this chapter which are Structured Analysis & Design Technique, Data Structured Systems Development and Integrated Design

Approach. The three research design methodologies were carefully chosen to provide a detailed step-by-step implementation of the overall design of the framework, to provide the implementation of the internal data structures of the architecture and finally, the design of a modular integrated prototype to effectively carry out a comprehensive evaluation and analysis of the framework.

3.2.1 Structured Analysis and Design Techniques

In this approach to the design methodology, the goal and objectives are first determined to afford the researcher the opportunity to outline the various parameters that will be needed in the design of the Convolutional Kernel Architecture and the Stealth Obfuscation Zero Knowledge Proof. The next phase of this technique is the determination of an acceptable performance success criteria to define the scope of the research as well as the framework of the research. This is also important as the goal and the objectives because, without benchmarks to analyze the kernel architecture, it will be impossible to measure the expected outcome of the product as there will be no performance metrics to measure and compare.

The third stage of this method in the research is the evaluation of the several approaches in OS Kernel development and choosing the basic level of all in achieving similar or same result with reference to the complexity. The most important motivation to selecting the basic level of complexity yet meeting the requirements and objectives is that, in OS Kernel Security development, the more complex the architecture the more computational resources required and the slower the overall system performance. The fourth stage is the qualitative reviews which admitted inputs from various reviewer comments to re-modify the design to meet expected result. The fifth step is the constraint assessment of the architecture to determine whether the design is achievable or not and if so, whether it could add value to the overall outcome of the framework.

In the next segment of this method, detailed performance based engineering design to put together all the various modular designs are conducted. This will therefore require series of validation, review and verification while comparing the analysis with acceptable criteria. Finally, a presentation of the prototype submitted to third parties including NITA, KNUST Cyber Security

Lab and Computer Science students of the Department for assessment. A full description of the Design Methodology as shown below in Figure 3.1.

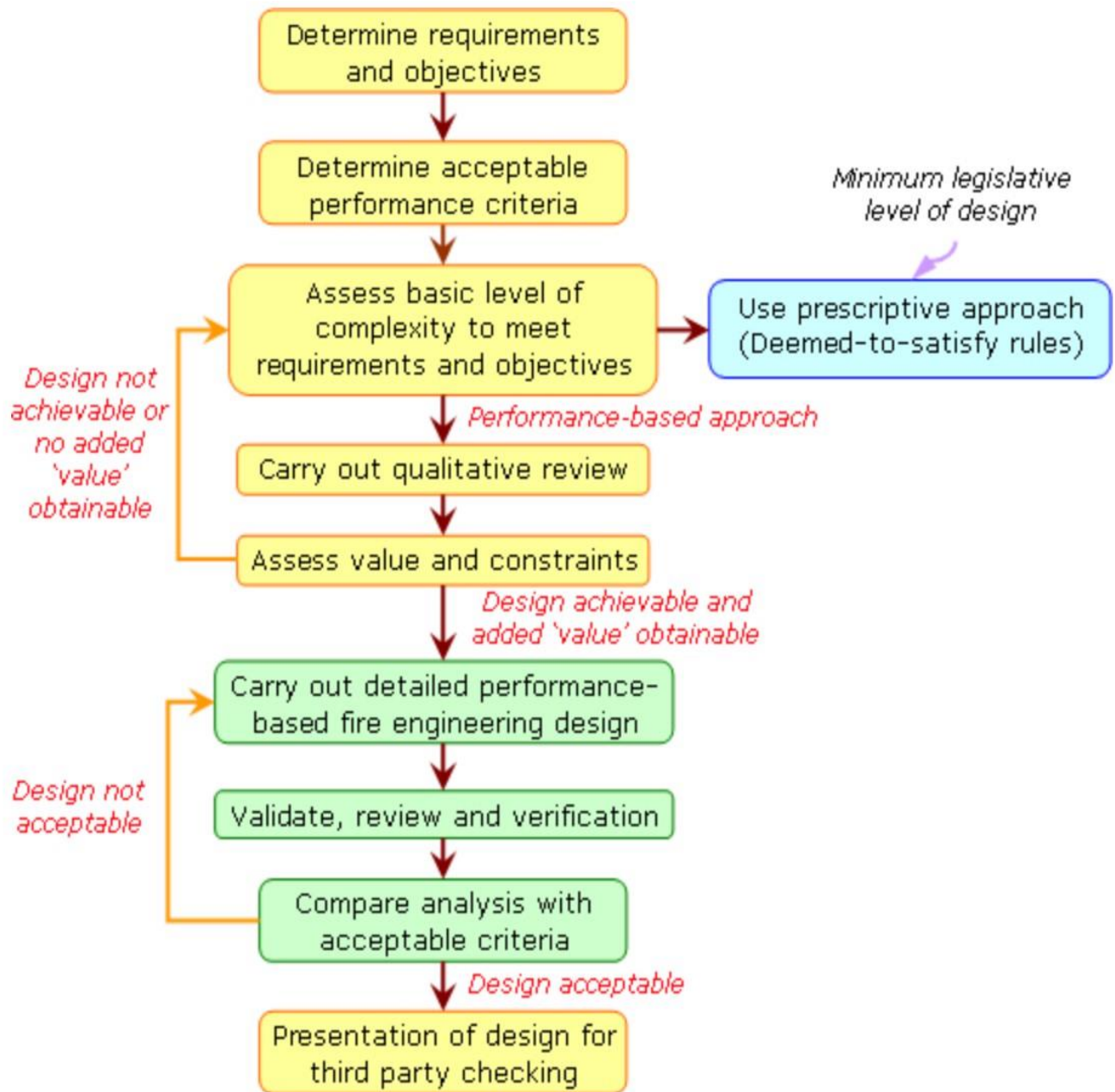


Figure 3.1 Design Methodology Flow Chart (Source: Colin, 2005).

3.2.2 Data Structured Systems Development

This section describes the Data Structures of the internals of the kernel. The following are the details.

Process table: this table contains an entry for each process. Additional data about each process is maintained in the u-area. The u-area and process table entry point to each other. In addition, entries in the process table point to each other. For example, each process points to its parent, and processes are linked to each other in the run queue and various wait queues.

File related tables: the u-area of each process contains the file descriptors table for that process. Entries in this table point to entries in the open files table. These, in turn point to entries in the i-node table. The information in each i-node contains the device on which the file is stored.

Memory related tables: the u-area also contains a per-process region table, whose entries point to entries in the global region table, which describe different memory regions. Data maintained for a region includes a pointer to the i-node used to initialize this region, and a page table for the region. This, in turn, contains a pointer to where the backup for each page is stored in the swap area.

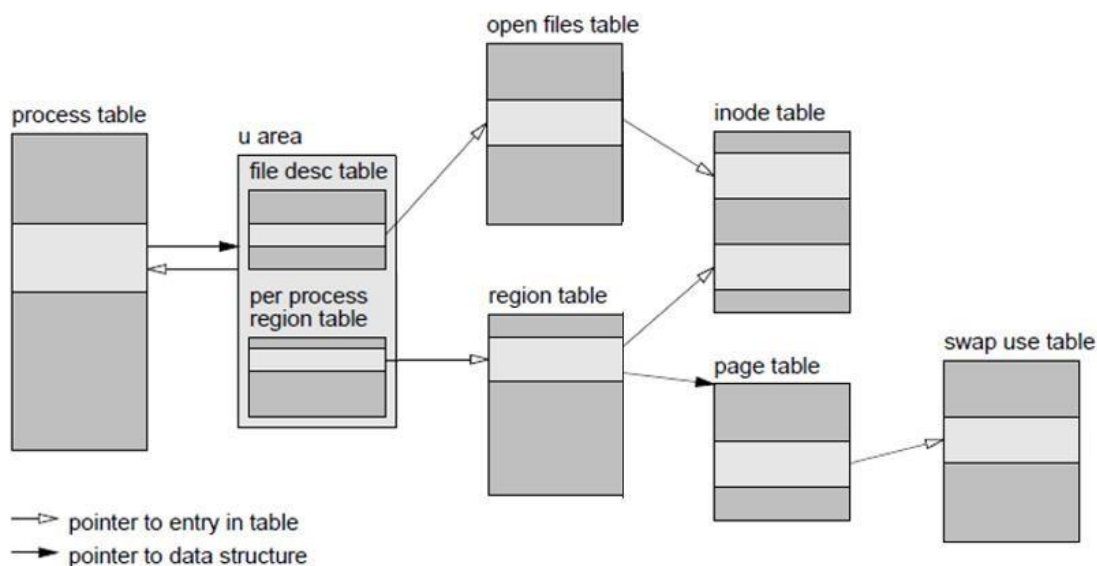


Figure 3.2 Data Structures Source: (Steven & Rago, 2005)

KERNEL FILE SYSTEM DATA STRUCTURE

To understand the file system, you must first think about how the kernel organizes and maintains information. The kernel does a lot of bookkeeping; it needs to know which process are running, what their memory layout is, what open files processes have, and etc. To support this, the kernel maintains three important tables/structures for managing open files for processes: the process table, the file table, and the v-node/i-node info.

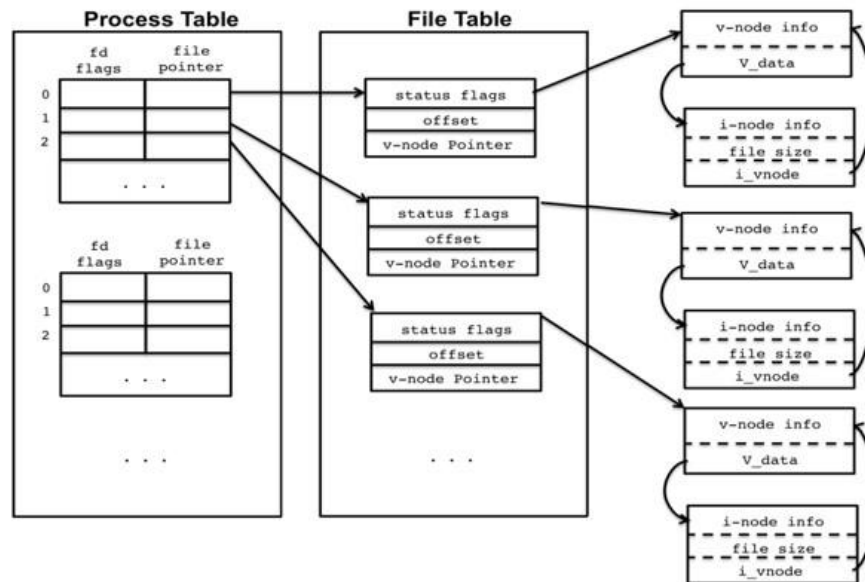


Figure 3.3 Kernel File System Data Structure Source: (Steven & Rago, 2005)

THE PROCESS TABLE

The first data structure is the process table, which stores information about all currently running processes. This includes information about the memory layout of the process, current execution point, and etc. Also included is the open file descriptors.

As we know, all process start with three standard files descriptors, 0, 1, 2, and these numbers and other file descriptors are indexes and stored in the open file table for that process's entry in the process table. Every time a new file is open a new row in the open file table is added, indexed at the value of the file descriptor, e.g., 3 or 4 or 5 or etc.

Each row in the open file table has 2 values. The first are the flags, which describe disposition of the file, such as being open or closed, or if some action should be taken with the file when it is closed. The second value is a reference to the file table entry for that open file, which is a global list, across all process, of currently opened files.

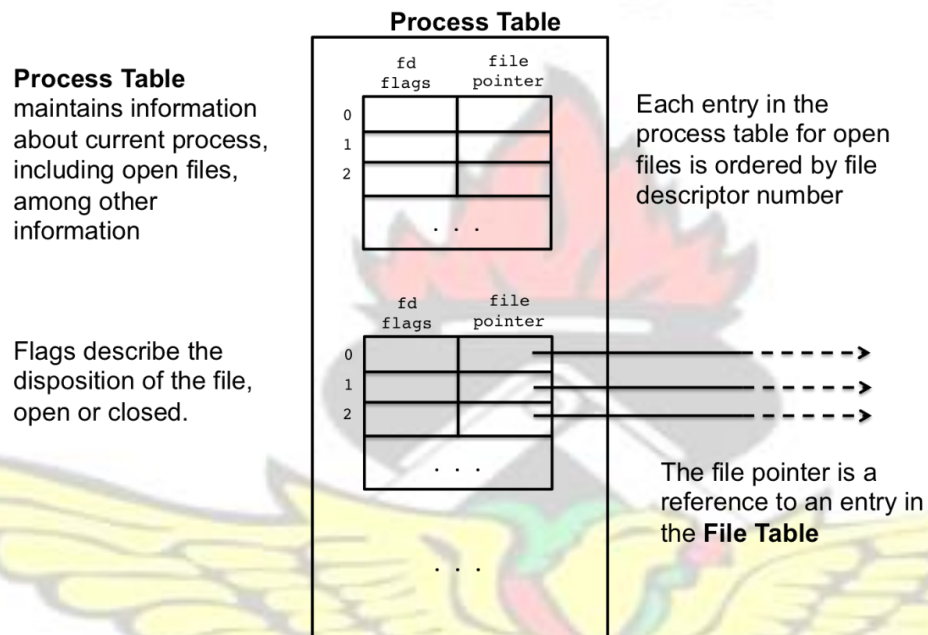


Figure 3.4. The Process Table Source: (Steven & Rago, 2005)

THE FILE TABLE

Whenever a new file is open, system wide, a new entry in the global file table is created. These entries are shared amongst all process, for example when a file is opened by two different process, they may have the same file descriptor number, e.g., 3, but each of the file descriptors will reference a different entry in the file descriptor table.

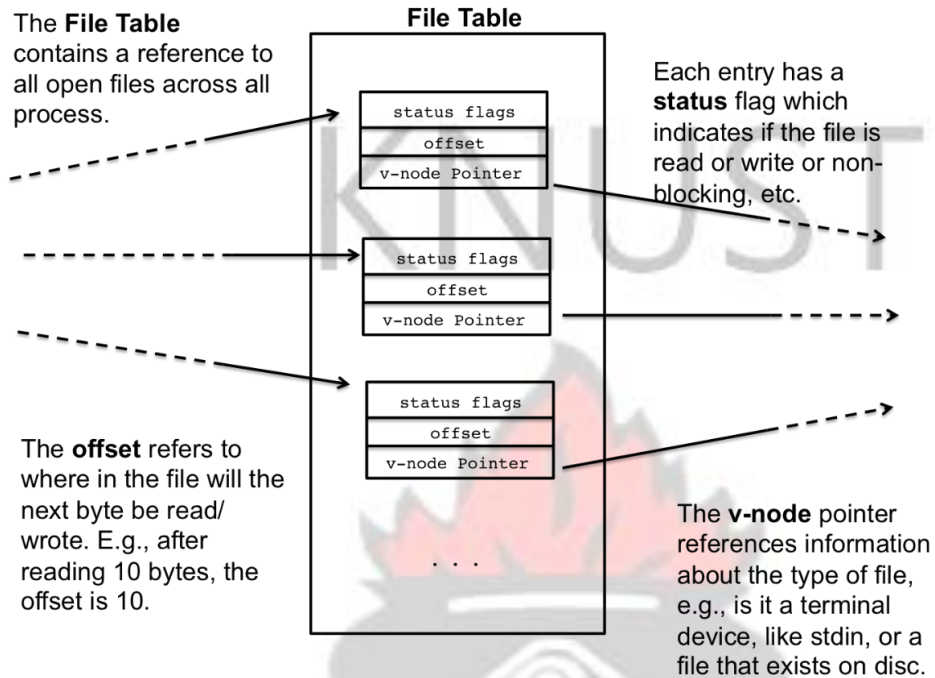


Figure 3.5. The File Table Source: (Steven & Rago, 2005)

V-NODE and I-NODE

The v-nodes and i-nodes are references to the file's file system disposition and underlying storage mechanisms; it connects the software with hardware. For example, at some point the file being opened will need to touch the disc, but we know that there are different ways to encode data on a disc using different file systems. The v-node is an abstraction mechanism so that there is a unified way to access this information irrespective of the underlying file system implementation, while the i-node stores specific access information

V-node and i-node

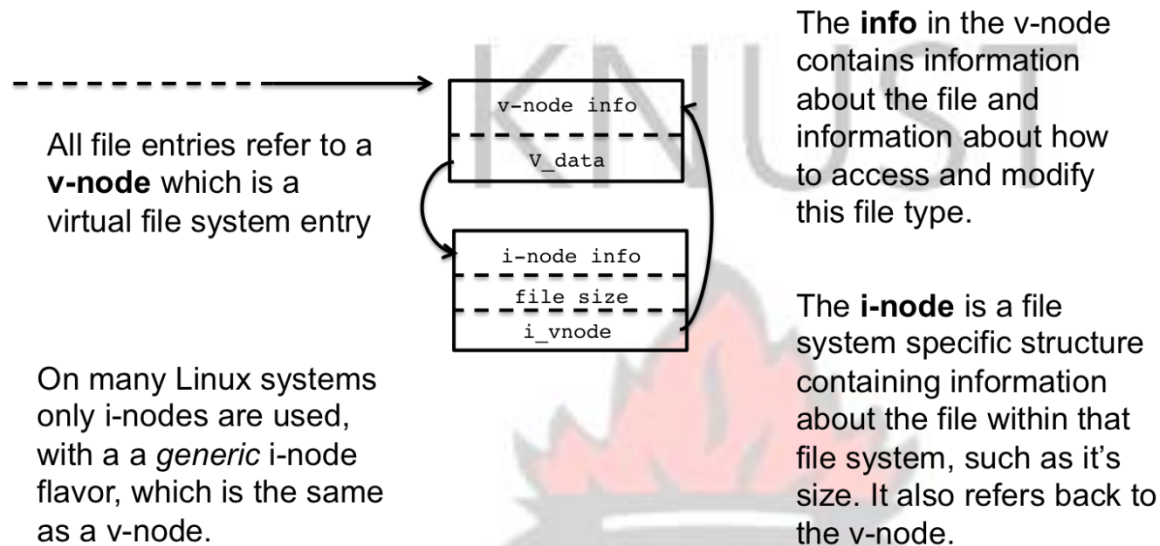


Figure 3.6. The v-node and i-node Source: (Steven & Rago, 2005)

STANDARD FILE DESCRIPTORS

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open or create as an argument to either read or write.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This convention is used by the shells and many applications; it is not a feature of the UNIX kernel. Nevertheless, many applications would break if these associations weren't followed.

The magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. These constants are defined in the `<unistd.h>` header.

Standard File Descriptors

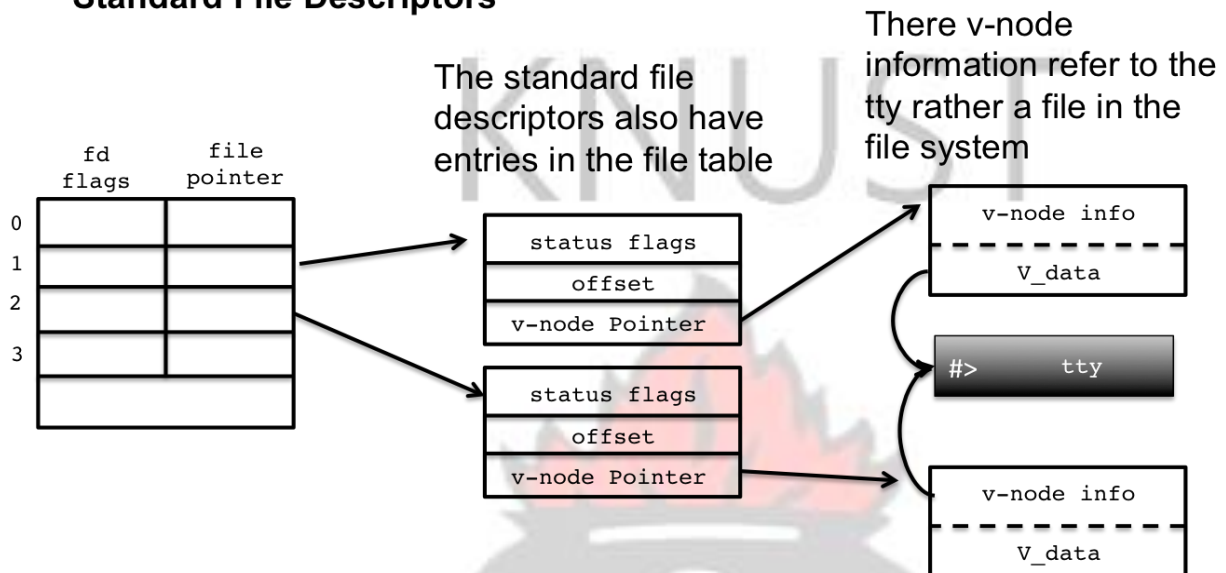


Figure 3.7. Standard File Descriptors Source: (Steven & Rago, 2005)

3.2.3 Integrated Design Method

This is a proposed kernel design methodology that views the kernel elements in some modular Iterative elements which considers the best approach in integrating the various modules in forming the whole architecture. Integrated Approach is an all-inclusive model which ensures that, modules required for operating system ‘hardening’ is integrated and the various elements that it is framed from are encapsulated as a single unit rather than separate independent application layer compartments. It also considers a repeated process of quickly implementing prototypes, gathering feedback and refining the design. The hardcode development is implemented using the best-fit technique in the overall architectural development.

The TrendOS platform was implemented in a 3-part sequence. Namely OS Host Setup and Configuration, Secure Kernel Build, ISO Distro Pipeline (iso-build-script). The following sections describe this process in much greater detail.

3.3 OS Host Setup

A dedicated host system is chosen upon which the TrendOS platform will be built. This system provides the necessary environment to support building the secure kernel in subsequent steps and also to provide a working environment for generating the distributable “iso”. In addition to that, the host system creates a highly configurable environment where its creators have the flexibility and ability to make adjustments to all parts of the operating system by essentially using it. This allows its creators to configure and customize user space binaries with so much precision. The TrendOS platform was built on a dedicated host with the following properties:

Table 3.1 Dedicated host properties

Property	Value
Base Distro	Ubuntu 14.04
Package Manager	DPKG (Debian Package Manager)
Linux Kernel Version	4.4.0-21-generic
Codename	Trusty Thar
OS Arch	x86_64, amd64
Shell	/bin/bash

3.4 Secure Kernel Build Step

The TrendOS platform runs on a 64bit Linux kernel patched with security features from grsecurity. The platform is heavily endowed with security features that prevent kernel based attach vectors throughout execution.

The integration of *grsecurity* was done by compiling it directly into the TrendOS Linux kernel by source. The Linux kernel version chosen for this implementation was 4.4.0. The steps below show the process involved in compiling the TrendOS Linux kernel with the necessary *grsecurity* patches.

NB: The following commands must be executed on the host operating system as described above

Build Requirements

The following packages need to be installed to process with the compilation procedure.

- `libncurses5-dev`
- `build-essential`
- `kernel-package`
- `git-core`
- `gcc-4.8`
- `gcc-4.8-plugin-dev`
- `make`

Command Line (Execute the following commands)

```
sudo apt-get update sudo
```

```
apt-get upgrade sudo
```

```
apt-get dist-upgrade sudo
```

*apt-get install
libncurses5-dev build-
essential kernel-package
git-core gcc-4.8 gcc-
4.8-plugin-dev make*

After you are done with the software installations create the folder we will use for the build.

*mkdir build cd
build*

Step 1

Download the Linux kernel source package for the 4.4.0 version from <http://kernel.org> and get the corresponding grsecurity patches based on the kernel version

Command Line Steps (Execute the following commands) *wget*

https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.4.0.tar.xz *wget*

https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.4.0.tar.sign *wget*

http://sw.1h.com/grsecurity/grsecurity-3.0-3.2.61-201407232156.patch *wget*

http://sw.1h.com/grsecurity/grsecurity-3.0-3.2.61-201407232156.patch.sig *git*

clone git://kernel.ubuntu.com/ubuntu/ubuntu-trusty.git

You have now downloaded the Linux kernel source and the required grsecurity patches for it. In the next step, we will apply these patches and proceed to configuring the kernel

Step 2

Apply the downloaded grsecurity patches to the Linux kernel using the following commands.

Note: You should have the following files in your build folder from the previous step

grsecurity-3.0-3.2.61-201407232156.patch sender-gpg-key.asc

grsecurity-3.0-3.2.61-201407232156.patch.sig

ubuntu-package/linux-4.4.0.tar.sign

ubuntu-trusty/linux-4.4.0.tar.xz

Command Line Steps

```
cp ubuntu-trusty/debian/control-scripts/p* ubuntu-package/pkg/image/
```

```
cp ubuntu-trusty/debian/control-scripts/headers-postinst ubuntu-package/pkg/headers/
```

The above command copies the required directories from the Ubuntu kernel overlay directory to the correct *ubuntu-package* directory. We now proceed to the patching of the kernel.

Command Line Steps

```
tar -xf linux-4.4.0.tar cd
```

```
linux-4.4.0/
```

```
patch -p1 < ../grsecurity-3.0-3.2.61-201407232156.patch
```

Step 3 Configure the kernel using the “ncurses” interface with the following options. This process involves choosing the set of secure options that will make up the features of the resultant kernel output.

Command Line Steps

```
make menuconfig
```

The following screen would be presented

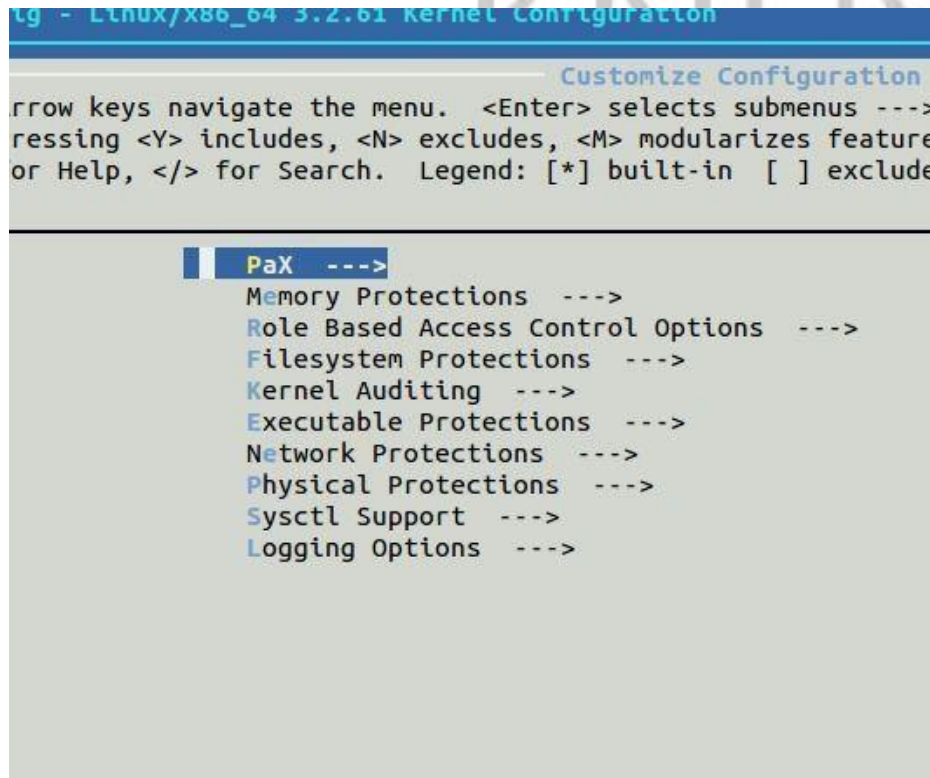


Figure 3.8 Kernel Configuration

Step 6

Start build with Debian package build options to order the build system to create installable “.deb” files after the process is complete

Command Line Steps *make-kpkg*
clean

sudo make-kpkg --initrd --overlay-dir=../ubuntu-package kernel_image kernel_headers **Step 7**

Install the generated “.deb” files to test them on the host computer

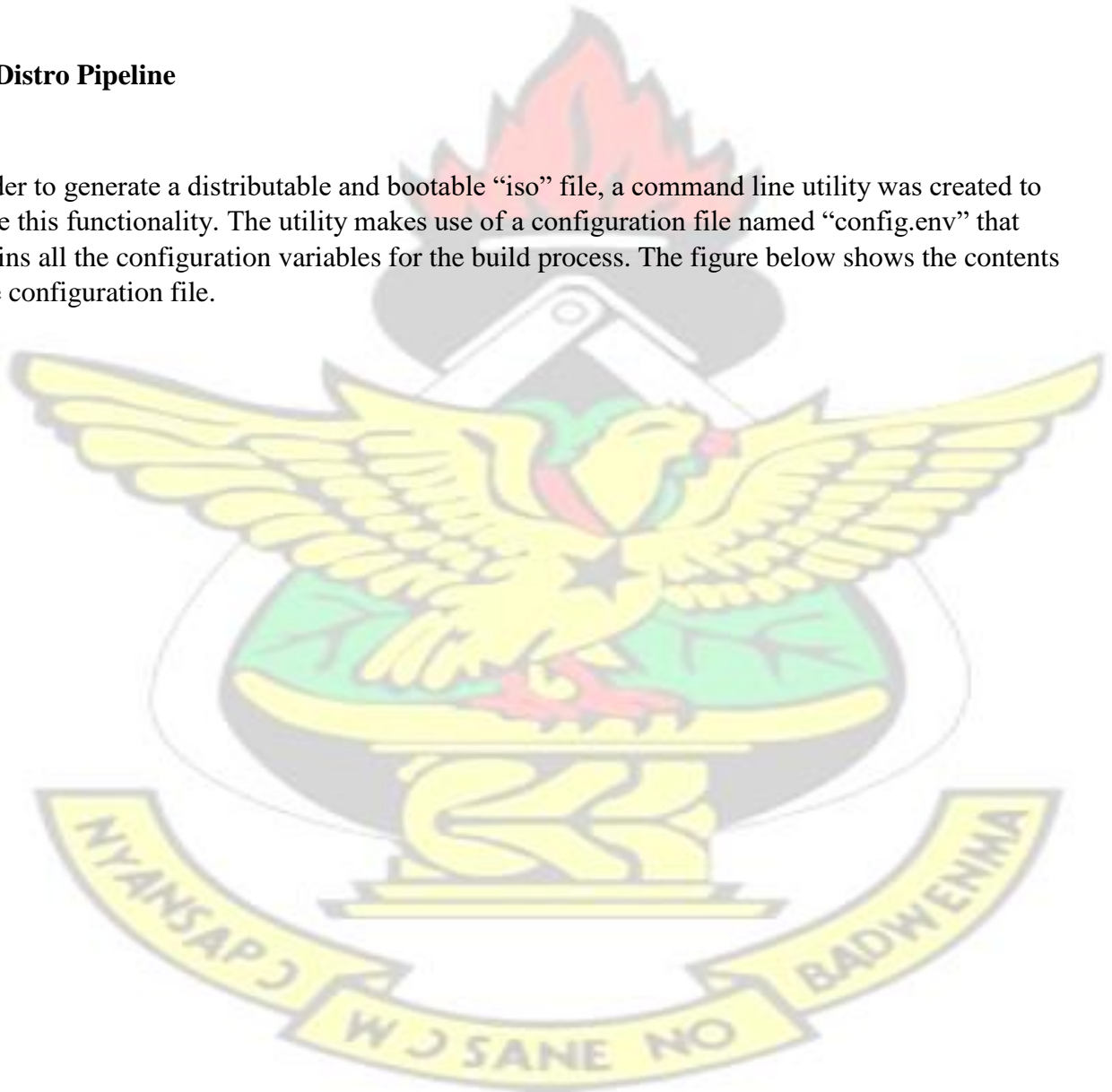
Command Line Steps *dpkg*

*-i linux-**

The command displayed above will install all the debian packages in the current directory so ensure you remain in the build process throughout this section.

ISO Distro Pipeline

In order to generate a distributable and bootable “iso” file, a command line utility was created to enable this functionality. The utility makes use of a configuration file named “config.env” that contains all the configuration variables for the build process. The figure below shows the contents of the configuration file.



```
config.env
1 # General Configurations
2 DISTRO_NAME="Trend-OS"
3 DISTRO_VERSION=1.0
4 DISTRO_AUTHOR="Engr. Danso Ansong"
5 DISTRO_BUILD_DATE="11-04-2016"
6 DISTRO_HOSTNAME=trend-server
7 DISTRO_CODENAME=pioneer
8 DISTRO_DESC="A Powerful Secure Linux Operating System"
9
10 # Important Directories
11 DIR_RES=resources
12 DIR_CONF=conf
13 DIR_EXT=extensions
14 DIR_WORK=work
15 DIR_EXTRAS=extras
16 DIR_LOGS=logs
17 DIR_LISTS=lists
18 DIR_OUTPUT=outputs
19 DIR_TMP_ROOT=$DIR_WORK/ROOT
20 DIR_TMP_ISO=$DIR_WORK/ISO
21 DIR_RES_BG=$DIR_RES/backgrounds
22 DIR_RES_AUDIO=$DIR_RES/backgrounds
23 DIR_RES_ICONS=$DIR_RES/icons
24 DIR_COPY_TMP_ROOT=/
25 DIR_COPY_TMP_ROOT_1=$DIR_WORK/bootstrap/
26
27 # Important Files
28 FILE_BOOT_SPLASH=$DIR_RES_BG/grub/splash.png
29 FILE_LOGS_STDOUT=$DIR_LOGS/stdout
30 FILE_LOGS_STDERR=$DIR_LOGS/stderr
31 FILE_EXCLUDES_LIST=$DIR_LISTS/excludes.list
32 FILE_RM_CONFIG_LIST=$DIR_LISTS/remove-config.list
33 FILE_RM_TMP_VAR_CONFIG_LIST=$DIR_LISTS/remove-tmp-var.list
34
```

Figure 3.9 Configuration environment

The utility goes through the following steps to generate the distributable “iso” file

Scaffold OS root folder structure

Copy template files from host into working directory

Remove Host specific configuration files

Remove Host specific cache and temporary files

Generate empty log files

Remove all users

Clean APT cache with chroot

Configure autologin for live user

Configure “casper”

Apply OS properties to the lsb_release file

Setup the Ubiquity installer for the live system

Prepare ISO tree

Copy initial ramdisk, memtest files and secure kernel into ISO tree

Apply compatibility patches to ISO tree

Squash the rootFS into a “filesystem.squashfs”

Generate MD5sums of all files in ISO

Generate ISO with “genisoimage”

To run the TrendOS utility execute the following command whilst in the project root folder

./install.sh

CHAPTER 4

CONVOLUTED KERNEL ARCHITECTURE

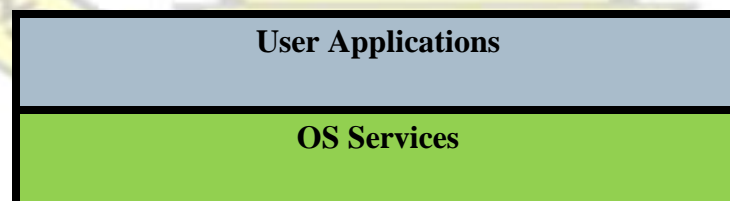
4.1 Overview of Kernel Architecture

An operating system (OS) kernel is the core of its architecture upon which all other modules or programming files (within the OS) are integrated. The kernel defines the architecture of the operating system and the hardware it supports. Over the past six decades, universities, research institutions, corporations and operating system engineers have all contributed to the development and expansion of Kernels. Since the late 1990's, there has been a paradigm shift in OS development from distributed environment to OS Security. This paper introduces a novel Kernel architecture,

dubbed the – Convolution Kernel – which is designed with the goal of contributing to the on-going research on operating system kernel security to protect the kernel against itself from vulnerabilities such as un-authorized kernel modification and privilege escalation. The scope of this research is tailored to mechanisms in developing a novel security architectural framework that could easily retrofit into monolithic kernels to further augment the already existing frameworks that falls under the Linux Security Module (LSM) to protect the kernel against itself and other applications.

The traditional OS architecture is generally made up of four major subsystems that work together to form a whole complete system which can be further classified into Kernel space and User space. The fundamental OS Architecture is made up of the hardware Controllers, which encompasses all the conceivable physical devices in the OS installation such as the CPU, memory module, network devices, Hard Drives among others. The next upper layer is the OS Kernel which serves as the integral part of the entire OS. In this layer, the kernel abstracts and mediate access to the hardware resources as captured in the previous layer which completes the kernel space (Engler et al, 1998).

The proceeding layers forms the user space section of the model. It is however made up of an interface level between the kernel space and the user space called the OS Services layer. This layer of the model essentially has two key sides. The lower part interfacing the kernel, which has compiler tools, libraries etc., and are considered part of the Kernel while the upper part interfacing the application, is considered part of the OS like the command shells etc. The top most layer of the architecture is referred to us the User Application which consists of set of applications executed by clients and servers (Bowman, 1998). Users are more familiar with this layer since their day to day interactions with the OS is interfaced with the various applications they install. The decomposition of an OS into four main subsystem architecture is as shown in figure 4.1.



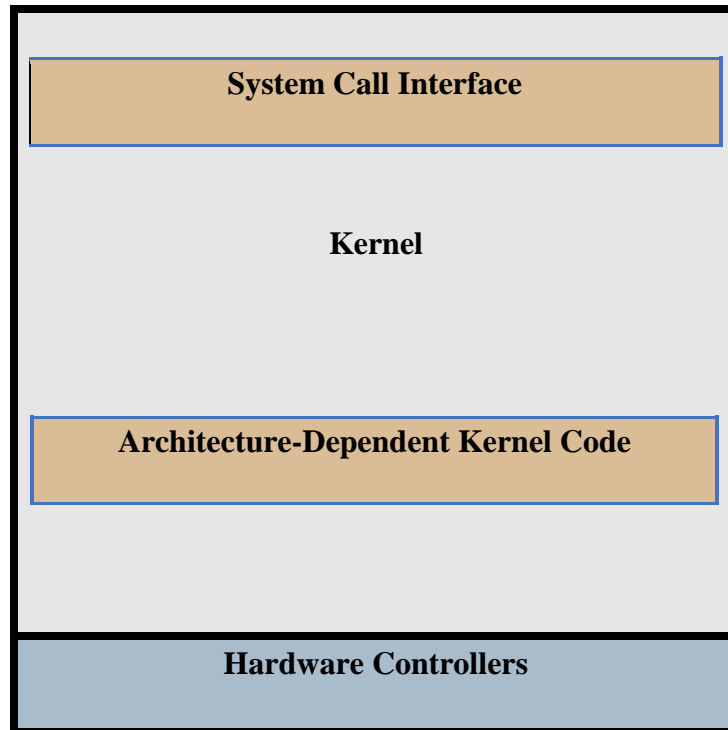


Figure. 4.1. – Breakdown of an Operating System into four major Subsystem.

OS kernel architecture is changing and expanding very fast to meet the ever increasing complexities of computer hardware designs and ever improving sophistication of software applications. The multicore hardware designs of processes has made it possible for a complex programs which hitherto could only run on huge, high-end and expensive servers, to currently run on low-end personal computers. These development have also been engineered by the numerous research by universities, corporations and computer engineers, to meet the growing need for secured yet fast kernel designs with minimal vulnerabilities.

However, core Linux architecture is monolithic by design (Reilly, 2002) and thereby, lack a resilient self-protection scheme when the security of the kernel space is breached (Dautenhahn et al, n.d.). Due to this characteristic feature, a single bit exploit in the kernel could lead to a fissure of the entire kernel mode of the OS including the MBR, memory module and other resource dependencies. Furthermore, vulnerabilities in most Linux based monolithic kernels have also made it predisposed to an array of kernel malware which exploits an internal kernel breach without any defense mechanism against internal attacks.

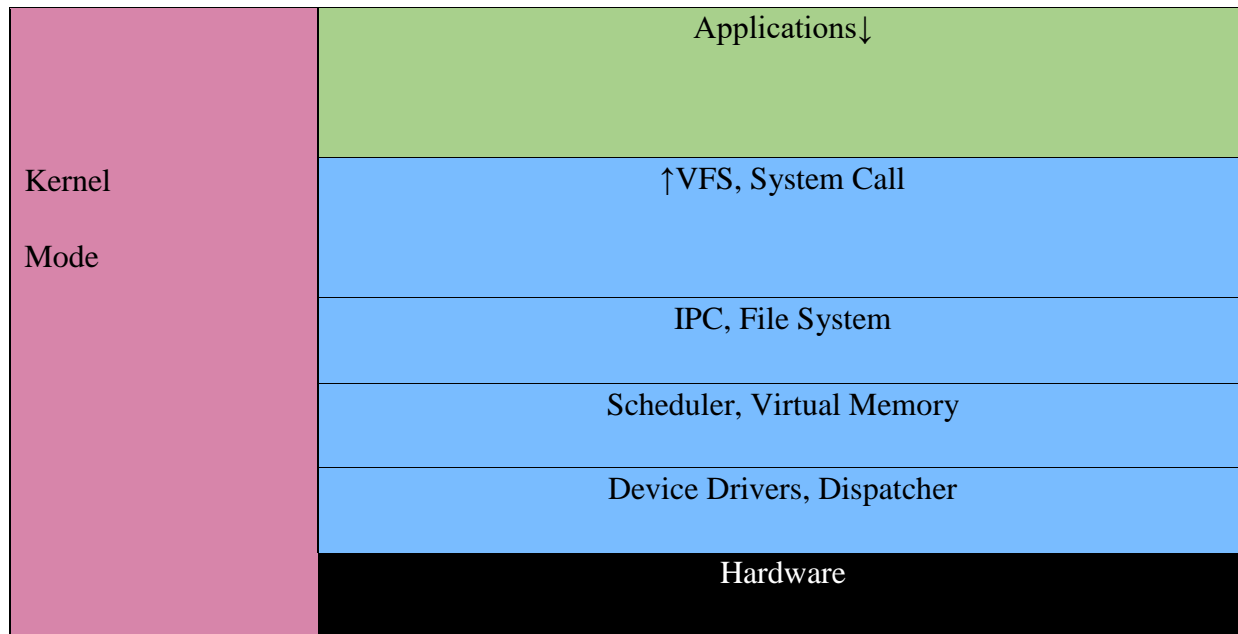


Figure 4.2. Monolithic Design

Even though copious developments have been made over the past decades to mitigate the drawback associated with the initial architectural design - the monolithic kernels – the evolvement of a principal alternative architecture to the traditional design became imperative. The Microkernel therefore became a perfect substitute to the monolithic as shown in Figure 4.2. The principal difference between the monolithic and the microkernel is that, in the former, every part of the kernel is executed in the unwieldy bottom-large kernel space which incidentally, happens to also run in the same address space. The key drawback therefore is a single process failure in any part of the kernel could have a grave consequence on the entire address space which frequently lead to a kernel panic (Chip et al., 2016).

In the microkernel however, unlike the monolith's huge kernel space, part of it is moved from the risky kernel space into a convenient and safer user space which is not susceptible to frequent crash of the kernel. This approach is considered less dangerous for the reason that, in the user space, each process runs in an isolated mode (aka servers) and therefore any bug in this design will obviously have a far less consequence since the processes involved may crash but the rest of the kernel will be operating in a safe mode.

Even though the microkernel appeared to have resolved the key challenges of the monolithic kernel, it also brought other limitations which are alien to the monolithic. Those weaknesses are code complexities which also leads to performance overheads (Holwerda, 2007).

Unlike the monolithic kernel's huge disproportionate kernel space with respect to the user space, the microkernel has the reverse forming its design. With this new design in the microkernel, it is therefore expedient to have effective communication of the various services which hitherto was located in the kernel space now situated in the user space. This service in the microkernel is referred to as the inter-process communication (Nteziryayo et al, 2015).

4.2 Linux Kernel Security Framework

With the increase in the number of software vulnerabilities recorded across the world, the global challenge by computer security scientists and engineers to develop various access control utilities to curb the increasing rate of software vulnerabilities also led to the proliferation of numerous security software. Even though this was considered as a breakthrough to improve commodity operating system security on most especially Linux and Unix OS, there is however the lack of coordination on the best framework and direction to follow to allow vendors to choose from their own flavors. This therefore created a lack of single standardized Application Programming Interface (API) at a time (Wright, et al, 2002).

With the wide acceptance of the need to develop a standard API for policy enforcement module by Linus Torvalds therefore had to resort to a policy framework to encapsulate all security modules to be able to function under a single API. The Linux Security Module (LSM) framework was therefore implemented to encapsulate the divers' access control modules to enhance the security of the operating system. The LSM framework supports the loading of secured kernel modules. Some of these enhanced security include POSIX.1e capabilities, SELinux, Domain and Type Enforcement (DTE) and Linux Intrusion System (LIDS) (Wright et al., 2002), Rule Set Based Access Control (RSBAC), GRsecurity, Trusted Computer System Evaluation (TCPE) among others (Biondi, 2003).

4.3 Drawback of Linux Security Module

With the adoption of LSM as a standard API for loadable access control modules for the kernel to enhance the security of the architecture, there were however some challenges associated with the implementation of the framework. While some engineers were considering the use of an integrated kernel structure, the founder of Linux and top maintenance group rejected such idea. Such kernel implementation were considered to be inflexible and uncompromising and as a result, some earlier development modules which could not adopt to the framework such as GRsecurity and RSBAC were obliterated from the list of standardized loadable modules because of their un-support for LSM API (Wright et al., n.d.).

This therefore denies an otherwise potentially best approach to enhancing security. With this reason, it also denies majority of users the opportunity to experiment and select from their own best kernel security module. Reason for the deprecation of others could be attributed to inactivity and excessive inflexibility in allowing the modules to easily port and other compatibility issues (Karlsson, 2010).

Even though modules such as SELinux, AppArmor, SMACK, TOMOYO and YAMA are among others are highly rated, the first two appears in some distributions as the default kernel security module precompiled, they also have their own vulnerabilities that keeps operating system developers a bit apprehensive on the best option to choose from as far as kernel security modules were concerned. The key challenge arising out of the use of Linux Security Modules is the capability to disable and enable the module as and when it becomes required (Backes et al, 2013, Merrill, 2012).

4.4 The Convoluted Kernel Architecture

Due to various concerns raised on the adoption of a single framework (Linux Security Module) to interface kernel security modules, the need for the development of an integrated kernel framework to provide enhanced security and resilience became indispensable. Even though some efforts have been made by operating system engineers and scientists such as capsicum and Secure Virtual Architecture (SVA) which are the two widely pronounced kernel architectural framework,

however, their emphasis do not involve amalgamation of other useful features of kernels such as High Availability and cryptography using zero knowledge (Mauerer, 2008).

In the absence of a kernel which could harness these functionalities to support server environment, the idea of an integrated kernel architecture with sandbox-virtualization (Yu & Science, 2007) and its prime focus on High Availability and secured authentication scheme was conceived. This kernel with these framework was referred to as the Convolute Kernel architecture.

Beginning from the bottom "Kernel mode" division (as shown in Figure 4.3) the Linux Kernel Layer contains the various subsystems that make up the kernel namely, the IO Manager, the Device Drivers, the Process manager, Virtual Memory Manager and more (Stallings & Hall, 2008).

Above this layer is the TrendOS Linux Security Module (TLSM) which aims at enabling the efficient and concurrent use of multiple LSMs in a well-coordinated manner. This module is included as a built-in module in the TrendOS Linux Kernel. TLSM enables capabilities that allow the coexistence of multiple LSMs (e.g. SELinux, AppArmor etc.) which greatly enhances system wide security (Gorman, 2003).

Above this layer is the system call interface where standard system call function (e.g. exec) are evaluated and handled. This layer denotes the beginning of the Kernel Mode Division. Moving upwards, the User mode Division also known as Secure Trust Level 1 (STL1). This layer begins with System binaries the libraries that eventually make contact with the System call interface (Bridge). These are stored programs and procedures that users and applications constantly make heavy use of (Mauerer, n.d.).

