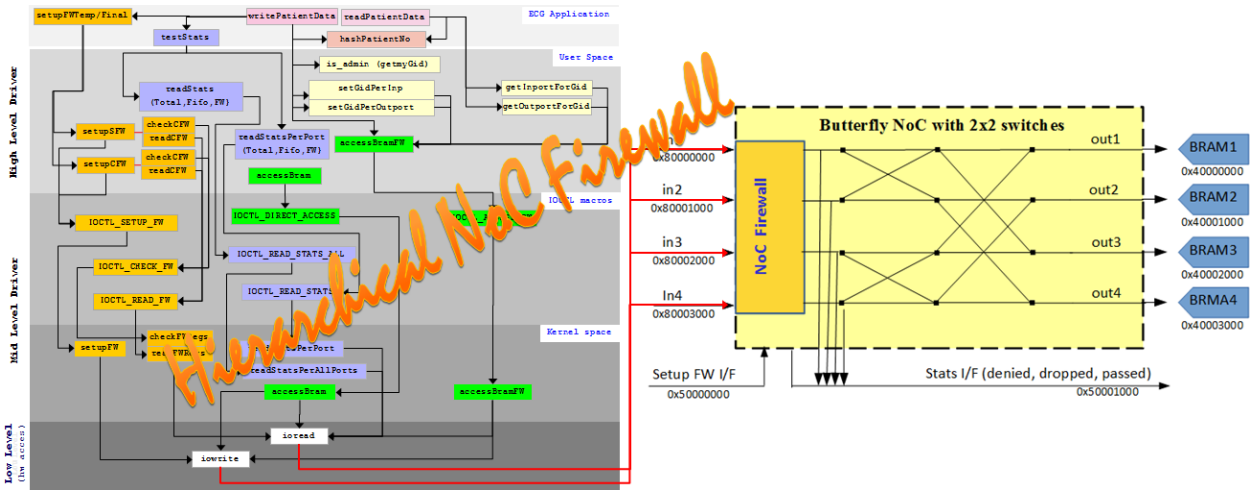
Hierarchical security primitives based on a Network-on-Chip Firewall: User and Reference Manual

Voula Piperaki and Angelos Mouzakitis
(Supervivor: Prof. Miltos D. Grammatikakis)
Technological Educational Institute of Crete
Heraklion 71410, Crete
GREECE
Email: {vpiperaki, angelos.mouzakitis, mdgramma}@cs.teicrete.gr
Team number: xohw18-187



Youtube: https://youtu.be/rYdyZacwYRw

GitHub:

Table of Contents

List of Figures

List of Tables

Abstract

We develop, implement and validate hierarchical GNU/Linux security primitives on top of a hardware Network-on-Chip (NoC) Firewall module embedded on the ARM-based Xilinx Zedboard FPGA. Our open source multi-layer design enables modularity and reuse across different use cases, and interfaces to system tools for high-level security visualization and notification services. Focusing, data privacy and anonymity in patient data for ECG application. We demonstrate how mid-level driver layer can be extended to implement high-level system security primitives for supporting data privacy and anonymity. Experimental results on Zedboard demonstrate low overhead of our security primitives.

1. Introduction

The issue of data privacy and security is of paramount importance in eHealth, since security risks increase when patient data is being disclosed across 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 ensure that a) each doctor can access only patient data that he has permission to, and b) patient identity (including pulse sensor MAC address) is hidden by using patient No (and possibly a key) for validation. Thus, in this contribution, 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 corrupt logical processes cannot read or write patient data in the eHealth system.

More specifically, we develop a hierarchical GNU/Linux driver infrastructure on top of a hardware NoC firewall architecture supporting memory access protection. The NoC Firewall is configured with smaller 2x2 internal switching nodes using input-output buffering and each of its output ports is attached to a BRAM via an AMBA AXI4 interface. Although a debug AMBA AXI4 interface provides direct access to the BRAMs.

The hierarchical Linux driver provides a highly reusable framework which reduces the code base and improves modularity, portability, maintenance and reusability and can be used to support data privacy and security in healthcare technologies and beyond, as our case study will prove.

Currently, most drivers in embedded operating systems (Linux and RTOS) follow a monolithic design approach, with very few supporting such a well-documented, layered approach. By examining the communication patterns between the driver, the kernel and the device, it becomes possible to better optimize for performance, power-efficiency, reliability and fault tolerance. A hierarchical driver implementation provides many benefits. Using multiple layers. This makes it possible to support a wider variety of hardware without having to rewrite large amounts of code. It also promotes flexibility by allowing the same protocol driver to plug into different hardware drivers at runtime.

Next, in Section 2, we examine hardware design of the NoC Firewall on Zedboard FPGA. In particular, Section 2.1 details a) NoC Firewall setup for simple and extended operating mode, including firewall registers, b) read/write access via NoC firewall, and c) firewall statistics for each port. In Section 2.2, we

examine hardware synthesis results on Zedboard FPGA (Vivado HLS and Vivado). In Section 3, we show the hierarchical Linux driver of NoC firewall driver. More specifically, we consider hierarchical design in Section 3.1, Low-level API in Section 3.2, and Mid-level API in Section 3.3. In addition, Section 4 serves as a reference manual of the High–Level Linux API of the hierarchical Linux Driver which aims to support modularity and reusability, Section 4.1, includes definitions for all user-level functions. In Section 5, validation and a healthcare application is presented. Hierarchical driver validation is studied in Section 5.1, ECG application is described in section 5.2 and consists of two parts: a) Section 5.3 which concerns system administrator setup, and b) Section 5.4 which relates to the clinic application.

2. Hardware Design of NoC Firewall on Zedboard FPGA

In this section, we show how a Network-on-Chip (NoC) Firewall hardware module embedded on Zedboard Z7020 FPGA can help protect data privacy. Our low-level firewall primitives, developed on Zedboard Z7020 FPGA, can be deployed via a *hierarchical Linux kernel module that supports setup of firewall rules and BRAM access* by sending/receiving information to/from kernel space using specialized IOCTL commands.

