Scientific Research Publishing

Journal of Computer and Communications, 2018, 6, 135-152
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

A Customized Authentication Design for Traffic Hijacking Detection on Hardware-Trojan Infected NoCs

Mubashir Hussain, Hui Guo, Sri Parameswaran

School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
Email: mhussain@cse.unsw.edu.au, huig@cse.unsw.edu.au, sridevan@cse.unsw.edu.au

How to cite this paper: Hussain, M., Guo, H. and Parameswaran, S. (2018) A Customized Authentication Design for Traffic Hijacking Detection on Hardware-Trojan Infected NoCs. *Journal of Computer and Communications*, **6**, 135-152.
https://doi.org/10.4236/jcc.2018.61015

Received: October 19, 2017
Accepted: December 26, 2017
Published: December 29, 2017

Abstract

Traffic hijacking is a common attack perpetrated on networked systems, where attackers eavesdrop on user transactions, manipulate packet data, and divert traffic to illegitimate locations. Similar attacks can also be unleashed in a NoC (Network on Chip) based system where the NoC comes from a third-party vendor and can be engrafted with hardware Trojans. Unlike the attackers on a traditional network, those Trojans are usually small and have limited capacity. This paper targets such a hardware Trojan; Specifically, the Trojan aims to divert traffic packets to unauthorized locations on the NoC. To detect this kind of traffic hijacking, we propose an authentication scheme in which the source and destination addresses are tagged. We develop a custom design for the packet tagging and authentication such that the implementation costs can be greatly reduced. Our experiments on a set of applications show that on average the detection circuitry incurs about 3.37% overhead in area, 2.61% in power, and 0.097% in performance when compared to the baseline design.

Keywords

Packet Hijacking Detection, Hardware Trojan, Network-on-Chip

1. Introduction

With the advance in semiconductor technology, many intellectual property (IP) cores can be integrated on a single chip. A typical example is the system-on-chip (SoC) [1] [2], where multiple processor cores, memory components, I/O interfaces are implemented on one chip and their communications are supported by an on-chip sub-system, called NoC (Network-on-Chip).

To reduce the cost and time-to-market, the SoC designers often use a third-party NoC IP. The third-party IPs may contain hardware Trojans (the malicious components unlawfully inserted into the design) and therefore the system can be exposed to various attacks [3] [4]. Even though there are a few of offline approaches to detect hardware Trojans by the circuit level testing [5] [6], tiny Trojans with a very small footprint can escape such detections and still appear in the final product to perform simple attacks. One simple attack may be hijacking packets to different locations where the data in the packet can be leaked or exploited. For example, some medical surgery equipment may contain a SoC that consists of many controllers to control various supplies (such as blood, oxygen and anesthetic) during a surgery. If a packet containing the command "increase amount" for oxygen, is hijacked to the controller that is for anesthetic, the patient may die of anesthetic overdose. Similarly, in the case of an online payment transaction, if the related packets are stealthily copied to an auditing application on the SoC chip in a mobile device, the user's credential information can be easily leaked.

1.1. Target System and Attack Model

In this paper, we focus on such a system that consists of an untrusted third-party NoC and a set of designer's processing units (PUs), as outlined in **Figure 1(a)**. In this system, each node in the NoC has a router to connect to other nodes and a network interface (NI) for communication with the local PU. The PU hardware and its system software from the designer are trusted, but some of its applications (for example software downloaded from the internet) are untrusted.

We assume that a single hardware Trojan (HT) is present in an arbitrary router of the NoC and the Trojan can be activated and deactivated by either external physical parameters or internal electrical signals. Once activated[1], the Trojan can read and modify the packets passing through its host router. Due to its limited capacity, the Trojan aims to hijack a targeted packet to a different location by altering the packet's destination and/or source address, as illustrated in **Figure 1(b)**, where the packet from node i to node j, $P(i, j)$, is diverted to node k. In addition, we assume that the Trojan may have a software accomplice (sitting on some PU) that can communicate with the Trojan to assist attacks.

1.2. Contributions of the Paper

In order to detect such a packet hijacking, we use a tag based authentication. At the source node, a tag is generated and inserted into the packet. Upon receiving the packet, the destination DU recalculates the tag based on the received packet and compares it with the original tag. If they are different, a hijacked packet is deemed detected, as shown in **Figure 1(c)**.

Our work makes the following contributions:

[1]Here we only focus on the detection of hijacked packets, our proposed detection approach is suitable to varied Trojan activation/deactivation triggers [3].

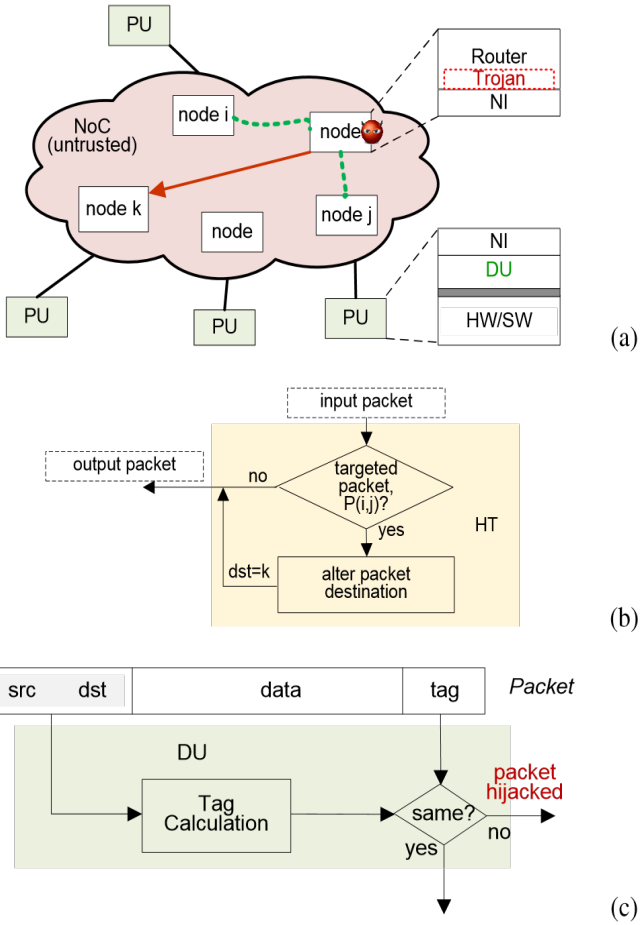

Figure 1. Problem overview. (a) Target system; (b) Packet hijacking attack by Trojan; (c) Packet hijacking detection by detection unit.