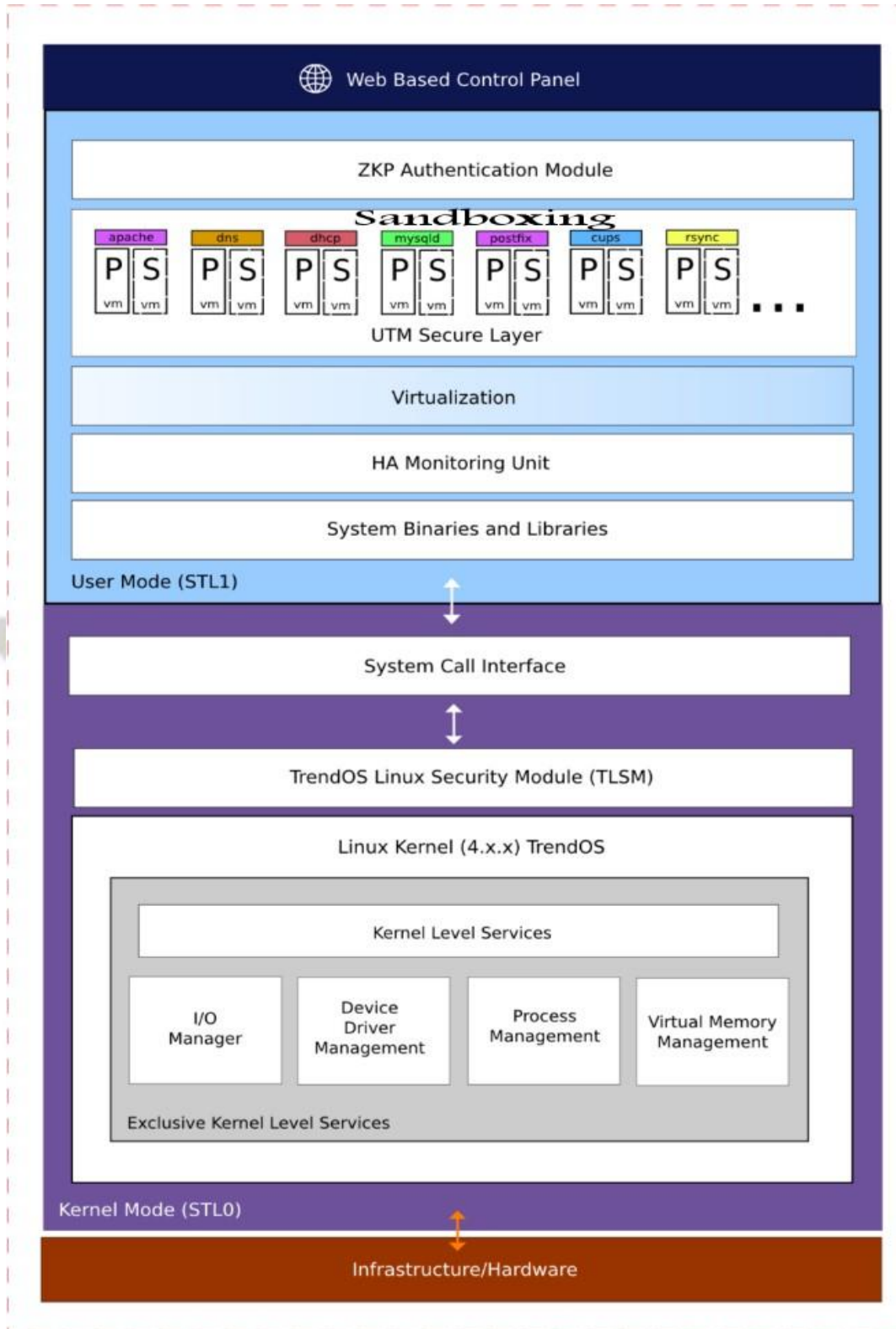


Figure 4.3. Convoluted Kernel Architectural Design

Above this Layer lies the High Availability Monitoring Unit as shown in Figure 4.4, which lies as a backbone to the Sandbox Environment. This layer is responsible for ensuring the all sandboxes are available 99.9% of the time. The services and operation of the High Availability technology ensures that continuous service is available between the private and secondary services at all times.

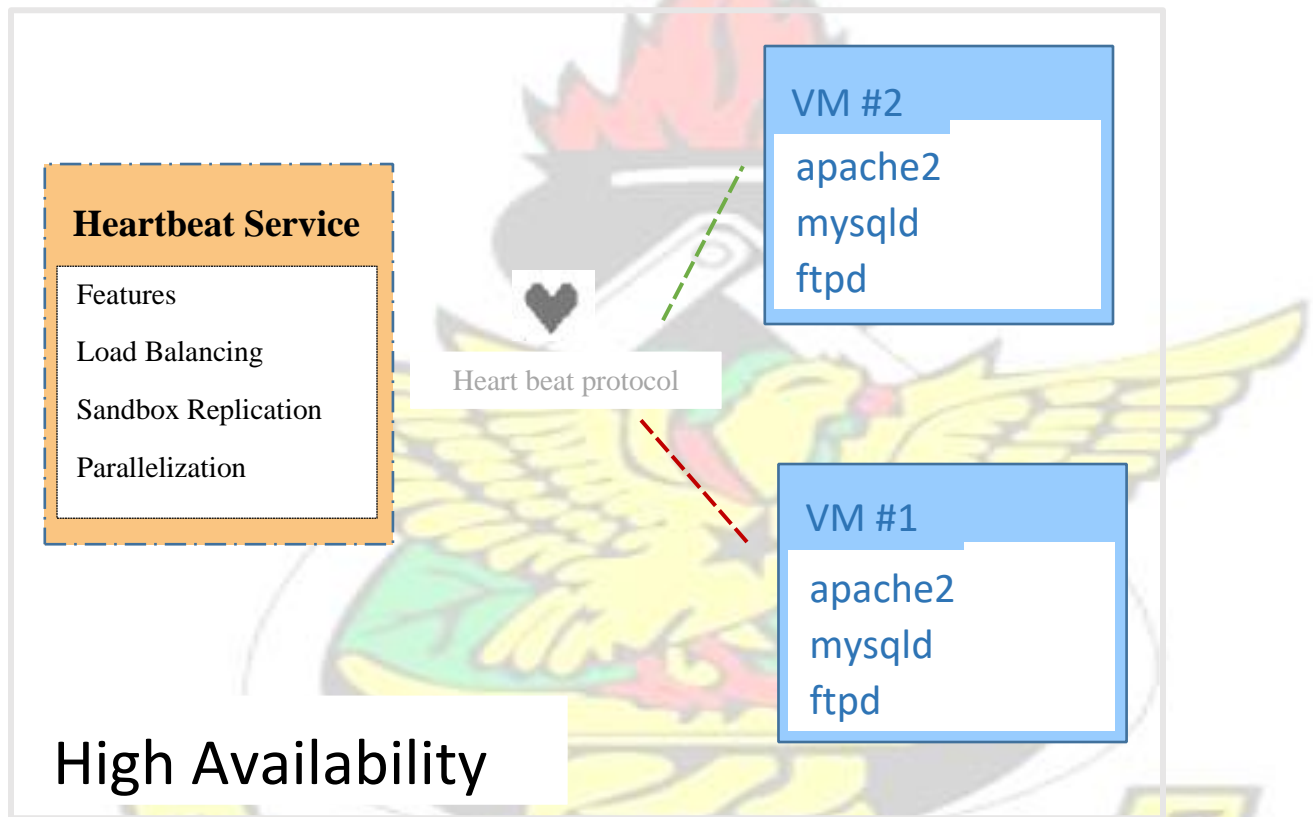


Figure 4.4. Heartbeat architecture of the of the Convolute Architecture

Techniques including the Heart Beat mechanism are utilized efficiently to ensure downtime and recovery time is greatly reduced (Shahapure, 2015).

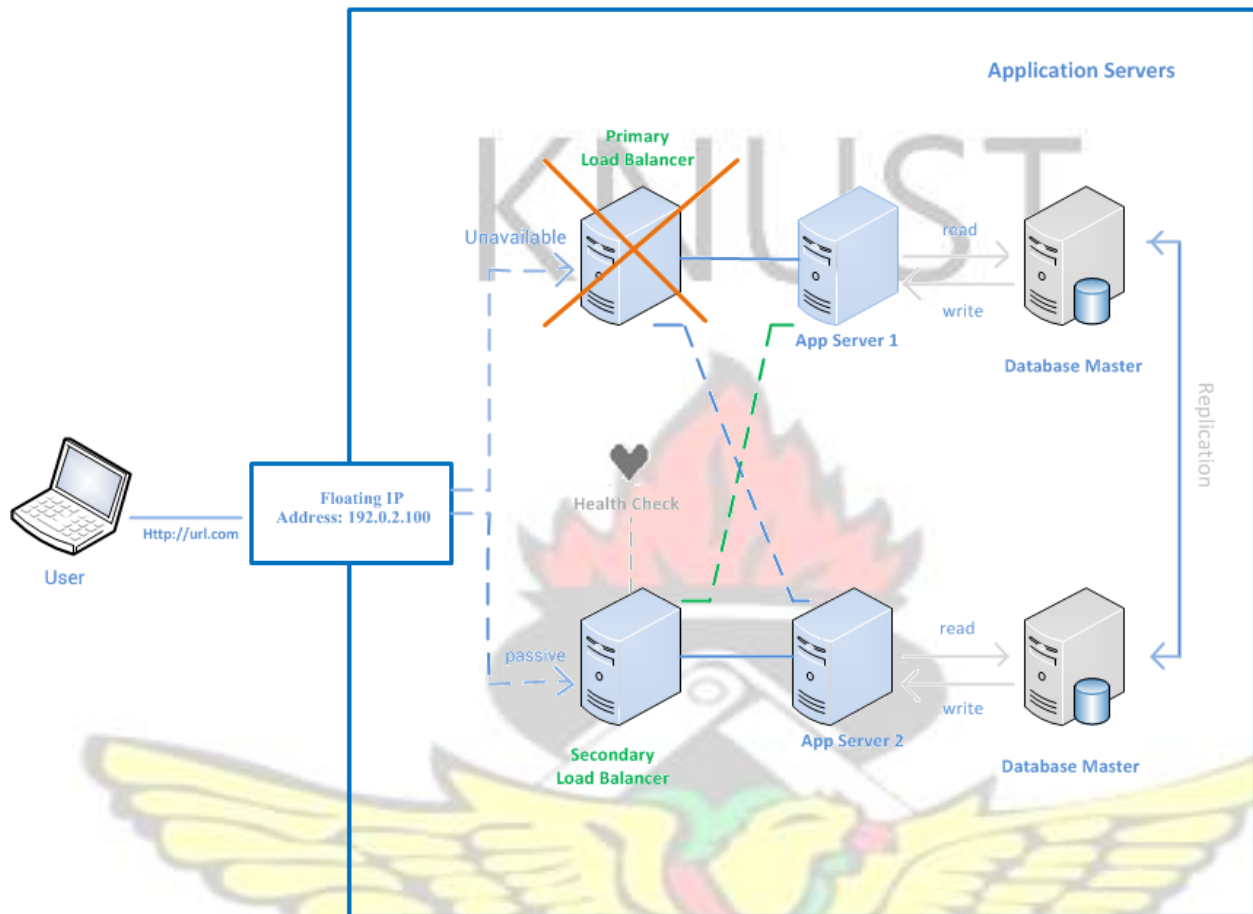


Figure. 4.5. Simplified High Availability (HA) Source (Manual et al., n.d.).

Next above the HA Monitoring Unit is the Sandbox Environment where Application services run in a safe isolated environment that is actively protected by UTM enabled system-default sandboxes. This Layer allows the creation of as many application sandboxes as desired that will run in a Secure Sandbox environment that is protected by system-default UTM enabled sandboxes using state-of-the-art techniques such as Content Inspection as shown in Figure 4.5 (Manual et al., n.d.).

Figure 4.6 shows the detailed design of the Secure UTM Layer of the main Architecture. The expanded design comprises of several Unified Threat Management (UTM) modules that make up

the UTM stack. This stack consists of security applications for Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) which Snort was adopted as well as an IPFirewall (ipfw).

What happens is that when network traffic arrives from the internet to or from the virtual machines. The UTM layer filters all traffic to ensure all network traffic is safe and provides a high level of protection to the sandboxes. It does so through a chaining mechanism that allows packets to be filtered thoroughly before they are finally routed to their final destinations.

The UTM stack has been built in such a way that, it is able to co-locate with third party security tools without any conflict. This UTM stack works transparently with the virtual machines and sandbox technique.

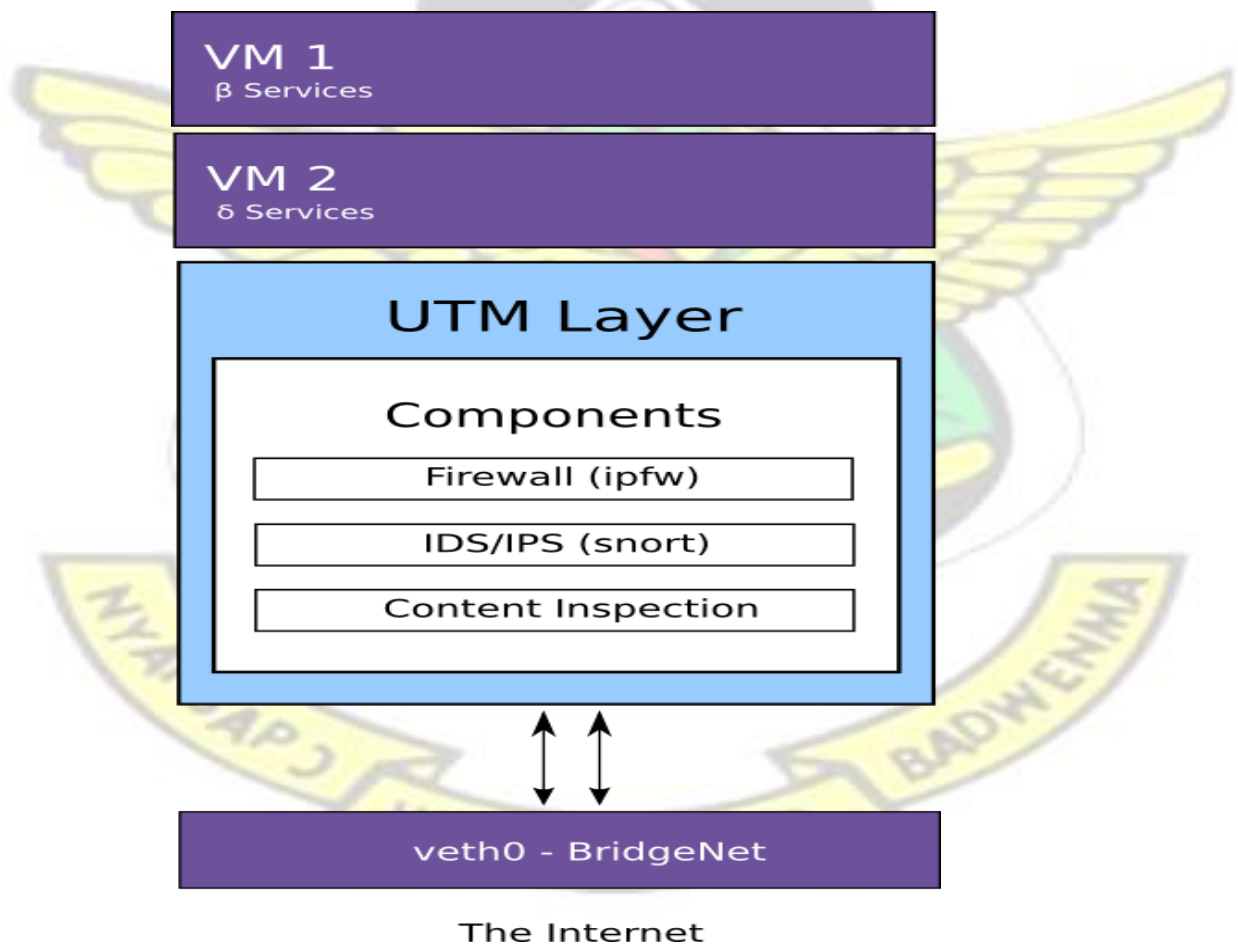


Fig. 4.6 Expanded Unified Threat Management

Above these levels is the ZKP Security module. This layer is essentially a Pluggable Authentication Module (PAM) responsible for System-wide authentication using the ZeroKnowledge Authentication technique. Using such a transparent framework, the all great benefits of using the ZKP technique can be realized seamlessly through PAM-aware system-based application like "ssh" (Soares, 2013).

The Last Layer that ends the STL1 is the Web Based Control Panel. This layer presents a Web GUI that provides a general overview of system performance and enables system administrators to regulate TrendOS system functionality in as simple a manner as just turning knobs and setting values. This Control panel is the official TrendOS dashboard that Trend System Administrators will be familiar with.

4.5 Security Enhanced Framework

There has been several security framework that over the years have been developed to secure the Linux kernel and to improve the security of the architecture in general. Several kernel and operating system developers are beginning to adopt the use of virtualization to protect the core kernel structure from unauthorized manipulation from illegitimate users to expose its vulnerabilities (Watson et al., 2014).

OS-level integrated level virtualization technique was therefore implemented into the framework in order to enhance the security of the core kernel to improve the resilience of the architecture (Shan, 2011).

The diagram presented in Figure 4.7 shows an expansion of the virtualization component of TrendOS System architecture which is the prototype of the Convolute Kernel Architecture. Starting from the bottom, we installed Linux Containers (LXC) as the underlying technology behind the virtualization component. LXC (which is an abbreviated way of saying Linux Containers) is an operating system-level virtualization method for running multiple isolated Linux

systems which are called containers on a single control host. This creates a high performance environment for the VMS (sometimes referred to as containers).

Above this layer exists the virtual machines or containers which essentially achieves virtualization at the OS Level. This is achieved through Linux cgroups and is beyond the scope of our discussion (Katzner, 2015).

As established earlier, the virtual machines share the host's kernel facilities. Above that is Bridge networking rules configured directly into every VM. This allows packets flowing from and to these containers to be analyzed by our UTM layer to protect application services from possible attack (Pa, n.d.).

Above this layer lies system libraries that make constant repetitive use during their execution. On top of this lies the actual application services that clients make use of. This is the ultimate goal of a secure server environment - To protect its application services (e.g. Apache) (Assurance & Report, 2011).

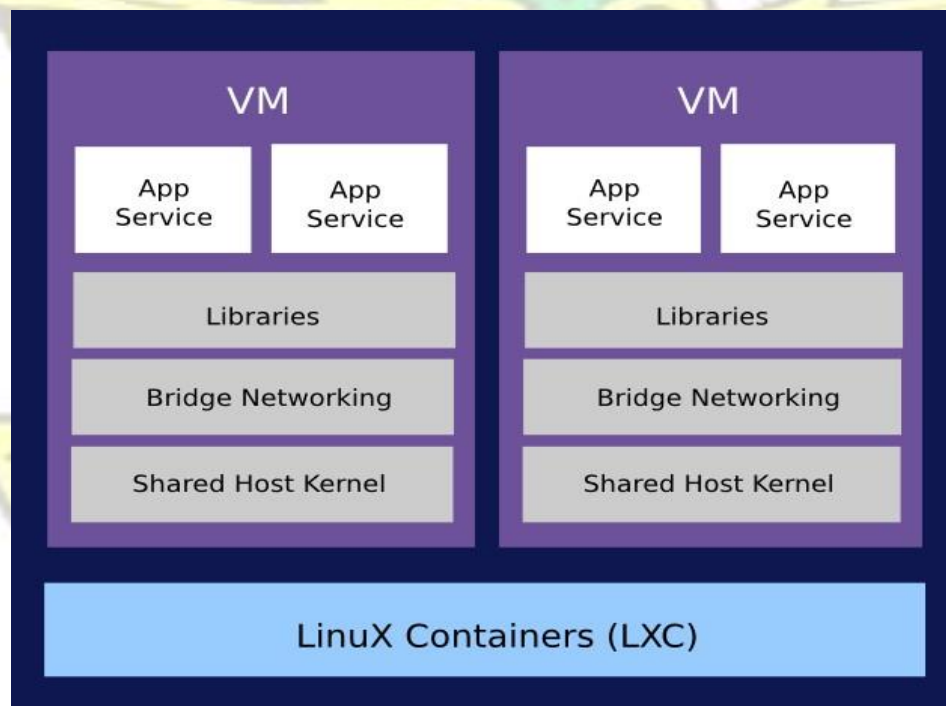


Fig. 4.7 The LXC design in the Architecture

4.6 Overall Privilege Separation of the Design

The architecture in its implementation is divided into three levels of rings. It begins with the highest privilege level ring 0, which is the core kernel which directly communicates with the hardware resources including the processor. The next higher level is the ring 1 which is the modified kernel of the architecture being implemented (Convuluted Kernel Architecture) to be able to execute the services integrated into the framework for smooth execution. It also contains the Stealth-Obfuscation Zero Knowledge Authentication which is integrated into the Portable Authentication Module of the Kernel to conceal any attempt by an intruder to intercept client server authentication. It also maintains Control Groups (CGs) among the various sandboxes and virtual machines to ensure a synchronized update of transactions between the various services. It implements both inter-process and intra-process communication using the heart-beat protocol (Bittau, 2009). A diagram of which as shown in figure 4.8.

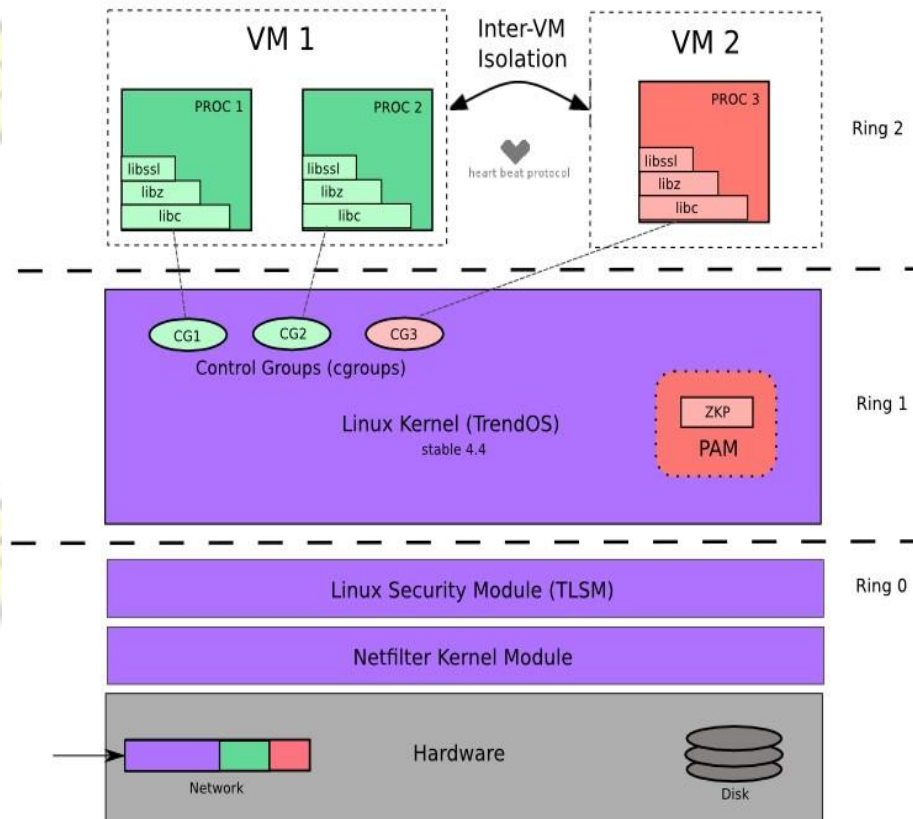


Fig. 4.8 Privilege Separation of the architecture

Figure 4.8 shows the High Availability setup which is an expansion of the High Availability Layer of the TrendOS architecture (O'Neil et al, 2013).

Installed is a Heartbeat daemon that enables the virtual machines to know about the presence of peer virtual machines. This enables the Heartbeat protocol to immediately determine whether a VM is unavailable. This triggers failover mechanisms such as VM Replication. The Heartbeat services also performs load balancing (Yang et al, 2011).

4.7. The implementation of the Convolutd Kernel Architecture – Trend-OS

The development of the Trend-OS operating system is based on the Convolutd Kernel Architecture which combines multiple user instances of virtual private servers with Stealth Obfuscation Zero Knowledge Proof for all its application level security authentication in place of the inbuilt authentication module in Linux PAM. It implements redundant layers of security in its kernel ostensibly to protect the kernel against itself in case it is attacked. The redundant security policy of this architecture ensures that all applications installed on this operating system runs on a virtual environment to abstract the Kernel's inner core from malicious or threats of the applications that runs on them with an additional kernel layer in a monolithic architecture.

The architecture also makes wide use of the GRsecurity kernel enhancements. Grsecurity is an extensive security enhancement to the Linux kernel that defends against a wide range of security threats. The PaX project is included, hardening both user space applications and the kernel against memory corruption-based exploits. Grsecurity includes a powerful Mandatory Access Control system with an effortless automatic learning mode and a host of other miscellaneous hardening features.

4.8. Adoption of Chroot Hardening Technique for compatibility

The Convolutd Kernel Architecture also adopted the Chroot Hardening technique in its design. Chroot is a common isolation mechanism for services. The technique adopted in the Convolutd

Kernel with the Grsecurity includes features for eliminating many common escape routes from chroots and can lock them down to the point where the confinement is equivalent to a container. These features can all be toggled on and off via sysctl switches. The following features are enabled in /etc/sysctl.d/05-grsecurity.conf file by default and are unlikely to cause any compatibility issues as shown in Listing 4.1.

Listing 4.1. grsecurity modules

```
kernel.grsecurity.chroot_deny_fchdir      = 1 kernel.grsecurity.chroot_deny_shmat      = 1
kernel.grsecurity.chroot_deny_sysctl      = 1 kernel.grsecurity.chroot_deny_unix      = 1
kernel.grsecurity.chroot_enforce_chdir    = 1 kernel.grsecurity.chroot_findtask    = 1
```

The remaining features are left off by default, to remain compatible with containers as shown in Listing 4.2.

Listing 4.2. kernel integration with grsecurity.

```
#kernel.grsecurity.chroot_caps            = 1
#kernel.grsecurity.chroot_deny_chmod      = 1
#kernel.grsecurity.chroot_deny_chroot     = 1
#kernel.grsecurity.chroot_deny_mknod      = 1
#kernel.grsecurity.chroot_deny_mount      = 1
#kernel.grsecurity.chroot_deny_pivot      = 1
#kernel.grsecurity.chroot_restrict_nice    = 1
```

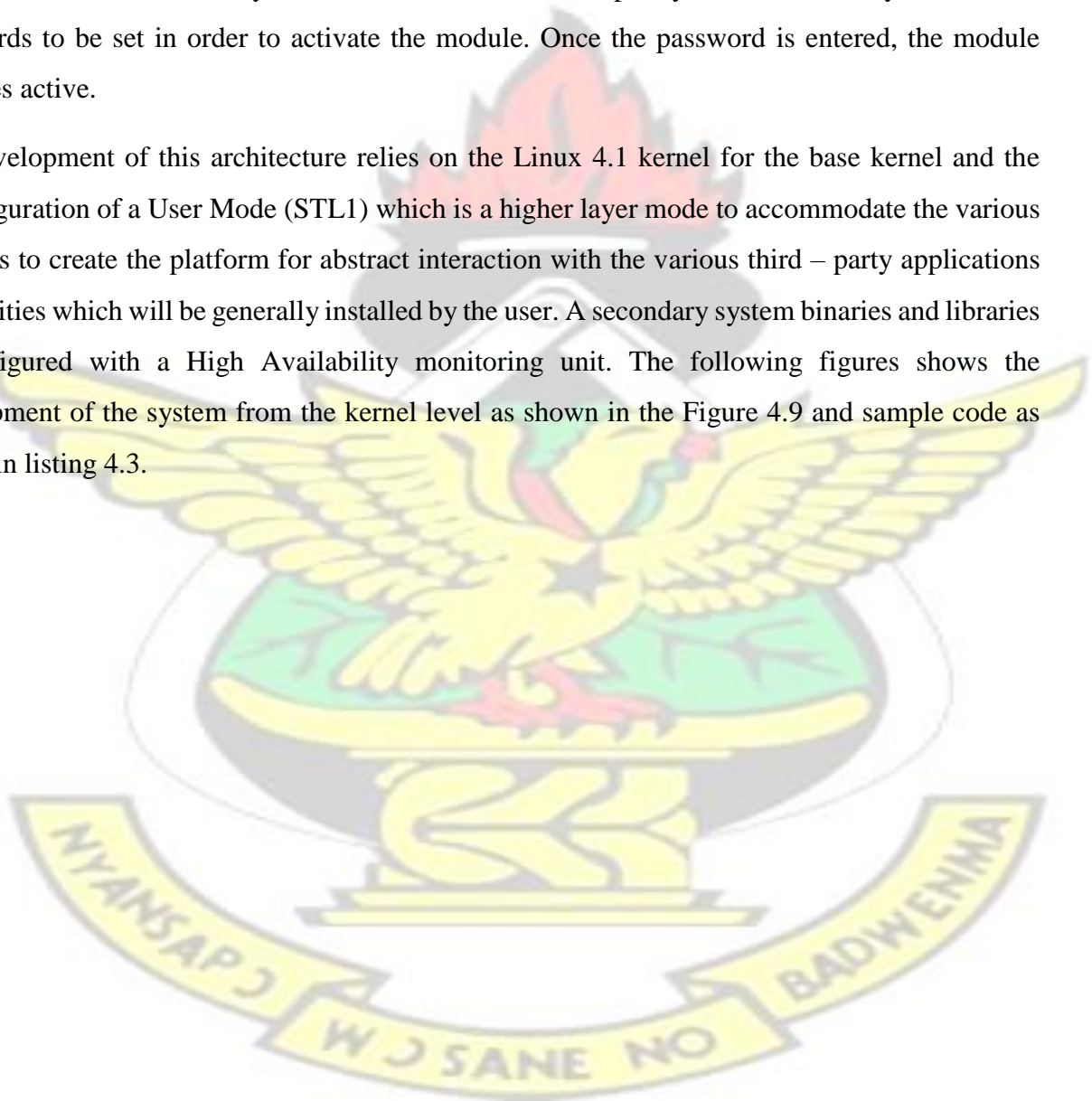
4.9. Insertion of Socket restrictions

In order to ensure that applications that exploits the vulnerabilities associated with remote procedure invocation does not manipulate the kernel, it utilizes three level of restricted access to sockets. Users in the socket-deny-client group are forbidden from connecting to other hosts. Users in the socket-deny-server group are unable to listen on a port. The socket-deny-all group includes both of the restrictions. The virtual private service of the architecture requires a utility to easily manage the system to perform all administrative tasks. The gradm tool allows you to administer

and maintain a policy for your system. With it, you can enable or disable the RBAC system, reload the RBAC roles, change your role, set a password for admin mode, and other privileges only allowed on servers.

The integration of the gradm in the CKA architecture has a default policy which is installed in `/etc/grsec/policy`. By default, the RBAC policies are not activated. It is the system administrator's job to determine when the system should have an RBAC policy enforced. The system allows passwords to be set in order to activate the module. Once the password is entered, the module becomes active.

The development of this architecture relies on the Linux 4.1 kernel for the base kernel and the reconfiguration of a User Mode (STL1) which is a higher layer mode to accommodate the various modules to create the platform for abstract interaction with the various third – party applications and utilities which will be generally installed by the user. A secondary system binaries and libraries is configured with a High Availability monitoring unit. The following figures shows the development of the system from the kernel level as shown in the Figure 4.9 and sample code as shown in listing 4.3.



```
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

Trend-OS 2.0 Installer #11-04-2016
Author: Engr. Danso Ansong

visit:
www.fredancybersecurity.com

This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. For details, see:
http://www.opensource.org/licenses/mit-license.php

Target architecture: x86_64

[16-11-20/16:54 UTC] Peforming Cleanup ...
[16-11-20/16:54 UTC] Cleaning work/ROOT
[16-11-20/16:54 UTC] Empty folder
[16-11-20/16:54 UTC] Cleaning ISO Directory
[16-11-20/16:54 UTC] Done - Cleaning
[16-11-20/16:54 UTC] [Installer Started]!
[16-11-20/16:54 UTC] Invocation Directory: /home/fredan/TrendOS/trend-os
[16-11-20/16:54 UTC] Building ISO for Trend-OS v2.0
[16-11-20/16:54 UTC] Logging [stdout] to /home/fredan/TrendOS/trend-os/
[16-11-20/16:54 UTC] Logging [stderr] to /home/fredan/TrendOS/trend-os/
[16-11-20/16:54 UTC]
[16-11-20/16:54 UTC] =====
[16-11-20/16:54 UTC] A. [ Preparing Build Stage ]
[16-11-20/16:54 UTC] =====
[16-11-20/16:54 UTC] Successfully Checked - 'dpkg' - [ONLINE]
[16-11-20/16:54 UTC] Checking Installed Binaries
[16-11-20/16:54 UTC] Checking for pv
[16-11-20/16:54 UTC] pv - [ALREADY INSTALLED]
[16-11-20/16:54 UTC] Checking for xfsprogs
[16-11-20/16:54 UTC] xfsprogs - [ALREADY INSTALLED]
```

Figure 4.9 Kernel deployment into the architecture

Listing 4.3 General Configuration

```
# General Configurations
DISTRO_NAME="Trend-OS"
DISTRO_VERSION=1.0
DISTRO_AUTHOR="Engr. Danso Ansong"
DISTRO_BUILD_DATE="11-04-2016"
DISTRO_HOSTNAME="nothing"
DISTRO_CODENAME=pioneer
DISTRO_DESC="Powerful and Secure Linux OS"

# Important Directories
DIR_RES=resources
DIR_CONF=conf
DIR_EXT=extensions
DIR_WORK=work
DIR_EXTRAS=extras
DIR_LOGS=logs
DIR_LISTS=lists
DIR_OUTPUT=outputs
DIR_TMP_ROOT=$DIR_WORK/ROOT
DIR_TMP_ISO=$DIR_WORK/ISO
DIR_RES_BG=$DIR_RES/backgrounds
DIR_RES_AUDIO=$DIR_RES/backgrounds
DIR_RES_ICONS=$DIR_RES/icons
DIR_COPY_TMP_ROOT=/
DIR_COPY_TMP_ROOT_1=$DIR_WORK/bootstrap/

# Important Files
FILE_BOOT_SPLASH=$DIR_RES_BG/grub/splash.png
FILE_LOGS_STDOUT=$DIR_LOGS/stdout
FILE_LOGS_STDERR=$DIR_LOGS/stderr
FILE_INSTALL_LIST=$DIR_LISTS/requirements.list
FILE_EXCLUDES_LIST=$DIR_LISTS/excludes.list
FILE_RM_CONFIG_LIST=$DIR_LISTS/remove-config.list
FILE_RM_TMP_VAR_CONFIG_LIST=$DIR_LISTS/remove-tmp-var.list

# Script Configuration
EXT_LOAD="helper.sh"

# File Configurations
TRENDOS_DETECT_FILE=.trendos

# ISO Linux Configuration
ISOLINUX_TIMEOUT=100
LIVE_CD_LABEL="$DISTRO_NAME Live CD"
LIVE_CD_URL="www.fredancybersecurity.com"
SQUASHFSOPTS="-no-recovery -always-use-fragments -b 1M -no-duplicates"

# ISO Configuration
ISO_FORMAT=squashfs
ISO_FILENAME="trend-os-1.0-desktop-amd64.iso"
```

ISO_SFSCOMPRESSION=gzip

ISO_MENU_TIMEOUT=10

69

KNUST



The modification of the script and deployment of the kernel into the architecture continues with the loading of the various modules as shown in Figure 4.10.

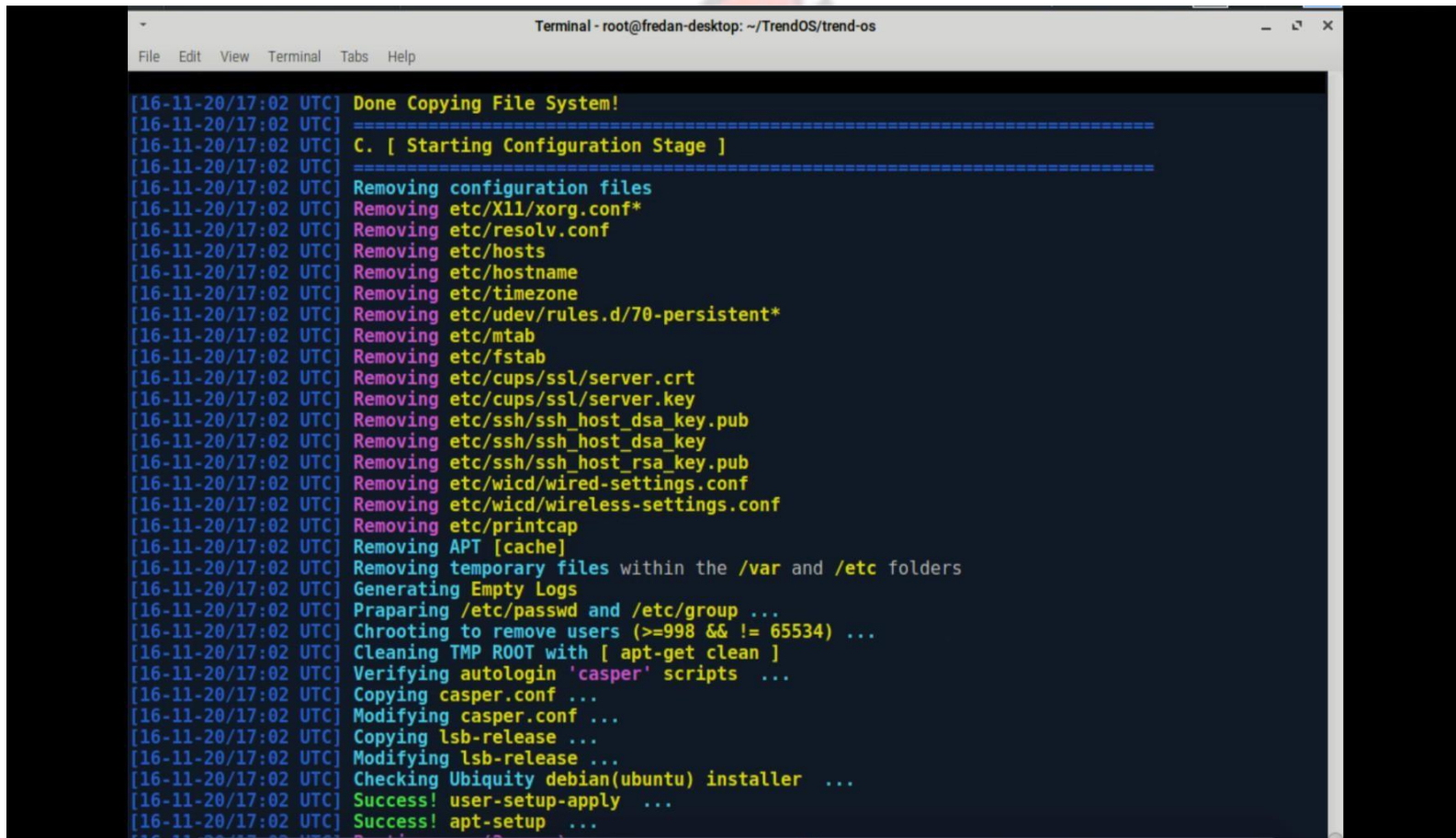
```

Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/mt6397-regulator.ko
18,662 100% 216.96kB/s 0:00:00 (xfr#6001, ir-chk=1041/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/palmas-regulator.ko
33,542 100% 389.95kB/s 0:00:00 (xfr#6002, ir-chk=1040/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pcap-regulator.ko
15,854 100% 173.96kB/s 0:00:00 (xfr#6003, ir-chk=1039/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pcf50633-regulator.ko
10,902 100% 119.62kB/s 0:00:00 (xfr#6004, ir-chk=1038/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pfuzel100-regulator.ko
27,094 100% 287.60kB/s 0:00:00 (xfr#6005, ir-chk=1037/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pwm-regulator.ko
8,614 100% 91.44kB/s 0:00:00 (xfr#6006, ir-chk=1036/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/qcom_spmi-regulator.ko
14,078 100% 149.44kB/s 0:00:00 (xfr#6007, ir-chk=1035/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rc5t583-regulator.ko
13,678 100% 143.63kB/s 0:00:00 (xfr#6008, ir-chk=1034/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rn5t618-regulator.ko
10,574 100% 111.03kB/s 0:00:00 (xfr#6009, ir-chk=1033/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rt5033-regulator.ko
9,598 100% 100.79kB/s 0:00:00 (xfr#6010, ir-chk=1032/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s2mpa01.ko
21,838 100% 229.31kB/s 0:00:00 (xfr#6011, ir-chk=1031/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s2mps11.ko
61,190 100% 635.70kB/s 0:00:00 (xfr#6012, ir-chk=1030/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s5m8767.ko
30,166 100% 313.39kB/s 0:00:00 (xfr#6013, ir-chk=1029/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/sky81452-regulator.ko
8,366 100% 82.52kB/s 0:00:00 (xfr#6014, ir-chk=1028/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps51632-regulator.ko
11,190 100% 98.45kB/s 0:00:00 (xfr#6015, ir-chk=1027/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps6105x-regulator.ko
9,510 100% 43.81kB/s 0:00:00 (xfr#6016, ir-chk=1026/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps62360-regulator.ko
14,374 100% 41.53kB/s 0:00:00 (xfr#6017, ir-chk=1025/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps65023-regulator.ko
17,246 100% 30.35kB/s 0:00:00 (xfr#6018, ir-chk=1024/9264)

```

Figure 4.10. Loading of the various security and driver modules into the kernel.

The generic monolithic Linux kernel lacks several modules that can meet the design of the new architecture. Therefore, several modules were removed while others modified to ensure that the novel CKA's design is replicated in the new architecture. Figure 4.11 shows part of the result during the building of the new kernel.

A terminal window titled "Terminal - root@fredan-desktop: ~/TrendOS/trend-os" displays the output of a configuration script. The script is in the "Starting Configuration Stage" and lists various files being removed from the system, including configuration files like /etc/X11/xorg.conf, /etc/resolv.conf, /etc/hosts, /etc/hostname, /etc/timezone, and /etc/udev/rules.d/70-persistent*. It also removes temporary files in /var and /etc, generates empty logs, prepares /etc/passwd and /etc/group, chroots to remove users, cleans the tmp root, verifies autologin scripts, copies and modifies casper.conf and lsb-release, checks the Ubiquity debian installer, and finally applies user-setup and apt-setup. The terminal output is as follows:

```
[16-11-20/17:02 UTC] Done Copying File System!
[16-11-20/17:02 UTC] =====
[16-11-20/17:02 UTC] C. [ Starting Configuration Stage ]
[16-11-20/17:02 UTC] =====
[16-11-20/17:02 UTC] Removing configuration files
[16-11-20/17:02 UTC] Removing etc/X11/xorg.conf*
[16-11-20/17:02 UTC] Removing etc/resolv.conf
[16-11-20/17:02 UTC] Removing etc/hosts
[16-11-20/17:02 UTC] Removing etc/hostname
[16-11-20/17:02 UTC] Removing etc/timezone
[16-11-20/17:02 UTC] Removing etc/udev/rules.d/70-persistent*
[16-11-20/17:02 UTC] Removing etc/mtab
[16-11-20/17:02 UTC] Removing etc/fstab
[16-11-20/17:02 UTC] Removing etc/cups/ssl/server.crt
[16-11-20/17:02 UTC] Removing etc/cups/ssl/server.key
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_dsa_key.pub
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_dsa_key
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_rsa_key.pub
[16-11-20/17:02 UTC] Removing etc/wicd/wired-settings.conf
[16-11-20/17:02 UTC] Removing etc/wicd/wireless-settings.conf
[16-11-20/17:02 UTC] Removing etc/printcap
[16-11-20/17:02 UTC] Removing APT [cache]
[16-11-20/17:02 UTC] Removing temporary files within the /var and /etc folders
[16-11-20/17:02 UTC] Generating Empty Logs
[16-11-20/17:02 UTC] Prparing /etc/passwd and /etc/group ...
[16-11-20/17:02 UTC] Chrooting to remove users (>=998 && != 65534) ...
[16-11-20/17:02 UTC] Cleaning TMP ROOT with [ apt-get clean ]
[16-11-20/17:02 UTC] Verifying autologin 'casper' scripts ...
[16-11-20/17:02 UTC] Copying casper.conf ...
[16-11-20/17:02 UTC] Modifying casper.conf ...
[16-11-20/17:02 UTC] Copying lsb-release ...
[16-11-20/17:02 UTC] Modifying lsb-release ...
[16-11-20/17:02 UTC] Checking Ubiquity debian(ubuntu) installer ...
[16-11-20/17:02 UTC] Success! user-setup-apply ...
[16-11-20/17:02 UTC] Success! apt-setup ...
```


Figure 4.11 modifying the generic kernel to suit the convoluted kernel Architecture

Because the architecture required several security enhancements in its design, several multi-level security enhancements is built into the architecture in the development stage in order to meet the objectives of the system as shown in Figure 4.12.

```

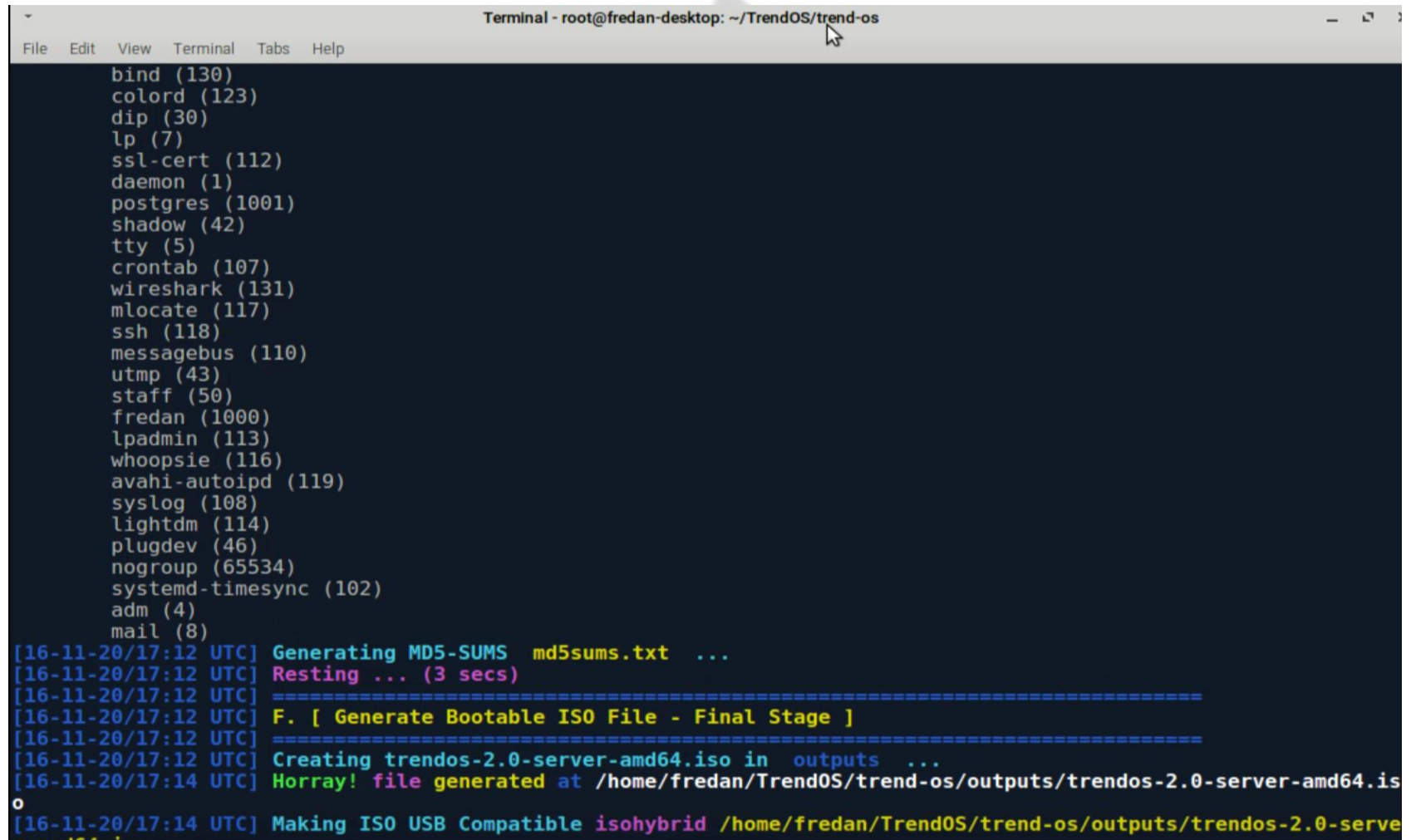
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

/abstract/schema_dumper.rb
1,777 100% 2.32kB/s 0:00:00 (xfr#20236, ir-chk=1010/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/abstract/schema_statements.rb
40,894 100% 53.39kB/s 0:00:00 (xfr#20237, ir-chk=1009/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/abstract/transaction.rb
5,080 100% 6.63kB/s 0:00:00 (xfr#20238, ir-chk=1008/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/array_parser.rb
2,726 100% 3.55kB/s 0:00:00 (xfr#20239, ir-chk=1009/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/column.rb
611 100% 0.80kB/s 0:00:00 (xfr#20240, ir-chk=1008/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/database_statements.rb
7,996 100% 10.43kB/s 0:00:00 (xfr#20241, ir-chk=1007/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/oid.rb
1,789 100% 2.33kB/s 0:00:00 (xfr#20242, ir-chk=1006/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/quotings.rb
3,164 100% 4.12kB/s 0:00:00 (xfr#20243, ir-chk=1005/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/referential_integrity.rb
1,043 100% 1.36kB/s 0:00:00 (xfr#20244, ir-chk=1004/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/schema_definitions.rb
4,398 100% 5.73kB/s 0:00:00 (xfr#20245, ir-chk=1003/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/schema_statements.rb
24,175 100% 31.48kB/s 0:00:00 (xfr#20246, ir-chk=1002/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/utils.rb

```

Figure 4.12 Integrating redundant security modules to protect the kernel against itself.

