

Received May 22, 2020, accepted June 3, 2020, date of publication June 5, 2020, date of current version June 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3000457

# A Systemic and Secure SDN Framework for NoC-Based Many-Cores

MARCELO RUARO<sup>1</sup>, LUCIANO L. CAIMI<sup>2</sup>, (Member, IEEE),  
AND FERNANDO GEHM MORAES<sup>1</sup>, (Senior Member, IEEE)

<sup>1</sup>School of Technology, PUCRS, Porto Alegre 90619-900, Brazil

<sup>2</sup>Department of Computer Science, Federal University of Fronteira Sul (UFFS), Chapecó 89802-112, Brazil

Corresponding author: Fernando Gehm Moraes (fernando.moraes@pucrs.br)

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. The work of Fernando Gehm Moraes was supported in part by the Brazilian Funding Agency Fundação de Amparo a Pesquisa do Estado do Rio Grande do Sul (FAPERGS) under Grant 17/2551-0001196-1, and in part by the Brazilian Funding Agency Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under Grant 302531/2016-5.


**ABSTRACT** Recent exploration of Software-Defined Networking (SDN) for Many-Core Systems-on-Chip (MCSocS) showed higher management flexibility and reduced physical complexity compared to other runtime communication management. In SDN, there is a software SDN Controller (control layer) that configures generic routers (data layer). The adoption of SDN makes the path establishment programmable and straightforward, according to different network policies, such as low power, QoS, fault-tolerance. It is also possible to change the path establishment policies at runtime without the need to redesign the NoC. Current works focus on proposing SDN architectures, lacking a systemic framework that describes the steps required to implement SDN into a Many-core environment. Security is an observed gap in SDN designs. A malicious task could configure SDN routers and take control of the NoC. The contribution of this work is to present a systemic and secure SDN framework (SDN-SS), detailing the steps required to support SDN in MCSocS. This work also describes the iteration between the hardware, operating system, and user's tasks. The SDN-SS manages a Multiple-Physical NoC, with one packet-switching subnet and a set of circuit-switching subnets. The originality of SDN-SS includes (i) a step-by-step framework description addressing the phases required to support a secure SDN management; (ii) a secure SDN router configuration protocol; (iii) a protocol to change the subnet at runtime. Experimental results show the framework's capability to avoid DoS and Spoofing attacks while presents a low SDN router configuration overhead, corresponding up to 2% of a related work delay and a small impact over the user's task communication.

**INDEX TERMS** Software-defined networking (SDN), many-core, MPSoc, many-core system-on-chip (MCSoc), network-on-chip (NoC).

## I. INTRODUCTION

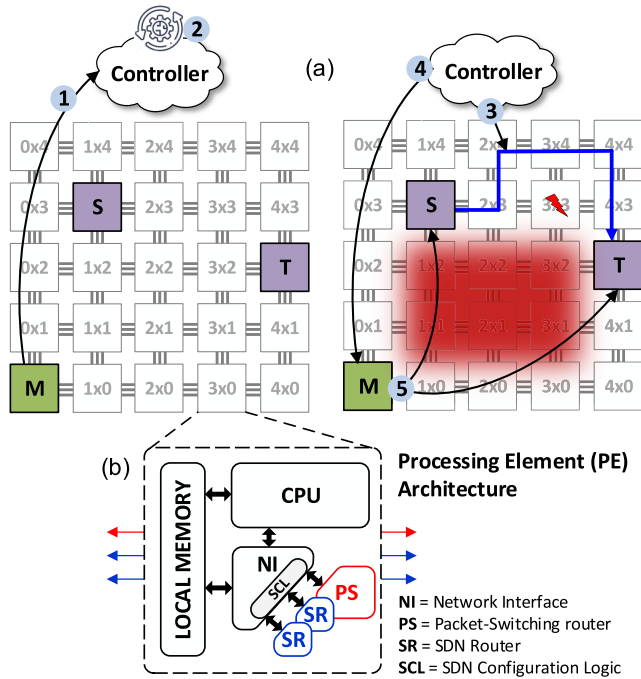
The Software-Defined Networking (SDN) paradigm is an emerging communication management technique adopted in computer networks [1]. The key benefit of SDN is to unify and simplify the network devices' management of different vendors by removing the control logic from the devices (at hardware) to an SDN controller (or simply *Controller*).

Recent researches have explored the pros and cons of using the SDN paradigm in the integrated circuit (IC) design, specifically for the management of the Many-core communication on Networks-on-Chip (NoC) [2]–[6]. The main benefits for SDN on NoCs are the higher flexibility for runtime

The associate editor coordinating the review of this manuscript and approving it for publication was Ting Wang .

and self-adaptive network management and reduced hardware complexity. The reason for explaining such benefits is that the NoC router is no longer overloaded with specific designs to support different features, like QoS, fault-tolerance, and power management. The router becomes a simple and configurable unity able to redirect NoC packets according to the Controller rules. Due to its global knowledge of NoC's resources, the Controller may adopt policies to mitigate faults, to balance the communication load, to manage the NoC power consumption [7], and to provide QoS for real-time flows [3], [4]. Moreover, all such features can be combined into the Controller, resulting in multi-objective management.

Figure 1(a) shows the steps to establish a path in a Many-Core System-on-Chip (MCSoc) that adopts the SDN



**FIGURE 1.** (a) SDN-based communication in an MCSoC. Step 1: path request; step 2: path search; step 3: SDN routers' configuration; step 4: requisition reply; step 5: configuration for the user's tasks to use the SDN path. (b) Overview of the Processing Element.

paradigm. The Controller runs in a specific part of the MCSoC. Some works assume a dedicated core [8], while others implement it as a high priority task [9]. The Controller abstracts the network management to the system manager (*M*), that performs application admission and task mapping. Due to such abstraction, when *M* needs a path between a source (*S*) and a target (*T*) task, it performs a path request to the Controller (step 1). The Controller searches the path guided by the network status and its current policy (step 2). The example in Figure 1(a) explores a scenario where the Controller searches the shortest path between *S* and *T*, avoiding hot-spots areas (in red) and faulty routers (router 3 × 3). After finding a path, the Controller physically configures the path (step 3) by sending a configuration packet to the respective SDN routers. When the configuration ends, the Controller sends an acknowledgment packet to *M* (step 4), which configures *S* and *T* to use the path (step 5).

Note that the SDN paradigm differs from existing software-based management techniques already explored in NoCs, e.g., to manage dynamic Time Division Multiplexing (TDM) [10] or Spatial Division Multiplexing (SDM) allocation for real-time guarantees [11]. Such techniques focus on specific goals and do not assume the NoC routers as a generic resource that can dynamically change its path search policy according to runtime constraints.

Although several works describe SDN designs for NoCs [7], [8], [12], [13], we identify the lack of SDN frameworks detailing the protocols and methods to achieve secure SDN management. Therefore, the work's *main contribution*

is a systemic and secure SDN framework (SDN-SS) for the design of MCSoCs, with the following original contributions:

- 1) a step-by-step framework description addressing the phases required to support a secure SDN management for MCSoCs (steps 1 to 5 in Figure 1(a));
- 2) a secure SDN router configuration mechanism, step 3, based on a hardware/software co-design;
- 3) a dynamic subnet change protocol allowing user's tasks to change at runtime its communicating network, without losing packets (step 5).

The path search (step 2) is out of the scope of this work. The path search consists of heuristics that vary according to the communication policies required by the designer.

The rest of this paper is organized as follows. Section II presents related work. Section III presents the reference MCSoC architecture. Section IV presents the SDN-SS framework. Section V presents experimental results, and Section VI presents conclusions and direction for future works.

**II. RELATED WORK**

SDN is a paradigm rather than an implementation or design, which drives to different proposals. Authors [3], [5], [6], [8], [14] propose a solution based on a centralized Controller. Authors in [7], [9], [13] address distributed approaches, aiming to improve scalability. For the sake of simplicity, the present proposal adopts a centralized Controller but also supports the distributed organization as proposed in [9].

Authors [6], [9], [13], [14] propose generic SDN management, without specializing the SDN architecture to a specific constraint, focusing on investigating the pros and cons of the SDN paradigm for intra-chip communication. On the other hand, Scionti *et al.* [7] address SDN for power savings by switching off links not used. Kostrzewa *et al.* [3] address QoS by making the Controller manage the communication demands by regulating the user's task injection rate according to the system resources and application's constraints.