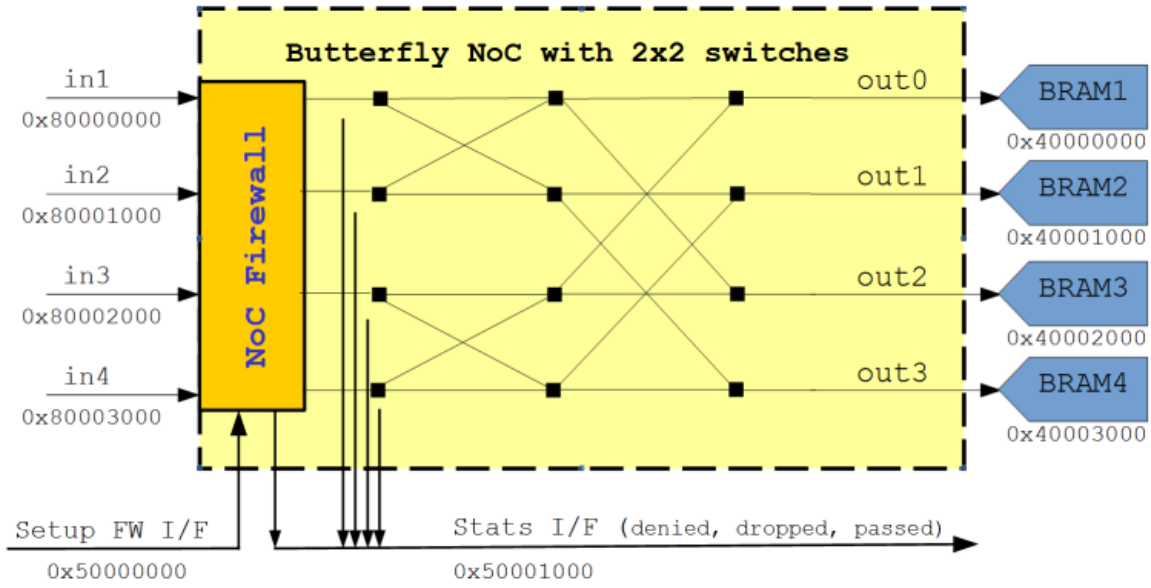


Figure 1: NoC Firewall

As shown in *Figure 1*, 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, we mainly focus on normal operation, i.e., setup of the firewall rules and normal access requests that travel through the firewall.

2.1 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, only the final destination matters, not the source node or path. In fact, for the Butterfly network topology, the source uniquely determines the path.

2.1.1 NoC Firewall Setup - Simple Operating Mode

In Simple operating 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 2*. This is an **absolute address** in BRAM1, BRAM2, BRAM3, and BRAM4 address region **[0x00000000, 0xFFFFFFFF].**

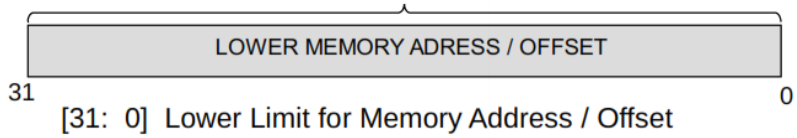


Figure 2: Setup low address range

- 32-bit high address register (`High`) specifying the end point of an address range, see Figure *3*. This is an **absolute address** in the corresponding BRAM1, BRAM2, BRAM3, and BRAM4 address region **[0x00000000, 0xFFFFFFFF].**

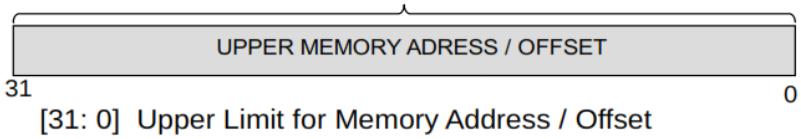


Figure 3: Setup high address range

- *32-bit rule register* (`firewall rules`) controlling protection of BRAM accesses to the corresponding `[Low, High]` address range. Moreover, two bits (`bit4` and `bit5`) correspond to four firewall operations: deny read, deny write, deny read & write, or accept all accesses **to the `[Low, High]` address range as shown** in *Figure 4*.

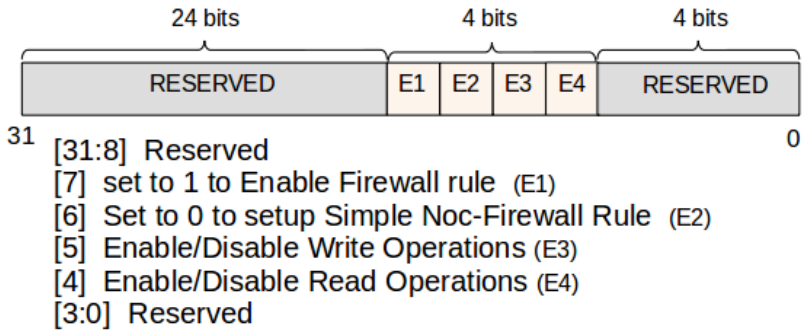


Figure 4: Setup rule address rage in Simple Mode.

Therefore, for Simple mode, as shown in *Figure 4* the 32-bit 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 of the operating mode (Simple or Extended). For example, a `bit7` field value of "1" specifies that the firewall is on, while "0" specifies that it 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 (e.g., `out1`, `out2`, `out3`, `out4`), while all pairs are assumed non-overlapping. This is asserted for each pair in the GNU/Linux driver.

2.1.2 NoC Firewall Setup - Extended Operating Mode

In *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 7*. 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 5*. This is a **relative address** in BRAM1, BRAM2, BRAM3, and BRAM4 address region, i.e., **[0x0000, 0x0FFF].**

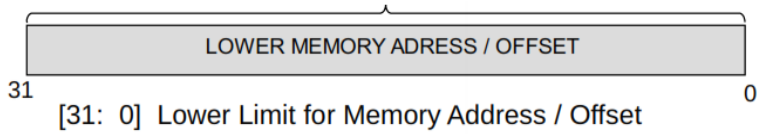


Figure 5: Low address range register (Setup)

- 32-bit high address register (`High`) specifying the end point of an address range, see *Figure 6.* This is a **relative address** in BRAM1, BRAM2, BRAM3, and BRAM4 address region, i.e., **[0x0000, 0x0FFF]**

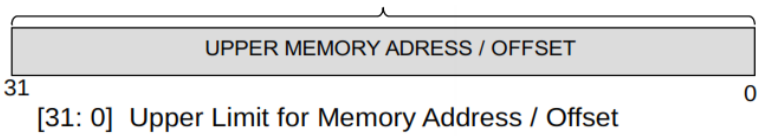


Figure 6: High address range register (setup)

- 32-bit rule register (firewall rules) protecting BRAM accesses to the corresponding `[Low, High]` address range. As shown in Figure 7, two bits (`bit4` and `bit5`) define four security patterns (deny read, deny write, deny read & write, or accept all) for access to the BRAM

selected by setting one of the bits in field E5; e.g., BRAM0 is selected by setting `bit3`.

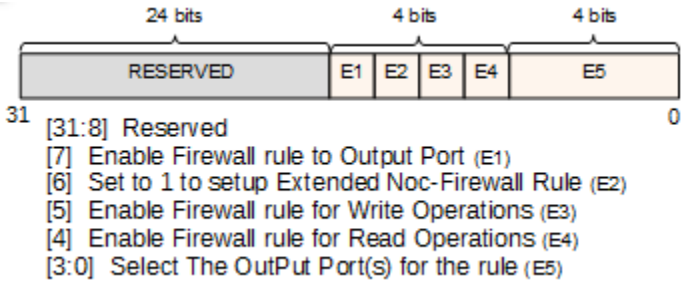


Figure 7: Rule address rage in Extended Mode register (setup)

Therefore, for Extended mode, as shown in Figure 7, the 32-bit setup rule register is split into 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 `bit6` 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 7*.