Figure 4.13 shows sample compilation of various files and how they are integrated into the various security protocols for compatibility.



```
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

bind (130)
colord (123)
dip (30)
lp (7)
ssl-cert (112)
daemon (1)
postgres (1001)
shadow (42)
tty (5)
crontab (107)
wireshark (131)
mlocate (117)
ssh (118)
messagebus (110)
utmp (43)
staff (50)
fredan (1000)
lpadmin (113)
whoopsie (116)
avahi-autoipd (119)
syslog (108)
lightdm (114)
plugdev (46)
nogroup (65534)
systemd-timesync (102)
adm (4)
mail (8)
[16-11-20/17:12 UTC] Generating MD5-SUMS md5sums.txt ...
[16-11-20/17:12 UTC] Resting ... (3 secs)
=====
[16-11-20/17:12 UTC] F. [ Generate Bootable ISO File - Final Stage ]
=====
[16-11-20/17:12 UTC] Creating trendos-2.0-server-amd64.iso in outputs ...
[16-11-20/17:14 UTC] Horray! file generated at /home/fredan/TrendOS/trend-os/outputs/trendos-2.0-server-amd64.iso
[16-11-20/17:14 UTC] Making ISO USB Compatible isohybrid /home/fredan/TrendOS/trend-os/outputs/trendos-2.0-server-amd64.iso
```


Figure 4.13 Building the various files and integrating into inbuilt protocols for compatibility

The installation process of the operating system has great user friendly experience. The screenshots from Figure 4.14 – Figure 4.20 shows some selected stages of installation processes from beginning to the completion of the prototype architecture.

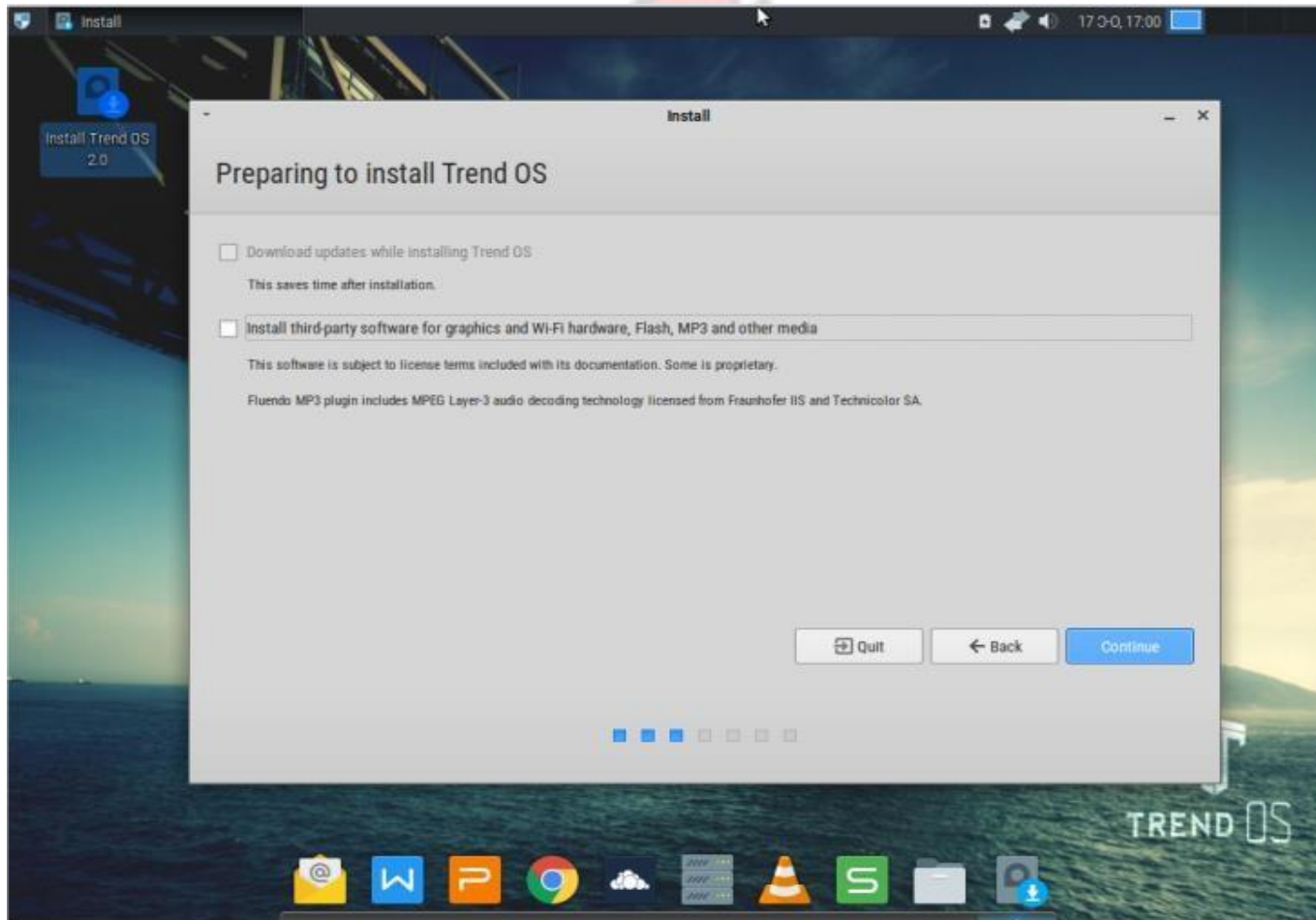


Figure 4.14 Good user friendly installation experience



Figure 4.15 Step by step system Installation guide.

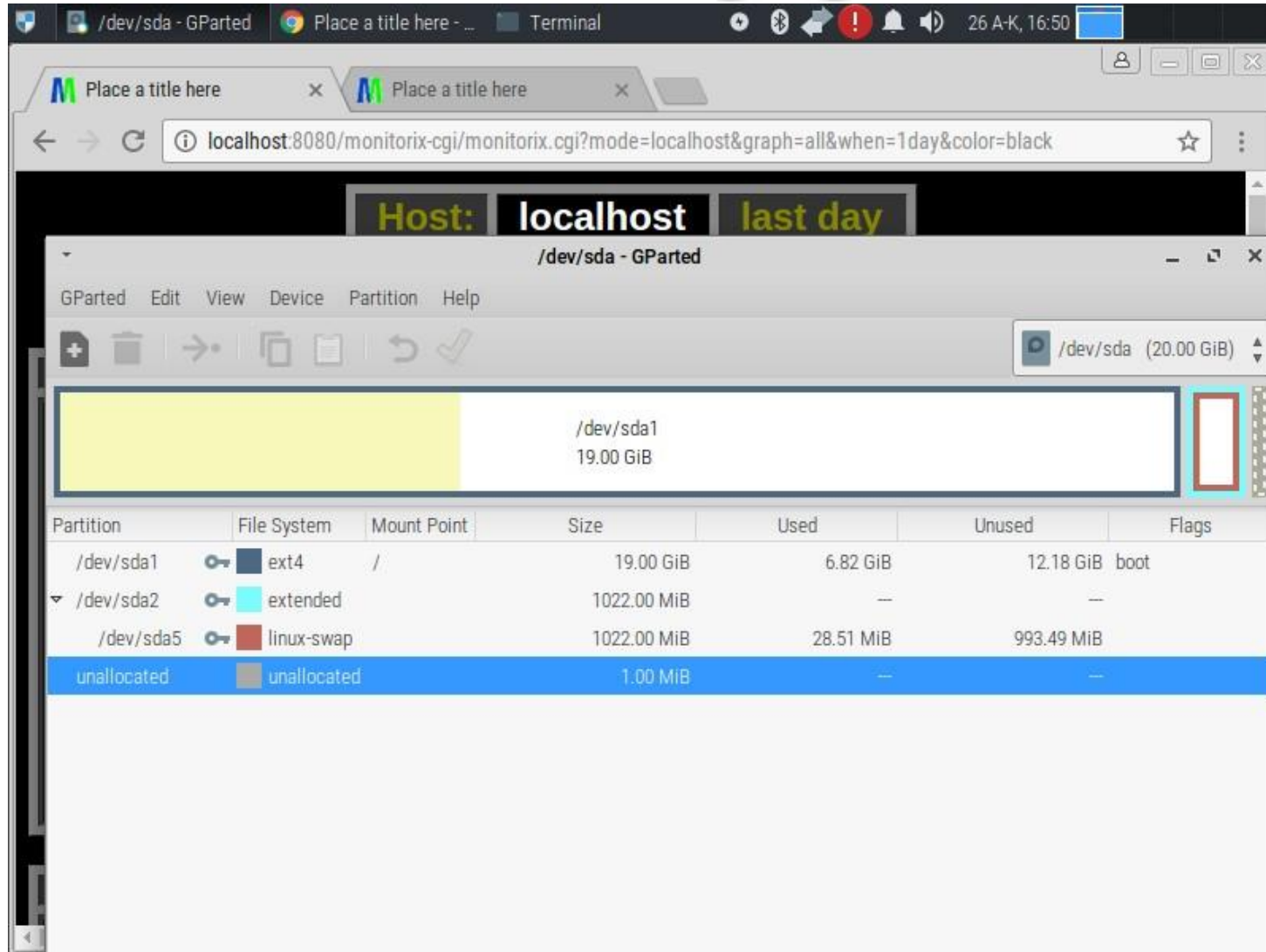


Figure 4.16 Effective Disk space utilization.

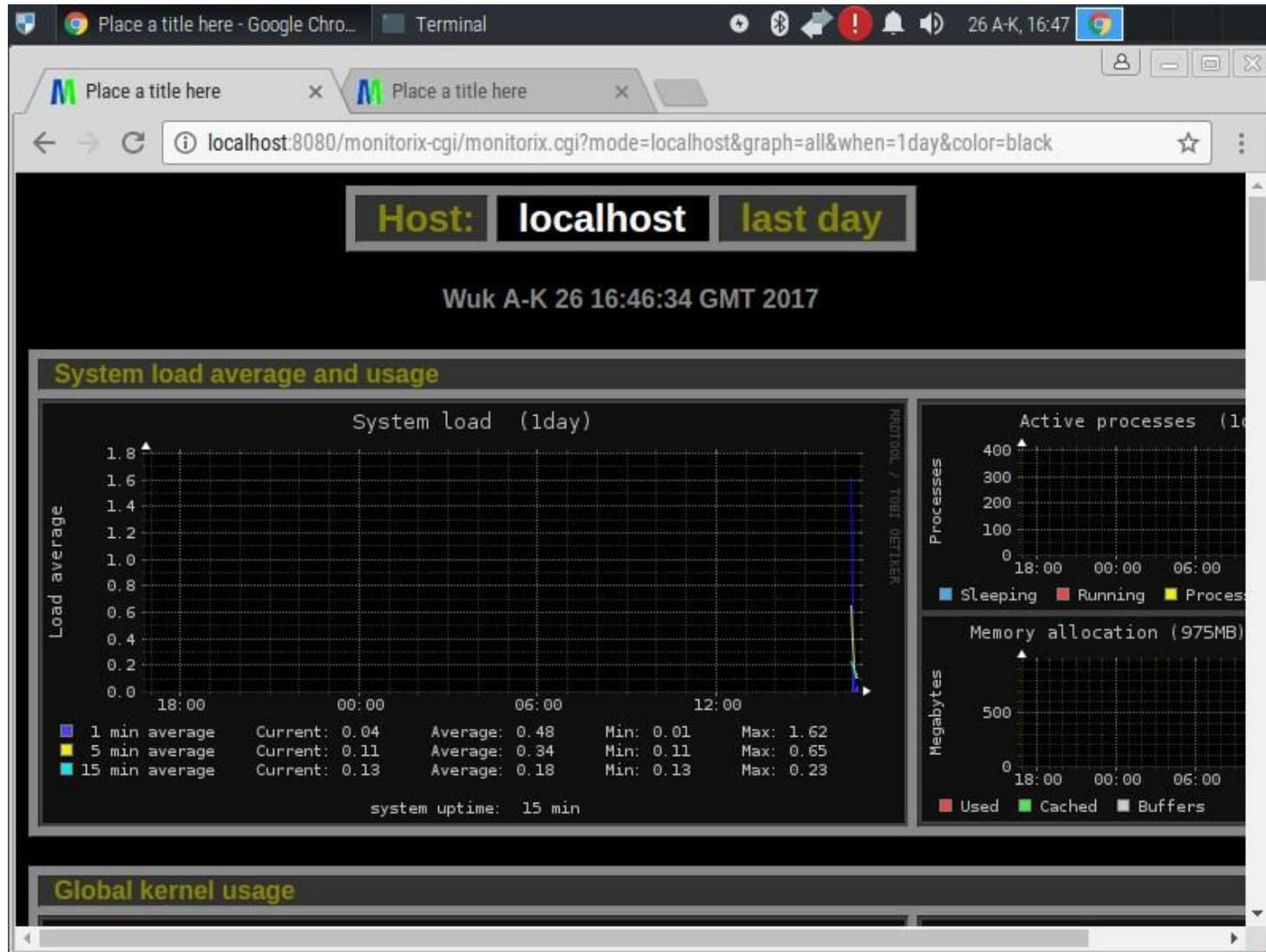


Figure 4.17 Minimum load system resource utilization.

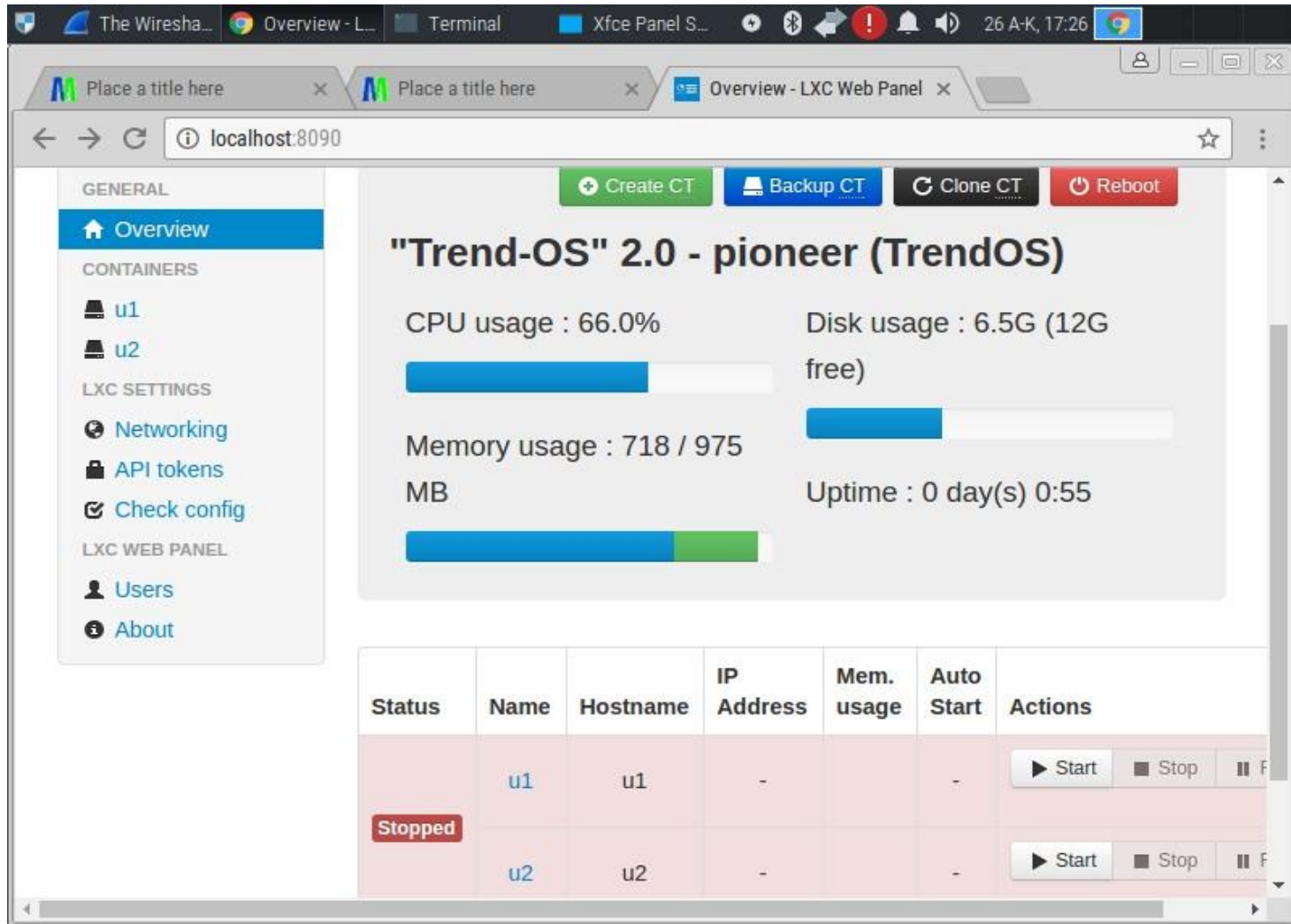


Figure 4.18 Support for multiple server instances running on same system – Linux containers.

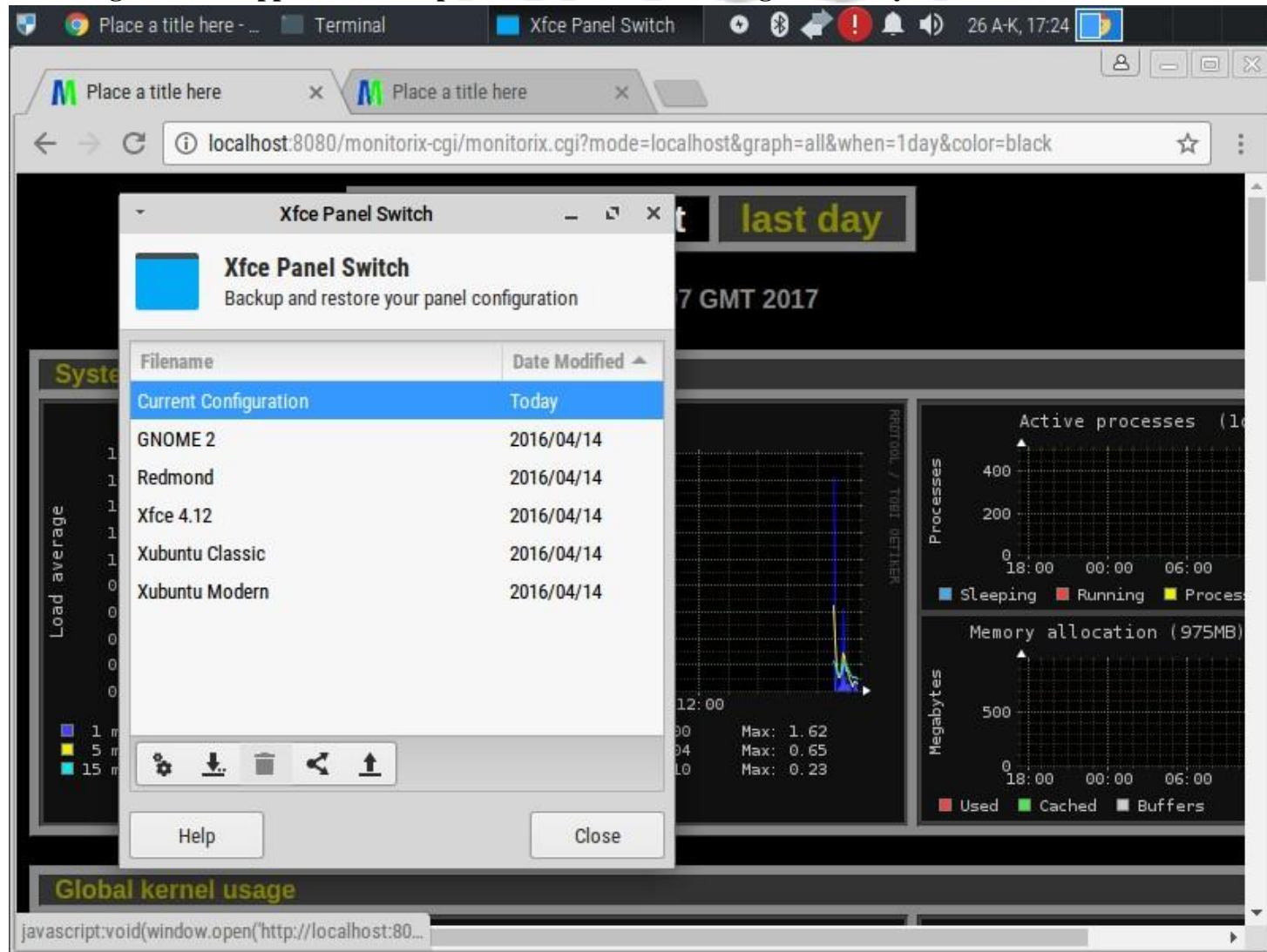


Figure 4.19 Live backup systems for High Availability.

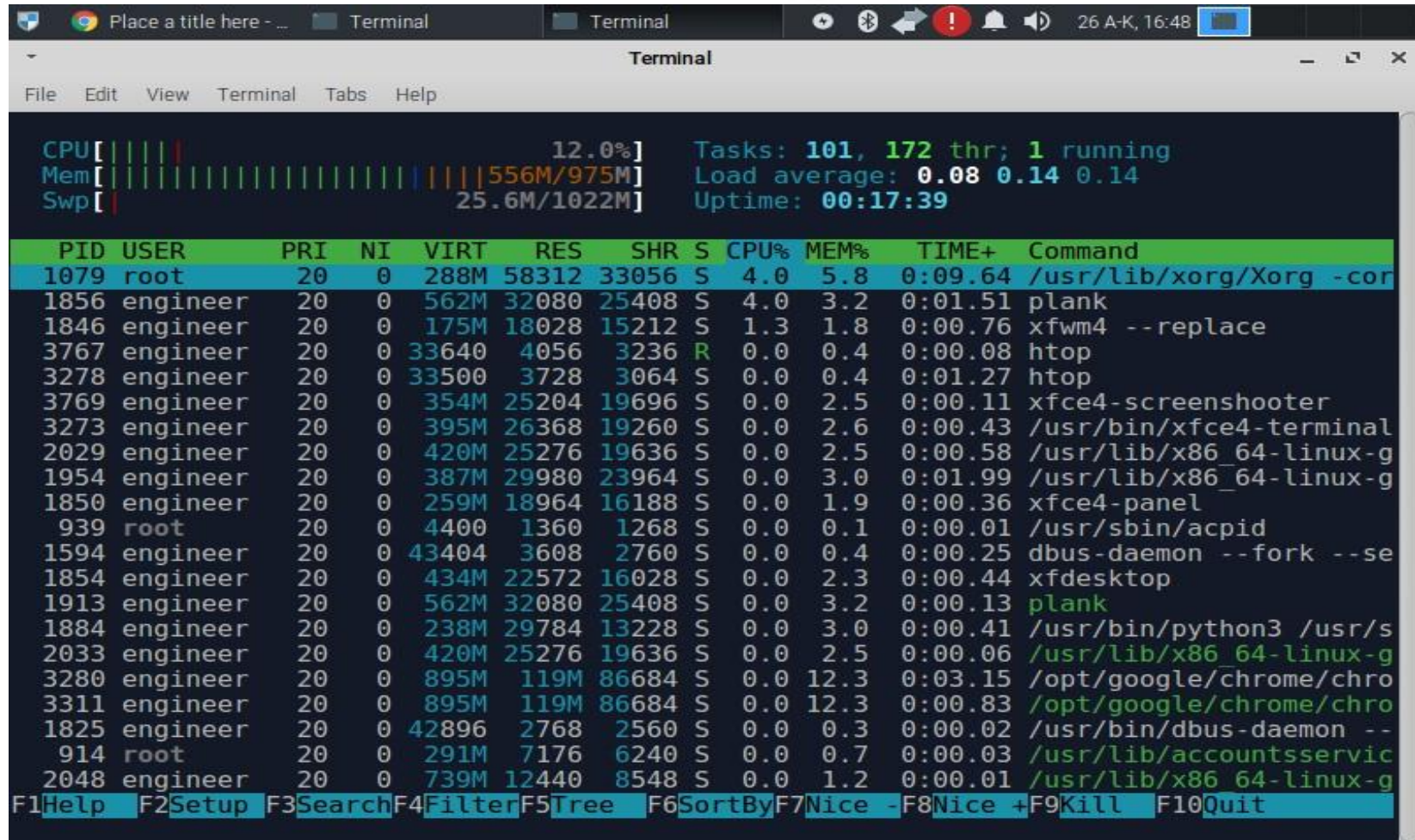


Figure 4.20 Minimum CPU and Memory utilization.

4.10. Summary

This chapter discussed the overview of the Convolutional Kernel Architectural framework and a comparative study with the traditional Linux kernel. The architecture is specially designed for trusted server environment. It has an integrated layer of a customized Unified Threat Management (UTM) which filters all traffic to ensure all network traffic is safe and provides a high level of protection through a chaining mechanism that allows packets to be filtered thoroughly before they are finally routed to their final destinations. The framework used is a combined monolithic and microkernel based (hybrid) architecture code-named – the integrated approach, to trade in the benefits of both designs. The architecture serves as the base framework for the *Trust Resilient Enhanced Network Defense Operating System (TREND-OS)* currently experimented in the lab. The aim is to develop an architecture that can protect the kernel against itself and applications (Snyder & One, n.d.).



CHAPTER 5

STEALTH-OBFUSCATION ZKP AUTHENTICATION PROTOCOL

5.1 Introduction

In this chapter, we present a password authentication protocol over untrusted networks. This password authentication protocol stores user password in a non-plaintext equivalent therefore, a breached database would not reveal enough information about the user password. This protocol relies on the strength of discrete logarithms with the Schnorr Signature Scheme and also goes further to satisfy all properties of a zero knowledge proof system.

User authentication systems have evolved throughout the years with the focus on securely proving a party's legitimacy to another. These user authentication systems can be categorized as "Biometrics", "Tokens – Cell Phone", "Password – Pin", Location to a combination of these factors which is known as a Multi-factor user authentication scheme. Although all these factors are designed to offer some protection against security attacks, the current trends in computing keeps introducing new threats.

Table 5.1 Categories of user Authentication Systems

Biometrics	What a Person is?
Tokens – Cell Phone	What a Person has?
Password – Pin	What a Person knows?
Location	Where a person is?

This model deals with a mechanism for authentication for "What the person knows" (Knowledge Factor) category of user authentication. In this scheme, the user's password and pin are the only secret available to the client whereas the network between the client and server is perceived to be untrusted. The only trusted parties in this authentication are client and the server application requesting the authentication hence the need to verify the knowledge of the secret keys without disclosing enough information about them on the network. Such a scheme requires no more than

just the client and the server requesting the authentication hence it is easy to implement in almost all Knowledge factor user authentication scheme since no additional hardware devices are required. To increase the entire security of such a scheme, other factors such as “What the person has” and “What the person is” can be employed as well to make it multi factor.

Knowledge factor authentication schemes have been the primary type of authentication to almost all web and software services. They are basically easy to design and implement in any architecture due to simplicity and low overhead in its implementation as compared to other factors like biometrics and location. Although a factor like the biometrics has proven to be more secure than the traditional Knowledge Factor authentication scheme, most of its implementation is still susceptible to over a decade old trick where fingerprints are raised from readers and used for authentication. Location based authentication is also a good contender for secure authentication but device latency, availability of device and its services hinders its implementation. Many researches have been carried out in the area of location based authentication and they lay out techniques for authentication, STAT I (Space – Time Authentication Technique) uses a GPS for determining a user’s location for authentication while the STAT II uses a proprietary IQRF technology for determining its user’s location for authentication (Ghogare et al, 2012). As an alternative to just location-based authentication, Hang et al. proposed a two-factor authentication scheme (location-based authentication with security questions) as a fallback mechanism to other authentication mechanism like a knowledge-based scheme. After implementation and testing of their proposed scheme, they found out that around 90% of their users were able to remember the locations to security questions within a 30 meter range while attackers could not successfully guess such locations (Hang et al, 2015).

5.2 Existing Models

Building authentication schemes with zero-knowledge proofs has been researched and implemented in many flavors along the years. The need to provide such secure authentication schemes for web applications and services has been driven by the influx of mobile devices and internet in our daily lives. Some of the researches in Zero Knowledge Proof Authentication

Schemes are as follows: “NARWHALL-An implementation of zero knowledge authentication” (Cheu et al, 2014). In this paper they discussed several vulnerabilities in existing website authentication systems and NARWHALL as a more secure alternative to such authentication systems. NARWHALL fixes most of the discussed vulnerabilities by building on an original protocol described by Lum Jia Jun. The protocol described by Lum Jia Jun is based on the Zero Knowledge Authentication with Zero Knowledge framework which allows an easy implementation of Zero Knowledge Authentication (“at-commerce-7,” n.d.).

In the available values to the prover of the system is his password and a public key, and the available values to the verifier of the system is the same public key as the prover’s and a pseudonym of the user which would will be calculated for during any authentication round. NARWHALL builds on such protocol by adding more components for more secure authentication. During a signup session with NARWHALL, a username, password of the user is entered and also the websites public unique identifier, the password is hashed and a public key is generated from it. The website unique identifier prevents users with the same username and password from two website to have the same public key and also the username prevents users of the same website to have the same public key. During a login session to a random challenge is sent to the users login form and stored in a cookie; on any login attempt the cookie information is updated to prevent brute forcing. Some implementation issues were identified the worse of all being the dependence of Javascript for client side processing. In browsers with disabled Javascript, the whole authentication fails. Another issue could be attributed to salting the user’s password before generating a pseudonym for the user. The user’s password were not salted hence an attacker could pre-compute values of the credentials and submit it for authentication.

Sławomir et al. proposed a Zero Knowledge Proof Authentication based on isomorphic graphs which allows authentication with varying confidence and also security level (Grzonkowski et al, 2008). This protocol follows strictly the ZKP challenge – response round for an authentication, so AJAX and xml are used to meet the requirement. During authentication a user makes a request and a server responds with a challenge and a user replies with a response and the server sends its final response denoting a successful or failed login attempt this is simulated with AJAX and XML. In an authentication process, a user enters his username and password and the browser calculate a

public and private key pair. The browser then calculate a challenge graph and sends it to the server and the server replies with a random challenge to the browser and the browser then chooses a response to the challenge and sends it to the browser. A response is finally sent from the server to the browser which denotes a successful authentication or failure. The public keys in this protocol are represented with two isomorphic graphs $G_1 = \pi_a(G_2)$ and the permutation of π_a is the private key. During authentication a prover will generate a random permutation and sends a graph G_r to the verifier (server) and depending on the challenge sent to the prover, the prover responds with either π_r or $\pi_r \circ \pi_p^{-1}$ then the verifier is able to check if the private and public keys are valid. In this implementation there could be attacks on the Graph isomorphism if an algorithm with Corneil et al. (Corneil, 1970) algorithm which determines if two graphs are isomorphic thereby increasing the speed of brute force attacks on it. Furthermore this implementation is also susceptible to dictionary attacks, hence when a website is breached and user login details are stolen, it could be used to attack this implementation.

Thiruvaazhi et al. proposed an elliptic curve discrete logarithm zero knowledge proof protocol for proving a user's binding to a public key and also his possession of a private key (Lomte, 2012).

In their scheme a user's visited domain is hashed and encoded and sent to the web server and the web server responds with its actual public key and it is hashed and encoded by the user and also verified against the original encoded hash of the domain name during registration to check it validity. In proving the private key, an elliptic curve will be generated over a finite field; a prover will choose a random value and compute the witness and send it to a verifier. A verifier will also randomly choose a challenge as either 0 or 1 and sends it to the prover and the prover will respond based on the challenge and finally the verifier will compute the validity of the response based on the prover's response. In this implementation there were some performance issues because of the iterations needed for the Zero Knowledge Proof challenge – response authentication round which could be detriment to its implementation on mobile platforms.

5.3 Zero Knowledge Proofs

A zero-knowledge proof is a method by which one party can prove to another that a statement is true by disclosing no other information than the fact that the statement is true. A derivative of this scheme is the Zero Knowledge Proof of Knowledge which allows a prover to prove to a verifier that a statement is true and also possesses a witness for the fact (Fiege, Fiat, & Shamir, 1987). For an authentication system to be zero knowledge it has to be

- Complete
- Sound
- Zero-Knowledge

A system is complete when a prover can convince the verifier that a statement is true and no cheating prover can convince the verifier otherwise and it is sound if when a statement is false no cheating prover can convince the verifier that it is true and it is zero-knowledge when a cheating verifier can only learn that the statement is true. Zero knowledge proofs are interactive protocols with zero knowledge. Interactive proof system was introduced (Babai, 1985) and the Zero Knowledge (Goldwasser et al, 1989). Although the study of Zero Knowledge Proofs is primarily focused on user authentication, it can further be implemented in digital payment systems and electronic voting systems (Boneh, 2004).

5.4 Schnorr's Identification Protocol

This is a three move protocol in which the exchanged messages; the commitment, challenge and response are exchanged between a prover and verifier to be able to prove the knowledge of a secret key. The first step involves the prover sending a commitment to the verifier and the verifier responding with a challenge and finally the prover sending its response for final verification of the knowledge secret key. To describe this scheme we can define the values available to the prover as $(g, q, y, \text{ and } x)$ and that of the verifier as $(g, q, \text{ and } y)$ where g, q and y are public keys and x is the secret key only known by the prover

- Commitment

Prover: $r \in_R \mathbb{Z}_q \Rightarrow t = g^r$ (Send t to Verifier)

- Challenge

Verifier: $C \in \{0,1\}^k$ (Send C to Prover)

- Response

Prover: $s = r - cx \pmod{q}$ (Send s to Verifier)

Verifier: $t = g^s y^c$ (Yes / No)

Let $g \in G$ be a generator of G (a finite group of order q). Let $y = g^x$ be the public key of the prover and x the secret key, we can then prove the knowledge of the secret key.

$s = r - cx$ and $y = g^x$

$$t_{\text{Verifier}} = g^{r-cx} g^{cx}$$

$$t_{\text{Verifier}} = g^r$$

$$t_{\text{prover}} = t_{\text{Verifier}}$$

Such proofs of knowledge are useful in the construction of signature schemes.

5.5 Schnorr's Signature Scheme

This is a digital signature scheme which is a variant of the ElGamal Signature scheme. It is efficient and generates shorter signatures as compared to the ElGamal signature scheme. A Schnorr signature of message $m \in \{0,1\}^*$ is a pair (c, s) with $c, s \in \mathbb{Z}_q$ and satisfying the verification equation $c = H(m || g^s y^c)$ where H is a collision-resistant cryptographic hash function $\{0,1\}^* \rightarrow \{0,1\}^l$ that maps to a fixed hashed output.

- Key Generation Phase: Secret Key = x , $y = g^x$
- Message Signing Phase:
 1. Choose a Random r from a set
 2. $R = g^r$

3. $c = H(m||R||y)$ 4. $s = r - cx \pmod{q}$ • Verification Phase:

$$H(m||y^c g^s) = H(m||g^r)$$

For correctness of the scheme we can verify it by using the values g^r and $y^c g^s$.

$$s = r - cx \text{ and } y = g^x$$

$$H(m||g^{cx} g^{r-cx}) = H(m||g^r)$$

The verification phase shows how a signed message can be verified by the other party.

5.6 Signatures Based Proof Knowledge – (SPK)

Signature based on proofs of knowledge is used to prove the possession of secret keys (Schnorr, 1990). For a pair $(c, s) \in \{0,1\}^l \times Z_q$ satisfying $c = H\{V|m\}$ with $s = g||y$ and $V = g^s y^c$ is an SPK of the discrete logarithm of a group element y to the base of g of the message $m \in \{0,1\}^*$ and is denoted $SPK_1\{(\alpha): y = g^\alpha\}(m)$. For an SPK_1 , the secret value can be expressed in terms of the public key as $y = g^x$ where x is the secret value and the random integer from the set Z_q can be expressed as $t = g^r$ as shown in Listing 1.2. The challenge can be expressed as $c = H(t|m)$ and the response as $s = r - cx \pmod{q}$. A general notation of a proof of knowledge of the secret keys α and β can be expressed as $SPK_1\{(\alpha, \beta): y = g^\alpha \wedge z = g^\beta h^\alpha\}(m)$.

5.7 Stealth Obfuscation ZKP Scheme

The scheme relies on SPK_1 for website authentication. The strength of our scheme as compared to NARWHAL (a challenge-response model authentication based on zero knowledge proofs) (Cheu et al, 2014) is based on the complexity of how passwords are stored. It doesn't share weakness with hashed passwords since an attacker with pre-computed values of passwords would not be able to look it up. Our scheme offers a stealth authentication over networks since not much information is leaked on the network during an authentication round.

5.8 Implementation of Stealth Obfuscation ZKP

a) User Registration

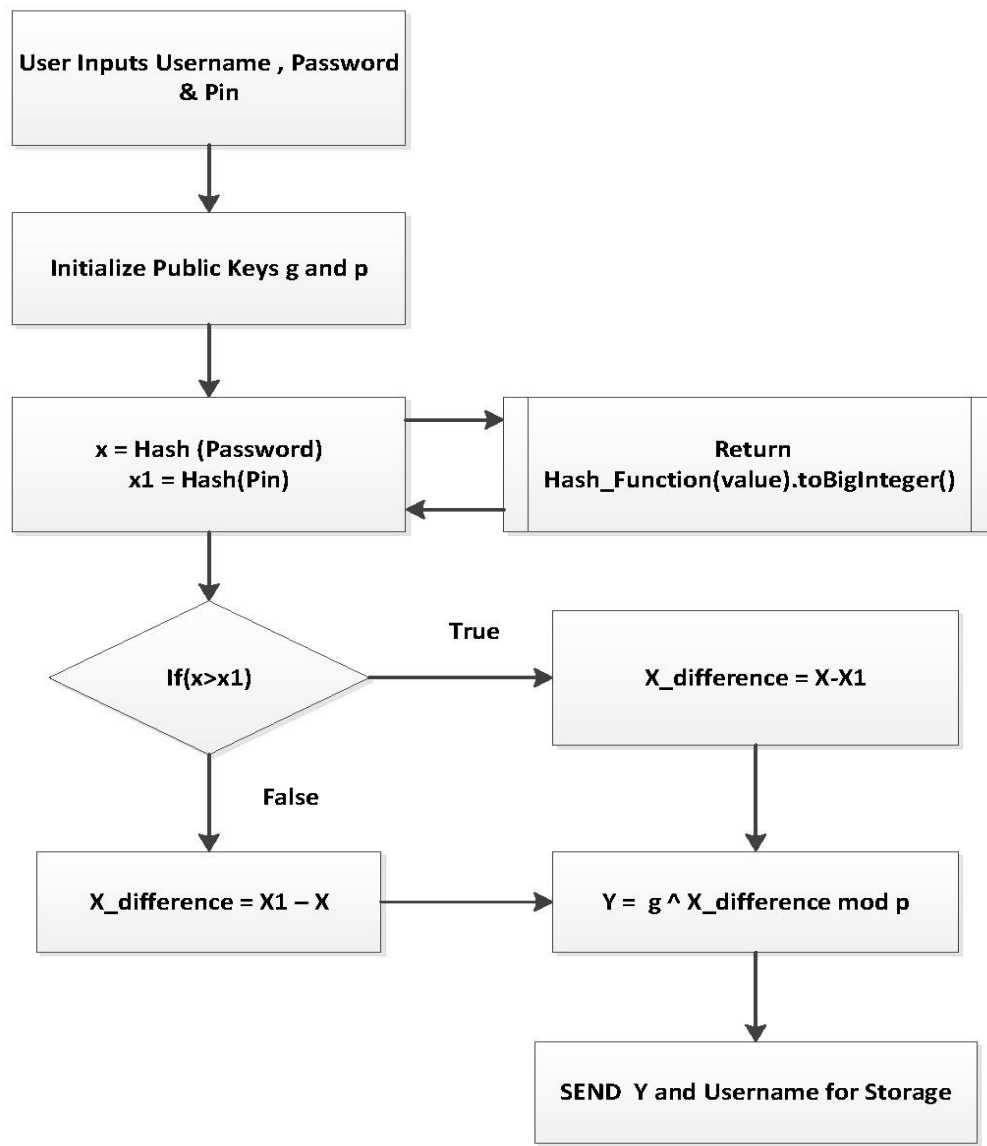


Figure 5.1 Stealth ZKP User Registration (Client Side).

SIGN UP ALGORITHM – CLIENT SIDE

1. Clients browser requests for the login page
2. The server responds with the login page and two public keys g and p
3. Client enter a password, pin and username
4. The pin and password are hashed and converted to BigInteger values
5. If the hash of the password in BigInteger value is greater than hash of the **pin** BigInteger Value you subtract the pin's value from the password value.

i.e. $\text{Difference} = \text{Password} - \text{Pin}$

else

Subtract the Password from Pin's BigInteger Values i.e.

$\text{Difference} = \text{Pin} - \text{Password}$

6. We find the public values **Y y string** on the server

$$Y = \text{public key}(y)^{\text{Difference}} \bmod \text{public key}$$

$$Y = g^{\text{Difference}} \bmod p$$

In the user registration process, a user chooses a username, password and pin of their choice and their pin and password are hashed with a collision -resistant hash function and converted to big integers (Schnorr, 1990).

A difference of the larger value from the smaller value is taken and computed with the cryptographic group element g_0 as $y = g_0^{\text{difference}}$ and stored with the username to the server. The value stored on the server does not reveal enough information about the private key of the client being (x). The value stored on the server (y) becomes the user's public key.

b) User Sign In (Client Side)

During sign-in the user enters his username, password and pin as they registered with and the password and pin are hashed using the same collision-resistant hash function as at signup and the values converted to big integer values for further computation. From $SPK_1\{(\alpha): y = g^\alpha\}(m)$ we can set our message parameter to null ($SPK_1\{(\alpha): y = g^\alpha\}$) and deduce the signing as follows:

$$1) y = g^{0x_{difference}}$$

$$2) T_{client} = g^{0r_{random}}$$

$$3) C = H(T_{client} || Y)$$

$$4) z = r_{random} - CX_{difference}$$

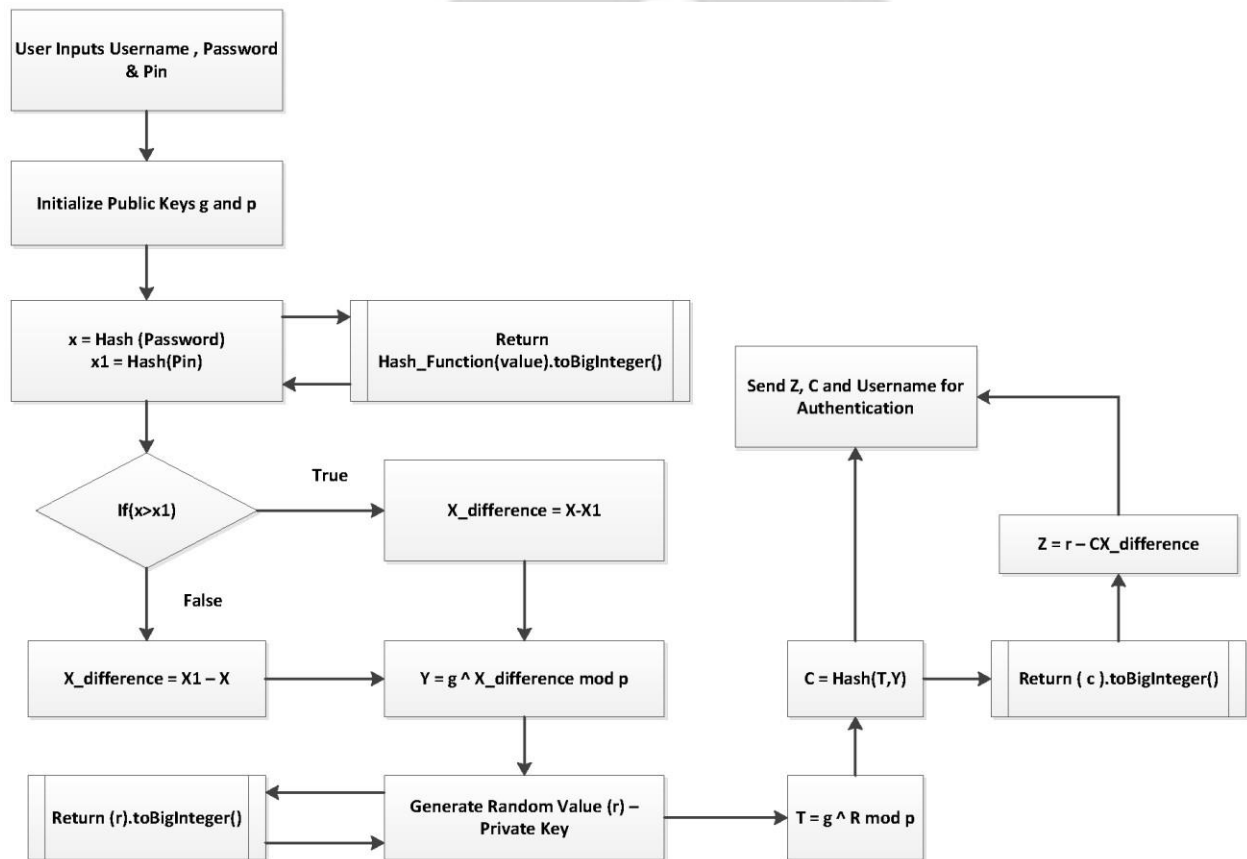


Figure 5.2 Stealth ZKP User Authentication (Client Side).

SIGN IN ALGORITHM – CLIENT SIDE

1. Clients browser requests for the sign in page
2. Server responds with a sign in page and public keys g and p
3. User enters username, password and pin
4. Password and Pin is hashed and connected to Big Integer Values
5. If the Big Integer value of the password is greater than that of the pin; You subtract the pin from the password i.e.

Difference = Password – Pin

Likewise, if the Pin is greater than the password, you subtract the password from the pin.

NB: This is done to ensure that the value is always going to be a non-negative value. 6.

Find the positive value Y as

$$Y = g^{\text{Difference}} \bmod p$$

7. Generate a random number (r) and convert it to BigInteger NB. This random value is the protocol secret key.
8. Perform the computation

$$T = g^r \bmod p$$

9. Hash the value of the computation on the secret key and Public Y key and convert to BigIntegers

$$C = \text{Hash}(T, Y)$$

Convert C to Big Integer

10. Finally, Get the difference of the product of C and Difference of the pin and password from the random private key r .

$$Z = r - C * \text{Difference}$$

Send Z , C and Username for authentication.

The client sends (C and z) to server for authentication. The sample code for this process is as shown in Listing 4.2.

Listing 5.1 Client side Login Sample Code

```

function random() {
var wordCount = 4;
var randomWords;

    if (window.crypto && window.crypto.getRandomValues) {
randomWords = new Int32Array(wordCount);
window.crypto.getRandomValues(randomWords);
    }
    else if (window.msCrypto && window.msCrypto.getRandomValues) {
randomWords = new Int32Array(wordCount);
window.msCrypto.getRandomValues(randomWords);
    }
    var string = '';

    for( var i=0; i<wordCount; i++ ) {
var int32 = randomWords[i];
< 0 ) int32 = -1 * int32;
string + int32.toString(16);
    }
    return string;
}
function hash(x) {
return SHA256(x).toLowerCase();
}

$(function(){

    $('#frm_login').submit(function(e){

        var r = random();
var p = new
BigInteger("EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D
674DF7496EA81D3383B4813D692C6E0E0D5D8E250B98BE48E495C1D6089DAD15DC7D7B46154D6
B6CE8EF4AD69B15D4982559B297BCF1885C529F566660E57EC68EDBC3C05726CC02FD4CBF4976
EAA9AFD5138FE8376435B9FC61D2FC0EB06E3", 16);
var g = new BigInteger("2",16);
username = $('#login_username').val();
password = $('#login_password').val();

        var x = hash(password);
var xbig = new BigInteger(x, 16);
var y = g.modPow(xbig, p);
    
```



```

        var rbig = new BigInteger(r, 16);
var t = g.modPow(rbig, p);          var yt
= t.add(y);          var c =
hash(yt.toString());          var cbig =
new BigInteger(c, 16);
        var cx = cbig.multiply(xbig);
var sub_r_cx = rbig.subtract(cx);

        var p1 = p.subtract(new BigInteger("1",16));

        if(cx.max(rbig)){
            sub_r_cx = cx.subtract(rbig);

            sub_r_cx = sub_r_cx.mod(p1);

            sub_r_cx = p1.subtract(sub_r_cx);
        }

        var z = sub_r_cx.mod(p1);

        $("#c").val(c);
        $("#z").val(z);
        $("#y").val(y);
        $("#login_password").val("");
    });

    $('#frm_signup').submit(function(e){
var p = new
BigInteger("EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D
674DF7496EA81D3383B4813D692C6E0E0D5D8E250B98BE48E495C1D6089DAD15DC7D7B46154D6
B6CE8EF4AD69B15D4982559B297BCF1885C529F566660E57EC68EDBC3C05726CC02FD4CBF4976
EAA9AFD5138FE8376435B9FC61D2FC0EB06E3", 16);
var g = new BigInteger("2",16);          var
username = $('#new_username').val();          var
password = $('#new_password').val();

        var x = hash(password);
var xbig = new BigInteger(x, 16);
var y = g.modPow(xbig, p);
        $("#new_password").val(y);
    });

});

```

The choice of a random module also requires a number of processes in order to make this possible. Part of the code for the execution of this process is as shown in Listing 5.2.

