A Comprehensive Approach to MPSoC Security

Achieving Network-on-Chip security: a hierarchical, multi-agent approach

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera Italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Slobodan Lukovic

under the supervision of

Prof. Mariagiovanna Sami

November 2012

Dissertation Committee

Prof. Laura Pozzi	University of Lugano
Prof. Evanthia Papadopoulou	Univeristy of Lugano
Prof. Miroslaw Malek	Humboldt University, Berlin, Germany
Dr. Guido Marco Bertoni	ST Microelectronics, Italy

Dissertation accepted on 13 November 2012

Research Advisor	PhD Program Director
Prof. Mariagiovanna Sami	**Prof. Antonio Carzaniga**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Slobodan Lukovic
Lugano, 13 November 2012

To my family

Scio me nihil scire.

Socrates

Abstract

Multiprocessor Systems-on-Chip (MPSoCs) are pervading our lives, acquiring ever increasing relevance in a large number of applications, including even safety-critical ones. MPSoCs, are becoming increasingly complex and heterogeneous; the Networks on Chip (NoC) paradigm has been introduced to support scalable on-chip communication, and (in some cases) even with reconfigurability support. The increased complexity as well as the networking approach in turn make security aspects more critical.

In this work we propose and implement a hierarchical multi-agent approach providing solutions to secure NoC based MPSoCs at different levels of design. We develop a flexible, scalable and modular structure that integrates protection of different elements in the MPSoC (e.g. memory, processors) from different attack scenarios. Rather than focusing on protection strategies specifically devised for an individual attack or a particular core, this work aims at providing a comprehensive, system-level protection strategy: this constitutes its main methodological contribution. We prove feasibility of the concepts via prototype realization in FPGA technology.

Acknowledgements

This doctoral dissertation would not have come to a successful completion, without the help I received from many people that have helped me over my challenging but interesting Ph.D. adventure. Professor Mariagiovanna Sami, my supervisor, is the first person that I would like to thank, without whom I do not think I would have been where I am now. She has been a true friend and a supporter. I am truly indebted to her for insightful guidance over my candidature.

Next I would like to acknowledge the support and patience of my family members. You have supported me in the most difficult times and believed in me even when I did not believe in myself.

My work could not be efficient and successful without great help of my environment. I'd like to thank to all ALaRI staff for collaboration in different aspects of my research. In particular I'd like to mention my officemates and at the same time travel-mates and flat-mates in some cases, who have helped me grow in areas not exclusively related to my core research. I'd certainly never would be able to structure, organize and write my papers in such a good manner without great guidance of Leandro Fiorin and I would certainly never perform so well in diplomacy and other 'soft-skills' without lessons learnt from my great Sicilian friend Anotnio Taddeo. Finally, I'd like to thank Igor Kaitovic, the youngest member but great asset to our team, for many advices, hints and tips in various areas. Still, it is worth mentioning here Umberto Bondi, ALaRI program manager who has never restricted me from traveling to conferences regardless the location. I'd also like to thank members of secretary office for their patience and understanding in resolving sometimes very complicated administrative situations. I would like to thank my former students Paolo Pezzino, Nikolaos Christianos and Anurbav Srinavasta who have greatly contributed to some of my publications. Their efforts are embedded in my thesis as well as their friendship to my memory.

At the end, I'd like to thank my Ph.D. thesis committee as well as all those unknown reviewers of my papers who have pointed out many inconsistence and also sometimes sparked good ideas for my work. They have provided me with valuable feedback on my work, and helped in improving it.

Contents

Figures

Tables

Chapter 1

Introduction and Motivation

In this Chapter we provide an introduction and motivation for the submitted work. First of all, basic concepts concerning Multiprocessor Systems-on-Chip (MPSoCs) and Networks-on-Chips - of fundamental importance for the present thesis - are presented. An overview of NoC-based MPSoCs is given afterwards, introducing at the same time the security challenges such systems face. The specific problem addressed and the desired goals are briefly exposed. Finally, the outline of the original contributions of the proposed solutions is given and their relevance is discussed.

The use of MultiProcessor Systems-on-Chips (MPSoCs) is constantly increasing, with reference to both number of fields where they are utilized in and diversity of applications involved. In particular these systems are widely used for signal processing, packet processing in computer networks, multimedia processing, in cell-phone processors (see, e.g. Wolf et al. [2008]). All these applications in turn involve networking and Internet enabled access. Such developments are posing many novel design challenges (as stated by Martin [2006]). One of the notable trends in MPSoC design is constituted by shifting towards communication centric design (as shown by D. [2000]; Ogras et al. [2005]; Henkel and Wolf [2004]; Grecu et al. [2004]).

With increased exposure of MPSoCs, we are experiencing an ever growing number of attack techniques: a fact that brings additional challenges for system designers.

Security risks caused by rapid penetration of MPSoCs in all segments of our lives on the one side and increasing concern about the evolution of malicious attack techniques on the other hand have created a global sensitivity to security in MPSoC design, as well as an interest in finding adequate protection solutions. Moreover, modern complex and sensitive embedded systems involve increasing

numbers of actors and technologies, a fact which requires modular integration of reliable and secure components. Therefore, there is a necessity to devise holistic approaches to security protection strategies. Such strategies would consider different aspects of system design and a variety of attack scenarios in order not only to protect from 'infection' but also to prevent the propagation of attacks through the system. Aligned with such approach we consider a comprehensive protection strategy which provides solid ground for system defense combining both distributed (i.e. the protection of individual cores) and centralized protection approaches (i.e. system level). The proposed security framework is built as a flexible, modular and scalable structure that incorporates different attack specific protection methods and integrates them in a system-wide hierarchical security system.

In the development of the solution we rely on Network-on-Chip communication architecture extending its components to host and support security enhancements.

1.1 Multi-Processor Systems-on-Chip

The steady technological evolution has enabled the adoption of many advanced architectural concepts and the integration of increasing numbers of cores onto a single chip. Considering in particular processing architectures, the development of the System-on-Chip (SoC) concept supports evolution towards multiprocessor-based and reconfigurable design. In turn, this has led to the Multi-Processor Systems-on-Chips (MPSoCs) solutions which group multiple standard CPUs together with customized IPs, DSPs, input-output interfaces and other hardware subsystems in the same chip. In particular, when embedded systems are considered, an MPSoC is not simply a traditional multiprocessor shrunk to a single chip but rather an innovative solution that has been designed to fulfill the unique requirements of embedded applications (see e.g. Wolf et al. [2008]).

Such scenario in turn requires innovative system design techniques, new programming models and many other novel strategies to support rapid development of many-core platforms and of the applications using them (Wolf et al. [2008]; Martin [2006]). At the same time, the increasing complexity that will be required in next generation's SoCs pushes designers to research new on-chip communication solutions due to limitations (in particular low scalability) of existing bus-based interconnecting means (Benini and De Micheli [2002]; Dally and Brian [2001]).

In summary, researchers have identified as follows the most important issues

regarding efficient MPSoC design (Martin [2006]; Patel et al. [2010]):

- The number and configuration(s) of processors required for the application. How homogeneous should the architecture be, versus how heterogeneous?
- Interprocessor communication - choosing the right communication solution, taking into account emerging network on chip approaches
- Concurrency, synchronization, control and programming model(s). In many instances, multiple models will be appropriate
- Memory hierarchy, types, amounts, and access methods, along with the capacity of estimating with acceptable accuracy the required latency
- Power reduction and low energy consumption
- Application partitioning, use of appropriate APIs and communications models, and associated design space exploration
- Design and platform scalability
- Security and reliability of architectures especially due to the increased role they play in modern society and to the growing attention payed by the criminal community to these systems Diguet et al. [2007]; Ahmad and Arslan [2005]; Wolf et al. [2008]; Zhou and Wu. T. snd Wu [2009]

In our work we focus on security aspects with special regard to Network-on-Chip architectures (refer to Section 2.1) as communication medium among MPSoC components. Our approach to protection of these systems relies mostly on three basic ideas:

- Extension of NoC components to support security related services
- Adoption of multi-agent system strategies when designing the protecting architecture
- Integration of different security aspects into one system level protection mechanism

These methods represent an original contribution of this work to MPSoC security. The validation of the proposed solutions is performed via design of demonstrators making use of FPGA technology.

1.2 Network-on-Chip based Multi-Processor Systems-on-Chip

As previously stated, increased chip density has created many new design challenges. Considering in particular on-chip communications, global wires, connecting many different functional units, are likely to have propagation delays exceeding the clock period, moreover, the arbitration process among growing numbers of units on the bus becomes very complex. The Network-on-Chips (NoCs) concept (Benini and De Micheli [2002]; Dally and Brian [2001]) represents a promising solution to these issues. NoC has been proven as a solid interconnection strategy that brings reliable, efficient, scalable and fast inter-core communication, capable of to providing a satisfactory answer to a number of issues in MPSoC design, including scalability, arbitration problems, power management etc. Moreover, scalable and modular NoC design facilitates porting at system level a number of advanced concepts not exclusively related to communications such as system monitoring, security, fault tolerance etc.

We focus on MPSoCs architectures based on the Network-on-Chip (NoC) paradigm, identifying their peculiar challenges insofar as security is concerned and developing solutions as well as a complete approach to meet such challenges. NoC-based MPSoCs are basically composed of three kinds of cores:

- Processing cores (e.g. microprocessors, DSPs etc.)
- Data storage elements (i.e. shared memories)
- Communication elements (Routers and Network Interfaces)

The work presented here takes such composition of the system as its reference architectural template. The basic system structure is presented in Figure 1.1.

1.3 Security vulnerabilities and protection strategies for NoC based Multi-Processor Systems-on-Chip

Increasing complexity of MPSoCs and increased networking possibilities make systems more vulnerable to different kinds of attacks (Diguet et al. [2007]; Ahmad and Arslan [2005]; Wolf et al. [2008]; Zhou and Wu. T. snd Wu [2009]). This problem acquires particular relevance as MPSoCs find increasing application in critical systems (even mission-critical or safety-critical ones); this in turn

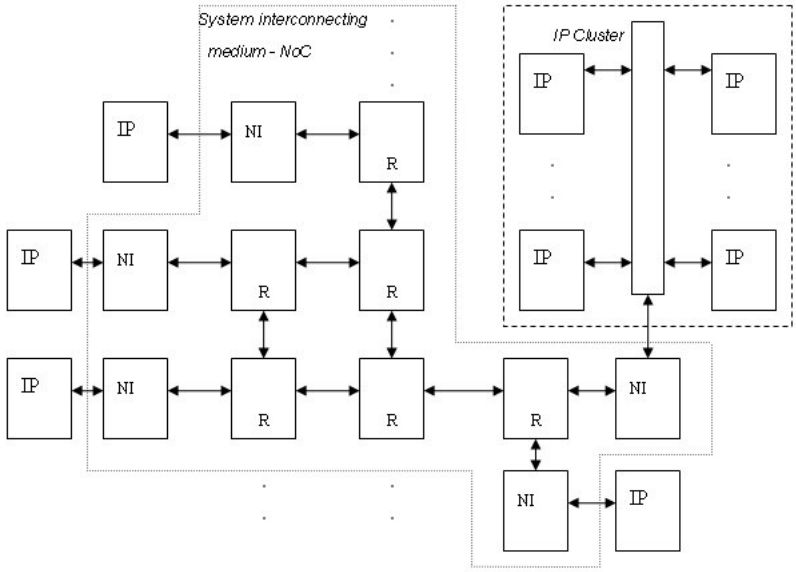

Figure 1.1. Network-on-Chip based Multi-Processor Systems-on-Chip - general structure

makes security aspects more important. On the other hand, we experience rapid growth of attack techniques and increasing flexibility of malicious software.

As a consequence, there is increased need of security-aware design solutions, in particular in the design of reconfigurable SoCs that by their nature are even more vulnerable (attacks can target not only the operation but even the very configuration of the device). However, even though research on NoC-related topics has been an emerging area of interest, security issues in this field have been often shadowed by other topics and have not been explored to the same extent, being only recently addressed by the research community (Diguet et al. [2007]; Fiorin et al. [2008]).

The presence on the chip of different kinds of data-processing cores as well as of various storage elements constitutes a common ground for a number of attacks. A variety of attack models have been detected so far as shown in Section 4.2 and accordingly a number of techniques have been devised to fight them as presented in Chapter 2. However, most of these protection strategies consider only specific types of attacks or specific architectures - a fact that represents a major obstacle for their wide adoption. Therefore, a need for comprehensive approach to security design emerges: the aim of this work is to develop such an approach.

1.4 Problem, objectives and contributions of the thesis

Integrating and coordinating all security-related aspects in an MPSoC requires flexible (i.e. modular and scalable) solutions to be devised. Rather than targeting ad-hoc solutions for specific system architectures, an approach valid for the widest spectrum of NoC-based architectures is needed. Developing of an efficient system-level protection mechanism for MPSoCs is the main challenge we are tackling. For this reason we propose a security framework based on a hierarchical multi-agent structure, capable of being specialized for a large variety of architectures and of attacks.