- We proposed a novel packet tagging and authentication scheme that has two key features for security:
  - The packet tag is random, dynamic and scrambled with the packet data, making it difficult for the Trojan to alter the tag to elude detection;
  - The packet tagging and the authentication are carried out in the secure detection units (DUs) and both are transparent to the application software so that the software accomplice cannot access tagged packets to perform cryptanalysis on the packet tag.
- We developed a custom design approach for the hardware detection unit. The design is built on the customized packet label size, minimized look-up-table and small block scrambling operations such that the packet space and computing costs caused by the attack detection can be greatly reduced.

1.3. Paper Organization

The rest of the paper is organized as follows. Section 2 briefly reviews the NoC-based security. A short discussion on the existing data authentication schemes is also given in this section. Section 3 provides the overview of our packet tagging and

authentication design; The related custom hardware implementation is presented in Section 4 and the experimental results are given in Section 5. The paper is concluded in Section 6.

2. Related Work

The NoC-based security in SoCs has been actively researched in the past decade. A few of isolation schemes, either software or hardware based, have been used to enhance the system security.

At the software level, different access control approaches to protect a shared memory from unauthorized accesses over the insecure NoC (which can cause the system denial of service (DoS), leak of information, and change of behaviour) have been investigated in [7] [8] [9].

To avoid interference between secure and non-secure applications on the system, [10] partitioned the NoC based on the security level of the applications, while Wassel *et al.* [11] proposed a time-multiplexing NoC to separate communication links between different applications. Sonics uses a SMART Interconnect [12] to protect system integrity and ARM applies what they call TrustZone [13] to separate the normal and secure execution environments.

Similarly, [14] proposed a two-level security wrapper to counter the DoS and masquerade attacks to improve the NoC availability.

There have also some research on preventing attacks by hardware Trojans. In [15] and [16], error detection codes are used to counter data integrity attacks by hardware Trojans on the NoC links, where a Trojan on a link can flip the bit value of the link.

In [17], a runtime auditor is proposed to prevent the bandwidth denial attack by the hardware Trojan. The auditor is based on the transmission latency difference between the original and duplicated transmissions. An attack will be detected if the latency difference is beyond a predefined threshold.

Ancajas in [18] proposed a three-layer security architecture, to prevent information leak by the NoC hardware Trojan that has varied sources of assistance, including side-channels.

Our work focuses on the detection of packet hijacking by the Hardware Trojan that is tiny, has limited computing power, and only attempts to divert the targeted packets to a different location on the NoC. We assume that both source and destination addresses can be altered by the Trojan. When the packet source address is altered, the reply packet can be hijacked. We use a tag-based authentication to identify any changes to the packet source and destination addresses.

There are many ways that a packet can be tagged for authentication. Plenty of approaches exist, such as the cryptographic hash function based [19] [20], the MAC (Message Authentication Code) scheme based [21] [22] [23], and Added Redundancy Explicit Authentication (AREA) [24]. However, those approaches basically focus on the integrity of data. They rarely consider the intention behind the attack.

In this work, we take the nature of packet hijacking attack into account in the packet tagging so that our design can be customized to meet the design requirements with "just enough" resources, which is elaborated in the next sections.

3. Design Overview of Packet Tagging and Authentication

We divide the contents of a packet into source (*src*), destination (*dst*), and the rest as data. The data includes payload data and other information auxiliary for the packet transmission. With the packet hijacking attack, data is supposed to remain unchanged; otherwise, the whole attack may become invalid to the attacker. Therefore, our first packet tagging strategy is mixing the tag with the packet data. If the tag position is not known, it is difficult for the attacker to replace the tag with a guessed value without altering the data bits.

For the tag generation, we start the design with a simple XOR operation:

$$tag = (src \,||\, dst) \oplus K_t \tag{1}$$

where $||$ is the bit string concatenation and K_t is a secret key. It can be seen from this tag design that if the Trojan only changes the *src* and *dst* of the packet, the tag will be changed and the hijacking attack can be readily detected.

However, if key K_t is fixed for all packets of the same (*src*, *dst*), the tag values would be identical. This fixed tag value could then be used to easily identify the tag bits in the data field. Moreover, even if the key is dynamically changed, the tag bit positions can still be identified if the Trojan is able to collect sufficient number of packets of a same data value for a given *(src, dst)*. Therefore, the key used for the tag generation should be dynamic, and tag bit positions need to be further concealed, which leads to our design for the packet tagging, as given in **Algorithm 1**.

For a packet with data ($data_i$) to be sent from source *(src)* to destination *(dst)*, **Algorithm 1** first generates a random number, (L_p), to label the packet (Line 1). L_p is then used to find a dynamic key ($K_t \,||\, K_s$) (Line 2) for the tag generation (Line 3) and tag bit concealment (Line 4 and Line 5). The tag is generated using Equation (1). To conceal the tag bits, the tag is initially inserted into the data field with a predefined and secret bit-position map (m_{tag})(Line 4); the resulting data field $data_t$ is then scrambled (Lines 5) under the control of key (K_s). After the tag bit concealment, the packet label (L_p) is inserted, also with another predefined and secret bit-position map, (m_{Lp}), (Line 6). In the last step

Algorithm 1. Packet tagging at the source.

input: data from processing core: $\boldsymbol{src}, \boldsymbol{dst}, \boldsymbol{data_i}$,bit position maps for tag and packet label: $\boldsymbol{m_{tag}}, \boldsymbol{m_{Lp}}$
output: formed packet to NoC: $\boldsymbol{P(src, dst, data_o)}$