Listing 5.2 Random Module Sample Code

```
var random16byteHex = (function() {  
    function random() {        var  
wordCount = 4;        var randomWords;  
  
        // First we're going to try to use a built-in CSPRNG  
if (window.crypto && window.crypto.getRandomValues) {  
    randomWords = new Int32Array(wordCount);  
    window.crypto.getRandomValues(randomWords);  
    }  
    // Because of course IE calls it msCrypto instead of being standard  
else if (window.msCrypto && window.msCrypto.getRandomValues) {  
    randomWords = new Int32Array(wordCount);  
    window.msCrypto.getRandomValues(randomWords);  
    }  
    // Last resort - we'll use isaac.js to get a random number. It's seeded  
from Math.random(),  
    // but we can run it for 100ms/0.1s to advance it a distance which will  
be dependent upon  
    // hardware and js engine. Also we will have the onkeyup skip a char  
worth of values.  
    else {  
        randomWords = [];  
        for (var i = 0; i < wordCount; i++) {  
randomWords.push(isaac.rand());  
        }  
    }  
    var string = '';  
  
    for( var i=0; i<wordCount; i++ ) {  
var int32 = randomWords[i];        if( int32  
< 0 ) int32 = -1 * int32;        string =  
string + int32.toString(16);  
    }  
    return string;  
};  
  
function isCrypto() {  
    if (window.crypto && window.crypto.getRandomValues) {  
return true;  
    }  
    else if (window.msCrypto && window.msCrypto.getRandomValues) {  
return true;    } else {        return false;  
    }  
}
```

```

};
var crypto = isCrypto();

function advance(ms) {
    if(
!crypto ) {
        var start =
Date.now();
        var end = start
+ ms;
        while( Date.now() <
end ) {
            var r = isaac.random() * 128 + (Date.now() - start);
            isaac.prng(Math.floor(r));
        }
    }
}
return {
    'random' : random,
    'isCrypto' : crypto,
    'advance' : advance
};
})();

// if it is isaac spend 0.1s advancing the stream
random16byteHex.advance(100);

```

c) User Sign In – (Server Side)

At the user authentication at server side, the process has to be proved for correctness.

- 1) $T_{Server} = ycgz$
- 2) $T_{Server} = g^{xdifference}c^{grandom-Cxdifference}$
- 3) $T_{Server} = g^{r_{random}}$
- 4) $T_{Server} = T_{client} = g^{r_{random}}$

For any round of authentication valid credentials can be verified. The process for the signup at the server side is also required as part of the protection and security measures as explained earlier.

Listing 5.3 Signup Server Side Sample Code

```

<?php

if ($ POST['action'] == 'new user'){

    $username = $ POST['username'];
    $password = $ POST['password'];

    $db = mysqli_connect('localhost', 'root', '', 'zkp'); if(
$db->query("insert into account (username, password) values
('".$username."', '".$password."')")){
        echo "User : ".$username." created succesfully with password
".$password;
    }
}

?>

```

Listing 5.4 Login Server Side Sample Code

```

<?php

require "zkp.php";

$username = $_POST['username'];

$db = mysqli_connect('localhost', 'root', '', 'zkp');

$result = $db->query("select password from account where username =
'".$username."' limit 1");

$_password = "";

while($row = mysqli_fetch_assoc($result))
{
    $ password = $row['password'];
}

$c = $_POST['c'];
$z = $_POST['z'];

var_dump($_POST);

$zkp = new Zkp($_password, $c, $z);

```

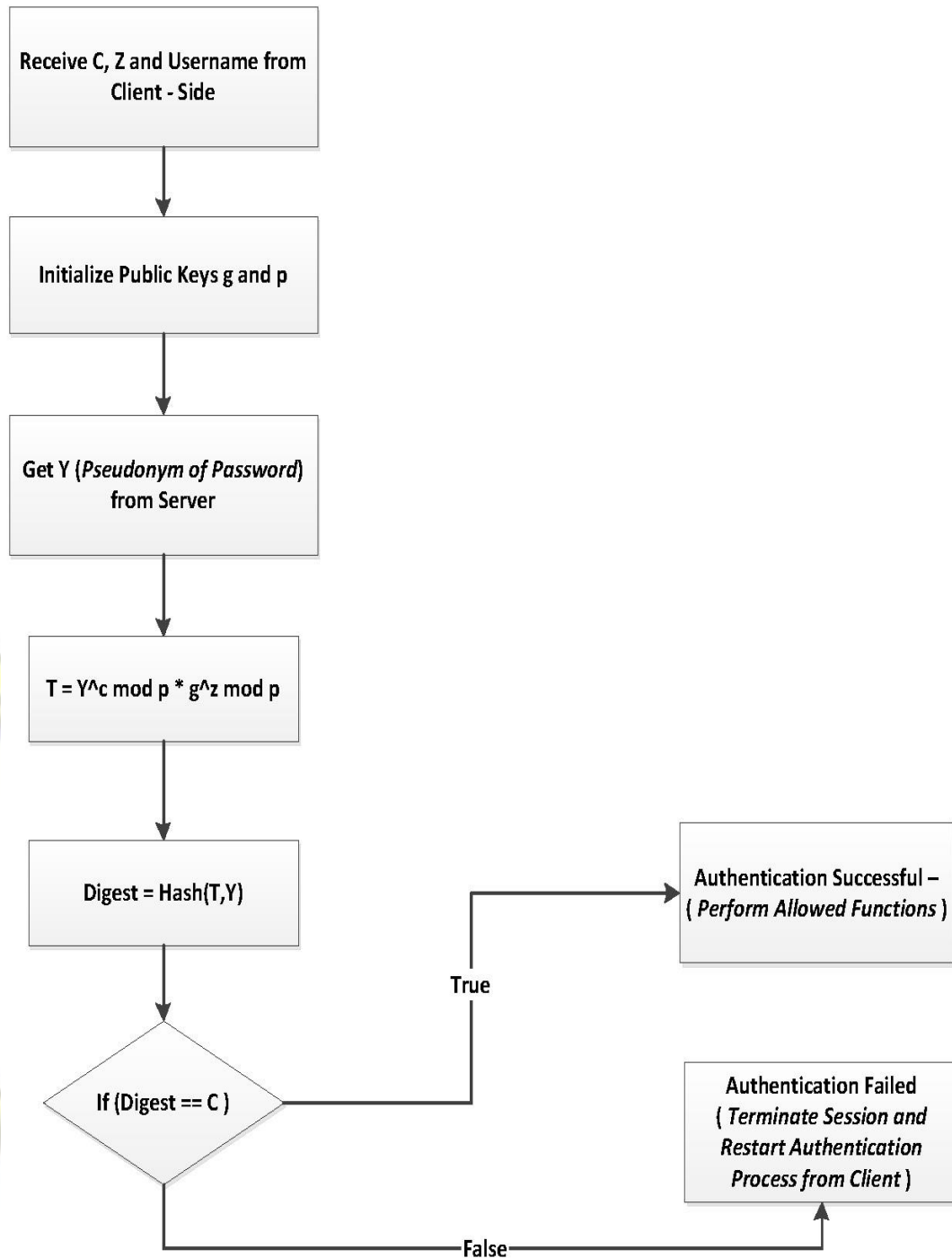



Figure 5.3 Stealth User Authentication (Server Side).

SIGN IN ALGORITHM – SERVER SIDE

1. **Recover** C, Z and username from client
2. Get the Pseudonym of password (Y) corresponding to the username stored on the server
3. Compute $TT = Y^c \bmod p * g^z \bmod p$

In our initial assumption from the **ym** users sign up process

$$Y = g^{\text{Difference}} \bmod p$$

From our sign up process

$$Z = r - c [\text{Difference}]$$

So $(g^{\text{Difference}})^c * g^{r-c [\text{Difference}]}$

Then $g^{r(c[\text{difference}] + c[\text{difference}])} \bmod p$

From the client sign in process

- a. $T = g^r \bmod p$
- b. So we reduce a

$$g^r \bmod p$$

Therefore if the C and Z values submitted the correct value of Y, The random value generated r can always be proven

i.e. $T = g^r \bmod p$ have logged in successfully else

Login error

Listing 5.5 Zero Knowledge Proof Sample Code

```
<?php

require 'BigInteger.php';

class Zkp {

protected $a;
protected $α;
protected $p;

protected $v;

protected $username;
protected $c;
protected $z;

function construct($ v, $ c, $ z){
$this->p = new
BigInteger("EEAF0AB9ADB38DD69C33F80AFA8FC5E86072618775FF3C0B9EA2314C9C256576D
674DF7496EA81D3383B4813D692C6E0E0D5D8E250B98BE48E495C1D6089DAD15DC7D7B46154D6
B6CE8EF4AD69B15D4982559B297BCF1885C529F566660E57EC68EDBC3C05726CC02FD4CBF4976
EAA9AFD5138FE8376435B9FC61D2FC0EB06E3", 16);
$this->α = new BigInteger("2",16);
$this->y = new BigInteger(strtoupper($_y));
$this->c = $ c;
$this->z = new BigInteger($_z);
$this->authenticate();
}

public function setY($ v){
    $this->y = new BigInteger($_y);
}

public function setC($ _c){
    $this->c = $_c;
}

public function setZ($ z){
    $this->z = new BigInteger($_z);
}

public function authenticate(){

    $bigC = new BigInteger($this->c, 16);

    $powYC = $this->y->powMod( $bigC , $this->p);

    $powGZ = $this->g->powMod($this->z, $this->p);

    $t = bcmul($powYC, $powGZ);
    $t = bccomp($t, $this->p);
    if ($t < 0) {
        $t = $t + $this->p;
    }
    $t = bcmod($t, $this->p);
    $this->authenticate();
}
```

```

$ty = bcadd($t, $this->y);

$hash = $this->hash($ty);
if($hash == $this->c){
    echo "<br>You succesfully Authenticated";
}

}

public function hash($x) {
    return strtolower(hash('sha256', $x));
}

}

?>

```



Listing 5.6 SHA256 Hashing Code Sample

```
/**
 *
 * Secure Hash Algorithm (SHA256)
 * http://www.webtoolkit.info/javascript-sha256.html
 * http://anmar.eu.org/projects/jssha2/
 *
 * Original code by Angel Marin, Paul Johnston.
 *
 */
function SHA256(s) {
    var chrsz = 8;
    var hexcase = 0;

    function safe_add(x, y) {
        var lsw = (x & 0xFFFF) + (y & 0xFFFF);
        var msw = (x >> 16) + (y >> 16) + (lsw >> 16);
        return (msw << 16) | (lsw & 0xFFFF);
    }

    function S(X, n)
    {
        return (X >>> n) | (X << (32 - n));
    }

    function R(X, n)
    {
        return (X >>> n);
    }

    function Ch(x, y, z) {
        return ((x & y) ^ ((~x) & z));
    }

    function Maj(x, y, z) {
        return ((x & y) ^ (x & z) ^ (y & z));
    }

    function Sigma0256(x) {
        return (S(x, 2) ^ S(x, 13) ^ S(x, 22));
    }

    function Sigma1256(x) {
        return (S(x, 6) ^ S(x, 11) ^ S(x, 25));
    }

    function Gamma0256(x) {
        return (S(x, 7) ^ S(x, 18) ^ R(x, 3));
    }

    function Gamma1256(x) {
        return (S(x, 17) ^ S(x, 19) ^ R(x, 10));
    }

    function core_sha256(m, l) {
```

```

    var K = new Array(0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5, 0xD807AA98, 0x12835B01,
0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
0xE49B69C1, 0xEFBE4786, 0xFC19DC6, 0x240CA1CC, 0x2DE92C6F, 0x4A7484AA,
0x5CB0A9DC, 0x76F988DA, 0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
0xC6E00BF3, 0xD5A79147, 0x6CA6351, 0x14292967, 0x27B70A85, 0x2E1B2138,
0x4D2C6DFC, 0x53380D13, 0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3, 0xD192E819, 0xD6990624,
0xF40E3585, 0x106AA070, 0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3, 0x748F82EE, 0x78A5636F,
0x84C87814, 0x8CC70208, 0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2);
var HASH = new Array(0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19);
var W = new Array(64);    var a, b, c, d, e, f,
g, h, i, j;    var T1, T2;
    m[l >> 5] |= 0x80 << (24 - l %
32);    m[((l + 64 >> 9) << 4) + 15] =
l;

    for (var i = 0; i < m.length; i += 16) {
a = HASH[0];    b = HASH[1];    c =
HASH[2];    d = HASH[3];    e =
HASH[4];    f = HASH[5];    g =
HASH[6];    h = HASH[7];

        for (var j = 0; j < 64; j++) {
if (j < 16) {
            W[j] = m[j + i];
        }
        else {
            W[j] = safe_add(safe_add(safe_add(Gamma1256(W[j - 2]), W[j - 7]),
Gamma0256(W[j - 15])), W[j - 16]);
        }

            T1 = safe_add(safe_add(safe_add(safe_add(h, Sigma1256(e)), Ch(e, f,
g)), K[j]), W[j]);
            T2 = safe_add(Sigma0256(a), Maj(a, b, c));
            h = g;    g =
f;    f = e;    e
= safe_add(d, T1);    d
= c;    c = b;
b = a;
            a = safe_add(T1, T2);
        }

        HASH[0] = safe_add(a, HASH[0]);
        HASH[1] = safe_add(b, HASH[1]);
        HASH[2] = safe_add(c, HASH[2]);
        HASH[3] = safe_add(d, HASH[3]);
        HASH[4] = safe_add(e, HASH[4]);
        HASH[5] = safe_add(f, HASH[5]);

```

```

        HASH[6] = safe_add(g, HASH[6]);
        HASH[7] = safe_add(h, HASH[7]);
    }
    return HASH;
}
function str2binb(str) {
var bin = Array();    var mask
= (1 << chrsz) - 1;
    for (var i = 0; i < str.length * chrsz; i += chrsz) {
        bin[i >> 5] |= (str.charCodeAt(i / chrsz) & mask) << (24 - i % 32);
    }
return bin;
}
function
Utf8Encode(string) {
    // METEOR change:
    // The webtoolkit.info version of this code added this
    // Utf8Encode function (which does seem necessary for dealing
    // with arbitrary Unicode), but the following line seems
    // problematic:
    //
    // string = string.replace(/\r\n/g, "\n");
var utftext = "";

    for (var n = 0; n < string.length; n++) {
        var c = string.charCodeAt(n);

        if (c < 128) {
            utftext += String.fromCharCode(c);
        }
        else if ((c > 127) && (c < 2048)) {
            utftext += String.fromCharCode((c >> 6) | 192);
            utftext += String.fromCharCode((c & 63) | 128);
        }
        else {
            utftext += String.fromCharCode((c >> 12) | 224);
            utftext += String.fromCharCode((c >> 6) & 63 | 128);
            utftext += String.fromCharCode((c & 63) | 128);
        }
    }
    return utftext;
}

function binb2hex(binarray) {
var hex_tab = hexcase ? "0123456789ABCDEF" : "0123456789abcdef";
var str = "";
    for (var i = 0; i < binarray.length * 4; i++) {
        str += hex_tab.charAt((binarray[i >> 2] >> ((3 - i % 4) * 8 + 4)) &
0xF) +

```

```

        hex_tab.charAt((binarray[i >> 2] >> ((3 - i % 4) * 8)) & 0xF);
    }
    return str;
}
s = Utf8Encode(s);
return binb2hex(core_sha256(str2binb(s), s.length * chrsz));
}

```

5.9 Strength of Stealth Obfuscation ZKP

The strength of our protocol is based on the strength of the discrete logarithm problem. The protocol will be able to solve the problem of user authentication over unsecure networks. During authentication, a user submits his public key and private key for authentication hence no other knowledge of the password is known. On unsecure networks, data sniffed would not reveal anything much about the user's secret key hence it cannot be replayed for another authentication session. The intractability of discrete logarithms and its easy implementation makes it a better candidate over Visual Cryptography, Pairing based Cryptography and Elliptic Curves for Zero Knowledge Proofs (Scott, n.d.) (Sahl, Samsudin, & Letchmunan, 2014).

5.10 Conclusion

With the defined problems addressed in this research, we can implement our zero knowledge proof scheme code-named Stealth Knowledge Authentication for devices with minimal computational resource. With HTTPS providing a secure transmission of authentication details and our Javascript assets files, we implement it to protect user details before and after transmission.

Although our implementation requires JavaScript which is supported by most browsers, some users disable it which would prevent it to run on such systems and also the authentication scheme is not multi – factor, which would make it susceptible to attack when the user's secret keys are known. Our implementation also doesn't follow the classical Zero Knowledge Proof Authentication's Challenge – Response round hence there could be a slight chance of a cheating verifier proving him or herself as an honest verifier hence Ajax and Sockets could be used to implement it hence reducing it iterating challenges.

KNUST



CHAPTER 6

RESULTS ANALYSIS

6.0 Introduction

The architecture having gone through development and implementation at the Lab was required of it to undergo a series of evaluation to identify the extent to which the objectives of the framework has been achieved. There was therefore the need to develop a prototype of the architecture using bare Debian kernel with a retrofit of the architecture modified in its kernel to begin the test experiment (Pourzolfaghar et al, 2011).

6.1 Evaluation

The architecture was made to undergo several comparative testing to establish the effectiveness of the architecture against its specified objectives. Evaluation on the architecture was carried out under the following parameters:

✓ Security Testing of the User Interface. Figure 6.1 – Figure 6.4

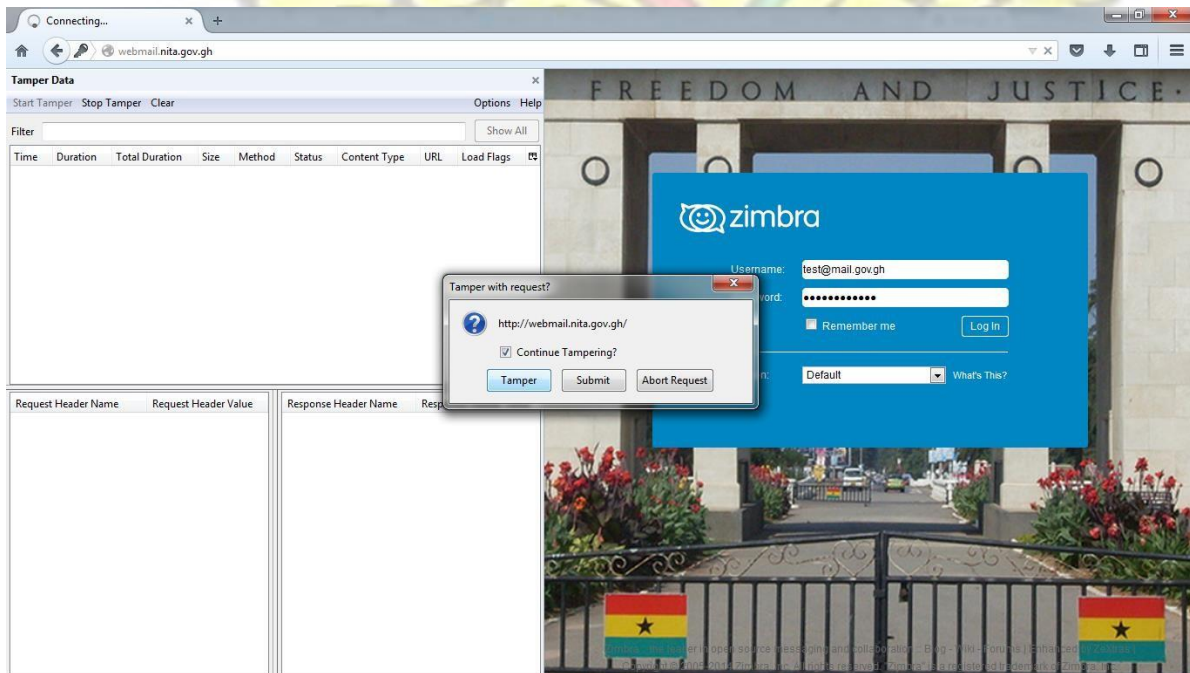


Figure 6.1 Security Testing of the User Interface

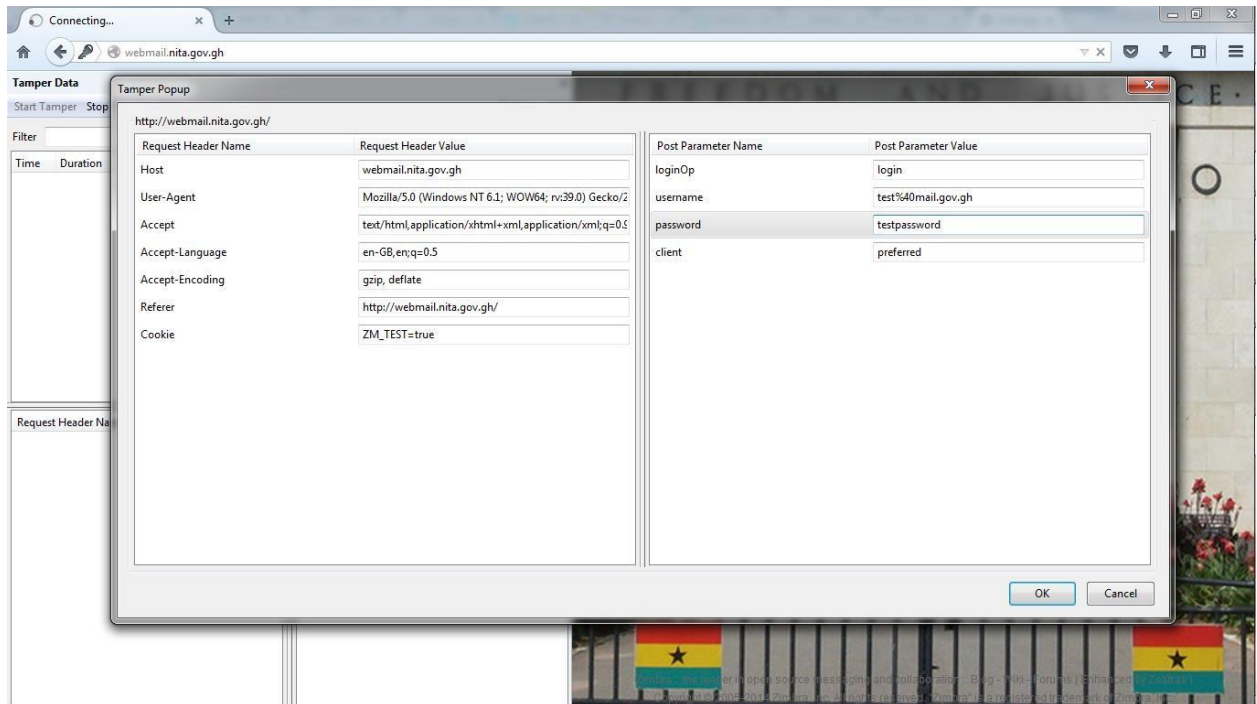


Figure 6.2 Tamper Popup

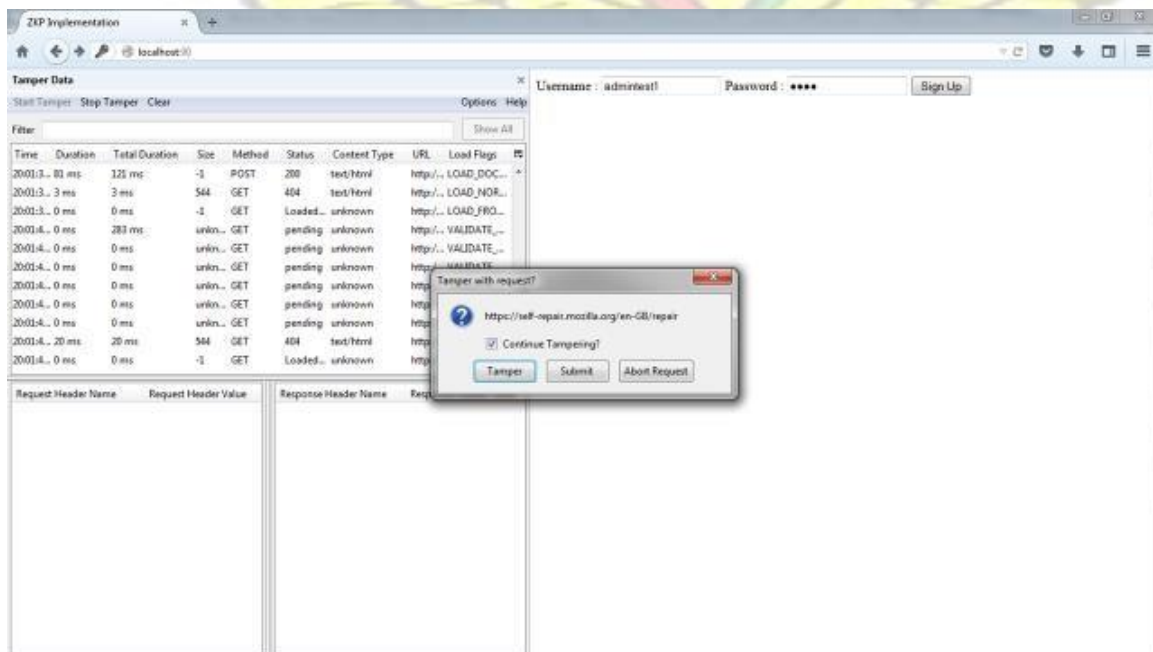


Figure 6.3 Tamper with request

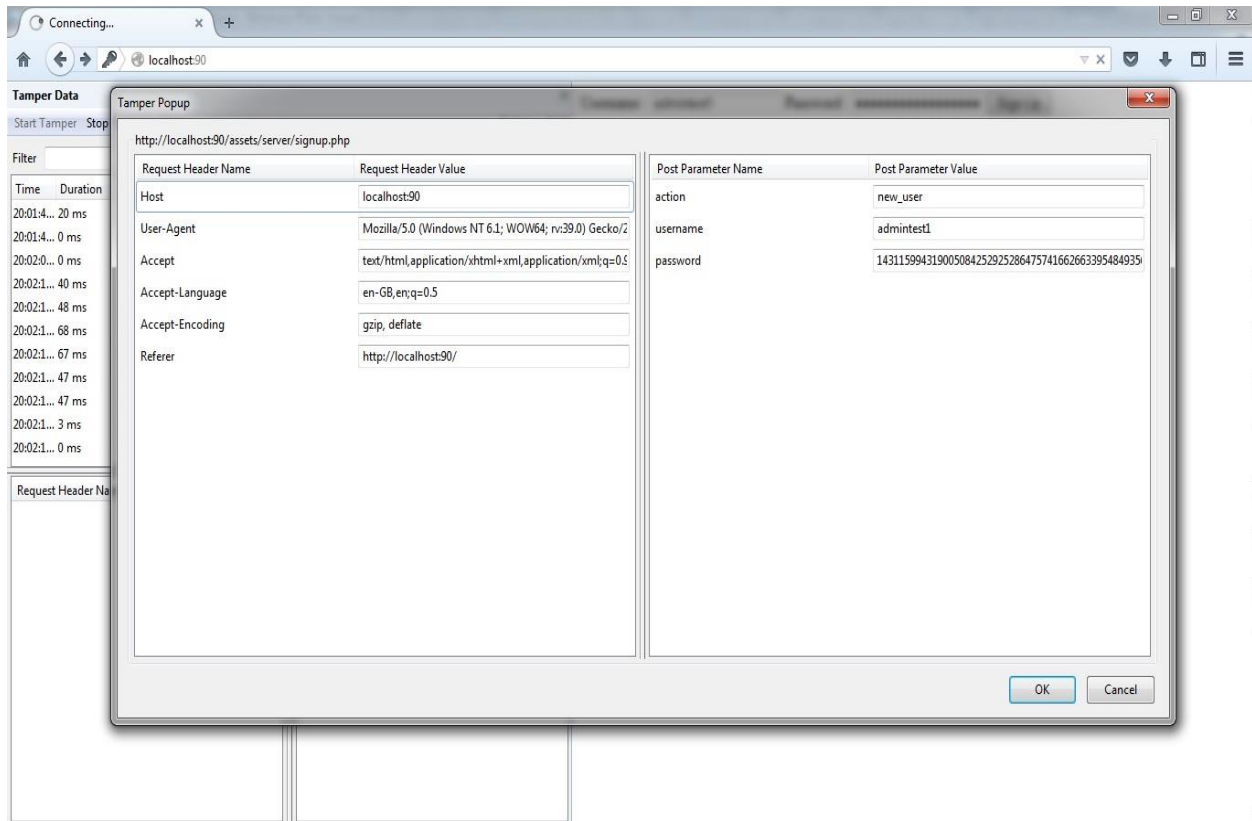


Figure 6.4 Tamper Popup with encrypted Password

The results show that, the actual password is hidden even after using software to intercept the http form request as this Man-in-the-Middle attack makes several protocols authentications vulnerable. A sample code of the program that implements the Stealth Obfuscation zero knowledge Authentication is also as shown in figure 6.5.




```

// JavaScript snippet
function login() {
    $('#login').submit(function() {
        var p = random();
        var g = new BigInteger("2", 10);
        var username = ($('#login_username').val());
        var password = ($('#login_password').val());
        // ... more code ...
    });
}

// PHP snippet
<?php
require "zxp.php";
$username = $_POST['username'];
$db = mysqli_connect('localhost', 'root', '', 'zxp');
$result = mysqli_query($db, "select password from account where username = '" . $username . "'");
$password = "";
while($row = mysqli_fetch_assoc($result)) {
    $password = $row['password'];
}
$sc = $_POST['c'];
$z = $_POST['z'];
var_dump($POST);
$zxp = new Zxp($password, $sc, $z);

// C# snippet
utf8text += String.fromCharCode((c & 63) | 128);
} else {
    utf8text += String.fromCharCode((c >> 12) | 224);
    utf8text += String.fromCharCode((c >> 6) & 63) | 128);
    utf8text += String.fromCharCode((c & 63) | 128);
}
return utf8text;

function binb2hex(binarray) {
    var hex_tab = hexcase ? "0123456789ABCDEF" : "0123456789abcdef";
    var str = "";
    for (var i = 0; i < binarray.length + 4; i++) {
        str += hex_tab.charAt((binarray[i >> 2] >> 4) + 8);
        str += hex_tab.charAt((binarray[i >> 2] >> 0) + 8);
    }
    return str;
}

// Java snippet
if ("c").val().length() > 0 {
    if ("z").val().length() > 0 {
        if ("y").val().length() > 0 {
            if ("login_password").val().length() > 0 {
                // ... more code ...
            }
        }
    }
}

```

Figure 6.5 Code snippet

- ✓ Scalability Testing
- ✓ CPU Utilization

Sysbench

SysBench is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load. After the prototype was taken through the test, the performance was appreciate compared with the generic kernels. The statistics as shown in figure 6.6.

CPU SYSTEM BENCHMARK

```
engineer@TrendOS:~/Downloads$ sysbench --test=cpu --num-threads=4 --cpu-max-prim
e=9999 run
Threads started!
Done.

Maximum prime number checked in CPU test: 9999

Test execution summary:
total time: 10.0260s
total number of events: 10000
total time taken by event execution: 40.0825
per-request statistics:
  min: 0.93ms
  avg: 4.01ms
  max: 46.77ms
  approx. 95 percentile: 15.35ms

Threads fairness:
events (avg/stddev): 2500.0000/7.25
execution time (avg/stddev): 10.0206/0.00
```

Figure 6.6 CPU System Benchmark

DD Benchmark

The dd benchmark command was carried out on the prototype while CPU stress testing benchmarks were still on going. The essence of the DD command is to provide simple sequential I/O performance measurements. The report of the analysis showed an appreciable output compared with the generic values as shown in figure 6.7 and figure 6.8.

DD BENCHMARK

```
engineer@Trend05:~$ cat /dev/sda1 | pipebench -q > /dev/null
cat: /dev/sda1: Permission denied
Summary:
Piped 0.00 B in 00h00m00.00s: 0.00 B/second
engineer@Trend05:~$ sudo cat /dev/sda1 | pipebench -q > /dev/null
[sudo] password for engineer:
Summary:
Piped 18.99 GB in 00h00m57.00s: 341.29 MB/second
engineer@Trend05:~$ sudo cat /dev/sda | pipebench -q > /dev/null
Summary:
Piped 20.00 GB in 00h00m49.96s: 409.85 MB/second
engineer@Trend05:~$ sudo cat /dev/sda5 | pipebench -q > /dev/null
Summary:
Piped 1022.00 MB in 00h00m03.29s: 310.27 MB/second
engineer@Trend05:~$ dd bs=16k count=102400 oflag=direct if=/dev/zero of=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 6.42428 s, 261 MB/s
engineer@Trend05:~$
```

Figure 6.7 DD Benchmark

DD CPU

```
engineer@Trend05:~/Downloads$ dd bs=16k count=102400 oflag=direct if=/dev/zero of=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 4.89916 s, 342 MB/s
engineer@Trend05:~/Downloads$ cat /dev/sda3 | pipebench -q > /dev/null
cat: /dev/sda3: No such file or directory
Summary:
Piped 0.00 B in 00h00m00.00s: 0.00 B/second
engineer@Trend05:~/Downloads$ dd bs=16k count=102400 oflag=direct if=/dev/zero of=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 5.26785 s, 318 MB/s
engineer@Trend05:~/Downloads$ dd bs=16K count=102400 iflag=direct if=test_data of=/dev/null
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 3.79158 s, 442 MB/s
```

Figure 6.8 DD CPU

IPERF CPU

The network speed test of the architecture was also tested using the IPERF benchmark tool. The output of the analysis produced a very impressive output where the architecture could generate between 26.7GBytes to 30.9 GBytes of bandwidth as shown in figure 6.9.

IPERF CPU 2

```
engineer@TrendOS:~/Downloads$ iperf -s -p 8000
-----
Server listening on TCP port 8000
TCP window size: 85.3 KByte (default)
-----
[ 4] local 127.0.0.1 port 8000 connected with 127.0.0.1 port 44452
[ ID] Interval      Transfer    Bandwidth
[ 4]  0.0-10.0 sec  26.7 GBytes 22.9 Gbits/sec
[ 5] local 127.0.0.1 port 8000 connected with 127.0.0.1 port 44454
[ 5]  0.0-10.0 sec  30.9 GBytes 26.6 Gbits/sec
```

Figure 6.9 IPERF CPU 2

With these parameters in mind, a lab was setup to analyze the architecture using a TrendOS prototype which has the Convolved Kernel Architecture (CKA) up against a Generic Kernel Architecture (GKA) to identify the performance level between the two. The GKA is a bare kernel architecture configured in the same manner as the TrendOS to run same applications and services (Srinivasan et al., 2009, Mallet et al, n.d.).

6.1.1 Scalability Testing

Scalability testing was carried out to determine the maximum user load the various programming applications could support. In view of this, the prototype architecture was taken through series of scalability testing. The initial result shows that, as more applications are added to the server, CKA

improves on performance as compared with the GKA which comparatively declines very sharply (Butler-Kisber, 2013).

The summary of the experiment as shown in figure 6.10

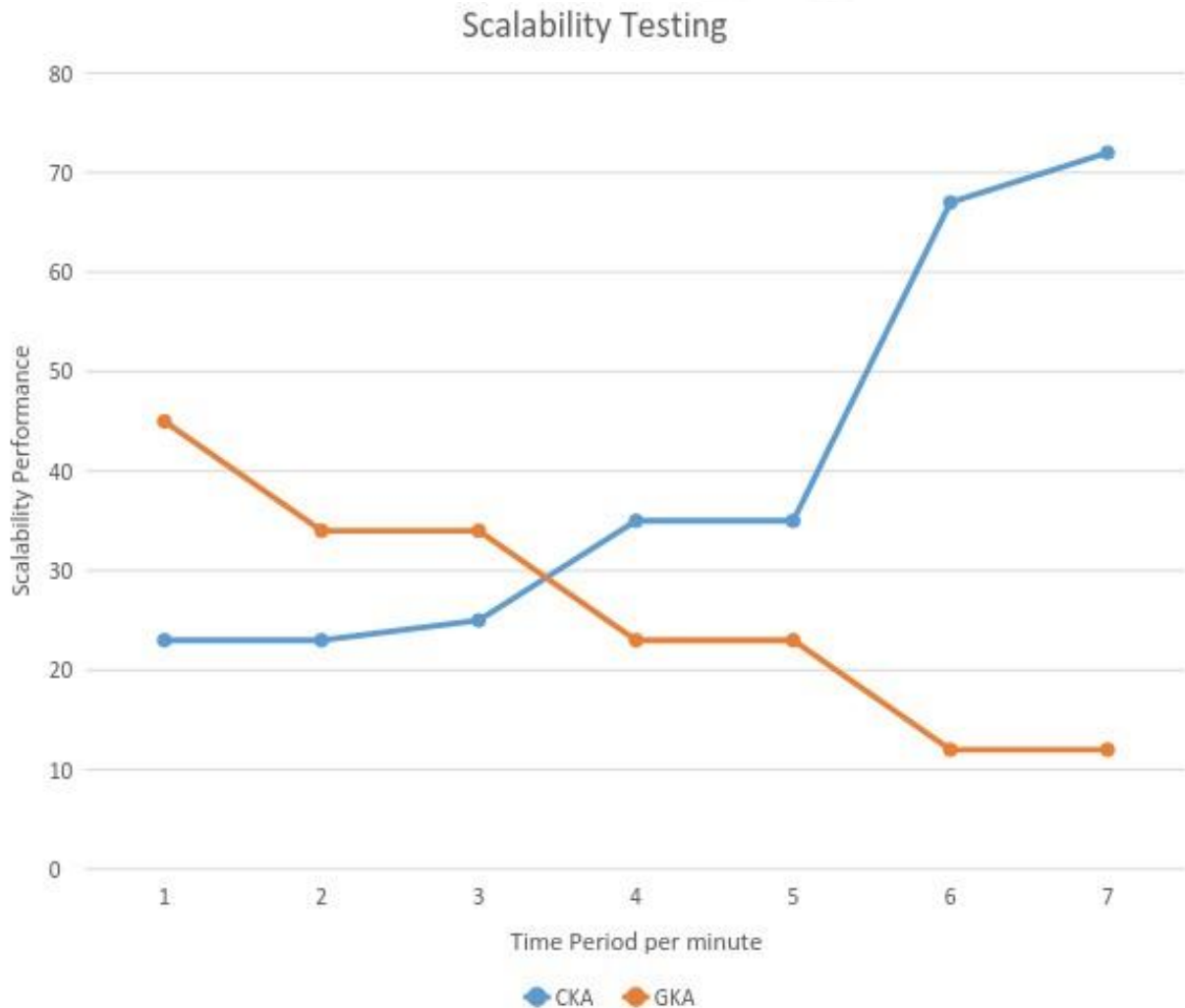


Figure 6.10 Scalability Testing CKA and GKA

.1.2 CPU Utilization

The CPU's resource processing unit of the architecture was also tested. The benchmark technique used to gauge the system performance as the tasks assigned to the system increases. The result from the two architecture also shows a comparative reduction in of CKA CPU utilization as compared with the GKA. The result is shown in Figure. 6.11.

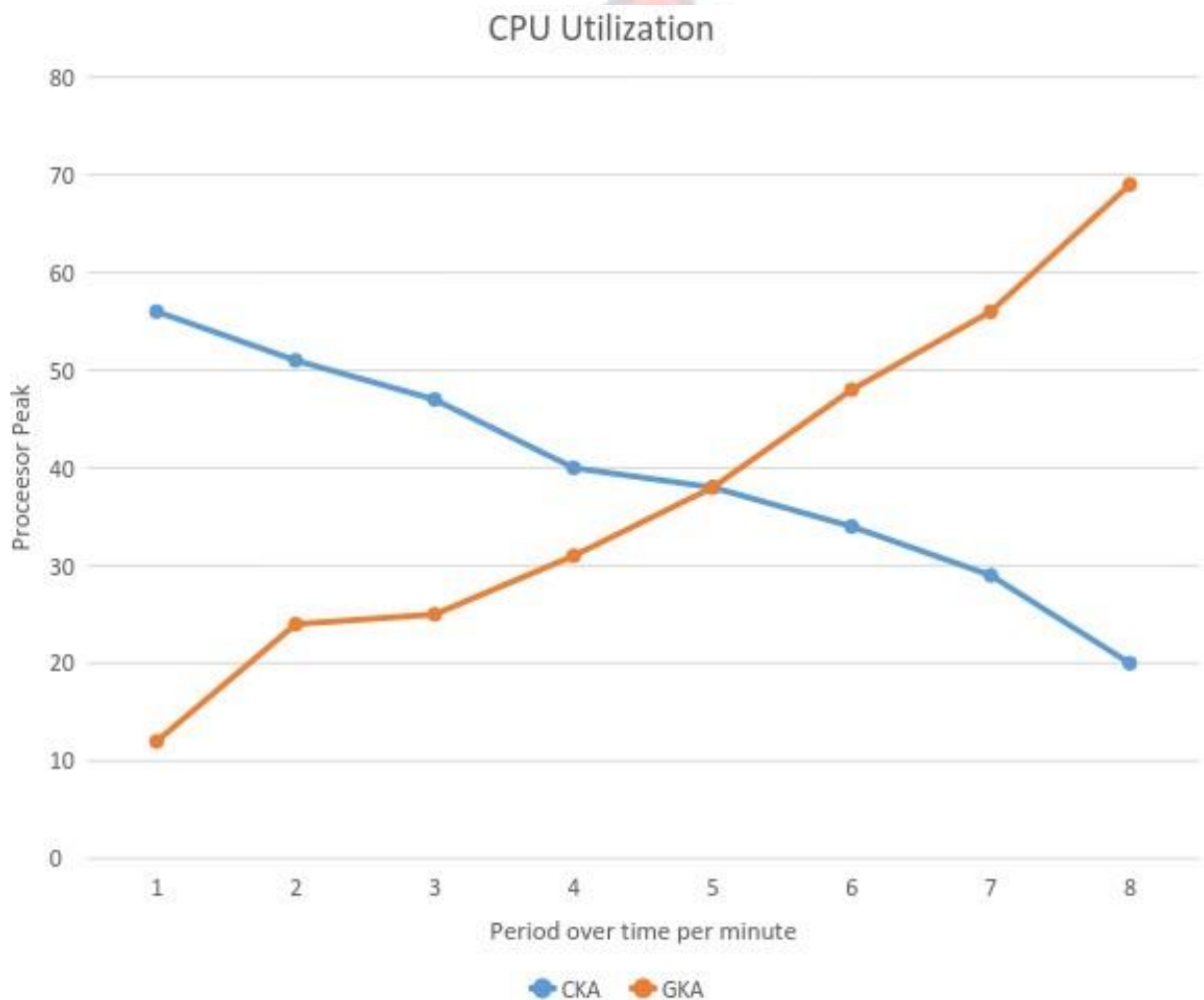


Figure. 6.11 CPU Utilization performance of CKA and GKA

.1.3 Memory utilization

With respect to memory utilization, the initial memory required to read the architecture is usually greater than the other generic kernel. However, the results shows that, the system becomes almost equivalent and even more efficient at a later stage as compared with other generic Linux kernel as shown in Figure. 6.12

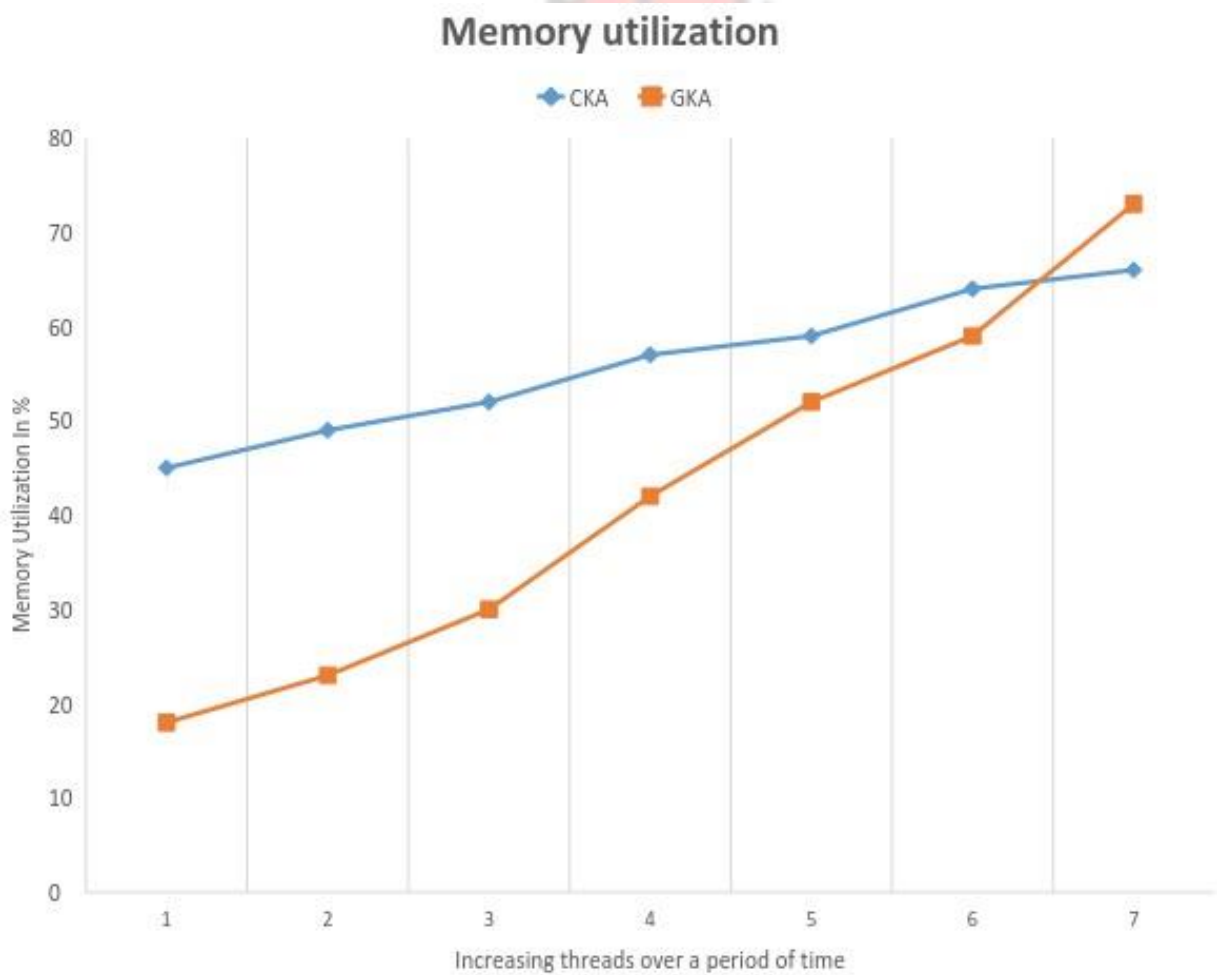


Figure. 6.12 Memory Utilization performance between CKA and GKA

.1.4 Operating System Limitations

The key advantage of this architecture over most other architecture is that, the system is able function more efficiently under a certain amount of threshold persistently. However, when the server is idle, it rather use more resources compared with other generic architecture (Welsh et al, 2003). The result is as shown in figure 6.13

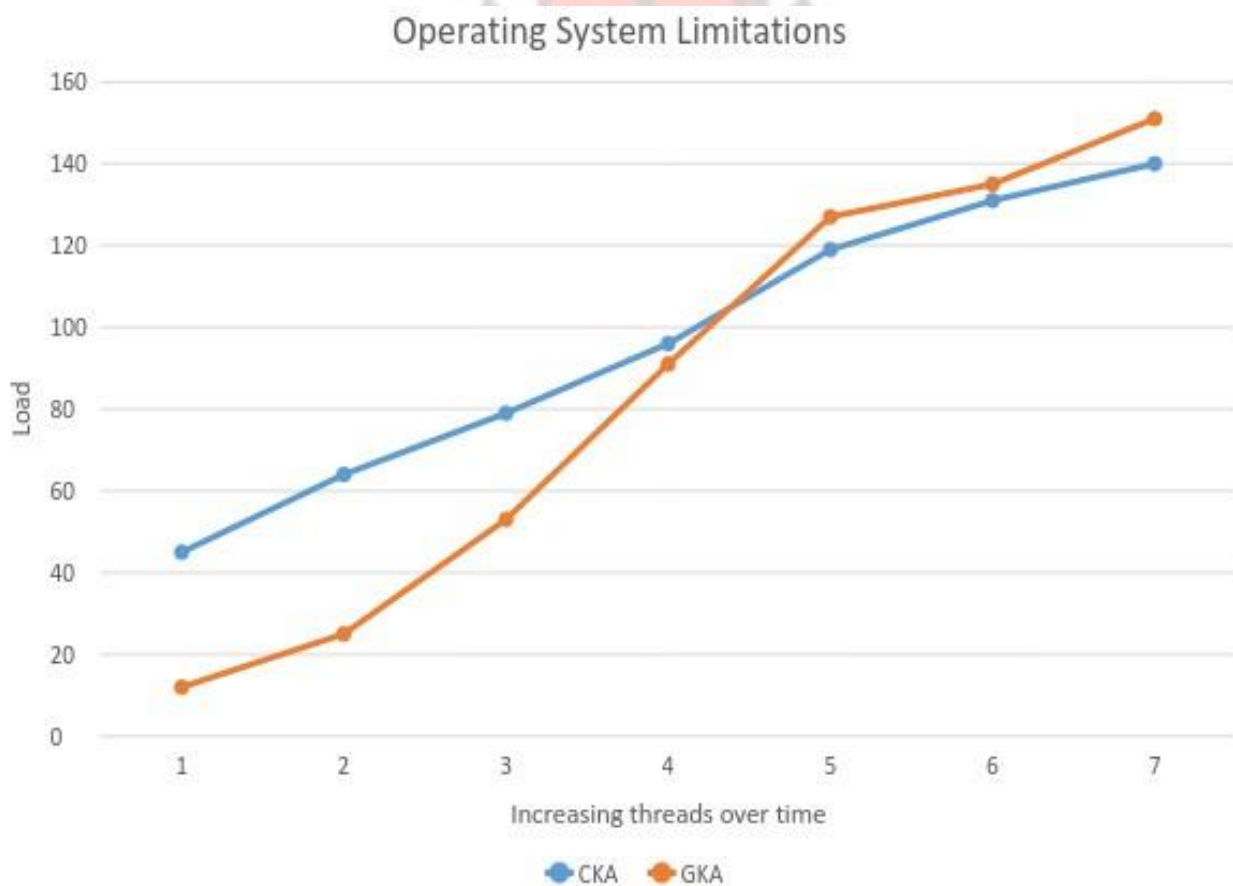


Figure. 6.13 Operating System Limitation between CKA and GKA

1.5 Virtualization Techniques

A comparative study of the various virtualization techniques adopted to identify which one is more efficient shows that, the use of virtual private servers in our framework seem to be the best option as compared with the Linux Kernel's own internal Kernel Virtual Machine (KVM) technology and even worse when implemented on the Hypervisor Virtual Machine (HVM) (Burgess et al., 2009).