The main concern of previous proposals is the SDN architecture and Controller's quality, e.g., path length and path search overhead. The SDN research moves to the system level, requiring SDN frameworks for Many-core systems, covering the interaction between the Controller, the hardware, the operating system (OS), and the impact on the user's applications. The SDN-SS proposed in this work addresses such gap, describing and evaluating actions involved in SDN-SS and its security properties.

Previous SDN proposals for MCSoCs do not address security, except for the work of Ellinidou *et al.* [8]. That work assumes an architecture called Cloud of Chips, where multiple ICs are placed in a Printed Circuit Board (PCB). Each IC has an SDN switch connected to other IC's switches and a Controller. The Controller runs in a dedicated off-chip processor into the PCB and is in charge of controlling the communication between ICs by configuring the SDN switches. The proposal defines a secure communication protocol between the Controller and switches with three phases:

(i) definition of a private key from a trusty key generation module; (ii) creation of a group of secure switches using a group key agreement protocol between switches and Controller; (iii) data transmission between switches and the Controller using the OpenFlow protocol [1]. Only switches of the group having the key can access the encrypted data. Authors evaluate their proposal using the Mininet 3 emulator, with groups having 2 up to 30 SDN routers. Results showed a Controller delay varying from 100 milliseconds up to 6 seconds for 6 and 30 switches, respectively. Regarding the Controller memory footprint, it varies from 23.40 KB (5 SDN routers) up to 32.70 KB (15 SDN routers).

This work and the Ellinidou *et al.* [8] proposal share the same underlying security issue: *how to make the Controller securely configure SDN routers*. Ellinidou *et al.* [8] adopt a different target architecture based on a Chiplet design, implementing the protocol entirely in software, requiring few interactions between the Controller and switches along with the execution of costly protocols for group key agreement. We adopt a different approach by targeting an intra-chip design, and exploiting a hardware/software co-design, reducing the overhead to configure SDN routers without compromise security, as addressed in Section V.

### III. REFERENCE MCSOC ARCHITECTURE

Figure 1(a) overviews the MCSoc architecture, with a set of Processing Elements (PEs) connected to a 2D-mesh NoC. Figure 1(b) shows the PE architecture. The PE computational resources comprise a local memory and a CPU.

At the communication level, the NoC adopts a Multiple Physical Network (MPN) design, with one Packet-Switching (PS) router and a set of SDN routers (SR) transmitting data by circuit-switching (CS). Best-effort flows and communication management use PS subnet. Flows with constraints, as QoS or security, use SDN subnets.

The adoption of MPN comes from its advantages compared to TDM and VC (Virtual Channels) [15], [16]: better scalability and implementation simplicity. Each SDN router is a simple hardware unit that connects an input port to an output port, according to a configuration process. The SDN router corresponds to 25% of the area and power of a PS router [4], with the same flit width. The PS subnet adopts a 32-bit flit-width, while the SDN subnets adopt 16-bit flit-width, reducing the SDN area even more.

Each PE has a Network Interface (NI), which abstracts the MPN NoC protocols to the PE. Due to the SDN paradigm, the NI has a sub-module called SDN Configuration Logic (NI-SCL), used to receive commands sent by the Controller and configure SRs (Section IV further details this process). The Controller configures a given SR by connecting an input port to an output port. Data coming from the input port is forwarded to the output with a delay of one clock cycle due to the buffering process required to avoid long wires. Further details related to the SR, the MPN, and the Controller designs are available at [4], [5].

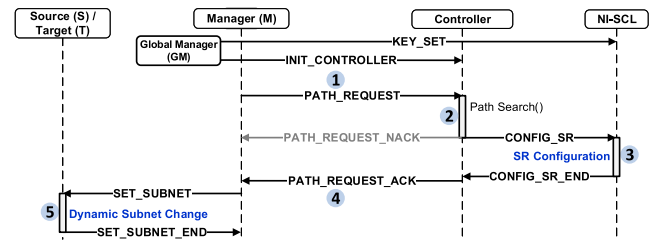


FIGURE 2. High-level messages of the SDN-SS framework.

At the management level, the MCSoc is partitioned into clusters [17], each one having a manager PE (*M* Figure in 1(a)) and a set of PEs that execute users' tasks. Although Figure 1(a) presents one *M* PE, the system contains several *M* PEs, being one the Global Manager PE (*GM*).

PEs that run the user's tasks have a tiny OS (~10KB), which controls multi-task scheduling, interruptions, system calls, memory management, and inter-task communication.

Manager and user's task PEs have the same hardware and differ from each one through operations performed by the OS. Some actions, such as cryptographic key generation, messages creation, messages authentication, among others, are crucial to the security of the system. To ensure the correct behavior of the OS, a secure boot process must ensure its integrity and authenticity. These features are obtained through mechanisms like encryption and Message Authentication Code (MAC). In this work, we assume the integrity and authenticity of the OS bootload process, although it is out of our scope. Works [18]–[20] describe methods enabling a secure boot process.

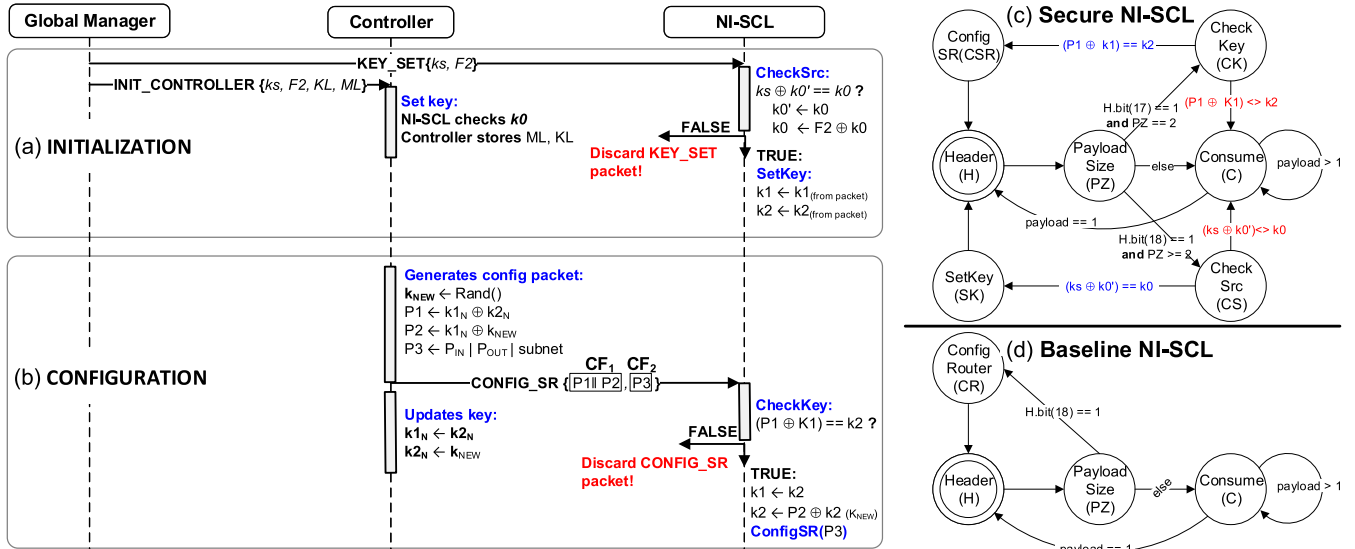
The user's tasks communicate through message passing, assuming non-blocking transmission and blocking reception. The source task (*S*) writes the packet to an OS memory area and continues its execution. The target task (*T*) sends a packet request, having its execution blocked. The PS subnet uses a request packet, while CS subnets use a dedicated signal signaling to *S* the requested packet. The *S* task sends the requested message once it has produced it. The advantage of this method is that the network is not blocked by packets waiting for its consumption, reducing congestion.

Due to the MPN communication, the OS uses, for each task, a data structure to control the subnet that a given task send/receive its packets (detailed in Section IV-F).

