SECURITY PRIMITIVES IN A HIERARCHICAL GNU/LINUX DRIVER OF A HARDWARE NOC FIREWALL

by

PIPERAKI PARASKEVI

BSc, Technological Educational Institute of Crete, 2013

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF INFORMATICS ENGINEERING

SCHOOL OF APPLIED TECHNOLOGY

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2017

Approved by:

Major Professor
Miltos D. Grammatikakis

Abstract

We develop design methodology for implementing and validating hierarchical GNU/Linux security primitives on top of a hardware Network-on-Chip (NoC) Firewall mechanism embedded in an FPGA development board (ARMv7-based Zedboard). Our open source multi-layer design framework enables modularity and reuse across different use cases and interfaces to system tools, such as Grace, HeatMap, and Secure Event Correlator (SEC) for high-level security visualization and notification services.

Focusing on a realistic out-of-hospital use-case that involves soft real-time ECG data processing on a Hosptial Media Gateway server, we demonstrate how mid-level driver layer can be extended to implement high-level system security primitives for supporting a) *data privacy and anonymity via SHA-3 algorithms,* and b) *NoC-based firewall access control of on–chip BRAM memory from internal denial-of-service attacks by mapping to Linux group id*. Our experimental results on Zedboard demonstrate low overhead of our security primitives.

Περίληψη

Στην παρούσα εργασία αναπτύχθηκε μία μεθοδολογία σχεδίασης για την υλοποίηση και επιβεβαίωση ενός ιεραρχικού GNU/Linux driver που στηρίζεται σε ένα μηχανισμό Δικτύου-σε-τσιπ (NoC) Firewall ενσωματωμένο σε FPGA (ARMv7-based Zedboard). Η σχεδίαση πολλαπλών επιπέδων ανοιχτού κώδικα επιτρέπει την διαμόρφωση και επαναχρησιμοποίηση κατά περίπτωση υπηρεσιών συστήματος και την παράλληλη διασύνδεση με εργαλεία οπτικοποίησης συστήματος, όπως το Grace, HeatMap, και Secure Event Correlator (SEC) για υψηλού επιπέδου υπηρεσίες οπτικοποίησης και ειδοποίησης θεμάτων ασφάλειας.
Εστιάζοντας σε ένα ρεαλιστικό σενάριο Telemedicine που περιλαμβάνει α) μεταφορά σε πραγματικό χρόνο καρδιολογικών δεδομένων ασθενών, ειδικότερα σημάτων ηλεκτροκαρδιογραφήματος που λαμβάνονται από wearable pulse sensors (ST BodyGateway), σε ένα διακομιστή Hospital Media Gateway για ανάλυση και οπτικοποίηση, παρουσιάζουμε μια επέκταση του μεσαίο επιπέδου του Hierarhical Driver για για υποστήριξη α) προστασίας προσωπικών δεδομένων μέσω ανωνυμίας ασθενών (SHA-3 hash), και β) έλεγχο πρόσβασης μέσω Linux group id της κρίσιμης πληροφορίας που βρίσκεται στη BRAM μέσω χρήσης τείχους προστασίας στο Δίκτυο-σε-τσιπ (NoC) που προστατεύει τα δεδομένα από εσωτερικές επιθέσεις πρόσβασης ή άρνησης εξυπηρέτησης. Τα πειραματικά αποτελέσματά μας για το Zedboard αποδεικνύουν χαμηλό κόστος των πρωτοκόλλων ασφαλείας.

Table of Contents

Abstract 2

Chapter 1 – Introduction 9

Chapter 2 - Anatomy of Embedded Security 10

2.1 Embedded Security 10

2.1.1 Vulnerabilities in Embedded Systems 10

2.2 System and Network Security and Privacy 11

2.2.1 System Security 11

2.2.2 System Privacy 13

2.3 Software Vulnerability in Application Layer - DDoS Attacks 13

2.3.1 An overview of flooding attacks 14

2.3.2 Denial of Service Attacks 14

2.3.3 Distributed Denial of Service attack 15

2.4 Authentication and Access Control 17

2.4.1 Authentication 17

2.4.2 User permissions 17

2.5 Log Analysis 18

Chapter 3 - NoC Firewall 19

3.1 Firewall Setup and Access Request via Firewall 19

3.2 Linux Driver – Hardware implementation 23

3.3 Hierarchical Design of NoC Firewall Driver 26

3.3.1 Low Level Driver API 26

3.3.2 Mid-Level Driver API 26

3.3.3 High-Level Driver – API 34

3.3.3.1 Creating users 34

3.3.3.2 User_mode File 35

3.3.3.3 System Administrator Privacy File (for Healthcare Scenario) 42

3.3.3.4 FWGroup Privacy File (for Healthcare Scenario) 44

3.3.3.5 Security Overheads 46

3.4 Hierarchical Driver Validation 47

Chapter 4 - Experimental Framework: Security in ECG Processing 49

4.1 Patient App 49

4.2 Zedboard Server for ECG Processing 50
4.2.1 Doctor Application 51
4.3 DDoS Attack via NoC - Cache Thrashing 52
4.3.1 Kernel Level Attack: exploit_firewall module 52
4.4 High-Level Security 58
4.4.1 Event Monitoring Tools 58
4.4.2 Real time monitoring tools 58
4.4.3 Experimental Study: SEC for Monitoring a DDoS Attack via NoC Firewall 59
4.4.3.1 Access to Brams: FwAccess module 59
4.4.3.2 Visualization and Alert using SEC 62
Chapter 5 - Conclusions and Future Work 64
References 65

List of Figures

Figure 1: An industrial embedded device[1] ... 10
Figure 2: Taxonomy of attacks on embedded systems [3] ... 11
Figure 3: OSI & TCP/IP model [7] ... 14
Figure 4: DoS Attack – one attacker [11] ... 15
Figure 5: DDoS Attack with thousands of requests [11] ... 15
Figure 6: Agent-Handler model ... 16
Figure 7: Internet Relay Chat (IRC) - based model ... 16
Figure 8: User permissions ... 17
Figure 9: Butterfly NoC with 2x2 switches ... 19
Figure 10: Setup low address range ... 20
Figure 11: Setup high address range ... 20
Figure 12: Setup rule address rage in Simple Mode ... 20
Figure 13: Low address range register (Setup) ... 21
Figure 14: High address range register (setup) ... 21
Figure 15: Rule address rage in Extended Mode register (setup) ... 21
Figure 16: Passed packets register (monitor). ... 22
Figure 17: Fifo dropped packets register (monitor). ... 22
Figure 18: Fw dropped packets register (monitoring statistics). ... 23
Figure 19: Access request in Simple/Extended Mode; SRC field used only in Extended Mode. ... 23
Figure 20: 4-Port Firewall Multicast Router – Synthesis on Zedboard ... 25
Figure 21: The NoC Firewall: a) circuit and b) utilization of the FPGA ... 25
Figure 22: Hierarchical Linux Driver for our Firewall ... 26
Figure 23: The NoCFW Linux driver hierarchy with full system administrator privileges. ... 42
Figure 24: The NoCFW Linux driver hierarchy with basic group (fwgroup) privileges ... 45
Figure 25: The healthcare application. ... 49
Figure 26: Doctor App Workflow Diagram ... 51
Figure 27: GUI for Patient (BT and WiFi) and Doctor App (authentication, and WiFi) ... 51
Figure 28: Kernel Level Attacks ... 52
Figure 29: Communication of exploit firewall module with NoC firewall module ... 53
Figure 30: Average BRAM access delay (ns) for different exploit scenarios ... 57
Figure 31: BRAM access delay vs Time (in ns) for different exploit scenarios with xmgrace ... 57
Figure 32: Communication of fw_access firewall module with NoC firewall module ... 60
Figure 33: Brams Accessing heat Map – 4 accesses in Bram1 ... 62
Figure 34: Brams Accessing Heat Map – 2 accesses in Bram1; color changes with access count ... 63

List of Tables

Table 1: Hardware Costs In NoC & NoCFW Implementations 24
Table 2: Compare - Hardware Costs In NoC & NoCFW Implementations 24
Table 3: Groups and users in the healthcare scenario 34

Acknowledgements

I would like to thank my parents for supporting me all these years for my studies.

Moreover, I would specifically like to thank my thesis advisor Dr. Grammatikakis Miltiadis for his guidance, patience and knowledge he gave me. I am also thankful to members of the Artificial Intelligence and System Engineering Laboratory (AISE Lab) of the Technological Educational Institute of Crete for their assistance during my thesis. Specifically, I would like to thank Antonis Papagrigoriou who provided the NoC firewall that he designed in order for me to develop the Linux infrastructure (drivers, high level application) in my thesis and helped me during my thesis. Finally, I would like to thank George Tsamis for providing access to the soft real time Smartphone ECG application (documentation, code) where development of our software hierarchical security primitives and services took place.

Chapter 1 – Introduction

The issue of data privacy and security in eHealth is of paramount importance, since security risks increase when patient data is being disclosed in eHealth systems and networks. By all means, users must rest reassured that private data is kept confidential when accessing different electronic healthcare services. In this context, a well designed security scheme is necessary to protect users by ensuring the privacy of transferred or processed sensitive data.

More specifically, focusing on an ideal data privacy protection scheme that thwarts threats from on-chip logical attacks, and assuming that each patient has a unique eHealth record (patient No) that could be distributed to his doctor (simple user) freely or upon demand, we attempt to ensure a) that each doctor can access only patient data that he has permission to, and b) patient identity (including wearable pulse sensor MAC address) are hidden by using patient No (and possibly a key) for validation. Thus, in this thesis, we attempt to design and implement modular firmware security solutions that

- support anonymity of patient data in FPGA memory by hashing to an appropriate location in FPGA memory (BRAM) using the unique patient No.
- protect patient data (hashed in system memory) when it must be exchanged with physicians for processing and visualization by providing hardware-based access control mechanisms. More specifically, we prevent access from malicious or unauthorized physicians by setting firewall rules (based on group ID) in a hardware-based NoC firewall that protects all BRAMs, Therefore, malicious or unauthorized logical processes cannot read or write patient data in the eHealth system by performing a surface attack.

This thesis develops a generic hierarchical GNU/Linux driver infrastructure on top of a hardware NoC firewall architecture supporting memory access protection. This framework can be used to support data privacy and security of healthcare technologies and beyond. In particular, we focus on an actual soft real-time out-of-hospital use-case with a hospital media gateway (cloud server) and examine performance overheads for supporting data privacy and memory protection. The application not only presents graphically the patient heartbeat signal (electrocardiogram, or ECG) from the ST BodyGateway wearable pulse sensor, but also uses calibrated diagnostic subsystems based on open source software (including digital filters) to identify cardiac events (e.g. non-fatal ventricular arrhythmias) and asynchronously annotate the ECG signal for the physician. Experimental results on Zedboard demonstrate a relatively small overhead of our high-level security services that can be merged with high-level security visualization based on Grace, HeatMap and SEC tool.

.

Chapter 2 - Anatomy of Embedded Security

This chapter has organized as follows: Section 2.1 defines an embedded system, and examines vulnerabilities in embedded system world. Section 2.2 separates in two parts. In the first part we focus on system security and the security principles that a system must have to be secure. The second part refers to system privacy issues. Section 2.3 describes the vulnerabilities in application layer and examines DDoS attacks. Finally, in Section 2.4 and 2.5 we discuss user permissions in Unix and existing mechanisms for logging analysis.

2.1 Embedded Security

Embedded system is a device that consists of hardware (electrical and mechanical parts) and software components with different security requirements[1]:

- Confidentiality is related to keeping sensitive system information safe from malicious users. Specifically, confidentiality prevents unauthorized users from accessing files or system data [2].
- Integrity is related to consistency, accurancy and trustworthiness of data. Here, the aim is to protect data from being edited [2].
- Availability refers to an embedded system that is accessible when required. For instance, availability ensures that denial of service attacks cannot succeed [2].

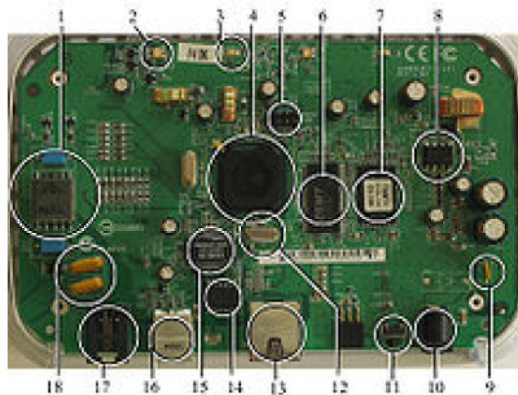

Figure 1: An industrial embedded device[1]

2.1.1 Vulnerabilities in Embedded Systems

In embedded systems, there are two layers of attacks as shown in Figure 2.

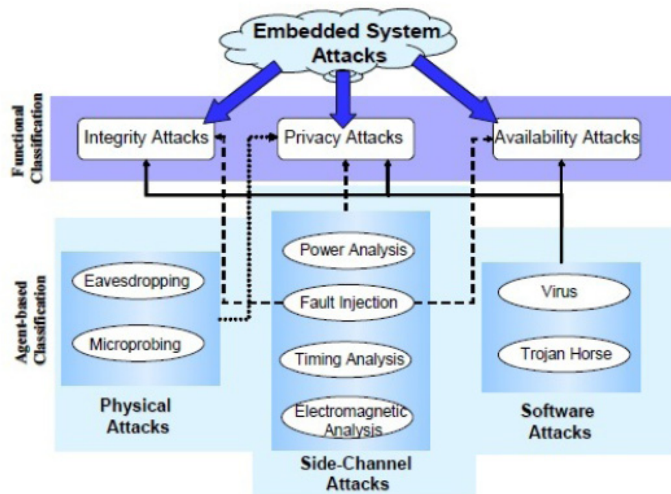

Figure 2: Taxonomy of attacks on embedded systems [3]

Layer1: The first layer consists of three main types of attack classified according to their functional objectives [3]:

- Privacy attacks: aim to obtain information from stored data in an embedded system.
- Integrity attacks: target editing data or code in an embedded system.
- Availability attacks: change the normal operating on embedded system by mis-appropriating system resources that they are unavailable for normal operation.

Layer2: The second layer also consists of three main types of attack depending on the agents used to execute the attacks [3]:

- Software (or logical) attacks: attackers spread the virus from one computer to another and aim to harm the software equipment and files. Software attacks refer to computer virus, worms, such as Trojan horses, buffer overflow vulnerability attacks and denial of service attacks. In the latter attacks, the attacker sends data repeatedly to minimize system performance.
- Physical or invasive attacks: Here, attackers spread the virus into the system like chip, board or in system level.
- Side-channel attacks: attackers can compromise the system by monitoring different system metrics, such as execution time and power consumption, to extract information on system characteristics or behavior.

2.2 System and Network Security and Privacy

2.2.1 System Security

Security in general, is related to the aspects of confidentiality, integrity, availability and accountability as referred previously. Moreover, besides traditional security concerns that involve network intrusion attacks, system security incorporates several challenges, such as Virtual Machine level attack, vulnerability, phishing, authorization, authentication and expanded network attack. According to Basin et al [4], in order to succeed in these goals, it is useful to examine the following best practices.

- Lower mechanism cost: Basin and others call this principle "Simplicity". This principle makes the system easier to develop, operate, analyze and maintain, and therefore more trustworthy and less prone to contain flaws.

- Open design: Basin supports that private tasks deteriorate system security. If When aspects are public, no one is curious for changing the system or re-engineering a design or implementation.
- Compartmentalization: This principle separates system security into zones or parts in order to protect system resources. This means that, if a zone or group of an area is infected, other parts are not infected. The same principle also applies to performance isolation. Basin also provides a few examples of compartmentalization.
  1. The use of distributed systems can minimize the possibility of a successful attack. For instance, this choice is relative to protecting data stored in a database by distributing data to different servers.
  2. Another choice relates to separating applications in different physical machines using Kernel/User mode or virtualization software, such as VirtualBox, VMware etc.
  3. In another example, firewalls can be used to split network access to separate zones (multicompartments). This approach is related to both off- and on-chip network firewalls.
  4. Using compartmentalization in software development that consists of different languages mechanisms like encapsulation, modularization and others.
- Minimum Exposure: The aim of this principle is to minimize the possibility for illegitimate users to post an attack. There are three actions that can be used:
  1. Disable system devices that are not needed, such as Bluetooth, WLAN and others. This can minimize the possibility of external attacks.
  2. Minimize the possibility of an attacker to collect system information. A typical example could be a web server that can provide information, such as versioning.
  3. Reduce the time available that illegitimate users can use. For example, brute-force attack refers to an attack that an attacker tests different user names and passwords to enter a system. In this point, security mechanisms could lock the account after some tries.
- Least Privilege: This principle supports that system privileges do not need to be used by other users. Next, Examples include
  1. Not allowing other users to access other resources, such as files or computers. For instance, access control is implemented to allow a user to work only on specific files.
  2. Not give other users root or specific user passwords to install a program to a computer. A good solution for this would be to create another user account.
  3. Protect network access. A well-designed firewall can solve network security issues.
- Minimum Trust and Maximum Trustworthiness: This principle separates trust from trustworthiness in the system. More specifically, when a user trusts a system without checking it, then malicious behavior may appear. Therefore, it is safer to minimize the trust and expectations one has from a system or maximize the system trustworthiness. The first principle means that users validate a system before use it.
- Return to secure mode: This safety principle must return to the initial mode when a negative event happens in a system like a failure. A good solution for this action could be to enable a security mechanism or activate extra precautions (e.g. a firewall rule) to revert the system in the initial/secure state if a failure, system crash or other error occurs.
- Complete mediation ensures system security during all times by controlling each access to a

security-relevant object independent of the system state (normal operation, shutdown, maintenance mode, or failure). Hence access control to different subsystems must be operational, no matter what happens to a system. Some examples are:

1. A memory management unit (MMU) can have a complete mediation by controlling every memory request in memory.
2. The technology for encrypting the file system also ensures complete mediation.

- No single point of failure or defense in depth: an ideal system security policy must contain protection in two or more places, so that if one security mechanism fails in a point, the other points could still be controlled by other security mechanisms so that "*no single accident, deception, or breach of trust is sufficient to compromise the protected information*". For example, many organizations instalantivirus in different points, such as mail servers, client computers, file server. Hence, if the antivirus in a mail server crashed due to some attack, antivirus protection of client computers can remain active
- Traceability aims to keep a log file from past traces in the system. A trace potentially helps locate suspicious records. As pointed out by Basin et al [4] log information is essential in order to *detect operational errors and deliberate attacks, to identify the approach taken by the adversary, to minimize the spread of such effects to other systems, to undo certain effects, and to identify the source of an attack.*
- Usability refers to the ease of use of security mechanisms.

2.2.2 System Privacy

This aspect establishes that all information will be kept classified. Notice that establishing a level of privacy in quantitative terms may relate to probability calculations. Privacy in general is related to the important aspects of Anonymity, Pseudonymity, Unlinkability, Unobservability.

- Anonymity: refers to the state of being anonymous or virtually invisible, operating online without being tracked, using a resource or service without disclosing identity information [5], and being unidentifiable within a group [6].
- Pseudonymity: expresses the state of using one or more aliases instead of a user's real identity. However, under certain circumstances, such as when the user abuses the system's terms of use or breaks the law, this data can be revealed [5].
- Unlinkability: refers to the inability to link related information and is vital for protecting user privacy. Unlinkability also refers to being impossible for a third party to verify the exact two parties that engage in a communication, or an attacker being unable to link information with the person that processes it [5].
- Unobservability: expresses protection provided to users from being observed or tracked, when accessing a service or browsing the Internet. An attacker usually aims to observe these actions in order to reveal identifiable user information [5].

2.3 Software Vulnerability in Application Layer - DDoS Attacks

Layered security is paramount in understanding how and where flooding attacks occur. In communication systems there are two models. The first consists of seven layers known as OSI (Open Systems Interconnections) model used for network communications. The second, TCP/IP model (Internet Protocol) has four layers namely application, transport, internet and link. Next, we focus on vulnerabilities in application layer, specifically flooding attacks used in both OSI and TCP/IP model, as in Figure 3.

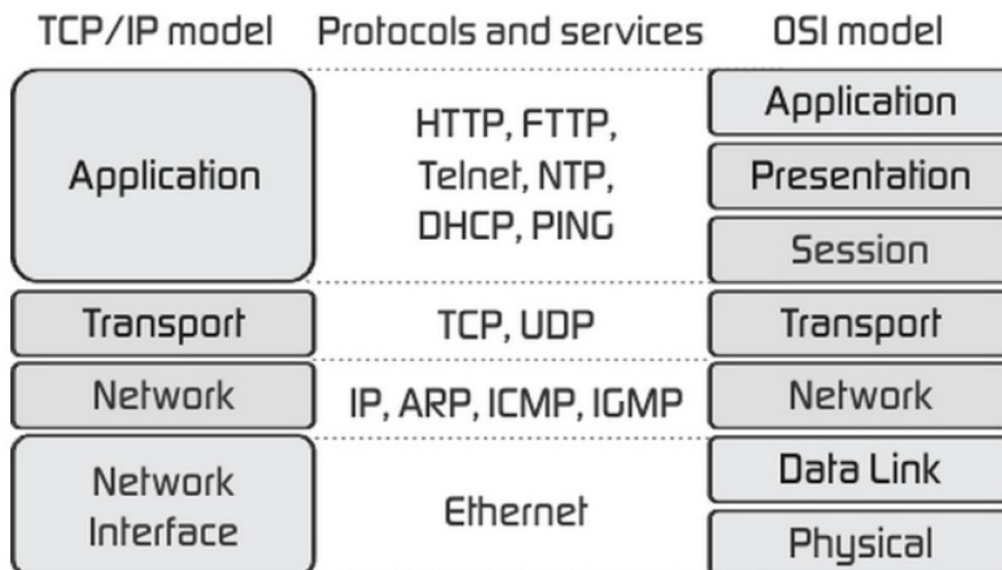

Figure 3: OSI & TCP/IP model [7]

2.3.1 An overview of flooding attacks

In 1980, a piece of malware code appeared spreading through floppy disks that adds itself in many programs, including operating systems. At the same period, another virus was incorporated in programs [8] According to John Howard & George Weaver in 1988, the first steps of DoS attacks took place on the Internet. In 1992, hackers could login by sniffing to an operating system as the real user. These DoS attacks did not usually aim to directly destroy data, but rather to have access to a computer or network resources [9]. Four years later, SYN Flood attacks aimed to make a system unable to serve other users. Over the period of 1989 to 1995 CERT measured 104 DoS attacks on the Internet and 39 DoS attacks on root and account level [9]. Since 1997, flooding attacks have increased at an exponential rate. One year later, DDoS (Distributed Denial of Service) attacks appeared and different DDoS attack scripts and/or detections tools, such as trinoo, TFN, stacheldraht were created. In fact, during this time a first workshop took place with the name "Distributed System Intruder Tools".