Therefore it is concluded that the convoluted architecture's framework is more efficient as compared with the other two virtualization technologies as shown in figure 6.14

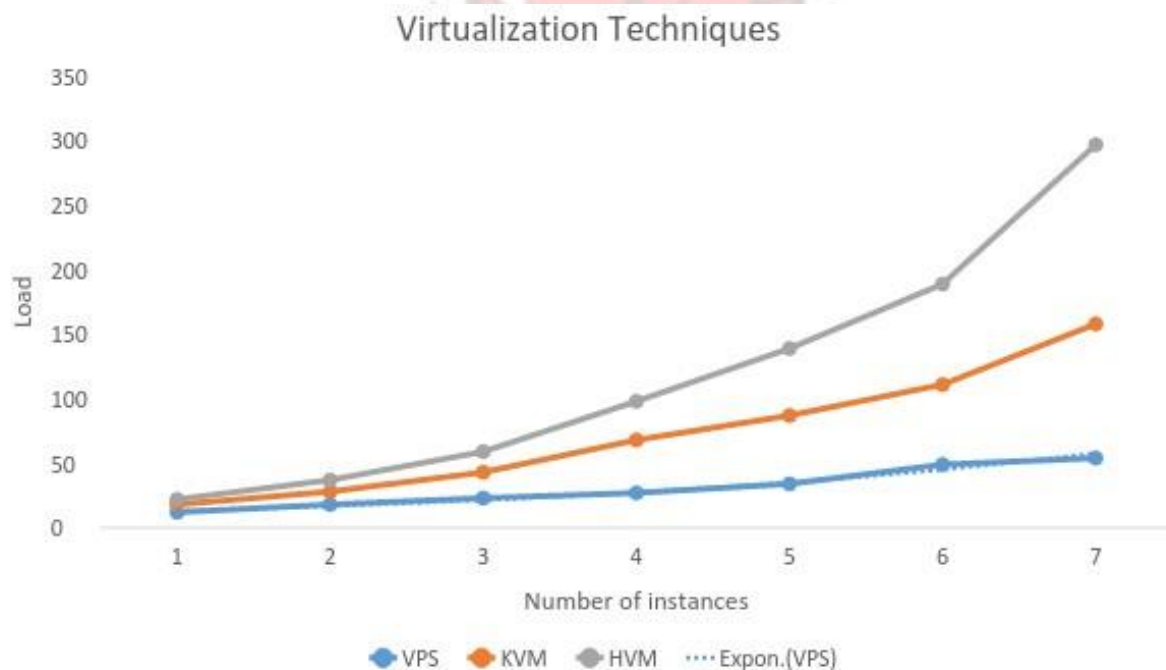


Figure. 6.14 Forms of Virtualization Techniques available against the method adopted

6

6.1.6 Performance Throughout

The server level throughput was considered to be more efficient with shorter response time when the load on the server was increased as compared with the other generic kernels as shown in figure 6.15



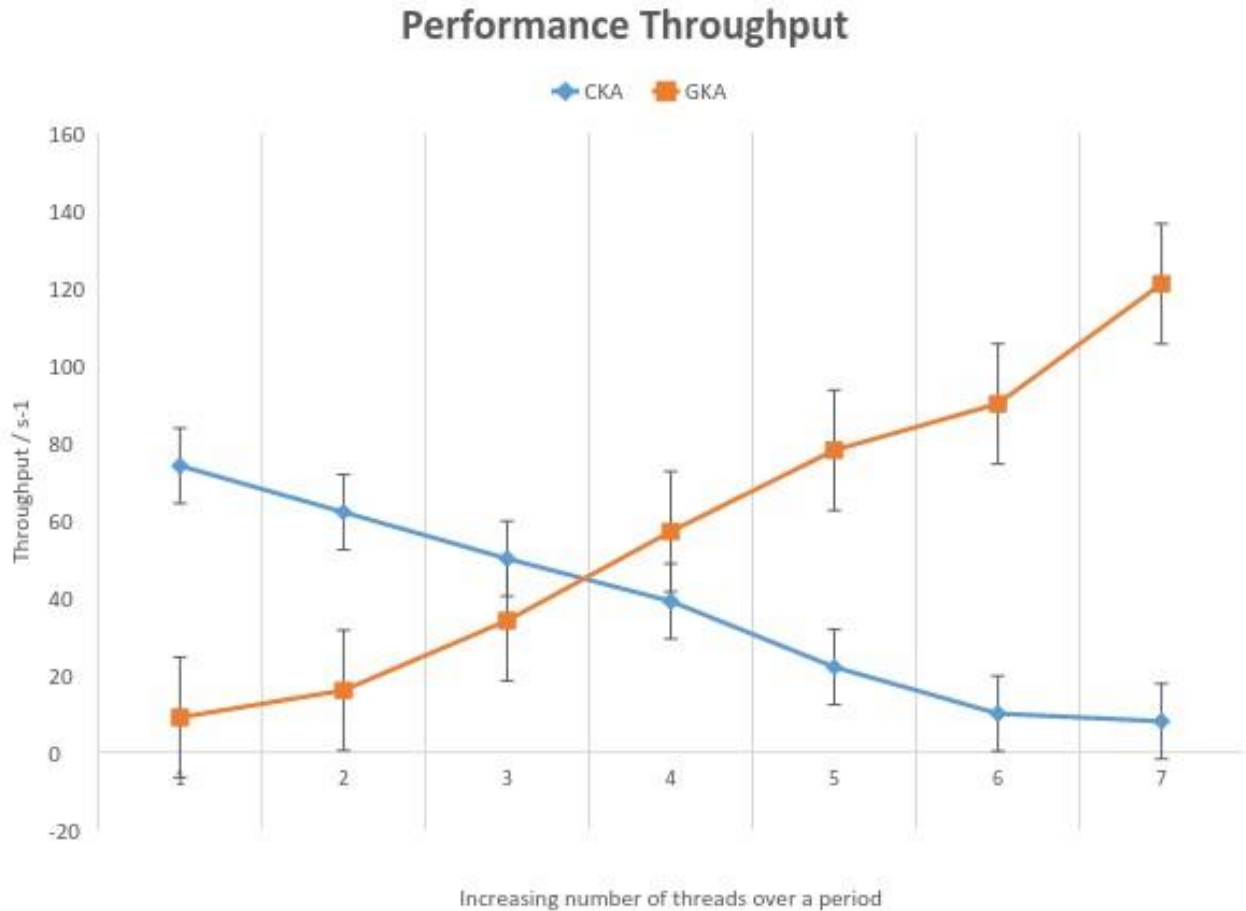


Figure. 6.15 Performance Throughput of CKA against GKA

6.2 Further Performance Analysis

Further analyzes on the architecture also found that, the security of the architecture is further enhanced in several modes. One of such areas has to do with the authentication level of the architecture. The Stealth-Obfuscation Zero Knowledge Authentication algorithm as elaborated in chapter four is retrofitted into the Linux's own Portable Authentication System (PAM) which forms an interface layer for the entire utilities and applications which is attempts to access services from the kernel. This helps address the challenges of remote scripts codes that attempts to steal passwords and PIN's from browsers because of the protection mechanisms it provides (Linux et al, n.d.).

The architecture creates and runs its services on virtual services that when the system detects has been compromised, is able to restore itself and any attack on the server does not affect the core kernel of the base architecture. The High Availability technique also ensures that through the heartbeat architecture, runs a parallel service which guarantees availability of the service at all times (Platform, 2015).

6.3 Summary of the Experiments

The experiments show a good performance of the Convuluted Architecture as against the other generic kernels only when the load on the server is high. The explanation to these results is as a result of the fact that the system runs other services in the background to guarantee availability of services at all times. It is also realized that multiple virtual services are easily created after the initial ones which supports any increase in the loads whenever it becomes indispensable to have the system running at optimal capacity.

CHAPTER 7

DISCUSSION AND CONCLUSION

7.0 DISCUSSION

7.1 CONTRIBUTION TO KNOWLEDGE

- The design of a novel OS Kernel Architecture to enhance the security of the core kernel against internal threats and attacks.
- The use of a kernel integrated UTM within an OS level virtualization.

- To identify the extent to which the performance of monolithic kernels could be improved via the use of High Availability (HA) techniques and OS level virtualization.
- The development of the Stealth Obfuscation Zero Knowledge Proof to be integrated in the PAM.
- The design of a prototype Operating System to integrate the Kernel Architecture to simulate the model.

7.2 CONCLUSION

This thesis presented the convoluted Kernel Architecture as a new framework in monolithic kernel development approach. This framework which provides a robust security and protection mechanism in ensuring that kernel is built on two principal approach on security and availability. The architecture also addresses the challenges of malware that exploits a kernel should in case all other approaches fail. This leads to a novel area of kernel development referred to as intra-kernel protection mechanism. The research also creates a highly modified version of the zero knowledge authentication referred to as the “Stealth-Obfuscation Zero Knowledge Proof Algorithm”. This algorithm enforces itself on all applications within the commodity operating system that requires some level authentication during their execution to provide a secured environment.

In as much as it could be argued that, some level of architecture modification has been developed over the past decades, it is equally clear that in most instances, some significant improvement in the overall security of the architecture is usually witnessed. This work even goes far by not just enhancing the security of the architecture but also integrating the framework with Virtual Private Service architecture to provide High Availability technique as part of its core operation. Its original goal is to enhance the performance of servers during their peak. The architecture improves the load balancer of the kernel integrated into its framework.

One of the novel techniques in this architecture that makes it even more innovative is its integration with Unified Threat Management (UTM) which has been hitherto positioned at a gateway section of a network. The findings of the Kernel integrated UTM showed that the internal architecture is

even more protected from applications running on the operating system as well as the protection of the Kernel against itself (Welsh et al., 2003).

Further study of the various Kernel architectures as well as Frameworks were also investigated. Even though most of them improved the existing security of those architecture, they however did not allow other security modules to run concurrently. Many of these security modules were only focused on the Mandatory Access Control (MAC) and moreover did not concurrent on other security options. The proposed architecture however, allows concurrent execution of the diverse security modules and also focused on High Availability to ensure the server is not easily attacked by Denial of Service attack (Kurmus, et al, 2011).

Finally, the various benchmark tools implemented on the prototyped Convolutated Kernel Architecture as well as other generic kernels suggested that, the proposed system works more optimally when the server is kept busy over a period of time as compared with the other kernels. However, when the system is kept in idle state, the performance is not optimal and therefore is recommended on servers.

7.3 Future work

Even though systems and network engineers are mostly interested in servers that could have good performance especially when the load is very high, it was however unexpected that the system rather performs more efficiently when the load is rather high and almost same or less when the load is at its normal interaction. It is however envisaged that future work must look at how the system could switch between two modes that could support an idle server environment and switch back to the proposed mode when the server network becomes busy.

KNUST



References

- Alm, C. (2006). Analysis of Manipulation Methods in Operating System Kernels and Concepts of Countermeasures, Considering FreeBSD 6 . 0 as an Example Diploma Thesis.
- Assurance, I., & Report, T. (2011). Information Assurance Tools Report. at-commerce-7. (n.d.).
- Babai, L. (1985). Trading group theory for randomness. *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Coputing (STOC'85)*, 421–429.
<http://doi.org/10.1145/22145.22192>
- Backes, P. M., & Bugiel, M. S. S. (2013). Android Security Lab WS 2013 / 14 Lab 4 : Android Kernel Extension Login on Lab Machines, 0, 1–10.
- Badger, L., Sterne, D. F., Sherman, D. L., & Walker, K. M. (1995). A domain and type enforcement UNIX prototype. *Proceedings of the 5th Conference on USENIX UNIX Security Symposium-Volume 5*, 9(1), 12–12. <http://doi.org/10.1109/SECPRI.1995.398923>
- Ben-Cohen, O. (Tel-A. U., & Wool, A. (Tel-A. U. (2008). Proceedings of the Linux Symposium. *Proceedings of the 2008 Linux Symposium*, 31–38.
<http://doi.org/http://ftp.kernel.org/pub/linux/kernel/people/lenb/acpi/doc/OLS2006ondemand-paper.pdf>
- Biondi, P. (2002). Security at Kernel Level: {LIDS}. *Fosdem 2002*.
- Biondi, P. (2003). Kernel Level Security. *Context*, (September), 1–23.
- Bishop, M. (2009). Reflections on UNIX vulnerabilities. *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 161–184.
<http://doi.org/10.1109/ACSAC.2009.25>
- Bittau, A. (2009). Toward Least-Privilege Isolation for Software, (November).
- Boneh, D. (2004). Punch card device Punched card other alternatives Touch-screen voting How votes are cast Problems with electronic voting Voter Verified Audit Trail. *Ieee Spectrum*, 1-

- Bowman, I. (1998). Conceptual Architecture of the Linux Kernel For Ric Holt List of Figures, 1–13.
- Brinkley, D. L., & Schell, R. R. (n.d.). Concepts and Terminology for Computer Security, 40–98.
- Bugiel, S., Heuser, S., & Sadeghi, A.-R. (2013). Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. *Proceedings of the 22nd USENIX Security Symposium*, 131–146. Retrieved from <http://dl.acm.org/citation.cfm?id=2534766.2534778>
- Burgess, M., Cardaci, L., Farrow, R., Hewson, J., Littell, R., Plonka, D., ... Yue, C. (2009). *LISA '09: 23rd Large Installation System Administration Conference. International Journal of Water Resources Development* (Vol. 17).
- Butler-Kisber, L. (2013). Teaching and Learning in the Digital World: Possibilities and Challenges. *LEARNing Landscapes*, 6(2), 424.
- Cheu, R., Yang, P., Lin, A., & Jaffe, A. (2014a). Massachusetts Institute of Technology An Implementation of Zero Knowledge Authentication.
- Cheu, R., Yang, P., Lin, A., & Jaffe, A. (2014b). Massachusetts Institute of Technology An Implementation of Zero Knowledge Authentication. Retrieved from <https://courses.csail.mit.edu/6.857/2014/files/15-cheu-jaffe-lin-yang-zkp-authentication.pdf>
- Chip, A., Movidius, M., Note, G., Bristol, G., Apus, R., Why, R., ... Recall, D. T. (2016). 1 Intel to Acquire Chip Maker Movidius, 7.
- Cisco. (2013). Cisco Application Centric Infrastructure, (Figure 1), 2014–2016. Retrieved from <http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-seriesswitches/at-a-glance-c45-730001.pdf>
- Control, A. (2007). Access Control (1), (1), 2–4.
- Corbet, J. (2008). How to participate in the linux community. *Ht Tp://Ldn. Linuxfoundation*.

Org/Book/How-Participate-Linux-Community, (August). Retrieved from http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf

Cyberspace, M. S. (n.d.). *Seymour E. Goodman and Herbert S. Lin, Editors Committee on Improving Cybersecurity Research in the United States Computer Science and Telecommunications Board Division on Engineering and Physical Sciences.*

Dallas, U. T. (n.d.). Part I Murat Kantarcioglu.

Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., & Adve, V. (n.d.). Nested Kernel : An Operating System Architecture for Intra-Kernel Privilege Separation.

Department of Defense. (1985). Trusted computer system evaluation criteria. *{Department of Defense}*, 1–116. <http://doi.org/DoD 5200.28-STD>

DISEC. (2015). Disarmament and International Security (DISEC) Study Guide.

Engler, D. R., Kaashoek, M. F., & Gould, L. A. (1998). The Exokernel Operating System Architecture by.

Fiege, U., Fiat, a, & Shamir, a. (1987). Zero knowledge proofs of identity. *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, 210–217. <http://doi.org/10.1145/28395.28419>

Finin, T., Joshi, A., Kagal, L., Niu, J., Sandhu, R., Winsborough, W., & Thuraisingham, B. (2008). ROWLBAC - Representing Role Based Access Control in OWL. *Scenario*, 35, 73–82. <http://doi.org/10.1145/1377836.1377849>

Ganapathy, V., Jaeger, T., & Jha, S. (2005). Automatic placement of authorization hooks in the linux security modules framework. *Proceedings of the 12th ACM Conference on Computer and Communications Security - CCS '05*, (November), 330. <http://doi.org/10.1145/1102120.1102164>

Ghogare, S. D., Jadhav, S. P., Chadha, A. R., & Patil, H. C. (2012). Location Based

- Authentication: A New Approach towards Providing Security. *International Journal of Scientific and Research Publications*, 2(1), 2250–3153. Retrieved from www.ijsrp.org
- Goldwasser, S., Micali, S., & Rackoff, C. (1989). The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1), 186–208. <http://doi.org/10.1137/0218012>
- Gorman, M. (2003). Mel Gorman. *Science*, 1–43.
- Grattafiori, A. (2016). Understanding and Hardening Linux Containers. *NCC Group Whitepapers*, (1), 1–122.
- Grzonkowski, S., Zaremba, W., Zaremba, M., & McDaniel, B. (2008). Extending web applications with a lightweight zero knowledge proof authentication. *Proceedings of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology - CSTST '08*, 65. <http://doi.org/10.1145/1456223.1456241>
- Haines, R. (n.d.). The SELinux Notebook The Foundations. *Source*, 1–365.
- Hang, A., Luca, A. De, Smith, M., & Richter, M. (2015). Where Have You Been ? Using Location-Based Security Questions for Fallback Authentication. *Symposium on Usable Privacy and Security*, 169–183.
- Harada, T. (2007). TOMOYO Linux BoF.
- Harada, T., & Handa, T. (2007). TOMOYO Linux: A Lightweight and Manageable Security System for PC and Embedded Linux. *Proceedings of Ottawa Linux Symposium*, 27–30.
- Hellerstein, J. M., Stonebraker, M., & Hamilton, J. (2007). Architecture of a Database System, 1(2), 141–259. <http://doi.org/10.1561/19000000002>
- Holwerda, T. (2007). Kernel designs explained, (March).
- Howard, M., LeBlanc, D., & Viega, J. (2010). *24 Deadly Sins of Software Security. Security Management*.
- Hu, V. C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014).

- Guide to attribute based access control (abac) definition and considerations. *NIST Special Publication*, 800, 162. <http://doi.org/10.6028/NIST.SP.800-162>
- Jaeger, T., Edwards, A., & Zhang, X. (2004). Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security*, 7(2), 175–205. <http://doi.org/10.1145/996943.996944>
- Karlsson, E. (2010). Evaluation of Linux Security Frameworks.
- Katzer, M. (2015). Security Best Practices. *Moving to Office 365*, 95–120. http://doi.org/10.1007/978-1-4842-1197-7_4
- Kim, S. K., Son, H. S., & Han, S. Y. (2008). Design and implementation of security os: A case study. *Computing and Informatics*, 27(6), 931–951.
- Klemm, F. (2001). Security Mechanisms in Distributed Component Models.
- Krátký, R., Prpič, M., Čapek, T., Wadeley, S., Ruseva, Y., & Svoboda, M. (2016). Red Hat Enterprise Linux 6.8 Security Guide. Retrieved from https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Security_Guide/Red_Hat_Enterprise_Linux-6Security_Guide-en-US.pdf
- Kurmus, A., Gupta, M., Pletka, R., Cachin, C., & Haas, R. (2011). A comparison of secure multi-tenancy architectures for filesystem storage clouds. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7049 LNCS, 471–490. http://doi.org/10.1007/978-3-642-25821-3_24
- LaPadula, L. (1995). Rule-Set Modeling of a Trusted Computer System. *ABRAMS M, JAJODIA S, PODELL H. Information Security: An Integrated Collection of Essays. Cambridge: IEEE Computer Society Press*, 187–195. Retrieved from <http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:RuleSet+Modeling+of+a+Trusted+Computer+System#0>
- Leggieri, M., Weth, C. Von Der, Serrano, M., & Nuig, D. (2014). Internet-Connected Objects b.

- Liaropoulos, A., & Tsihrintzis, G. (2014). 13th European Conference on Cyber Warfare and Security. *13th European Conference on Cyber Warfare and Security The University of Piraeus Greece 3-4 July 2014*, (July), 326. <http://doi.org/10.13140/RG.2.1.4331.5281>
- Lindskog, S. (2000). Observations on Operating System Security Vulnerabilities, 1–144.
- Linux, H., & Works, C. (n.d.). How Linux Capability Works in 2.6.25, 1–4.
- Lomte, V. M. (2012). A Secure Web Application: E-Tracking System. *International Journal of UbiComp*, 3(4), 1–18. <http://doi.org/10.5121/iju.2012.3401>
- Mallet, A., Chung, J. D., & Smith, J. M. (n.d.). Operating System Support for Protocol Boosters 1 Introduction 2 Implementation choices and strategy.
- Manual, P. (2011). Section, United nations security management system, (February).
- Manual, R., Protection, A. F., Other, C., Features, F., Services, C., Profiles, Q., & Profiles, B. (n.d.). UTM Basic Firewall Configuration Use Rules to Block or Allow Specific Kinds of Traffic, 1–38.
- Masumoto, K., & Takeda, K. (2007). TOMOYO Linux – Tutorial session, 1–14.
- Mauerer, W. (n.d.). *Linux ® Kernel Architecture*.
- Mauerer, W. (2008). Linux ® Kernel Architecture. *Auditing*, 1368.
- Melrose, J., Perroy, R., & Careas, S. (2016). XSM-Flask. *XenSummit, 1*. <http://doi.org/10.1017/CBO9781107415324.004>
- Merrill, S. (2012). State Of The Linux Kernel 2011 | TechCrunch. Retrieved from <http://techcrunch.com/2012/04/03/state-of-the-linux-kernel-2011/>
- Nteziryayo, C., & Kibe, A. (2015). *International Journal of Advanced Research in Computer Science and Software Engineering*, 5(9), 91–94.
- O’Neil, H. F., Kunkler, K., Friedi, K. E., & Perez, R. S. (2013). Designing and Using Computer Simulations in Medical Education and Training. *Military Medicine*, 178(10). Retrieved from http://publications.amsus.org/pb/assets/raw/Supplements/178_10_Supplement.pdf

- Ott, A., & Fischer-hübner, S. (1995). The “ Rule Set Based Access Control ” (RSBAC) Framework for Linux. *Control*, 1–18.
- Pa, I. T. E. (n.d.). Next Generation Security with VMware ® NSX and Palo Alto Networks ®.
- Papachristodoulou, L., & Antakis, S. (2008). Security-Enhanced Linux in a Health Information System.
- Phones, S. O. (2008). UU Securing Open-Source Phones UU Learn the Path ABERDEEN.
- Platform, S. K. (2015). White Paper: An Overview of Samsung KNOX. *White Paper*, (June).
- Piromsopa, K., & Enbody, R. J. (2011). Survey of protections from buffer-overflow attacks. *Engineering Journal*, 15(2), 31–52. <https://doi.org/10.4186/ej.2011.15.2.31>
- Pourzandi, M., Gordon, D., Yurcik, W., & Koenig, G. A. (2005). Clusters and security: Distributed security for distributed systems. *2005 IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005*, 1, 96–104. <http://doi.org/10.1109/CCGRID.2005.1558540>
- Pourzandi, M., Haddad, I., Levert, C., Zakrzewski, M., & Dagenais, M. (2002). A Distributed Security Infrastructure for Carrier Class Linux Clusters. *Ottawa Linux Symposium*, 439–450. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.8887&rep=rep1&type=pdf#page=439>
- Pourzolfaghar, Z., Bezbradica, M., & Helfert, M. (2011). Types of IT Architectures in Smart Cities – A review from a Business Model and Enterprise Architecture Perspective, *1*(1987). *Proceedings of 6*. (2005), 947649.
- Provos, N. (2003). Improving host security with system call policies. *Proceedings of the 12th Conference on USENIX Security Symposium-Volume 12*, 18–18. <http://doi.org/1251353.1251371>

- Publications, R., & Publications, R. (2008). National Cyber Defense Initiative “ Opening Moves ” Workshop Report, (December 2007).
- Red, S., & Enterprise, H. (n.d.). Red Hat Enterprise Linux 6 . 3 Beta Security Guide.
- Reilly, A. J. (1998). ,, Three Approaches To Organizational Learning. *The Pfeiffer Library*, 16(2).
- Reilly, P. O. (n.d.). *Understanding the Linux Kernel*.
- Safford, D., Zohar, M., & Boulanger, A. (2005). Trusted Computing for Linux, (August).
- Sahl, A. N., Samsudin, A., & Letchmunan, S. (2014). Visual Zero-Knowledge Proof of Identity Scheme by Using Color Images, 21(8), 1188–1196.
<http://doi.org/10.5829/idosi.mejsr.2014.21.08.648>
- Schmidt, A. (1945). *Secrecy Versus Science*. *The Lancet* (Vol. 246).
[http://doi.org/10.1016/S0140-6736\(45\)91080-4](http://doi.org/10.1016/S0140-6736(45)91080-4)
- Schnorr, C. P. (1990). Efficient identification and signatures for smart cards. *Eurocrypt 1989*, 434, 688–689. http://doi.org/10.1007/3-540-46885-4_68
- Scott, M. (n.d.). M-Pin: A Multi-Factor Zero Knowledge Authentication Protocol.
- Shahapure, N. H. (2015). Replication : A Technique for Scalability in Cloud Computing, 122(5), 13–18.
- Shan, Z. (2011). Virtualizing System and Ordinary Services in Windows-based OS-Level Virtual Machines, 579–583.
- Snyder, J., & One, O. (n.d.). Evaluating Unified Threat.
- Soares, L. F. B. (2013). Secure authentication mechanisms for the management interface in cloud computing environments, (October). Retrieved from [Secure authentication mechanisms for the management interface in cloud computing environments.pdf](#)
- Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., & Lepreau, J. (1999). THE FLASK SECURITY ARCHITECTURE : The Flask Security Architecture : System Support for Diverse Security Policies.

Spengler, B. (2003). Grsecurity - ACL Documentation v1.5, 1–29.

Spengler, B., & Security, O. S. (2012). The Case For.

Srinivasan, S. R., Lee, J. W., Liu, E., Kester, M., Schulzrinne, H., Hilt, V., ... Khan, A. (2009).

NetServ: dynamically deploying in-network services. *Proceedings of the 2009 Workshop on Re-Architecting the Internet*, 37–42. <http://doi.org/10.1145/1658978.1658988>

Stallings, W., & Hall, P. (2008). *T l o s*.

SUSE Linux Enterprise Server 10 SP1 EAL4 High-Level Design. (n.d.).

Thion, R. (2008). Access Control Models. *Cyber Warfare and Cyber Terrorism*, 1–28. Retrieved from

<http://books.google.com/books?hl=en&lr=&id=XWK9AQAQBAJ&oi=fnd&pg=PT349&dq=Access+Control+Models+Thion+Romuald&ots=26ZIC8tpnp&sig=EuFy4JSZSzJpEVyuhHYItrPYBS0%5Cnpapers2://publication/uuid/C7D20B60-DDBA-40FB-A2DFA3068B4544F9>

Universit, P. (2010). Measuring the Semantic Integrity of a Process Self. *Informatica*.

Val, D., & Marco-gisbert, H. (2016). Universitat Polit e CyberSecurity research group Exploiting Linux and PaX ASLR ' s weaknesses on 32- and 64-bit systems. *Black Hat*.

Wang, Z. (2012). Securing Virtualization: Techniques and Applications.

Watson, R. N. M. (2012). System Security Extensibility, (818).

Watson, R. N. M. (2013). A Decade of OS Access-control Extensibility. *Commun. ACM*, 56(2), 52–63. <http://doi.org/10.1145/2408776.2408792>

Watson, R. N. M., Chisnall, D., Davis, B., Koszek, W., Moore, S. W., Murdoch, S. J., ...

Woodruff, J. (2014). Capability Hardware Enhanced RISC Instructions: CHERI User's guide, (851).

Welsh, M., Culler, D., & Brewer, E. (2003). SEDA : An Architecture for Well-Conditioned , Scalable Internet Services. *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03)*, 14. <http://doi.org/10.1145/502059.502057>

- Works, H. S. M. (2000). Set-UID Privileged Programs Vulnerabilities of Set-UID Programs, 1–9.
- Wright, C., Cowan, C., Labs, N. A. I., Associates, N., Morris, J., & Kroah-hartman, G. (n.d.). Linux Security Module Framework, 8032.
- Wright, C., Cowan, C., & Morris, J. (2002). Linux security module framework. *Ottawa Linux Symposium*, 8032. Retrieved from http://courses.cs.vt.edu/~cs5204/fall05gback/papers/ols2002_proceedings.pdf#page=604
- Wright, C., Cowan, C., Morris, J., Smalley, S., & Kroah-Hartman, G. (2003). Linux security modules: General security support for the Linux Kernel. *Foundations of Intrusion Tolerant Systems, OASIS 2003*, 8032, 213–226. <http://doi.org/10.1109/FITS.2003.1264934>
- Yang, C.-T., Chou, W.-L., Hsu, C.-H., & Cuzzocrea, A. (2011). On Improvement of Cloud Virtual Machine Availability with Virtualization Fault Tolerance Mechanism. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 122–129. <http://doi.org/10.1109/CloudCom.2011.26>
- Yu, Y., & Science, C. (2007). OS-level Virtualization and Its Applications, (December).
- Zakrzewski, M. (2002). Mandatory Access Control for Linux Clustered Servers. *Linux Symposium*, 22 Suppl 4, 1–16. Retrieved from <http://www.ncbi.nlm.nih.gov/pubmed/21696489>
- Edge, J. (2012). The return of loadable security modules? [LWN.net]. Lwn.net. Retrieved 23 July 2017, from <https://lwn.net/Articles/526983/>

APPENDIX A

FIGURES

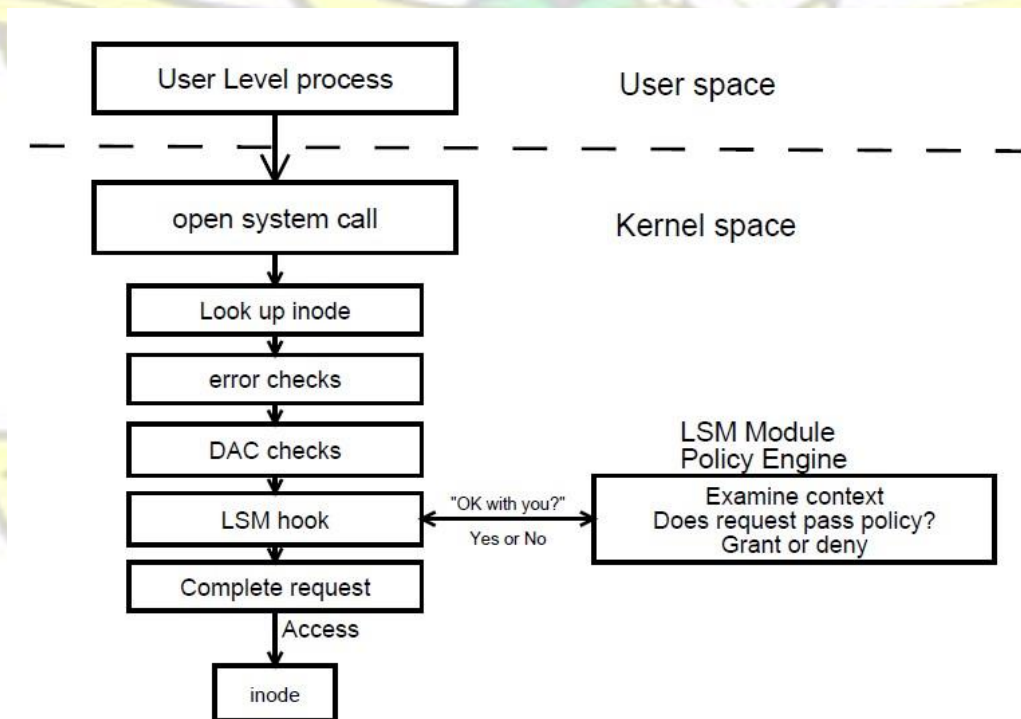


Figure 2.1 Linux security module architecture (Source: Wright et al, 2003)

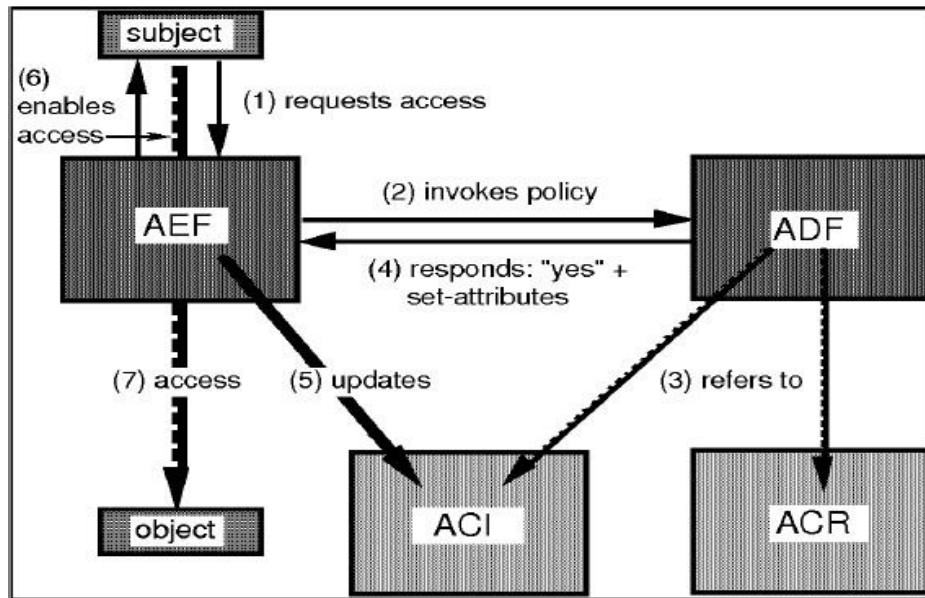


Figure 2.2. Overview of GFAC Architecture (Source: Karlsson, 2010)

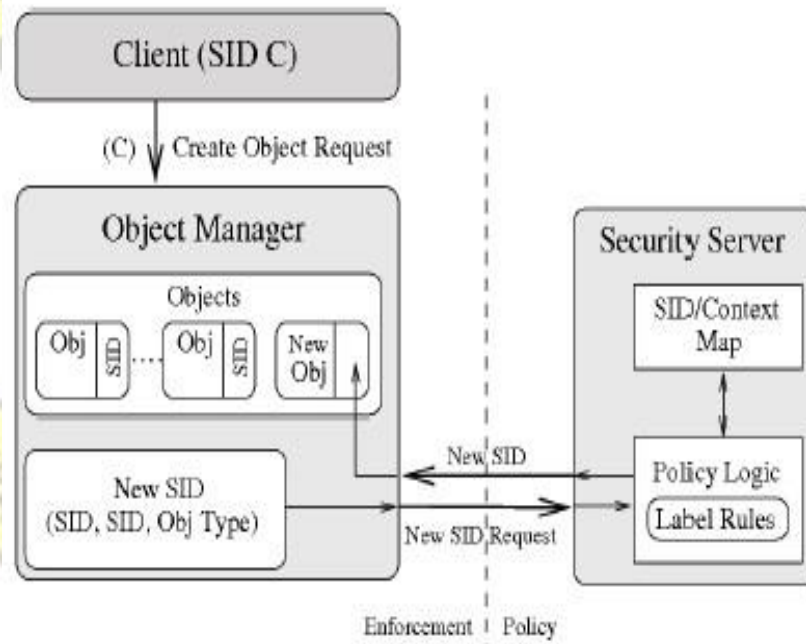


Figure 2.3. Overview of FLASK architecture (Source: Karlsson, 2010)

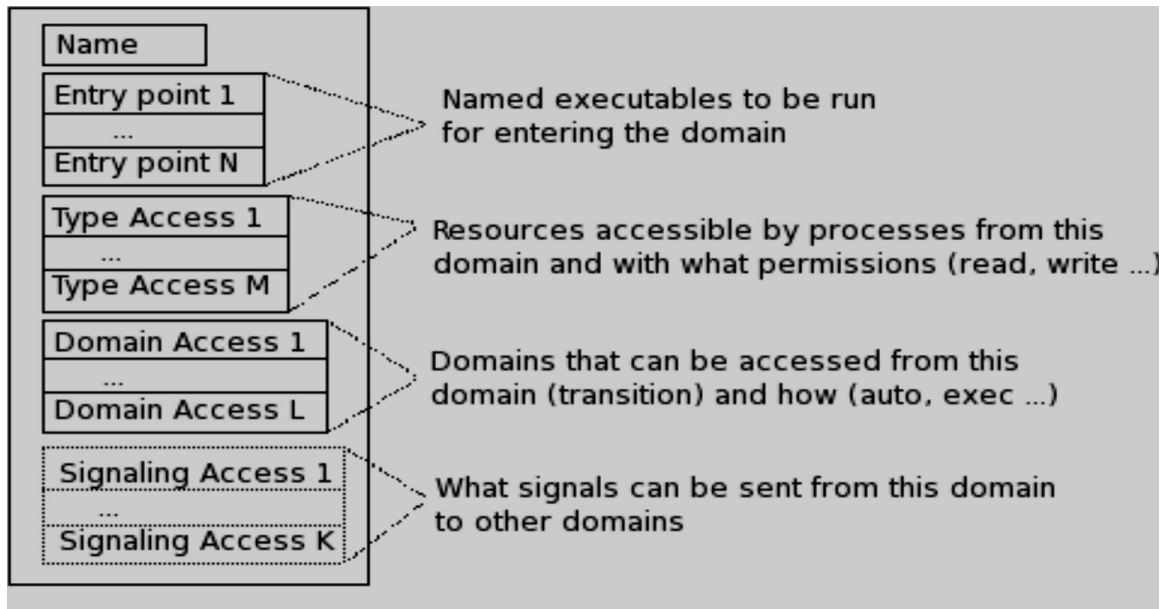


Figure 2.4. Domain and Type Enforcement (Badger et al, 1995)

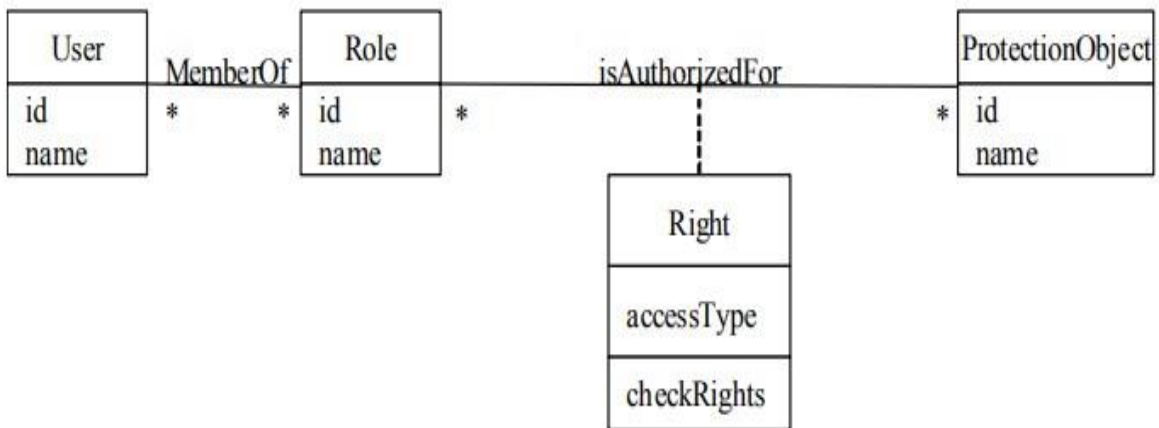


Figure 2.5. Role Based Access Control Source: (Fernandez, Pernul, & Larrondo-Petrie, 2008)

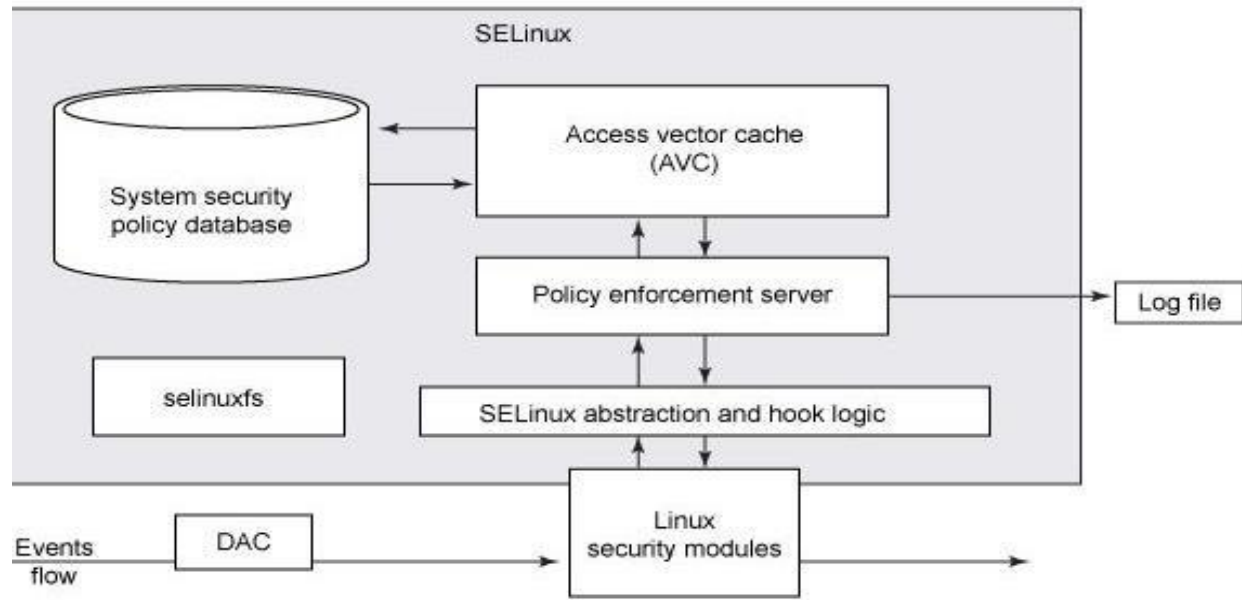
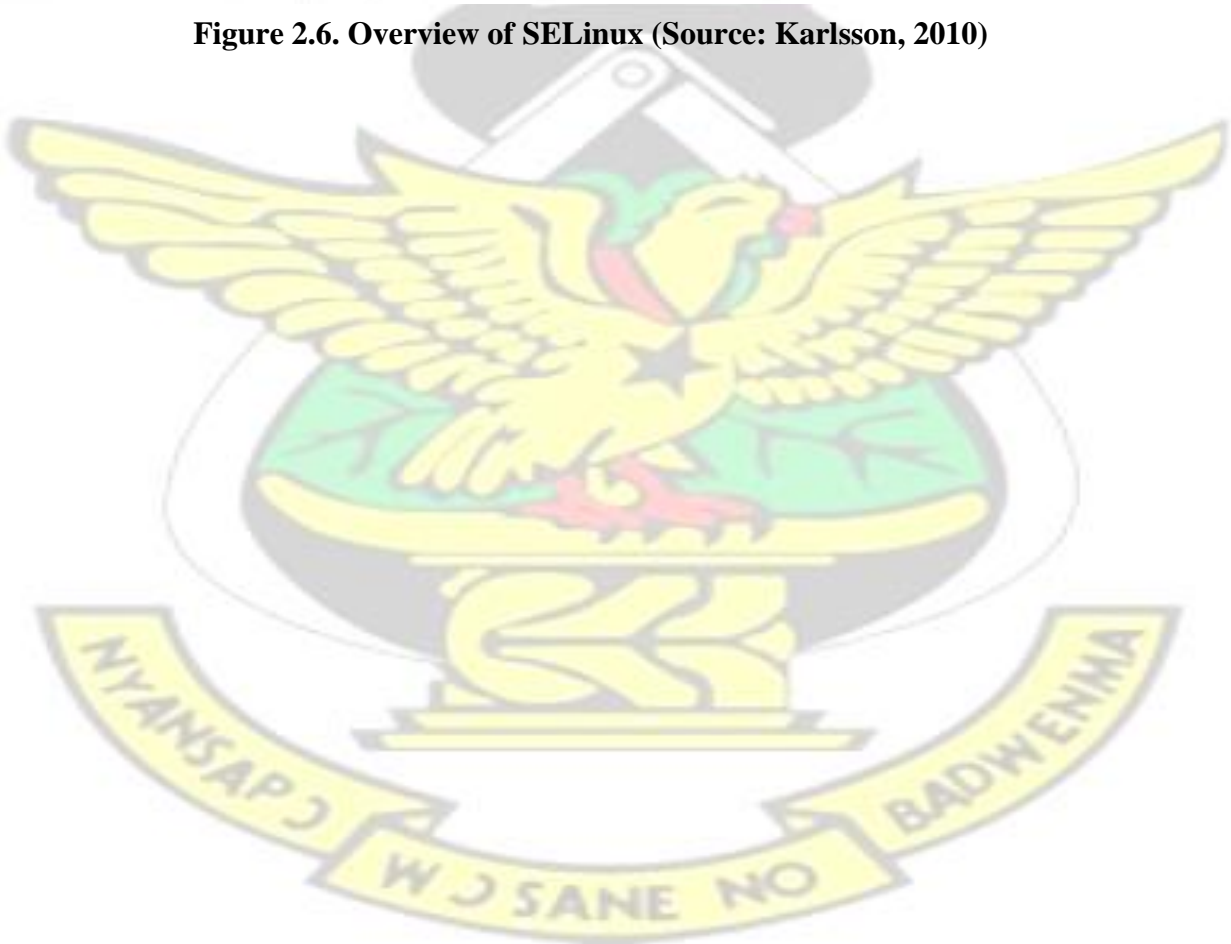


Figure 2.6. Overview of SELinux (Source: Karlsson, 2010)



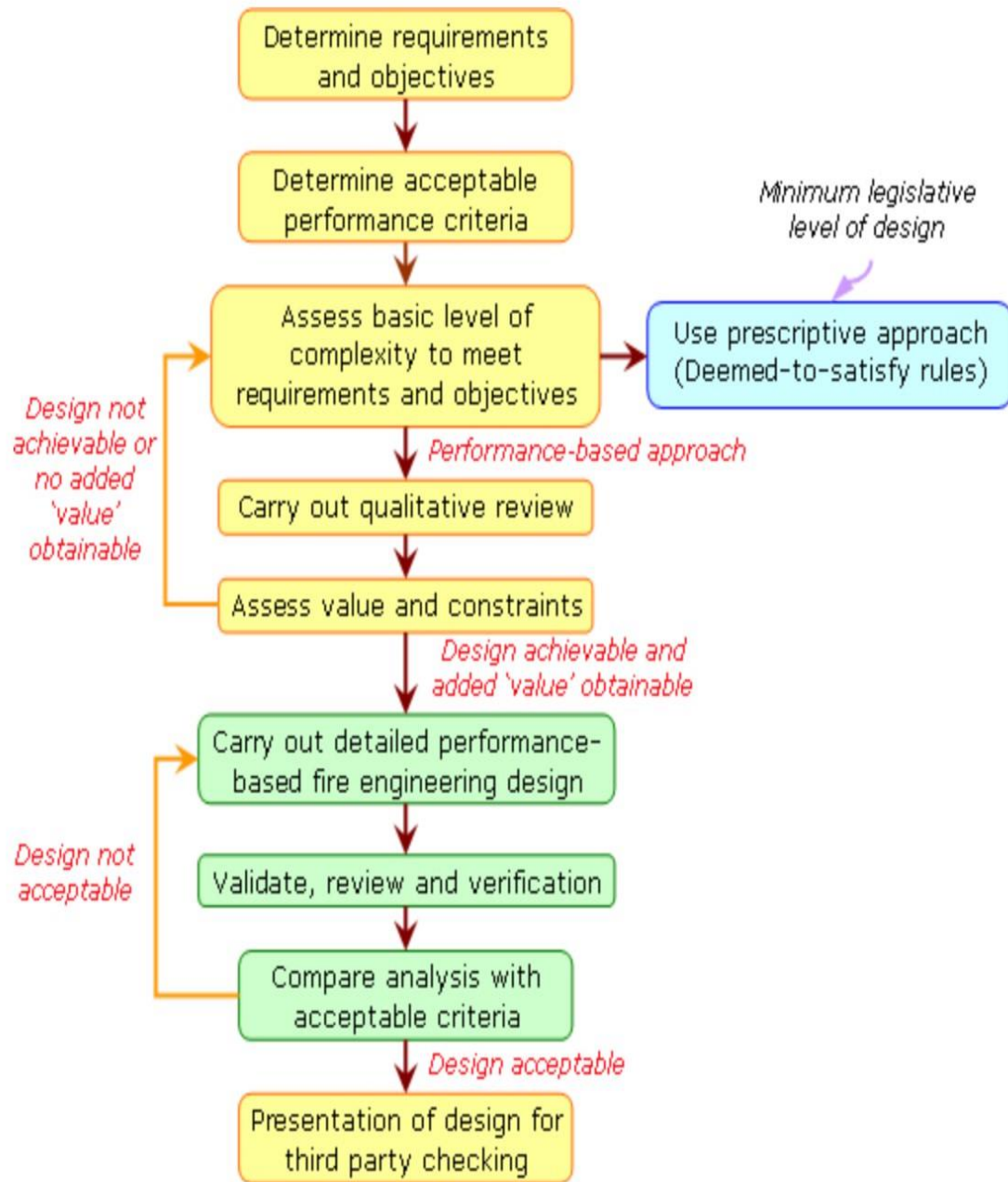


Figure 3.1 Design Methodology Flow Chart (Source: Colin, 2005).