1 $\boldsymbol{L_p} \leftarrow$ getRandomPacketLabel ();
2 $\{\boldsymbol{K_t}||\boldsymbol{K_s}\} \leftarrow$ getKey ($\boldsymbol{L_p}$);
3 $\boldsymbol{tag} \leftarrow$ generateTag($\boldsymbol{src, dst, K_t}$);
4 $\boldsymbol{data_t} \leftarrow$ insertTag($\boldsymbol{tag, data_i, m_{tag}}$);
5 $\boldsymbol{data_s} \leftarrow$ scramblingData($\boldsymbol{K_s, data_t}$);
6 $\boldsymbol{data_o} \leftarrow$ insertLp($\boldsymbol{L_p}$, $\boldsymbol{data_s, m_{Lp}}$);
7 $\boldsymbol{P}$ ($\boldsymbol{src, dst,\ data_o}$) = $\boldsymbol{src||dst||data_o}$;

(Line 7), the data field, ($data_o$), together with the packet source and destination addresses forms the final packet to be transferred over the NoC.

For an illustration, **Figure 2** shows how the final packet data field $data_o$ is formed with the initial 8-bit data (01011110), 4-bit tag (1101, underlined in the figure), and 3-bit L_p (010, italicized).

Based on the packet tagging design, to obtain the tag from a packet for authentication, the destination needs to extract L_p (according to its bit-map, m_{tag}) from the received packet, and use L_p to find the key ($K_t \| K_s$). With K_s , the data field $data_s$ can be restored to its initial position so that the *tag* can be extracted from the original position, as detailed in **Algorithm 2**.

Based on the packet tagging design, to obtain the tag from a packet for authentication, the destination needs to extract L_p (according to its bit-map, m_{tag}) from the received packet, and use L_p to find the key ($K_t \| K_s$). With K_s , the data field $data_s$ can be restored to its initial position so that the *tag* can be extracted from the original position, as detailed in **Algorithm 2**.

Similar to the tag generation in the packet tagging, the *tag* calculation in **Algorithm 2** (Line 5) for the received packet also uses Equation (1). When the calculated tag is the same as the original tag, the packet is deemed valid and the related data is passed to the processing application software; otherwise, the packet is invalid and dropped.

From the packet tagging design, we can see that with the tag scrambling, the tag bit positions in the data field change from packet to packet and appear random to the attacker. The randomness of the tag bit values and positions in turn makes it hard for the attacker to identify the secret L_p bit positions since the L_p value is also random even for packets with a same (src, dst) and fixed data.

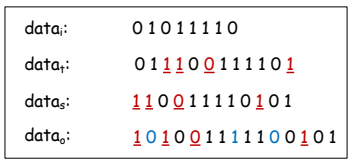

Figure 2. Packet tagging example: tag (1101) is inserted in an 8-bit data, scrambled, and mixed with the packet L_p (010).

Algorithm 2. Packet authentication at the destination.

input: packet from NoC: $\boldsymbol{P}$ ($\boldsymbol{src}, \boldsymbol{dst}$, $\boldsymbol{data_o}$), bit position maps for tag and packet label:$\boldsymbol{m_{tag}}, \boldsymbol{m_{Lp}}$
output: packet validity: $\boldsymbol{vld}$
1 ($\boldsymbol{L_p}, \boldsymbol{data_s}$)← extractLp($\boldsymbol{data_o}, \boldsymbol{m_{Lp}}$);
2 {$\boldsymbol{K_t}||\boldsymbol{K_s}$}←getKey ($\boldsymbol{L_p}$);
3 $\boldsymbol{data_t}$←descrambling ($\boldsymbol{K_s}, \boldsymbol{data_s}$);
4 ($\boldsymbol{tag_{orig}}, \boldsymbol{data_i}$)← extractTag($\boldsymbol{data_t}, \boldsymbol{m_{tag}}$);
5 $\boldsymbol{tag_{calc}}$ = calculateTag($\boldsymbol{src}, \boldsymbol{dst}, \boldsymbol{K_t}$);
6 if$\boldsymbol{tag_{calc}}$ == $\boldsymbol{tag_{orig}}$then
7 $\boldsymbol{vld}$ = **1**;
8 Pass $\boldsymbol{data_i}$ to processing core;
9else
10 $\boldsymbol{vld}$ = **0**;
11 Drop the packet;
12end if

4. Custom Hardware Implementation

As can be seen from the packet tagging, the effectiveness of the authentication is closely related to how the dynamic key is generated and how the tag scrambling is implemented, which are discussed in the next two sub-sections.

4.1. Key Generation

It is desired that a different packet label L_p should have a different key value. Assume the size of L_p is S_{Lp}. We aim for a design that can generate $2^{S_{Lp}}$ unique random keys.

A straightforward way is using a lookup table (**LUT**) to hold pre-generated key values, as illustrated in **Figure 3(a)**, where p is the key size. Each entry in the table holds a key value and L_p points the key to be used. However, the table grows with the key size and the number of key values used, which may incur a large overhead on the storage space. Here we propose a design with a reduced LUT, as outlined in **Figure 3(b)**. The approach to reducing LUT and forming a key is detailed below.

For simplicity, we first assume the key size p is a power of 2. We narrow the table by reducing the table width to p/q bits; q is also a power of 2. To ensure the uniqueness of the key generated from the table, we want the value in each table entry to be unique. Therefore, there are a maximum of $2^{p/q}$ entries. We call each entry in the table a word. A key is formed by a concatenation of q words from the table, hence it is unique.

To obtain q words, we divide the table into q sub-tables. Each sub-table provides a word for the key.

For q sub-tables, and $2^{S_{Lp}}$ p-bit keys, the following conditions should be satisfied:

$$2^{p/q}/q \geq 1 \tag{2}$$

$$(2^{p/q}/q)^q * q! \geq 2^{S_{Lp}} \tag{3}$$

where $q!$ is the number of permutations of q words. The above equations state that q cannot be arbitrarily large. A sub-table should contain at least one entry

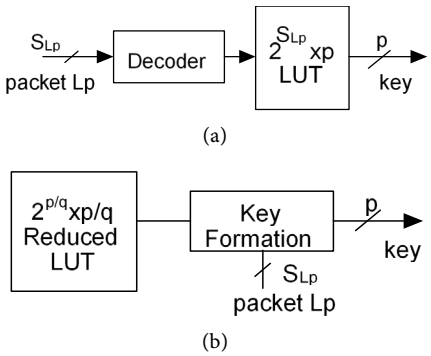

Figure 3. Key generation. (a) Key directly selected from a full LUT; (b) Key generated from a reduced LUT.