The proposed solution relies on the implementation of a multi-agent system, based on a hierarchy of agents: at the lowest level, there are - *Local Security Agents* (LSAs) employed locally on the individual cores and integrated in a hierarchical system by means of *Cluster Security Agents*, that are in turn coordinated by a *Central Security Agent* - CSA. The system architecture is shown in Figure 1.2. It can be noted from this figure that LSAs represent logical encapsulations of locally deployed protection mechanisms (that actually are Attack Specific Agents - ASAs). The solution is highly modular, so that ASAs can be individually devised and optimized without requiring an overall re-design.

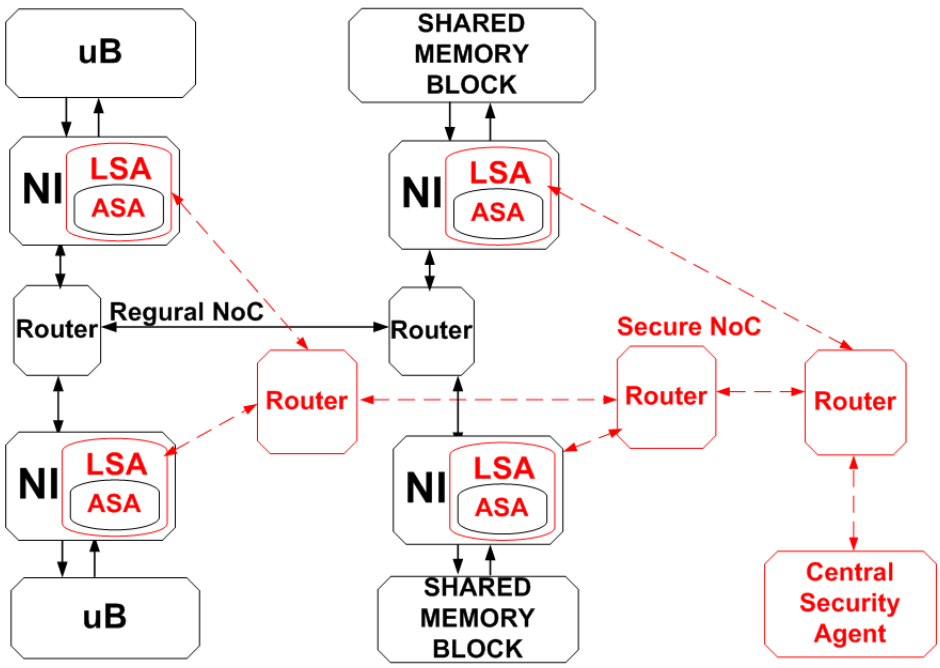

Figure 1.2. Architecture of the fully secured system (security related elements are denoted in red)

Given the general framework proposed, in our work we then focus on protection from software-based attacks to processing and memory cores as these emerge to be the most common type of problems when security of embedded systems - and especially of networked ones (Internet-enabled) - is considered. In order to prove the viability of the solution one of the most common types of attacks 'code-injection'- a type of attack that actually exercises in an very extended way our proposed solution is emulated. It is actually a technique to redirect execution of a trusted application to the malicious code by taking over the instructions' control flow. This is usually done by exploiting buffer overflows and smashing parts of the stack in which the correct return address has been saved. Another major security threat that is considered in this work is related to the Denial-of-Service attack type. This attack is widely present in computer networks and it mostly exploits limitations of communication channels throughputs which are prone to congestion in case of massive data transmission (this is usually done by initiating different kinds of irregular traffic). To validate our approach with real-world experiments we address here such type of problems in an NoC-based MPSoC system implemented in FPGA. As a demonstrator for the attack scenarios and the proposed protection concepts we use a system based on MicroBlaze CPUs (available on the Xilinx FPGAs) running uClinux embedded operating system. Considering individual characteristics of each type of core, autonomous protection solutions able to secure single core from targeted attack should be designed. Accordingly, in the scope of this work and bearing in mind the system composition exposed in the Section 4.1.1 we propose appropriate protection techniques, developed and integrated in the wider security framework in form of *Attack Specific Agents*. More specifically:

- *MemPROT* is based on Data Protection Unit (DPU), a firewall-like structure that filters unauthorized memory access requests. DPU is employed in the Network Interface attached to a shared memory block (Fiorin, Palermo, Lukovic and Silvano [2007]; Fiorin et al. [2008]). The solution which represent one type of Attack Specific Agent (ASA) is explained in details in Section 6.3.1

- *InjectPROT* is a set of combined SW/HW units replicating functions' return addresses and providing protection from buffer-overflow code-injection types of attacks. Stack Protection Unit (SPU), as a central part of InjectPROT, is employed directly at each processing core (Lukovic et al. [2010]; Lukovic and Christianos [2010a]) and represents another ASA, explained in details in Section 6.3.2.

- *DoSPROT* is a Denial-of-Service detecting system that is embedded in the security framework based on traffic monitoring units (deployed in form of ASAs in each core) and detecting algorithms employed in the CSA. The solution has been implemented in a form of ASA, the detailed description is given in Section 6.3.3.

- A dedicated secure NoC (SNoC) (Lukovic and Christianos [2010b]) is implemented in parallel with the 'nominal' (regular) one, connecting security elements embedded in NIs with a central security manager. It is implemented as a simplified version of the regular NoC and it is probably more efficient than a virtual channel based structure, at least for systems with a limited number of cores as suggested by Yoon et al. [2010]

The general system representation containing these components is given in Figure 1.2, that shows also the place of all aforementioned components inside the architecture. Based on such core-level protection structure we further build, in bottom-up fashion, a system-level security architecture which in fact relies on coordination of all security related actions by CSA.

In order to prove the feasibility of the concept we have adapted and synchronized with each other a number of different Attack Specific Agents encapsulated in Local Security Agents (one per each core). Protections against code injection type of attacks, unauthorized memory access and Denial-of-Service are combined in the prototype being integrated in appropriate LSAs. Both, LSA for MicroBlaze processor and LSA for shared memory cores are designed and implemented. Moreover, all the security related elements of the system are interconnected by means of a dedicated secure NoC. Monitoring and coordination of all agents is done through defined policies implemented by CSA. The experimental evaluation of the model is provided in described in details in Chapter 7.

In summary, the main contribution of the work lays in the novel approach to system level security, based on multi-agent technology, which integrates in hierarchical fashion, a variety of protection strategies establishing in such a way as to provide a system wide, comprehensive security framework.

The rest of the document is organized as follows: Chapter 2, provides an overview of ongoing research on security protection techniques in MPSoCs and NoCs. In Chapter 3, fundamental problems and challenges tackled are exposed; while Chapter 4 describes reference architecture and attack models, Chapter 5 shows the proposed conceptual solution. Chapter 6 provides detailed information on the solution implementation, and Chapter 7 shows synthesis and testing

results. Chapter 8 discusses the costs introduces by the proposed solution and provides comparison with other relevant related solutions. Finally, Chapter 9 presents conclusions and future work of the present thesis.

Chapter 2

Background and Related Work

Over the last decades the number of transistors integrated on a chip has been steadily increasing more or less according to Moore's Law. Intel 4004 introduced in 1971 had 2300 transistors and was running at 106KHz, while the first Intel Pentium introduced in 1992 had 3.1 million transistors running at 66MHz. The transistor count for modern processors is in a order of few billion transistors and operating frequencies can reach up to 5GHz. Still, it might be worth noting that while transistor count has kept increasing following forecasts made in the '90s, since the early 2000's frequency increase has slowed down, due to excessive power consumption problems, heating and increased impact of wire-delay on the communication. The exponentially increasing number of transistors has since then been invested in ever larger on-chip caches, but even there we have reached the point of saturation (Duato [2008]; De Bosschere et al. [2007]). Such trends urged for a massive paradigm shift towards multi-core architectures. Improvements in performance are now achieved by putting multiple cores on a single chip, effectively integrating a complete multiprocessor on one chip. Since the total performance of a multi-core is improved without increasing the clock frequency, multi-cores offer a better performance/Watt ratio than a single core solution with similar performance (De Bosschere et al. [2007]).

The many-core paradigm poses new challenges for researchers, including such aspects as (according to De Bosschere et al. [2007]):

- new design complexity issue (as special-purpose computing nodes can have a significant impact on the memory hierarchy of the system)
- on-chip communication (that requires more scalability and flexibility)
- low power design strategies

- security in different aspects
- new programming paradigms (programming environments supporting heterogeneous multi-core systems enabling the user to manually express concurrency as

Today, heterogeneous processing units such as dedicated units as DSPs, different kinds of accelerators or application specific processors might be integrated in the same system. This introduces problems of application mapping and optimizations. Architectures like this require tools for fast and efficient prototyping and testing of the design. Moreover, design trends in MPSoCs design go in the direction of blurring the traditional distinction between SW and HW elements of a system Vahid [2003]; modern system design is actually HW/SW co-design, so that designer decides based on cost-performance analysis on how specific functionality can be realized.

In the scope of this work we focus our attention on Network-on-Chip interconnection solutions and on security issues relevant for MPSoC design.

2.1 On-chip interconnecting strategies - Networks-on-Chips

As the number of cores per chip is growing, followed by increased cores heterogeneity, many challenges for the architecture designer derive from the interconnection solutions. Chip size will scale up slightly while gate delays decrease compared to wiring delays. A simple computation shows that delays on wires that span the chip could extend longer than the clock period ((Sylvester and Keutzer [2000]; Ho et al. [2001])) which is already the case, e.g., for Intel multicore chips (buses are pipelined, for that reason). Synchronization of future chips with a single clock source and negligible skew will be extremely difficult; energy and device reliability concerns will impose small logic swings and power supplies. Electrical noise due to crosstalk, electromagnetic interference, and radiation-induced charge injection will likely produce data errors. Thus, transmitting digital values on global wires will be inherently unreliable (Benini and De Micheli [2002]).

The Network-on-Chip (NoC) concept (Benini and De Micheli [2002]; Dally and Brian [2001]) has emerged as a promising solution for the communication needs initiated by aforementioned trends. NoCs scale well, eliminate the

need for arbitration when accessing the communication medium, provide well-defined interfaces for the cores to be attached and, due to their flexibility, provide fertile soil for deployment of many auxiliary services. Researchers have proposed enhancements of NoC architectures for supporting system monitoring (Ciordas et al. [2006]; Fiorin et al. [2009]) as well as security services (Diguet et al. [2007]; Fiorin et al. [2008]).

In general the trend of paradigm shift from bus-centric toward core-centric interconnection design has been well documented and advocated by the research community (D. [2000]; Grecu et al. [2004]). The main differences between the two can be summarized as follows:

- Bus-Centric Protocol Interface
  - Cores are forced to interface to the particular bus facing all the limitation such bus may bring
  - Interfaces to a different buses may cause incompatibilities
- Core-Centric Protocol Interface
  - Facilitates unrestricted delivery of all core signals
  - Enables unconstrained interface bridge to any interconnect
  - Number of gates in interface is typically lower than for bus interface

One of the most widespread open core protocols is OCP (*Open Core Protocol* [2000]), that is aimed at defining a common standard for intellectual property (IP) core interfaces, or sockets, capable of facilitating 'plug and play' SoC design. Its foundation is strongly supported by leading industrial and academic institutions targeting at facilitating design of complex SoC so as to make it more efficient for wider audiences. Moreover, OCP defines a set of standard signals for test and debugging. A number of NoCs implement the OCP communication interface (Bjerregaard et al. [2005]; Dall'Osso et al. [2005]; Fiorin et al. [2008]).

Fundamental NoC advantages over traditional bus solutions are considered to be (according to Benini and De Micheli [2002]; Bjerregaard and Mahadevan [2006]):

- Only point-to-point wires are used in NoCs (as compared to buses where each attached unit adds parasitic capacitance)

- NoCs are layered structures that separate and encapsulate different issues at different levels (e.g. transaction and transport) leaving more space to deal with problems like timing and Quality of Service.
- Bus arbiter delays grow with the number of masters while in NoCs routing decisions are distributed
- While bus bandwidth is shared by all units attached, in the case of NoCs aggregated bandwidth scales with network size

Surely there are also NoC shortcomings, such as (according to Guerrier and Greiner [2000]):

- Internal network congestion may introduce latencies
- Network introduces significant area overhead
- Cores need wrappers to adapt to NoC
- The concept is novel and requires acquisition of new skills by the designer

In general Network-on-Chip architectures are characterized by (Benini and De Micheli [2002]; Benini and Bertozzi [2005]): Network topology; Switching and Routing strategies; Flow control mechanism.