`bit3`: Value set to 1 to enable rule or set to 0 to disable for output port 1 (BRAM 1)
`bit2`: Value set to 1 to enable rule or set to 0 to disable for output port 2 (BRAM 2)
`bit1`: Value set to 1 to enable rule or set to 0 to disable for output port 3 (BRAM 3)
`bit0`: Value set to 1 to enable rule or set to 0 to disable for output port 4 (BRAM 4)

2.1.3 Access Request via NoC Firewall

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 providing a NoC output port. a NoC input port, and a BRAM offset relative to the specified BRAM base address. More specifically, the access request register is split into five parts, as shown in *Figure 8*.

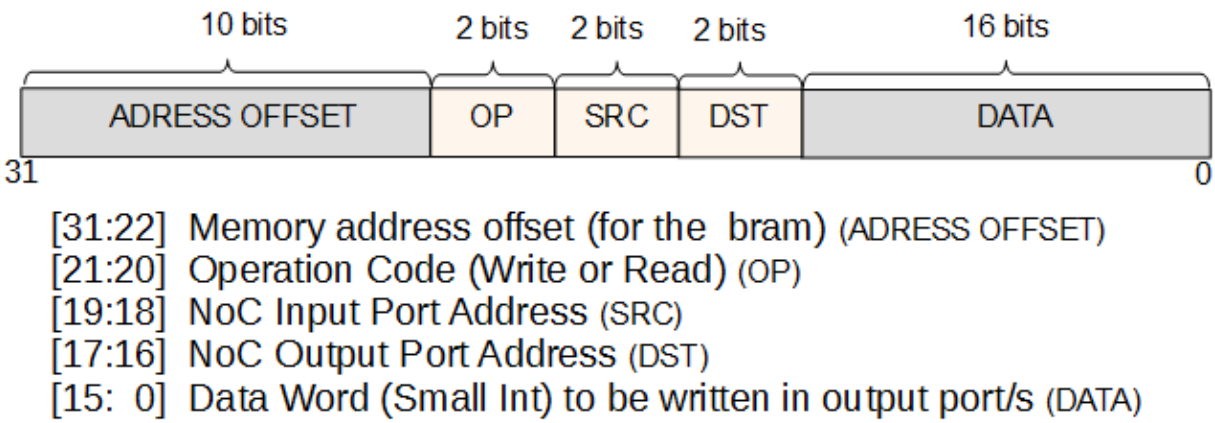


*Figure 8: Access request (in `Write_Packet_Register`) for Simple/Extended mode; SRC field used **only** in Extended Mode.*

- ✓ 10-bit [31:22] for address offset. This means that 10 bits can be used to write 16-bit BRAM data payload to a specific address, e.g., 0x00000000 in BRAM1.
- ✓ 2-bit [21:20] 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 [19:18] source port specifies the NoC source input port.
- ✓ 2-bit [17:16] destination port specifies the NoC output port.
- ✓ 16-bit [15:0] data.

For read operations, the 16-bit output is obtained from the 16 least significant bits of a 32-bit `Read_Packet_Register`.

2.1.4 Statistics Interface

Finally, a read-only interface for monitoring firewall statistics is provided using four sets of registers, one in each port at memory address 0x50001000 (see *Figure 1)*. Three registers (`startingStatisticsTotal`, `startingStatisticsFifo` and `startingStatisticsFw`) are defined as follows:

- 32-bit `startingStatisticsTotal` register (alias `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.

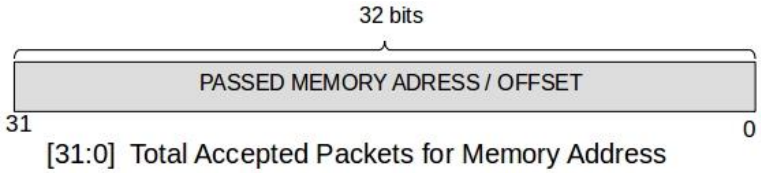


Figure 9: Register counting successful BRAM access requests via the NoC Firewall (for a given input port)

- 32-bit `startingStatisticsFifo` register (alias `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.

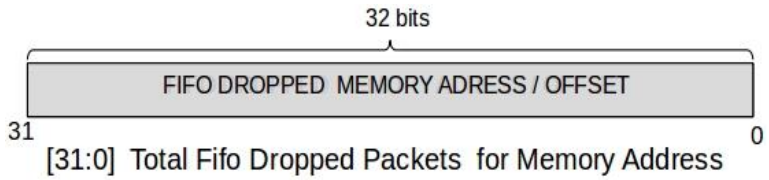


Figure 10: Register counting packets dropped from an Input Fifo (for a given input port)

- 32-bit firewall dropped register (alias `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.

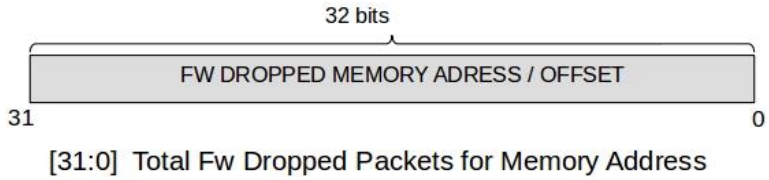


Figure 11: Register counting packets dropped from the NoC Firewall (for a given input port)

2.2 Results from High Level Synthesis (Vivado HLS and Vivado)

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 an termediate implementation 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. Recent Vivado HLS synthesis results show that increasing the data path (read/write payload) from 16bit to 128-bit and adding 16 bit CRC requires 19808 FF and 40287 LUTs. However, this design barely fits in Zedboard.

	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%)

Table 1: Hardware Costs In NoC & NoCFW Implementations

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.

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

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

In addition, *Table 2 ,* further compares the cost of NoCFW hardware module to AMBA AXI4, and a larger instance of a 12-node STMicroelectronics STNoC network-on-chip synthesized 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 8) 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.