(see Equation (2)). Since a sub-table offers $2^{p/q}/q$ words, a maximum of $(2^{p/q}/q)^q * q!$ keys can be formed based on the table. Therefore, the total number of keys required should be no larger than this value (see Equation (3)).

Based on Equations (2) and (3), to reduce the cost we can select a q such that the LUT is small yet sufficient for a required number of unique keys.

If p or q is not of a power of 2, the closest upper power value $2^{\lceil \log p \rceil}$ or $2^{\lceil \log q \rceil}$ will be used to find a reduced LUT. From the reduced LUT, q sub-tables will be adopted for key generation.

Given the reduced LUT table, our design for $2^{S_{Lp}}$ unique keys consists of three stages:

1) Stage one: Word selection (WSL), selecting q words from the sub-tables;

2) Stage two (optional): Word shuffling (WSF), changing the order of selected words;

3) Stage three: Word concatenation (WC), concatenating the shuffled words to form a key.

The first two stages are controlled by the S_{Lp}-bit packet label. The word selection uses different control bits for different sub-tables; therefore, a total of $q * \log(2^{p/q}/q)$ bits from L_p are used. The remaining bits, if there are any, in L_p are then used for word shuffling, where the selected words are partitioned into groups such that words in each group can be shuffled by the available control bits. As an example, shuffling four words, ($x0 \sim x3$), with two control bits, ab, (denoted by the symbol shown in **Figure 4(a)**), is given in **Figure 4(b)**, where the different control value for each 4-to-1 multiplexer will select a different input. For instance, when $ab = 01$, $y0 = x1$, $y1 = x0$, $y2 = x3$, $y3 = x2$; Therefore, inputs $(x0, x1, x2, x3)$ are shuffled into $(x1, x0, x3, x2)$.

To demonstrate the reduced-LUT based approach, here we consider three designs for: 256 16-bit keys, 256 20-bit keys, and 512 16-bit keys, respectively. For the three designs, according to Equations (2) and (3), the largest q for the reduced LUT is 4 and the table size is $2^4 \times 4$ bits. (Notice that without the LUT reduction, the table sizes for the three designs would be $2^8 \times 16$, $2^8 \times 20$ and $2^9 \times 16$ bits, correspondingly.)

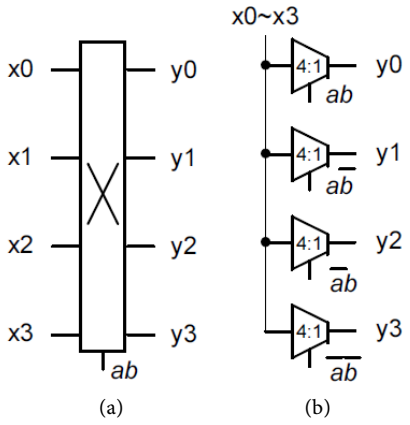

Figure 4. Four-word shuffle. (a) Symbol; (b) Circuit design.

The design for 256 16-bit unique keys is shown in **Figure 5(a)**, where the ($2^4 \times 4$)-LUT are divided into four sub-tables, each having four words. The 8 bits of L_p (I_{1-8}) are all used for word selection. No control bits are left for shuffling. The selected four words are, therefore, directly concatenated to form a key.

For the 20-bit keys, since 20 is not a power of 2 and its closest upper power value is 32, we first divide the ($2^4 \times 4$) LUT into eight sub-tables for 32-bit keys; each sub-table now contains two words. From the table, we choose five sub-tables for 20-bit key generation, as shown in **Figure 5(b)**, where five control bits select five words from the sub-tables. For the rest of three control bits, the selected words are partitioned into two groups. The four words in one group are shuffled under two control bits and the group is then shuffled with the second group under one control bit.

Similarly, we can come up a design with the same 4-bit LUT for 512 keys, as shown in **Figure 5(c)** where the 9-bit L_p is used. The selected four words are partitioned into two groups and the two word groups are shuffled by one control bit.