In 2000, DDoS attacks threatened routers. The same year, on February, famous sites include yahoo, Amazon, Dell etc. was infected with DDoS attacks by Michael Calce known as Mafia Boy [10]. According to an internet measure study, in 2001 detecting 4,000 attacks per week [8].

2.3.2 Denial of Service Attacks

A Denial of Service (DoS) attack is an attack that aims to make resources of a computer service (e.g. cache, memory, or CPU) or network service (e.g. website) unable to accept other connections or serve other users. The DoS attack scenario aims to collapse services (vulnerability attacks) and flood them (flooding attacks).

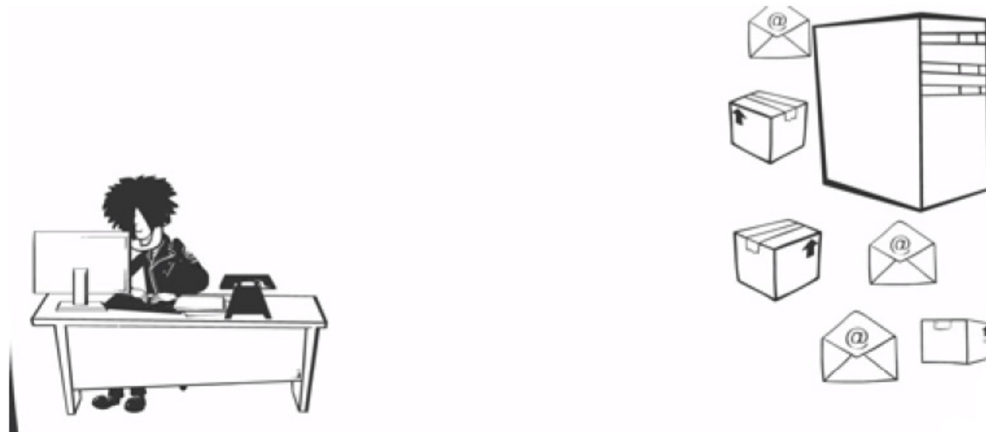

Figure 4: DoS Attack – one attacker [11]

A collapse service attack refers to vulnerability DoS attacks. The aim of these attacks is to take advantage of weaknesses (or vulnerabilities) in services that have been installed in order to gain access to system resources [12]. A typical example is a skillfully constructed fragmented Internet Protocol (IP) datagram that may crash a system due to a serious fault in operating system software [13].

With flooding attacks, the attacker may exercise brute force methods to attack at different levels.

1. Network/transport level: Refers to user connectivity. The attacker sends useless data, such as junk packets, to the victim in order to waste the network bandwidth or router capacity.
2. Application level: Refers to user services. The attacker sends useless data to the victim in order to fill the memory, disk/database, and increase memory, CPU, or I/O bandwidth[14].

In order become more effective a DoS attack can be organized from different sources simultaneously (Distributed DoS, DDoS) .

2.3.3 Distributed Denial of Service attack

A Distributed Denial of Service (DDoS) attack is a coordinated attack on the availability of services of a target system or network that is launched indirectly through many compromised computing systems, e.g Botnets [13]; notice that the word botnet emanates from the words robot and network and refers to network connected-computers that send malware data (viruses) to other computers that belong to the same network.

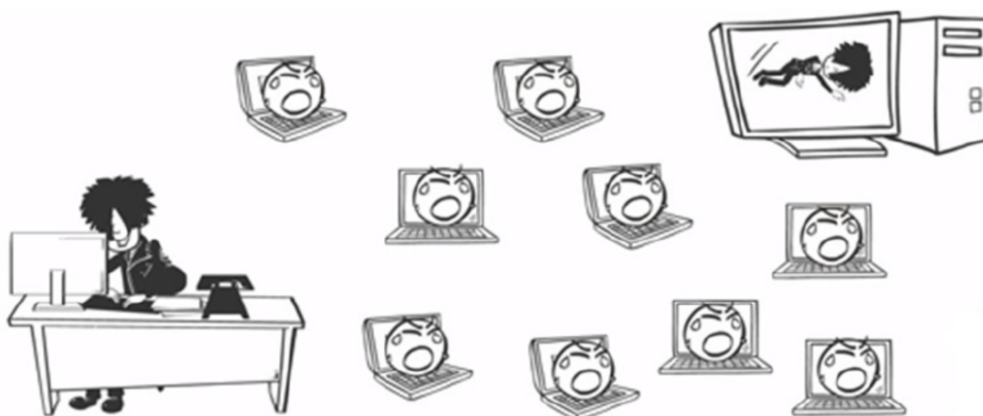

Figure 5: DDoS Attack with thousands of requests [11]

At present, many DDoS attacks use Botnets. They can be separated in three types.

1. An Agent-Handler model uses the concepts of attackers, handlers, agents or zombies and victim (legitimate users). Attacker is responsible for the DDoS attack, it initiates handlers in the network who aim to communicate with agents/zombies and plan a DDoS attack. Handlers are software malware programs that can be installed in a network server or a router and try to invoke compromised agents or zombies in a number of computers. Attackers can use multiple handlers to communicate with agents or zombies who carry out malware actions specified by handlers, without knowing that they participate in a DDoS attack. The main concept of agents/zombies is to waste the bandwidth of network or router processing capacity (flooding attack), sending at the same time useless data to the victim [15]

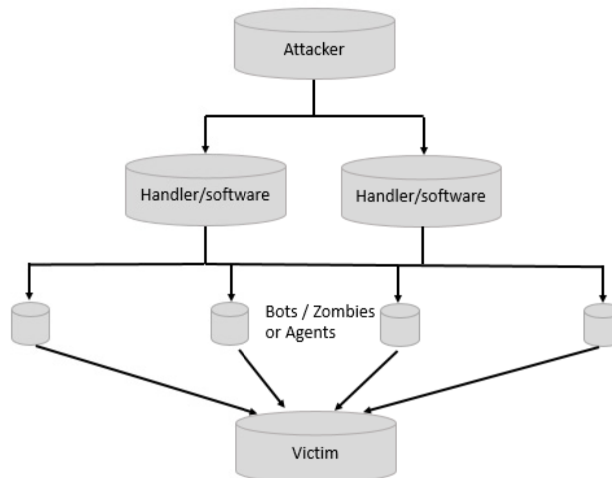

Figure 6: Agent-Handler model

2. Internet Relay Chat (IRC) is similar to Agent-Handler model. Based on IRC protocol, clients and servers exchange messages via network. In this case, handlers are channels used by attackers to control and send rules to agents via legitimate ports. IRC-based model uses a centralized command and control (C&C) server to coordinate a number of agents or zombies. An IRC-based model provides several advantages to the attacker, For example, an attacker can use legitimate ports to make him invisible, sending large amounts of useless data. The IRC protocol can give an advantage to attackers connecting to IRC server, e.g. by providing a list of available agents or zombies. IRC-based tools are Trinity v3 and kaiten [14].

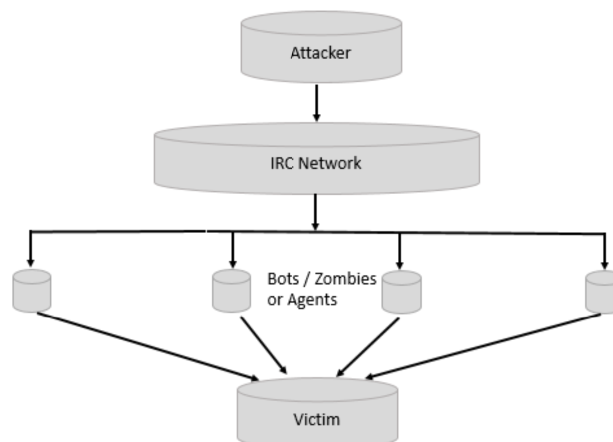

Figure 7: Internet Relay Chat (IRC) - based model

3. Web-based model can use HTTP/HTTPS network protocol to control and send rules to agents or zombies. These agents or zombies can use 80/443 port to send results or statistics to a web site via scripts and encrypted communications. Web-based model provides advantages [16], e.g. an attacker can use HTTP protocol to become invisible, sending useless data via of 80/443 port from a large number of agents or zombies. Today, serveral Web-based tools have been developed, e.g. BlackEnergy, Low-Orbit Ion Cannon and Aldi [14].

2.4 Authentication and Access Control

2.4.1 Authentication

The authentication refers to user verification prior to entry in a system [4]. When a user or process wants to use system resources, e.g. memory, CPU, or network connections, identity must be authenticated first, commonly using user name and password. Related security info, including user ID (uid) and group id), is stored in `etc/passwd` file, where the password is shadowed in root space; the entry in this file is similar to `username:x:1001:1003::/home/userfolder`, where group ids are defined in `/etc/group` file. By default , root user has user ID and group ID 0 (zero). There are also others types of authentication, such as one-time passwords (e.g TAN lists) or certificates (e.g in ssh).

2.4.2 User permissions

User permissions are based on access control lists (ACL) supported in Linux kernel. Each file or directory that exists in a Unix System belongs to a user. When a file or directory is created default permissions for read, write or execute are provided (using `umask`). These permissions define the user that creates the file or directory, the user groups (many users) and other users in system. File permissions in Linux can dynamically change with the commands `chmod` (change file permissons), `chown` (change file owner and group), `chgrp` (change group ownership) and `chattr` (change file attributes on a Linux file system). For instance, as shown in Figure 8, a directory with permission number 750 gives all permissions (7) to user, read and execute only (5) for group and no permission (0) to others.

		Read	write	Execute
All permissons	Owner (user)	4 (r)	2 (w)	1 (x)
Not write	Group	4 (r)	0 (not write)	1 (x)
None permission	Others	0 (not read)	0 (not read)	0 (not read)

Figure 8: User permissions

In the following frame we show an example of `chmod` command. The `-rw-r--r`–is the default permission to users that consists of 10 characters. The first character corresponds to the type of file (`d` means directory, - means simple file), the next three characters are for user permissions, the next three concern group permissions, and the last three correspond to other users. We can see that `chmod 750 example.txt` command can be used to convert the permissions of this file.

```
root@user:/home# vi example.txt
root@user:/home# ls -l example.txt
-rw-r--r-- 1 root root 5 Απρ   6 19:58 example.txt
root@user:/home# chmod 750 example.txt
root@user:/home# ls -l example.txt
-rwxr-x--- 1 root root 5 Απρ   6 19:58 example.txt
```

Notice that many important files for system functionality are in root directory, where only root has access.

2.5 Log Analysis

In Unix systems each action or error reported to log files [4]. These actions are called events and they are important for the operating system to detect and diagnose security issues. Different logging mechanisms are examined below.

- Write to `stdout` or `stderr`: each action in Unix system can create log files that relate to `stdout` (standard output) or `stderr` (standard error messages).
- Syslog: the `syslog` daemon is a standard for log analysis. It allows the system to save its messages in syslog file located in `/var/log/syslog` path in Unix. Event monitoring tools performing log analysis tools use this file for system management, security auditing and other reasons.
- `dmesg/klogd/journalctl`: The `/var/log/kern.log` file (or `systemd journalctl` options in the more recent in Linux kernel 4.0+) log kernel actions not logged in `syslog` file due to separation between user and kernel space. The following dmesg file lists different Linux kernel information.

```
[   37.704257] 00:00:00.001471 main     Process ID: 1391
[   37.704257] 00:00:00.001472 main     Package type: LINUX_64BITS_GENERIC
[   37.709678] 00:00:00.006880 main     5.0.6 r103037 started. Verbose level = 0
[   37.792768]  00:00:00.089891  automount  VBoxServiceAutoMountWorker:  Shared
folder "Documents" was mounted to "/media/sf_Documents"
```

Chapter 3 - NoC Firewall

In this Chapter, we show how to use a NoC Firewall mechanism to help resolve security issues that exist in eHealth platforms, focusing on privacy of healthcare data. Our mechanisms rely on developing a hierarchical Linux kernel module that we can use to setup firewall rules and perform BRAM access by sending/receiving information to/from kernel space using specialized IOCTL commands.

As shown in Figure 9, a NoC firewall module with four sets of registers (one set per each input port) is placed in front of the input FIFOs of a 4x4 Butterfly NoC. The firewall sets are independent and their number is scalable and depends only on the size of the FPGA. The Butterfly NoC itself is configured with smaller 2x2 internal switching nodes using input-output buffering and each of its output ports is attached to a BRAM. Although a debug AMBA AXI4 interface provides direct access to the BRAMs, in this Section we only focus on normal operation, i.e. setup of the firewall rules and normal access requests that travel through the firewall.

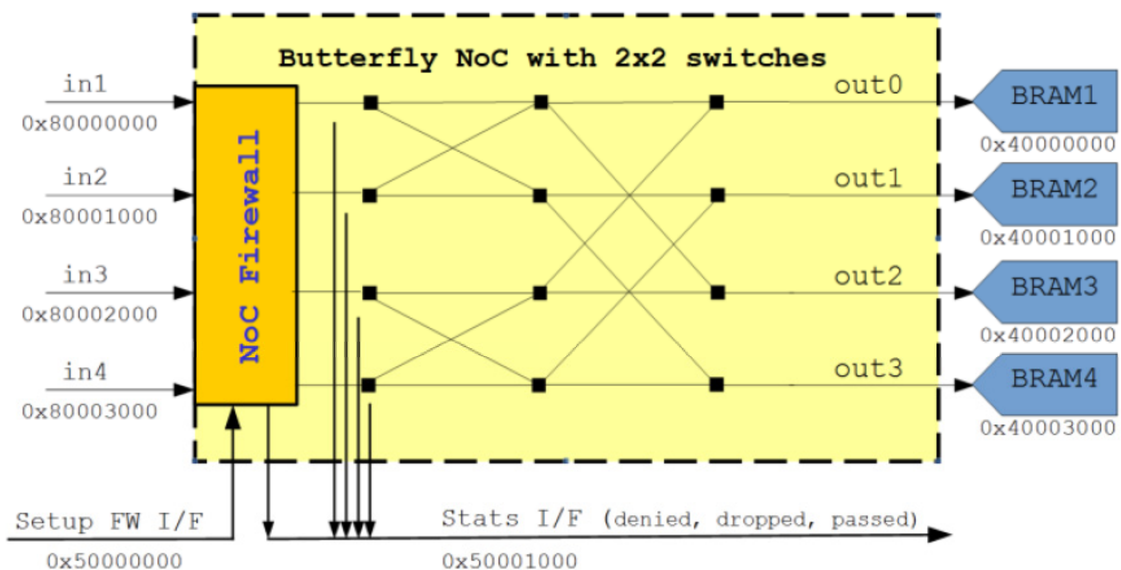

Figure 9: Butterfly NoC with 2x2 switches

3.1 Firewall Setup and Access Request via Firewall

Each set of firewall registers can be configured independently to operate in one of two operating modes, e.g Simple and Extended mode that are discussed below.

A firewall register set configured in the *Simple Mode* is configured to protect data in each of the BRAMs' memory-mapped address space from illegitimate access requests from all input ports. More specifically, the register set protects accesses to the following memory regions:

[0x40000000, 0x40000FFF] for BRAM1,

[0x40001000, 0x40001FFF] for BRAM2,

[0x40002000, 0x40002FFF] for BRAM3, and

[0x40003000, 0x40003FFF] for BRAM4.

independent of how access request packets travel through the NoC, i.e. only the final destination matters,

not the source node or path. In fact, for the Butterfly network topology, the source uniquely determines the path. In Simple mode, deny or allow rules for a protected memory address range are specified for each BRAM using four set of registers, one in each input port. Each set has three registers mapped to the address range [0x50000000, 0x50000FFF] and consists of `L_addr_reg`, `H_addr_reg`, and `Rule_reg` register defined as follows.

- *32-bit low address register* (`Low`) *specifying the start point* of an address range, see Figure 10. This is an *absolute address* in BRAM1, BRAM2, BRAM3, and BRAM4 address region,

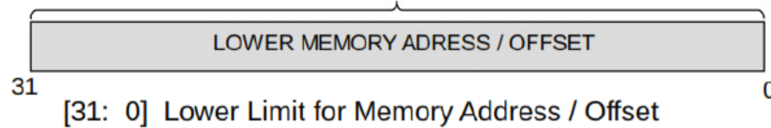

Figure 10: Setup low address range

- *32-bit high address register* (`High`) *specifying the end point* of an address range, see Figure 11. This is an *absolute address* in the corresponding BRAM1, BRAM2, BRAM3, and BRAM4 address region.

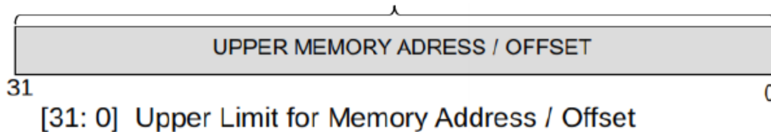

Figure 11: Setup high address range

- *32-bit rule register* (`firewall rules`) controlling protection of BRAM accesses to the corresponding `[Low, High]` address range as shown in Figure 12.

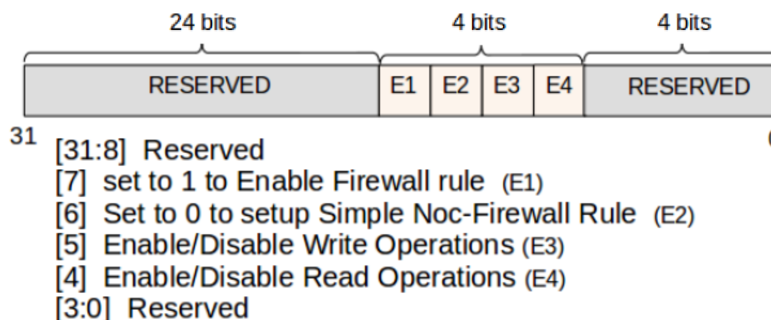

Figure 12: Setup rule address rage in Simple Mode.

Notice that for Simple mode, the 32-bit setup rule register is split to three parts, whereas the first 24 and last 4 bits are reserved. The intermediate 4 bits used in the rule register consists of two parts:

✓ 2-bit output port field specifying enable/disable of the firewall, and the operating mode (Simple or Extended). For example, a `bit7` field value of “1” specifies that the firewall is on, while “0” specifies that the firewall is turned off. Similarly, a `bit6` field value of “0” specifies that Simple firewall rule is enabled.

`bit7` : Value set to 1 to enable Firewall or set to 0 to disable Firewall
`bit6` : Value set to 0 to operate in simple mode (0 for Simple, 1 for Extended)

✓ *2-bit rule register* (`Rule`) controlling protection of BRAM accesses to the corresponding `[Low, High]`

address range. These two bits correspond to four firewall operations: *deny read, deny write, deny read & write,* or *accept all* accesses to the [Low, High] address range.

`bit5` : Value set to 1 to enable rule or set to 0 to disable rule for Write operations
`bit4` : Value set to 1 to enable rule or set to 0 to disable rule for Read operations

Notice that we require that each pair `[Low, High]` corresponds to a given BRAM, while all pairs are assumed non-overlapping. This is asserted for each new pair in the GNU/Linux driver.

In the *Extended Mode*, the firewall operates as a true NoC-based firewall. More specifically, firewall rules in the Extended mode are specified by providing both the input ports of the access request, as well as the BRAM (destination) output port, i.e. `out1`, `out2`, `out3` and `out4` in Figure 15. This is in contrast to Simple mode, whereas the rule for a particular input ports destination port applies to all BRAMs. Thus, in Extended mode, deny or allow rules for a protected memory address range are specified again for each BRAM using four set of registers, one in each input port. Each set has three registers mapped to the range [0x50000000, 0x50000FFF] and consists of `L_addr_reg`, `H_addr_reg`, and `Rule_reg` registers defined as follows.

- *32-bit low address register* (`Low`) *specifying the start point* of an address range, see Figure 13. This is a *relative address* in BRAM1, BRAM2, BRAM3, and BRAM4 address region, i.e. [0x0000, 0x0FFF],

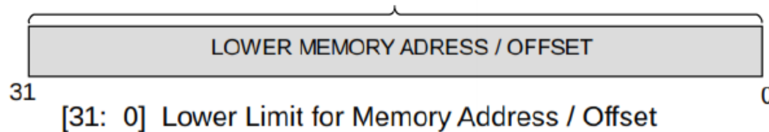

Figure 13: Low address range register (Setup)

- *32-bit high address register* (`High`) *specifying the end point* of an address range, see Figure 14. This is a *relative address* in the corresponding BRAM1, BRAM2, BRAM3, and BRAM4 address region, i.e. [0x0000, 0x0FFF].