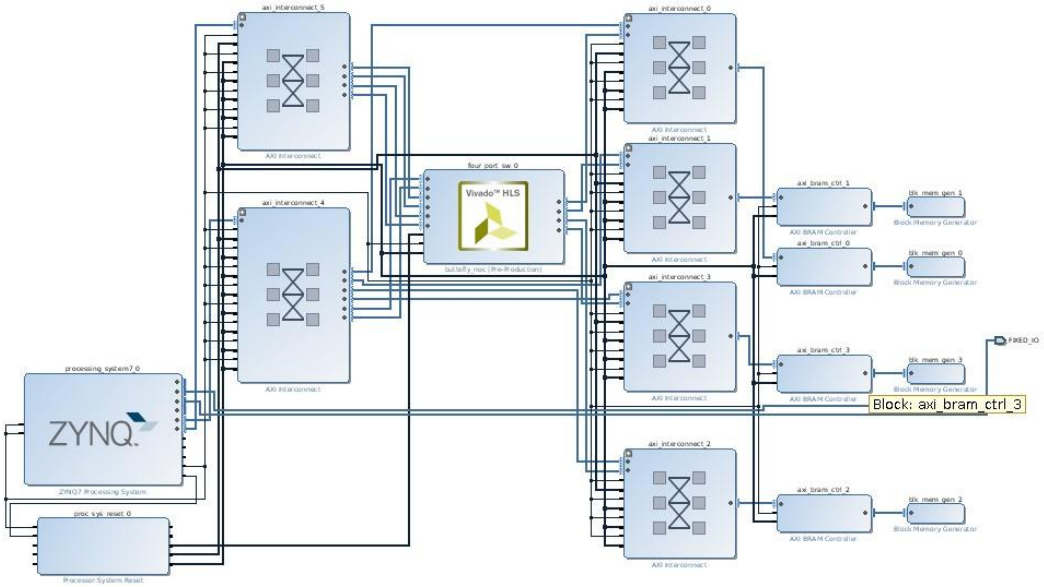


Figure 12: 4-Port Firewall Multicast Router – Synthesis on Zedboard

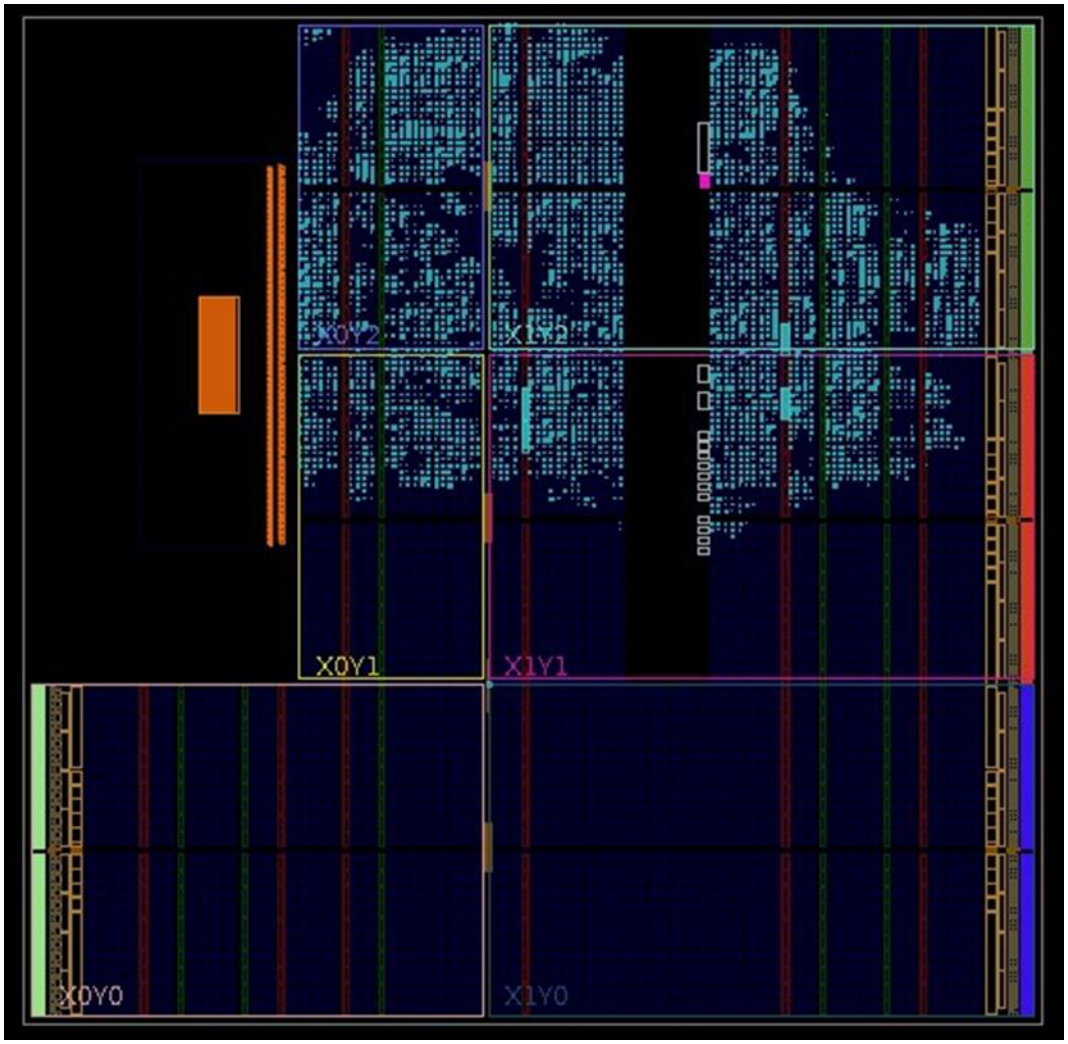


Figure 13: The NoC Firewall: a) circuit and b) utilization of the FPGA

Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε. 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 13*: The NoC Firewall: a) circuit and b) utilization of the FPGA**Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.** indicates the low utilization of the Zedboard FPGA by the NoCFW circuit (~20% used).

3. Hierarchical Software Driver in Linux for NoC Firewall

3.1 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 14,* 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.

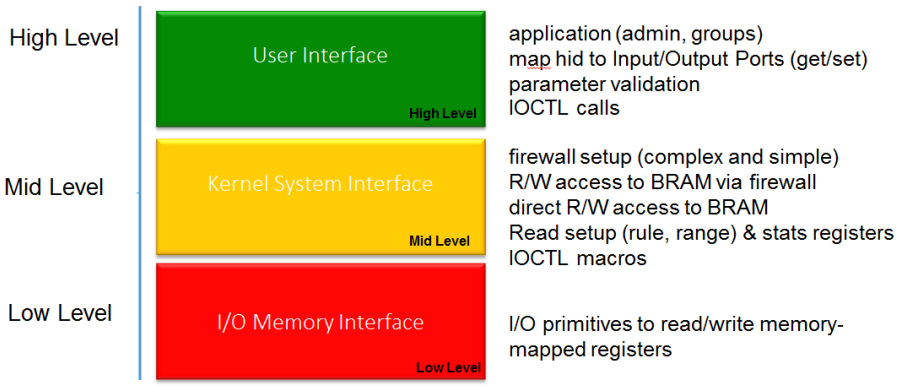


Figure 14: Hierarchical Linux Driver of NoC Firewall

3.2 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. Section 3.3. 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 normal access to BRAM via the firewall and direct access to BRAM for debug.

3.3 Mid-Level Driver API

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

3.3.1 Fundamental Data Structures and IOCTL Calls

We have implemented the MLD in two files. The first one (`ioctl_fw.h`) contains fundamental data structures for setting up the firewall, reading statistics, and accessing BRAM either via the NoCFW or directly via another AXI bus interface (for debug purposes).