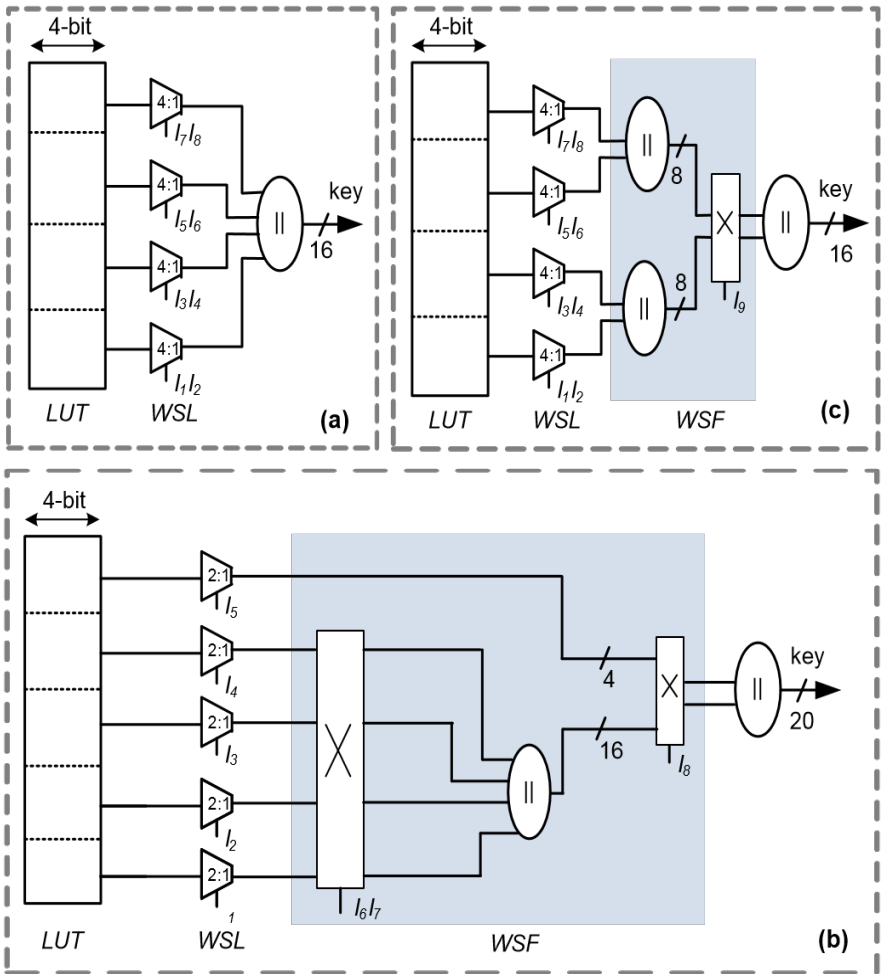

Figure 5. Example of key generation design for (a) 256 16-bit keys; (b) 256 20-bit keys; (c) 512 16-bit keys.

As can be seen from the example, with the increase of unique keys, the hardware overhead is only slightly increased. But more control bits (*i.e.*, L_p) are required. A one-bit increase in L_p means the packet will carry one-bit less payload data. For security, on the other hand, we want a large L_p size to avoid tag collisions among the packets of the same (*src*, *dst*). Therefore, a trade-off should be made between the security and the overhead of the packet space consumption.

Here we aim to improve the uniqueness of keys for a node within an execution time window, which is related to the traffic load on the network. For a healthy system, the network should not be saturated. For a given a NoC of γ nodes, assume that the average packet injection rate without causing the network saturation is β and the average packet latency is τ. The average traffic per second on the NoC is $\beta\tau\gamma$. We determine the L_p size based on the following equation:

$$S_{Lp} = \log(\beta\tau\gamma) \tag{4}$$

4.2. Data Field Scrambling

Data field scrambling is to make tag bit positions in the data field appear random to the attacker. Since the tag value is dynamic and random, randomly rotating the data field for each packet will make each bit value in the field random, hence hiding the tag position.

We use key K_s (see **Algorithm 1**) to control the rotation. A simple design is rotating the date field by K_s bits. The size of K_s should be at least $S_{Ks} = \log S_d$ for a full range rotation (where S_d is the data field size). However, this design has two major problems: one, it just creates up to S_d values for a given tag position; the tag can only be shuffled and hidden in this value space; and two, the cost of the rotation increases exponentially with the data size.

To see how small block rotation helps to reduce the design cost, we implemented the designs for 16-, 32-, 64-, and 128-bit data rotations based on 8-bit blocks. **Table 1** shows the related savings on the area, power, and execution delay as compared to the related full data rotation designs. As can be seen from the experiments, the savings from the small-block-based designs are significant.

Therefore, here we propose to rotate the data on small blocks such that a large shuffle space can be created and, at the same time, the rotation cost can also be reduced.

Table 1. Cost saving of block based design over full rotation designs.

Data Size	Savings		
	Area	Power	Delay
16	21.22%	11.49%	13.0%
32	41.37%	17.17%	32.6%
64	47.68%	25.29%	48.4%
128	50.61%	28.92%	52.3%

Assume the block size is S_b, we divide the data field into $n = S_d / S_b$ blocks. The rotation of each block requires $\log S_b$ control bits. Given key K_s, we can extract S_{Ks} random control values from the bit stream ring formed be K_s, as demonstrated in **Figure 6**, where the last key bit is adjacent to the first bit. A control value C_i is a bit segment starting from bit i in the ring. We, then, group the n blocks into S_{Ks} groups of a similar size. The data field size may not be a multiple of the block size. In this case, the smaller block of the size less than S_b will form a separate group. A bit segment $(C_0, C_1, \cdots, C_{S_{Ks}-1})$ from the control ring is selected to control block rotation in a group.

As a demonstration, **Figure 7(a)** shows a design for 108-bit data with a block size of 8 bits and the control of 7 bits (that is used in our experiment). The data field is divided into 14 blocks $(b_1, b_2, \cdots, b_{14})$ with the last block of 4 bits, which are then partitioned into 7 groups. Three bits control the rotation of each group. **Figure 7(b)** shows blocks (b_1 and b_2) in the first group are rotate-left shifted 1bit when the control value for the group is 001 $(C_0 = 001)$.

The block based design allows the tag bits to be concealed by up to $S_b^{S_{Ks}}$ values, larger than the space (up to $2^{S_{Ks}}$ values) provided by the design with the full range rotation.

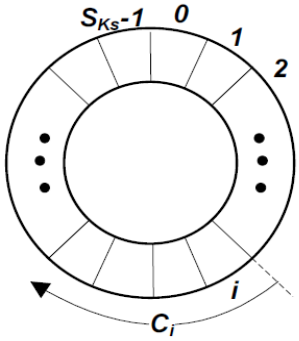

Figure 6. Control bit ring for data field shuffle.

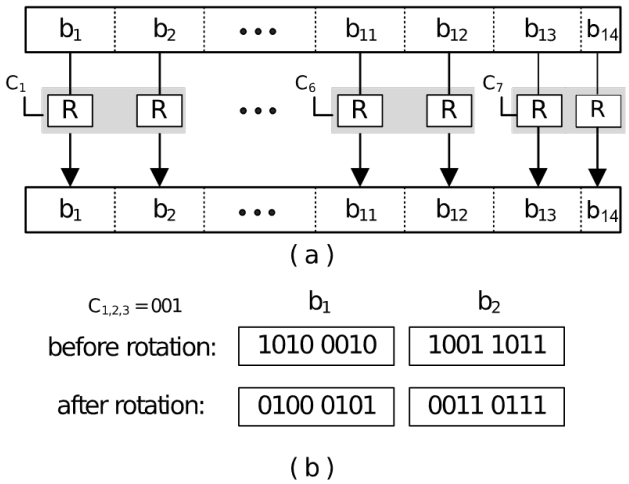

Figure 7. Example of scrambling design (108-bit data). (a) Overview; (b) Block rotation in group 1.