Network topology (Pande et al. [2005]; Bartic et al. [2005]) denotes the physical interconnection structure of the network graph (e.g. mesh, torus, binary tree or irregular, see e.g. DeMicheli and Benini [2006]). It may be: direct (each node is connected to every switch) - or indirect - nodes are connected to specific subsets of switches.

Routing represents determination of an optimal source-destination path while *switching* describes a way the data is actually transferred through the switching elements (Hu and Marculescu [2004b]; Bjerregaard and Mahadevan [2006]; Palesi et al. [2006]). Several different aspect can be considered for these strategies (according to Bjerregaard and Mahadevan [2006] and DeMicheli and Benini [2006]):

- *Circuit vs packet switching*. In the first strategy, the link from source to destination is established and reserved until the transport of data is complete. Packet switched traffic, on the other hand, is performed on a per-hop basis, each packet contains routing information as well as data.

- *Connection-oriented vs connectionless*. Connection-oriented mechanisms involve a dedicated (logical) connection path established prior to data transport. The connection is then terminated upon completion of communication. In connectionless mechanisms, the communication occurs in a dynamic manner with no prior arrangement between the sender and the receiver. Thus circuit switched communication is always connection-oriented, whereas packet switched communication may be either connection-oriented or connectionless.

- *Deterministic vs adaptive routing*. In a deterministic (static) routing strategy, the traversal path is determined by its source and destination alone. Adaptive schemes involve dynamic arbitration mechanisms which results in more complex design.

- *Central vs distributed control*. In centralized control mechanisms, routing decisions are made globally. In distributed control, the routing decisions are made locally.

As for packet switching several forwarding strategies which show how the packets are passed from one to another router are present. Three main strategies are defined (Bjerregaard and Mahadevan [2006]):

- *Store-and-forward* is a packet switched protocol in which the node stores the complete packet and forwards it based on the information within its header. Thus the packet may stall if the router in the forwarding path does not have sufficient buffer space.

- *Wormhole routing* combines packet switching with the data streaming quality of circuit switching to minimize packet latency. The node looks at the header of the packet to determine its next hop and immediately forwards it. The subsequent flits are forwarded as they arrive. This causes the packet to move in 'worm' fashion through the network, hence the name.

- *Virtual cut-through* routing has a forwarding mechanism similar to that of wormhole routing. But before forwarding the first flit of the packet, the node waits for a guarantee that the next node in the path will accept the entire packet. Thus if the packet stalls, it aggregates in the current node without blocking any links.

There are variety of popular routing algorithms with different properties (e.g. table-based routing, source routing, node-table routing, 'hot potato' etc.).

Flow Control Mechanism is defined as the mechanism that determines the packet movement along the network path by Dally and Brian [2001]. Flow control addresses the issue of ensuring correct operation of the network. It also encompass issues on optimal utilization of network resources optimally Bjerregaard and Mahadevan [2006]. A concept of *Virtual Channels* VCs implements flow control in a particular way, in that it relies on the sharing of a physical channel by several logically separate channels with individual and independent buffer queues.

A number of NoC related topics have been raised in recent years and have attracted remarkable attention from both academic and industrial community. In essence, main research topics includes:

- Architecture and design flow (Kumar et al. [2002]; Benini and Bertozzi [2005]; Goossens, Dielissen and Radulescu [2005])
- Programming models (e.g. message passing vs. shared memory) Bjerregaard and Mahadevan [2006]; Benini and De Micheli [2006]; Goossens, Dielissen and Radulescu [2005]
- Low power design (Silvano et al. [2011]; Lee et al. [2006]; Shacham et al. [2007])
- Fault-tolerance (Pullini et al. [2005]; Tamhankar et al. [1297–1310]; Derin et al. [2011])
- Cache coherence (Petrot et al. [2006]; Bolotin et al. [2007]; Kurian et al. [2010]; Seiculescu, Volos, Khosro, Falsafi and De Micheli [2011])
- Buffer sizing and queue management (Marculescu et al. [2005]; Hu and Marculescu [2004a])
- QoS and Security (Guz et al. [2006]; Harmanci et al. [2005]; Faruque et al. [2006]; Rijpkema et al. [2003])

The most recent NoCs research trends go in the direction of design of photonic Network-on-Chips (Shacham et al. [2007]; Gu et al. [2009]; Kurian et al. [2010]) and also introducing 3D design (Pavlidis and Friedman [2007]; Kim et al. [2007]; Feero and Pande [2009]; Seiculescu, Murali, Benini and De Micheli [2011]). Nevertheless, the solutions we propose are independent of specific interconnect technology so that their validity will keep.

In the following we will provide an outlook on security with special regard on the most common attacks and on-chip protection solutions.

2.2 Security overview - basic terms and taxonomies

Security is, broadly speaking, considered to be a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) data integrity (according to Avizienis et al. [2004]).

The increasing pervasiveness of computer science and spreading of computers through practically all segments of modern life were followed by steady increase of criminal activities directed to data theft of proprietary information, financial frauds, virus attacks, sabotage etc. The dimensions of computer crime have been investigated in time span of thirteen years involving hundreds of computer security practitioners and reported by Computer Security Institute (CSI) in (Richardson [2003]). The most expensive computer security incidents were those involving financial fraud with an average reported cost of close to $500,000. The second-most expensive, on average, was dealing with 'bot' computers within the organization's network, reported to cost an average of nearly $350,000 per respondent. The overall average annual loss reported was just under $300,000. Virus incidents occurred most frequently, being reported at almost half (49 percent) of the respondents' organizations.

Remarkable research efforts, mostly done in general purpose computing and communications, have resulted in robust protection solutions involving sophisticated cryptographic algorithms, communication protocols, anti-virus programs etc. Security in MPSoCs, that is the focus of this work, represents a fairly specific case, and because of that it has been separately highlighted and detailed in Section 2.3. Prior to going into the details of security in this particular field we will expose basic definitions and later in Section 2.2.1 security taxonomies.

We briefly list here the definitions of key terms regarding security (according to Howard and Longstaff [1998]):

- *vulnerability* represents a weakness in a system allowing unauthorized action
- *tool* - a means of exploiting a computer or network vulnerability
- *event* is an action directed at a target which is intended to result in change of state (status) of the target
- *attack* represents a series of steps taken by an attacker to achieve an unauthorized result

- *incident* denotes a group of attacks that can be distinguished from other attacks because of the distinctiveness of the attacker, attack, objectives, sites and timing

- *unauthorized result* - an unauthorized consequence of an event (e.g. disclosure if information, corruption of information, denial of service, recourses theft and so forth)

These terms represent basic 'building blocks' of the security theory and taxonomies shown in Section 2.2.1.

2.2.1 Security taxonomies

A **taxonomy** is a classification scheme (a structure) that partitions a body of knowledge and defines the relationship of the pieces (Howard and Longstaff [1998]). Security related taxonomies are commonly organized as attacks' and countermeasures' ones (Ramanauskaite and Cenys [2011]; Igure and Williams [2008]). Still, there are several ways to derive a taxonomy as for instance *action-based* security taxonomy presented in (Stallings [1995]). A more comprehensive *incident* taxonomy is developed by (Howard and Longstaff [1998]). It correlates all the actors, their interactions, actions and objectives as presented in Figure 2.1.

General overview of computer attack specification is given in (Paulauskas and Garsva [2006]). The taxonomy of the attacks on embedded systems is presented in (Ravi et al. [2004]). In general it groups all the attacks in two disjoint groups namely, physical and logical attacks (refer to Figure 2.2).

In this work we focus on software attacks and we do not consider any kind of physical or side-channel attacks. In the general case, software attacks address existing (i.e. installed) software applications in the targeted system. They mostly aim at redirecting regular application's control flow to inserted malicious code by means of 'code injection' or similar techniques. Usually these attacks are fought off by diverse software countermeasures. Unfortunately, this approach is restricted to known attack scenarios and it cannot protect from an unknown threat that avoids the existing protection.

In this work we consider, without diminishing the general validity, a case study targeted on the most usual software attacks such as unauthorized memory access, Code Injection and Denial of Service (as they are the most widespread according to Patel et al. [2010] and the most expensive as stated in Richardson [2003]).

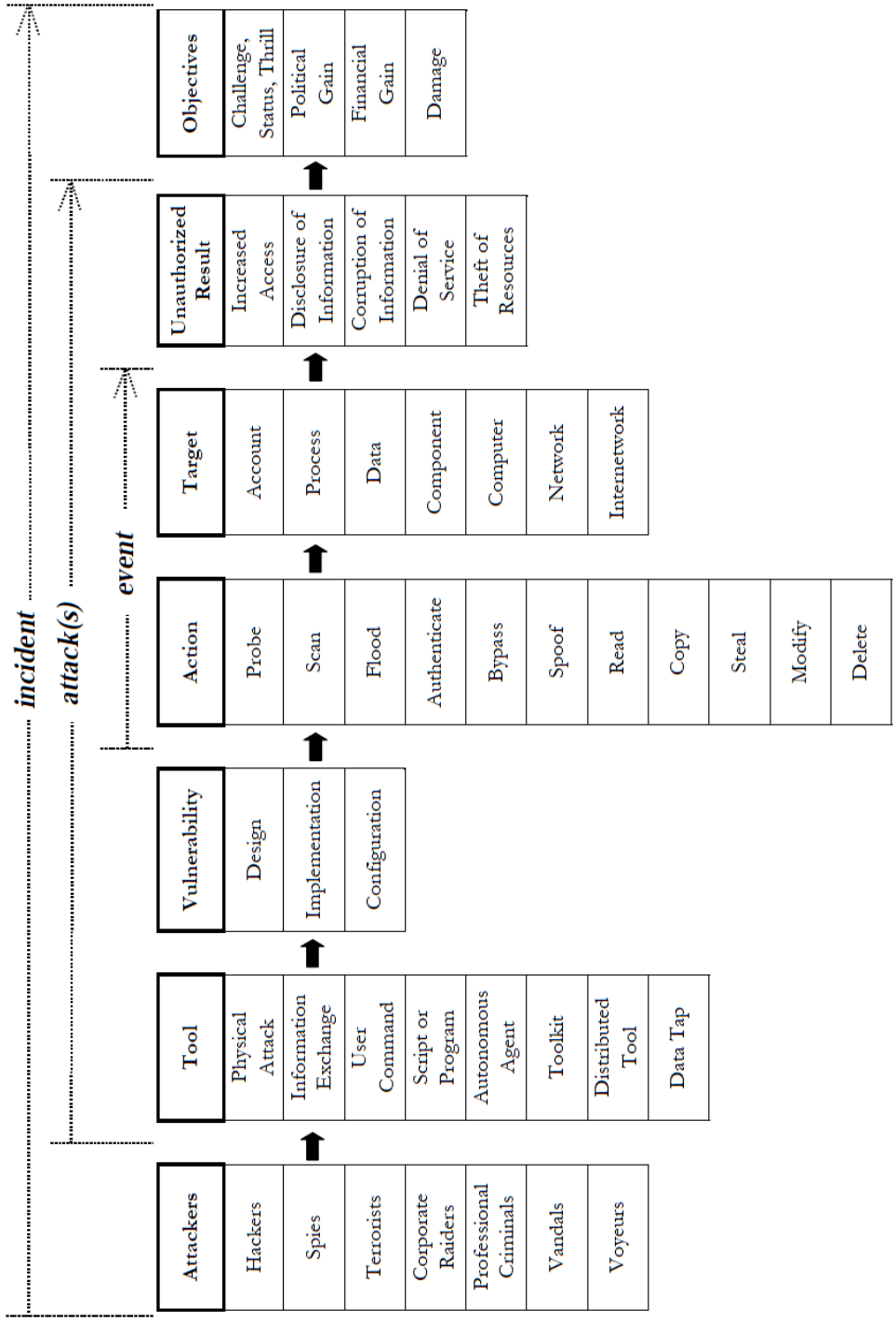

Figure 2.1. Computer and Network incident taxonomy by (Howard and Longstaff [1998])

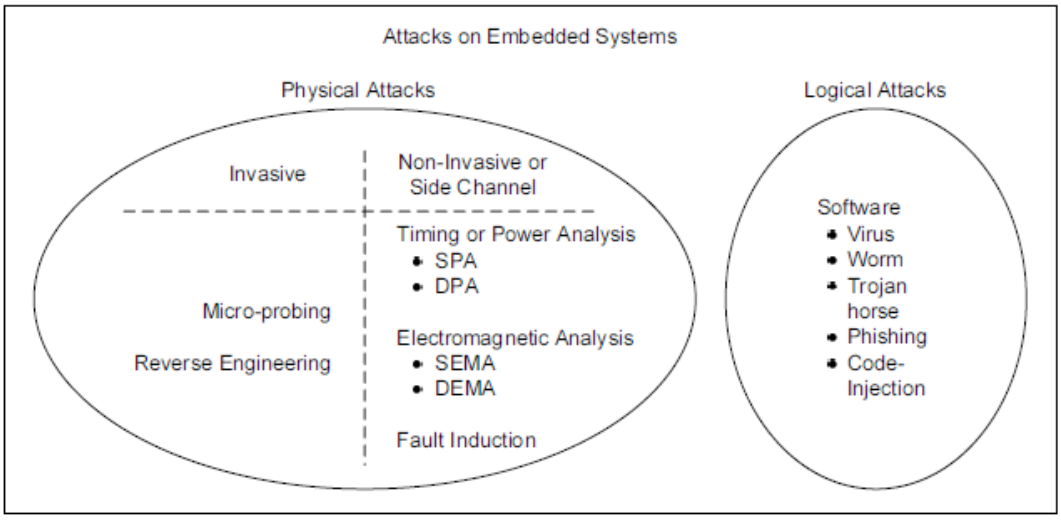

Figure 2.2. Taxonomy of security attacks on embedded systems according to (Ravi et al. [2004]; Hardware-software design methods for security and reliability of MPSoCs [2009])