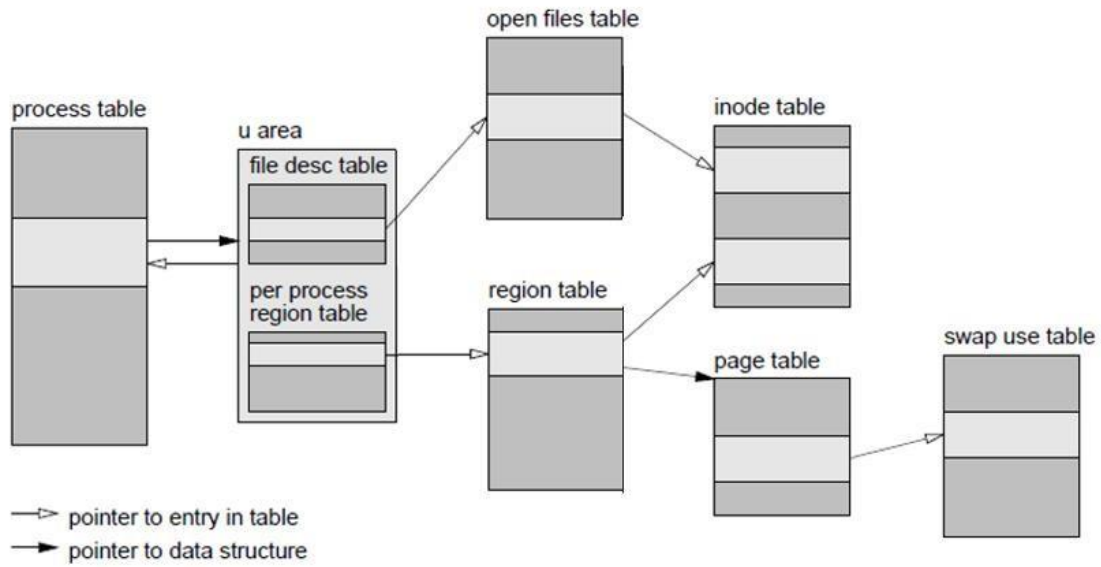


Figure 3.2 Data Structures Source: (Steven & Rago, 2005)

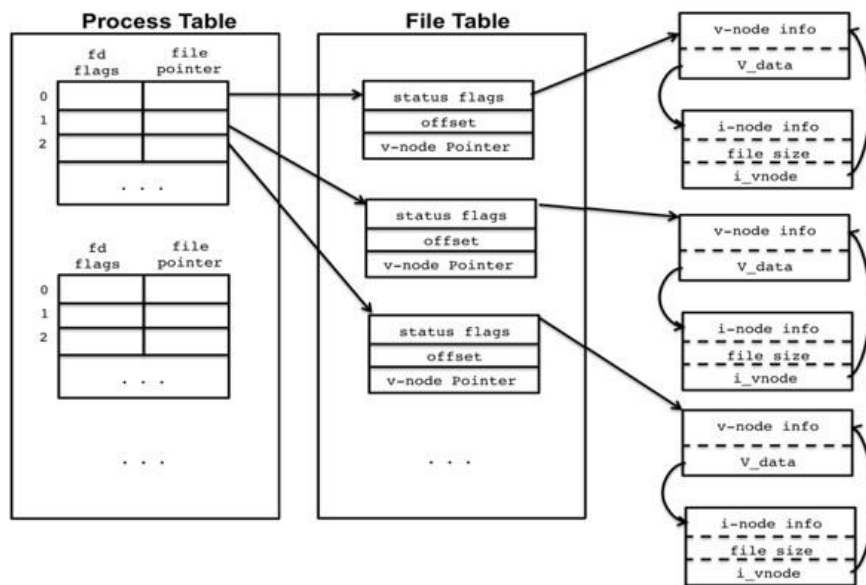


Figure 3.3 Kernel File System Data Structure Source: (Steven & Rago, 2005)

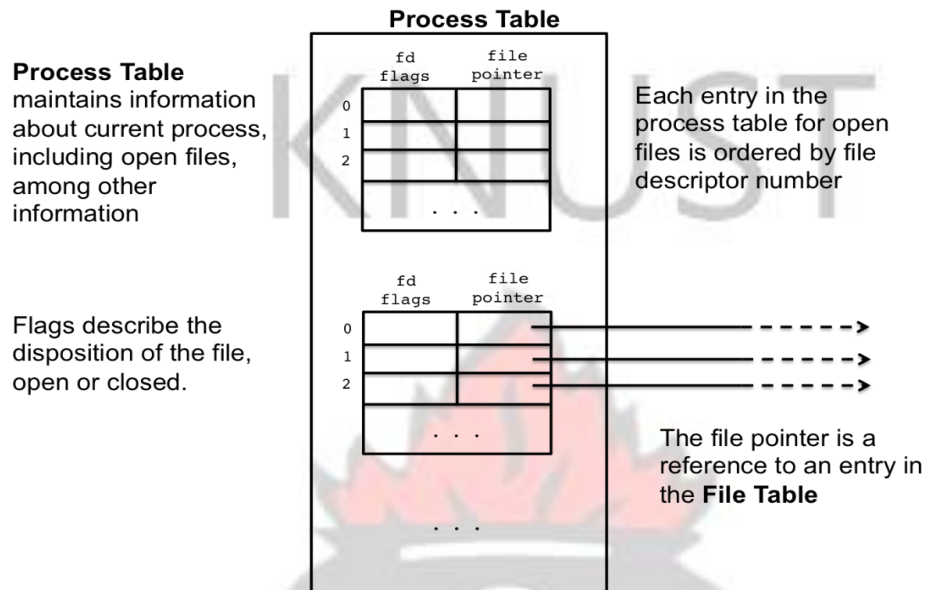


Figure 3.4. The Process Table Source: (Steven & Rago, 2005)

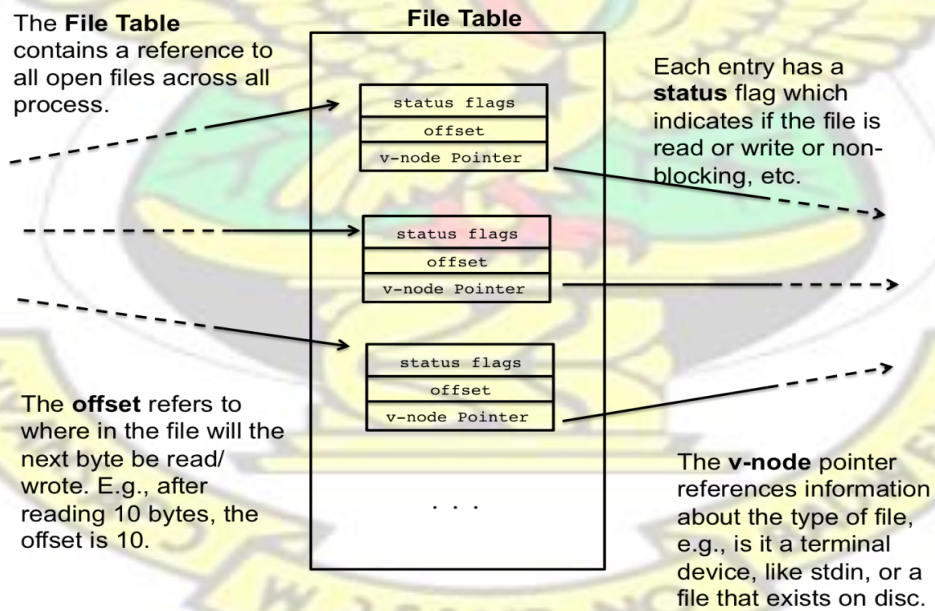


Figure 3.5. The File Table Source: (Steven & Rago, 2005)

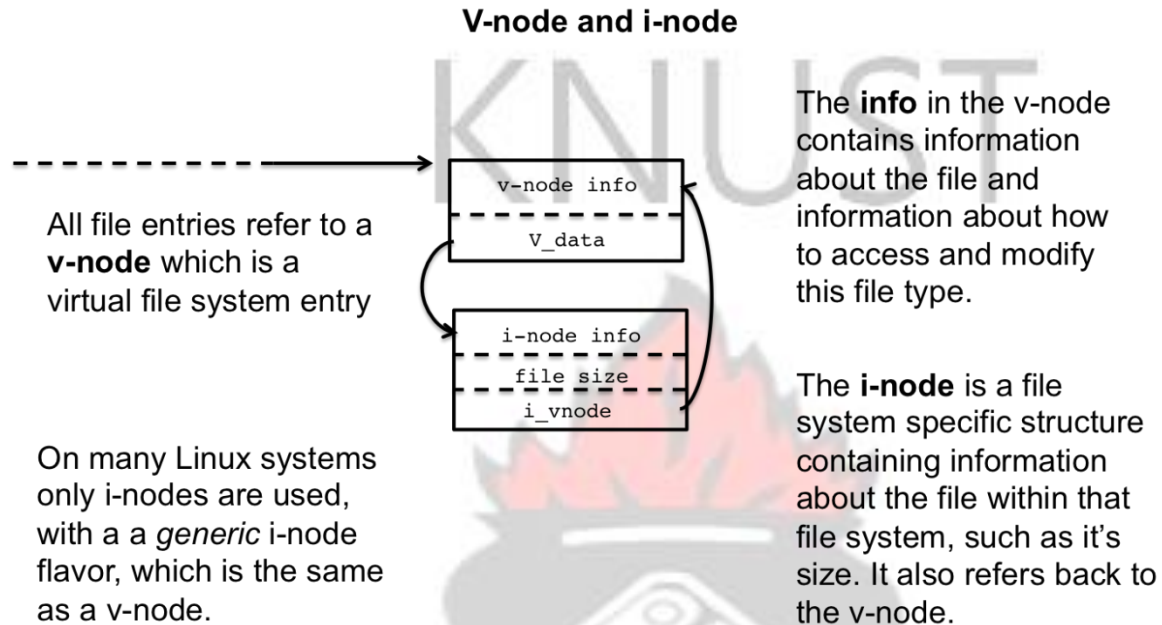


Figure 3.6. The v-node and i-node Source: (Steven & Rago, 2005)

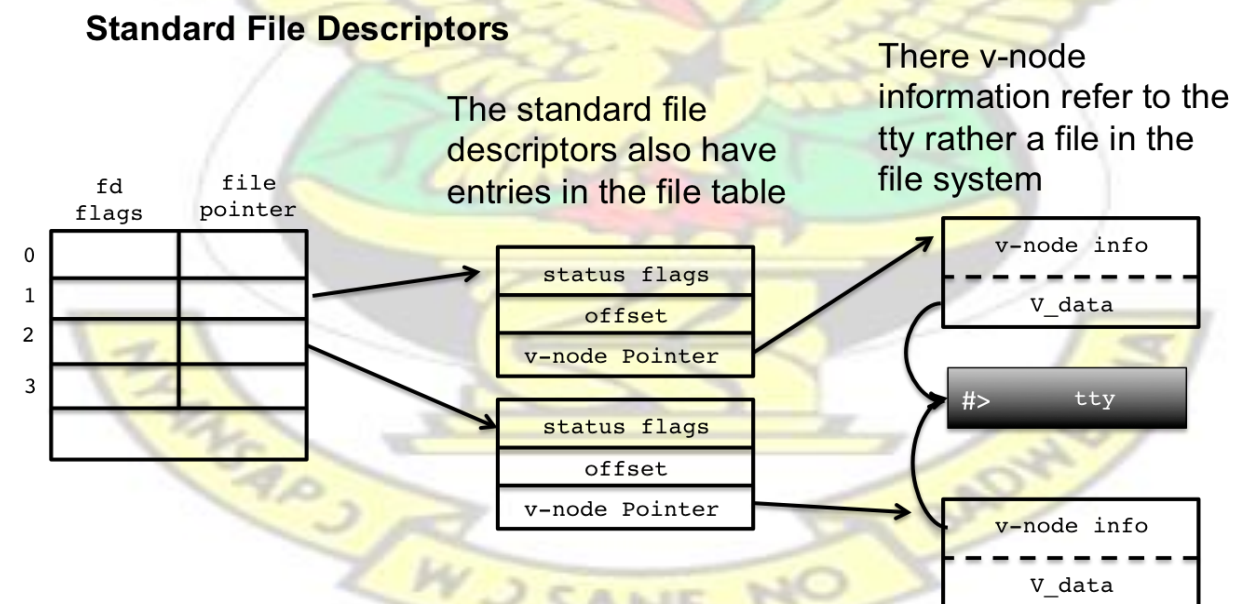


Figure 3.7. Standard File Descriptors Source: (Steven & Rago, 2005)

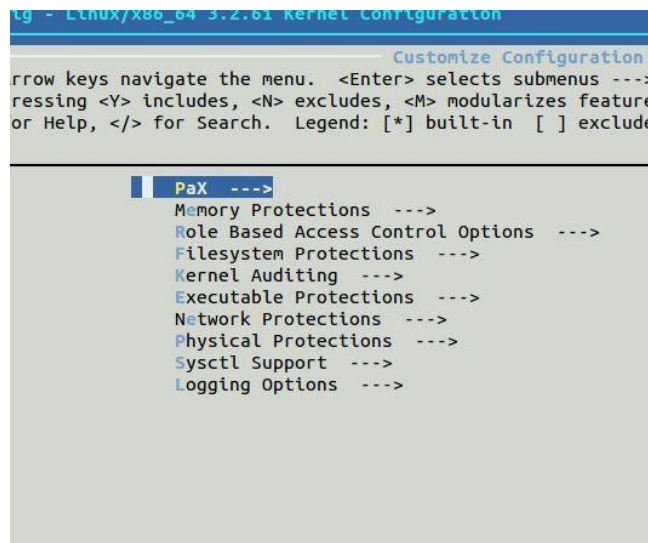


Figure 3.8 Kernel Configuration

```
config.env
1  # General Configurations
2  DISTRO_NAME="Trend-OS"
3  DISTRO_VERSION=1.0
4  DISTRO_AUTHOR="Engr. Danso Ansong"
5  DISTRO_BUILD_DATE="11-04-2016"
6  DISTRO_HOSTNAME=trend-server
7  DISTRO_CODENAME=pioneer
8  DISTRO_DESC="A Powerful Secure Linux Operating System"
9
10 # Important Directories
11 DIR_RES=resources
12 DIR_CONF=conf
13 DIR_EXT=extensions
14 DIR_WORK=work
15 DIR_EXTRAS=extras
16 DIR_LOGS=logs
17 DIR_LISTS=lists
18 DIR_OUTPUT=outputs
19 DIR_TMP_ROOT=$DIR_WORK/ROOT
20 DIR_TMP_ISO=$DIR_WORK/ISO
21 DIR_RES_BG=$DIR_RES/backgrounds
22 DIR_RES_AUDIO=$DIR_RES/backgrounds
23 DIR_RES_ICONS=$DIR_RES/icons
24 DIR_COPY_TMP_ROOT=/
25 DIR_COPY_TMP_ROOT_1=$DIR_WORK/bootstrap/
26
27 # Important Files
28 FILE_BOOT_SPLASH=$DIR_RES_BG/grub/splash.png
29 FILE_LOGS_STDOUT=$DIR_LOGS/stdout
30 FILE_LOGS_STDERR=$DIR_LOGS/stderr
31 FILE_EXCLUDES_LIST=$DIR_LISTS/excludes.list
32 FILE_RM_CONFIG_LIST=$DIR_LISTS/remove-config.list
33 FILE_RM_TMP_VAR_CONFIG_LIST=$DIR_LISTS/remove-tmp-var.list
34
```

Figure 3.9 Configuration environment

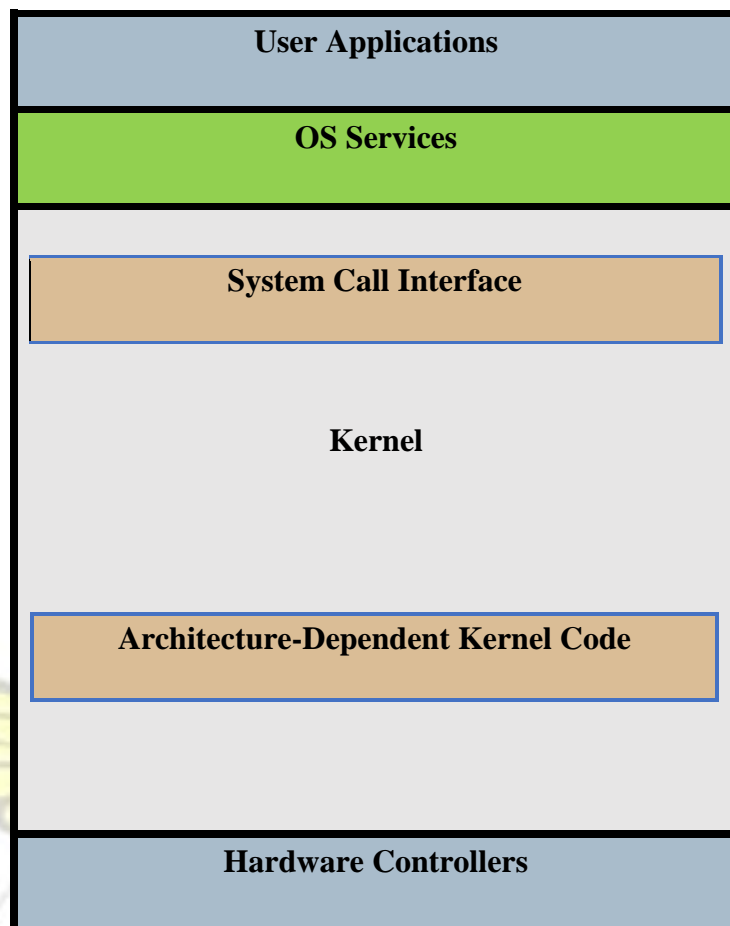
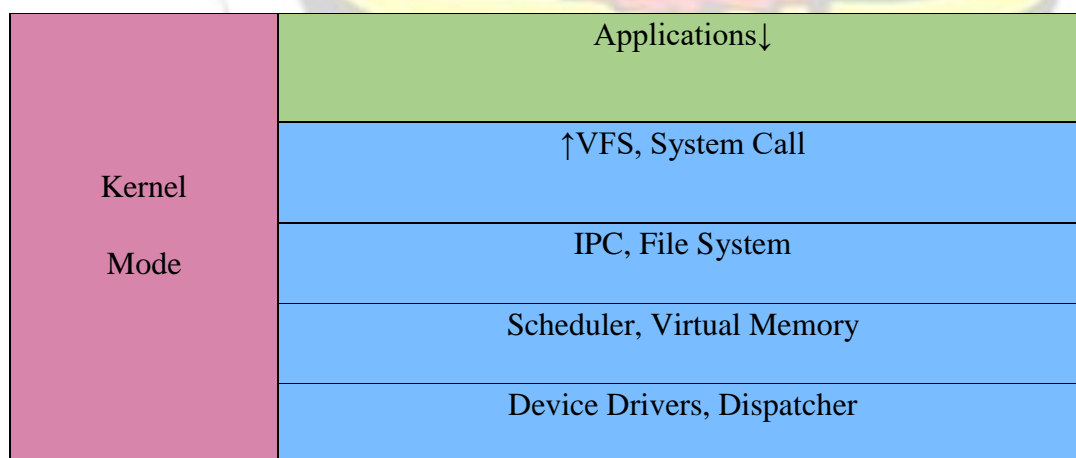


Figure. 4.1. – Breakdown of an Operating System into four major Subsystem.



KNUST



Figure 4.2. Monolithic Design

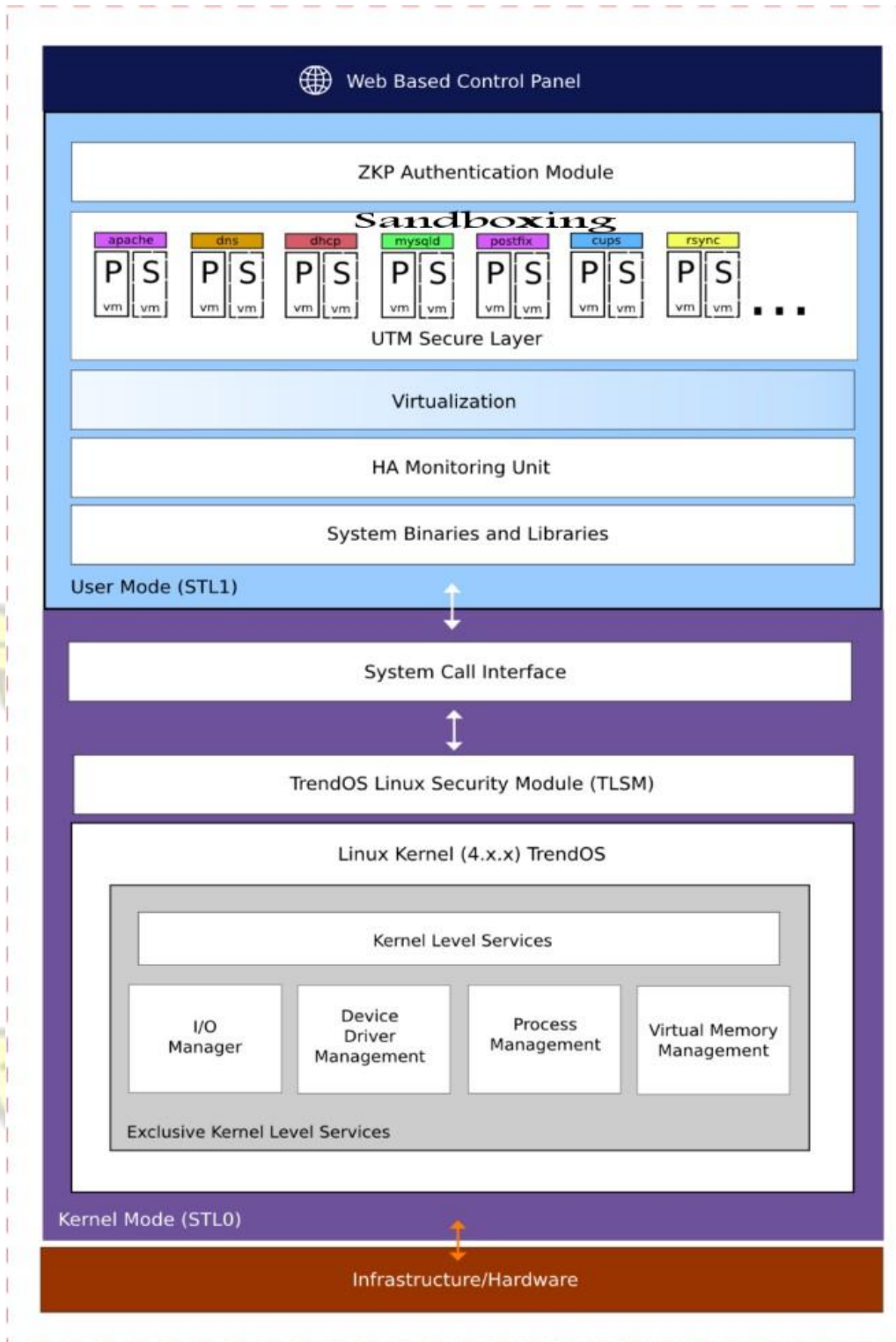


Figure 4.3. Convoluted Kernel Architectural Design

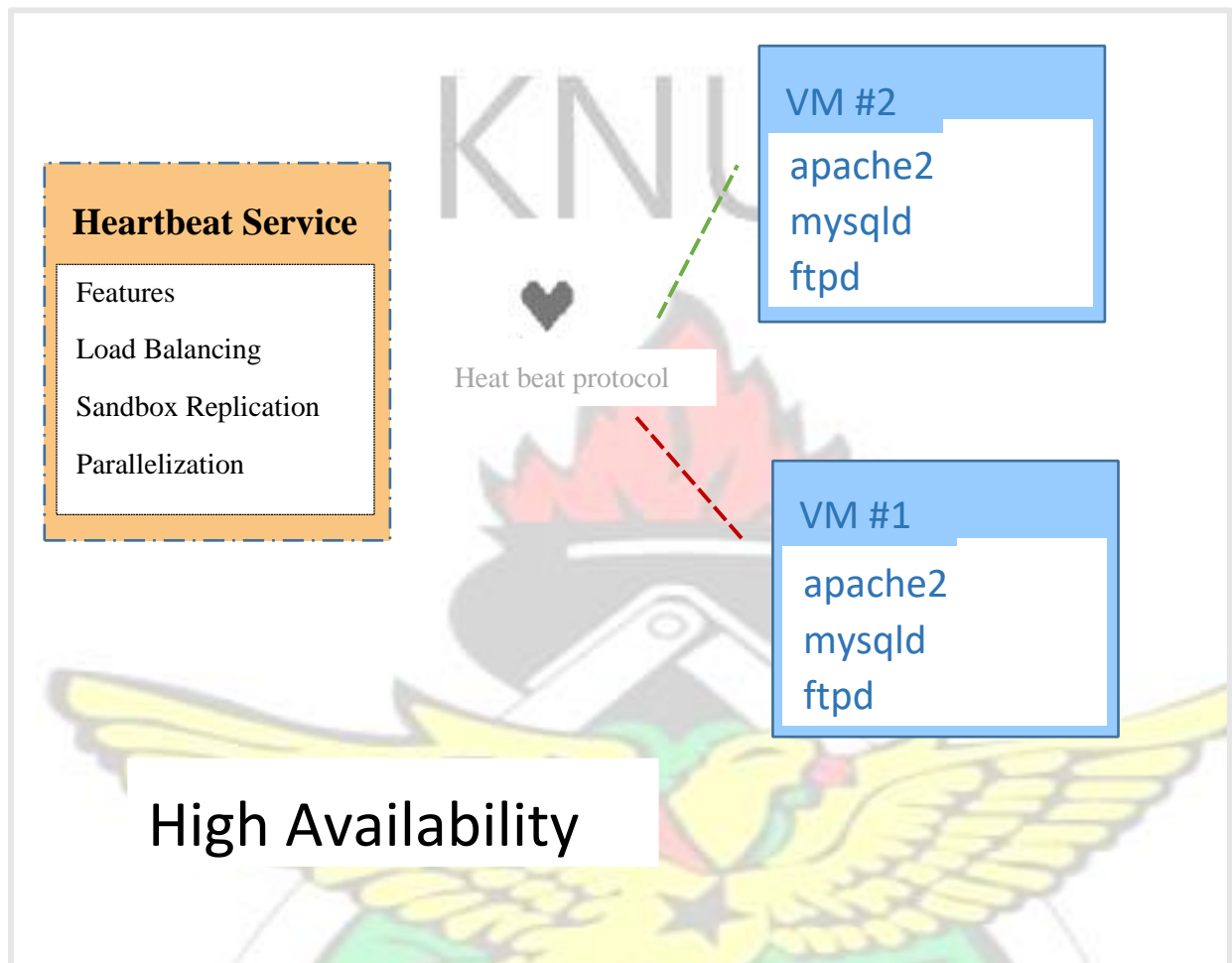


Figure 4.4. Heartbeat architecture of the of the Convoluted Architecture

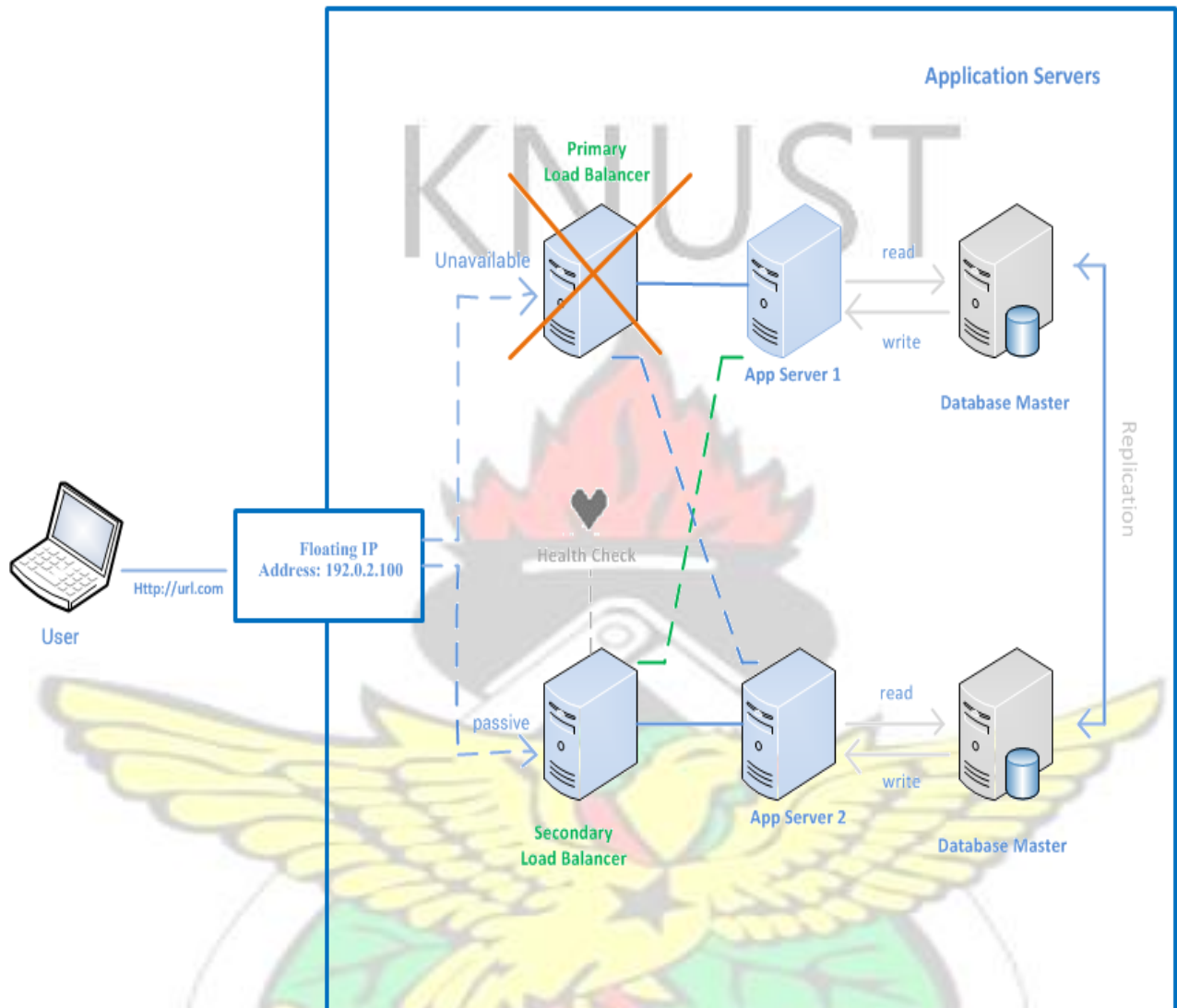


Figure. 4.5. Simplified High Availability (HA) Source (Manual et al., n.d.).

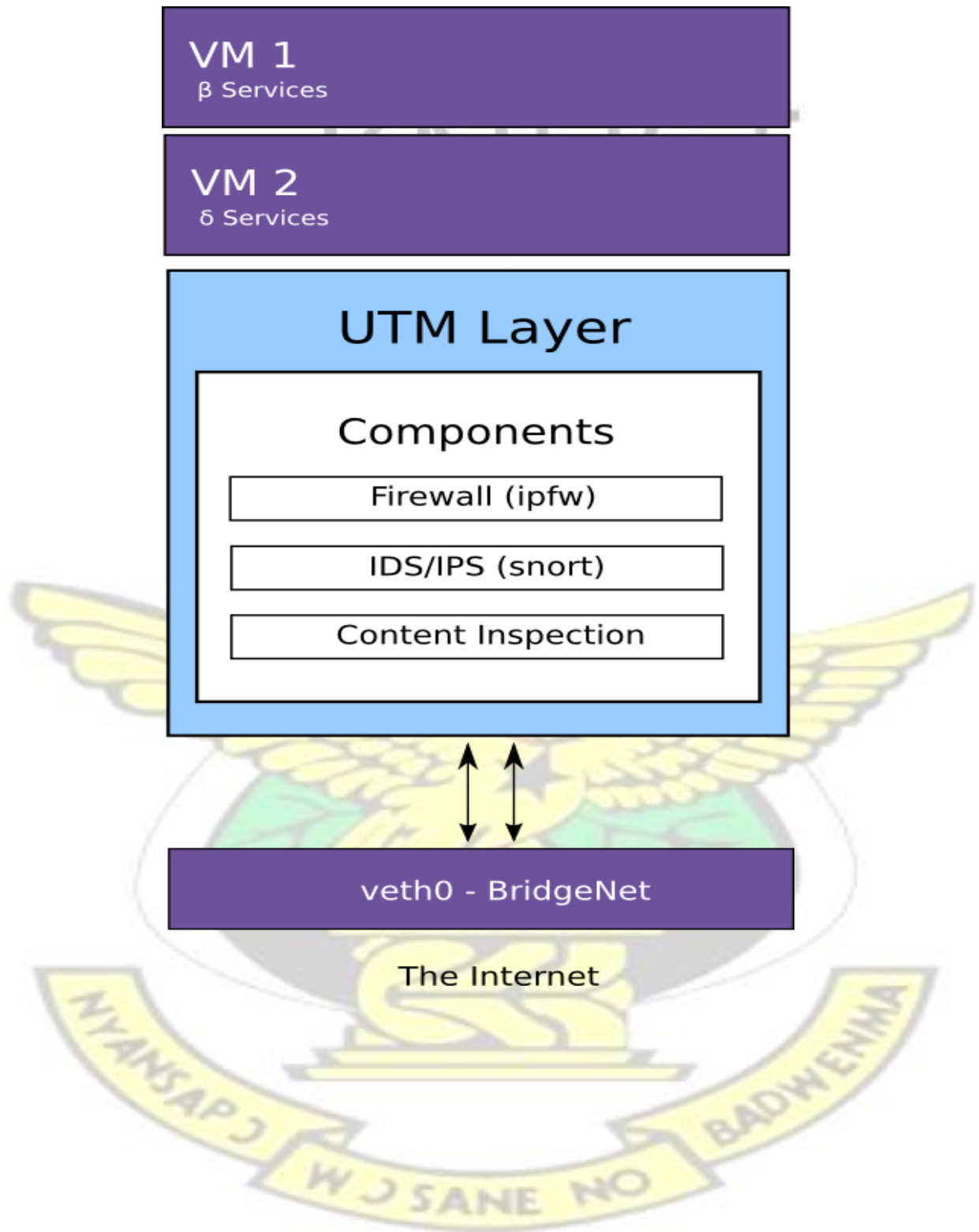


Fig. 4.6 Expanded Unified Threat Management

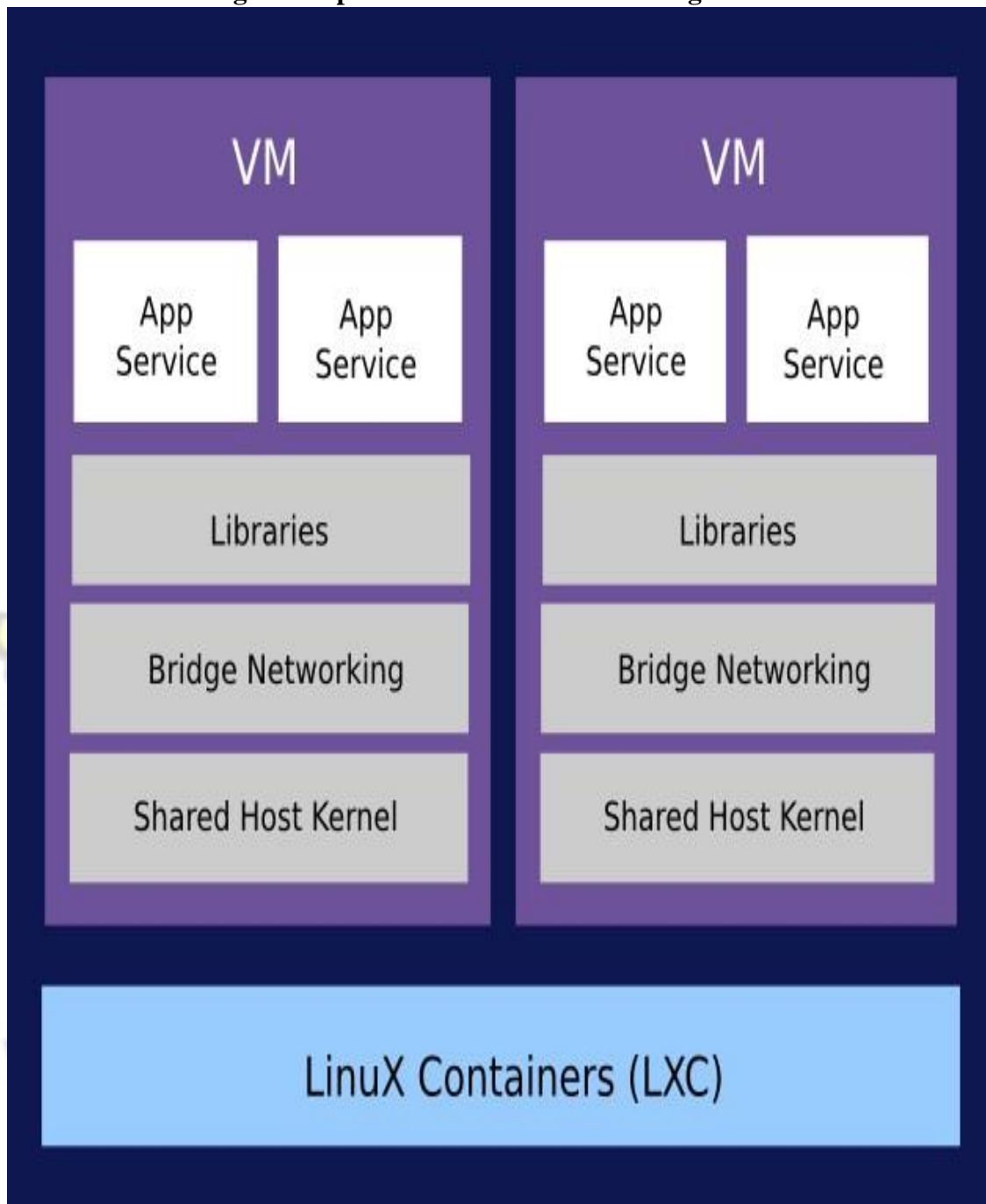


Fig. 4.7 The LXC design in the Architecture

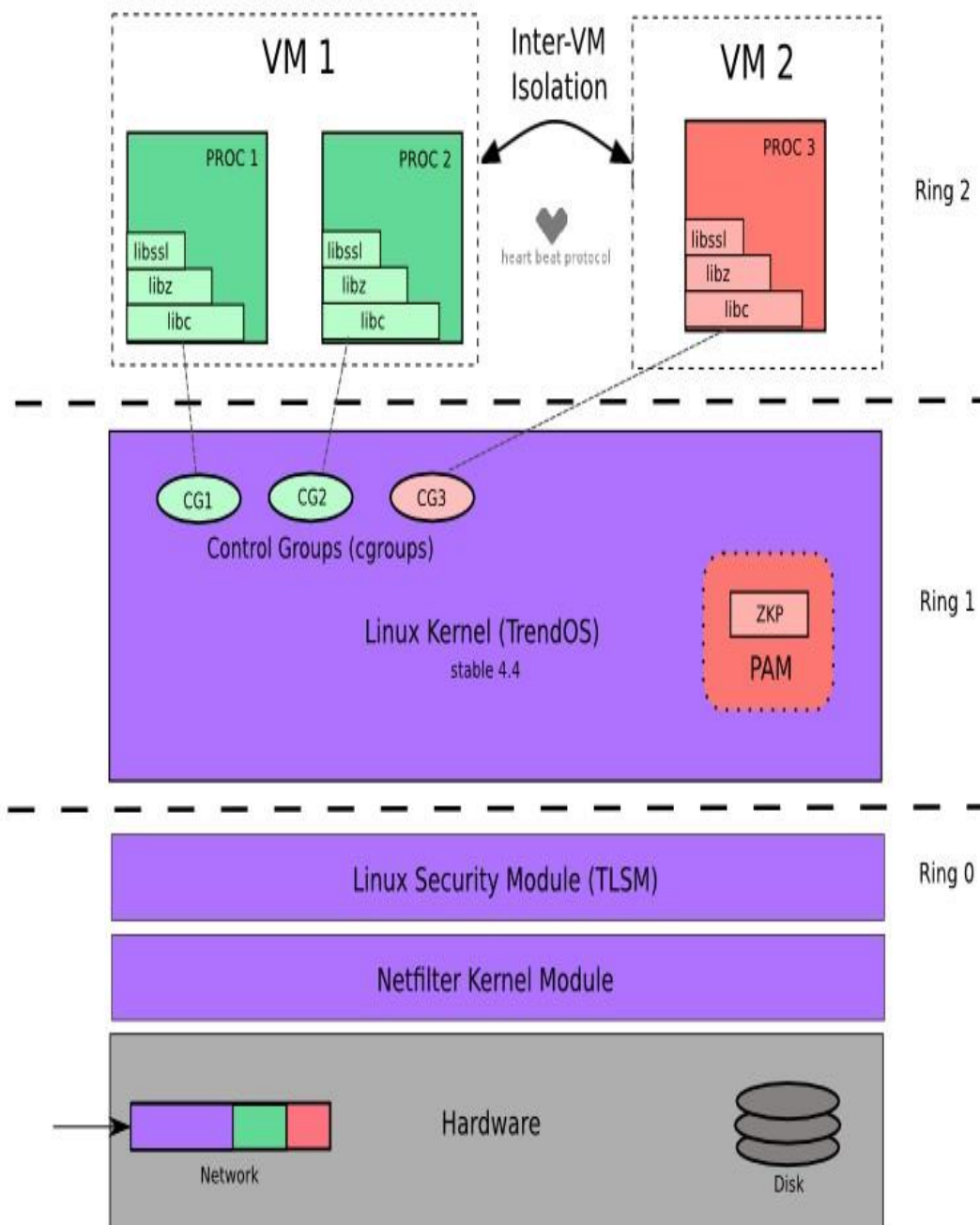


Fig. 4.8 Privilege Separation of the architecture

```

Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

Trend-OS 2.0 Installer #11-04-2016
Author: Engr. Danso Ansong

visit:
www.fredancybersecurity.com

This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. For details, see:
http://www.opensource.org/licenses/mit-license.php

Target architecture: x86_64

[16-11-20/16:54 UTC] Performing Cleanup ...
[16-11-20/16:54 UTC] Cleaning work/ROOT
[16-11-20/16:54 UTC] Empty folder
[16-11-20/16:54 UTC] Cleaning ISO Directory
[16-11-20/16:54 UTC] Done - Cleaning
[16-11-20/16:54 UTC] [Installer Started]!
[16-11-20/16:54 UTC] Invocation Directory: /home/fredan/TrendOS/trend-os
[16-11-20/16:54 UTC] Building ISO for Trend-OS v2.0
[16-11-20/16:54 UTC] Logging [stdout] to /home/fredan/TrendOS/trend-os/
[16-11-20/16:54 UTC] Logging [stderr] to /home/fredan/TrendOS/trend-os/
[16-11-20/16:54 UTC] =====
[16-11-20/16:54 UTC] A. [ Preparing Build Stage ]
[16-11-20/16:54 UTC] =====
[16-11-20/16:54 UTC] Successfully Checked - 'dpkg' - [ONLINE]
[16-11-20/16:54 UTC] Checking Installed Binaries
[16-11-20/16:54 UTC] Checking for pv
[16-11-20/16:54 UTC] pv - [ALREADY INSTALLED]
[16-11-20/16:54 UTC] Checking for xfsprogs
[16-11-20/16:54 UTC] xfsprogs - [ALREADY INSTALLED]

```

Figure 4.9 Kernel deployment into the architecture

```
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/mt6397-regulator.ko
18,662 100% 216.96kB/s 0:00:00 (xfr#6001, ir-chk=1041/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/palmas-regulator.ko
33,542 100% 389.95kB/s 0:00:00 (xfr#6002, ir-chk=1040/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pcap-regulator.ko
15,854 100% 173.96kB/s 0:00:00 (xfr#6003, ir-chk=1039/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pcf50633-regulator.ko
10,902 100% 119.62kB/s 0:00:00 (xfr#6004, ir-chk=1038/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pfuzel00-regulator.ko
27,094 100% 287.60kB/s 0:00:00 (xfr#6005, ir-chk=1037/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/pwm-regulator.ko
8,614 100% 91.44kB/s 0:00:00 (xfr#6006, ir-chk=1036/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/qcom_spmi-regulator.ko
14,078 100% 149.44kB/s 0:00:00 (xfr#6007, ir-chk=1035/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rc5t583-regulator.ko
13,678 100% 143.63kB/s 0:00:00 (xfr#6008, ir-chk=1034/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rn5t618-regulator.ko
10,574 100% 111.03kB/s 0:00:00 (xfr#6009, ir-chk=1033/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/rt5033-regulator.ko
9,598 100% 100.79kB/s 0:00:00 (xfr#6010, ir-chk=1032/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s2mpa01.ko
21,838 100% 229.31kB/s 0:00:00 (xfr#6011, ir-chk=1031/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s2mps11.ko
61,190 100% 635.70kB/s 0:00:00 (xfr#6012, ir-chk=1030/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/s5m8767.ko
30,166 100% 313.39kB/s 0:00:00 (xfr#6013, ir-chk=1029/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/sky81452-regulator.ko
8,366 100% 82.52kB/s 0:00:00 (xfr#6014, ir-chk=1028/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps51632-regulator.ko
11,190 100% 98.45kB/s 0:00:00 (xfr#6015, ir-chk=1027/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps6105x-regulator.ko
9,510 100% 43.81kB/s 0:00:00 (xfr#6016, ir-chk=1026/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps62360-regulator.ko
14,374 100% 41.53kB/s 0:00:00 (xfr#6017, ir-chk=1025/9264)
lib/modules/4.4.0-21-generic/kernel/drivers/regulator/tps65023-regulator.ko
17,246 100% 30.35kB/s 0:00:00 (xfr#6018, ir-chk=1024/9264)
```

Figure 4.10. Loading of the various security and driver modules into the kernel.


```
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

[16-11-20/17:02 UTC] Done Copying File System!
[16-11-20/17:02 UTC] =====
[16-11-20/17:02 UTC] C. [ Starting Configuration Stage ]
[16-11-20/17:02 UTC] =====
[16-11-20/17:02 UTC] Removing configuration files
[16-11-20/17:02 UTC] Removing etc/X11/xorg.conf*
[16-11-20/17:02 UTC] Removing etc/resolv.conf
[16-11-20/17:02 UTC] Removing etc/hosts
[16-11-20/17:02 UTC] Removing etc/hostname
[16-11-20/17:02 UTC] Removing etc/timezone
[16-11-20/17:02 UTC] Removing etc/udev/rules.d/70-persistent*
[16-11-20/17:02 UTC] Removing etc/mtab
[16-11-20/17:02 UTC] Removing etc/fstab
[16-11-20/17:02 UTC] Removing etc/cups/ssl/server.crt
[16-11-20/17:02 UTC] Removing etc/cups/ssl/server.key
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_dsa_key.pub
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_dsa_key
[16-11-20/17:02 UTC] Removing etc/ssh/ssh_host_rsa_key.pub
[16-11-20/17:02 UTC] Removing etc/wicd/wired-settings.conf
[16-11-20/17:02 UTC] Removing etc/wicd/wireless-settings.conf
[16-11-20/17:02 UTC] Removing etc/printcap
[16-11-20/17:02 UTC] Removing APT [cache]
[16-11-20/17:02 UTC] Removing temporary files within the /var and /etc folders
[16-11-20/17:02 UTC] Generating Empty Logs
[16-11-20/17:02 UTC] Praparing /etc/passwd and /etc/group ...
[16-11-20/17:02 UTC] Chrooting to remove users (>=998 && != 65534) ...
[16-11-20/17:02 UTC] Cleaning TMP ROOT with [ apt-get clean ]
[16-11-20/17:02 UTC] Verifying autologin 'casper' scripts ...
[16-11-20/17:02 UTC] Copying casper.conf ...
[16-11-20/17:02 UTC] Modifying casper.conf ...
[16-11-20/17:02 UTC] Copying lsb-release ...
[16-11-20/17:02 UTC] Modifying lsb-release ...
[16-11-20/17:02 UTC] Checking Ubiquity debian(ubuntu) installer ...
[16-11-20/17:02 UTC] Success! user-setup-apply ...
[16-11-20/17:02 UTC] Success! apt-setup ...
```

Figure 4.11 modifying the generic kernel to suit the convoluted kernel Architecture

```

Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help
/abstract/schema_dumper.rb
  1,777 100% 2.32kB/s 0:00:00 (xfr#20236, ir-chk=1010/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/abstract/schema_statements.rb
  40,894 100% 53.39kB/s 0:00:00 (xfr#20237, ir-chk=1009/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/abstract/transaction.rb
   5,080 100% 6.63kB/s 0:00:00 (xfr#20238, ir-chk=1008/26276)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/array_parser.rb
   2,726 100% 3.55kB/s 0:00:00 (xfr#20239, ir-chk=1009/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/column.rb
    611 100% 0.80kB/s 0:00:00 (xfr#20240, ir-chk=1008/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/database_statements.rb
   7,996 100% 10.43kB/s 0:00:00 (xfr#20241, ir-chk=1007/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/oid.rb
   1,789 100% 2.33kB/s 0:00:00 (xfr#20242, ir-chk=1006/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/quotings.rb
   3,164 100% 4.12kB/s 0:00:00 (xfr#20243, ir-chk=1005/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/referential_integrity.rb
   1,043 100% 1.36kB/s 0:00:00 (xfr#20244, ir-chk=1004/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/schema_definitions.rb
   4,398 100% 5.73kB/s 0:00:00 (xfr#20245, ir-chk=1003/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/schema_statements.rb
  24,175 100% 31.48kB/s 0:00:00 (xfr#20246, ir-chk=1002/26278)
opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/activerecord-4.2.7.1/lib/active_record/connection_adapters
/postgresql/utis.rb
  
```

Figure 4.12 Integrating redundant security modules to protect the kernel against itself.

```
Terminal - root@fredan-desktop: ~/TrendOS/trend-os
File Edit View Terminal Tabs Help

bind (130)
colord (123)
dip (30)
lp (7)
ssl-cert (112)
daemon (1)
postgres (1001)
shadow (42)
tty (5)
crontab (107)
wireshark (131)
mlocate (117)
ssh (118)
messagebus (110)
utmp (43)
staff (50)
fredan (1000)
lpadmin (113)
whoopsie (116)
avahi-autoipd (119)
syslog (108)
lightdm (114)
plugdev (46)
nogroup (65534)
systemd-timesync (102)
adm (4)
mail (8)

[16-11-20/17:12 UTC] Generating MD5-SUMS md5sums.txt ...
[16-11-20/17:12 UTC] Resting ... (3 secs)
[16-11-20/17:12 UTC] =====
[16-11-20/17:12 UTC] F. [ Generate Bootable ISO File - Final Stage ]
[16-11-20/17:12 UTC] =====
[16-11-20/17:12 UTC] Creating trendos-2.0-server-amd64.iso in outputs ...
[16-11-20/17:14 UTC] Horray! file generated at /home/fredan/TrendOS/trend-os/outputs/trendos-2.0-server-amd64.iso
[16-11-20/17:14 UTC] Making ISO USB Compatible isohybrid /home/fredan/TrendOS/trend-os/outputs/trendos-2.0-server-amd64.iso
```

Figure 4.13 Building the various files and integrating into inbuilt protocols for compatibility.