5. Experimental Evaluation and Results

To evaluate our design, we built a simulation platform, as shown in **Figure 8**. We use the 2D-mesh NoC model developed by [25] as our baseline NoC. Each router in the network performs five basic operations: link traversal (LT) to transfer a packet from the output of one router to the input of another router, buffer write (BW) to save incoming packets, routing computation (RC) to determine the switching direction for a packet, switch allocation (SA) to arbitrate multiple packet switching in the router, and switching traversal (ST) to move a packet from the incoming buffer to the router's output.

The five operations are pipelined into 3 stages. In addition, the XY routing algorithm, the wormhole switching scheme and the matrix arbitration, commonly used in the NoC designs, are also implemented in the baseline model.

A cycle-accurate NoC simulator, Booksim 2.0 [26], is used to run application traces collected by Netrace 1.0 [27]. We modify the simulator to simulate the Trojan attack and the attack detection over the NoC for each application.

The hardware models are synthesized with Synopsys Design Compiler [28] based on the TSMCs 65 nm cell library [29] for cost estimation. The experiment results are presented below.

5.1. Determining Packet Label Size

With our design, we first need to decide the packet label size. In our experiments, we use 8x8-mesh network with the packet size of 128 bits. The packet source and destination addresses each require 6 bits so the tag size is $6+6=12$ bits.

We run the application traces with Booksim to simulate their network performance on the given NoC. **Table 2** shows the average packet latency in terms of clock cycles (*cc*) and the traffic injection rate in packets per clock cycle *(ppc)*, for each application. Based on Equation (4), we find the label size, as shown in the last column in the table. For the set of applications, we choose the largest size (namely, 8 bits) in our designs for the Trojan and packet hijacking detection unit.

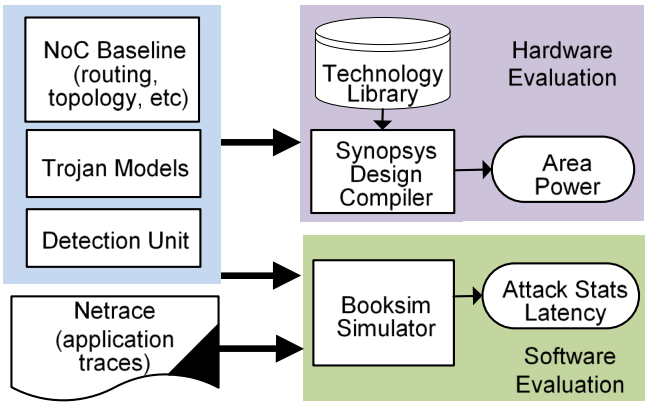

Figure 8. Experimental setup.

Table 2. Average injection rate, latency and packet label size.

Applications (abbr.)	Injection Rate (ppc)	Latency (cc)	SLp (bits)
Blackscholes (blks)	0.015192	31.3914	5
Bodytrack (bdyt)	0.084288	37.5532	8
Canneal (canl)	0.016099	32.3142	5
Dedup (dup)	0.079224	32.5551	7
Fluidanimate (flud)	0.018139	33.0537	5
Swaptions (swap)	0.176881	30.6503	8
Vips (vps)	0.061652	33.3856	7
X264	0.009839	32.2271	4

5.2. Costs of Hardware Trojan

We implement two Trojans (HT1 and HT2) that have following functionality and capability in playing with packet tags:

- HT1: using randomly guessed tag value and bit positions during an attack.
- HT2: working with the software accomplice in an attempt to uncover the tag bit positions before hijacking takes place. To do that, the software accomplice repeatedly sends a constant data to the Trojan. By comparing the received packets from the accomplice, the Trojan may be able to identify some tag bit positions.

The area, power and delay overheads of the two hardware Trojans are given in Columns 3 & 5 in **Table 3**. The relative values in percentage (labelled *r*) as compared the costs of a baseline router (see Column 2, where the delay is the longest path delay, which determines the clock cycle time) are also given in the table. The two Trojans consume less than 3% area and 0.4% power of the single router, and incur less than one clock cycle delay (27% of the router's clock cycle time). **Figure 9** shows the area and power consumption of each Trojan relative to the NoC size. For a single Trojan inserted in a large network of many nodes, the overheads become ignorable. With our 8 × 8-mesh network, the footprints of HT1 and HT2 are, respectively, just 0.024% and 0.043% of the overall NoC. The performance overhead may also very unlikely be detected at the circuit testing stage without knowing how to activate the Trojan.

5.3. Security Evaluation

We evaluate the security of our design in terms of the detection rate. To see the effectiveness of our tag bit scrambling, we also investigate the attack success rate and the data invalid rate. The average data invalid rate, detection rate and attack success rate under two detection designs (with and without tag bit scrambling) are given in **Table 4**.

From Column 2 of **Table 4**, we can see that even though there are some HT1 attacks that can pass through the authentication, chances that the packet data are made invalid are very high; the average probability is about 99.35%. It can also

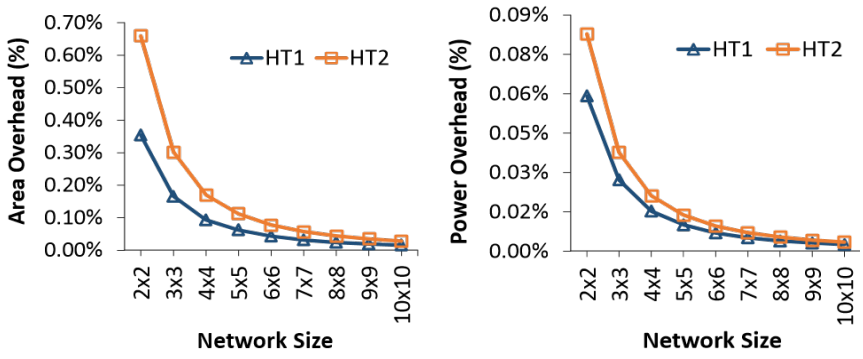

Figure 9. Trojan overhead vs network size.