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

```
/* Define simple commands with _IO.
   Generate a unique code (integer) for _IOR & _IOW, e.g 1, 2, 3 */
#define MYIOC_TYPE 'k'
```

```
#define MYIOC_WRITEONE    _IO(MYIOC_TYPE, 1)    // write command one
#define MYIOC_WRITETWO    _IO(MYIOC_TYPE, 2)    // write command two
#define MYIOC_WRITETHREE  _IO(MYIOC_TYPE, 3)    // write command three
#define MYIOC_READONE     _IO(MYIOC_TYPE, 1)    // read command one
#define MYIOC_READTWO     _IO(MYIOC_TYPE, 2)    // read command two
#define MYIOC_READTHREE   _IO(MYIOC_TYPE, 3)    // read command three
#define MYIOC_READFOUR    _IO(MYIOC_TYPE, 4)    // read command four
```

```
struct  fw_ds{ // firewall's setup structure
      unsigned int op_code;
      unsigned int inport;
      unsigned int outport;
      unsigned int Low;
      unsigned int High;
      unsigned int Rule;
};
struct stat_ds { // firewall's statistics structure
      unsigned int inport;
      unsigned int Total passed;
      unsigned int Fifo_dropped;
      unsigned int Fw_dropped;
};
struct  access_ds {  // firewall's access structure
      unsigned int op_code;
      unsigned int inport;
      unsigned int outport;
      unsigned int addr;
      unsigned int *data;
};
struct  dir_access_ds { //firewall's direct read/write access to BRAM
      unsigned int op_code;
      unsigned int bram_no;
      unsigned int addr;
      unsigned int *data;
};
```

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).
```

3.3.2 Kernel-Level Functions for NoC Firewall Operation

The second file in MLD (`kernel_mode.c`) provides supported kernel-level functions for firewall operation (MLD functions are also shown together with user-level functions in Figure 15):

```
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 *);
```

3.3.2.1 setupFW

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) );
```

3.3.2.2 readFWRegs and checkFWRegs

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)

3.3.2.3 readStatsPort and readStats

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 BRAM1, 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 to BRAM4).

```
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)

3.3.2.4 accessBram and accessBramFW

Finally, MLD supports both direct access (accessBram) and access via the firewall (accessBramFW). 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` (both read/write request 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_Response_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 *(described in Section 4)*, we call:

```
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 overflow.

4. Reusability: High-Level Linux Driver API

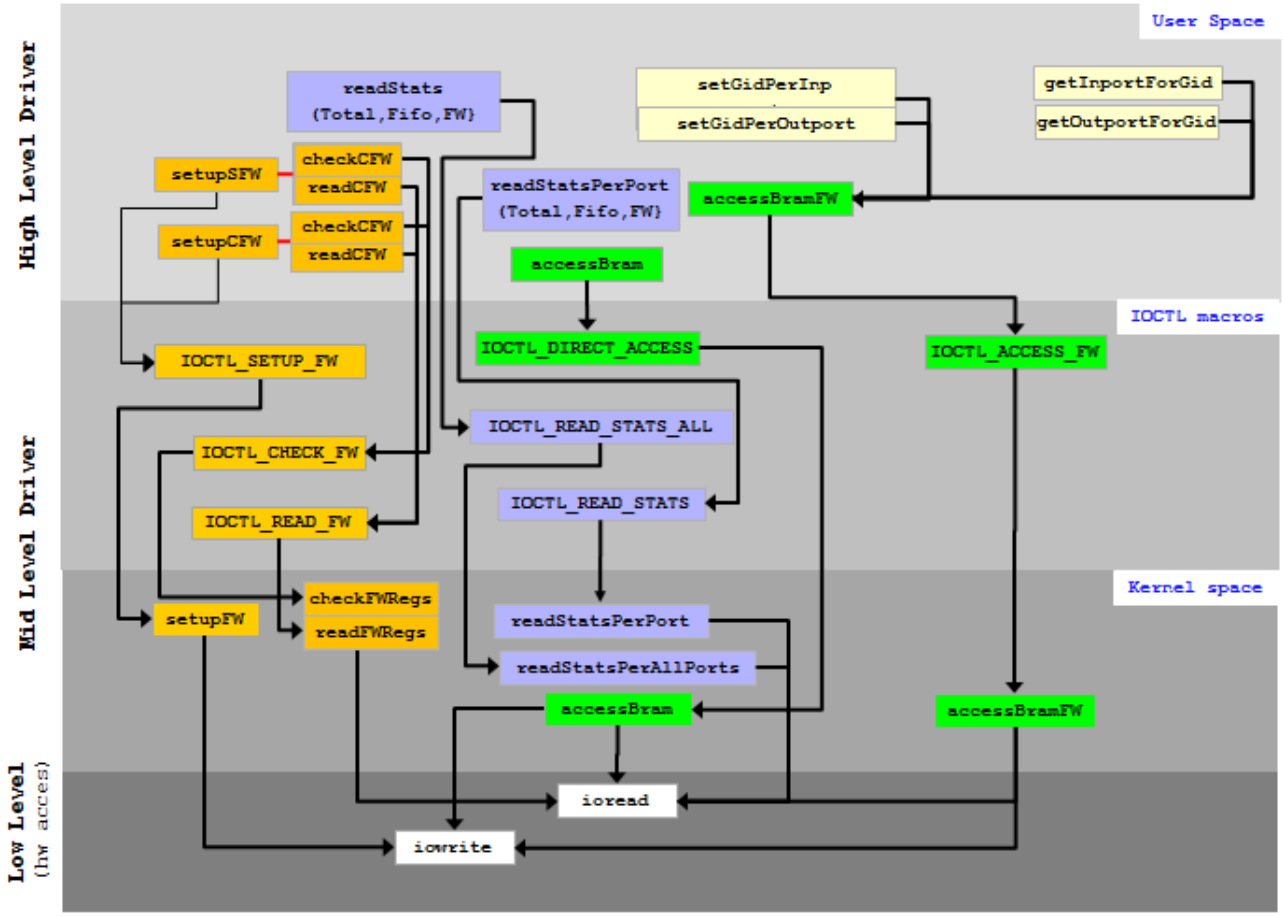


Figure 15: The Hierarchical Linux Driver for our Firewall

This section outlines the User Level API for developing applications. It serves as a Reference Manual for all NoC Firewall commands, e.g., setting the firewall and accessing BRAMS or statistics registers via the firewall.

4.1 User-Level Functions for NoC Firewall Operation

HLD supports specializations or extensions of the following high-level functions. These include separate routines `setupSFW`/`readSFW` (for Simple mode) and `setupCFW`/`readCFW` (for Extended mode), as well as *accessBram and accessBramFW*. Moreover, specializations of statistic functions `readStatsPerPort`/`readStatsPerAllPort` that focus on specific arguments, e.g., *readsStatsTotalPerPort*, *readsStatsFifoPerPort*, *readsStatsFWPerPort* or `readStatsTotal`, `readStatsFifo` and `readStatsFw`.