### IV. SYSTEMIC AND SECURE SDN FRAMEWORK (SDN-SS)

This Section details the SDN-SS framework, covering the steps required to support secure SDN management over an MCSoc system. Figure 2 presents the SDN-SS high-level messages. The highlighted numbers (1 to 5) corresponds to the steps presented in Figure 1. The framework includes the following actors:

- *S*, *T* (software): source and target user's tasks;
- *M* (software): cluster manager, does not execute user tasks, is responsible for task mapping, requests for connections



**FIGURE 3.** Secure SR configuration: (a) initialization protocol, (b) configuration protocol. Finite state machines responsible for handling configuration packets: (c) secure NI-SCL. (d) baseline NI-SCL.

to the Controller, takes self-adaptive actions, and manages the dynamic subnet change protocol among  $S$  and  $T$ ;

- *Controller* (software): responsible for the SDN management, such as path search, and SR configuration;
- *NI-SCL* (hardware): SDN Configuration Logic, runs in the NI, handling configuration packets sent from the Controller and configuring SRs;
- *GM* (software): its role in the context of this proposal is to initialize the Controller and NI-SCLs. Besides, it executes the  $M$  PE functions and controls the admission of new applications at runtime by interacting with off-chip components.

At system startup, the *GM* randomly generates and transmits a key tuple  $\{k_0, k'_0\}$  for each *NI-SCL*. The adoption of key tuples allows hiding the keys during their transmission. Each *NI-SCL* stores its keys in internal registers. The *GM* keeps in its memory all key tuples. After generating and transmitting the key tuples, the *GM* initializes the Controller and the *NI-SCLs* (Section IV-A) and releases the applications' admission.

The next subsections detail the SDN-SS framework based on the sequence diagram of Figure 2.

### A. CONTROLLER AND NI-SCL INITIALIZATION

The Controller and *NI-SCL* initialization is the protocol step 0 since its execution occurs after system startup or when the Controller detects an attack in a given *NI-SCL*. Figure 3(a) details the Controller and *NI-SCL* initialization steps.

#### 1) NI-SCL INITIALIZATION

All *NI-SCLs*, excepting the PE(s) running the Controller(s), receives a 2-flit *KEY\_SET* packet from the *GM*, with the following contents:

- 1)  $k_s$  (Equation 1), is a one-time key enabling the authentication between the *GM* and *NI-SCL*.

$$k_s = k_0 \oplus k'_0 \quad (1)$$

- 2) key-tuple  $(k_1, k_2)$ , *SRs* uses these keys to authenticate the configuration packet coming from the Controller during the SR configuration step (Section IV-D). The *GM* generates a different tuple for each *NI-SCL*. To avoid exposing these keys during their transmission, the second flit ( $F_2$ ) is encoded according to Equation 2.

$$F_2 = (k_1 || k_2) \oplus k_0 \quad (2)$$

where:  $k_1$  and  $k_2$  are 16-bit keys;  $||$ : concatenation;  $\oplus$ : bitwise XOR operation.

The *NI-SCL* uses the 1<sup>st</sup> *KEY\_SET* flit to check the packet authenticity, retrieving  $k_0$  from  $k_s$ , comparing it with the stored  $k_0$ . The packet is discarded if values do not match. If they match, the *NI-SCL* extracts from the 2<sup>nd</sup> flit the key tuple  $(k_1, k_2)$ , storing it. The Controller receives the same key tuple, enabling their synchronization.

Figure 3(c) shows the FSM responsible for handling configuration packets. The initialization comprises the loop with states H-PZ-CS-SK. Figure 4, cycles 2–6, shows the waveform corresponding to the *KEY\_SET* packet reception, which has four flits: *H* – header; *PZ* – payload size;  $k_s$  (Equation 1);  $k_1$  and  $k_2$  keys (Equation 2).

A packet arriving in the *NI-SCL* goes through states H and PZ, with both flits stored in registers. Once in PZ state, the FSM verifies if the 18<sup>th</sup> bit of the packet header (H) is equal to one (flag to signalize a *NI-SCL* configuration, avoiding the packet consumption by the PE), and  $PZ = 2$ . Meeting these conditions, the FSM advances to state CheckSrc (CS), retrieving  $k_0$  from  $k_s$ , which must be equal to the stored  $k_0$ . If the comparison returns true, the FSM advances to SetKey (SK)

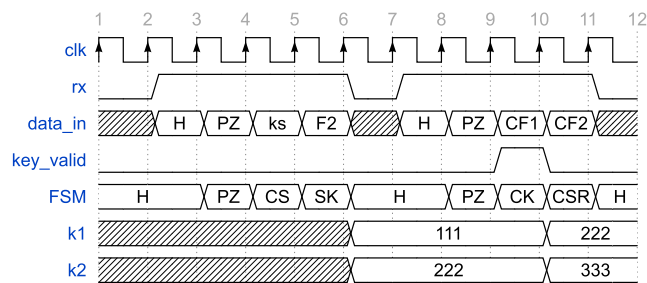


FIGURE 4. Waveform at NI-SCL of key set and SR configuration.

state, extracting  $k1$  and  $k2$  from the packet, storing them in key registers (values 111 and 222 in the Figure). Otherwise, the packet is assumed malicious, and the FSM discards the packet, advancing to the Consume state (C).

2) CONTROLLER INITIALIZATION

The PE executing the Controller receives from the GM an INIT\_CONTROLLER packet with the following fields:

- 1)  $k_s$  (Equation 1), is a one-time key enabling the authentication between the GM and the Controller.
- 2) key-tuple  $(k1, k2)$  (Equation 2), to initialize the NI-SCL of the PE where the Controller is running.
- 3) list  $KL = \{(k1, k2)_{1 \times 0}, (k1, k2)_{2 \times 0}, \dots, (k1, k2)_{n \times m}\}$ , private keys tuples for all NI-SCLs. Each NI-SCL received its key tuple in the NI-SCL initialization step.
- 4) list  $ML = \{A_{M1}, A_{M2}, \dots, A_{Mn}\}$ , addresses of all  $M$  PEs authorized to request SDN services to the Controller.

As in the NI-SCL initialization, the NI-SCL uses the 1<sup>st</sup> flit ( $k_s$ ) to check the packet authenticity, discarding the packet if  $k_0$  does not match with the stored one. The 2<sup>nd</sup> flit,  $F2$  (Equation 2), initializes the NI-SCL key tuple, as previously explained, at state SK. Differently from the KEY\_SET packet, the FSM releases the consumption of the remaining flits by the processor by verifying if  $PZ > 2$ .

The packet enters into the Controller, where the remaining packet flits containing the  $KL$  and  $ML$  lists are read and stored. The  $KL$  flits are XORed with  $k_0$ . This process avoids exposing the keys during their transmission.

3) INITIALIZATION SECURITY MECHANISM

Authentication is the method preventing initialization packets from being forged, based on the one-time key tuple  $\{k_0, k'_0\}$ , initially defined at system startup. After the configuration packets (KEY\_SET and INIT\_CONTROLLER), the GM and the NI-SCLs update their key tuples, as follows:  $k_0 \leftarrow k'_0$  and  $k'_0 \leftarrow k1 || k2$ . If the protocol needs reinitialization, the process uses new key tuples, with new  $k1$  and  $k2$  randomly generated by the GM.

Such authentication process also provides flexibility to the initialization process of the SDN-SS framework, allowing the modification of  $KL$  and  $ML$  lists at runtime.

B. STEP 1 – PATH REQUEST

The  $M$  PEs decide when a communicating task pair needs an SDN path. Examples of events triggering path requests include: fault notification in a given subnet; the PS subnet is no longer able to sustain QoS; a set of communicating flows requires isolated communication channels to enforce security.

$M$  requests an SDN path for a given communicating task pair by sending to the Controller a PATH\_REQUEST packet (Figure 2). This packet contains the manager address ( $A_M$ ) and the  $S$  and  $T$  addresses.

The Controller accepts the PATH\_REQUEST packet iff  $A_M \in ML$ , avoiding unauthorized path requests. The Controller rejects any packet whose address is not in  $ML$ .

It is possible to enforce the security of the  $A_M$  transmission using LFSRs (Linear Feedback Shift Register). In this case, the LFSR value generated by the source PE ( $M$ ) encodes  $A_M$  (XOR operations), and the LFSR value in the target PE (Controller) decodes  $A_M$  (XOR operations). The requirement enabling these operations is a common seed to maintain both LFSRs synchronized.