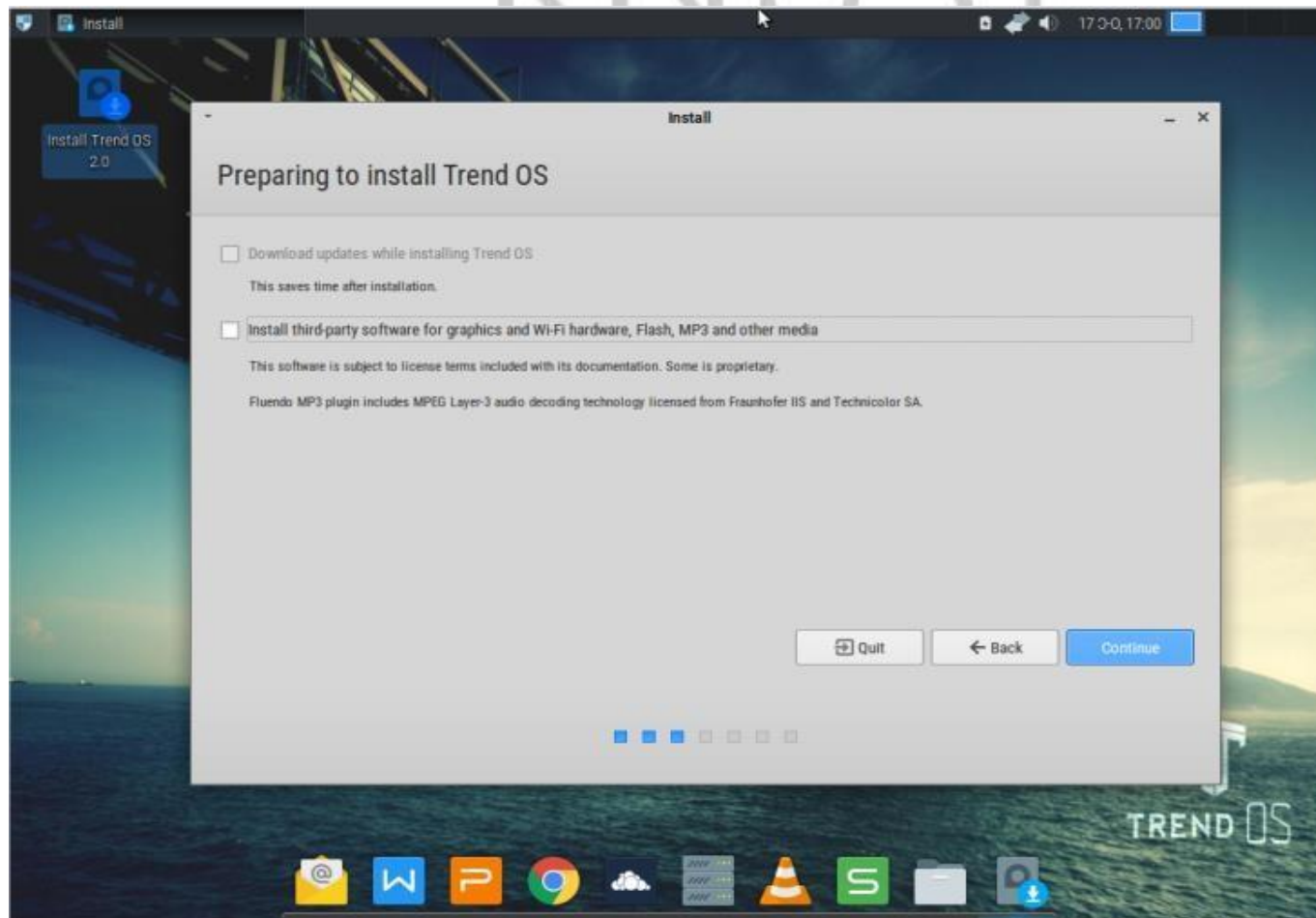


Figure 4.14 Good user friendly installation experience



Figure 4.15 Step by step system Installation guide.

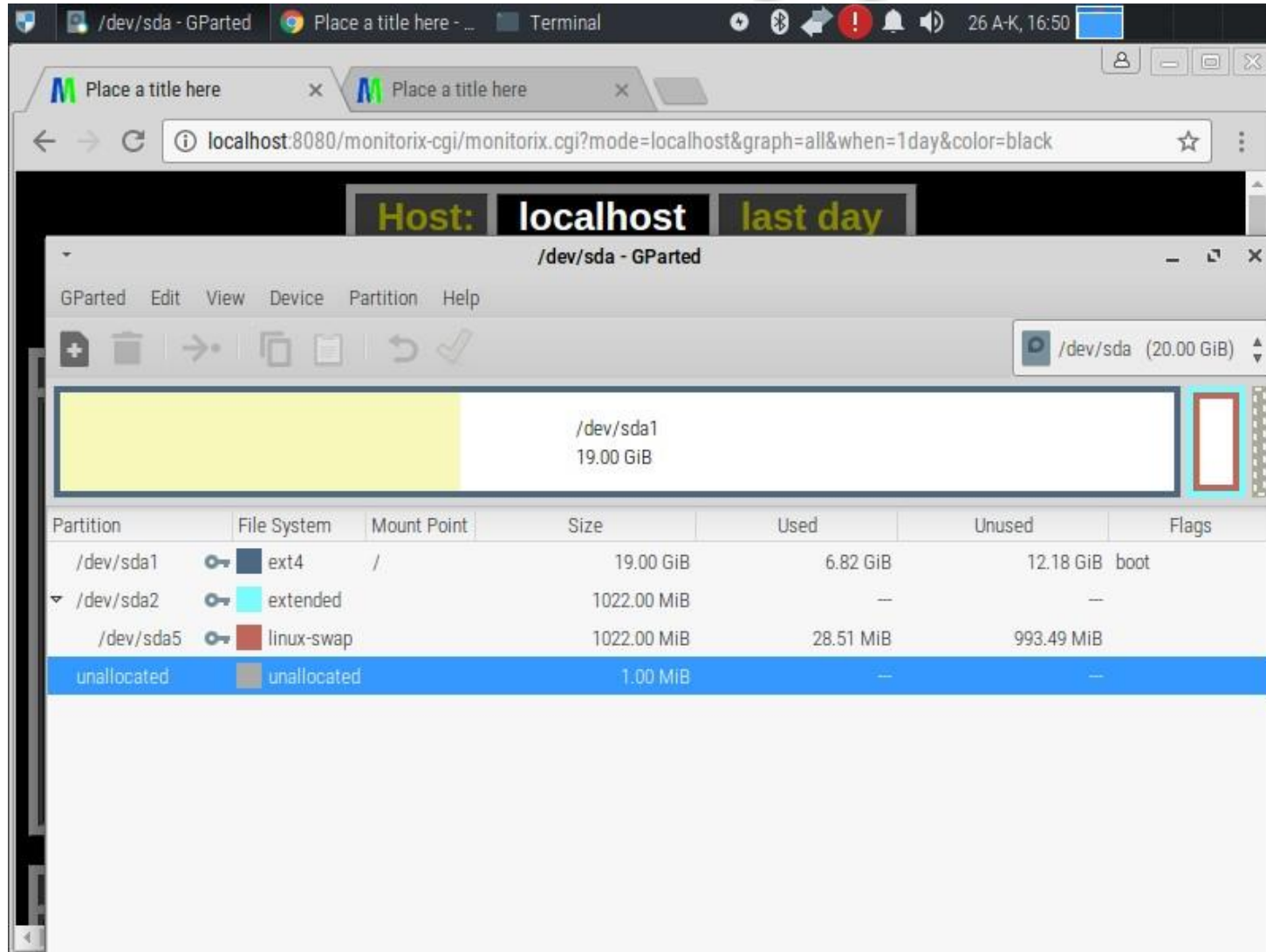


Figure 4.16. Adopts to minimum hardware resource.

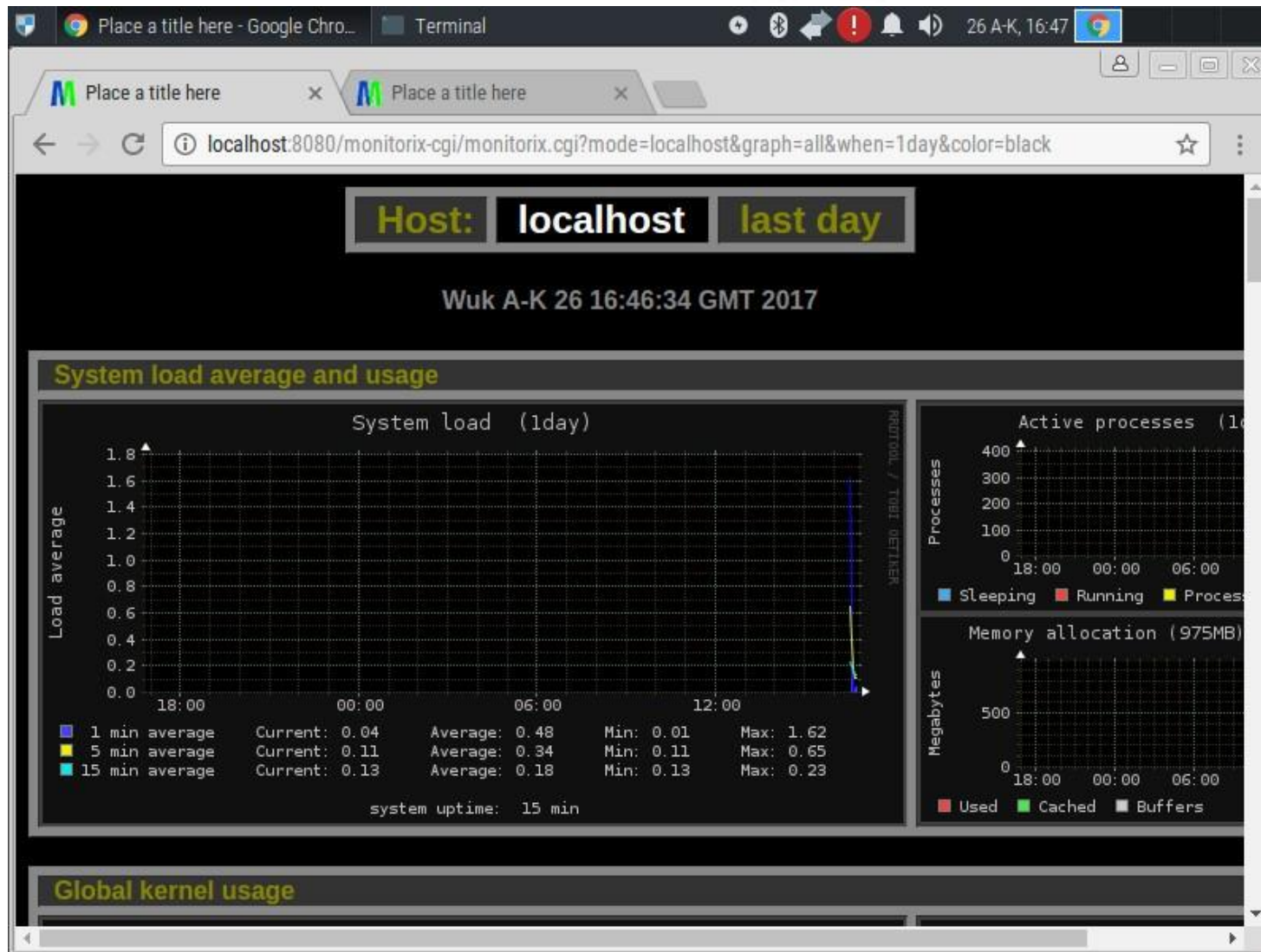


Figure 4.17. Minimum load system resource utilization.

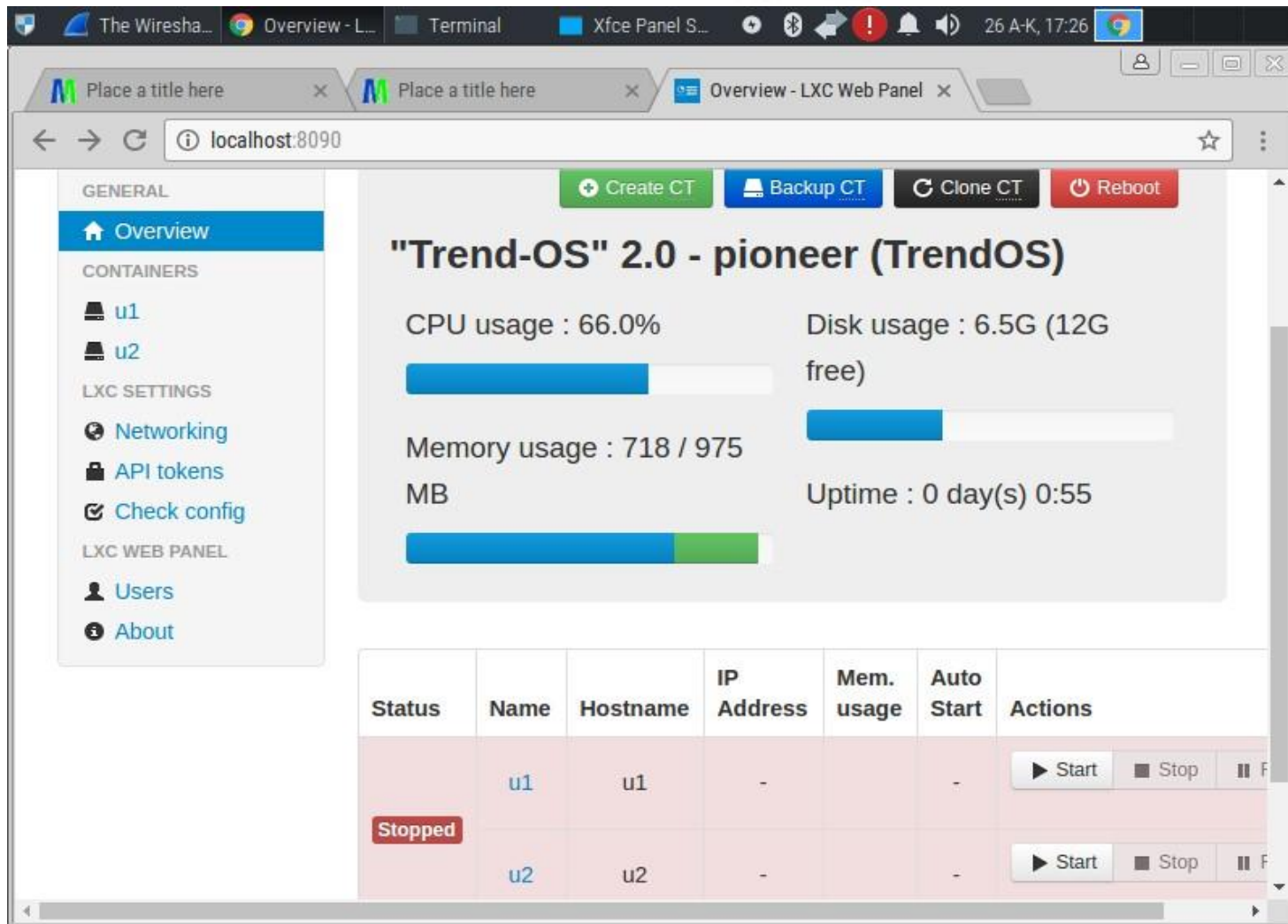


Figure 4.18 Support for multiple server instances running on same system – Linux containers.

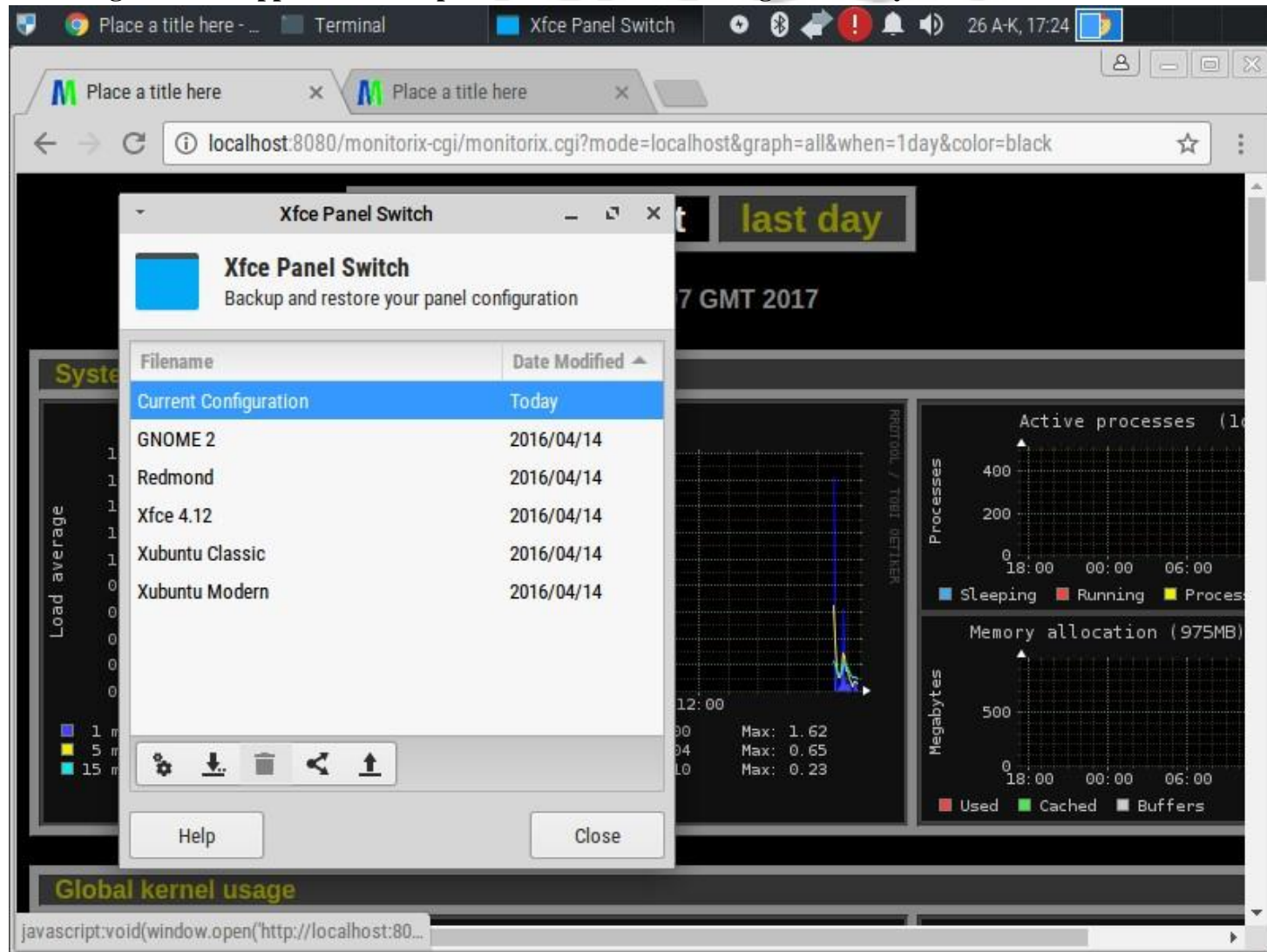


Figure 4.19 Live backup systems for High Availability.

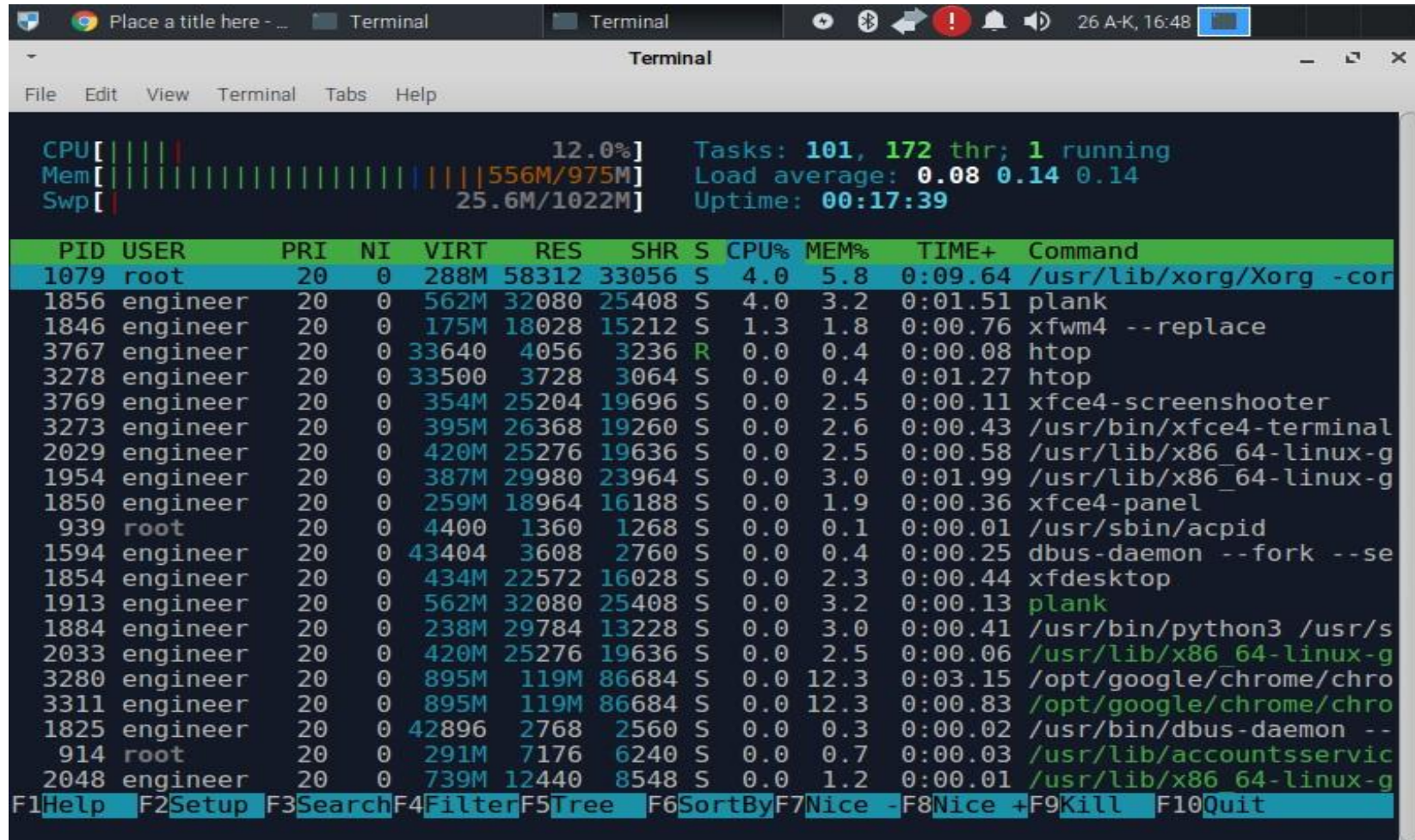


Figure 4.20 Minimum CPU and Memory utilization.

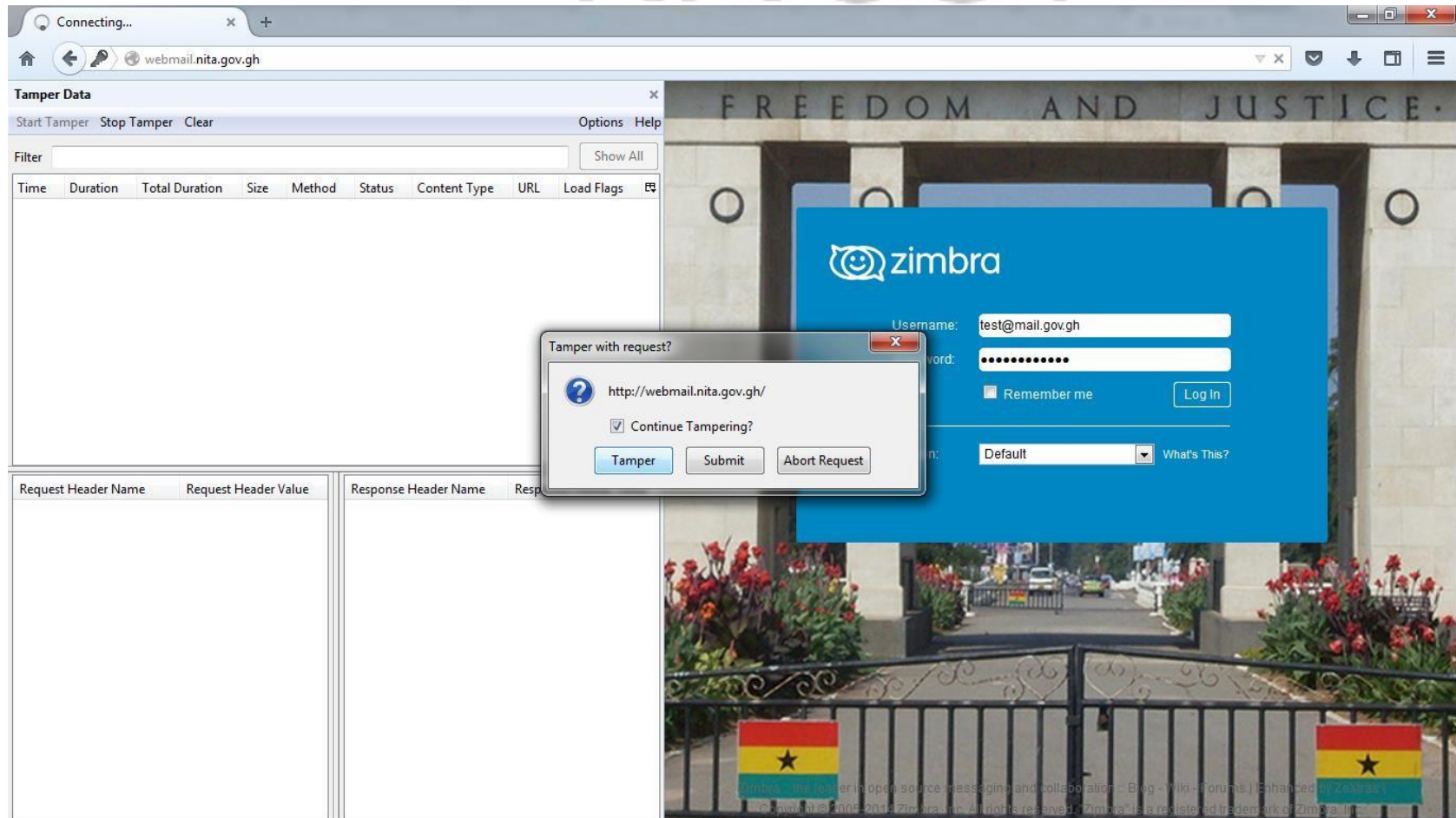
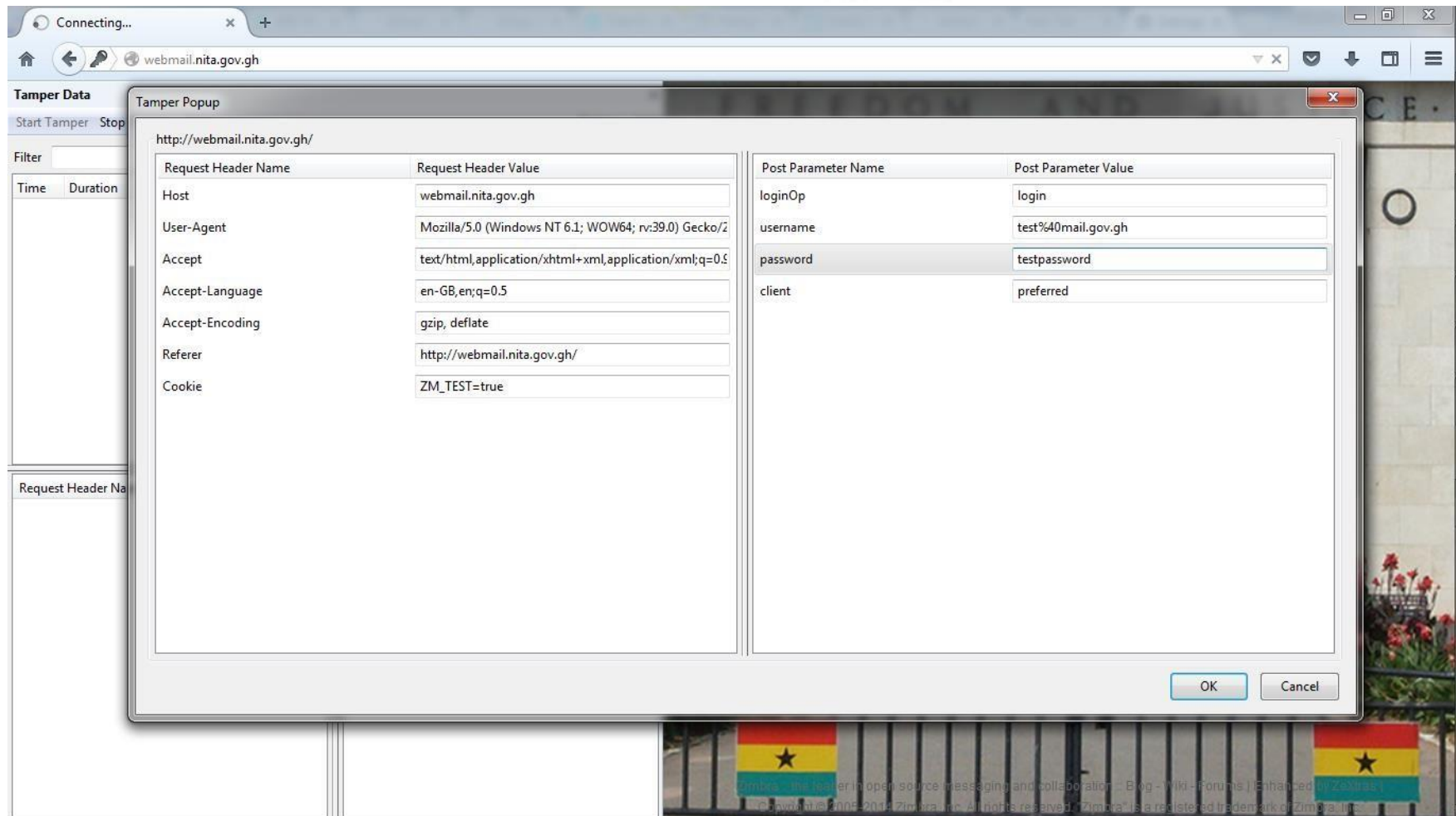


Figure 6.1. Security Testing of the User Interface



KNUST

Figure 6.2. Tamper Popup

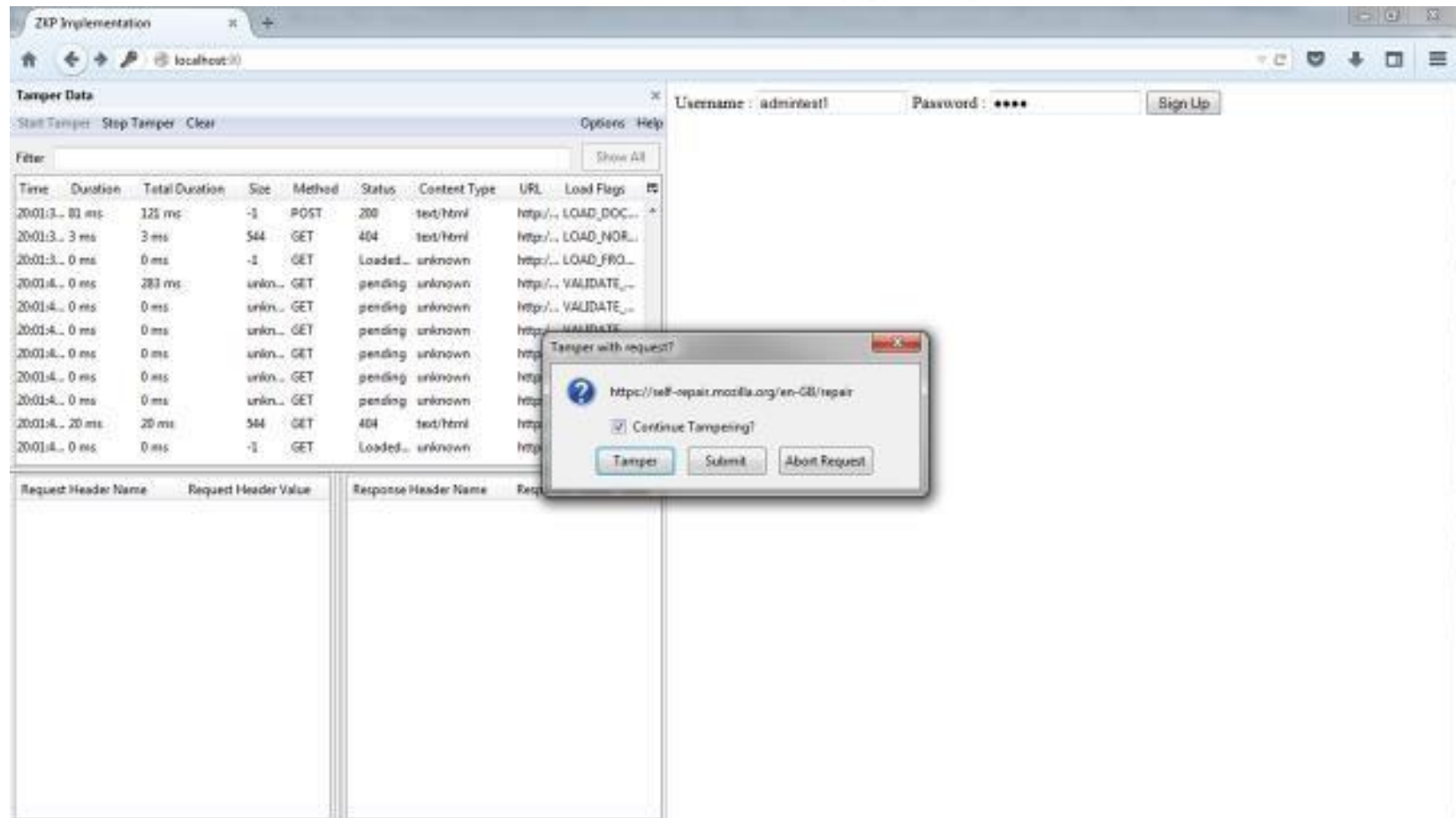
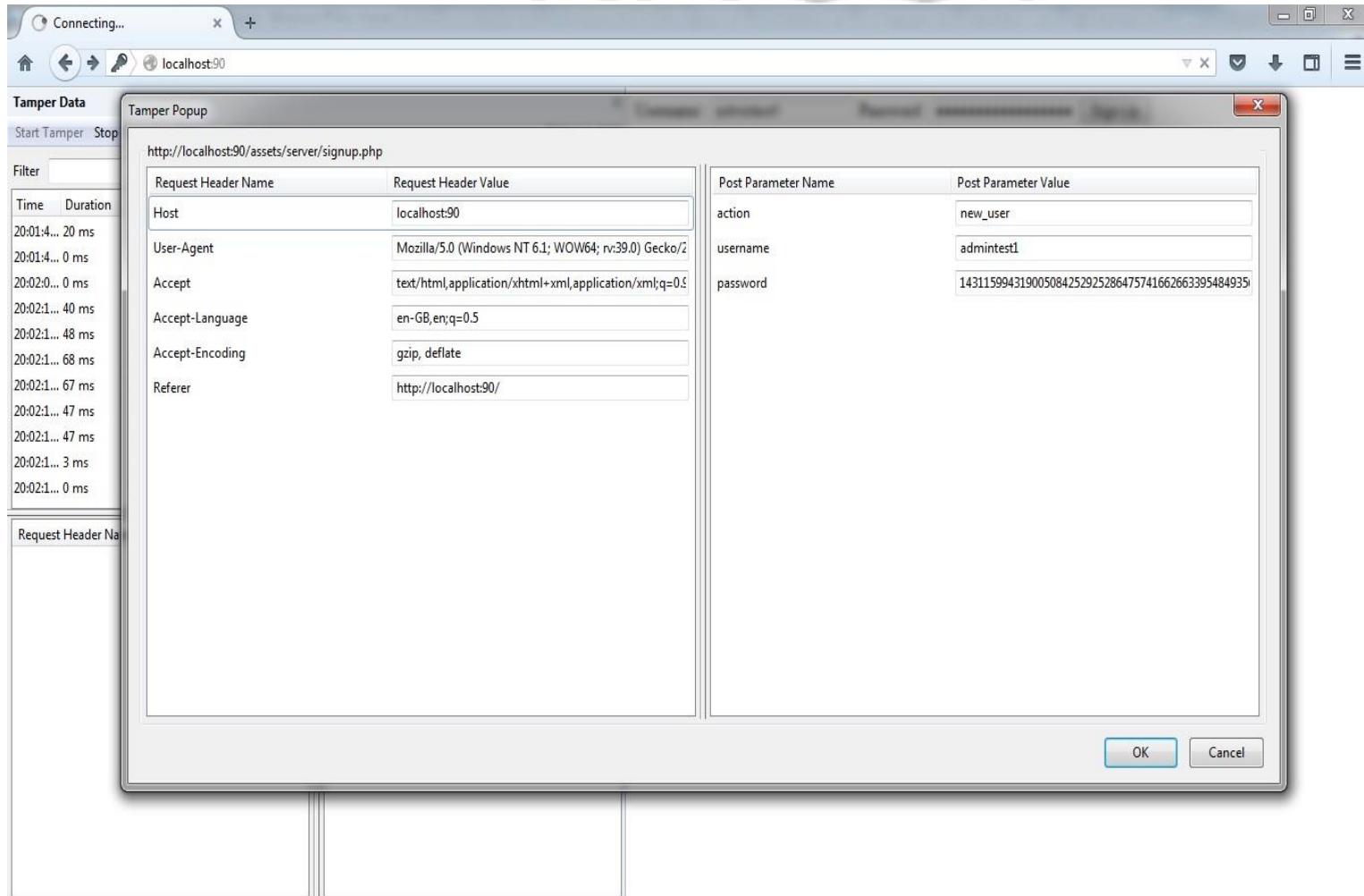


Figure 6.3. Tamper with request



KNU IST

CPU SYSTEM BENCHMARK

```
engineer@TrendOS:~/Downloads$ sysbench --test=cpu --num-threads=4 --cpu-max-prim
e=9999 run
Threads started!
Done.

Maximum prime number checked in CPU test: 9999

Test execution summary:
  total time:                10.0260s
  total number of events:    10000
  total time taken by event execution: 40.0825
  per-request statistics:
    min:                      0.93ms
    avg:                      4.01ms
    max:                      46.77ms
    approx. 95 percentile:    15.35ms

Threads fairness:
  events (avg/stddev):       2500.0000/7.25
  execution time (avg/stddev): 10.0206/0.00
```

Figure 6.6. CPU System Benchmark

KNU IST

DD BENCHMARK

```
engineer@Trend05:~$ cat /dev/sda1 | pipebench -q > /dev/null
cat: /dev/sda1: Permission denied
Summary:
Piped 0.00 B in 00h00m00.00s: 0.00 B/second
engineer@Trend05:~$ sudo cat /dev/sda1 | pipebench -q > /dev/null
[sudo] password for engineer:
Summary:
Piped 18.99 GB in 00h00m57.00s: 341.29 MB/second
engineer@Trend05:~$ sudo cat /dev/sda | pipebench -q > /dev/null
Summary:
Piped 20.00 GB in 00h00m49.96s: 409.85 MB/second
engineer@Trend05:~$ sudo cat /dev/sda5 | pipebench -q > /dev/null
Summary:
Piped 1022.00 MB in 00h00m03.29s: 310.27 MB/second
engineer@Trend05:~$ dd bs=16k count=102400 oflag=direct if=/dev/zero of=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 6.42428 s, 261 MB/s
engineer@Trend05:~$
```

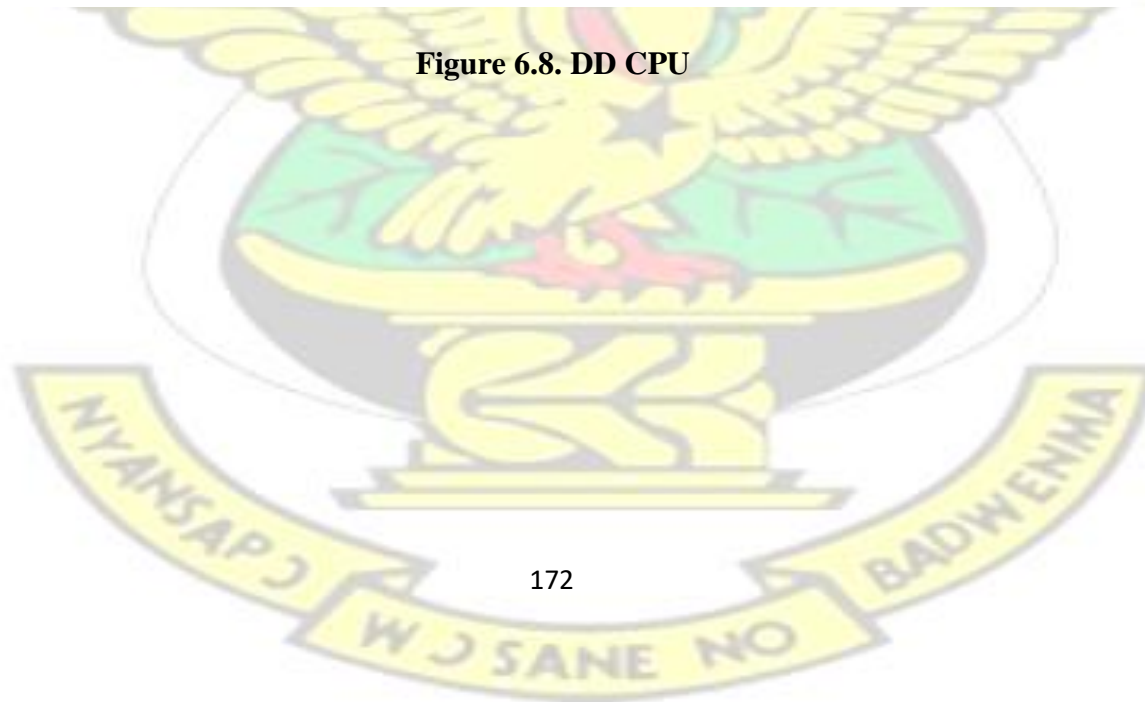
Figure 6.7. DD Benchmark

KNU IST

DD CPU

```
engineer@Trend05:~/Downloads$ dd bs=16k count=102400 oflag=direct if=/dev/zero o
f=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 4.89916 s, 342 MB/s
engineer@Trend05:~/Downloads$ cat /dev/sda3 | pipebench -q > /dev/null
cat: /dev/sda3: No such file or directory
Summary:
Piped 0.00 B in 00h00m00.00s: 0.00 B/second
engineer@Trend05:~/Downloads$ dd bs=16k count=102400 oflag=direct if=/dev/zero o
f=test_data
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 5.26785 s, 318 MB/s
engineer@Trend05:~/Downloads$ dd bs=16K count=102400 iflag=direct if=test_data o
f=/dev/null
102400+0 records in
102400+0 records out
1677721600 bytes (1.7 GB, 1.6 GiB) copied, 3.79158 s, 442 MB/s
```

Figure 6.8. DD CPU

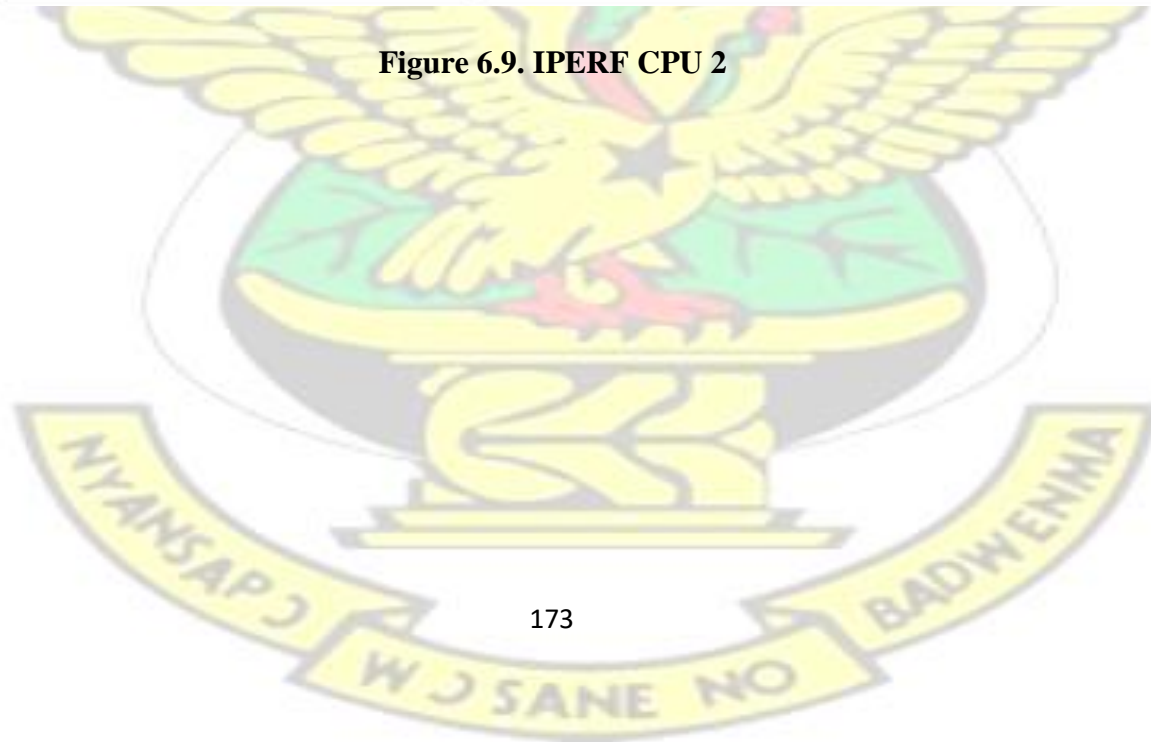


KNU IST

IPERF CPU 2

```
engineer@TrendOS:~/Downloads$ iperf -s -p 8000
-----
Server listening on TCP port 8000
TCP window size: 85.3 KByte (default)
-----
[ 4] local 127.0.0.1 port 8000 connected with 127.0.0.1 port 44452
[ ID] Interval      Transfer    Bandwidth
[ 4] 0.0-10.0 sec  26.7 GBytes 22.9 Gbits/sec
[ 5] local 127.0.0.1 port 8000 connected with 127.0.0.1 port 44454
[ 5] 0.0-10.0 sec  30.9 GBytes 26.6 Gbits/sec
```