- The complete HLD API (user_mode.h) 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 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 readSFW(gid_t group_id); //read simple FW setup registers*
- *void readCFW(gid_t group_id); //read complex FW setup registers*
- *int accessBram(unsigned int op_code, unsigned int bram_no, unsigned int addr_reg, unsigned int *data); //* direct *access to BRAM*
- *int accessBramFW(unsigned int op_code, unsigned int inport, unsigned int outport, unsigned int addr_reg, unsigned int *data); //access BRAM via FW*
- *unsigned int readsStatsTotalPerPort(unsigned int port); // passed packets*
- *unsigned int readStatsFifoPerPort(unsigned int port); // dropped packets entry fifo)*
- *unsigned int 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*

4.1.1 setupSFW

The **setupSFW** function configures the firewall in Simple operating mode.

- ***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 functions uses the following six arguments:

- **inport** corresponds to the input port number (1 to 4);
- **L_addr_reg** (and **H_addr_reg**) define the lower (resp. upper) limit of the memory address range to be protected; for example, if input port is given as 1, the maximum address range is [0x40000000, 0x40000FFF], while if it is set to 4, the maximum address range is [0x40003000, 0x40003FFF].
- **rule** must be set to 1 to enable the firewall rule for each subsequent read/write access,
- **write_ops** (and **read_ops**) are set independently, either to value 1 to indicate a deny write (resp. deny read), or value 0 to define an accept write (resp. accept read) rule.

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*.

In addition, the internal `checkFWRegs` function uses IOCTL calls to validate if specific firewall configuration values have been written to kernel space.

4.1.2 setupCFW

The **setupCFW** performs firewall setup in extended mode.

- **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);

This method is similar to **setupSFW**, but provides an extra argument.

- **outport** corresponds to the output port number (1 to 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`.

After parameter range checking, data must be packetized into kernel-level data structures (described in MLD) so that IOCTL calls are invoked to communicate with the corresponding MLD `setupFW` function.

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

4.1.3 readSFW and readCFW

The **readSFW** and **readCFW** methods provide a way to read the firewall setup registers in Simple or Extended mode.

- ***void readSFW(gid_t group_id);***
- ***void readCFW(gid_t group_id);***

Both functions have one argument:

- **gid** corresponds to group ids

This function is allowed only for system administrator group_id (e.g gid 0). IOCTL calls are invoked to allow user space to communicate with the corresponding MLD `readFW` function.

4.1.4 accessBRAM and accessBRAMFW

Finally, HLD supports both direct access via **accessBram** and access via the firewall **accessBramFW**. The **accessBram** function performs direct access (read/write) without passing via the firewall (backdoor).

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

This is accomplished by four arguments:

- **op_code** sets to 0x0 or 0x1 for write or read, resp.
- **bram_no** corresponds to bram number 1 to 4 (or outport).
- **addr_reg** sets the address range for output port
- ***data** refers to data that will be read or write.

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

Similarly, the **accessBramFW** method performs access (read/write) via the NoC firewall.

- **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 argument regarding input ports:

- **inport corresponds to the input port number (1 to 4);**

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`).

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 both functions parameter range checking, data is packetized into the kernel-level data structures and IOCTL calls are invoked to allow communication with corresponding MLD `accessBram` function.

4.1.5 setGidPerInport and setGidPerOutport

- **`void setGidPerInport(gid_t gid, unsigned int inport);`**
- **`void setGidPerOutport(gid_t gid, unsigned int outport);`**

These functions are used to map NoC input and output ports (ports 1, 2, and 3, since port 4 is used by the administrator) to Linux group ids. The functions use two arguments:

- `group` corresponds to a Linux group (e.g., `fwgroup`, `fwgroup1`, `fgroup2`, also see Section 5)
- `inport/outport` corresponds to the input/outport port number (1 to 3), specifying when both functions are used a given BRAM.

Both set functions must be called by the system administrator. Information is written in BRAM4 via input and output port 4.

4.1.6 getGidPerInport and getGidPerOutport

- **`int getInportPerGid(gid_t gid);`**
- **`int getOutportPerGid(gid_t gid);`**

These functions are return the ports to access the corresponding BRAMs during normal operation by setting one argument:

- `gid` defines the id og groups

The *`get`* functions use `accessBramFW` to read (opcode *`0x1`*) group names that have been previously mapped. These functions can be called from administrator and also from simple users to find the input and output port that have been setup from administrator for access to the BRAM dedicated for each Linux group.

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

5. Validation and Healthcare Application

5.1 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 (*test_setupsfw_READ.c , test_setupsfw_WRITE.c, test_setupcfw_READ.c , test_setupcfw_WRITE.c files*). 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*
   - `x` and `z` are equal, and the setup pattern for that port is deny read (accept write) or accept all, or
   - `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*
   - `x` and `z` are equal, and the setup pattern for that port is deny read (accept write) or accept all, or
   - `x` and `z` are non-equal, and setup pattern for that port is deny write (accept read) or deny all.

5.2 ECG Application

In addition, we have developed a healthcare demonstrator based on the NoC Firewall. In this context, three users are created (root plays the role of system administrator). More specifically, we define three clinicicans (doctors) implemented as `clinic`, `clinic1`, and `clinic2` who belong to three different Linux groups (fwgroup, fwgroup1, fwgroup2), essentially representing hospital departments.

Table 3: Linux groups and users (clinicians) in the healthcare scenario

Group	Users
root	system administrator
fwgroup	clinic
fwgroup1	clinic1
fwgroup2	clinic2

5.3 ECG Application - Part I: System Administrator Setup

Our scenario exploits **design and reuse principles of the high-level NoC Firewall driver**. It allows only a privileged user (i.e., system administrator code: `systemadmin_setup.c`) to write tables that uniquely associate clinician Linux groups to input/output ports (***Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.***). Then, the administrator writes patient data to BRAMs 1 to 3, using a hashing scheme based on Patient ID. This allows registered clinicians to retrieve patient name, history and a **key (MAC) for accessing patient data, e.g., an electrocardiogram (ECG) data file**.