2.3 Security aspects and on-chip systems

Protection against malicious attacks is considered at different levels and from various points of view. Although security has been so far often neglected by embedded system designers, it is in fact, a new dimension that designers should consider in the design process, along with other metrics such as area cost, performance, and power. The challenges unique to embedded systems require new approaches to security covering all aspects of embedded system design from architecture to implementation (Ravi et al. [2004]). Many solutions employed in general purpose computing and communication systems (e.g cryptography strategies, anti-virus programs etc.) introduce relevant communication and computation overhead, which in turn impacts on performance and power consumption in a measure that is not tolerable in systems with limited resources, such as MPSoCs. Hence, custom solutions for protecting such systems by enhancing existing architecture (communication structures) have been proposed by various researchers (Coburn et al. [2005]; Fiorin et al. [2008]; Porquet et al. [2011]). Commercial solutions are present as well. Trusted Platform Module (TPM) (see *Trusted Platform Module (TPM) Specifications* [2011]) introduces secure generation of cryptographic keys, in addition to a hardware pseudo-random number generator. It is implemented by some Dell BIOS settings.Several other designs appear, such as: Mobile Trusted Modules (MTM) (Ekberg and Kylanpaa [2007])

by Nokia; M-Shield (Srage and Azema [2008]) by Texas Instruments for security on mobile handsets, which are either hardware agnostic (e.g., MTM), or designs that augment the processing core for increased security (Ekberg and Asokan [2010]; Alves and Rudeli [2007]).

ARM has developed the TrustZone system (refer to Alves and Felton [2004]), described in details in Section 2.3.1 of the thesis. In (Schellekens et al. [2008]) authors present a protocol by which an external component containing non-volatile memory, some logic, additional write-once memory for shared keys, and an integrated HMAC primitive can be used to provide external authenticated non-volatile memory (EANVM) sufficient to support the required protection of secure state. A similar solution has been used for Intel Authenticated Memory developed to complement ARM TrustZone (Ekberg and Asokan [2010]); Sonics introduced its' own protection system (*SonicsMX SMART Interconnect Datasheet* [2008]) (also explained in details in Section 2.3.1 of the thesis). It is expected that Intel will experiment with on-chip anti-virus security solutions in light of recent acquisition of McAfee (refer to Greenberg [2011]).

Regarding the protection of MPSoCs against malicious attacks, recently proposed solutions can be mostly classified in two groups, namely:

1. Establishment of different *security domains* being defined as applications' execution environments used to confine their operation effects and resources utilization (usually related to the level of the trust or importance that is assigned to certain application). Access to system resources is then allowed/restricted accordingly to the 'trustiness' of the domain to which the application belongs. Considerable numbers of solution based on such approach have been implemented in general purpose computing, but their cost in terms of resources consumption may represent a serious obstacle for wider adoption in MPSoCs. So far, in this sense two different approaches have been adopted in the case of embedded systems:

    - Implementation of *virtual security domains* (Inoue et al. [2005]; Hiroaki et al. [2008]). Dedicated execution environments are established for execution of untrusted applications in a way that separates them from the system itself (Gong and Ellison [2003]). This approach is usually very costly in terms of resources utilization and performance degradation.
    - Implementation of specific structures separately dedicated to different application execution environments. One of such approaches consists in introduction of different *compartments* with dedicated identifiers and accordingly assigned privileges. For instance, the NoC-MPU

(Memory Protection Unit by Porquet et al. [2009]) aims at defining a set of access rights on various address regions for different compartments at platform level.

2. Monitoring of application execution and validation of behavior correctness (Patel and Parameswaran [2008]; Arora et al. [2005]; Patel et al. [2010]). The method relies on defining 'permissible behavior' by identifying suitable or expected program properties in run-time execution. The application code is instrumented by inserting special instructions so that control flow can be monitored in desired 'resolution'. An additional processing core is usually employed as system monitor and verification unit. The drawbacks of this strategy are in problems with code insertion which imposes additional design efforts and performance degradation. Moreover, 'permissible behavior' is usually fairly difficult to be precisely defined and verified.

Our work can be roughly classified in the second group, even thought it does not fully implements system monitoring and on the other hand it still implements some aspects of security domains approach. A portion of the proposed security framework (see for instance MemPROT described in Section 6.3.1) relies on a firewall-like protection mechanism, that filters access to the shared memory according to assigned rights following a philosophy not far from the security domains concept (actually different privileges establishment and verification of the access rights prevent access to resources in a similar fashion as security domains do). In that sense this particular solution can be classified in the first class of protection strategies listed above. Still, the proposed work differs from the aforementioned solutions in the general approach to system security. In other words, we target system-level protection rather then 'localized' IP custom solutions. In that sense the proposed solution is platform-oriented rather than processor oriented.

The architectural framework for security and reliability of MPSoCs proposed by (Patel et al. [2010]) represents in a way a solution similar to ours since it targets security at the level of MPSoC by employing a dedicated security processor and tests it against buffer overflow type of attack. Nevertheless, there are several considerable differences in methodology and implementation between their work and our envisioned solution. In (Patel et al. [2010]) the customized implementation for an Application Specific Instruction set Processor (ASIP)is presented; the solution there presented does not consider NoC architectures but relies on application code instrumentation which is not needed in our case. A

detailed comparison of the two solutions in terms of performance, power and area overhead is given in Section 7.3 and Chapter 8.

A proposal for establishment of a trustworthy system based on a set of dynamically reconfigured firewalls embedded in NoC is presented in (Sepulveda et al. [2011]). They propose a security architecture similar to (Fiorin, Palermo and Silvano [2007]) system with additional support for dynamical reconfiguration of security policies. The implementation allows the MPSoC protection by means of communication management. The security mechanism uses the information embodied in the packets that flow through the NoC to enforce the different security policies. This work has been recently enhanced (Sepulveda et al. [2012]) so that they introduce QoSS (quality of security service) exploiting the NoC components to detect and prevent a wide range of attacks. They present the implementation of a layered dynamic security NoC architecture that integrates dynamic security firewalls in order to detect attacks based on different security rules. They provide SystemC-TLM cycle-accurate model of the architecture and no hardware implementation has been provided. Comparison with this solution is discussed in Section 8.

2.3.1 Protection against specific attacks

Considering specific attacks and protection solutions for specific platforms, many efforts have been invested in research in general purpose computing and recently in embedded systems design. In this section we review the most relevant work concerning attacks of greatest relevance for us, namely on-chip memory protection, 'code injection' and Denial-of-Service attacks.

On-chip memory protection

Data protection in MPSoCs is one of the relevant topics for our work (Fiorin et al. [2008]). An implementation of a protection unit for data stored in memory is described in (Coburn et al. [2005]). The proposed module enforces access control rules that specify how a component can access a device in a particular context. AMBA bus transactions are monitored and a lookup table (indexed by the concatenation of the master identifier signals and the system address bus) is employed to store and check access rights for the addressed memory location and to stop potential not-allowed initiators.

As mentioned above, similar problems have been addressed as well in industrial solutions. The on-chip memory protection unit developed by ARM, in systems adopting the ARM TrustZone technology (Alves and Felton [2004]),

provides the possibility of including a specific module - the AXI TrustZone memory adapter - to support secure-aware memory blocks. A single memory cell can be shared between secure and nonsecure storage areas. Transactions on the bus are monitored to detect the addressed memory region and security mode in order to cancel nonsecure accesses to secure regions and accesses outside the maximum address memory size. The module is configured by the TrustZone Protection Controller, which manages the secure mode of the various components of the TrustZone-based system and provides the software interface to set up the security status of the memory areas.

The SMART Interconnect solution by Sonics (refer to *SonicsMX SMART Interconnect Datasheet* [2008]), introduces on-chip programmable security 'firewall' which is employed to protect the system integrity and the media content passed among on-chip processing blocks, various I/Os, and the memory subsystem. The firewall is implemented through an optional access protection mechanism to designate protection regions within the address space of specified targets. The mechanism can be dynamic, with protection region sizes and locations that can be programmed at runtime. It can also be role dependent, with permissions defined as a function not only of which initiator is attempting to access but also which processing role the initiator is playing at that time. Protection regions subdivide a target's address space, where each target can have up to eight protection regions. Each protection region is assigned to one of the four levels of priority.

NoC-MPU described in (Porquet et al. [2011]) is a dedicated Memory Protection Unit allowing to support the secure and flexible co-hosting of multiple native software stacks running in multiple protection domains, on any shared memory MP-SoC using a NoC.

Our solutions goes a step farther than previous implementations of data protection techniques in the sense that, for the first time, it addresses the problem of the data protection on an NoC-based MPSoC. The details of the solution for this particular type of the attackis given in Section 6.3.1.

Code insertion

A Code Injection attack exploits a vulnerability or an error in a program injecting or introducing malicious code into a program execution. The Code Injection attack could inject some malicious code or some code that points to a malicious code that may already be present in a system. Executing the malicious code aims to change the control flow of the main program and may have unpredictable consequences (*Hardware-software design methods for security and*

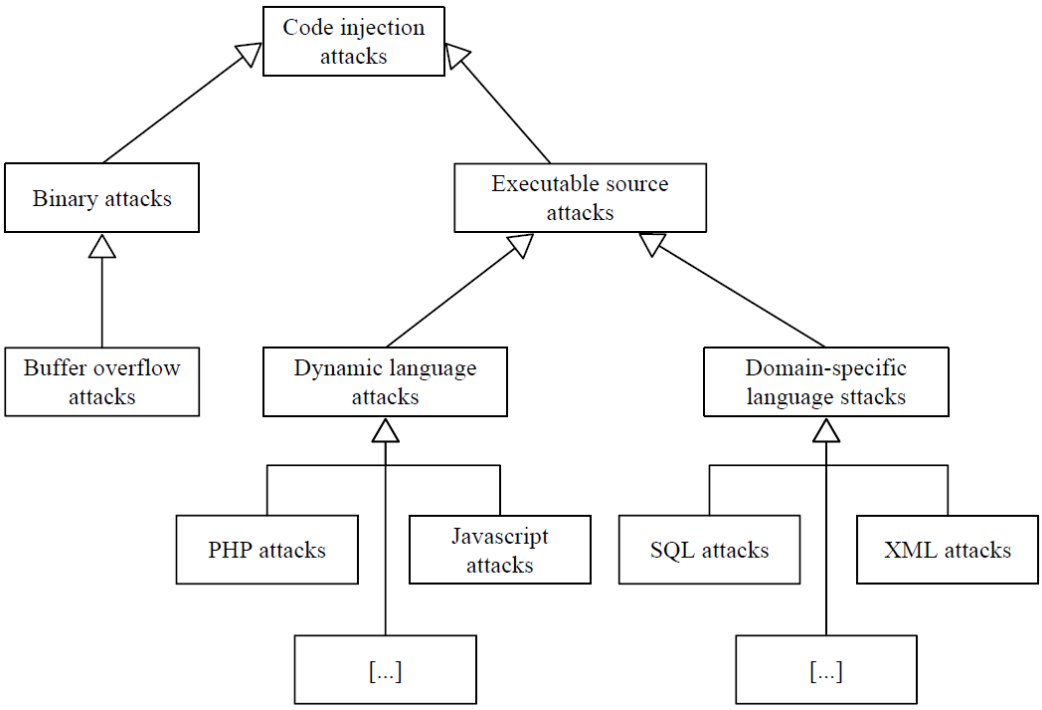

Figure 2.3. A taxonomy of code injection attacks according to (Mitropoulos et al. [2011])

reliability of MPSoCs [2009]).

Weaknesses in system implementation inevitably remain and are often exploited by the attackers in the form of either physical, software or side-channel attacks. Software attacks that exploit vulnerabilities in software code or weaknesses in the system design are the most common type of attacks (Coburn et al. [2005]). Stack and heap based buffer overflows are the most common type of Code Injection attacks (Pincus and Baker [2004]).