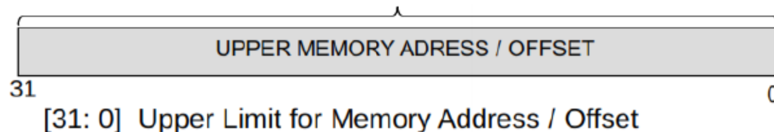

Figure 14: High address range register (setup)

- *32-bit rule register* (`firewall rules`) controlling protection of BRAM accesses to the corresponding `[Low, High]` address range. These two bits correspond to four firewall operations: deny read, deny write, deny read & write, or accept all accesses to the `[Low, High]` address range.

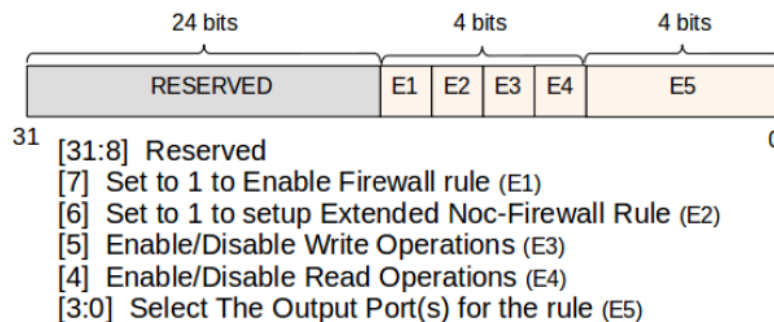

Figure 15: Rule address rage in Extended Mode register (setup)

Notice that for Extended mode, the 32-bit setup rule register is split to three parts, whereas the first 24 bits

are reserved. The last 8 bits used in the rule register consists of three parts:

- ✓ 2-bit output port field specifying enable/disable of the firewall, and the operating mode (Simple or Extended). For example, a `bit7` field value of "1" specifies that the firewall is on, while "0" specifies that the firewall is turned off. Similarly, a `bit7` field value of "0" specifies that Simple mode is enabled.

`bit7` : Value set to 1 to enable Firewall or set to 0 to disable Firewall
`bit6` : Value set to 0 to operate in simple mode (0 for Simple, 1 for Extended)

- ✓ *2-bit rule register* (`Rule`) controlling protection of BRAM accesses to the corresponding `[Low, High]` address range. These two bits correspond to four firewall operations: *deny read, deny write, deny read & write,* or *accept all* accesses to the [Low, High] address range.

`bit5` : Value set to 1 to enable rule or set to 0 to disable rule for Write operations
`bit4` : Value set to 1 to enable rule or set to 0 to disable rule for Read operations

- ✓ 4-bit output port field which specifies the NoC output port (equivalently, the BRAM connected to this output port) for which this firewall rule will apply for example, a field value of "1111" specifies that the rule will apply for accesses from all four output ports (respectively, BRAMs) in Figure 15.

`bit3`: Value set to 1 to enable rule for output port 1 (BRAM 1)
`bit2`: Value set to 1 to enable rule for output port 2 (BRAM 2)
`bit1`: Value set to 1 to enable rule for output port 3 (BRAM 3)
`bit0`: Value set to 1 to enable rule for output port 4 (BRAM 4)

In addition, a read-only interface for monitoring firewall statistics is provided using four set of registers, one in each port at memory address 0x50001000. Three registers (`startingStatisticsTotal`, `startingStatisticsFifo` and `startingStatisticsFw`) are defined as follows:

- 32-bit passed register (PASSED) specifies the total number of packets (read or write) accepted per port; notice that for each of the 4 ports, we have a dedicated register.

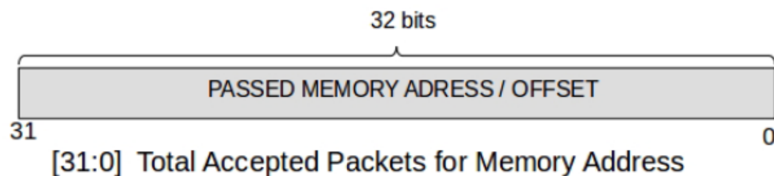

Figure 16: Passed packets register (monitor).

- 32-bit fifo dropped register (FIFO DROPPED) specifies the dropped packets lost due to full input fifos; notice that for each of the 4 ports, we have a dedicated register and that each FIFO can hold up to 5 access request packets.

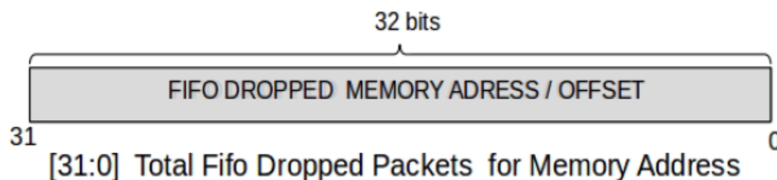

Figure 17: Fifo dropped packets register (monitor).

- 32-bit firewall dropped register (FW DROPPED) specifies the number of denied packets due to firewall rule per each input port; notice that for each of the 4 ports, we have a dedicated register.

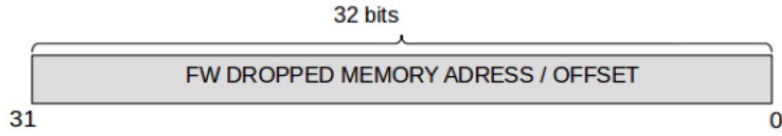

Figure 18: Fw dropped packets register (monitoring statistics).

In our implementation, a 32-bit register is used for packetizing BRAM access (read or write) via the firewall in Simple and Extended mode. The access request to a BRAM is specified by a NoC output port and a NoC input port and BRAM offset is relative to the specified BRAM base address. More specifically, the register is split to five parts, as shown in Figure 19.

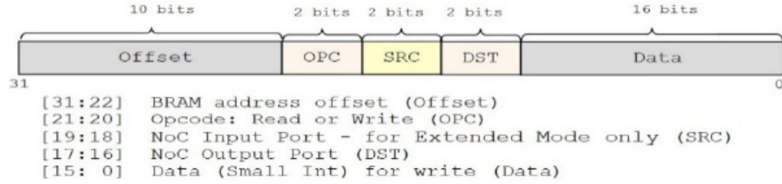

Figure 19: Access request in Simple/Extended Mode; SRC field used only in Extended Mode.

- ✓ 10-bit for address offset. This means that 10 bits can be used to write (16-bit) data to specific address, e.g. 0x00000000 in BRAM1.
- ✓ 2-bit operation code specifies write and read operation. For example, a field value of "0x1" operates the write rule and "0x0" operates the read rule.
- ✓ 2-bit source port specifies the NoC source input port.
- ✓ 2-bit destination port specifies the NoC output port.
- ✓ 16-bit packet payload.

3.2 Linux Driver – Hardware implementation

During synthesis we have followed an incremental process for validating our final hardware module by building and testing separately individual components. This process has helped manage complexity issues and evaluate relative hardware costs (related to the Zedboard FPGA) of the proposed 4x4 Butterfly NoC with firewall functionality on all 4 ports (called NoCFW) with a preliminary working implementation of the 4x4 Butterfly NoC without firewall (called NoC). Results from another intermediate implantation of the 4x4 Butterfly NoC with firewall functionality supporting Extended Mode on only one input port (the first, `in1`) are predictable and for this reason they are omitted.

Both the NoC and NoCFW modules were defined in bit-accurate SystemC and co-simulated/validated in a high-level synthesis tool (Xilinx Vivado HLS). We have used simple Vivado HLS directives to optimize the hardware associated with code structure (data pack for the data structures, horizontal array map for the FIFOs, and unroll for loops). The co-validated IPs were subsequently packaged in Vivado HLS tool and exported to Xilinx Vivado tool for synthesis on Zedboard Z7020 FPGA. In Vivado, all related circuits (AXI interconnects & BRAM cells and controllers) were connected and the register layout was manually adjusted to be symmetric for different ports symmetric in order to simplify driver development. Table 1 presents FPGA resource utilization for NoC and NoCFW and compares the more accurate estimations from Vivado with those from Vivado HLS. Vivado results point out that NoCFW implementation takes ~10-20% of the logic elements on the FPGA and its cost in terms of FPGA LUTs and flip-flop count is marginal.

Table 1: Hardware Costs In NoC & NoCFW Implementations

	Vivado HLS		Vivado	
Type (Available)	NoC	NoCFW	NoC	NoCFW
FF (106400)	10019 (9%)	11164 (10%)	10867 (10%)	12012 (11%)
LUT (53200)	9598 (18%)	9818 (18%)	11029 (21%)	11421 (21%)
LUT as BRAM	1027 (2%)	1027 (2%)	1372 (3%)	1372 (3%)
BRAM (32)	-	-	4 (3%)	4 (3%)
Clocking BUFGCTRL (32)	-	-	1 (3%)	1 (3%)

In addition, Table 2, further compares the cost of NoCFW hardware module to AMBA AXI4, and a larger instance of STMicroelectronics STNoC network-on-chip on the same FPGA fabric.

The delay in the firewall module is very small. Estimation from Vivado tool gives a data path delay of 8.32 ns (excluding AXI) for the NoC and a corresponding delay of just 9.12 ns for NoCFW. Moreover, by examining the critical path, we discover that almost 80% of the delay is due to the NoC, the remaining due to interface logic for packetization (see Figure 15) and firewall access. Finally, notice that including AXI, the total delay rises to 12.10 ns for NoC or 12.97 ns for NoCFW, respectively.

Table 2: Compare - Hardware Costs In NoC & NoCFW Implementations

Component	LUTs	Registers	BRAM
NoCFW	11421	12012	4
AXI Bridge	723	971	
STNoC (12-nodes,4x4 routers)	24939	17983	

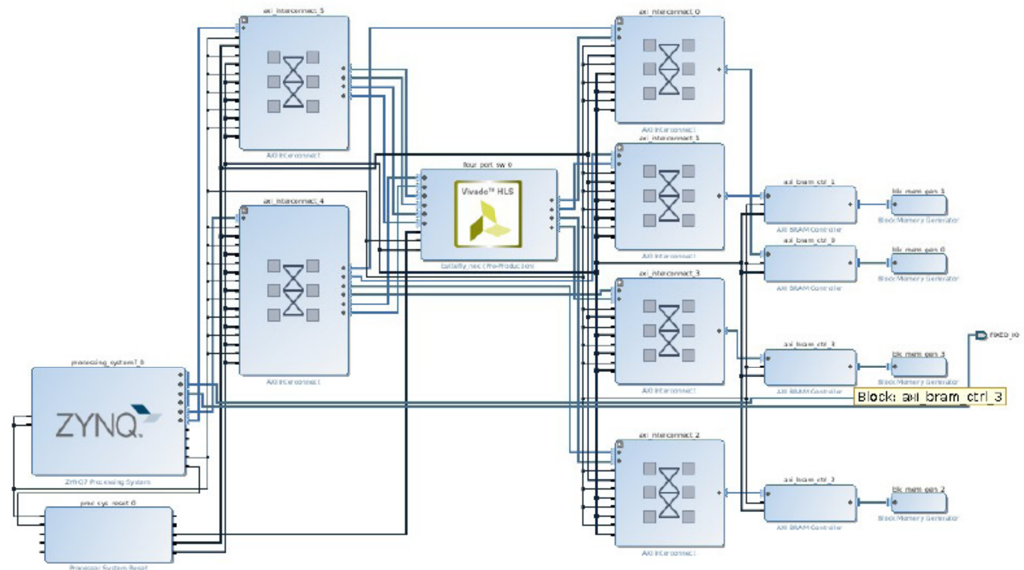

Figure 20: 4-Port Firewall Multicast Router – Synthesis on Zedboard

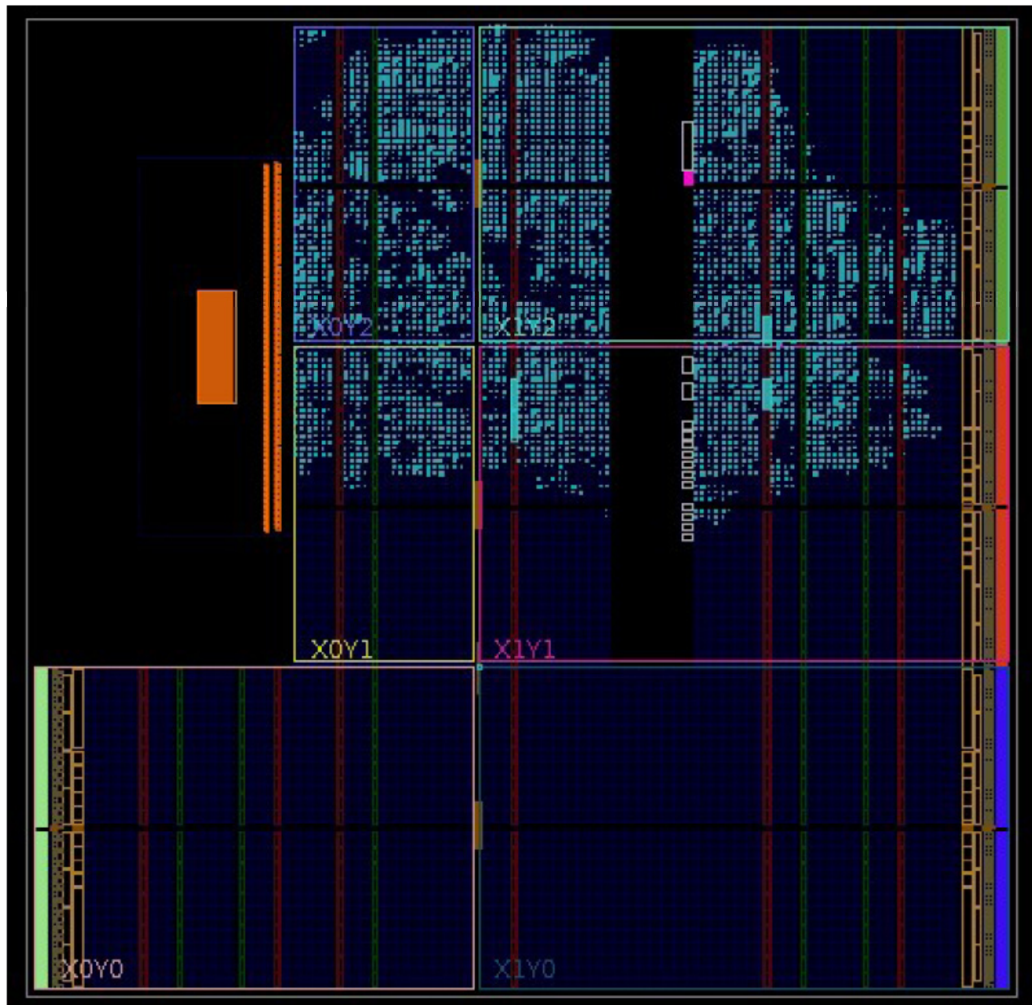

Figure 21: The NoC Firewall: a) circuit and b) utilization of the FPGA

Figure 20 and 21 show the NoCFW circuit created in Vivado. The circuit includes BRAMs connected to each output port via AXI). Moreover, notice that all input ports are connected by AXI with the Core Xilinx module (ARM Cortex-A9 processors). In addition, the bottom part of Figure 21, indicates the low utilization of the Zedboard FPGA by the NoCFW circuit (~20% used).

3.3 Hierarchical Design of NoC Firewall Driver

The NoC Firewall driver is organized hierarchically into 3 layers in order to enhance modularity and reusability. As shown in Figure 22, the proposed three layers includes a low-level I/O memory interface, as well as mid-level kernel functions and user-level functions, with callbacks defined only between adjacent protocol layers.

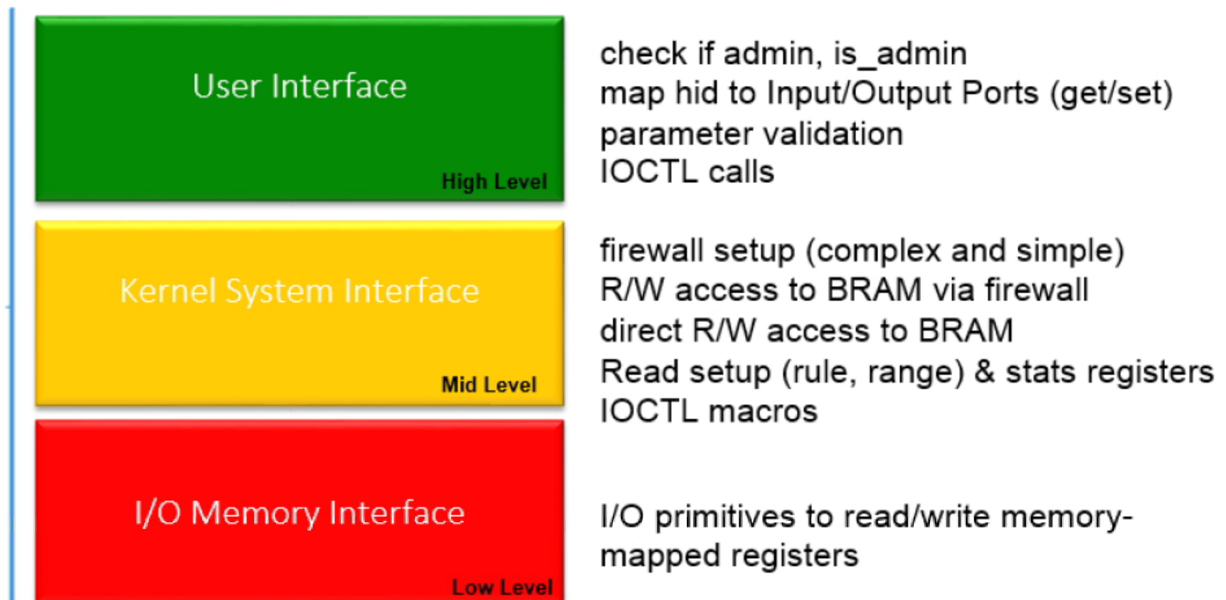

Figure 22: Hierarchical Linux Driver for our Firewall

3.3.1 Low Level Driver API

The Low-Level Driver API (LLD) defines I/O memory methods to

- write in kernel space using `iowrite32()` method and
- read from kernel space using `ioread32()` method.

These methods are called by Mid-Level Driver API, cf. Chapter 3.1.4. For example, LLD is used to perform firewall setup by writing using `iowrite32` the memory-mapped setup registers, and is also used to read firewall setup information from memory setup registers using `ioread32` method. LLD also calls `ioread32` to access to memory-mapped statistics counters. Finally, LLD uses `iowrite32` and `ioread32` methods to support both direct access access to BRAM and access via the firewall.

3.3.2 Mid-Level Driver API

The Mid-Level Driver API (MLD) defines firewall setup in specific output and input ports ranges, supports read/write access to input and output ports via NoC Firewall, provides direct access to output port (Bram), and enables different types of statistics for total passed/dropped packets via IOCTL macros.