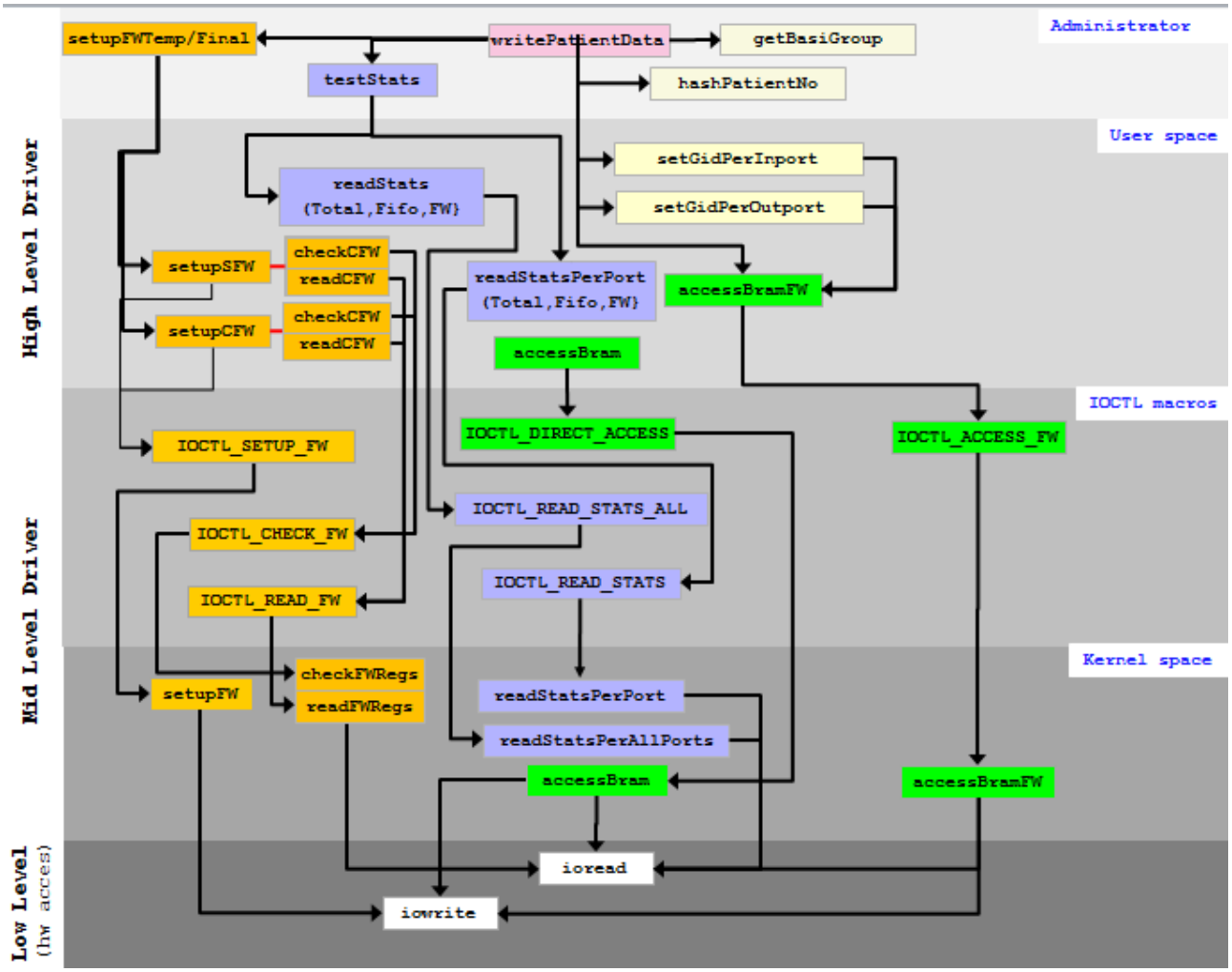


Figure 16 : The NoCFW Linux driver hierarchy with full system administrator privileges.

More specifically, the administrator setupuses the following functions.

5.3.1 setupFW_Temp

The **`setupFW_Temp`** sets up temporarily firewall rules for `clinic`, `clinic1`, `clinic2`. Specifically, this functions calls `setupCFW` and `setupSFW` to allow read and write to all input and output port only by the administrator. For example:

```
//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);
```

5.3.2 setupFW_Final

The setupFW_Final sets up the final firewall rules for clinic, clinic1, clinic2. Specifically, it defines deny writes for all ports, but accepts reads to a specific input and output ports 1-3 depending on the clinician.
For example:

```
// clinic/clinic1/clinic2 (mapped to inport 1/2/3) CAN respectively access BRAM1/2/3 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);
```

5.3.3 getBasicGroup

`gid_t getBasicGroup();`

This method first certifies the associated (current) group id against root, by first calling `groupIdFromName()` to obtain the specific group id for root and then calling `grouplist(list_of_groups, &ngroups)` to validate the current Linux group id, 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); //find in which groups each user belongs
for (i=0; i<ngroups; i++)
      if (list_of_groups[i] == gid_fwgroup) {
         myGid = gid_fwgroup;
            break;
   }
   return myGid;
}
```

5.3.4 testStats

`void testStats();`

This functions calls the NoC Firewall statistics functions for validation.

```
readStatsTotalPerPort(inport);
readStatsFifoPerPort(inport);
readStatsFwPerPort(inport);
```

5.3.5 hashPatientNo

`int hashPatientNo(unsigned int patient_no);`

This function uses a keccak SHA-3 to hash a **`PatientNo`** to a patient location in BRAM.

5.3.6 writePatientData Function

```
void writePatientData(unsigned int patient_clinic, unsigned int patient_no,
                      char *mac_address, char *patient_name);
```

Calls the following four arguments:

- ***`patient_clinic`*** sets the input and output ports,
- ***`patient_no`*** sets a patient number (i.e., 15, 16, 25, 35)

- ***mac_address*** sets a mac address (i.e., E4:EE:FD:00:0E:68)
- ***patient_name*** defines a patient_name (i.e., patient_A_clinic)

This function is called from system admin to a) define firewall rules temporatrily and b) write patient data (i.e., patient number, mac address (or key) and patient name) per NoC output port (equvalently BRAM number). We assume clinic/clinic1/clinic2 (mapped previously to inports 1/2/3) access patient data from BRAM1/2/3 .Patient data location in BRAM based on hashing patient no. An example of code described below:

```
// 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);
...
// Write patient data to Bram via  accessBramFw )16-bit at a time)
// 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);
      accessBramFW(0x0, inport, bram_no, addr_reg, &iMac[i]);
// check if MAC address has been written to BRAM by calling from
// user space the HLD direct access function accessBram (op code 0x1)).
...
// write patient number in BRAM via  accessBramFw
for(i=0; i<(name_size + 1); i++){
...
accessBramFW(0x0, inport, bram_no, addr_reg, &iName[i]);
```

5.3.7 How System Admin Setup works?

The following steps explain the sequence that functions in the application scenario.

Step 1: Initially, **`getBasicGroup`** Function is called. If system administrator is verified, it can call the following functions.

Step 2: **`setGidPerNoCInport`** and **`setGidPerNoCOutput`** functions are called to map Linux groups to input and output ports.

Step 3: Firewall is temporarily setup via **`setupFWTemp`** to allow the system administrator to write patient information o BRAM.

Step 4: Then, system administrator uses **`writePatientData`** function to write patient data (i.e., patient**`_clinic[i], patient_no[i], patient_mac[i], patient_name[i]`**) to an appropriate BRAM (defined in Step 2). In the demonstrator example, we use the following data.