C. STEP 2 – PATH SEARCH

After accepting the PATH\_REQUEST packet, the Controller executes a path search heuristic. It is possible to adopt different routing management policies, e.g., avoid hot-spot regions, faulty routers, ensure QoS, isolate confidential communication. It is out of the scope of this work the proposal of an SDN path search heuristic. In the experiments, we adopt the Hadlock algorithm (shortest path [4], [21]), chosen due to its advantages compared to similar routing algorithms [22].

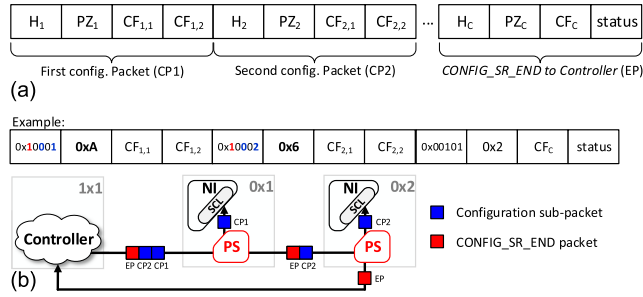
D. STEP 3 – SDN ROUTER (SR) CONFIGURATION

The major SDN security vulnerability is the SR configuration, made by configuration packets generated at the software level [23]. Thus, the main security concern of this work consists in securing the SR configuration, avoiding that malicious tasks configure SRs at runtime taking control of the SDN subnets.

In summary, the secure SR configuration uses two private keys:  $k1$  and  $k2$ . The protocol initialization ensures that both NI-SCL and Controller have the same keys. When the Controller needs to configure an SR, it sends an SR configuration packet with two flits,  $\{CF1, CF2\}$ . Flit  $CF1$  contains two keys, derived from  $k1, k2$ , and  $k_{new}$ , randomly generated. The NI-SCL receives the SR configuration packet. If it retrieves the correct keys, it configures the SR and updates its keys.

Using one key would be enough to NI-SCL authenticate the Controller, but this also could expose the key to an attacker. The adoption of two keys allows hiding the keys during their transmission to the communicating pair. Additionally, the new key,  $k_{new}$ , updates the keys at each configuration, avoiding the use of static keys and, consequently, replay attacks.

Figure 3(b) overviews the proposed secure SR configuration further detailed into the next Subsections.



**FIGURE 5. (a) Configuration packet format. (b) Example showing a packet composed of three sub-packets. The first two sub-packets configure SR routers of PE  $0 \times 1$  and  $0 \times 2$ . The last sub-packet contains the CONFIG\_SR\_END packet that is sent back to Controller.**

### 1) SR CONFIGURATION PACKET

When the path search step finishes, the Controller sends a configuration packet (CONFIG\_SR) to configure the SRs in the path using the PS subnet.

Figure 5(a) details the CONFIG\_SR packet structure, and Figure 5(b) shows an example of a packet configuring the SR into PE  $0 \times 1$  and  $0 \times 2$ . The Controller sends the packet to the NI-SCL of the S PE. Arriving at the S PE, the packet advances PE by PE in the path, entering into the NI-SCL and configuring the respective SR. Reaching the T PE, the packet returns to the Controller to acknowledge it that the configuration process finished.

The SR packet contains a set of sub-packets (CP1, CP2, and EP in Figure 5). Each sub-packet has four flits: header (H), Payload Size (PZ), configuration flit 1 ( $CF_1$ ), configuration flit 2 ( $CF_2$ ). The process to handle this packet requires minimal changes to the router. When an input port detects a header flit having the 17<sup>th</sup> bit equal to 1, the second flit is replaced by the constant two. Thus, the packet is divided into two packets, one consumed locally (flits  $CF_1$  and  $CF_2$ ), and a second packet that is forward to the other PS routers in the path.

The last sub-packet is the CONFIG\_SR\_END, which is sent back to the Controller after configuring all SRs in the path. It also contains two flits, being the first one ( $CF_c$ ) an XOR operation among all  $k_{new}$  embedded into the CONFIG\_SR packet.  $CF_c$  enables the Controller to authenticate the CONFIG\_SR\_END packet, discarding malicious path notifications. The second one flit (*status*) notifies the Controller of the path creation status (successful or fail).

The mechanism adopted to notify the Controller of a failed configuration is the modification of the status flit. When a key verification mismatch occurs on a given NI-SCL (CK state in the FSM), it generates a signal to the PS router, which activates an error bit in the status flit before the CONFIG\_SR\_END packet leaves the router. The Controller receiving the status flit with the error bit activated may ask to GM to reinitialize the protocol as detailed in Section IV-A3.

The configuration packet is always transmitted using the XY routing algorithm. When the packet reaches a router in the path, it is broken into two packets. The NI-SCL consumes the first one, which has 2 payload flits. The second part of

the packet advances to the next router in the SR path. The last part of the packet returns to the Controller using XY routing.

The advantage of this transmission method is the simplification of the software required for sending data since only one packet is injected into the PS network. The cost in the router is minimal, corresponding to the verification of one bit of the header flit (one *and* gate), a multiplexer to allow the insertion of the constant 2 (packet size locally consumed), and an *or* gate to set the error bit in the status flit.

The Controller generates each configuration sub-packet with  $CF_1$  and  $CF_2$  flits.  $CF_1$  contains the keys ( $k_1, k_2, k_{new}$ ), and  $CF_2$  stores the SR configuration commands. Equation 3 presents  $CF_1$ :

$$CF_1 = (k_1 \oplus k_2) || (k_2 \oplus k_{new}) \quad (3)$$

where:  $i$ : PE address of the SR to configure;  $||$ : concatenation;  $\oplus$ : bitwise XOR operation.

Equation 4 details  $CF_2$ :

$$CF_2 = P_{IN} || P_{OUT} || subnet \quad (4)$$

where:  $P_{IN}$ : input port;  $P_{OUT}$ : output port; *subnet*: SDN subnet addressing the SR to configure.

### 2) NI-SCL CONFIGURATION

Figure 4 (cycles 7 – 11) shows the waveform corresponding to the reception of a CONFIG\_SR sub-packet by NI-SCL. When it detects the packet (17<sup>th</sup> bit of H and PZ = 2), the FSM advances to the CheckKey (CK) state, where  $k_2$  is retrieved from  $CF_1$ , as depicted in Figure 3(b).

If the retrieved  $k_2$  matches with the stored  $k_2$ , signal *key\_valid* rises (cycle 9 in Figure 4), allowing the transition to the ConfigSR (CSR) state. If  $k_2$  does not match with the stored  $k_2$ , the FSM assumes that this is a malicious configuration packet, transitioning to state C, which discards the remaining flits of the packet and signaling to set the status bit of the EP sub-packet, indicating an error in SDN configuration.

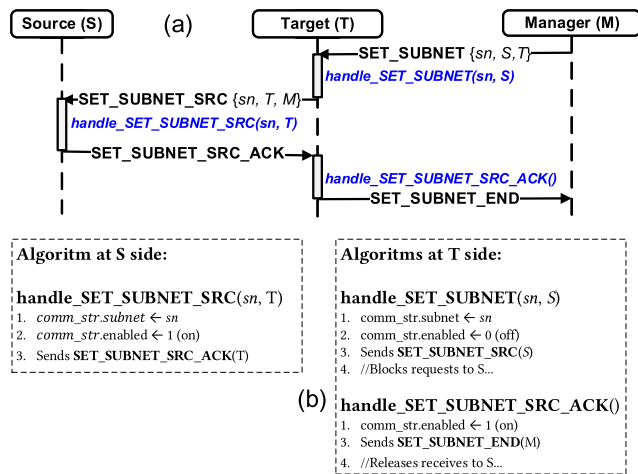
The CSR state reads the configuration parameters from  $CF_2$ :  $P_{IN}$ ,  $P_{OUT}$ , and *subnet*. These parameters configure the SR internal crossbar specified on the *subnet*, connecting port  $P_{IN}$  to port  $P_{OUT}$ . A NI-SCL register receives the *subnet* number. This register makes the hardware-software interface. The OS reads this register at step 5 to certify that this subnet was the most recently configured subnet.