Table 3. Overheads of HT1 and HT2.

Metric	Baseline Router	HT1		HT2	
			r (%)		r (%)
Area (um^2)	83962	1313	1.56	2362	2.81
Power (uW)	53920	136	0.25	187	0.35
Delay (ns)	0.56	0.07	12.5	0.15	26.8

Table 4. Security evaluation against HT1 and HT2 attacks.

Metric	without scrambling		with scrambling	
	HT1	HT2	HT1	HT2
Data Invalid rate (%)	99.3509	0	99.3857	98.9684
Detection rate (%)	99.9846	99.9740	99.9870	99.9753
Attack success rate (%)	0.0006	0.0260	0.0005	0.0009

be seen that the success rate is not exactly the complement of the detection rate. If an attack has escaped the authentication but the data of the hijacked packet has been altered, the attack cannot be successful. However, if the HT2 Trojan is used, the data invalid rate is reduced to 0, as shown in Column 3 in the table. This is because the HT2 Trojan can identify the static tag bit positions and alter the tag on the correct bits in the data field. We also observe that with HT2, the attack success rate is increased and accordingly the attack detection rate is reduced due to the increased tag collision frequency. But if the mixed tag and data field is dynamically scrambled, the security of the design is greatly improved, as shown in the last two columns in the table, where 99.9% attacks can be detected and for the rest of attacks that have gone undetected, a majority of them are still unsuccessful due to the invalid data they contain. Most importantly, the low attack success rate (0.0005% from HT1and 0.0009% from HT2) is accompanied with a high data invalid rate (99.38% from HT1 and 98.96% from HT2).

5.4. Overhead of Detection Unit

With 8 × 8-mesh network, the address of 12 bits, and the L_p size of 8 bits ob-

tained in Section 5.1., the remaining 108-bit data field in the 128-bit packet will be used for data and tag. Therefore, the key size is 20 bits (the sum of the address size and L_p size). The design shown in **Figure 5(b)** is used for key generation and the design given in **Figure 7** (based on the 8-bit Barrel Shifter) is applied for data field scrambling in our experiments.

To see how effective is the cost saving on the reduced LUT and block-based scrambling, we investigate the design with full LUT (see **Figure 3(a)**) and with full data field scrambling. The relative costs as compared to the baseline router design for the three designs are given in Columns 2 - 4 of **Table 5**. As can be seen from the table, our DU design incurs very small overheads, 3.37% on area and 2.61% on power. However, without the LUT reduction, the costs would be very high (increased to 76.90% and 77.86% in the area and power); the costs would also be higher if the block-based scrambling is not used.

For the performance overhead, we first measure the delay of DU with Synopsys Design Compiler. The delays are then incorporated into the BookSim simulation, based on which we obtain the execution time for each application. **Figure 10** shows the normalized execution time with and without the detection unit; on average, only 0.097% (as given in the rightmost group, AVG) execution delay is incurred.

6. Conclusions

In this paper, we proposed a tag-based authentication design to detect the packet hijacking by a hardware Trojan that is embedded in a third-party NoC. Our design incorporates two customization techniques: one, the packet tagging and authentication that are tailored to the packet hijacking attack, with the tag mixed in the packet data so that the tag bit positions are hard to be identified by the

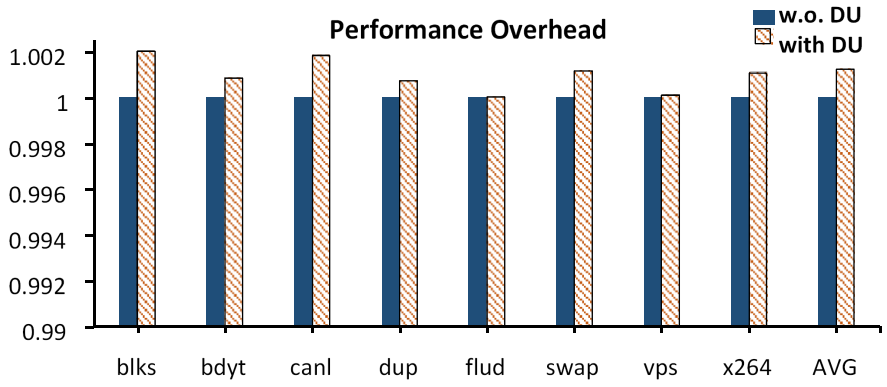

Figure 10. Normalized execution time.

Table 5. Area and power overheads of DU.

Metric	With full LUT rel. (%)	With full scram (%)	DU rel (%)
Area (um^2)	76.90	5.08	3.37
Power (uW)	77.86	5.70	2.61

attacker; and two, the tag size is determined by the NoC size and the packet label size is customized to the applications so that the packet space consumption incurred by the detection is reduced. To further reduce the design overhead, we introduced a lookup table based design for dynamic-tag generation and a block based data field rotation for tag scrambling. Our experiments on an 8 × 8-mesh network show that the detection unit incurs about 3.37% area, 2.61% power, and 0.097% performance overhead as compared to the baseline design.

It must be pointed out that like other security designs, our design, especially the tag bit position, can be cracked by an exhaustive brute-force search. However, the brute-force search often requires large computing and storage supports. Our proposed scheme targets the small Trojan that lacks sufficient resources for such an expensive search and its software accomplice also has no opportunities to decode the packet tag since the detection unit has filtered out the tagging information for each packet before it reaches to the destination software.