We have implemented the MLD in two files. The first one (`ioctl_fw.h`) contains the following data structures for setting up the firewall, reading statistics, and accessing BRAM either via the NoCFW or

directly via an AXI bus (for debug purposes) and uses IOCTL calls to send commands to user-space.

```
struct  fw_ds{ //setup FW
      unsigned int op_code;
      unsigned int inport;
      unsigned int outport;
      unsigned int Low;
      unsigned int High;
      unsigned int Rule;
};
struct stat_ds { // statistics info
      unsigned int inport;
      unsigned int Total passed;
      unsigned int Fifo_dropped;
      unsigned int Fw_dropped;
};
struct  access_ds {  // access via NoCFW
      unsigned int op_code;
      unsigned int inport;
      unsigned int outport;
      unsigned int addr;
      unsigned int *data;
};
struct  dir_access_ds { // direct access
      unsigned int op_code;
      unsigned int bram_no;
      unsigned int addr;
      unsigned int *data;
};
```

Moreover, MLD defines the IOCTL macros used in High-Level Driver (HLD).

```
#define MYIOC_TYPE 'k'
#define MYIOC_WRITEONE    _IO(MYIOC_TYPE, 1)    //command one (WRITEONE)
#define MYIOC_WRITETWO    _IO(MYIOC_TYPE, 2)    //command two (WRITETWO)
#define MYIOC_WRITETHREE  _IO(MYIOC_TYPE, 3)    //command three (WRITETWO)

#define MYIOC_READONE     _IO(MYIOC_TYPE, 1)    //command one (READONE)
#define MYIOC_READTWO     _IO(MYIOC_TYPE, 2)    //command two (READTWO)
#define MYIOC_READTHREE   _IO(MYIOC_TYPE, 3)    //command three (READTHREE)
#define MYIOC_READFOUR    _IO(MYIOC_TYPE, 4)    //command four (READFOUR)
```

Thus, after generating a unique code for _IOR and _IOW, we can compose a unique IOCTL name for each firewall action.

```
// compose a unique ioctl number to setup firewall in specific port
#define IOCTL_SETUP_FW_PORT _IOW(MYIOC_TYPE, MYIOC_WRITEONE, struct fw_ds)
// compose a unique ioctl number to check the setup of firewall in specific port
#define IOCTL_CHECK_FW_PORT _IOR(MYIOC_TYPE, MYIOC_READTWO, struct fw_ds)
// compose a unique ioctl number to read the setup of firewall in specific port
#define IOCTL_READ_FW_PORT _IOR(MYIOC_TYPE, MYIOC_READONE, struct fw_ds)
```

```
// compose a unique ioctl number to read firewall statistics on specific port
#define IOCTL_READ_STATS _IOR(MYIOC_TYPE, MYIOC_READTHREE, struct stat_ds)
// compose a unique ioctl number to read firewall total statistics on all ports
#define IOCTL_READ_STATS_ALL _IOR(MYIOC_TYPE, MYIOC_READFOUR, struct stat_ds)
// compose a unique ioctl number to access the bram via the switch
#define IOCTL_ACCESS_BRAM _IOW(MYIOC_TYPE, MYIOC_WRITETWO, struct access_ds)
// compose a unique ioctl number to direclty access bram
#define  IOCTL_DIRECT_ACCESS_BRAM  _IOW(MYIOC_TYPE,  MYIOC_WRITETHREE,  struct
direct_access_ds).
```

The second file in MLD (`kernel_mode.c`) provides fundamental functions for firewall operation:

```
static void setupFW(struct swfw_ds *);
static void readFWRegs(struct swfw_ds *);
static int checkFWRegs(struct swfw_ds *);
static void readStatsPort(struct stat_ds  *);
static void readStats(struct stat_ds *);
static int accessBram(struct dir_access_ds *)
static int accessBramFW(struct access_ds *);
```

More specifically, MLD performs firewall setup (by calling `setupFW`) for both Simple and Extended mode. This call involves LLD `iowrite32` to the memory-mapped setup registers. Due to symmetry in the memory layout (defined in Vivado) for all input ports, calls are similar for setting Low, High range, or Rule registers.

```
iowrite32(L_addr_reg, noc_setup_registers + startingLowerReg + ((inport-1)*2) );
iowrite32(H_addr_reg, noc_setup_registers + startingHighReg + ((inport-1)*2) );
iowrite32(Rule_reg, noc_setup_registers      + startingRuleReg + ((inport-1)*2)
);
```

MLD can *read setup registers* by calling `readFWRegs` (or check their values via `checkFWRegs function`) which invokes LLD `ioread32` to the memory-mapped setup registers.

```
//retrieve data from kernel space and eventually to user space
fw_ds->L_addr_reg = ioread32(noc_setup_registers + startingLowerReg
                                                 + (inport-1)*2 );
fw_ds->H_addr_reg = ioread32(noc_setup_registers + startingHighReg
                                                 + (inport-1)*2 );
fw_ds->Rule_reg =   ioread32(noc_setup_registers + startingRuleReg
                                                 + (inport-1)*2 );
```

Next, `checkFWRegs` method can validate firewall rules by calling `readFWRegs`

```
fw_ds _temp.input_port = fw_ds->input_port;
fw_ds _temp.L_addr_reg = fw_ds ->L_addr_reg;
fw_ds _temp.H_addr_reg = fw_ds ->H_addr_reg;
fw_ds _temp.Rule_reg = fw_ds ->Rule_reg;
// check initial data with data from readFWRegs method
readFWRegs(fw_ds);
if ((fw_ds _temp.L_addr_reg== fw_ds ->L_addr_reg)
&&  (fw_ds _temp.H_addr_reg== fw_ds ->H_addr_reg)
&&  (fw_ds _temp.Rule_reg== fw_ds ->Rule_reg) ) {
```

```
 flag = 1; // valid firewall rules
}else {
 flag = -1; // invalid firewall rules
return flag;
}
```

The address layout of the firewall setup registers refers to four register triplets (*startingLowerReg*, *startingHighReg*, *startingRuleReg*) and corresponds to:

0x50000014 : Setup Firewall Lower Address Range Register for port 0 (`startingLowerReg`) *Offset:* 13 (in decimal)
`0x5000001C` : Setup Firewall Lower Address Range Register for port 1 (`startingLowerReg`) Offset: 15 (in decimal)
`0x50000024` : Setup Firewall Lower Address Range Register for port 2 (`startingLowerReg`) Offset: 17 (in decimal)
`0x5000002C` : Setup Firewall Lower Address Range Register for port 3 (`startingLowerReg`) Offset: 19 (in decimal)
`0x50000034` : Setup Firewall Upper Address Range Register for port 0 (`startingHighReg`) Offset: (5 in decimal)
`0x5000003C` : Setup Firewall Upper Address Range Register for port 1 (`startingHighReg`) Offset: (7 in decimal)
`0x50000044` : Setup Firewall Upper Address Range Register for port 2 (`startingHighReg`) Offset: (9 in decimal)
`0x5000004C` : Setup Firewall Upper Address Range Register for port 3 (`startingHighReg`) Offset: (11 in decimal)
`0x50000054` : Setup Firewall Rule Register for port 0 (`RuleReg`) Offset: (21 in decimal)
`0x5000005C` : Setup Firewall Rule Register for port 1 (`RuleReg`) Offset: (23 in decimal)
`0x50000064` : Setup Firewall Rule Register for port 2 (`RuleReg`) Offset: (25 in decimal)
`0x5000006C` : Setup Firewall Rule Register for port 3 (`RuleReg`) Offset: (27 in decimal)

The MLD API also monitors re*ad access to memory-mapped via statistics counters* by calling LLD *ioread32* function with the proper base address and offset. For example, for bram one, the total number of packets passed is accessed in **readStatsPort** via:

```
my_stat_ds->Total_passed_Bram1=ioread32(iobram1+ startTotal) - init_stat_total1;
my_stat_ds-> Fifo_dropped_Bram1=ioread32(iobram1 + startFifo) - init_stat_fifo1;
my_stat_ds-> Fw_dropped_Bram1=ioread32(iobram1 + startFw) - init_stat_fw1;
```

Notice that subtraction of `init_total_stat`, `init_total_fifo` and `init_stat_fw` above refers to software reset during module re-installation; initial values have been read in this variable in `init_module.h`.

Moreover, cumulative statistics for all ports are defined by the *readStats* function which sums statistics counters for all output ports (Bram1, Bram2, Bram3, Bram 4).

```
my_stat_ds->Total_passed = totalPassedBram1 + totalPassedPort2 +
                           totalPassedPort3 + totalPassedPort4;
my_stat_ds>Fifo_dropped = fifoDroppedBram1 + fifoDroppedBram2 +
                          fifoDroppedBram3 + fifoDroppedBram4;
my_stat_data_ptr>Fw_dropped = fwDroppedBram1 + fwDroppedBram2 +
                              fwDroppedBram3 + fwDroppedBram4;
```

The address layout of the firewall setup registers used in the `ioread32` command is as follows.

Read total packets passed from each port
`0x80000024` : Read from Port 1 (`startTotal`) (Offset: 9 in decimal)
`0x80001024` : Read from Port 2 (`startTotal`) (Offset: 9 in decimal)
`0x80002024` : Read from Port 3 (`startTotal`) (Offset: 9 in decimal)
`0x80003024` : Read from Port 4 (`startTotal`) (Offset: 9 in decimal)

Read dropped packets (due to full Fifo) from each port
`0x8000002C` : Read from Port 1 (`startFifo`) (Offset: 11 in decimal)
`0x8000102C` : Read from Port 2 (`startFifo`) (Offset: 11 in decimal)
`0x8000202C` : Read from Port 3 (`startFifo`) (Offset: 11 in decimal)
`0x8000302C` : Read from Port 4 (`startFifo`) (Offset: 11 in decimal)

Read dropped packets (due firewall rule) from each port
`0x80000034` : Read from Port 1 (`startFw`) (Offset: 13 in decimal)
`0x80001034` : Read from Port 2 (`startFw`) (Offset: 13in decimal)
`0x80002034` : Read from Port 3 (`startFw`) (Offset: 13 in decimal)
`0x80003034` : Read from Port 4 (`startFw`) (Offset: 13 in decimal)

Finally, MLD supports direct access, in addition to access via the firewall. Direct access to BRAM is carried out via LLD normal calls to functions *`ioread/iowrite`*, such as *`iowrite32(data, iobram1_direct`*

```
+ addr_reg);
```

Access via the firewall is much more complicated. It involves a) packetization of the write access request (see Figure 12) at the NoC interface, as well as transfer of the NoC packet for execution at the BRAM b) both packetization of the read access request (see Figure 12) and response depacketization for read access. More specifically, for NoC write operation, packetization to 32-bit write packet register is accomplished via the following `iowrite` (write opcode is 0).

```
if (op_code == 0x0){ // write option
      write_value = addr_reg << 24; // offset
      write_ value += inport << 18; // src port
      write_value += outport << 16; // dest port
      write_value += data; // 16-bit payload
      switch (inport) { //correction in code
       case 1: // access via input port 1 to write packet register
         iowrite32(wvalue , (iobram1 + 5));
         ...
}
```

For NoC read operation, packetization to 32-bit read packet register (`r_reg`) is similar, except for opcode.

```
...
else { //read option
      rvalue = addr_reg << 24; // offset
      rvalue += 0x1 << 20; // read opcode
      rvalue += inport << 18; // src port
      rvalue += outport << 16; // dest port
      rvalue += 0x0; // no payload for read
      switch (inport) {
      case 1:
          iowrite32(rvalue, (iobram1 + 5));
         break;
       case 2:
         iowrite32(write_value, (iobram2 + 5));
         break;
      case 3:
         iowrite32(write_value, (iobram3 + 5));
         break;
      case 4:
         iowrite32(write_value, (iobram4 + 5));
         break;
      }
```

Finally, depacketization from each 32-bit read response register (`r_reg`) located at each input port is accomplished via the following code.

```
switch (inport) {
       case 1:
          data = ioread32(iobram1 + 7);
          break;
      case 2:
```

```
            data = ioread32(iobram2 + 7);
            break;
        case 3:
              data = ioread32(iobram3 + 7);
            break;
        case 4:
            data = ioread32(iobram4 + 7);
            break;
}
// keep 16 least significant bits
data =(data << 16) | (data >> 16) //chop last 16 bits for 16-bit value
final_data = (print_data) >> 16;
```

The values + 5 in write operation (and + 7 in read operation) above correspond to writing the packet in the Write Packet Register (resp. reading response from the Read Response Register). The layout of these registers is as follows.

Write Packet Register to switch Port
0x80000014 : Write Packet to Port 1 (Offset: 5 in decimal)
0x80001014 : Write Packet to Port 2 (Offset: 5 in decimal)
0x80002014 : Write Packet to Port 3 (Offset: 5 in decimal)
0x80003014 : Write Packet to Port 4 (Offset: 5 in decimal)

Read Responce Register (after read request) from switch Port
0x8000001C : Read from Port 1 (Offset: 7 in decimal)
0x8000101C : Read from Port 2 (Offset: 7 in decimal)
0x8000201C : Read from Port 3 (Offset: 7 in decimal)
0x8000301C : Read from Port 4 (Offset: 7 in decimal)

In addition, MLD provides the `static long my_unlocked_ioctl(struct file *, unsigned int, unsigned long)` function that defines commands that can be used from user space. These commands use traditional `copy_from_user` and `copy_to_user` to transfer data structures from user to kernel space and vice-versa. We list below ioctl macros from `ioctl_fw.h` file, as referred above.

```
case IOCTL_SETUP_FW_PORT:
 rc = copy_from_user(&my_fw_ds, ioargp, size);
 setupFW(&my_fw_ds);
```

```
 return rc;
case IOCTL_CHECK_FW_PORT:
rc = copy_from_user(&my_fw_ds, ioargp, size);
rc = checkFWRegs(&my_fw_ds);
rc = copy_to_user(ioargp, &my_fw_ds, size);
 return rc; // return code 1 -> check ok
case IOCTL_DIRECT_ACCESS_BRAM:
rc1 = copy_from_user(&my_direct_access_ds, ioargp, size);
accessBram(&my_direct_access_ds);
 rc2 = copy_to_user(ioargp, &my_direct_access_ds, size);
return rc2; //return ((rc1 == 0) && (rc2 ==0));
case IOCTL_ACCESS_BRAM:
rc1 = copy_from_user(&my_access_ds, ioargp, size);
accessBramFW(&my_access_ds);
rc2 = copy_to_user(ioargp, &my_access_ds, size);
return rc2;
case IOCTL_READ_FW_PORT:
rc = copy_from_user(&my_fw_ds, ioargp, size);
readFWRegs(&my_fw_ds);
rc = copy_to_user(ioargp, &my_fw_ds, size);
return rc;
case IOCTL_READ_STATS:
rc = copy_from_user(&my_stat_ds, ioargp, size);
readStatsPerPort(&my_stat_ds);
rc = copy_to_user(ioargp, &my_stat_ds, size);
return rc;
case IOCTL_READ_STATS_ALL:
rc = copy_from_user(&my_stat_ds, ioargp, size);
readStatsPerAllPorts(&my_stat_ds);
rc = copy_to_user(ioargp, &my_stat_ds, size);
return rc;
```

A typical IOCTL call requires passing the appropriate data-structure. For example, in the simple setup firewall function the `user_mode.h` file, we use:

```
rc = ioctl(fd, IOCTL_READ_SFW_PORT, &my_fw_ds)
```

Finally, notice that for security reasons access to all LLD and MLD functions is protected. Most can be called only from privileged users, i.e. system administrators as described next, except from accessBramFW (and some other get functions) which can be called from all system users. In all other cases, kernel panic is caused and no stack information is made available. As shown in the code snippet below, the check is made on the group id that the user belongs to.

```
gid = current_gid().val;
if ((gid != root_pid) &&  (op_code == 0x0)) { // only root group can perform write via firewall
      pr_info("accessBramFW: NON-ROOT TRIED TO PERFORM write access via FW \n");
      BUG_ON(gid != root_pid);
}
```

Locking is provided to avoid consistency issues with simultaneous accesses from different user-space applications (e.g. using pthreads) or kernel-space programs (e.g. using kthreads). This avoids the possibility of buffer overflows, cf. note at end of Chapter 4.

3.3.3 High-Level Driver – API

This API completes the driver hierarchy by providing a configurable way to map a particular use-case or programming scenario to mid-level firewall functionality. The High-Level Driver interface of the hierarchical Linux driver of the NoC Firewall is split into three files, whereas the first file is more generic, while the last two files specifically support data privacy and anonymity in a healthcare scenario; the scenario is described in full in Section 4.

- The first one implemented in `user-mode.h` is generic and invokes kernel space primitives and exchanges correct data with kernel space using IOCTL commands.
- In addition, in our healthcare scenario, a second file implemented in `systemadmin_privacy.c` is specific to the use-case. It calls methods from `user_mode.h` file to setup protection for three different groups of doctors (`fwgroup`, `fwgroup1` and `fwgroup2`).
- The third file consists of three doctors implemented as `clinic`, `clinic1`, and `clinic2` which refer to different clinics (i.e. groups of doctors, or essentially hospital `departments`). We should mention that for testing our scenario, the different doctors in the system (`clinic`, `clinic1` and `clinic2`) belonged to different groups as shown in Table 3.

Table 3: Groups and users in the healthcare scenario

Group	Users (Doctors)
`root`	`system administrator`
`fwgroup`	`clinic`
`fwgroup1`	`clinic1`
`fwgroup2`	`clinic2`

3.3.3.1 Creating users

In Linux system from `/etc/passwd` or `groups` command, it is possible to view you the users and groups that exist in the system. Specifically, each user or group created in the system is added to `/etc/passwd`, or respectively `/etc/groups` file.

For adding a user in Linux, we have used the `groupadd username`. For giving a password we have used the `passwd username` command. For example, `groupadd clinic1` and `passwd clinic1`.

For adding a user to a group we have used `useradd -g groupname -d userpath -m userdirectory`. For example, `useradd -gfwgroup1 -d /home/clinic1 -m clinic1.` All users and groups are created in the same manner by system administrator (root).

3.3.3.2 User_mode File

The HDL API (`user_mode`) supports two functions (`setGidPerInport` and `setGidPerOutport`) to map NoC input and output ports (ports 1, 2, and 3, since port 4 is used by the administrator) to group ids that users belong to and another two functions for recovering these ports based on the group id (`getInportGidPerGid`, and `getOutporGidPerGid`). Set functions can only be called by the system administrator who can write this information to BRAM4 via input and output port 4, with the proper firewall rules to limit BRAM access for specific user groups to given input and output ports. For example, in our healthcare scenario, each of the three user groups can access one BRAM in the range of 1 to 3 via the corresponding input/output ports 1 to 3, while system administrator group manages BRAM4 from input/output port 4. Get functions can be used to recover the ports to access the corresponding BRAMs during normal operation; these functions are available to users. An example of these cases, will be provided in our healthcare scenario. More specifically, these functions are as follows.

```
int is_admin()
void setGidPerInport(gid_t gid, inport)
void setGidPerOutport(gid_t gid, outport)
int getInportPerGid(gid_t gid)
int getOutportPerGid(gid_t gid)
```

The method `is_admin` is also related. It checks if current user is system administrator by using the `grouplist` method. It is called from other functions to validate access privileges of the current user. A code snippet is shown below.

```
// get systemadmin groupid
gid_systemadmin = groupIdFromName("root");
grouplist(list_of_groups, &ngroups); //call function grouplist to take in which
groups belong each current user
for (i=0; i<ngroups; i++){
            if (list_of_groups[i] == gid_systemadmin) {
                  systemadmin_flag = 1;
                  break;
            }
      }
```

Notice that

- `setGidPerInport` function uses `accessBramFW` to write (opcode 0x0) group names (equivalently clinics, and indirectly doctors) via the provided NoC input ports . More specifically, it calls:
  ```
  rc = accessBramFW(0x0, ADMIN_INPUT_PORT, ADMIN_OUTPUT_PORT,
                    0x00000000 + ((inport-1)*0x4), &data);
  ```
- `setGidPerNoCOutport` function uses `accessBramFW` to write (opcode *0x1*) group names (equivalently clinics, and indirectly doctors) via the provided NoC output ports. More specifically, it calls:
  ```
  rc = accessBramFW (0x0, ADMIN_INPUT_PORT, ADMIN_OUTPUT_PORT,
                     0x0000000C +((outport-1) *0x4), &data);
  ```

These functions are called only from administrator to set via input port 4 (ADMIN_INPUT_PORT) and output port 4 (ADMIN_OUTPUT_PORT) the specific input and output ports (1, 2 and 3) for users (e.g. `clinic`, `clinic1`, `clinic2`) to access their dedicated BRAM (1, 2 and 3). Both functions call `is_admin`

to validate the current user identity. The following is an example for input port.

```
// only if user is root, save matching pair in table (inport, group-id)
if(is_admin()){
      // set group id to input ports.
      admin_group = groupIdFromName("root"); // get current groupid
      gid = groupIdFromName(group_name); //get group id from group name
      if((int)gid < 0){
            printf("setGidPerNoCInport: Invalid group.\n");
            exit(1);
      }
      data = (unsigned int) gid;
      …
```

The *getInportGidPerInport* function uses `accessBramFW` to read (opcode *0x1*) group names (equivalently clinics, and indirectly doctors) that have been previously configured. More specifically, it calls:

```
rc = accessBramFW(0x1, inport, outport, 0x00000000 + (i*0x4), &data);
```

The `getOutportGidPerInport` function uses *accessBramFW* to read (opcode *0x1*) group names (equivalently clinics, and indirectly doctors) that have been previously configured. More specifically, it calls:

```
rc = accessBramFW(0x0, ADMIN_INPUT_PORT, ADMIN_OUTPUT_PORT, 0x0000000C
                                                  + ((outport-1)*0x4), &data);
```

These functions can be called from administrator via ADMIN_INPUT_PORT 4 and ADMIN_OUTPUT_PORT 4 and from simple users (e.g. `clinic`, `clinic1`, `clinic2`) to find the input and output port that have been setup from administrator for access to their dedicated BRAM.