After configuration, the NI-SCL updates its keys, with  $k_1$  receiving  $k_2$  and  $k_2$  receiving  $k_{new}$ . The Controller proceeds in the same way, allowing it always to be paired with each NI-SCL, with new keys at each configuration.

### 3) SECURITY ATTRIBUTES OF THE SR CONFIGURATION

Mechanisms ensuring the secure SR configuration include:

- 1) The keys used in SR configuration never remains the same. At each configuration the Controller generates a random  $k_{new}$ , updating  $k_1$  and  $k_2$  at both Controller and NI-SCL;



**FIGURE 6.** (a) Dynamic subnet change protocol. (b) Dynamic subnet change algorithms of the OS running S and T tasks.

- 2) Keys  $k_1$  and  $k_2$  are not exposed during their transmission in the NoC, due to adoption of the procedures presented in Equation 3.

Despite costly, a brute force attack could discover  $k_1$  or  $k_2$  from  $CF_1$  flit. This attack is unlikely to occur, as it would be necessary to determine from 1 flit three distinct values ( $k_1$ ,  $k_2$ , and  $k_{new}$ ). Moreover, the keys are renewed at each new configuration and are unique for each  $NI-SCL$ . If the attack occurs, there will be a loss of synchronization between the Controller and the compromised  $NI-SCL$ . The Controller identifies this issue when it tries to establish a new path using the attacked  $NI-SCL$ , and the countermeasure is the protocol reinitialization (Section IV-A).

#### E. STEP 4 – PATH REQUEST ACKNOWLEDGMENT

After receiving the CONFIG\_SR\_END (end of step 3), the Controller sends the PATH\_REQUEST\_ACK (or NACK) to the  $M$  PE that requested the path. If the path was created successfully, the acknowledgment packet also carries the configured subnet number. Manager PEs only accept CONFIG\_SR\_END packets from the PE where the Controller is running.

As in the PATH\_REQUEST packet (step 1), it is possible to enforce the security of this step by using an LFSR to encode the payload content (ACK/NACK).

#### F. STEP 5 – DYNAMIC SUBNET CHANGE

The *dynamic subnet change* is the process of configuring the user's tasks  $S$  and  $T$  to use the created SDN path. The Controller starts this protocol once it receives the PATH\_REQUEST\_ACK packet.

Figure 6(a) details the *dynamic subnet change* protocol. This protocol enables the transition from any subnet to another one without packet loss. Its implementation includes the  $M$  PE and the operating systems of the communication task pair  $\{S, T\}$ , and it is no accessible at the task level.

The  $M$  PE starts the protocol by sending to  $T$  a packet having in its payload  $\{sn, S_{address}, T_{address}\}$ , where  $sn$  is the subnet number transmitted by the PATH\_REQUEST\_ACK packet.  $S$  and  $T$  performs a handshake protocol, executing the algorithms detailed into Figure 6(b).

The goal of the algorithms is to ensure that both tasks use the new subnet without losing packets transmitted on the previous subnet. To achieve this goal, the algorithms control the *comm\_str* variables, ensuring inter-task communication synchronization as follows:

- $T$  executes the *handle\_SET\_SUBNET()* algorithm when it is not blocked waiting for a packet from  $S$ . The SET\_SUBNET packet reception does not result in the immediate execution of this algorithm. The  $T$  task should not be waiting for packets, ensuring that there are no packets to  $T$  traversing the NoC.
- The *handle\_SET\_SUBNET()* algorithm execution assigns  $sn$  to  $T$ , notifies  $S$  that it should use a new subnet (SET\_SUBNET\_SRC packet), and blocks new packets' requests.
- $S$  receiving SET\_SUBNET\_SRC packet updates the *comm\_str* structure enabling the subnet use, and notifies  $T$  the successful subnet change;
- $T$  activates the new subnet (*comm\_str.enable* ← 1), releasing packets' requests.

#### 1) SECURITY ATTRIBUTES OF THE DYNAMIC SUBNET CHANGE PROTOCOL

The PE receiving the SET\_SUBNET packet ( $\{sn, S_{address}, T_{address}\}$ ) verifies the packet authentication through 3 comparisons: (i) the packet source must be a manager PE; (ii)  $T_{address}$  should correspond to the current PE address; (iii)  $sn$  should correspond to the last configured  $SR$  (set on the read-only mapped register during the  $SR$  configuration - Section IV-D2).

#### G. OVERVIEW OF THE SECURITY MECHANISMS

The security mechanisms adopted at each stage of the SDN-SS framework (original contribution 1) are summarized below:

- System Startup
  - $GM$  generates a key tuple for each  $NI-SCL$ ,  $(k_0, k'_0)$ , before applications' admission.
- Controller and  $NI-SCL$  Initialization
  - initialization packets, INIT\_CONTROLLER and KEY\_SET, authenticated with a one-time session ID key,  $k_s$ , derived from  $(k_0, k'_0)$ . Key  $k_s$  changes at each initialization, preventing replay attacks.
- Step 1 - Path Request
  - requester ( $A_M$ ) should be in the  $ML$  list of controllers;
  - additional mechanism (not implemented) - encode  $A_M$  by using an LFSR.
- Step 2 - Path Search

- there is no need for security mechanisms since this step is executed in the Controller.
- Step 3 - SDN Router (SR) Configuration (original contribution 2)
  - adoption of key tuple,  $(k1, k2)$ , for each *NI-SCL*, updated at every configuration;
  - key obfuscation during its transmission through the PS subnet;
  - configured subnet number stored in a register, which is read-only by the processor.
- Step 4 - Path Request Acknowledgment
  - the *M* PE verifies if the packet comes from the controller address.
  - additional mechanism (not implemented) - encode the packet payload by using an LFSR.
- Step 5 -Dynamic Subnet Change (original contribution 3)
  - PE verifies if the packet comes from an *M* PE;
  - PE verifies if subnet specified in the SET\_SUBNET packet matches with the value stored in a register at step 3.

A final remark, all keys used in the SDN-SS framework are dynamic, using key tuples  $((k_0, k'_0), (k1, k2))$ . This method prevents replay attacks as well as obfuscate the packets' payload.

**V. EXPERIMENTAL RESULTS**

The presentation of the results has three parts. The first one (Subsection V-A), evaluates the performance of all steps of the SDN-SS framework and the impact over the user's tasks communication performance. The second one (Subsection V-B), evaluates the hardware/software co-design costs and performance along with the secure *SR* configuration. The last one (Subsection V-C) exploit examples of attacks.

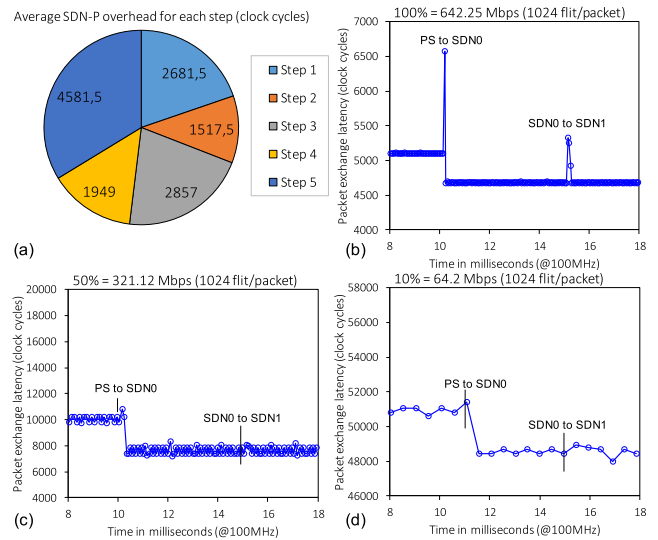
The MCSoC system and hardware architecture was modeled using the *Memphis* MCSoC [24]. The baseline MPN, *SR* router design, and Controller were previously proposed in [4], [5], [9]. The design proposed in this work is public available at [25]. The hardware components are implemented using an RTL hardware description (SystemC and VHDL). The CPU is a RISC processor [26]. The software (OS, Controller, user's tasks) is modeled in C code (*mips-gcc* cross-compiler, version 4.1.1, optimization *O2*). Table 1 summarizes the main system setup parameters for experiments.