A variety of forms of *code injection attack* have been determined and a taxonomy of this kind of attack has been developed and presented by (Mitropoulos et al. [2011]) as shown in Figure 2.3. Appropriate countermeasures have been designed as well, these are presented in form of taxonomy (Mitropoulos et al. [2011]) in Figure 2.4.

In the present work we consider the *buffer overflow* (also known as 'stack smashing') type of the attack, actually constituting a special form of *binary code injection* group of attacks: therefore, we analyze it here in particular detail.

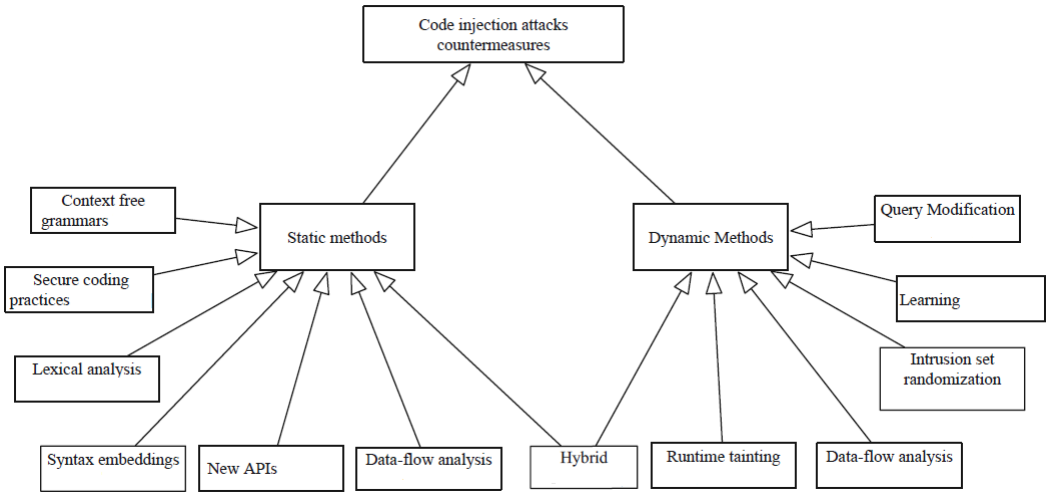

Figure 2.4. A taxonomy of code insertion attack countermeasures according to (Mitropoulos et al. [2011])

Binary code injection - Buffer overflow Binary code injection involves the insertion of binary code in a target application to alter its execution flow and execute inserted compiled code (Mitropoulos et al. [2011]).

Buffer overflow represents one of the most widespread types of software attacks, and CERT reports that nearly 11% of discovered vulnerabilities pertain to this type of the code injection attacks (Pincus and Baker [2004]; Patel et al. [2010]). The attack scenario is rather simple: the attack is performed by writing an array to the stack without checking its upper bound (e.g. using C function *strcopy()*, one can overwrite the data of the valid stack frames). Even if the stack memory is not executable, and/or separated, overwriting the return address of the caller and saved registers is still possible. This means that the attacker can redirect the control flow by giving to the targeted return function the corrupted values. The simple scenario is presented in Figure 2.5

Protection strategies

Several related works on the topic of *buffer-overflow* protection focus on proposing a variety of static methods for detecting *code injection*, to be applied at design or at compilation time (Dor et al. [2003]; Wagner and Dean [2001]). However, due to the increased portability and networking of MPSoC devices, very often these systems are exposed to threats that become manifest during run time execution of untrusted applications (usually downloaded from Internet).

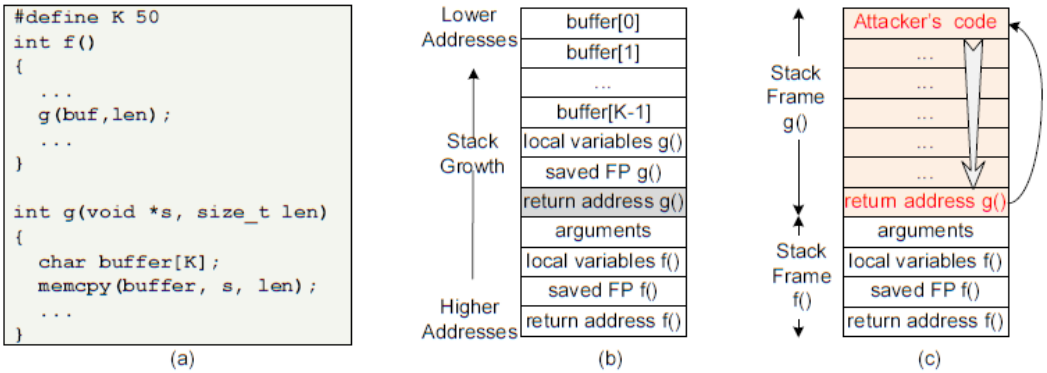

Figure 2.5. A stack based buffer overflow attack by overwriting the return address register (a) Vulnerable code (b) Layout of the stack before an attack (c) Layout of the stack after an attack, from (Hardware-software design methods for security and reliability of MPSoCs [2009])

StackGuard by (Cowan et al. [1998]), a compiler-based solution, inserts a randomly generated value in the stack position next to the return address as well as in a general purpose register. These two values are compared, at the time the execution is returned to the caller function, in order to detect buffer overflow occurrence. While this solution is simple (in hardware terms) and elegant, some techniques to overcome this protection have been devised (e.g. by Bulba and Kil3r [2000]).

StackGhost (presented by Frantzen and Shuey [2001]) proposes an approach that partially protects from corruption of return addresses having stack space allocated in register windows, without requiring re-compilation of the application source code. This technique is developed for Sun Microsystems SPARC architectures and relies on its specific features. Secure Return Address Stack (SRAS) (presented by Lee et al. [2004]) represents a combination of hardware and software supports. It introduces a special hardware unit that keeps track of all return addresses of callee functions and compares them with actual ones. Some kernel modifications are needed to support this model.

In this thesis we will focus on the aforementioned type of attack. In the implemented prototype, the proposed protection strategies are tuned to combat aforementioned threats. The solution for the attack specific protection that we propose is based on a concept similar to that proposed in (Lee et al. [2004]).

Denial-of-Service Attack

Denial-of-Service (DoS) attacks represent one of the most widespread and also financially the most expensive security incidents on Internet (Richardson [2003]). The most general taxonomy of these methods is presented by (Ramanauskaite and Cenys [2011]) and it is shown in Figure 2.6. DoS may appear in a number of different forms but in general the main goal is always the same - jeopardizing normal operation of the system in some of following ways (Blazek et al. [2001]):

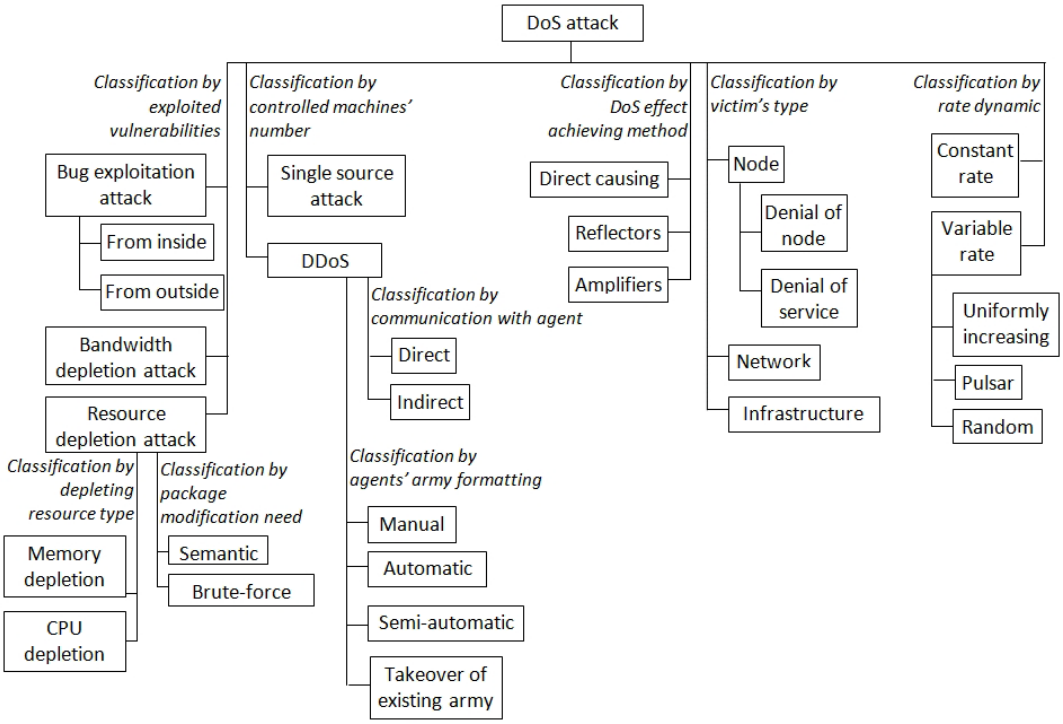

Figure 2.6. Denial-of-Service attacks taxonomy, from (Ramanauskaite and Cenys [2011])

- flooding the network in order to disturb the legitimate network traffic
- disturbing the (point-to-point) connection between two machines (users)
- preventing a regular client from accessing the service
- disrupting the service to a specific system or service

In line with such goals three basic ways of attack performing can be identified:

- consumption of scarce, limited resources
- destruction or alteration of configuration information
- physical destruction or alteration of network components

We particularly focus on the first type of the attacks listed above, concentrating on the attacks that invoke the network traffic congestion. Therefore our security framework provides protection against the two most relevant representatives of this kind - *vulnerable* and *flooding* DoS attacks.

Vulnerable attacks are software specific. They commonly rely on packet malformation exploiting certain security weaknesses in the application which cause excessive memory consumption, CPU performance degradation and general system slowdown. Popular examples include: *Neptune* or *Transmission Control Protocol synchronization (TCP SYN) flag*, *ping 'o death* and the *targa3* attacks (Carl et al. [2006]).

A *flooding* attack forces the unbounded sending of the packets and can be either a *single-source* attack originating only in one host or *multi-source* attack where multiple hosts flood the victim with a barrage of attack packets. Such an attack does not require any software vulnerability. The naive *single-source* DoS attack is relatively easier to detect than a Distributed *multi-source* DoS attack (Mirkovic and Reiher [2004]).

In our work we emulate an event similar to Internet DoS flooding attack assuming that the intruder will use our own on-chip resource to stimulate the distributed unregulated traffic on the network by executing the malicious application or by sending a large amount of useless packets through the NoC. The result is consumption of total network bandwidth and sabotage of regular transactions.

DoS detection techniques

There have been attempts to classify DoS protection strategies in different fields. The most general taxonomy of these methods is presented by (Ramanauskaite and Cenys [2011]) and it is shown in Figure 2.7. In the present work we consider pattern/anomaly detection at a victim machine in different cooperation degrees.

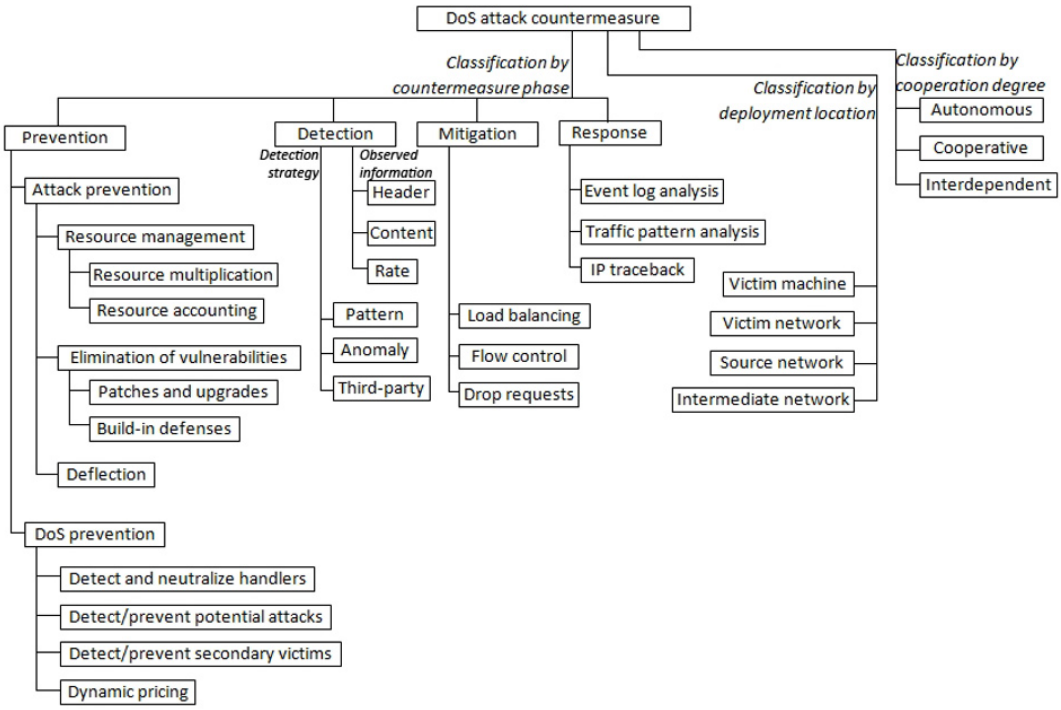

Figure 2.7. Denial-of-Service countermeasure classification scheme, from (Ramanauskaite and Cenys [2011])

All the detection techniques against this kind of attacks face the same problem - how to distinguish a network flooding attack from sudden increase in legitimate activity or burst (flash events) in regular applications' data exchange. Clearly, some statistics of network traffic must be involved. Several techniques can be seen as efficient and all of them include an evaluation of different network traffic statistic (Carl et al. [2006]):

- *Activity profiling* - An activity profile of the network is created by observing the packet headers. In this way the average packet rate is determined since these packets usually consists of similar data like destination and source address, protocol information and so on. By measuring the elapsed time between these packets the average packet rate of the network can be determined. In order to determine the attack some statistics tests should be performed.