In addition, HLD supports specializations or extensions of the above high-level functions. These include separate routines `setupSFW` and `readSFW` (for Simple mode). and `setupCFW/readCFW/checkCFW` (for Extended mode), as well as specializations of statistic functions `readStats/readStatsPort` that focus on specific arguments, e.g. `readStatsFW`, `readStatsFifo` and `readStatsTotal` or `readStatsPortFW`, `readStatsPortFifo` and `readStatsTotal`. The complete HLD API for these functions is as follows.

```
void setupSFW(unsigned int inport, unsigned long L_addr_reg, unsigned long
H_addr_reg, unsigned int rule, unsigned int write_ops, unsigned int read_ops)
void readSFW(gid_t group_id);
void setupCFW(unsigned int inport, unsigned int L_addr_reg, unsigned int
H_addr_reg, unsigned int rule, unsigned int write_ops, unsigned int read_ops,
unsigned int outport);
void readCFW(gid_t group_id);
int accessBram(unsigned int op_code, unsigned int bram_no,
      unsigned int addr_reg, unsigned int *data);
int accessBramFW(unsigned int op_code, unsigned int inport, unsigned int
outport, unsigned int addr_reg, unsigned int *data);
unsigned int readStatsTotalPassedPerPort(unsigned int inport);
unsigned int readStatsFifoDroppedPerPort(unsigned int inport);
unsigned int readStatsFwDroppedPerPort(unsigned int inport);
void readsStatsTotalPerPort(unsigned int port); // passed packets
void readStatsFifoPerPort(unsigned int port);   // dropped packets (entry fifo)
```

```
void readStatsFwPerPort(unsigned int port);     // dropped packets (firewall)
unsigned int readStatsTotal(); // total passed packets
unsigned int readStatsFifo(); // dropped packets from FIFO
unsigned int readStatsFw();   // denied packets from firewall
```

The `setupSFW` function configures the firewall in Simple mode. This is accomplished by seting the input port, low/high range, and rule for read/write operations. (Notice that `rule` is set to 1 to enable the firewall rule, `write_ops` and `read_ops` are set to 1 for deny write and deny read respectively.

```
void setupSFW(unsigned int inport, unsigned long L_addr_reg, unsigned long H_addr_reg, unsigned int rule, unsigned int write_ops, unsigned int read_ops);
```

The `setupSFW` kernel method places restrictions on the input ports (1-4), rule (0 or 1), write/read field (0 or 1) and low/high range of memory to be protected: *0x40000000* to *0x40003FFF* depending on input port; for example for input port 1 it is *0x40000000* to *0x40000FFF*. For example,

```
// checking parameters
        if (inport <= 0 || inport > 4){
                printf("setupSFW: Invalid input port:%d (1 to 4) \n", inport);
                exit(1);
        }
        if (rule < 0 || rule > 1){
                printf("setupSFW: Invalid rule:%d (0 or 1)\n", rule);
                exit(1);
        }
        if (write_ops < 0 || write_ops > 1){
                printf("setupSFW: Invalid deny write:%d (0 or 1)\n", write_ops);
                exit(1);
        }
        if (read_ops < 0 || read_ops > 1){
                printf("setupSFW: Invalid deny read:%d (0 or 1)\n", read_ops);
                exit(1);
        }
      ok_bram1 = ((L_addr_reg >= 0x40000000 || L_addr_reg < 0x40001000) || 
(H_addr_reg >= 0x40000000 || H_addr_reg < 0x40001000));
      if (!ok_bram1) {
            ok_bram2 = ((L_addr_reg >= 0x40001000 || L_addr_reg < 0x40002000) || 
(H_addr_reg >= 0x40001000 || H_addr_reg < 0x40002000));
            if (!ok_bram2) {
                  ok_bram3 = ((L_addr_reg >= 0x40002000 || L_addr_reg < 
0x40003000) || (H_addr_reg >= 0x40002000 || H_addr_reg < 0x40003000));
                  if (!ok_bram3) {
                        ok_bram4 = ((L_addr_reg >= 0x40003000 || L_addr_reg < 
0x40004000) || (H_addr_reg >= 0x40003000 || H_addr_reg < 0x40004000));
                        if (ok_bram4) {
                        }
                  } else
                        printf("setupSFW: ok BRAM3 \n");
            } else
```

```
                printf("setupSFW: ok BRAM2 \n");
        } else {
                printf("setupSFW: ok BRAM1 \n");
        }
        if (!ok_bram1 && !ok_bram2 && !ok_bram3 && !ok_bram4) {
                printf("setupSFW: ERROR in offsets above \n");
                exit(1);
        }
```

After parameter range checking, data must be packetized into the kernel-level data structures (described in MLD) so that IOCTL calls can be invoked to allow user-space to communicate with the corresponding generic MLD `setupFW` function as shown below.

At first, the rule setup register packetizes the info in Simple mode as explained in Section 3.1.

```
Rule_reg = rule << 7; // enable(1) or disable(0) rule
Rule_reg += 0 << 6;   // set to 0 to operate in compatibility mode
Rule_reg += write_ops << 5; //enable/disable rule for write operations
Rule_reg += read_ops << 4; //enable/disable rule for read operations
Rule_reg += 0x0; //unused
```

Then, firewall setup data structures are populated as follows.

```
// fill setup data structure
my_fw_ds.input_port = inport;
my_fw_ds.L_addr_reg = L_addr_reg;
my_fw_ds.H_addr_reg = H_addr_reg;
my_fw_ds.Rule_reg = Rule_reg;
// ioctl call for simple fw setup (use iowrite32)
rc = ioctl(fd, IOCTL_SETUP_FW_PORT, &my_fw_ds);
if(rc < 0) {
        printf("setupSFW: ioctl failed - rc:%d meaning: %s \n", rc,
strerror(errno));
        exit(1);
} else {
#ifdef FW_USER_LOGS
                printf("setupSFW: ioctl passed - rc:%d \n", rc);
#endif
}
```

The `checkFWRegs` function uses IOCTL calls to validate if specific firewall configuration values have been written to kernel space. The initial data is packetized into kernel-level data structures for validation

```
// providing empty data (apart from inport) to read
          check_my_fw_ds.input_port = inport;
          check_my_fw_ds.L_addr_reg = 65536; // maximum is 65535
          check_my_fw_ds.H_addr_reg = 65536; // maximum is 65535
          check_my_fw_ds.Rule_reg = 0; // 0 is never used
        // new ioctl call for validation of setupSFW setup data
          rc = ioctl(fd, IOCTL_CHECK_FW_PORT, &check_my_fw_ds);
        if(rc < 0){
                ...
```

The `readSFW` method provides a way to read the firewall setup registers in Simple mode. Access to this function is based on group id, i.e. it is allowed only for system administrator. IOCTL calls are invoked to allow user space to communicate with the corresponding MLD `readFW` function.

```
//check if the current user id  belongs to pecific groups
if (gid != groupIdFromName("root")){
      printf("readSFW: You are not authorized as system admin.\n");
      exit(1);
}
…
// ioctl call to read simple fw setup data
        rc = ioctl(fd, IOCTL_READ_FW_PORT, &my_fw_ds);
```

The `setupCFW` performs firewall setup in Extended mode. This requires providing not only an input port, but also an output port, in addition to low/high range, rule and read/write flags.

```
void setupCFW(unsigned int inport, unsigned int L_addr_reg, unsigned int H_addr_reg, unsigned int rule, unsigned int write_ops, unsigned int read_ops, unsigned int outport);
```

The `setupCFW` method has one more restriction regarding the output port (1-4). Other restrictions are the same for input ports (1-4), rule (0 or 1), write/read field (0 or 1), while the low/high range of memory to be protected is relative: `0x0000` to `0x3FFF` depending on input port; for example for input port 1 it is `0x0000` to `0x0FFF`. An example of parameter validation is shown below.

```
// checking parameters
      if (outport <= 0 || outport > 4){
            printf("setupCFW: Invalid value for output port:%d - must be 1, 2, 3, or 4.\n", outport);
            exit(1);
      }
      …
      …
      …
if((L_addr_reg >= H_addr_reg) || (L_addr_reg < 0x00000000 || L_addr_reg >= 0x00001000)|| (H_addr_reg < 0x00000000 || H_addr_reg >= 0x00001000)){
printf("setupCFW: Invalid range [%x %x] - must be in [0x00000000, 0x00001000].\n", L_addr_reg, H_addr_reg);
            exit(1);
      }
```

For `setupCFW` we must packetize the 8-bit rule register for Extended Mode as shown in Figure 15. The register is set as follows (notice that bit 6 must be set and an output port must be selected properly).

```
// setup output port
if(outport == 1)
   Rule_reg += 1 << 3; //Value set to 1 to enable rule for output port 1 (BRAM 0)
else Rule_reg += 0 << 3;
if(outport == 2)
   Rule_reg += 1 << 2; //Value set to 1 to enable rule for output port 2 (BRAM
```

```
0)
else Rule_reg += 0 << 2;
if(outport == 3)
   Rule_reg += 1 << 1; //Value set to 1 to enable rule for output port 3 (BRAM
0)
else Rule_reg += 0 << 1;
if(outport == 4)
   Rule_reg += 1; //Value set to 1 to enable rule for output port 4 (BRAM 0)
else Rule_reg += 0;
```

After parameter range checking, data must be packetized into kernel-level data structures (described in MLD) so that IOCTL calls can be invoked to allow user-space communication with the corresponding generic MLD `setupFW` function as shown below.

At first, the rule setup register packetizes the info in Simple mode as explained in Section 3.1.

```
// prepare rule register for complex setup
Rule_reg = rule << 7; //enable or disable rule
Rule_reg += 1 << 6;   //Value set to 1 to operate in NoC mode
Rule_reg += write_ops << 5; //enable or disable rule for write operations
Rule_reg += read_ops << 4; //enable or disable rule for read operations
// fill up setup data structure
my_fw_ds.input_port = inport;
my_fw_ds.L_addr_reg = L_addr_reg;
my_fw_ds.H_addr_reg = H_addr_reg;
my_fw_ds.Rule_reg = Rule_reg;
my_fw_ds.output_port = outport;
// ioctl call for complex fw setup (use iowrite32)
rc = ioctl(fd, IOCTL_SETUP_FW_PORT, &my_fw_ds);
if(rc < 0) {
…
```

Function `checkFWRegs` allows validating complex setup values that have written to kernel space. The function initializes data packetized into kernel-level data structures.

The `accessBram` method: performs direct access (read/write) without Switch firewall. This is accomplished by op_code (`0x0` or `0x1` for write or read, resp.), output port (`bram_no`), address range, and data.

```
int accessBram(unsigned int op_code, unsigned int bram_no,
                  unsigned int addr_reg, unsigned int *data);
```

The direct access method also places restrictions on output ports (1-4) and op_code (0x0 or 0x1). For example,

```
// checking parameters
if (bram_no <= 0 || bram_no > 4){
  printf("accessBram: Invalid value for output port.\n");
  exit(1);
}
if (op_code < 0 || op_code > 1){
  printf("accessBram: Invalid value for operation code.\n");
  exit(1);
```

```
}
```

After parameter range checking, data is packetized into the kernel-level data structures (described in MLD) and IOCTL calls are invoked to allow user space to communicate with corresponding MLD `accessBram` function as shown below.

```
// fill up the data structure
my_direct_access_ds.bram_no = bram_no;
my_direct_access_ds.addr = addr_reg;
my_direct_access_ds.op_code = op_code;
if (op_code == 0x1) // read option
  my_direct_access_ds.data = 0; // always initialize data before read operation
else
  my_direct_access_ds.data = *data;
#ifdef FW_USER_LOGS
if (op_code == 0x0)
printf("accessBram: TRY_W - BRAM%d[%x]<=%x (in decimal %u) \n", bram_no,
addr_reg, my_direct_access_ds.data, my_direct_access_ds.data);
else
printf("accessBram: TRY_R - BRAM%d[%x]=>%x (INIT in decimal %u) \n", bram_no,
addr_reg, my_direct_access_ds.data, my_direct_access_ds.data);
#endif
// direct access
rc = ioctl(fd, IOCTL_DIRECT_ACCESS_BRAM, &my_direct_access_ds);
if(rc < 0) {
...
```

The `accessBramFW` method performs access (read/write) via the NoC firewall. The packet created travels from the specific input port (`inport`) to the specific output port (`outport` which defines the BRAM), and performs write/read operation to the specific address (`addr_reg`) depending on `op_code` (0x0 or 0x1), reading or writing the data field (`data`).

```
int accessBramFW(unsigned int op_code, unsigned int inport, unsigned int
outport, unsigned int addr_reg, unsigned int *data);
```

The `accessBramFW` method has an extra restriction regarding input ports (1-4). Other restrictions are for output ports (1-4) and `op_code` (0x0 or 0x1). For example,

```
// checking parameters
if (inport <= 0 || inport > 4){
  printf("accessBramFW: Invalid value for input port:%d - must be 1, 2, 3 or 4
\n", inport);
  exit(1);
}
...
```

After parameter range checking, data is packetized into the kernel-level data structures (described in MLD) and IOCTL calls are invoked to allow user space communication with corresponding MLD accessBramFW function as explained above.

In the following Sections, we examine how the hierarchical Linux driver of the NoC Firewall can be used to support data privacy and anonymity. This is performed using one administrator and one or more user files.

3.3.3.3 System Administrator Privacy File (for Healthcare Scenario)

The administrator file systemadmin_privacy.c supports access control on all BRAMs (1-3) as shown in Figure 23 for all users by allowing only a privileged user (i.e. a system administrator) to write tables that uniquely associate groups and input/output ports in BRAM4. Then, the administrator can enter patient information to BRAMs 1 to 3, using a hashing scheme. This allows retrieving the patient name, history and a **key for accessing the ECG file** which contains the corresponding ECG data.

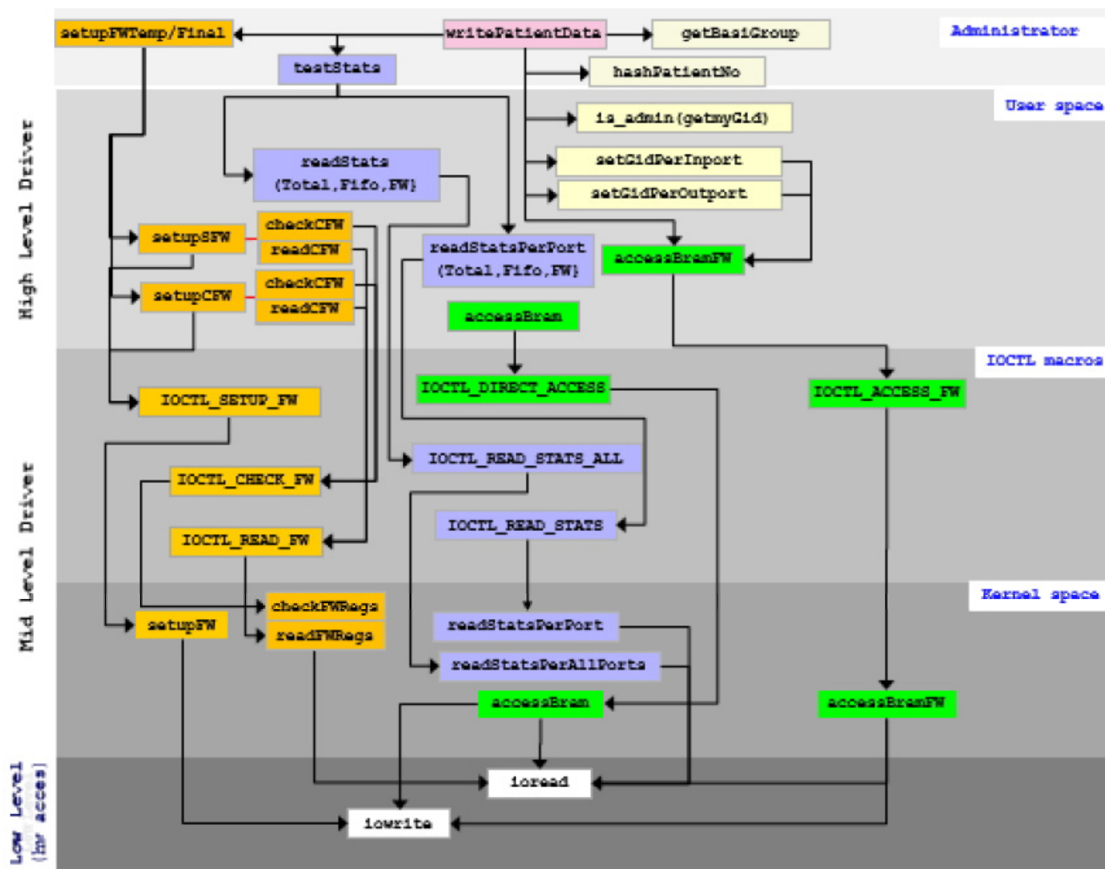

Figure 23: The NoCFW Linux driver hierarchy with full system administrator privileges.

Initially, firewall is temporarily setup via setupCFWTemp to allow the system administrator to write this info. Notice that this function must call both setupCFW and setupSFW.

```
// clinic/clinic1/clinic2 (mapped to inport 1/2/3) CANNOT access anything
//inport, L_addr_reg, H_addr_reg, rule, write_ops, read_ops, outport
setupCFW(1, 0x00000000, 0x00000100, 1, 0, 0, 1);
setupCFW(2, 0x00000000, 0x00000100, 1, 0, 0, 2);
setupCFW(3, 0x00000000, 0x00000100, 1, 0, 0, 3);
// admin CAN access BRAM4 in read/write mode
setupSFW(ADMIN_OUTPUT_PORT, 0x40003000, 0x40003100, 1, 0, 0);
```

Then, as explained before, after setting the tables, system administrator calls writePatientData to write patient data (e.g. name, history, key) to BRAMs 1, 2 and 3 for groups fwgroup1, fwgroup2, fwgroup3.

```
void writePatientData(unsigned int patient_clinic, unsigned int patient_no, char
```

`*mac_address, char *patient_name).`

This function calls the function `gid_t getBasicGroup()` which is very similar to `is_admin()` function. This method first certifies the associated (current) group id against root, by calling first calling `groupIdFromName()` to obtain the specific group id for root and then calling `grouplist(list_of_groups, &ngroups)` to validate the current group id with root, as shown in this code segment.

```
//save fwgroup groupid
gid_t gid_fwgroup = groupIdFromName("root");
//check if the current user id  belongs to systemadmin group (fwgroup)
grouplist(list_of_groups, &ngroups); //call function grouplist to take in which groups belong each current user
for (i=0; i<ngroups; i++)
   if (list_of_groups[i] == gid_fwgroup) {
      myGid = gid_fwgroup;
break;
   }
      return myGid;
}
```

If system administrator is verified, it can call `accessBramFW` function to write groups in input and output ports via NoC Firewall.

```
// Write patient data to specific (input port, ouput port, offset in BRAM)
// op_code (0:w), inport, outport, addr_reg, unsigned int *data
accessBramFW(0x0, inport,  bram_no, addr_reg, &patient_no);
#ifdef FW_USER_LOGS
Use of user_mode accessBram function to read (ox1) if patient data has written in Bram
// check values directly to bram
written_value=0xFFFFFFFF;
//op_code (0:w), inport, outport, addr_reg, unsigned int *data
accessBram(0x1, bram_no, addr_reg, &written_value);
printf("writePatientData:  BRAM%d[%x)<-(%d  %d)  (patient_no)  \n",  bram_no, addr_reg, patient_no, written_value);
#endif
```

At this point, we can write patient data to Bram via `accessBramFw`.

For example,

```
// Write MAC to specific (input port, ouput port, offset in BRAM)
for(i=0; i<6; i++){
      addr_reg = N*OFFSET_PATIENT_NO + OFFSET_MAC_ADDR + (0x4*i);
      //op_code (0:w), inport, outport, addr_reg, unsigned int *data
      accessBramFW(0x0, inport, bram_no, addr_reg, &iMac[i]);
```

while also checking if MAC address has been written to BRAM by calling from user space the HLD direct access function `accessBram` (op code `0x1`).

```
#ifdef FW_USER_LOGS
// check values by reading directly bram
written_value=0xFFFFFFFF;
```

```
accessBram(0x1, bram_no, addr_reg, &written_value);
printf("writePatientData:  BRAM%d[%x]<-(%x  %x)  (patient_mac)  \n",  bram_no,
addr_reg, iMac[i], written_value);
#endif
```

We also write patient number in BRAM via `accessBramFw`.

```
for(i=0; i<(name_size + 1); i++){
addr_reg = N*OFFSET_PATIENT_NO + OFFSET_PATIENT_NAME + (0x4*i);
#ifdef FW_USER_LOGS
printf("writePatientData:  (inport:  %d)  -  BRAM%d[%x]<-%c   (patient_name)  (in
decimal %d) \n", inport, bram_no, addr_reg, iName[i], iName[i]);
#endif
accessBramFW(0x0, inport, bram_no, addr_reg, &iName[i]);
```

while also checking if patient number been written to BRAM by calling from user space the HLD direct access function `accessBram` (op code `0x1`)).

```
#ifdef FW_USER_LOGS
// check values
accessBram(0x1, bram_no, addr_reg, &written_value);
printf("writePatientData:  BRAM%d[%x]<-(%c  %c)  (patient_name)  \n",  bram_no,
addr_reg, iName[i], written_value);
#endif
```

Later after group tables and patient data are written, the firewall setup is modified to limit user access to these tables by calling `setupFWFinal`. Notice that `clinic`, `clinic1`, `clinic2` (mapped to input ports 1, 2, and 3, respectively) **can access only** the corresponding BRAM 1, 2, or 3 for read only.

```
//inport, L_addr_reg, H_addr_reg, rule, write_ops, read_ops, outport
setupCFW(1, 0x00000000, 0x00000100, 1, 1, 0, 1);
setupCFW(2, 0x00000000, 0x00000100, 1, 1, 0, 2);
setupCFW(3, 0x00000000, 0x00000100, 1, 1, 0, 3);
// clinic/clinic1/clinic2 (ALL) CAN access BRAM4 for read-only. Since they come
from diff ports, must use SFW
//inport, L_addr_reg, H_addr_reg, rule, write_ops, read_ops
setupSFW(ADMIN_OUTPUT_PORT, 0x40003000, 0x40003100, 1, 1, 0);
```

Group tables are used to limit access from specific user groups only to the predefined (by the administrator) NoC firewall input/output ports, thereby limiting access to BRAMs via the NoC firewall. In our current setup, each group corresponding to a different hospital department would have access to a different BRAM containing data for all its patients; group-port associations can either be placed in the same BRAM as discussed before (shared across different groups, with protection enforced by group id), or alternatively in different BRAMs (in which case protection is enforced by the firewall rule).

3.3.3.4 FWGroup Privacy File (for Healthcare Scenario)

This scenario provides read access to all BRAMs (1, 2 and 3) respectively from groups (`fwgroup1`, `fwgroup2` and `fwgroup3`) as set up by system administrator (`systemadmin_privacy.c`). There are 3 files: one for fwgroup1 (`fwgroup1_privacy.c`), another for fwgroup2 (`fwgroup2_privacy.c`) and finally one for fwgroup3 (`fwgroup3_privacy.c`). Each group consists of simple users that can be

considered as physicians or healthcare personnel that is given permissions to only read data from the corresponding BRAM; as we will show later the maximum theoretical value for the number of physicians with current setup is 64. Access to data is made via `readPatientData` (which sets the ports properly and calls `accessBramFW` function to read only) as shown in Figure 24.