**A. SDN-SS FRAMEWORK COSTS**

Figure 7 details the average execution time (AET) of each step of SDN-SS framework, and its impact on the user's tasks with different communication profiles. The methodology to obtain these results consisted of instantiating two synthetic tasks, *S* and *T*, with *S* sending packets to *T* with a payload having 1024 32-bits flits. Tasks start communicating using the PS subnet. At 10 ms and 15 ms, the *M* PE transmits

**TABLE 1. System setup adopted in the experimental results.**

Parameter	Setup
Clock frequency	100 MHz
Hardware description	VHDL, System-C RTL
Software description	C language
CPU	RISC Processor
System dimension	5x5 (25 PEs)
<i>GM</i> position	PE 0x0
Controller position	PE 0x1
Inter-task communication	MPI
PS flit width	32 bits
SR flit width	16 bits
SR subnets	2



**FIGURE 7. Performance evaluation of the SDN framework.**

PATH\_REQUEST packets to the Controller, triggering the execution of all steps of the SDN-SS framework.

Figure 7(a) details AET for each step. The SDN-SS requires, on average, 13,586 clock cycles to concluded all steps (135.8  $\mu$ s@100MHz).

- Step 1 (path request) - 19.7% of the AET, measured from the moment that *M* starts to create the PATH\_REQUEST packet up to the moment the Controller certifies that the packet comes from a trusty *M*.
- Step 2 (path search) - 11.2% of the AET, measured from the end of step 1 up to the moment the Controller starts to generate the *SR* configuration packet. Although step 2 is out of the work's scope, we include its execution time in Figure 7(a) to present the whole process's cost. Factors as network status, path establishment polices, and the path search algorithm impact the path search overhead.
- Step 3 (*SR* configuration) - 21.0% of the AET, measured from the end of step 3 until the reception of the CONFIG\_SR\_END packet by the Controller. The *SR* configuration overhead is proportional to the path size (number of *SR*s to configure). Next Subsection exploits the overhead of step 3, evaluating different numbers of *SR*s in the path.



- Step 4 (path request ack.) - 14.3% of the AET, measured from the end of step 3 until the moment  $M$  receives the PATH\_REQUEST\_ACK packet. As in step 1, step 4 has a constant execution time.
- Step 5 (dynamic subnet change) - 33.8% of the AET. Its execution time does not increase with the system size, because this step always involves two PEs,  $S$  and  $T$  tasks. However, step 5 can impact the user's task communication latency since tasks need to synchronize to change its subnet at runtime.

Figures 7(b-d) present in the y-axis the average latency to transmit packets from  $S$  to  $T$  in  $cc$ , and in the x-axis, the simulation time (ms). The evaluation addressed three communication profiles: (**P1**) communication-intensive exploring the maximum throughput provided by the system (624.25 Mbps); (**P2**) intermediate communication profile, using 50% of the available throughput (321.12 Mbps); (**P3**) low communication profile, using 10% of the available throughput (64.2 Mbps).

The first SDN-SS actuation, in 10ms, changes the switching mode from PS to CS. As expected, we observe a latency reduction in all scenarios. A latency peak was observed during the actuation due to the execution of the dynamic subnet change protocol (step 5), which is proportional to the traffic profile. The evaluated scenarios presented a latency increase during actuation of: 19.42% (+1456  $cc$ ), 7.1% (+ 642  $cc$ ), and 0.9% (+468  $cc$ ) for P1, P2 and P3 respectively.

The second actuation, in 15 ms, only changes the CS subnet. Since the packet request mechanism is performed with a dedicated signal, and not by a request packet, the latency peak is noticeably lower.

Communicating tasks with low communication rates (as in P3 scenario) are less susceptible to interference from step 5 because the dynamic subnet change protocol can partially or fully be executed in moments that tasks are computing or are in an idle state waiting for data.

Related works on SDN for MCSocS lack a similar evaluation. For comparison purposes, task migration protocols used to mitigate congestion effects takes on average 200,000  $cc$  [27], while the SDN-SS required 13,586  $cc$  (6.7% of a task migration).

## B. SECURE SR CONFIGURATION

This Subsection addresses the evaluation of the Controller and  $NI$ - $SCL$  design complexity, along with the evaluation of the secure  $SR$  configuration.

### 1) HARDWARE COMPLEXITY

Experiments to obtain area and power results used Cadence Genus tool, with a 28nm-FDSOI library, at 1 GHz. The simulation generated switching activity reports, TCF (Toggle Count Format) files, enabling the power estimation. Test-benches explored worst-cases scenarios with the maximum possible switching activity.

Total NI power (28nm-FDSOI@1GHz)				
Design	Cell Count	Leakage ( $\mu$ W)	Dynamic ( $\mu$ W)	Total ( $\mu$ W)
Baseline	3488	4.90	845.05	849.95
Secure	3725 (+6.7%)	5.18 (+5.7%)	878.97 (+4%)	884.16 (4+%)
Total NI area (28nm-FDSOI@1GHz)				
Design	Cell Area ( $\mu$ m <sup>2</sup> )	Net Area ( $\mu$ m <sup>2</sup> )	Total Area ( $\mu$ m <sup>2</sup> )	
Baseline	5947.824	2104.590	8052.414	
Secure	6290.054 (+5.7%)	2244.378 (+6.6%)	8534.433 (+5.9%)	

NI-SCL power (28nm-FDSOI@1GHz)				
Design	Cell Count	Leakage ( $\mu$ W)	Dynamic ( $\mu$ W)	Total ( $\mu$ W)
Baseline	274	0.306	44.51	44.82
Secure	518 (89+%)	0.597 (95+%)	78.87 (+77.1%)	79.47 (+77.3%)
NI-SCL area (28nm-FDSOI@1GHz)				
Design	Cell Area ( $\mu$ m <sup>2</sup> )	Net Area ( $\mu$ m <sup>2</sup> )	Total Area ( $\mu$ m <sup>2</sup> )	
Baseline	350.717	147.133	497.850	
Secure	686.909 (+95.8%)	293.277 (+99.3%)	980.186 (96.8+%)	

FIGURE 8. NI and NI-SCL area and power evaluation.

Figure 8(a-b) presents the power and area for the NI and  $NI$ - $SCL$ , respectively. The  $NI$ - $SCL$  hardware contains an FSM, registers to store the two key tuples ( $\{k1, k2\}$ ,  $\{k0, k'0\}$ ), and registers to store the  $SR$  configuration. Figure 3(d) presents the baseline  $NI$ - $SCL$  [4], without security mechanisms. The comparison of the proposed secure  $NI$ - $SCL$  with the baseline  $NI$ - $SC$  shows an area and power increase of 96.8% on 77.3%, respectively. However, as the NI has other modules in charge to receive and send packets, memory interface, and interruption generation, the NI overhead for supporting the secure SCL was, on average, 5.9% higher on area, and 4% higher on power.

The  $SR$  memory size to store the configuration coming from Controller has a small size. Each input has to keep the output where it has to redirect incoming flows, as the number of output ports is the same as inputs, the  $SR$  memory size is equal to  $I \times N_{bits}$ , where  $I$  is the input port number, and  $N_{bits}$  is the number of bits representing the number of output ports plus the free status.

Such results demonstrate the small hardware overhead of the proposed secure  $SR$  configuration mechanism. Work [8] does not perform such analysis since its scope is exclusively at the software level.

### 2) SOFTWARE COMPLEXITY (CONTROLLER)

The path search heuristic complexity dominates the Controller *computational* complexity, which is our case  $O(H_C)$ .  $H_C$  is the Hadlock's algorithm complexity:  $H_C = SR_N * S_N$ , where  $SR_N$  is the number of routers managed by the Controller, and  $S_N$  is the number of SDN subnets.

The computational complexity related only to the security features at Controller's scope comprises the generation of  $k_{new}$  for each  $SR$  configuration packet. The generation of  $k_{new}$  uses a clock cycle counter to create a pseudo-random number. Such operation has a fixed cost (a memory-mapped register), with a complexity of  $O(n)$  to create the  $SR$  packet, where  $n$  is the number of  $SR$  routers in the packet.