Figure 6.9. IPERF CPU 2



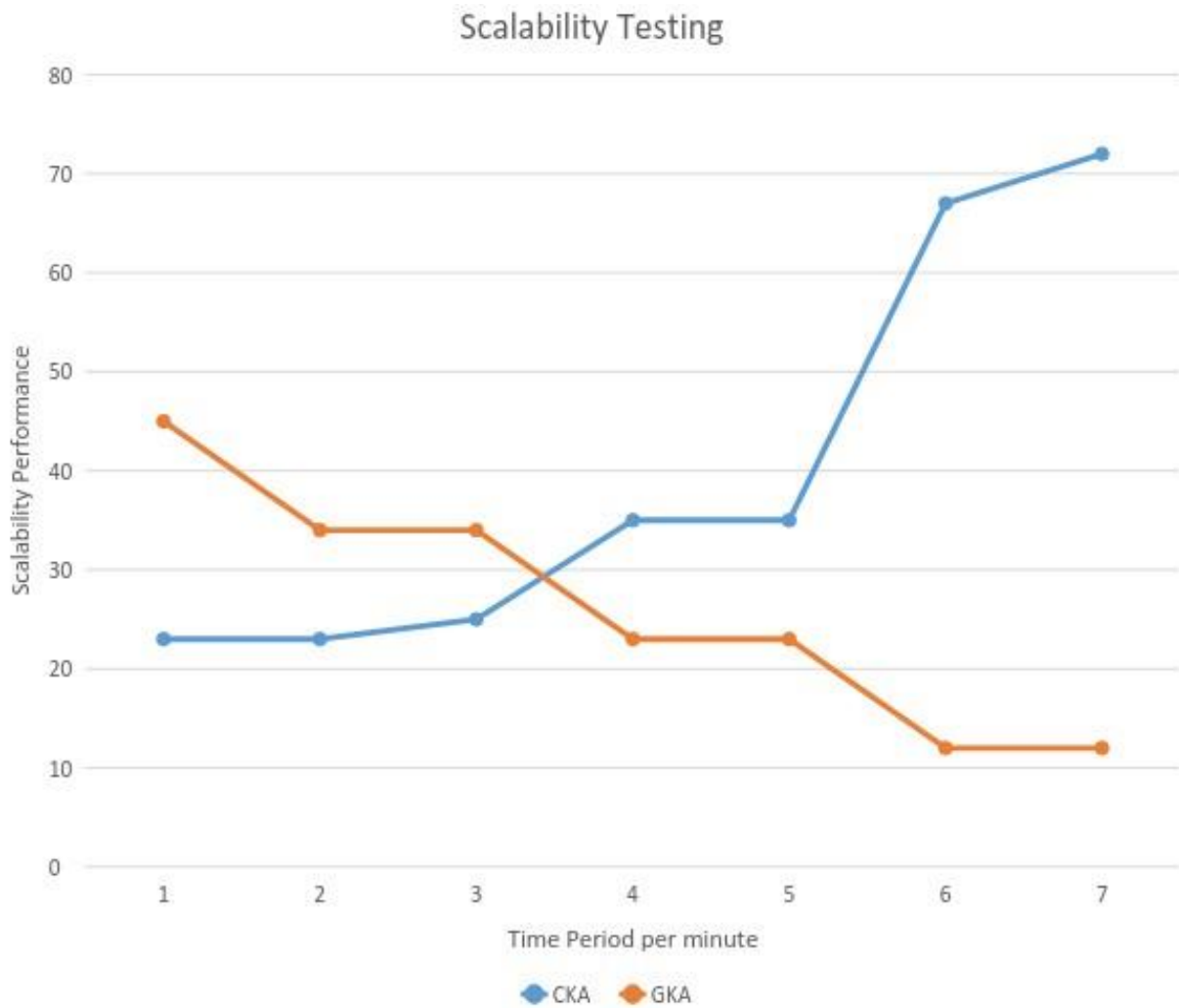


Figure 6.10. Scalability Testing CKA and GKA

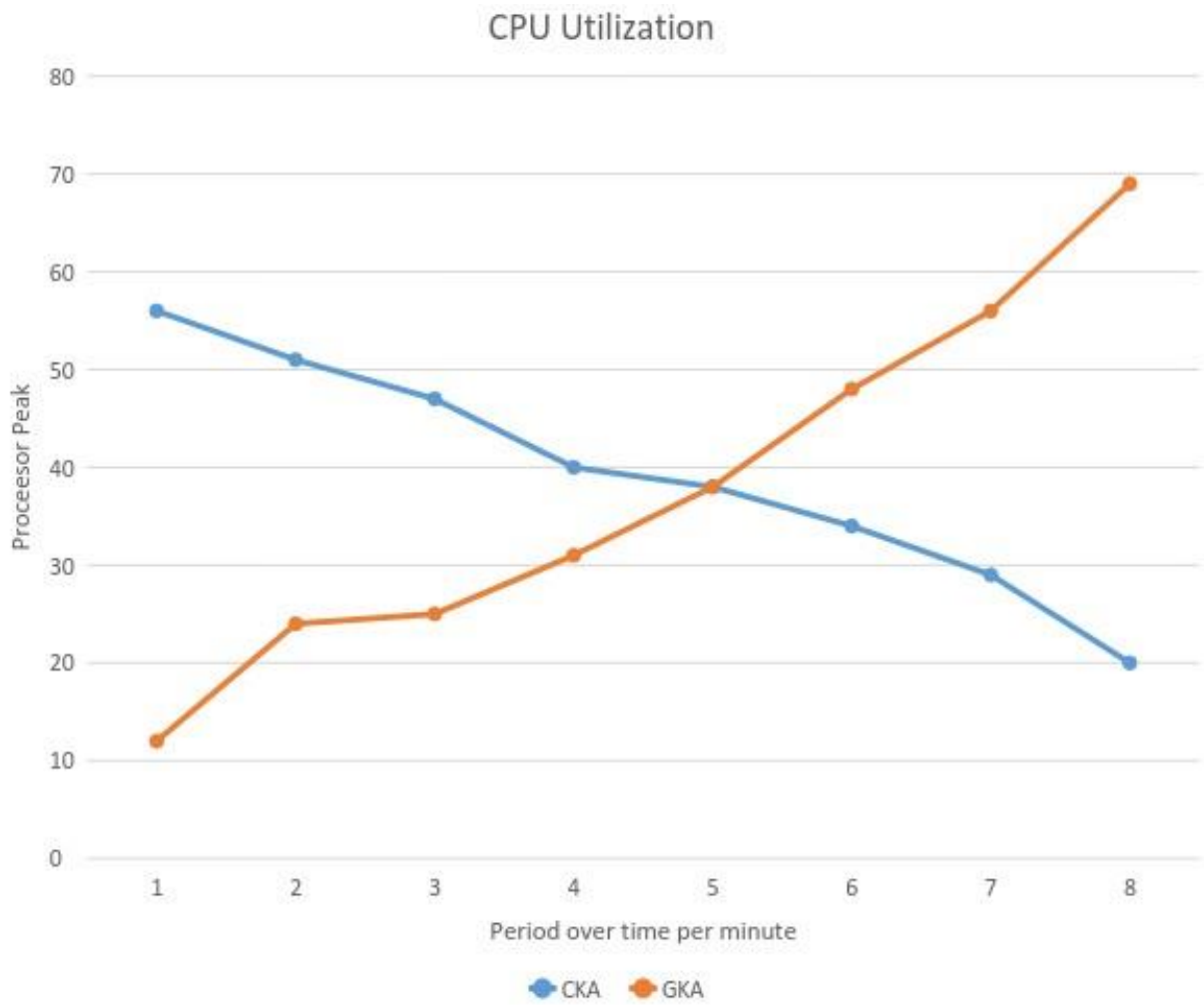


Figure. 6.11. CPU Utilization performance of CKA and GKA

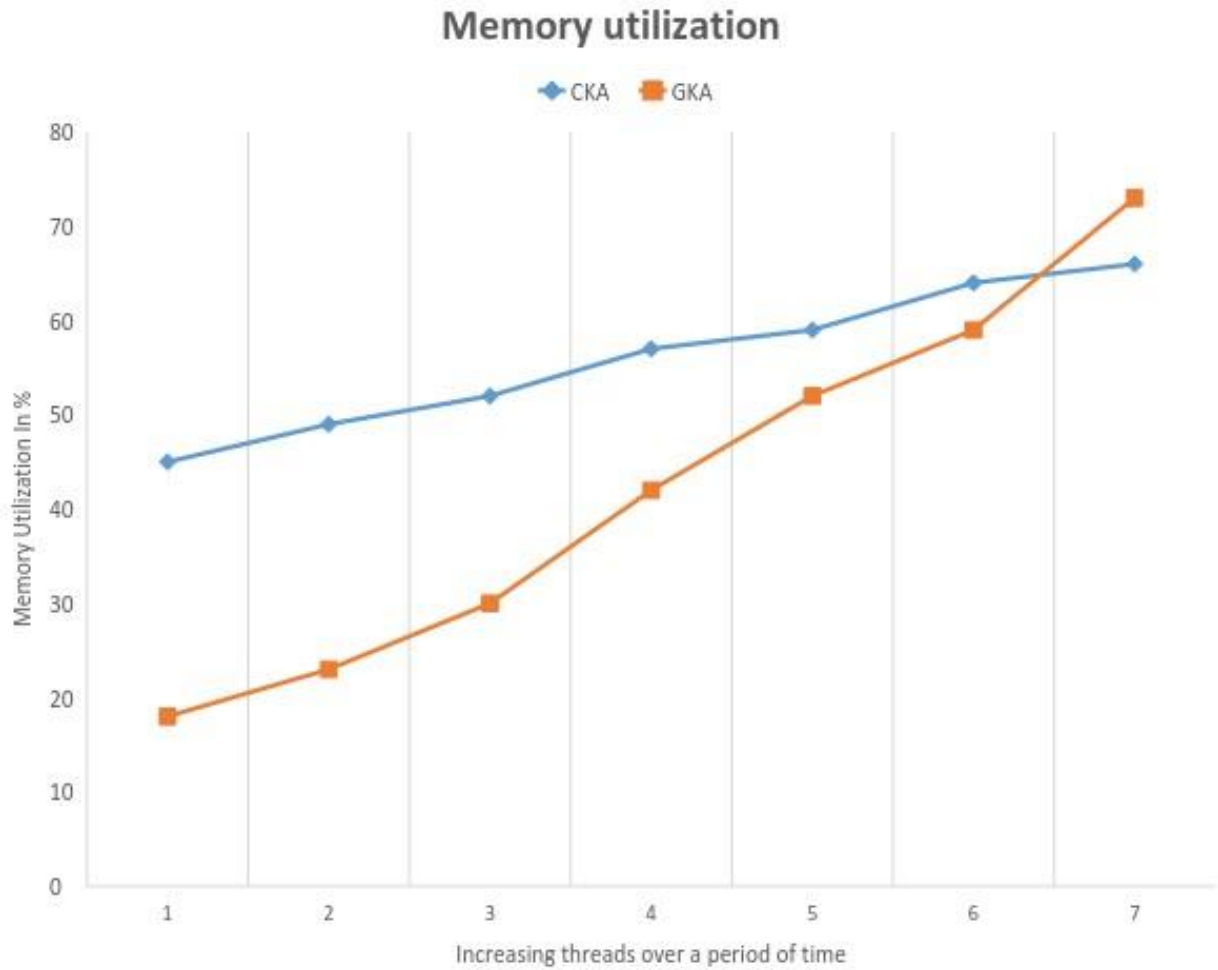


Figure. 6.12 Memory Utilization performance between CKA and GKA

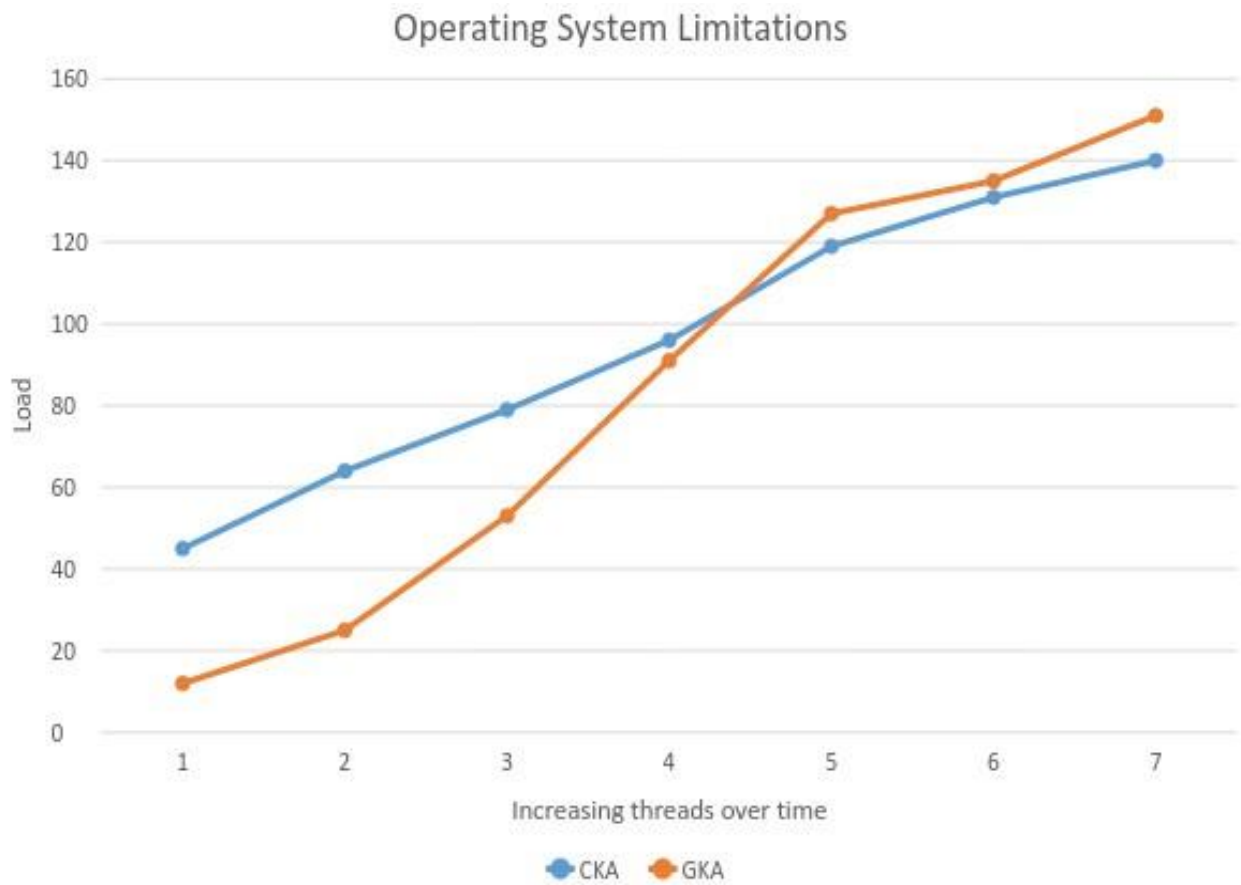


Figure. 6.13 Operating System Limitation between CKA and GKA

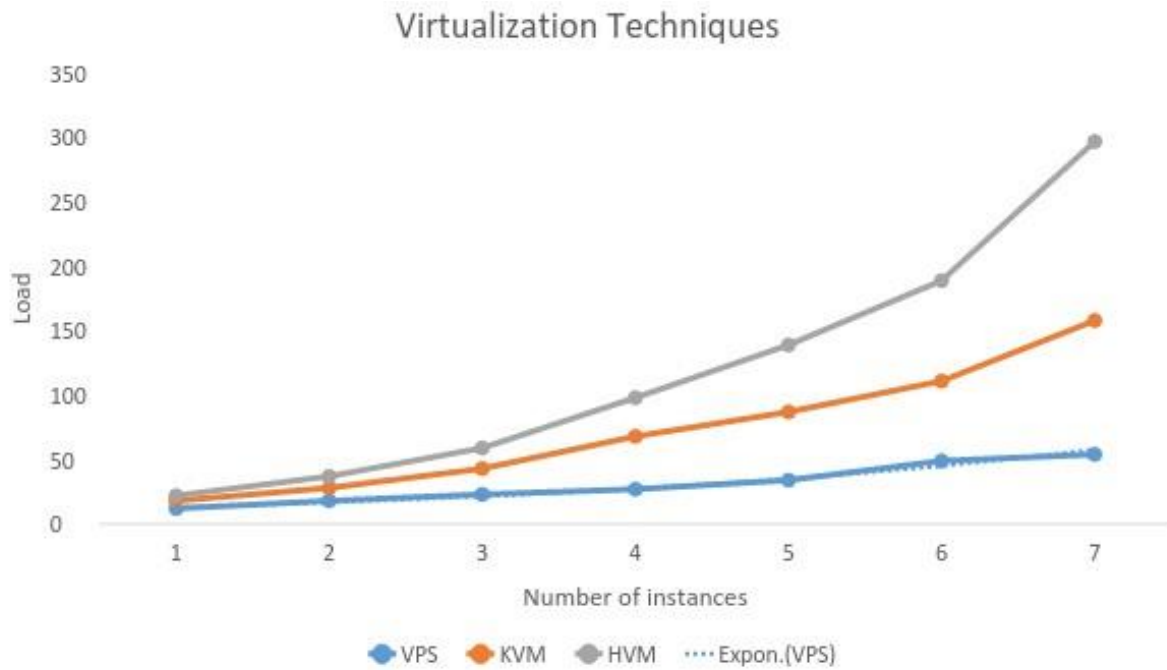


Figure. 6.14 Forms of Virtualization Techniques available against the method adopted



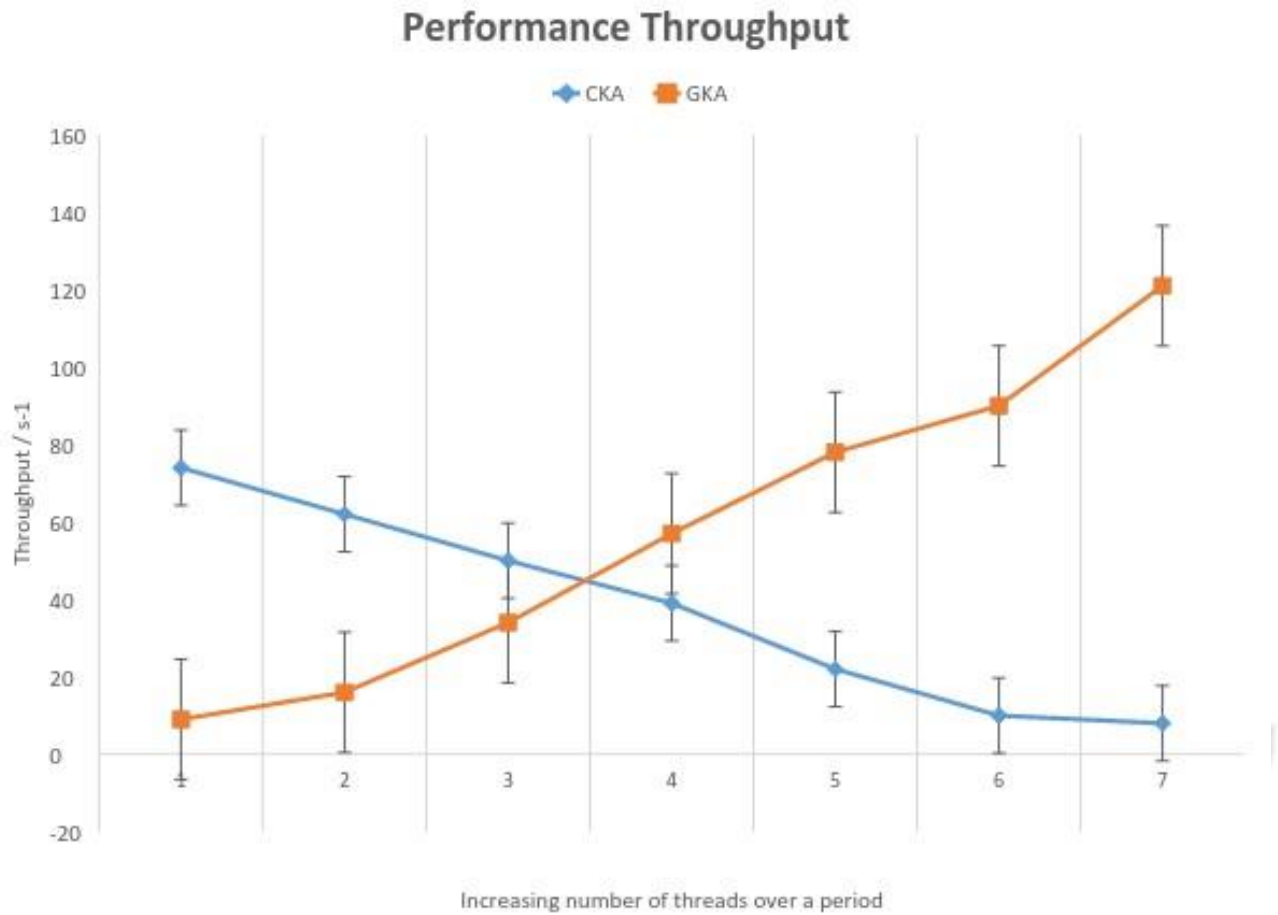
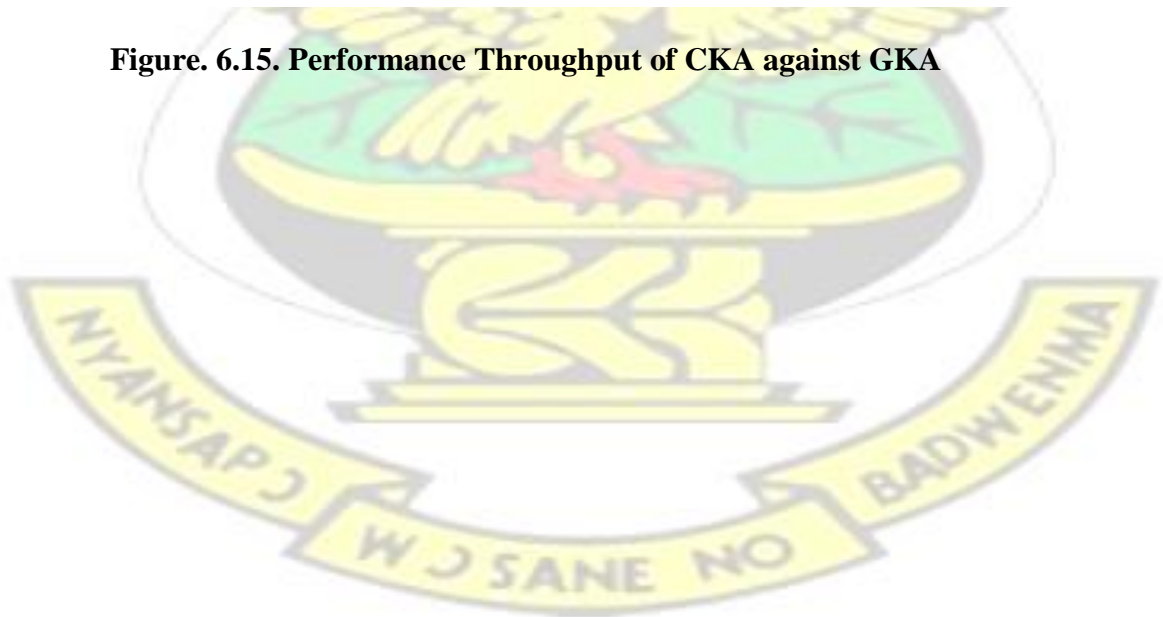


Figure. 6.15. Performance Throughput of CKA against GKA



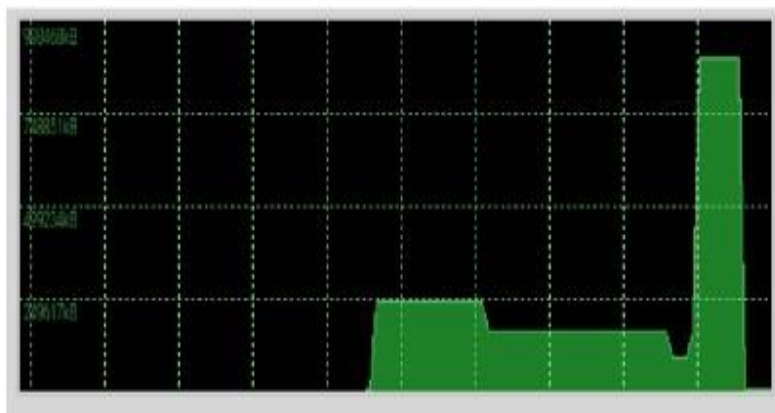
PHORONIX 3



Figure 6.10. Cgroup CPU Usage (Phoronix 3)

✓ Memory utilization

HARDINFO MEMORY



Total Memory	908468 kB
Free Memory	177020 kB
MemAvailable	538108 kB
Buffers	85452 kB
Cached	265828 kB
Cached Swap	12556 kB
Active	324624 kB
Inactive	323028 kB
Active(anon)	142896 kB
Inactive(anon)	172344 kB
Active(file)	181728 kB
Inactive(file)	150684 kB
Unretrievable	32 kB
Mlocked	32 kB
Virtual Memory	1046524 kB
Free Virtual Memory	947944 kB
Dirty	0 kB

Figure 6.11. Hardinfo Memory
HARDINFO MEMORY 2

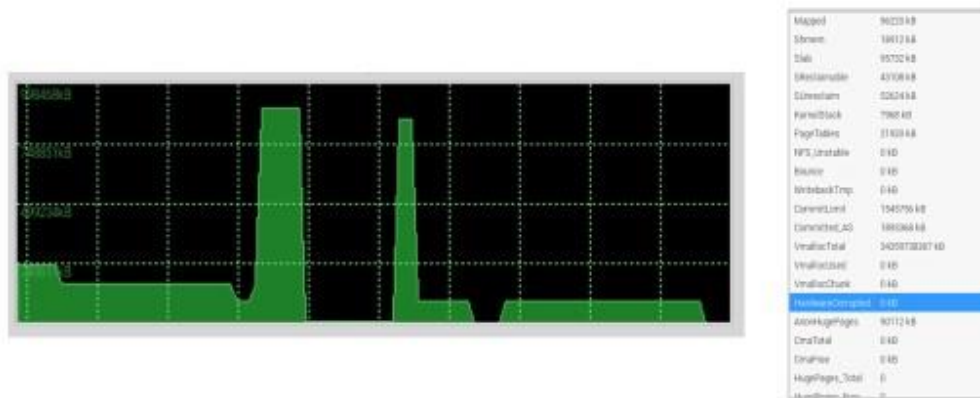


Figure 6.12. Hardinfo Memory 2

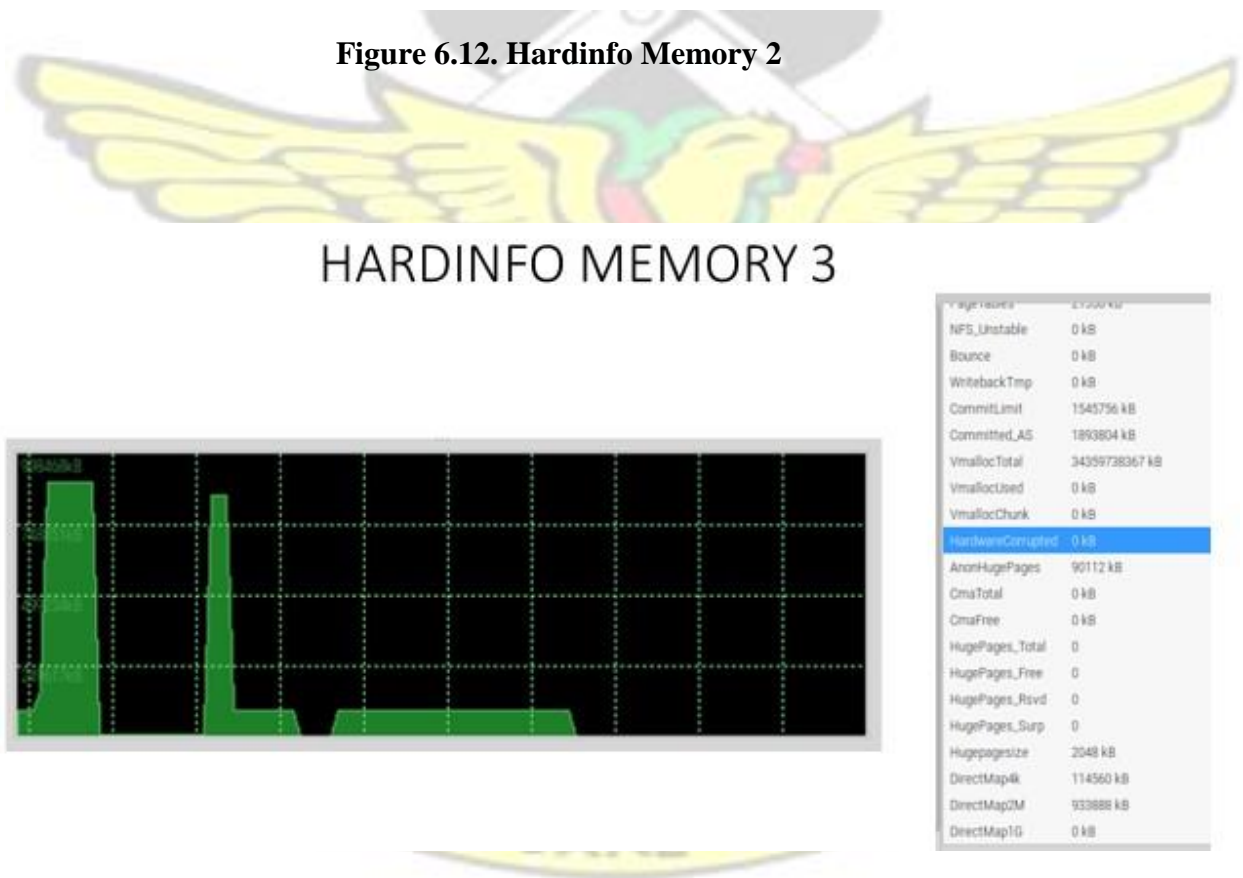


Figure 6.13. Hardinfo Memory 3

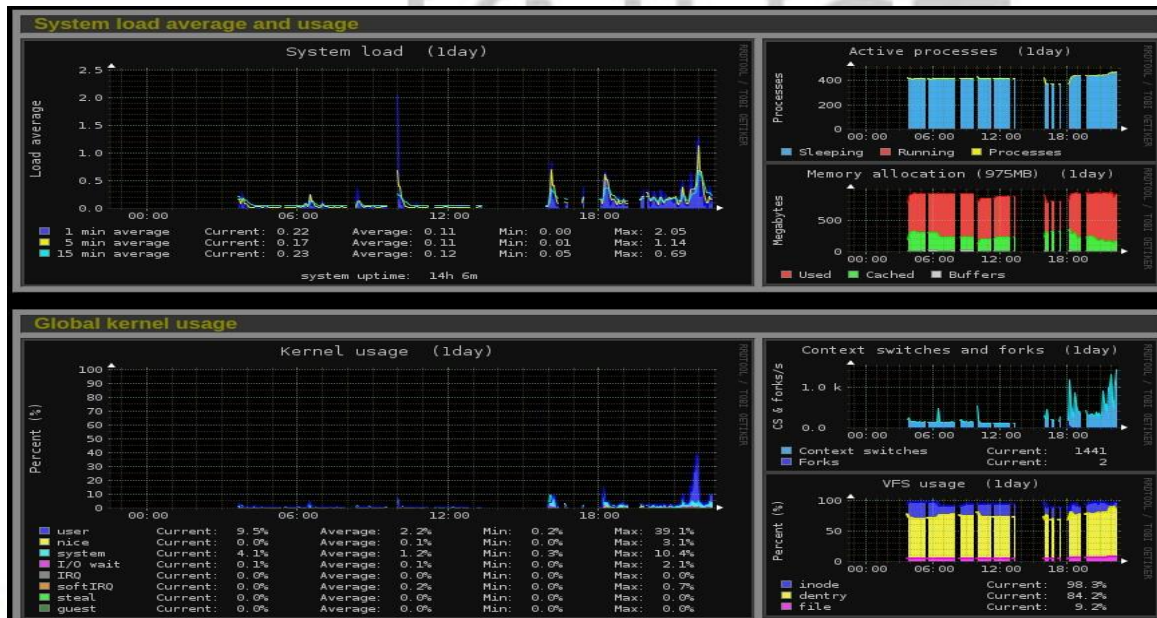


Figure 6.14. Global kernel and System load average usage

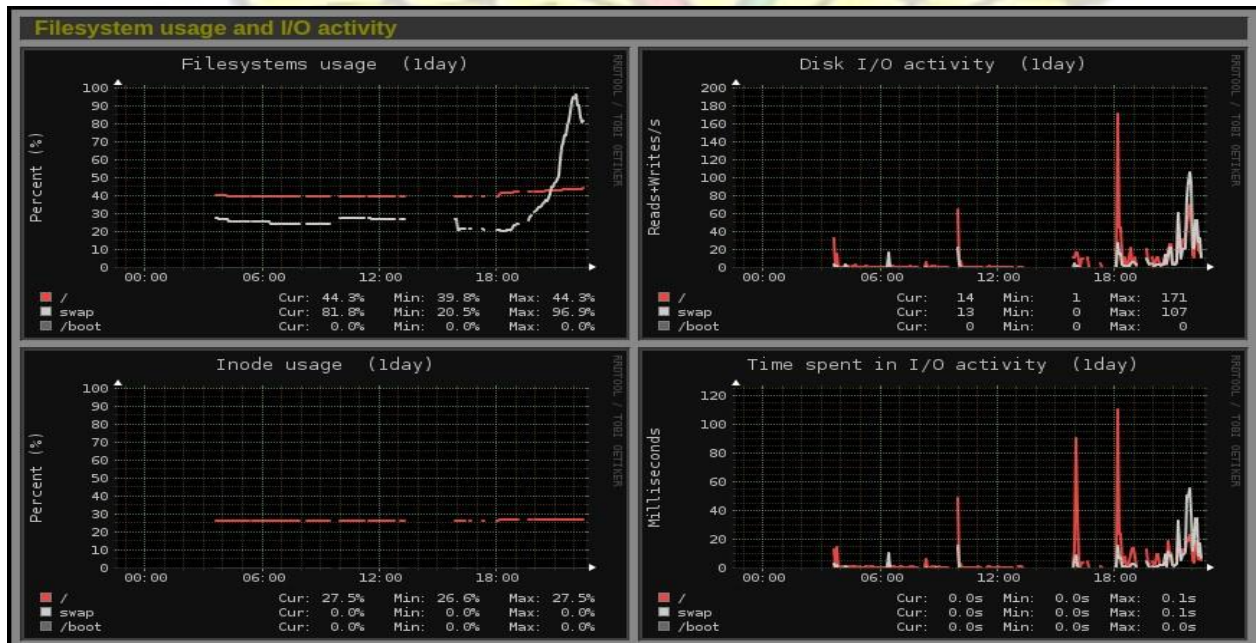


Figure 6.15. Filesystem usage and I/O activity

PHORONIX 5

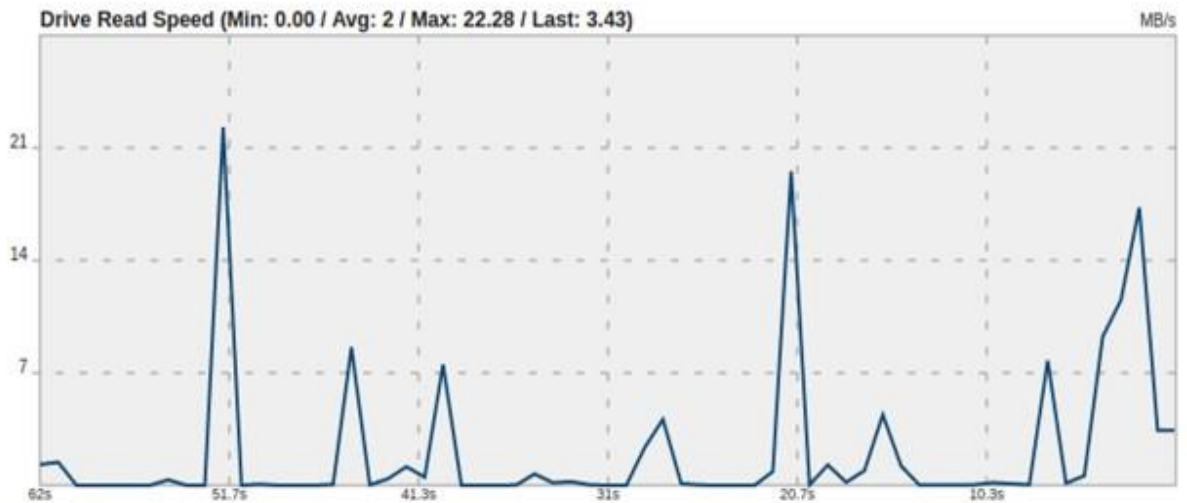


Figure 6.16. Drive Read Speed

PHORONIX 6

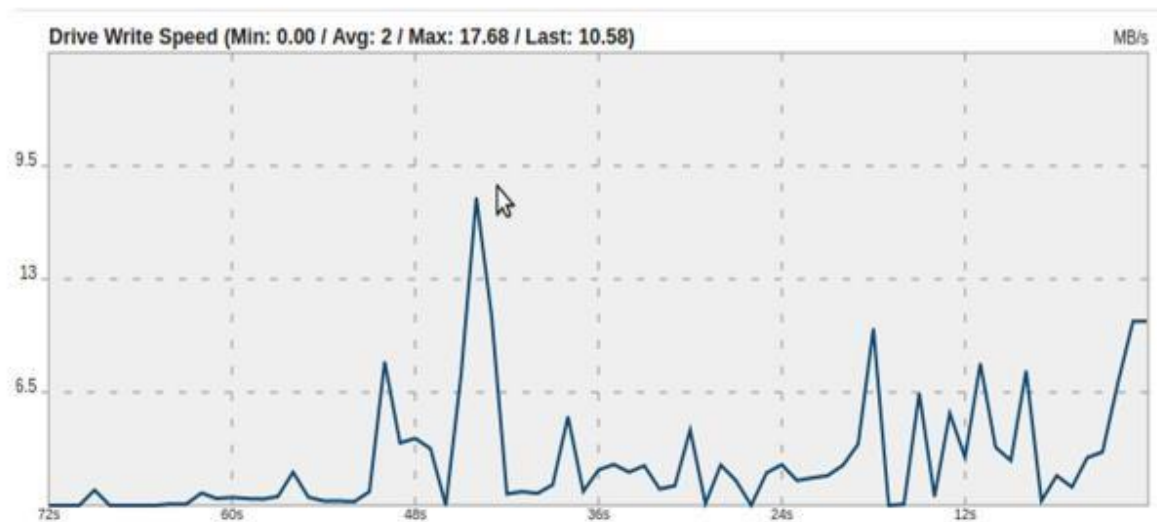


Figure 6.17. Drive Write Speed

```
SYSTEM BENCH
engineer@TrendOS:~$ sysbench --test=cpu --num-threads=4 --cpu-max-prime=9999 run
sysbench 0.4.12: multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 4

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 9999

Test execution summary:
total time: 9.9001s
total number of events: 10000
total time taken by event execution: 39.5669
per-request statistics:
  min: 0.93ms
  avg: 3.96ms
  max: 25.05ms
  approx. 95 percentile: 13.09ms

Threads fairness:
events (avg/stddev): 2500.0000/3.24
execution time (avg/stddev): 9.8917/0.01
```

Figure 6.18. System Bench

✓ Performance Throughout

HD PARM

```
engineer@TrendOS:~/Downloads$ sudo hdparm -Tt /dev/sda*
[sudo] password for engineer:

/dev/sda:
Timing cached reads: 10500 MB in 2.00 seconds = 5252.17 MB/sec
Timing buffered disk reads: 518 MB in 3.01 seconds = 172.31 MB/sec

/dev/sda1:
Timing cached reads: 10630 MB in 2.00 seconds = 5316.73 MB/sec
Timing buffered disk reads: 1196 MB in 3.00 seconds = 398.19 MB/sec
```

Figure 6.19. HD PARM



Figure 6.20. GLX Gears

PIPE BENCH

```
engineer@TrendOS:~$ cat /dev/sdal | pipebench -q > /dev/null
cat: /dev/sdal: Permission denied
Summary:
Piped 0.00 B in 00h00m00.00s: 0.00 B/second
engineer@TrendOS:~$ sudo cat /dev/sdal | pipebench -q > /dev/null
[sudo] password for engineer:
Summary:
Piped 18.99 GB in 00h00m57.00s: 341.29 MB/second
engineer@TrendOS:~$ sudo cat /dev/sda | pipebench -q > /dev/null
Summary:
Piped 20.00 GB in 00h00m49.96s: 409.85 MB/second
engineer@TrendOS:~$ sudo cat /dev/sda5 | pipebench -q > /dev/null
Summary:
Piped 1022.00 MB in 00h00m03.29s: 310.27 MB/second
engineer@TrendOS:~$
```

Figure 6.21. Pipe Bench

STRESS

```
engineer@TrendOS:~$ stress --cpu 4 --timeout 300s
stress: info: [10397] dispatching hogs: 4 cpu, 0 io, 0 vm, 0 hdd
stress: info: [10397] successful run completed in 300s
engineer@TrendOS:~$
```

Figure 6.22. Stress



Figure 6.23. Devices Interrupt Activity



Figure 6.24. Netstat statistics

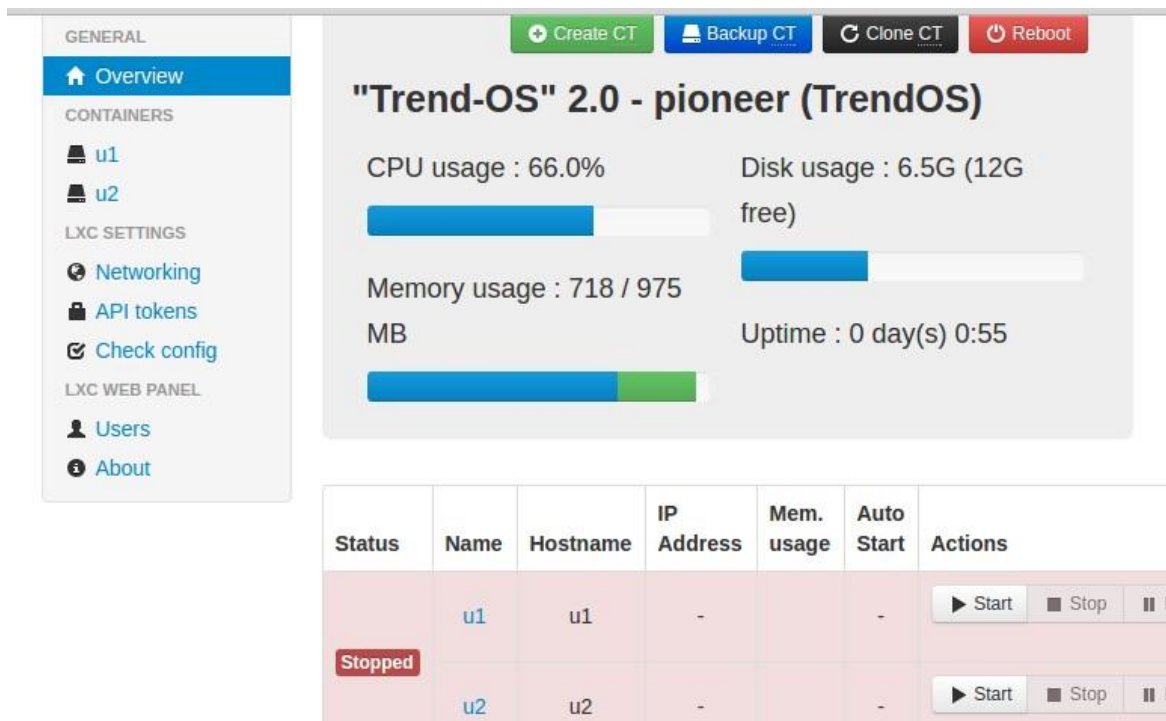
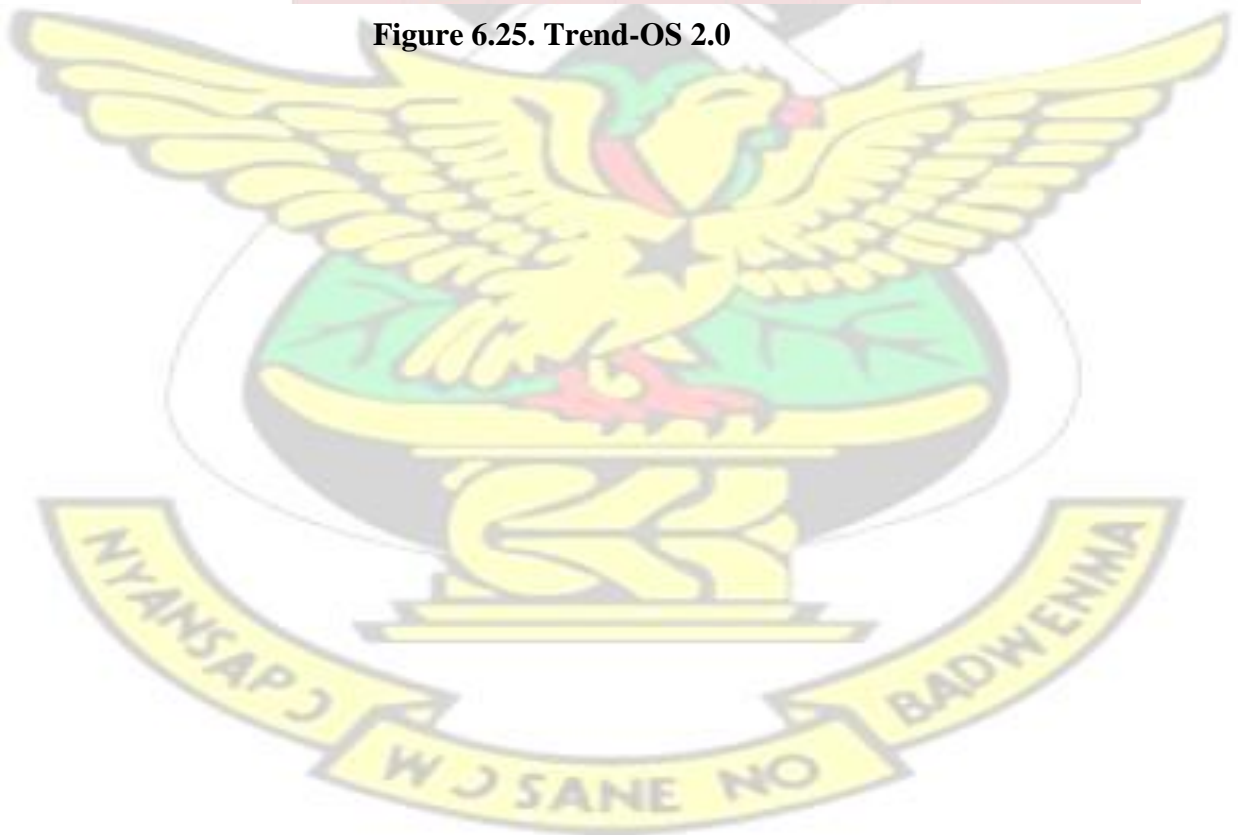


Figure 6.25. Trend-OS 2.0



APPENDIX B

PUBLISHED

ARTICLES