References

[1] Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K. and Chang, K. (1996) The Case for a Single-Chip Multiprocessor. *ACM Sigplan Notices*, **31**, 2-11. https://doi.org/10.1145/248209.237140

[2] Maeurer, T.R. and Shippy, D. (2005) Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, **49**, 589-604. https://doi.org/10.1147/rd.494.0589

[3] Bhunia, S., Hsiao, M.S., Banga, M. and Narasimhan, S. (2014) Hardware Trojan Attacks: Threat Analysis and Countermeasures. *Proceedings of the IEEE*, **102**, 1229-1247. https://doi.org/10.1109/JPROC.2014.2334493

[4] Tehranipoor, M. and Koushanfar, F. (2010) A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Design and Test of Computers*, **27**, 10-25. https://doi.org/10.1109/MDT.2010.7

[5] Agrawal, D., Baktir, S., Karakoyunlu, D., Rohatgi, P. and Sunar, B. (2007) Trojan Detection Using Ic Fingerprinting. *IEEE Symposium on Security and Privacy*, 296-310. https://doi.org/10.1109/SP.2007.36

[6] Jin, Y. and Makris, Y. (2008) Hardware Trojan Detection Using Path Delay Fingerprint. *IEEE International Workshop on Hardware-Oriented Security and Trust*, HOST 2008, 51-57.

[7] Diguet, J.-P., Evain, S., Vaslin, R., Gogniat, G. and Juin, E. (2007) Noc-Centric Security of Reconfigurable Soc. *Proceedings of the* 1*st International Symposium on Networks-on-Chip*, 223-232.

[8] Fiorin, L., Palermo, G., Lukovic, S., Catalano, V. and Silvano, C. (2008) Secure Memory Accesses on Networks-on-Chip. *IEEE Transactions on Computers*, **57**, 1216-1229. https://doi.org/10.1109/TC.2008.69

[9] Porquet, J., Greiner, A. and Schwarz, C. (2011) Noc-Mpu: A Secure Architecture for Flexible Co-Hosting on Shared Memory Mpsocs. *IEEE Design, Automation & Test in Europe*, 1-4. https://doi.org/10.1109/DATE.2011.5763291

[10] Sepulveda, J., Pires, R., Gogniat, G., Chau, W.J. and Strum, M. (2012) Qoss Hierarchical Noc-Based Architecture for Mpsoc Dynamic Protection. *International Journal of Reconfigurable Computing*, **2012**, 3. https://doi.org/10.1155/2012/578363

[11] Wassel, H.M.G., Gao, Y., Oberg, J.K., Huffmire, T., Kastner, R., Chong, F.T. and Sherwood, T. (2014) Networks on Chip with Provable Security Properties. *IEEE Micro,* **34**, 57-68. https://doi.org/10.1109/MM.2014.46

[12] SonicsMX Smart Interconnect Datasheet. http://www.sonicsinc.com

[13] Alves, T. and Felton, D. (2004) Trustzone: Integrated Hardware and Software Security. ARM White Paper, 3.

[14] Baron, S., Wangham, M.S. and Zeferino, C.A. (2013) Security Mechanisms to Improve the Availability of a Network-on-Chip. *IEEE International Conference on Electronics, Circuits, and Systems*, 609-612.

[15] Yu, Q. and Frey, J. (2013) Exploiting Error Control Approaches for Hardware Trojans on Network-on-Chip Links. *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (*DFTS*), 266-271.

[16] Boraten, T. and Kodi, A.K. (2016) Packet Security with Path Sensitization for Nocs. *IEEE Design*, *Automation & Test in Europe*, 1136-1139. https://doi.org/10.3850/9783981537079_0180

[17] Rajesh, J.S., Ancajas, D.M., Chakraborty, K. and Roy, S. (2015) Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip. *Proceedings of the* 9*th International Symposium on Networks-on-Chip*, 8.

[18] Ancajas, D.M., Chakraborty, K. and Roy, S. (2014) Fort-Nocs: Mitigating the Threat of a Compromised Noc. *Proceedings of the* 51*st Annual Design Automation Conference*, 1-6. https://doi.org/10.1145/2593069.2593144

[19] Peterson, W.W. and Brown, D.T. (1961) Cyclic Codes for Error Detection. *Proceedings of the IRE*, 228-235. https://doi.org/10.1109/JRPROC.1961.287814

[20] Bellare, M., Canetti, R. and Krawczyk, H. (1996) Keying Hash Functions for Message Authentication. *Annual International Cryptology Conference*, Springer.

[21] Rogers, A. and Milenkovic, A. (2009) Security Extensions for Integrity and Confidentiality in Embedded Processors. *Microprocess. Microsyst* (*Netherlands*), **33**, 398-414.

[22] Yan, C., Rogers, B., Englender, D., Solihin, D. and Prvulovic, M. (2006) Improving Cost, Performance, and Security of Memory Encryption and Authentication. 33*rd Int. Symposium on Computer Architecture*. https://doi.org/10.1145/1150019.1136502

[23] Hong, M., Guo, H. and Hu, S.X. (2012) A Cost-Effective Tag Design for Memory Data Authentication in Embedded Systems. *Proceedings of the* 2012 *International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (*CASES* 2012), 17-26. https://doi.org/10.1145/2380403.2380414

[24] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M. and Martinez, A. (2010) Block-Level Added Redundancy Explicit Authentication for Parallelized Encryption and Integrity Checking of Processor-Memory Transactions. *Transactions on Computational Science X. Special Issue on Security in Computing*, *Part I*, **6340**, 231-260.

[25] Bokhari, H., Javaid, H., Shafique, M., Henkel, J. and Parameswaran, S. (2015) Supernet: Multimode Interconnect Architecture for Manycore Chips. *Proceedings of the* 52*nd Annual Design Automation Conference*, 85. https://doi.org/10.1145/2744769.2744912

[26] Jiang, N., Balfour, J., Becker, D.U., Towles, B., Dally, W.J., Michelogiannakis, G. and Kim, J. (2013) A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *IEEE International Symposium on Performance Analysis of Systems and Software*,

86-96. https://doi.org/10.1109/ISPASS.2013.6557149

[27] Hestness, J. and Keckler, S.W. (2011) Netrace: Dependency-Tracking Traces for Efficient Network-on-Chip Experimentation. The University of Texas at Austin, Dept. of Computer Science, Tech. Rep.

[28] Synopsys Design Compiler. http://www.synopsys.com.

[29] TSMC 65nm GP Standard Cell Libraries-tcbn65gplus. https://www.cmc.ca/en/whatweoffer/products/cmc-00200-01411.aspx