```
//group names
   char group_name[][20] = { "fwgroup", "fwgroup", "fwgroup1", "fwgroup2"};
// note: take inport = outport for patient setup
   char patient_clinic[20] = { 1, 1, 2, 3 }
//pairs of patient numbers, names & MAC (keys)
   int patient_no[20] = { 15, 16, 25, 35 };
   char patient_mac[][20] = { "E4:EE:FD:00:0E:68", "E4:EE:FD:00:07:17",
   "E4:EE:FD:00:0E:68", "E4:EE:FD:00:07:17"};
   char patient_name[][20] = { "patient_A_clinic",    "patient_B_clinic",
   "patient_A_clinic1", "Patient_B_clinic2" },
```

Step5: Later after group tables and patient data are written, the firewall setup is modified to limit user access to these tables by calling setupFWFinal.

5.4 ECG Application – Part II: Clinic Applications

This scenario provides read access to all BRAMs (1, 2 and 3) respectively from groups (`fwgroup`, `fwgroup1` and `fwgroup2`) as set up by system administrator (`systemadmin_privacy.c`). There are 3 files: `fw_clinic_access.c` for `fwgroup`, `fw_clinic1_access.c` for `fwgroup1`, and finally `fw_clinic2_access.c` for `fwgroup2`. Each Linux group consists of physicians is given permission (by our NoC Firewall) to only read data from its corresponding BRAM. The maximum value of the number of physicians on Zedboard FPGA is limited by BRAM size; for our current NoC Firewall configuration it is section 5.1.

Clinic access to data is made via `readPatientData` (which sets the ports properly and calls `accessBramFW` function to read only) as shown in *Figure 17.*

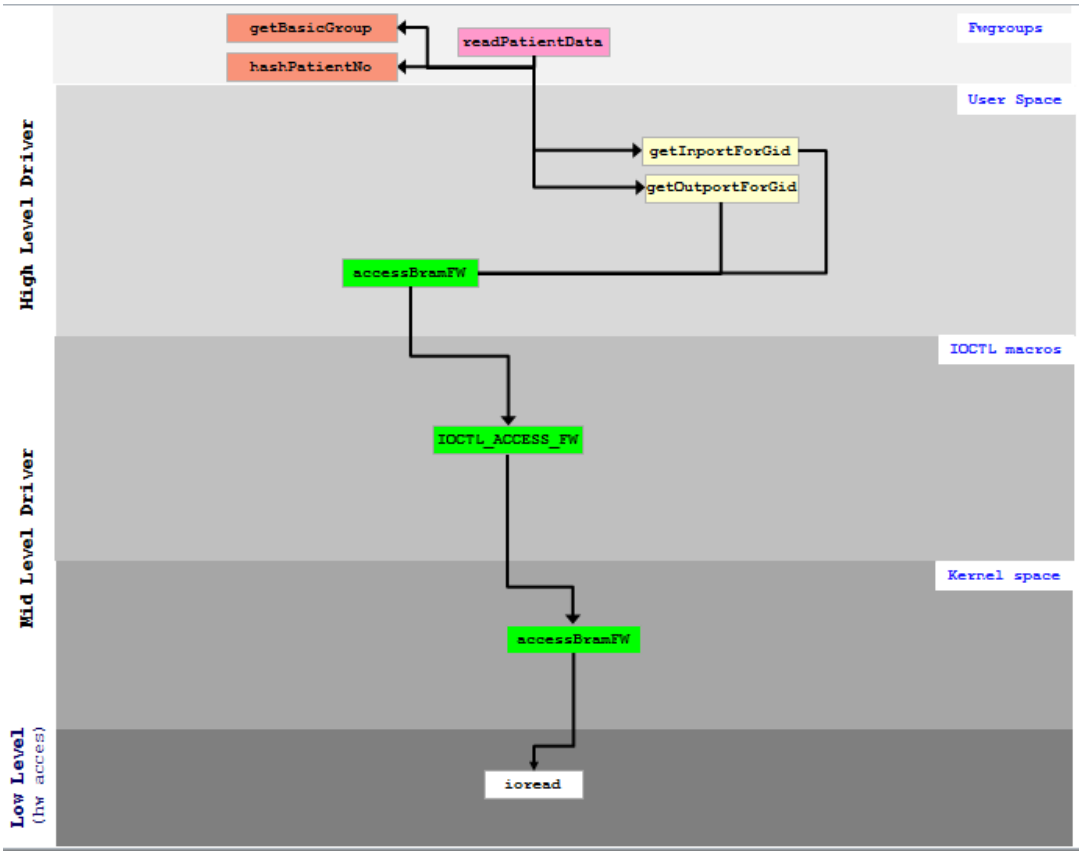


Figure 17: The NoCFW Linux driver hierarchy with basic group (fwgroups) privileges.

5.4.1 readPatientData

- **`void readPatientData (unsigned int patient_no, char **mac_address, char ** patient_name)`**

The ***`readPatientData`*** function read patient data and consists of the following arguments:

* ***`patient_no`*** sets a patient number (i.e., 15, 16, 25, 35)

* **mac_address** sets a mac address (i.e., E4:EE:FD:00:0E:68)
* **patient_name** defines a patient_name (i.e., patient_A_clinic)

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");
// access BRAM data 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** 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.

5.4.2 Results from Healthcare Application - Security Overheads

Besides our results from hardware design in Section 2.2, we have also computer the security overhead of our NoC Firewall with the healthcare application.

More specifically, the software administrator cost for involving the Linux kernel-space driver to setup anonymity service relates to

- mapping Linux 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 costs on limited devices, 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 pulse sensor devices without missing deadlines, it is obvious that data protection costs are relatively small [2, 3].

6. Conclusions & 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 high-level security visualization and notification services.

The hierarchical Linux driver of the NoC Firewall can be applied to a real-time telemedicine application. In this scenario, we can consider *soft real-time monitoring of patients inside and ouside the hospital environment* with the aim to evaluate performance overheads for supporting transmission security and anonymity. In this case, the 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*. In this respect, we are exploring how the use of multipath NoC routing can enhance Linux system security.

References

[1] V. Piperaki, Security primitives in a hierarchical GNU/Linux driver of a hardware NoC firewall, Msc Thesis, Dept. Informatics Engineering, TEI of Crete, July 2017.

[2] G. Tsamis, M.D. Grammatikakis, A. Papagrigoriou, P. Petrakis, and V. Piperaki. A. Mouzakitis and M. Coppola, "Soft Real-Time Smartphone ECG Processing", *in Proc. 12th IEEE International Symposium on Industrial Embedded Systems*, 14 - 16 June 2017, Toulouse, France

[3] G. Tsamis, M.D. Grammatikakis, A. Papagrigoriou, P. Petrakis, and V. Piperaki, "Evaluation of Healthcare Demonstrator", *Distributed Real-Time Architecture for Mixed-Criticality Systems*, Eds. H. Abhamadian, R. Obermaisser, J. Perez, CRC Press, 2019, to appear.