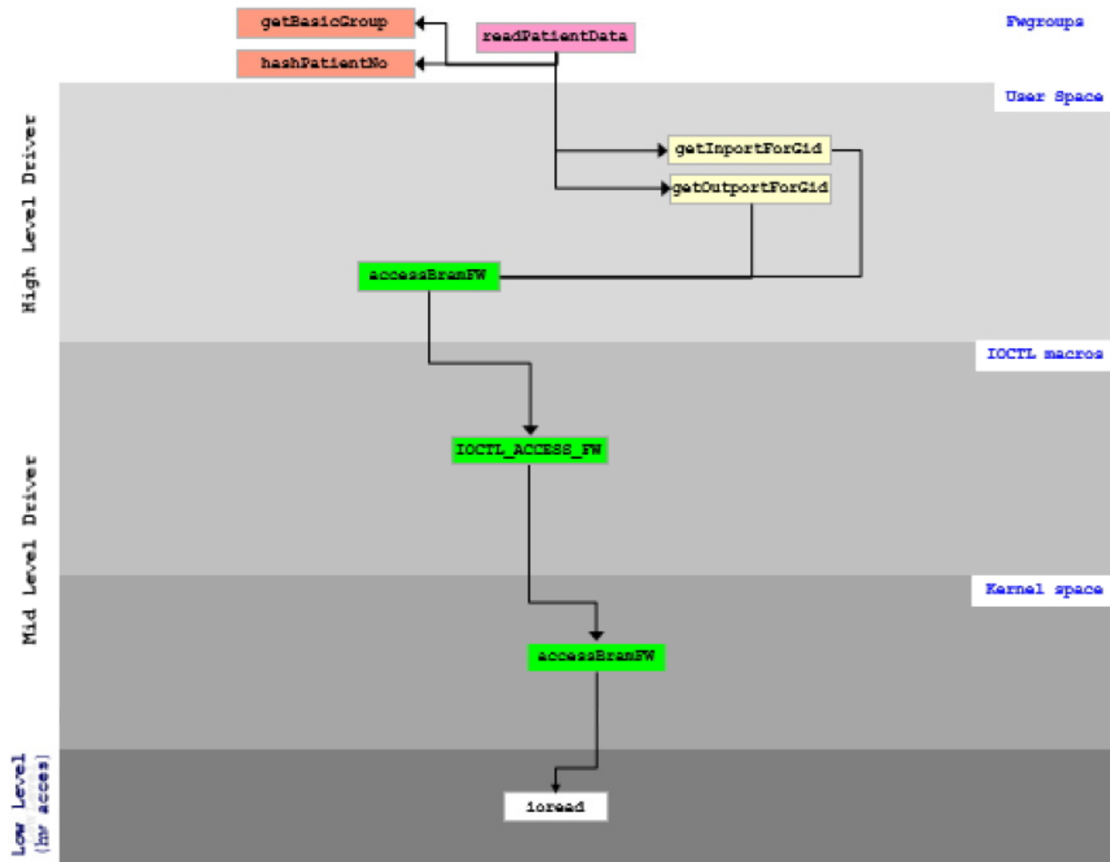

Figure 24: The NoCFW Linux driver hierarchy with basic group (fwgroup) privileges

In addition to user access control, an elaborate hashing mechanism (`hashPatientNo`) is used to identify offsets with which different healthcare patient data (patient no, mac address, patient name for anonymity) is stored in BRAM (typically by root).

```
int x = keccak(patient_no)%16; // use Keccak sha-3 algorithm to hash to
                               // 16 different regions in each BRAM
return x;
```

This data can be subsequently read by simple users (physicians) via `readPatientData` function depending on the group-port associations assigned by the system administrator. The code is similar to

```
//gid_t myGid = groupIdFromName("fwgroup");
gid_t myGid = getBasicGroup();
inport = getInputPortPerGid(myGid);
bram_no = getOutputPortPerGid(myGid);
printf("readPatientData:myGid:%d->inport:%d bram_no:%d\n",myGid,inport,bram_no);
N=hashPatientNo(patient_no);
printf("readPatientData: patient_no:%d H:%d\n", patient_no, N);
```

```
bram_patient_no = 0xFFFFFFFF; // overwritten in order to be read from BRAM again
//(op_code, inport, outport, addr_reg, *data)
addr_reg = N*OFFSET_PATIENT_NO;
printf("ok1\n");
```

At this point read patient number from BRAM via `accessBramFw`.

```
rc1 = accessBramFW(0x1, inport, bram_no, addr_reg , &bram_patient_no);
printf("ok2\n");
printf("readPatientData: Read_BRAM (inport:%d - output_port:%d - offset:%d) ->
bram_patient_no:%d  H:%d\n", inport, bram_no, addr_reg, bram_patient_no, N);
```

If `rc1>0` and the patient number provided agrees with the one in BRAM, we can proceed to read patient info from the BRAM by calling the HLD user mode `accessBramFW` function as shown below. For example,

```
if((rc1 >= 0) && (patient_no == bram_patient_no)){
// read mac_address in specific inport_and_bram_no
for(i=0; i<6; i++){
// (op_code, inport,  outport, addr_reg, *data)
addr_reg = N*OFFSET_PATIENT_NO + OFFSET_MAC_ADDR + (0x4*i);
rc2 = accessBramFW(0x1, inport, bram_no, addr_reg, &iMac[i]);
printf("readPatientData: Read_BRAM (inport:%d - output_port:%d - offset:%d) ->
mac: %x (in decimal %d) \n",inport, bram_no, addr_reg, iMac[i], iMac[i]);
}
```

Patient name data and key are also retrieved from BRAM via HLD user_mode `accessBramFW` function.

```
for(i=0; i<22; i++){
// (op_code, inport,  outport, addr_reg, *data)
addr_reg = N*OFFSET_PATIENT_NO + OFFSET_PATIENT_NAME + (0x4*i);
rc1 = accessBramFW(0x1, inport, bram_no, addr_reg, &iName[i]);
}
```

Notice that function `writePatientData (unsigned int patient_no, char *key, char* patient_name)` can only be called from root, otherwise it will fail. In fact, in our healthcare scenario, we allow only privileged users to perform write access via firewall. Such failures (and reasons behind them) can be easily detected by calling `testStats` which provides access to statistics logs.

3.3.3.5 Security Overheads

The administrator cost for involving the Linux kernel-space driver to setup anonymity service relates to

- mapping group id to input/output ports of the NoC Firewall; this takes on average 0.72ms and 1.45ms, respectively,
- setting up the firewall (initially for administrator setup, and later for user); this takes on average 4.42ms and 4.76ms.
- storing the patient key in BRAM by invoking sha-3 hash (keccak algorithm) takes on average 17.10ms.

Finally, retrieving the anonymization key info in order to start streaming the healthcare data (specifically, the ECG signal) takes an average of 10.08ms. Unlike AES cryptography, anonymity costs are small compared to total cost of soft real-time healthcare application which are in the order of 1 sec. Although, the healthcare application requires optimization and porting to a parallel ARMv8 server to support efficiently more BGW devices without missing deadlines, it is obvious that data protection costs are relatively small.

3.4 Hierarchical Driver Validation

Different types of tests have been developed to debug and validate our hierarchical driver.
The first tests focused on validating direct BRAM access for different BRAMs and offsets. This test is simple and requires sequential write/read accesses and validating the output. For example,

```
accessBram(0x0, 1, addr_reg, &bram_write);
accessBram(0x1, 1, addr_reg, &bram_read);
```

Once these tests worked, complete coverage tests were designed to verify read/write access via Firewall for all possible input and output ports and for all possible different settings of the firewall (Simple and/or Extended Mode). For simplicity, we assumed in our tests a limited address range (LOW, HIGH), exactly the one used in our healthcare scenario.
These tests were designed to take into account (and avoid) synchronization issues related to accessing BRAM via two different paths: a) via NoC firewall and b) via direct access for testing. For this reason, as we explain below, certain read/write accesses were made sequential.
In the *Write Test*, we perform the following steps.

1. Setup NoC Firewall independently for each output port, by specifying *deny read, deny write, deny read & write or accept all*. This gives a total of $4^4 = 256$ different combinations. If we allow mixing of Simple and Extended Mode, we have a total of 512 different cases.
2. Perform direct write to each BRAM (at a specific offset) from each input port and subsequently verify with subsequent direct read from BRAM that data (with a default value w) has indeed been written. Assuming testing for a single BRAM offset, the total number of (orthogonal) cases corresponding to input/output port combinations is 4 x 4 = 16, raising the total number of unique cases to 16 x 512 = 8192. The direct read operation is needed to avoid the possibility of write/read reorder (possible case on ARM Cortex-A9). A code snippet for this step is as follows.
```
rc1=accessBram(0x0, inport, 0x0, &w);
do { // read
rc2=accessBram(0x1, inport, 0x0, &r);
} while (r != w);
```
3. Write BRAM memory location via NoC Firewall with a different (always increasing) value z.
```
rc3=accessBramFW(0x0, inport, outport, 0x0, &z);
```
4. Read the BRAM value directly (value x). The read operation examines the return code from previous write operation to avoid the possibility of a write/read reorder. A code snippet for this step is as follows.
```
if (rc3 == 0)
rc4=accessBram(0x1, inport, 0x0, &x);
```
5. Finally, we call a custom `check_write` function with the complete setup pattern, x and z. For each port, the above *Write Test* ***passes*** *if and only if*
   - values x and z are equal and the setup pattern for that port is deny read (accept write) or accept all, or
   - values x and z are non-equal and setup pattern for that port is deny write (accept read) or deny all.

Assuming that write via NoC Firewall (as demonstrated by the Write Test) works correctly, a *Read Test* can be performed in a relatively simple way. After writing an increasing value to BRAM, a subsequent read

operation via the NoC Firewall is performed, and data is compared with the data written. The number of cases is alike the Write Test, i.e. 8192 cases.

Thus, in the *Read Test*, we perform the following steps. For example,

1. Write BRAM memory location via NoC Firewall with a different (always increasing) value `x`
   ```
   rc1 = accessBramFW(0x0, 1, 1, addr_reg, & x);
   ```
2. Read BRAM memory location via NoC Firewall in `z`.
   ```
   rc2 = accessBramFW(0x1, inport, outport, addr_reg, &z);
   ```
3. Finally, we call a custom `check_read` function with the complete setup pattern, `x` and `z`. For each port, the above *Read Test* ***passes*** *if and only if*
   - values `x` and `z` are equal and the setup pattern for that port is deny read (accept write) or accept all, or
   - values `x` and `z` are non-equal and setup pattern for that port is deny write (accept read) or deny all.

Chapter 4 - Experimental Framework: Security in ECG Processing

The hierarchical Linux drivers presented in Chapter 3 are applied to the healthcare scenario. In this scenario, we consider *soft real-time smartphone monitoring of patients outside the hospital environment* with the aim to evaluate performance overheads for supporting transmission security and anonymity. More specifically, patients wearing the ST Microelectronics BodyGateway device (BGW) and carrying an Android device (usually a smartphone) with a Bluetooth (BT) connection to the device and a WiFi connection to a Cloud environment (receiving data and performing ECG analysis) can help medical personnel monitor annotated patient signals on their own smartphone.

Notice that annotations are added asynchronously in on top of the ECG signal to indicate specific chronic cardiac diseases, more specifically non-fatal arrhythmias, such as ventricular cardiac events. Some common cardiac arrhythmias, such as atrial fibrillation, can be associated with embolic strokes, while ventricular tachycardia can herald sudden cardiac death.

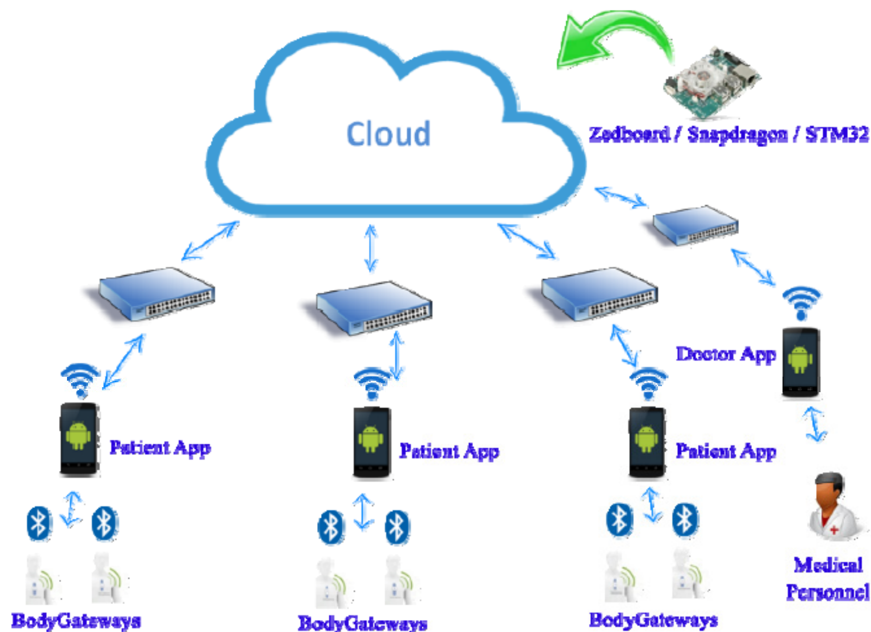

Figure 25: The healthcare application.

More specifically, as shown in Figure 25, a *Patient App* executing on a client device (e.g. an Android smartphone) connects to BT to receive ECG data from the BGW device, and Wi-Fi 802.11 to retransmit it to a *distributed Linux-based Cloud system architecture* (e.g. based on ARMv6, v7 or v8 boards) for ECG Analysis. Then, a *Doctor App* executed by medical personnel on a similar Android device can connect to the Cloud infrastructure requesting to monitor and visualize ECG data for a given patient.

4.1 Patient App

The Patient App allows the patient to configure connections to BT device (pairing) and to the cloud server (WiFi). Before transmitting ECG data to the Cloud, the Patient App **also sends a unique *Patient No*** and current Android patient device time when the device connects to the BGW.

- **The Patient No provides a way to support anonymization, i.e. it uniquely determines (more specifically, hashes to) the MAC Address of the BGW that the patient is wearing, as well as patient data**. Our anonymity scheme may also permit to conduct a variety of post-processing analysis on aggregated unencrypted anonymized ECG data on the Cloud server, without endangering privacy of the patients.
- The **timestamp defines a relevant time point which can be used to provide metrics related to real-time operation**. This can help identify issues, e.g. when BGW device enters storing mode (saving data instead of transmitting due to BT connection delay) or if the WiFi connection from Patient App to the Cloud or from the Cloud to Doctor App has failed or inactive.

4.2 Zedboard Server for ECG Processing

In the patient Application that is referenced above, we use a Xilinx Zedboard FPGA platform associated with our hierarhical Linux drivers. The Xilinx Zedboard FPGA platform works as a server node in the distributed cloud system that is mapped (e.g. via a DNS service) to a number of smartphone devices. Zedboard is also configured with our NoC Firewall on FPGA, associated Linux drivers equipped, a `server_rx` process that receives ECG data, and our cardiac heartbeat detection and classification algorithms which are based on WFDB and OSEA libraries (developed in C language).

We first perform conversion from a BGW ECG signal with 256 samples/second to a standard ECG-13 compliant format with 200 samples/second using WFDB's `wrsamp` function which creates 2 new files: a binary "`.dat`" file which contains the standardized ECG signal and an asci "`.hea`" file which contains significant info about ECG data stored in the previous file e.g. the total samples.

Then, we perform heart beat detection and classification using a modified version of `easytest` algorithm in order to generate a binary "`.atest`" file which contains the annotation tags. This tool originates from OSEA (Open Source ECG Analysis) and uses different types of filters (e.g. noise reduction, QRS, SQRS) and related computations (R-R interval) on a standardized ECG-13 compliant signal to classify each detected beat as Normal or Ventricular.

The `easytest` tool is part of Harvard's WFDB Physionet software which provides a stable ECG detection and analysis tool. The tool provides a QRS detection feature which achieves very high positive predictivity when using MIT/BIH and AHA arrhythmia databases. In our case, we examine issues related to real-time annotation of the ECG signal received from the sensor, without excessive re-computation by redrafting the design of the `easytest` algorithm.

Using the above WFDB and OSEA libraries, the Zedboard server is able to receive via WiFi, decrypt, store, convert to a standard format and eventually process the received ECG signal compliant with ECG13 standard for heart beat detection and classification. Heart diseases (and in our case arrhythmias) are detected automatically by the presence of ventricular cardiac events during ECG processing.

In addition, a `server_Tx` process is in charge for communicating asynchronously with the Doctor App when a request is made to retrieve a given patient's ECG File. This is better described in Patient App. Notice that Patient No is used by the server (in fact by a `server_Tx` child process which inherits privileges of the connecting doctor) to access patient data via the NoC Firewall from an appropriate BRAM (according to group permissions) and a given location (via a hash). Patient data includes private info (patient name, sex, etc), as well as a key related to the requested patient's annotated ECG file. This key is used to encrypt this file before transmitting this file to Doctor App (when the lock is obtained).

4.2.1 Doctor Application

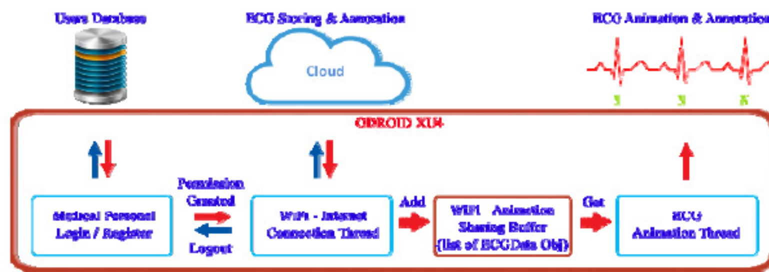

Figure 26: Doctor App Workflow Diagram

As shown in Figure 26, the Doctor App implements an *authentication mechanism* based on a Medical personnel ID to control access to sensitive medical data stored in the cloud. Then, upon successful login, the Doctor must select to monitor a specific unique *Patient No*, thus providing *anonymized access* to the patient's vital signs stored on the server.

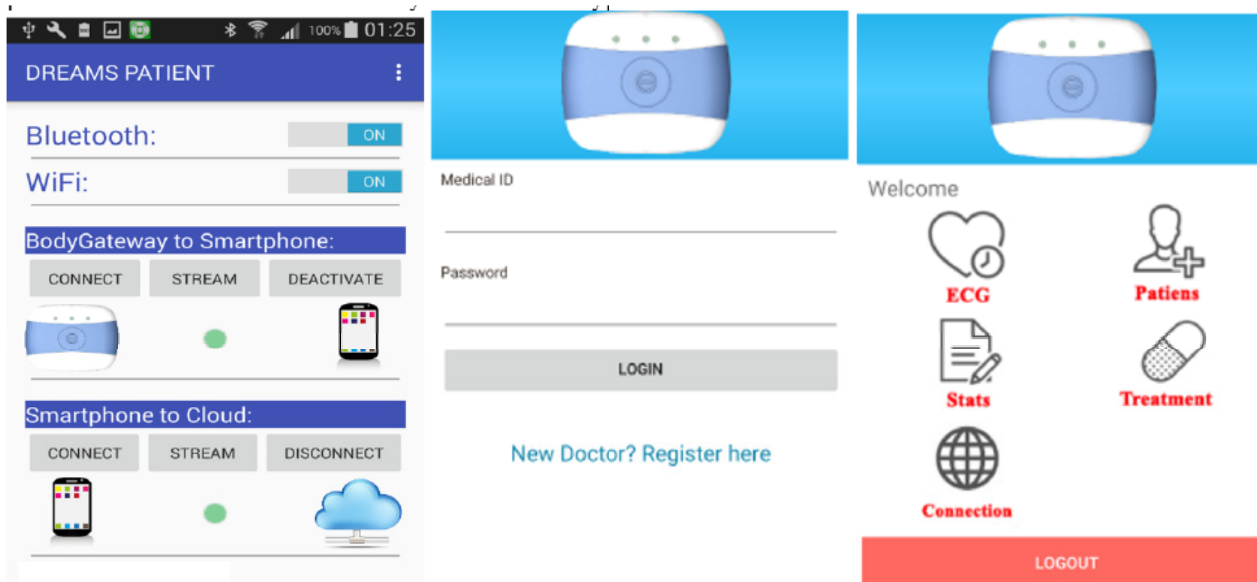

Figure 27: GUI for Patient (BT and WiFi) and Doctor App (authentication, and WiFi)

After sending the Patient No. a WiFi Connection Thread receives ECG data (which includes an incremental sample id, the ECG signal, and a possibly asynchronous annotation) from Cloud server and appends it into a shared buffer (linked list) and eventually a file. Another ECG Animation Thread also accesses this file (via a file lock) in order to decrypt info using the same key, use MPAndroidChart library to chart ECG time series plot for the corresponding points, eventually emptying the file. Asynchronous posting of annotations on the cart appears as a character "N" for Normal and "V" for Ventricular, see Figure 22. This graph shows the value of the ECG Signal with the units (mV), and provides calibration, scroll and zoom functions. the Cloud server to Patient App to allow patients to monitor their heart rate signal. Another potential extension refers to logging patient data and providing high-level notification services to medical personnel in real-time.

4.3 DDoS Attack via NoC - Cache Thrashing

This Section focuses on the NoC Firewall and examines BRAM protection from on-chip malicious processes (e.g. malicious drivers or corrupt devices). We examine how network performance deteriorates when more than one kernel threads use system resources.