- *Sequential Change-Point Detection* - This detection technique is based on the determining traffic statistic's change caused by the attack. Total traffic is tracked by the appropriate methods and the resultant flow is stored

as the time series. Any unusual phenomenon illustrated by the sudden large increase in network flow can be a possible DoS attack. Some algorithms like CUSUM (cumulative sum control chart, see Blazek et al. [2001]) identify the deviations in original versus expected local average in the time series. If the difference exceeds some upper bound, CUSUM statistics increases. If during time intervals containing only normal traffic the difference is bellow this bound and the CUSUM decreases until reaches zero. The disadvantage of this algorithm lies in difficulties in defining the appropriate parameters like initial conditions or upper statistical bound, although it gives opportunity for achieving a good trade-off between real and false alarms.

- *Wavelets Analysis* - Wavelet analysis describes an input signal in terms of spectral components. It provides concurrent time and frequency description, and can thus determine the time at which certain frequency components are present. Analyzing each spectral window's energy determines the presence of anomalies As proved in (Barford [2002]) the expected time series results in mid- and high-band spectral energies. To identify anomalies they weight the combination of these two spectral domains and than threshold its variability.

Obviously all the aforementioned techniques face the same major challenge namely, distinction between legitimate user activity and flooding attack. All the strategies shows limited success. A combination of various approaches will most likely produce the best result. For the purpose of our work we have adopted Sequential Change-Point Detection technique based on CUSUM algorithm as detailed in Section 6.3.3.

2.3.2 Trusted Systems, Trusted Computing and Trusting Policy

A trusted system is conceived as a distributed system in which security is achieved partly through the physical separation of its individual components and partly through mediation of trusted functions performed within some of these components (Rushby [1981]). In trusted systems the subject of security policy is assigned to the private and physically separated units which are able to communicate through secure and trusted network which also resides in a separate and isolated environment. Achieving this kind of the system solves many security problems and makes them considerably simpler. Systems must be designed to constrain access to confidential data and to confine any damage resulting from malicious software.

Developing of highly trusted computing systems is becoming a fundamental concern. The goal of decoupling trusting from the applications and putting trust entirely in the OS kernel of the systems turned out to be unfeasible. The first step to implement HW trust components began in the late 1990s and it was proposed by the *Trusted Computing Group (TCP)* that made the extension of the common computing platforms by implementation of the specific HW unit - the *Trusted Platform Module (TCP)* (see Mitchell [2005]).

It is a common case that nodes also communicate with each other, so different kind of trusting policies were also developed. The main purpose of these policies is to verify the reliability of the nodes determining and assigning proper trusting values to them. These are checked prior to executing any application on the node. The trusting value of the node can be determined in different ways: as an example, it can be computed, according to the ability of the node to execute the task within some time limits given constrained resources (Ferrante et al. [2008]).

Different applications of trusting policies have been proposed in variety of fields (Gallery and Mitchell [2009]). In the case of embedded systems, a design technique for security and trust has been proposed by (Verbauwhede and Schaumont [2007]). Still, secure isolation between security critical and non-security critical task on a single embedded device represents one of major concerns (Gehrmann and Lofvenberg [2011]) together with devising an efficient light-weighted trusting protocol (see Ferrante et al. [2008]). The concept of trust could be applied also in multi-core systems security, since efficient determination of certain types of attacks (e.g. DoS) may require wider system interaction and coordination among nodes. In our work we rely on an elegant and straight-forward trusting policy (as presented in Ferrante et al. [2008]), combining information obtained from different kinds of *agents* deployed in various cores (as explained in Section 5.3).

2.4 Multiple-agent systems

Multiple-agent systems (MAS) engineering strategies have emerged as a promising solution to tackle various problems in complex, distributed and highly dynamic systems (Wooldridge [1997]). An agent-based system is one in which the key abstraction used is that of an agent. By an *agent*, we mean a system that enjoys the following properties (according to Wooldridge [1997]):

- *autonomy*: agents encapsulate some state, and make decisions about what to do based on this state;

- *reactivity*: agents are situated in an environment, which they are able to perceive, and are able to respond in a timely fashion to changes that occur in it;
- *pro-activeness*: agents do not simply respond to inputs from the environment, they are able to perform in goal-directed fashion by taking the initiative;
- *interactivity*: agents interact with other agents, and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

Being so generally defined and on the other hand offering great flexibility and portability, *agents* have found the way to application in many fields. Probably the most well-established use of agents can be found in complex, distributed software systems (Jennings [2001]). More recently multiple agent systems have been actively employed in medicine (Xiao et al. [2007]); mobile communications (Borselius [2003]); power systems (McArthur et al. [2007]), in particular for condition monitoring and diagnostics (Davidson et al. [2006]; McArthur et al. [2004]), power system restoration (Nagata and Sasaki [2002]), market simulation (Zhou et al. [2011]), network control (Dimeas and Hatziargyriou [2011]) and so-forth.

Just recently, applicability of MAS strategies for security has been attracted researchers attention in particular in the framework of information systems security applied to telecommunication infrastructures (Bonhomme et al. [2010]; Feltus et al. [2010]). On the other hand, as MAS are used in open, distributed and heterogeneous applications, the security issues may endanger the success of the application. Different research studies have addressed security issues in various MAS application fields (Wong and Sycara [1999]; Cavalcante et al. [2012]; Borselius [2003]; Rashvand et al. [2010]).

Considering the necessity for flexibility, portability and scalability of our approach, we have adopted the MAS approach to MPSoC security; to the best of our knowledge this is the first utilization of this strategy for that purpose.

2.5 FPGA technologies and their utilization for design of Multi-Processor Systems-on-Chip

FPGA technologies are considered as very suitable for rapid prototyping in hardware-software co-design; in fact they have been proposed for this purpose

since their introduction (Benner et al. [1994]). FPGA-based prototyping for IP cores simulation and evaluation has been presented in (Siripokarpirom and Mayer-Lindenberg [2004]), where an approach that enables the user to integrate hardware-implemented IP cores into a software-based simulation environment is realized using FPGA technology.

Use of FPGAs to prove design concepts in NoC-based system implementation has been adopted by a number of authors. One the first implementations of NoC concept in FPGA technology (using Xilinx Virtex2Pro board) has been presented by (Bartic et al. [2003]). In (Hecht et al. [2005]) an FPGA architecture employing NoC as interconnection medium is discussed. The system proposed is implemented on top of the communication infrastructure, providing a cost-efficient statically and dynamically reconfigurable architectural solution. Circuit-switched PNoC has been presented by (Hilton and Nelson [2006]), implemented again in Virtex2Pro. An FPGA based open source NoC architecture has been described in (Ehliar and Liu [2007]).

In (Bobda and Ahmadinia [2005]), the communication problem among modules dynamically placed on a reconfigurable device is approached using a dynamic NoC, through which the components placed at run-time on the device can mutually communicate. A run-time resource management scheme that is able to efficiently manage a NoC containing fine grain reconfigurable hardware tiles is proposed in (Nollet et al. [2005]), while in (Ahmad et al. [2006]) a dynamically reconfigurable NoC architecture is proposed for reconfigurable Multiprocessor System-on-Chip (MPSoC), with the aim of satisfying increased communication needs, low cost silicon implementation, Quality of Service and scalability.

The problem of IP-core automated generation from VHDL description was addressed by Ferrandi et al. [2006]. This work has been extended for a partial dynamic reconfiguration workflow; the IP-core Generator framework for EDK was realized and presented in Murgida et al. [2006]. Automation of processing nodes generation for multiprocessor SoCs has been discussed in Collin et al. [2001]; Rowen and Leibson [2004]. In Bartic et al. [2004] and Goossens, Dielissen, Gangwal, Pestana, Radulescu and Rijpkema [2005], automation of Network-on-Chip implementation has been presented.

To the best of our knowledge, our work represents the first solution proposing a complete comprehensive and scalable hardware-based security structure; to prove its viability we have implemented it in FPGA technology.

Chapter 3

Refined Problem Statement

In this Chapter the specific problems addressed by the thesis work are presented. We focus here on security challenges posed by modern trends in MPSoC design whose evolution results in increased security vulnerabilities. Moreover, we list fundamental issues concerning the proposed security framework discussed in the thesis. Finally, the specific attack models that are considered as the greatest threats are listed and their choice is justified.

While recently several very relevant works concerning this problem area appeared in the literature (as discussed in Chapter 2), the lack of a comprehensive and flexible solution capable of tackling heterogeneous architectures of MPSoCs at different levels has been a basic inspiration for the present work. We aim at providing a holistic, system-level protection strategy which relies on collecting and processing security related information coming from different components of the system. Our goal is to achieve security-related 'data fusion' by extracting and relating useful information from different sources and different aspects. A modular and scalable, agent-based security framework constitutes its main methodological contribution. We aim at proving feasibility of the concepts via prototype realization in FPGA technology.

The modern trends that have motivated the work for the thesis are exposed in the sequel.

3.1 Security vulnerabilities of Multi-Processor Systems-on-Chip

Several modern design trends have led to increasing vulnerabilities of MPSoCs to security attacks. Among them the most important are:

- Increased complexity of the systems that are performing a wider range of applications employing increasing numbers of HW/SW components (Wolf et al. [2008])
- Increased networking and exposure to Internet (Zhou and Wu. T. snd Wu [2009])
- Introduction of dynamically reprogrammable and reconfigurable elements in MPSoC design (Diguet et al. [2007]; Ahmad and Arslan [2005])

If these trends are combined within the same system, vulnerabilities to attacks become a real concern. It should moreover be noticed that, most security attacks are focused on exploiting implementation weaknesses rather than breaking cryptographic algorithms (Coburn et al. [2005]). Therefore, security must be carefully considered in all phases and at all levels of system design, taking into account solutions targeted at both design-time and at run-time.

3.2 Main challenge addressed - Security from a system wide perspective

As shown in Chapter 2, most research on MPSoC security is oriented towards protecting some elements of the system from specific types of the attack. On the other hand, NoC-based MPSoCs are complex systems integrating heterogeneous resources into a single entity performing a variety of tasks. Therefore, a system-wide security strategies are needed in order to foster reliability of the entire system.

In our work we aim at preserving scope and coverage of each specific security solution (adopting them to particular individual cores in the system) boosting at the same time overall efficiency of system protection. Therefore, development of a framework that would integrate, coordinate and correlate a variety of security approaches from core level up to system level represents a main challenge of the present thesis. In achieving that goal we face three main research challenges:

- Is it possible to enhance Networks-on-Chip by adding new structures capable of building protection mechanism?
- How would it be possible to correlate security related information obtained from different sources and on different aspects in order to improve each individual security aspect considered as well as security of the system as a whole?

- What should be an optimal overall system design (in terms of architecture, policies etc.) that would enable efficient and inexpensive solution for system level protection?

The approach adopted in this work has been organized towards solving problems in bottom-up fashion. In other words, security strategies and solutions for securing individual cores from the NoC side, have been addressed first, while their integration and the subsequent system level protection has been developed in later stages of the work. In addition to architectural structures some additional techniques, such as system protection policies and desig of the related supporting system elements (as for instance Secure NoC and Central Security Agent) have been developed as well.