The *storage* complexity of the Controller is also dominated by its search path heuristic. The Hadlock's algorithm requires a storage complexity of (in bytes):

$$H_M = (SR_N * S_N * 5) + (3 * SR_N) \quad (5)$$

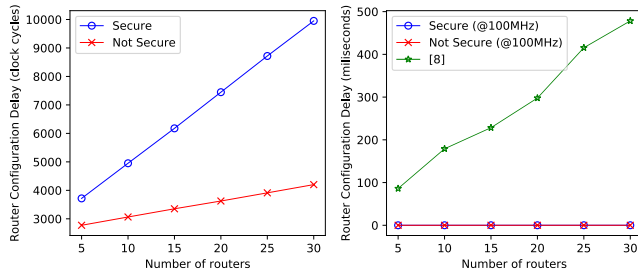


FIGURE 9. Performance evaluation of SR configuration. (a) Secure vs. not secure (baseline). (b) Comparison with related work.

where:  $SR_N$  is the total number of SRs of the system,  $S_N$  is the number of SDN subnets, and 5 is the number of ports per router (east, west, north, south, local).

Equation 6 presents the Controller’s storage complexity related only to the security features (in bytes):

$$SEC_M = 4 * n + 2 * m \tag{6}$$

where:  $4*n$  is the storage for keeping the 16-bit keys  $k1$  and  $k2$  for  $n$  NI-SCLs managed by the Controller ( $KL$  storage).  $2*m$  corresponds to the  $m$  2-bytes address of each  $M$  PEs managed by the Controller ( $ML$  storage).

For comparison purposes, [8] reports 32.7 KB (Controller’s memory) to support 15 switches. This work requires 2.1 KB to manage 15 NI-SCLs, 5 SDN subnets, and 1  $M$ , therefore, resulting in 75 SRs.

### 3) SR CONFIGURATION PERFORMANCE

Figure 9 addresses the performance comparison of SR configuration, varying the number of SRs. The router configuration delay (y-axis) comprises the time spent in step 3. Therefore, it is embedded into such overhead the Controller and NI-SCL delay. The evaluation exploits SR configurations from 5 up to 30 SRs.

Figure 9(a) compares the secure implementation with the baseline (not secure). As expected, the addition of the security mechanisms increased the delay to configure the SRs, from 34% to 136.8% for 5 to 30 SRs, respectively. Despite this difference, the secure approach requires less than 10,000  $cc$  for 30 SRs, which can be considered a low configuration overhead. Such overhead represents 0.1 ms@100 MHz (a conservative frequency).

Figure 9(b) compares our approach to [8]. Ellinidou et al. [8] present the configuration delay in milliseconds without specifying the clock frequency. The authors mention that their results were obtained with Mininet 3 simulator. For the comparison, we adopted a conservative clock frequency (100MHz) in our approach and selected the better results achieved in [8], which is the delay using the Teng’s group key agreement protocol. This comparison showed that the proposed router configuration represents less than 2% of the overhead achieved in [8].

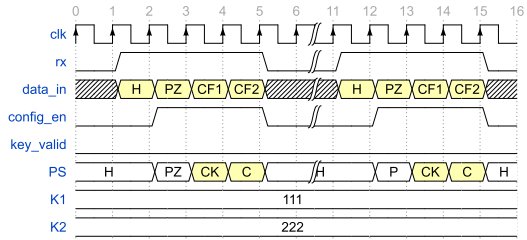


FIGURE 10. Example of a flooding attack.

### C. ATTACKS EXPERIMENTS

The assumed threat model comprises a malicious software task able to perform the following attacks:

- 1) Denial-of-Service (DoS): generation of an incorrect SR configuration packet, aiming to crash the NI-SCL;
- 2) Flooding: flood the NI-SCL with malicious SR configuration packets;
- 3) Spoofing: malicious packet trying to assume the identity of one actor of the framework ( $GM$ , Controller,  $M$ );

A malicious task ( $M_t$ ) running in a given PE executes the attacks in the SDN infrastructure.

#### 1) DoS AND FLOODING

In a DoS attack, we assume that  $M_t$  can generate a malicious SR configuration packet. The FSM responsible for handling incoming configuration packets (Figure 3(c)) verifies if the packet header has its 17<sup>th</sup> or 18<sup>th</sup> bit asserted, and the payload size is greater than or equal to 2. The FSM discards packets not meeting both conditions. If the malicious packet meets both conditions,  $M_t$  should try to forge  $CF_1$ , but as previously mentioned, this condition is unlikely to occur due to the scrambling key mechanisms contained in the  $CF_1$  field. Thus, a DoS attack does not occur because the FSM responsible for receiving incoming packets has protection barriers and mechanisms to discard malicious packets.

Figure 10 shows a flooding attack, where  $M_t$  has a loop that sends configuration packets continuously. The Figure details the reception of two packets, with 7  $cc$  between them due to OS overheads, limiting the injection rate. The flooding attack would occur if the FSM responsible for receiving the configuration packages took too long to handle them. Due to the hardware implementation of the discarding mechanism, these malicious packets are discarded during their reception. We observed that the Controller configuration latency is slightly penalized. Without the flooding attack, the Controller packet took 15  $cc$  to arrive at the NI-SCL against 20  $cc$  with this attack.

The NI-SCL cannot avoid DoS and flooding attacks since malicious tasks may run in the system. However, due to the presence of hardware mechanisms to discard malicious packets, DoS or flooding attacks do not stall the NI-SCL. The flooding attack may increase the latency to receive the configuration packet, but do not prevent the reception of the correct configuration packets due to the arbitration logic in the PS routers.

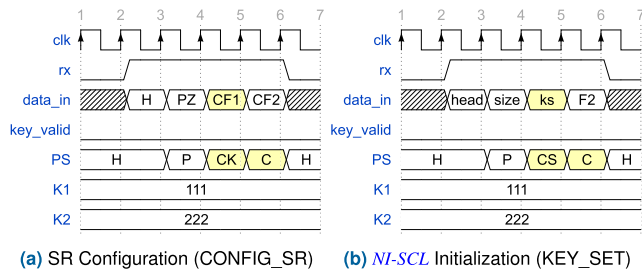


FIGURE 11. NI-Spoofing attacks example.

## 2) SPOOFING

Spoofing involves sending messages using a false source address, which makes the receiver assume that the packet comes from a trusted source (IP-like spoofing).

The *NI-SCL* avoids such attack in two ways:

- *key set*: the  $k_s$  authenticates packets coming from a trusted *GM*;
- *SR configuration*: prevented by the key tuple  $\{k_1, k_2\}$ .

Figure 11(a) shows an *SR* configuration spoofing attempt into *NI-SCL*. The *CK* state checks (cycle 4) the keys embedded into  $CF_1$ . As the keys do not match with *NI-SCL*'s key, *NI-SCL* does not configure the *SR* router and moves to the *C* state (cycle 5) that consumes the remaining flits of the packet. Similar behavior occurs in the key set attack of Figure 11(b). The  $k_s$  embed into the third flit of  $M_t$ 's packet allows the *NI-SCL* to verify that the packet comes from a false *GM*, which leads to the packet rejection by the consumption of its remaining flits (cycles 4 and 5).

$M_t$  may also try to forge the identity of an *M* PE, by sending a malicious *PATH\_REQUEST* to the Controller. A malicious *PATH\_REQUEST* sent by  $M_t$  can make the Controller unavailable, working to define a false path. Such an attack is avoided because the Controller receives the trusted manager list (*ML*) (during system initialization) authenticated with  $k_s$ . At runtime, the Controller checks the *src* address of the requester *M* PE, and only accepts the one coming from an *src* that is in the list.  $M_t$  cannot forge the *src* address because it is stamped by the OS *NI* driver just before the packet is injected into the NoC. Therefore, this method avoids tasks to forge a source address.

## 3) NOT SUPPORTED ATTACKS

The secure *SR* configuration does not support Hardware Trojans (HT) attacks. For instance, a hardware monitor can intercept an authentic *SR* configuration packet (man-in-the-middle), and transmit a new packet, keeping the same value of the  $CF_1$  flit, but changing the  $CF_2$  flit (configuration command). As  $CF_1$  stores the keys used to authenticate the packet, the *NI-SCL* will accept the configuration and will use the tampered value of  $CF_2$  to configure the *SR* router.