4.3.1 Kernel Level Attack: exploit_firewall module

Different tests of involving attacks to BRAM have been developed to examine and measure BRAM performance. In order to plan these attacks a second module (called `exploit_firewall`) is created that communicates with the initial module named `sw_firewall` as shown in Figure 23. A user can select different scenarios to run, whereas each scenario defines

- a varying number of kernel threads (safe and malicious) to access physical memory via I/O write (we examine only `iowrite`, since read operation is blocking, thus posing additional restrictions), and
- sets the appropriate deny rules.

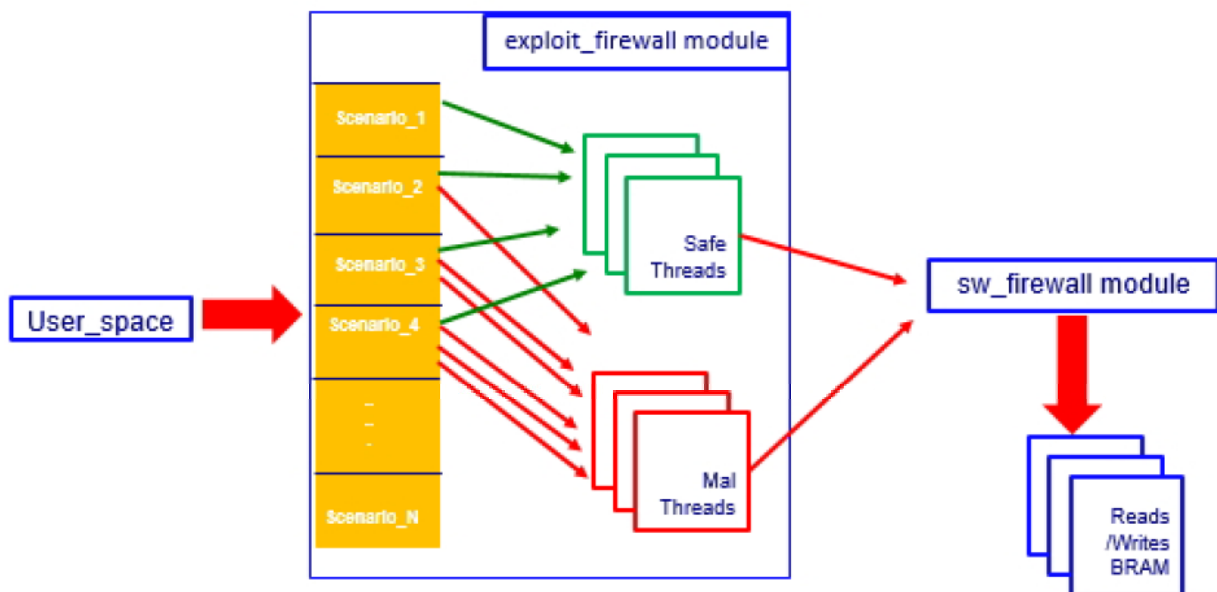

Figure 28: Kernel Level Attacks

More specifically, the exploit module provides the following API; notice that these functions call NoC Firewall module functions to access FPGA memory (Figure 29), i.e. the kernel firewall module must have been installed first.

```
my_write(struct file *f, const char __user *buf, size_t len, loff_t *off);
my_read(struct file *f, char __user *buf, size_t len, loff_t *off);
static void setup_FW_Attack();
static void define_scenario_1();
static void define_scenario_2();
static void define_scenario_3();
static void define_scenario_4();
static int safeFun(void *arg);
static int malFun(void *arg);
```

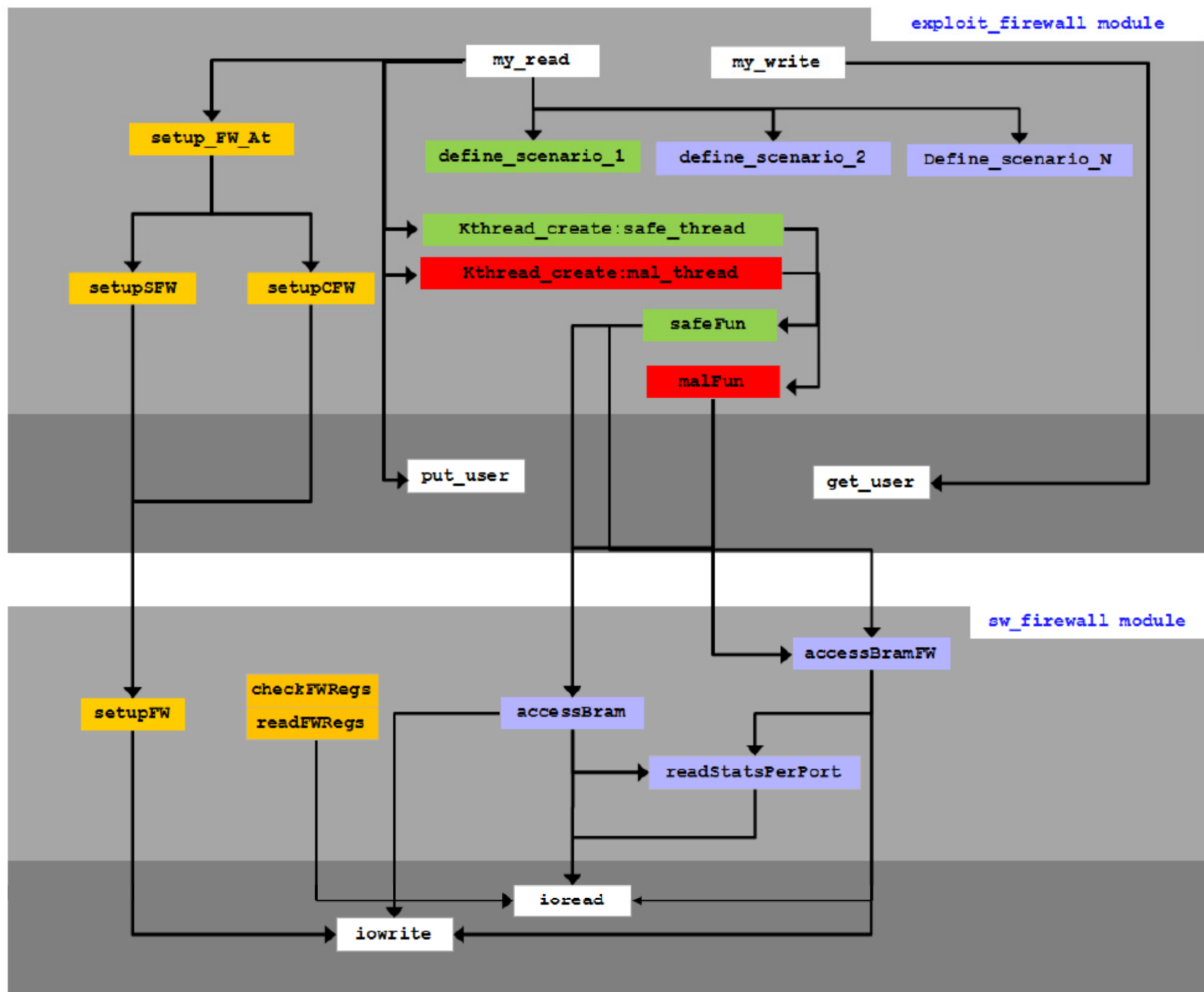

Figure 29: Communication of exploit firewall module with NoC firewall module

`my_write` method: The `exploit_firewall` module allows the user set the scenario by providing messages (echo command with scenario number, mode of firewall access and access method via `my_write` method. An example of code can see below.

```
for (i = 0; i < len && i < BUF_LEN; i++)
   get_user(Message[i], buf + i);
   Message_Ptr = Message;
   printk(KERN_INFO "Incoming message: %s\n", Message_Ptr);
   //two arguments in my_write_function
    if (sscanf(Message, "%d %d %d", &scenario_num, &fw_type, &fw_access) != 3) {
             printk (KERN_INFO KERN_ALERT "Invalid input: \"%s\".\n", Message);
             return -EINVAL;
    }
   else
             printk (KERN_INFO "scenario_num:%d fw_type:%d fw_access:%d \n",
                       scenario_num, fw_type, fw_access);
```

The user passes three messages:

1. The scenario (scenario_num) that must be executed. For example, the values 1, 2, 3 and 4 correspond to scenario_1, scenario_2, scenario_3, and scenario_4.
2. The firewall setup (fw_type) defined either Simple or Extended mode. For example, if fw_type=0 the firewall is setup using setupSFW, while if fw_type=1 we perform complex setup (Extended mode) using setupCFW. The firewall setup itself is performed in setup_FW_Atttack method.
3. The firewall access field (fw_access) defines direct access or access via firewall. For example, if fw_access=0 we invoke accessBramFW to access BRAM via the NoC Firewall, while if fw_access=1 invoke accessBram to access BRAM directly.

my_read method: The messages passed from user can be read via my_read method (cat command). Here, we can read our scenarios in kernel space. For example,

```
switch(scenario_num){
      case 1:
            define_scenario_1();
            break;
      case 2:
            define_scenario_2();
            break;
      case 3:
            define_scenario_3();
            break;
      case 4:
            define_scenario_4();
            break;
      default:
            printk(KERN_INFO "Wrong scenario number\n");
            sprintf(Message, "The scenario_num is out of range (1-2)\n");
                while (len && *Message_Ptr) {
                        put_user(*(Message_Ptr++), buf++);
                        len--;
                        bytes_read++;
                }
                off += bytes_read;
                return bytes_read;
```

The exploit_firewall module supports simultaneous (read or write) access to Bram by creating two kinds of kernel threads named safethread and malthread (malicious thread). These threads are created upon a call to my_read method, and the safe thread (safe_thread) calls safeFun, while the malicious thread (mal_thread) calls malFun function.

```
//Create kernel threads for malicious users
for (i=1; i<=safe_num; i++){
safe_thread[i-1] = kthread_run(safeFun,(void*)i, "safethread");
}
//Create kernel threads for safe users
```

```
for (i=1; i<=mal_num; i++){
   mal_thread[i-1] = kthread_run(malFun,(void*)i, "malthread");
}
```

`setup_FW_Attack method:` This method uses `(fw_type)` to manage the Firewall setup.

If `fw_type=0`, we use simple firewall setup `(setupSFW)` in Normal mode. Thus, user can select `inport`, `L_addr_reg`, `H_addr_reg`, `rule`, `write_ops`, and `read_ops`. In the example below, simple firewall setup is used to allow access from input port `i` to $BRAM_i$ `i=1,2,3,4`.

If `fw_type=1`, we activate complex setup `(setupCFW)`. Hence, user can select the `inport`, `L_addr_reg`, `H_addr_reg`, `rule`, `write_ops`, `read_ops`, and `outport`. In this case, we allow read/write access from input port 1 to BRAM1, but disallow access to BRAM1 from all other input ports.

```
if(fw_type==0){
   //simple inport, L_addr_reg, H_addr_reg, rule, write_ops, read_ops
   setupSFW(1, 0x40000000, 0x40000fff, 1, 0, 0);
   setupSFW(2, 0x40001000, 0x40001fff, 1, 0, 0);
   setupSFW(3, 0x40002000, 0x40002fff, 1, 0, 0);
   setupSFW(4, 0x40003000, 0x40003fff, 1, 0, 0);
}
...
else if(fw_type==1){
   //complex inport, L_addr_reg, H_addr_reg, rule, write_ops, read_ops, outport
   setupCFW(1, 0x00000000, 0x00000fff, 1, 0, 0, 1);
   setupCFW(2, 0x00000000, 0x00000fff, 1, 1, 1, 1);
   setupCFW(3, 0x00000000, 0x00000fff, 1, 1, 1, 1);
   setupCFW(4, 0x00000000, 0x00000fff, 1, 1, 1, 1);
}
```

`scenario_N methods:`

There are four scenarios that focus on simultaneous BRAM access from malicious and safe users (threads) (via I/O read/write). All scenarios dynamically create a number of safe threads `(safe_num)` and a number of malicious threads `(mal_num)` that all access the same Bram (e.g. BRAM1). The four scenarios that used are:

- Scenario_1: 1 safe thread
- Scenario_2: 1 safe – 1 malicious threads
- Scenario_3: 1 safe – 2 malicious threads
- Scenario_4: 1 safe – 3 malicious threads

The code in the following example refers to `scenario_2`. In this scenario, we run 1 safe and 1 malicious thread. Each thread accesses the same BRAM (BRAM1) from a different input port (safe from input port 1, while malicious from input port 2). All scenarios follow the same logic.

```
safe_num = 1;
mal_num = 1;
safe_ptr=kmalloc(safe_num*sizeof(struct attack_access_ds), GFP_KERNEL);
mal_ptr=kmalloc(mal_num*sizeof(struct attack_access_ds), GFP_KERNEL);
//Safe user
```

```
safe_ptr[0].op_code=0; //write access
safe_ptr[0].input_port=1;
safe_ptr[0].output_port=1;
safe_ptr[0].addr=0x0;
safe_ptr[0].data=0xA;
// Malicious user
mal_ptr[0].op_code=0;  //write access
mal_ptr[0].input_port=2;
mal_ptr[0].output_port=1;
mal_ptr[0].addr=0x0;
mal_ptr[0].data=0xB;
```

safeFun & malFun methods:

Each thread (safe, malicious) calls the corresponding function (`safeFun`, `malFun`) to perform a number of simultaneous accesses (e.g. 200.000) to Bram via the NoC Firewall or directly. Moreover, by calling `ktime_get()`, we can get an insight into driver performance.. An example of safeFun functionfollows.

```
if (fw_access==0){
//start timer
  start = ktime_get(); //returns nanosecond resolution
for(i=0;i<200000;i++) { //times that thread read Bram(memory)
  accessBramFW(&my_access_ds);
}
//end timer
  end = ktime_get();
  cur_access_time = ktime_to_ns(ktime_sub(end, start)); //Returns the remainder
of the substraction in ns
  printk(KERN_INFO "_SAFE MODULE with id:%d - Time pased for protected memory
access: %llu\n", i,  cur_access_time);
 }else if(fw_access==1){
start = ktime_get(); //returns nanosecond resolution
for(i=0;i<200000;i++) {
  accessBram(&my_direct_access_ds);
//end timer
  end = ktime_get();
  cur_access_time = ktime_to_ns(ktime_sub(end, start)); //Returns the remainder
}
```

In Figure 30, we show performance for four scenarios: 1 Safe (called 1S), 1 Safe – 1 Malicious (1S -1M), 1 Safe – 2 Malicious (1S - 2M), 1 Safe – 3 Malicious (1S – 3M) that perform simultaneous access (write) to BRAM. More specifically, we measure the average delay of write accesses when other threads attack in the same Bram (Bram 1). Notice the gradual increase of the delay with the number of malicious threads.

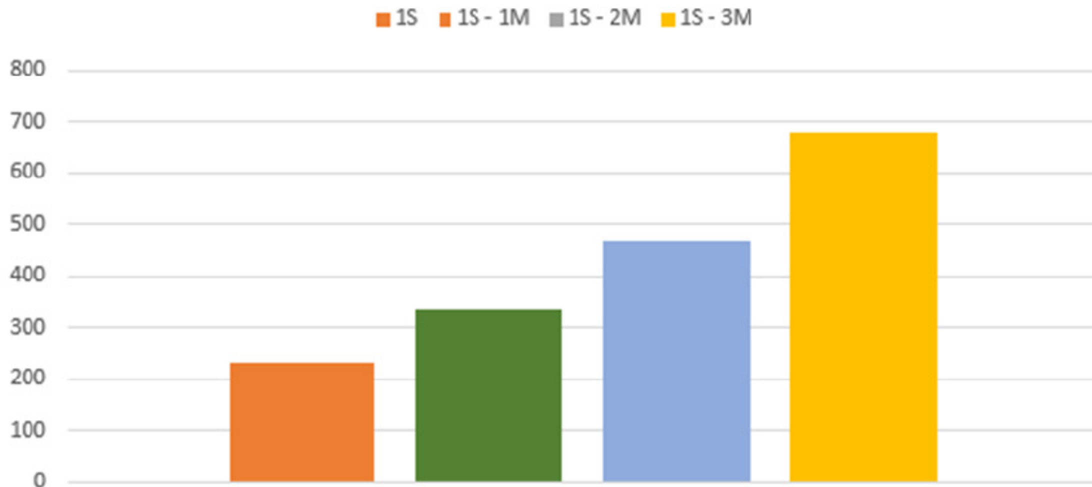

Figure 30: Average BRAM access delay (ns) for different exploit scenarios

Similar to Figure 30, Figure 31 presents an online xmgrace plot derived from a demo. We examine access time for four different scenarios: 1 Safe (called 1S), 1 Safe – 1 Malicious (1S -1M), 1 Safe – 2 Malicious (1S - 2M), 1 Safe – 3 Malicious (1S – 3M) that perform simultaneous access (write) to BRAM. More specifically, we measure the time of write access when all threads one-by-one start attacking the same Bram (Bram 1). Notice the gradual increase of the delay and gitter with the number of malicious threads.

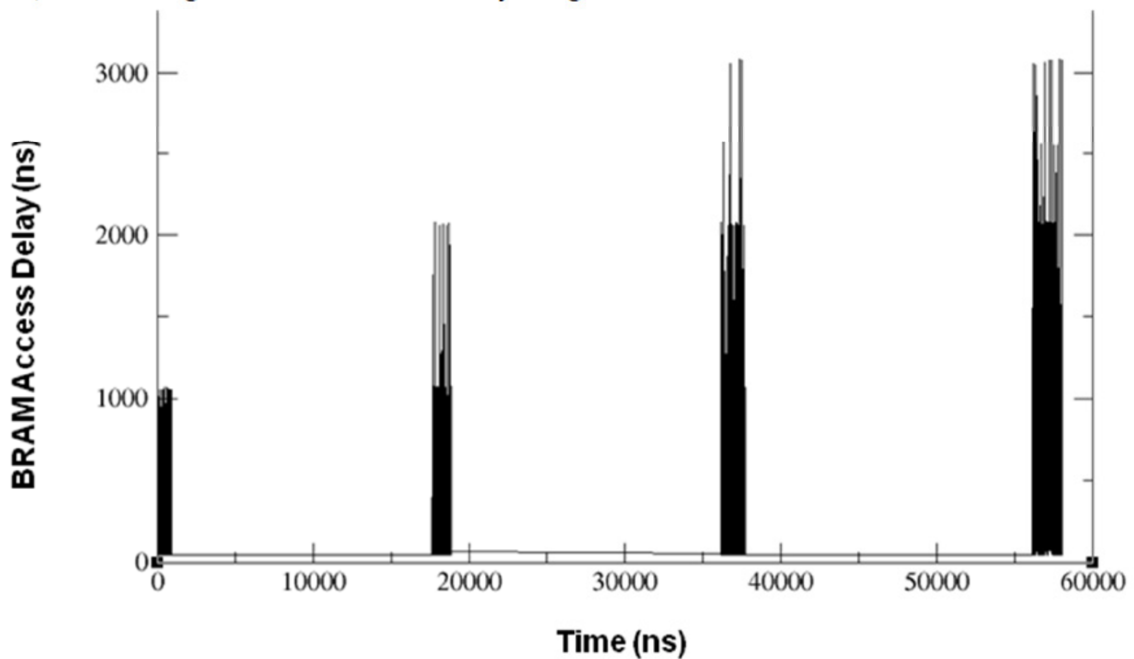

Figure 31: BRAM access delay vs Time (in ns) for different exploit scenarios with xmgrace

Although, we have disabled the firewall for all ports (e.g. input port 1, 2, 3, 4), delay did not decrease significantly. This is due to the fact that the extra overhead for accessing the BRAM is small compared to accessing the firewall module in FPGA and network congestion is limited due to packet drops at intermediate and final stages of the NoC that are not accounted by our statistics methods.

More specifically, design of the NoC Firewall has used FIFOs (`sc_fifo`) for interconnecting switches to

implement the NoC, with a packet drop policy adopted in the NoC if the number of packets queued in the switches exceeded the size of the buffers (i.e. a `nonblocking sc_fifo write` operation is used). Due to packet drops at intermediate and final stages of the NoC, the input buffer never becomes full in our experiments (and probably never contains more than one packet) and delays at intermediate and final stages of the NoC cannot be accounted for.

If a push-back mechanism had been implemented in switches at different stages of the NoC, then the delay would be dependent on the number of packets entering the network. Hence, in this case, firewall operation would limit the number of packets entering the network packets and could help reduce delay or exhibit less packet drops due to a full FIFOs at input buffers. With the current setup, the only metric that could help estimate congestion would be the number of NoC drop packets at intermediate stages of the NoC. However, the current hardware design does not support this metric (only at the input switches).

4.4 High-Level Security

4.4.1 Event Monitoring Tools

Event logs and monitoring tools are important for detecting malicious activities and suspicious traffic. Many systems or network devices can use log events to monitor security vulnerability attacks and issue alerts. Thus, event logs play an important role in information and data security and several different methodologies and tools have been developed.

Most popular security event monitoring tools follow one of two different approaches. Certain tools track suspicious network traffic in real-time, such as SEC [17], SwatchTools and Techniques for Event Log [18], Logsurfer [19], and OSSEC[20] while other tools, such as logwatch [21] and SLAPS-2 [21] focus on offline monitoring analysis, i.e. first capturing log events in a file and then processing them. Offline tools differ in the type of log files that can be processed, e.g. Guardian Files, OSS, EMS, VHS logs.

Next, we concentrate on real-time monitoring tools only, in particular SEC which is used in our application.