In general, the framework aims at enlarging the coverage and boosting efficiency of the MPSoC protecting mechanism. We have identified the key requirements that such a security framework must fulfill:

- Considering that MPSoCs are increasingly heterogeneous, the framework must be capable of embracing the widest span of processing cores, shared memories and other system elements. On the other hand it should take care of the widest variety of security threats - in other words it must be **comprehensive** to the largest extent
- Due to the steady evolution of threats in terms of both targeted devices and of new forms of attacks, the framework must be easily expandable (i.e. **modular**) so that new protection techniques and solutions can be efficiently accommodated in existing architecture
- As the number of the integrated cores and of the applications they run is constantly growing, the framework must be very **scalable** so that it can sustain further evolution of MPSoCs in terms of increased number of IPs as well as tasks they execute
- The solution should be as much **portable** and platform independent as possible
- The deployment of the framework should not be too demanding in terms of design efforts: this leads to consider that some suitable well-know and elegant technology such as **Multi-Agent Systems** should be used. Moreover, the framework should not burden processing resources: therefore, an elegant solution for hosting it should be found - e.g. by enhancing NoC components

- Finally, the solution must be capable of processing and **correlating** in an intelligent manner the gathered data carrying information on various security aspects and providing a trustful estimation of the security state of the system

These fundamental requirements have been used as an orientation and guidelines through the entire process of thesis development. On the other hand, based on analysis presented in Section 2 we have determined the realistic scenarios for challenging MPSoC security. Accordingly, appropriate attack models have been developed to validate our approaches. They are based on the identified the most widespread threats to embedded systems security:

- Unauthorized memory access
- Code injection attacks
- Denial-of-Service attacks

The detailed description of these attack models is given in Section 4.2. According to identified challenges and system requirements when considering the given attack models, we base our solution on the adoption of the multiple-agent systems concept, as a consequence of which our final architecture is built as a hierarchical centralized structure. The design of such an agent architecture represent one of central issues tackled by the present thesis and it is addressed in the Chapter 5.

Part I

Research Approaches and Conceptual Solution

Chapter 4

Reference Architecture and Attack Models

Prior to presenting in detail the proposed solutions and their implementation we briefly introduce the reference model of the architecture developed to serve as a host to envisioned security framework. In this work, development, testing and validation of all conceived and proposed solutions have been performed on prototyped implementations in FPGA technology.

The experimental setup represents an FPGA implementation of an MPSoC using MicroBlaze processors (provided on the Xilinx devices) and shared memory blocks. We have implemented a custom-built NoC architecture as a communication medium among these components.

The proposed security upgrade implementation will be detailed in Section 6.3. A fairly large portion of the proposed framework is embedded in Network Interface components present in the NoC, described in the sequel.

4.1 Reference architecture

In this Section, we detail the architecture of our reference NoC-based system, giving a short overview of the general system implementation and providing architectural information about the basic NoC components, i.e., the Network Interface (NI) and the Router. We highlight also the challenges and design issues related to their implementation on FPGA.

The described NoC has been customized for implementation on a Xilinx Virtex-II Pro FPGA board (Xil [2005]). The final system implemented is shown in Figure 4.1. It is composed of four MicroBlazes, shown in the Figure as *uBlaze*, and a block of shared memory implemented using part of the BRAM available in

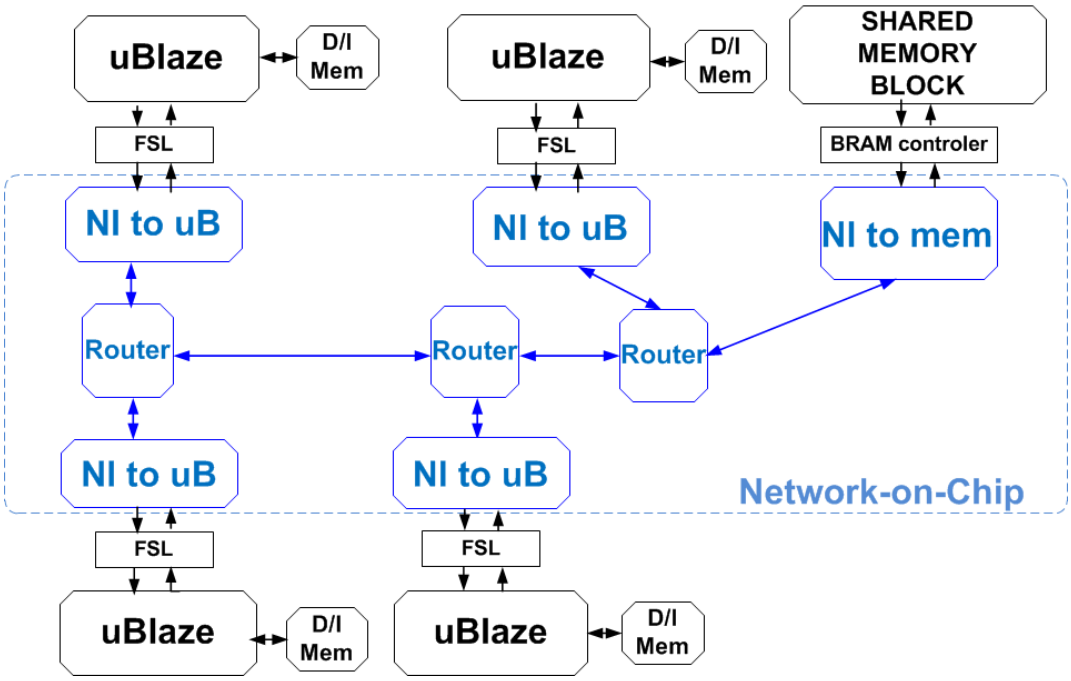

Figure 4.1. Overview of the multiprocessor system implemented to serve as experimental platform

the XIlinx board. MicroBlaze is the soft-core RISC processor provided by Xilinx, while BRAM is the on-chip Block RAM synthetisable using board resources. In this setup we experiment with four processing cores as it reflects current trends in architecture design - quad-core processors are employed in majority of newly released lap-tops and smart-phones on the market. The interconnection infrastructure is composed of a three-port router and the NIs, providing a custom interface to the two type of IP blocks present in the system. As shown in Figure 4.1, MicroBlazes are connected to the NoC through the interface to the Fast Simplex Link (FSL), an uni-directional point-to-point communication channel bus available on Xilinx FPGAs to perform fast communication between any two design elements. A detailed description of the single blocks of the NoC is given in the next subsections.

4.1.1 MPSoC components

As previously mentioned we have used as core MPSoC components the Xilinx-provided soft-core processor MicroBlaze (Xil [2005]) and BRAM (Block RAM) memory block. We briefly expose here their basic properties.

Processing elements

MicroBlaze is a reduced-instruction set computer (RISC) optimized for implementation on Xilinx FPGAs. The core is highly configurable, allowing users to select a specific set of features required by their design (e.g. cache size, Memory Management Unit, pipeline depth etc.). In our system, processing nodes include data and instruction memories, connected to the CPU through the dedicated Local Memory Bus (LMB) and whose dimension can be specified as the input of the the design flow. We connect MicroBlazes to the rest of the system through their interface to the Fast Simplex Link (FSL), an uni-directional point-to-point communication channel bus available on Xilinx FPGAs to perform fast communication between any two design elements (see Xil [2005]).

Memory elements

Shared memory blocks in our system are implemented using part of the BRAM available on-chip in Xilinx boards. Memory cores are fully synchronous and support three write mode options: Read-After-Write, Read-Before-Write, and No-Read-On-Write. A controller is associated with the BRAM component, in order to manage data transfers from and to the memory bus.

4.1.2 NoC architecture

In this Section, we detail the implementation of the NoC-based system on FPGA giving a short overview of the general system implementation and providing architectural information about the basic NoC components, i.e., the Network Interface (NI) and the Router. We highlight also the challenges and design issues related to their implementation on FPGA.

Routers

We design a router with variable length of the input and output queues and implementing a table-based routing algorithm. The router can be automatically generated with a variable number of input and output ports (see Figure 4.2). The needed information for the routing is extracted from the first flit of the packet. The destination address in the header of the packet (*DestID*) is looked up and the related output port is calculated. A request of utilization is therefore raised to the arbiter associated to the selected port. The arbiter, in case of non utilization of the associated port, assigns in Round Robin fashion the use of the queue to the input port requesting it and sets up the switch fabric in order to

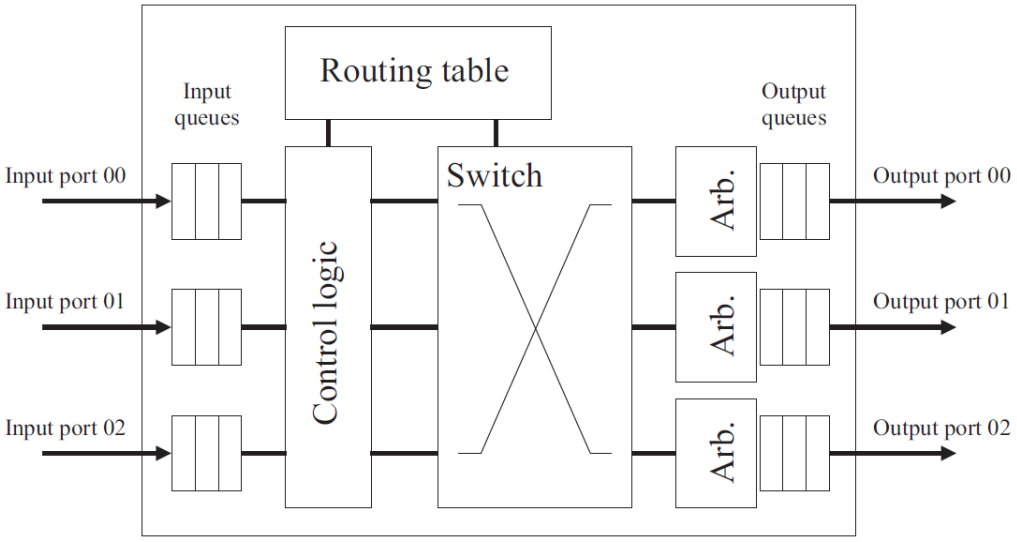

Figure 4.2. Architecture of the Router

directly connect the selected input port with the output port. We implemented the switch fabric in FPGA using a combination of multiplexers. When the last flit of the packet is received, the arbiter releases the queue and assigns it to the next input port requesting its use.

Network Interfaces

The Network Interface is employed to adapt a communicating core to the on-chip network. The module, acting as interface between the core and the communication subsystem, hides to the processing elements all the issues related to a reliable and efficient transmission of the data through the network. The NI is in charge of structuring data as packets, and of managing transmission of necessary control flow information. It is in charge of providing an interface to the transmission protocol implemented in the core and an efficient packetization of the data to be transmitted. Moreover, among the other tasks, it is also has to guarantee the necessary bandwidth and latency for the transmission and to provide additional services, such as security (DeMicheli and Benini [2006]).

In our proposed solution, the NI implements basic transmission services, supporting *Best Effort* (BE) type of traffic; a wormhole control flow strategy is implemented. As shown in Figure 4.3, NI embeds two FIFOs, one per each direction of the data flow, together with control logic implementing the basic services pre-

viously described. The NI takes data and control signals from the Fast Simplex Link (FSL) interfaces of the soft-core processor. The FIFO-like structure of the FSL allows its easy integration in the NI, adapting the control signals of the internal queues to those of the FSL interface. Any other type of core (e.g. a shared memory block) requires a dedicated attached NI that bridges communication standards between the core and the NoC.

Interface to the MicroBlaze

Figure 4.3 shows the internal structure of the NI connected to the MicroBlaze. As shown in the figure, the NI consists of two FIFOs, one for each direction of the data flow, and control logic that implements the basic services previously described. The NI takes data and control signals from the FSL interfaces of the soft-core processor. In fact, the FIFO-like structure of the FSL allows its easy integration in the NI, adapting the control signals of the internal queues to those of the FSL interface. The MicroBlaze drives the *Master* FSL interface (*FSL - M*) to transmit information to the NI, where we implemented a *Slave* interface (*FSL - S*). Transmission of data from the NI is driven through the *Master* FSL interface implemented in the NI.

In order to distinguish between control signals and data sent by the MicroBlaze, we define the following protocol. As first step, the processing element sends information on the communication (see Figure 4.4), such as the destination address (*DestAddr*), the data length (*DLength*) and the type of operation to be performed on the destination (i.e. load (*L*) or store (*S*)). *IDKey* is added for future implementation of identification techniques for the IP, while the *Opt* can be used for some extra tasks.

A control bit is provided by the FSL to indicate whether the transmitted information is a control or data word. The control bit is set to high (1) if the word is a control one (as shown in Figure 4.4, it contains destination address, length, type of operation and other optional information bits). When transmitting data, this control bit is set to 0. A number of data words, equal to the value specified in *DLength*, follows the control word. Furthermore, transitions of the control bit indicate that a new packet is to be sent to the network.