HTs have the power to perform several invasive attacks, which require costly hardware designs to mitigate them. As this work deals with SDN, our concern specially addresses

the software level by assuming threats coming from the user's tasks.

## VI. CONCLUSION AND FUTURE WORK

This work proposed a systemic and secure SDN framework running over an MPN architecture in MCSocS, allowing that only a trusted SDN Controller can define the communication path, a critical gap scarcely addressed in related works. Using the systemic details provided by this work, designers of MCSoc can implement a secure SDN management paradigm for the communication resources and use their communication management rules into step 2 (path search). The presented framework steps are systemic because it covers functional details from hardware modules up to the OS, evaluating the impacts on the user's task. Due to the hardware/software co-design, the proposed techniques achieved low overheads, overcoming related work, and being feasible to the MCSoc design context.

Future works are towards the following subjects: (i) integration of the SDN-SS with techniques that support security at computation and application admission levels; (ii) exploit the runtime secure *SR* configuration features under faults or successfully attacks of a given *SR*.

## REFERENCES

- [1] S. Singh and R. K. Jha, "A survey on software defined networking: Architecture for next generation network," *J. Netw. Syst. Manage.*, vol. 25, no. 2, pp. 321–374, Apr. 2017.
- [2] R. Sandoval-Arechiga, R. Parra-Michel, J. L. Vazquez-Avila, J. Flores-Troncoso, and S. Ibarra-Delgado, "Software defined networks-on-chip for multi/many-core systems: A performance evaluation," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2016, pp. 129–130.
- [3] A. Kostrzewa, S. Tobuschat, and R. Ernst, "Self-aware network-on-chip control in real-time systems," *IEEE Design Test*, vol. 35, no. 5, pp. 19–27, Oct. 2018.
- [4] M. Ruaro, H. M. Medina, and F. G. Moraes, "SDN-based circuit-switching for many-cores," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2017, pp. 385–390.
- [5] M. Ruaro, H. M. Medina, A. M. Amory, and F. G. Moraes, "Software-defined networking architecture for NoC-based many-cores," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 385–390.
- [6] K. Berestizshesky, G. Even, Y. Fais, and J. Ostrometzky, "SDNoC: Software defined network on a chip," *Microprocessors Microsyst.*, vol. 50, pp. 138–153, May 2017.
- [7] A. Scionti, S. Mazumdar, and A. Portero, "Towards a scalable software defined network-on-chip for next generation cloud," *Sensors*, vol. 18, no. 7, pp. 1–24, 2018.
- [8] S. Ellinidou, G. Sharma, T. Rigas, T. Vanspouwen, O. Markowitch, and J.-M. Dricot, "SSPSoc: A secure SDN-based protocol over MPSoc," *Secur. Commun. Netw.*, vol. 2019, pp. 1–11, Mar. 2019.
- [9] M. Ruaro, N. Velloso, A. Jantsch, and F. G. Moraes, "Distributed SDN architecture for NoC-based many-core SoCs," in *Proc. 13th IEEE/ACM Int. Symp. Netw.-on-Chip*, Oct. 2019, pp. 1–8.
- [10] R. Stefan, A. B. Nejad, and K. Goossens, "Online allocation for contention-free-routing NoCs," in *Proc. Interconnection Netw. Archit. On-Chip, Multi-Chip Workshop (INA-OCMC)*, 2012, pp. 13–16.
- [11] A. Leroy, D. Milojevic, D. Verkest, F. Robert, and F. Catthoor, "Concepts and implementation of spatial division multiplexing for guaranteed throughput in networks-on-chip," *IEEE Trans. Comput.*, vol. 57, no. 9, pp. 1182–1195, Sep. 2008.
- [12] A. Abousamra, A. K. Jones, and R. Melhem, "Proactive circuit allocation in multiplane NoCs," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*, 2013, pp. 35:1–35:10.

- [13] L. Cong, W. Wen, and W. Zhiying, "A configurable, programmable and software-defined network on chip," in *Proc. IEEE Workshop Adv. Res. Technol. Ind. Appl. (WARTIA)*, Sep. 2014, pp. 813–816.
- [14] R. Sandoval-Arechiga, J. L. Vazquez-Avila, R. Parra-Michel, J. Flores-Troncoso, and S. Ibarra-Delgado, "Shifting the network-on-chip paradigm towards a software defined network architecture," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2015, pp. 869–870.
- [15] Y. J. Yoon, N. Concer, M. Petracca, and L. P. Carloni, "Virtual channels and multiple physical networks: Two alternatives to improve NoC performance," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 12, pp. 1906–1919, Dec. 2013.
- [16] S. Liu, A. Jantsch, and Z. Lu, "MultiCS: Circuit switched NoC with multiple sub-networks and sub-channels," *J. Syst. Archit.*, vol. 61, no. 9, pp. 423–434, Oct. 2015.
- [17] W. Quan and A. D. Pimentel, "A hierarchical run-time adaptive resource allocation framework for large-scale MPSoC systems," *Design Autom. Embedded Syst.*, vol. 20, no. 4, pp. 311–339, Dec. 2016.
- [18] T. Kai, X. Xin, and C. Guo, "The secure boot of embedded system based on mobile trusted module," in *Proc. 2nd Int. Conf. Intell. Syst. Design Eng. Appl.*, Jan. 2012, pp. 1331–1334.
- [19] Y. Liu, J. Briones, R. Zhou, and N. Magotra, "Study of secure boot with a FPGA-based IoT device," in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2017, pp. 1053–1056.
- [20] J. Sepulveda, F. Willgerodt, and M. Pehl, "SEPUFSoc: Using PUFs for memory integrity and authentication in multi-processors system-on-chip," in *Proc. Great Lakes Symp. VLSI*, May 2018, pp. 39–44.
- [21] F. O. Hadlock, "A shortest path algorithm for grid graphs," *Networks*, vol. 7, no. 4, pp. 323–334, 1977.
- [22] H. Chen and Y.-W. Chang, *Electronic Design Automation: Synthesis, Verification, and Test*. San Mateo, CA, USA: Morgan Kaufmann, 2009.
- [23] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, "Distributed SDN controller system: A survey on design choice," *Comput. Netw.*, vol. 121, pp. 100–111, Jul. 2017.
- [24] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, "Memphis: A framework for heterogeneous many-core SoCs generation and validation," *Design Autom. Embedded Syst.*, vol. 23, nos. 3–4, p. 103–122, Aug. 2019.
- [25] PUCRS GAPH Group. (2020). *System and Secure Memphis-GitHub*. [Online]. Available: <https://github.com/GaphGroup/MMemphis/tree/secure-sdn>
- [26] S. Rhoads. (2016). *Plasma-Most MIPS I(TM)*. Accessed: Mar. 3, 2020. [Online]. Available: <https://opencores.org/projects/plasma>
- [27] M. Ruaro and F. G. Moraes, "Demystifying the cost of task migration in distributed memory many-core systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.



working and security for many-core systems.

**MARCELO RUARO** was born in Três de Maio, Brazil, in 1988. He received the M.Sc. and Ph.D. degrees in computer science from PUCRS University, Porto Alegre, Brazil, in 2014 and 2018, respectively. He has eight years of research experience in the field of NoC and many-cores SoC architectures and two years of experience in the embedded system industry. He is currently a Post-doctoral Researcher with PUCRS. His primary research interests include software-defined net-



**LUCIANO L. CAIMI** (Member, IEEE) received the M.Sc. degree in electrical engineering from the Federal University of Santa Catarina (UFSC), Florianopolis, Brazil, in 1998, and the Ph.D. degree in computer science from PUCRS University, Porto Alegre, Brazil, in 2019. He is currently an Adjunct Professor with the Federal University of Fronteira Sul (UFFS). His main research interests include multiprocessor systems on chip (MPSoCs), and security for embedded systems.



of VLSI design. His primary research interests include microelectronics, FPGAs, reconfigurable architectures, NoCs, and MPSoCs.

**FERNANDO GEHM MORAES** (Senior Member, IEEE) received the Electrical Engineering and M.Sc. degrees from the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1987 and 1990, respectively, and the Ph.D. degree from the Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier, France, in 1994. He has been a Full Professor with PUCRS, since 2002. He has authored or coauthored 38 peer-refereed journal articles in the field

...