4.4.2 Real time monitoring tools

Swatch is a real-time event monitoring program based on Perl language [18]. Similar to SEC, Swatch analyzes log files by examining rules described as regular expressions in a configuration file. In this approach, if each line of the log file matches a regular expression, we carry out an action (sends email, write a warning etc.) Swatch can be considered simple to use, but unlike SEC, it cannot support more than one event each time.

Logsurfer is similar to Swatch, i.e. it follows rules stored to a configuration file to analyze log files, but has additional advantages. It is based on C language and processes events using regular expressions gathering all log-related data into one log file. Logsurfer uses contexts (similar to buffers) that store data instead of reading each line of a log file to check if each new message matches a regular expression [19]. Moreover, Logsurfer is based on dynamic rules, this means that actions that take place may provide additional actions.

OSSEC (initial name syscheckd) is based on an open source host-based intrusion detection system (HIDS) [20]. OSSEC can be used in Linux, Windows, Solaris and other operation systems. Its prime responsibility is to detect and analyze events in real-time, while monitoring for system integrity. OSSEC uses syslog, sms, or smtp to inform customers when data has changed or when suspicious behavior (e.g. cyber attack, system error) occurs at the time it happens. OSSEC uses custom technology to monitor data, and does not rely on an independent agent, service or daemon to monitor the data.

Simple Event Correlator (SEC) is a real-time correlation tool for high level security used to detect actions

based either on normal activity (e.g. the start of a process) or special events (e.g. system interrupts or failures). SEC is written in the open source programming language Perl and can run in different operating systems. It is simple to use and lightweight compared to other event monitoring tools, such as Logsurfer. It can run as a daemon managing many system events (tasks) at the same time.

In principle, SEC reads events that occur from files, pipes or `stdin` (standard input) and checks the file line by line for possible matches based on specified patterns rules, described as regular expressions, strings or Perl functions. If rules match, corresponding actions, e.g. sending an email or using snmp trap, are executed. If a number of specified rules match, then several simultaneous actions can be taken. Rules managed from SEC event correlation tool are as follows.

- Single Rules: an action is taken when the log file matches a specified rule.
- SingleWithScript: an external program is executed when the logfile matches a specified rule.
- SingleWithSupress: if rule and correlation action are present, then execute the action list immediately. Otherwise, SEC creates the action dynamically, and executes it in T seconds.
- Pair: this rule characterizes a pair of conditions in T seconds, where the T value defines a window field and the initial value is 0. If an event matches the pair of conditions based on pattern and context, then a message is printed defined by the desc field.
- PairWithWindow: this rule catches a pair of condition in T seconds, where the T value defines a window field. For example, if an event matches a rule characterized by pattern and context, then a message is printed by this field. If pattern and context do not match, SEC will check all fields (operations) in the given rule.
- SinglewithThreshold: start correlator operations that count the double lines of the same events N in T seconds. In this approach, If N events are present, we execute an action in N1 seconds
- Singlewith2Threshold: start correlator operations that count the double lines of the same events N in T seconds. In this approach, If N1 events are present, execute an action in N1 seconds. After that, if N2 events are present, execute an action in N2seconds.
- Suppress: make no action when a line of logfile matches to pattern. This rule keep events that match for later rules in the configuration file.
- Calendar : This rule is a little different for the other rules due to listen to system clock. In this approach, execute an action if present time matches of the specific time.
- Jump: jump a rule, since an event matches to rule.
- Options: set options when different lines of a log file match a pattern. An option that matches a configuration file may cancel a previous one.

4.4.3 Experimental Study: SEC for Monitoring a DDoS Attack via NoC Firewall

This section focuses on how to apply SEC to monitor and visualize how access patterns to BRAMS (1, 2, 3, or 4) dynamically change over time.

4.4.3.1 Access to Brams: FwAccess module

In order to plan accesses to Brams a second module (called `fwAccess`) is created that communicates with the initial module named "`sw_firewall`" . Two kernel threads in this module (*`access_thread`* and *`sample_thread`*) use I/O write to make accesses to BRAM 1, 2,3, or 4.

More specifically, as shown in Figure 32, `FwAccess` module provides the following API function calls to

setup the firewall and access FPGA memory via NoC Firewall, i.e. the kernel firewall module must have been installed first.

```
my_write(struct file *f, const char __user *buf, size_t len, loff_t *off);
my_read(struct file *f, char __user *buf, size_t len, loff_t *off);
static void setup_FW_Attack();
static int accessFun(void *arg);
static int sampleFun(void *arg);
```

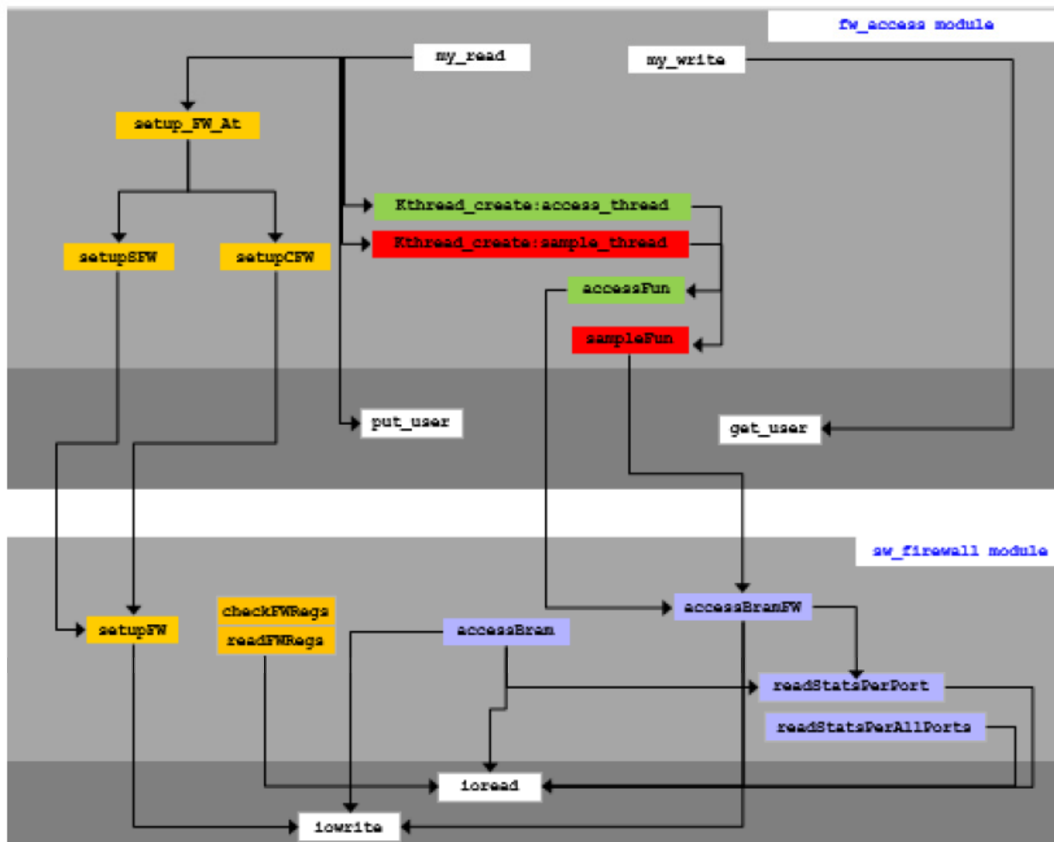

Figure 32: Communication of fw_access firewall module with NoC firewall module

`my_write` method: The `fw_access` module allows the user to program the mode of firewall access (`fw_type`) by sending an `echo` command to `my_write` method. Notice that the firewall setup mode (`fw_type`) is defined as either Simple or Extended mode. For example, if `fw_type = 0`, the firewall is set using Simple Mode, i.e. using `setupSFW`, while if `fw_type = 1`, the firewall is set in Extended Mode using `setupCFW`. The NoC firewall setup itself is performed in `setup_FW_Atttack` method. A code snippet is shown below.

```
for (i = 0; i < len && i < BUF_LEN; i++)
            get_user(Message[i], buf + i);
      Message_Ptr = Message;
      printk(KERN_INFO "Incoming message: %s\n", Message_Ptr);
      //passes one message when have write to kernel
        if (sscanf(Message, "%d", &fw_type) != 1) {
              printk (KERN_INFO KERN_ALERT "Invalid input: \"%s\".\n", Message);
              return -EINVAL;
```

```
    }
  else
        printk (KERN_INFO "fw_type:%d \n", fw_type);
```

`my_read` method: The `fw_access` module supports serial access to BPAMs and takes timing samples from kernel threads `access_thread` and `sample_thread`. These threads are created upon a call to `my_read` method, where the access thread (`access_thread`) calls `accessFun`, while the sampling thread (`sample_thread`) calls `sampleFun` function.

```
// create kernel thread for accessing
printk(KERN_INFO "my_read: create threads \n");
access_thread = kthread_create(accessFun, NULL, "accessthread");
// create kernel thread for sampling
sample_thread = kthread_create(sampleFun, NULL, "samplethread");
```

`accessFun` method: Thread (`access_thread`) calls its associated function (`accessFun`) to perform a number of serial accesses (e.g. 10.000) to BRAM 1, 2, 3, and 4 via the NoC Firewall as follows.

```
while (i<=NO_RUNS_WRITE) { //NO_RUNS_WRITE = 10.000
      accessing_completed = 0;
      if(bram_num>4){
            bram_num = 1;
      }
      my_access_ds.output_port=bram_num;
      accessed_bram_num=bram_num;
      accessBramFW(&my_access_ds);
      printk(KERN_INFO "accessFun:  id:%d ended Bram:%d \n", i, bram_num);
      accessed_bram_num=0;
      bram_num++;
      i++;
}
printk(KERN_INFO "accessFinished: %d", i);
accessing_completed = 1;
do_exit(0);
return 0;
```

`sampleFun` method: Thread (`sample_thread`) calls its associated function (`sampleFun`) to sample access to BRAMS, taking samples approximately each 100ms. Related code from `sampleFun` follows.

```
unsigned int accessing_completed = 0;
start = ktime_get();
while (!accessing_completed) {
      point_end = ktime_get();
      //find total time passed for all points in ms
      point_time = ktime_to_ms(ktime_sub(point_end, start));
      printk(KERN_INFO "sampleFun point_time_ms:%llu accessed_bram_num:%d\n",
                                          point_time, accessed_bram_num);
      msleep(SAMPLE_RATE); //SAMPLE RATE=100
```

}

4.4.3.2 Visualization and Alert using SEC

The Security Event Correlation Tool can be used to monitor BRAM 1 to 4 by examining the syslog file for the following pattern:

```
Apr 21 08:31:52 linaro-ubuntu-desktop kernel: timeFun point_time_ms:46309
accessed_bram_num:1.
```

This message is saved in syslog file when `fw_access` and `sw_firewall` module run. More specifically, the following simple rule of type single searches for the kernel time (corresponding to a time interval) and corresponding access delay and prints results in a file `results_sec.txt` for visualization (`action= write results_sec.txt $1 $2`).

The SEC rule (`sec.rule.sh` file) is shown below:

```
type=Single
ptype=RegExp
pattern=sampleFun point_time_ms:(\d+) accessed_bram_num:([1-4])
desc=time point with time $1 and bram num $2
action=write results_sec.txt $1 $2
```

As shown below when SEC is running with the above rules, the `sec.rule.sh` script executes all associated events in a file for high-level security visualization, e.g. by GRACE or HeatMap.

```
Writing event '196789 40' to file 'results_sec.txt'
```

For example, HeatMap can be used to represent accesses to BRAMS 1 to 4 in real time. Figure 33 shows that 4 accesses happen in BRAM1 in a small time window around time 5s, while Figure 34 shows that 2 accesses happen in BRAM1 in a small time window around time 7s. When more accesses happen in a Bram, red color becomes more intense.

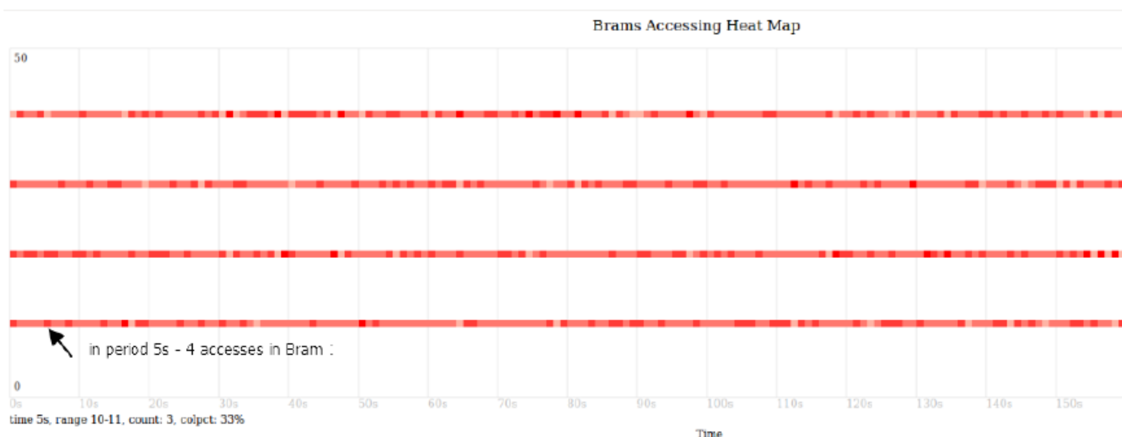

Figure 33: Brams Accessing heat Map – 4 accesses in Bram1

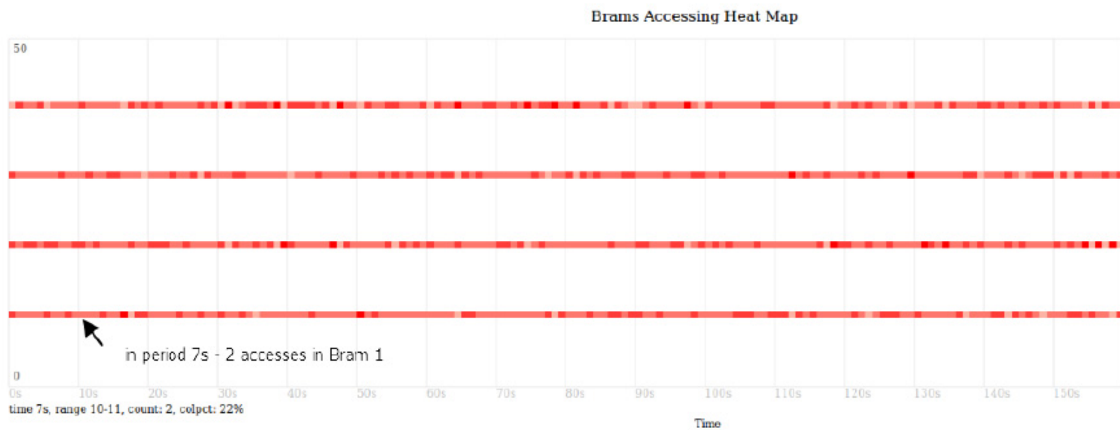

Figure 34: Brams Accessing Heat Map – 2 accesses in Bram1; color changes with access count

Finally, a second "complex rule" used in SEC records accesses to each BRAM per second. More specifically, the following rule of type SingleWithThreshold, place a threshold, i.e. if 10 accesses (`thresh=10`) happen in each BRAM in a time window of 25 seconds (`window = 25`), then a message appears according to field action (`action=write - BRAM $2 ACCESSED 10 TIMES`)

The SEC rule (`sec.rule.sh` file) is shown below:

```
type=SingleWithThreshold
ptype=RegExp
pattern=sampleFun point_time_ms:(\d+) accessed_bram_num:([1-4])
desc=accessed bram $2
action=write - BRAM $2 ACCESSED 10 TIMES
window=25
thresh=10
```

As shown below when SEC is running with the above single and complex rules, the `sec.rule.sh` script executes all associated events which may involve window popup, email alerts, or high-level security visualization.

```
Writing event '3739 30' to file 'results_sec.txt'
Writing event 'BRAM 3 ACCESSED 10 TIMES' to file '-'
BRAM 3 ACCESSED 10 TIMES
```

Chapter 5 - Conclusions and Future Work

In this work, we design a hierarchical driver based on a hardware NoC firewall with deny rules and show how to implement on-chip memory protection and anonymity services that target an in-hospital eHealth application. The firewall is attached into each port of a router, whereas firewall deny rules depend not only on the physical address, but also on the input and output ports of the router that the memory request from the processor is routed through.

In particular, our design methodology focuses on implementing and validating basic data protection primitives and complex hierarchical GNU/Linux services that support rule configuration and access control with statistical event logging on top of a hardware Network-on-Chip (NoC) Firewall mechanism embedded in an FPGA development board (ARMv7-based Zedboard). Our open source multi-layer design framework provides primitives that enable modularity and reuse across different use cases and interfaces to system tools, such as GRACE, HeatMap, and Secure Event Correlator (SEC) for high-level security visualization and notification services.

Focusing on a realistic out-of-hospital use-case that involves soft real-time ECG data processing on a Hospital Media Gateway server, we demonstrate how mid-level driver layer can be extended to implement high-level system security primitives for supporting a) *data privacy and anonymity,* and b) *access control of on–chip BRAM memory from internal denial-of-service attacks*. Experimental results on Zedboard reveal low overhead of our security primitives.

Our proposed solution is modular and reusable across other distributed eHealth applications and beyond. We currently enhance our security protocols with key management and event logging functions in order to demonstrate related security objectives in an EU project FP7-DREAMS demonstrator that considers soft real-time in-hospital ECG processing.

In addition, in a more rare scenario, we can thwart threats from an authenticated, but malicious (or corrupt) patient connecting to the server and impersonating another patient (e.g. by spoofing) by supporting a similar mechanism to the one presented in Section 4.2 for malicious doctors. More specifically, a `server_rx` child process will inherit group permissions of the connecting patient and safely transfer patient data via the NoC Firewall to an appropriate BRAM and a specific location (via AES-CCM hash).

References

[1] *Embedded System*, Wikipedia. (May 5, 2017) [Online]. Available: https://en.wikipedia.org/wiki/Embedded_system.

[2] «Attacks on Secure Embedded Systems” Protogenist Blog, 2012. (May 5, 2017) [Online]. Available: https://protogenist.wordpress.com/tag/software-attacks

[3] S. Ravi, A. Raghunathan, and S. Chakradhar, “Tamper resistance mechanisms for secure embedded systems”, in Proc. *17th Int. Conf. VLSI Design*, 2004, pp. 605—611.

[4] D. Basin, P. Schaller, and M. Schläpfer, *Applied information security: a hands-on approach*, Springer, Berlin Heidelberg, 2011.

[5] E. Kavakli, C. Kalloniatis, and S. Gritzalis, "Addressing privacy: matching user requirements with implementation techniques," *in Proc. 7th Hellenic European Research on Comp. Math & Appl. Conf.*, Athens, Greece, 2005, pp. 1—5.

[6] P. Mittal, M. Wright, and N. Borisov, "Pisces: Anonymous Communication Using Social Networks", Netw. and Distrib. System Security Symp., 2013.

[7] Whatisnetworking, Available: http://www.whatisnetworking.net/tag/advantages-and-disadvantages-of-osi-model

[8] K. Jeffay, DDoS history. (May 5, 2017) [Online]. Available: www.cs.unc.edu/~jeffay/courses/nidsS05/slides/3-DDoS-History-Defense.pdf

[9] J. D. Howard, "An analysis of security incidents on the Internet”, *PhD Thesis,* Dept. Engineering and Public Policy, CMU, Pittsburgh, Pennsylvania, 1997.

[10] MadiaBoy, Wikipedia. (28 April 2016) [Online]. Available: https://en.wikipedia.org/wiki/MafiaBoy

[11] Y. Zha και M. Chen, “DoS vs DDoS Attack”. (May 5, 2017). [Online]. Available: https://www.youtube.com/watch?v=c9EjuOQRUdg

[12] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms", *SIGCOMM Comput. Commun. Rev.*, **34 (2)**, 2004, pp. 39—53.

[13] J. Mölsä., "Mitigating denial of service attacks in computer networks”, *J. Comput. Secur.*, **13 (6)**, 2005, pp. 807—837.

[14] S. T. Zargar and e. al, "A survey of defense mechanisms against distributed denial of service flooding attacks”, *ACM SIGCOMM Comp. Comm. Review*, **15 (4)**, 2013, pp. 2046-2069.

[15] S. M. Specht., "Distributed denial of service: taxonomies of attacks, tools and countermeasures," *in Proc. Workshop Security in Parallel and Distrib. Syst.*, 2004.

[16] E. Alomari and et al, "Botnet-based distributed denial of service (attacks on Web servers: classification and art", *Int. J. Comp. Appl.*, **49 (7)**, 2012, pp. 24—32.

[17] SEC manual. (May 5, 2017) [Online]. https://simple-evcorr.github.io/man.html#lbAQ

[18] J.P. Rouillard, "Real-time log file analysis using the Simple Event Correlator (SEC)," in *Proc. 18th USENIX conference on Large Installation System Admin.*, 2004, pp. 133—150.

[19] Logsurfer. (May 5, 2017) [Online]. http://logsurfer.sourceforge.net

[20] Ossec. (May 5, 2017) [Online]. http://ossec.github.io

[21] Ubuntu Wiki. (May 5, 2017) [Online]. Available: https://help.ubuntu.com/community/Logwatch

[22] J. Finegan, “System log analysis and profiling system”, Open Channel Foundation, (May 5, 2017) [Online]. Available: http://www.openchannelfoundation.org/projects/SLAPS-2