In the NI, we implemented a memory-mapped protocol in which the operations are expressed in terms of *read* and *write* to memory addresses. The NI translates a range of memories in the related identifier of the node in the network. The transaction-based protocol implemented is shown in Figure 4.5. A *store* request from the initiator of the transaction, directed to the desired target, is immediately followed by the data that have to be transfered. The target an-

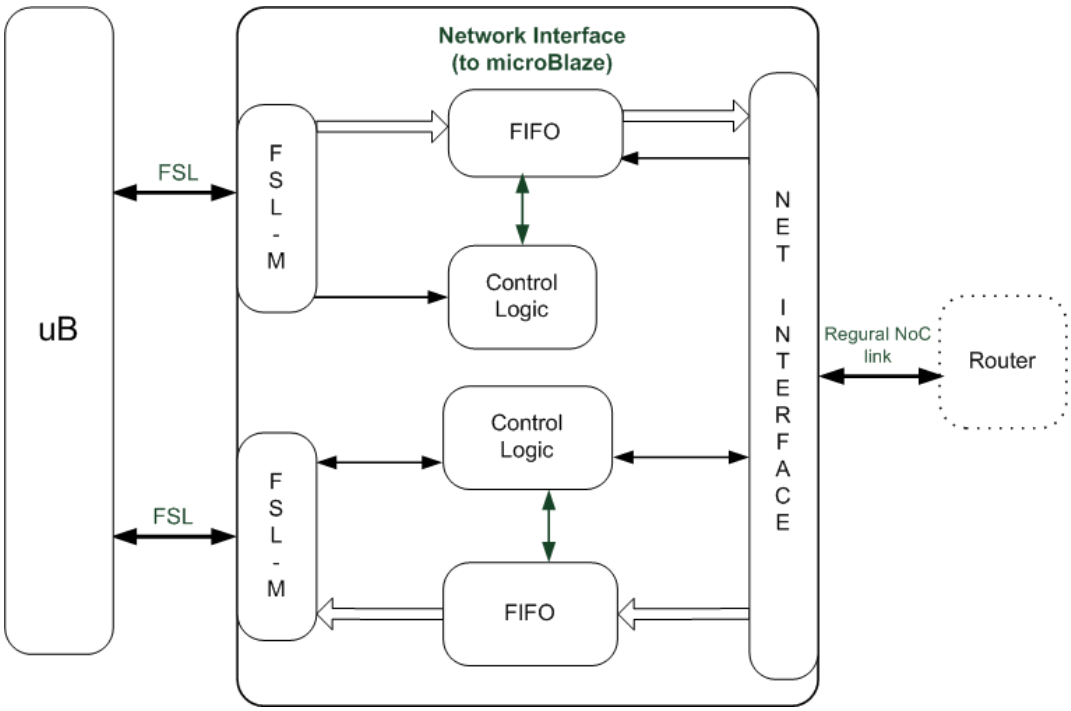

Figure 4.3. Internal implementation of the Network Interface to the MicroBlaze

DestAddr	*DLength*	*IDKey*	*L/S*	*Opt.*
0 – 5	5 – 10	10 – 15	15 – 16	16 – 31

Data
0 – 31

Figure 4.4. Control and data words sent by the MicroBlaze to the Network Interface through the Fast Simplex Link interface

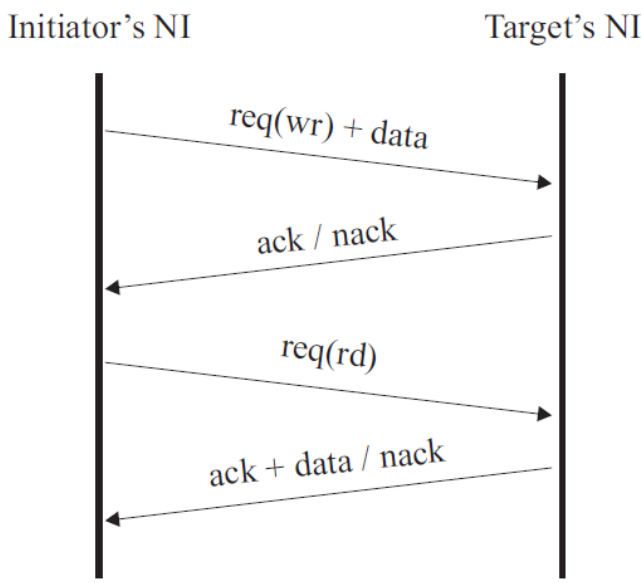

Figure 4.5. Transaction based protocol implemented between Initiators and Targets

swers with an positive acknowledgment in case of successful transaction; with a negative acknowledgment in case of unsuccessful ones. A *load* request from the initiator of the transaction is followed as positive acknowledgment by data transmitted by the target. A negative acknowledgment is sent in case of problems in the transaction. In our discussion we focus on the network level. Therefore, we assume no signal loss in the transmission of the packets through the NoC.

Structure of packets in NoC

The packets' structure used within the network is shown in Figure 4.6. We adopted a wormhole control flow in the transmission of the packets. Therefore, our packet is divided in flits, which in our case represent the smallest information logically and physically transmitted through the network. As shown in the Figure 4.6, there are different types of flits.In order to distinguish which type of flit is transmitted, two control bits are used (*Flit Type* in Figure 4.6 - bus width is therefore of 34 bits: 32 of data plus 2 of control).

As shown in Figure 4.6(a), the first flit of a packet is labeled setting the *Flit Type* control bits to '10'. The last flits, closing the packets, are labeled with '01', while intermediate flits are identified with *Flit Type* set to '00'. Packets composed of just one flit are labeled setting *Flit Type* equal to '11' (Figure 4.6(b)).

The first flit of each packet contains the header, which carries information

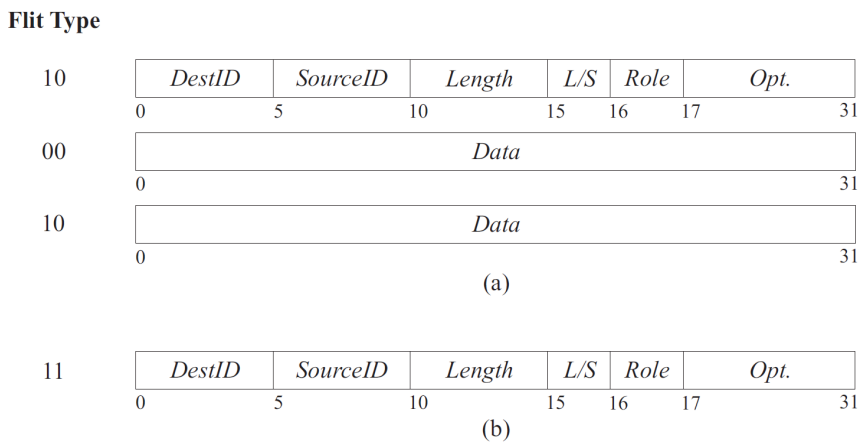

Figure 4.6. Structure of the packet used within the NoC

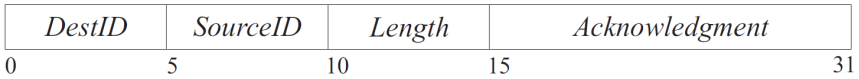

Figure 4.7. Structure of the acknowledgment packets

about the network layer (bit 0 to bit 9) and about the transaction-based protocol implemented. We grouped the two information in the same flit, in order to reduce the overhead associated with the header of the packet. *DestID* identifies the target node and its value is calculated translating the *DestAddr* send by the processing element. *SourceID* univocally identifies the source node of the transaction and its value is given by a hard-wired register in the NI. The register is necessary in order to be able to identify the initiator of the transmission. *Length* represents, in number of words, the length of the data that follows the header of the packet. The type of access requested, i.e. load or store (*L/S*), is also sent, as well as the role (*Role*) assumed by the processing element (*super-user* or *user*) and some optional bits. *Role* has been included for future improvements of the system, allowing an identification of the operative mode of the processing elements.

The structure of the packet used for positive and negative acknowledgments (ACK and NACK) is shown in Figure 4.7. As previously said, acknowledgments are sent in case of successful/allowed writing (ACK) or unsuccessful/rejected read or write (NACK). Acknowledgments are sent directly back to the NI of the processing element issuing the request. Acknowledgment packets are composed only of the header and therefore of just one flit. The fields containing information related to the routing are equivalent to those present in packets transferring data, while the following bits are set to one to notify an ACK and to zero for a

NACK.

4.2 Attack models

We consider the general case and, in order to bring the solution onto a specific use case, we target several widely present types of attacks. These include: unauthorized memory access, buffer overflow and Denial-of-Service. The considered attacks are listed as major security threats by a number of reports such as (Namestnikov [2011]; *Trends in IT Security Threats: Executive Summary* [2007]). In Section 2.3.1 we describe specific attack models considered in the present work. We show what specific scenarios, actually simulations, we have built to test provided solutions.

4.2.1 Unauthorized memory access

Protection of critical data stored in shared memory blocks inside MPSoCs represents a challenging task. Unauthorized access to data and instructions in memory can compromise the execution of programs running on the systems - by tampering with the information stored in selected areas - or cause the acquisition of critical information by external entities, such as in the case of systems dealing with the exchange and management of cryptographic keys (Coburn et al. [2005]; Fiorin et al. [2008]). Two different types of memory attacks can be considered:

- Attacks to internally shared memory (Fiorin et al. [2008])
- Attacks to shared memory that is physically outside of the system but connected to it (Coburn et al. [2005]; Cotret et al. [2011])

In our work we focus on shared memory blocks inside MPSoC. We consider initiated unauthorized memory access targeted at:

- Extraction of confidential information
- Data tamper
- Compromise the execution of programs running on the systems

By means of performing the attack itself we exploited Code Injection attacks explained in the sequel.

```
#include <stdio.h>
#include <string.h>

void func(char *p)
{
    char stack_temp[20];
    strcpy(stack_temp, p);
    printf(stack_temp);
}

int main(int argc, char* argv[])
{
    func("This text causes an overflow!");
    return 0;
}
```

Figure 4.8. Buffer overflow attack scenario

4.2.2 Code Injection Attacks - Buffer Overflow

Binary code injection attacks are enabled if the bounds of memory areas are not checked, and access beyond these bounds is possible by the program. Exploiting this vulnerability, an attacker can inject additional data overwriting the existing one of adjacent memory. From there, they can take over application control flow, or even take control of the entire system under the attack. In particular, C and C++ programming languages are prone to this kind of attacks since typical implementations lack a protection scheme against overwriting data in any part of the memory. In fact, the languages do not provide a mechanism of checking if the data written to an array is within its boundaries (Mitropoulos et al. [2011]).

There are different ways of performing the buffer overflow attack such as (*Hardware-software design methods for security and reliability of MPSoCs* [2009]):

- Overwriting the Return Address Register
- Overwriting the Frame Pointer
- Modifying a Data Pointer

In this thesis we will focus on the first listed type of attacks. We consider buffer overflow caused in a scenario similar to one give in Figure 4.8.

In the implemented framework, the security strategies are tuned to combat the aforementioned threats by implementing proper attack specific protection integrated in wider system environment.

4.2.3 Denial-of-Service attack

There are different ways of performing DoS attack but usually it is done by forcing consumption of communication resources. With regard to the taxonomy presented in Figure 2.6 we consider bandwidth depletion in form of direct Distributed DoS (DDoS) attack using direct causing method with variable rate. In terms of exact attack technique, in this work we focus on the protection against *vulnerable* and *flooding* DoS attacks. *Vulnerable* attacks are software specific, in fact, malformed packets interact with installed programs causing resources consumption corrupting system operation. They usually rely on packet changing and they are easily detectable. Popular examples include *Neptune* or *Transmission Control Protocol synchronization (TCP SYN) flag*, *ping 'o death* and the *targa3* attacks Carl et al. [2006]. A *flooding* attack forces the unbounded sending of the packets and can be either *single-source* attack originating only in one host or *multi-source* attack where multiple hosts flood the victim with a barrage of attack packets. Such an attack requires no SW vulnerability Hussain et al. [2003].

The key challenge in detecting DoS attacks is represented by correct distinction between unexpected burst of regular traffic and increased packet transfer due to the attack. In Section 7.5.1 we discuss simulation and test-benchmark techniques used to emulate realistic application execution (i.e. communication) scenarios (Basseville and Nikifirov [1993]).

Chapter 5

The proposed conceptual solution - a Hierarchical Agent-Based Security Framework

In this Chapter we introduce our approach to incorporating security-related techniques and methods into an efficient integrated system-level protection mechanism. We give an overview of the security framework structure which represents the core of our proposal. We show the composition and structure of the system which is organized in hierarchical fashion based on security agents as elementary building blocks (as presented in Section 5.1). The hierarchical organization of the protection system has been chosen as it best suits to the need of coordinating different cores' protections strategies with an unique system-wide protection mechanism: the proposed solution is discussed in Section 5.2. Furthermore, the agent based structure proposed by us balances centralized and distributed solutions considering energy dissipation and area consumption. Contrary to other solutions (such as Patel et al. [2010]) no code instrumentation is required which simplifies the deployment and improves portability of the solution. Results and comparisons between the proposed and other approaches are reported in Section 7.

In addition to the architecture, a comprehensive security policy embracing all the different protection mechanisms has been developed and presented. It is based on trusting values calculations and accordingly security domains assignments as shown in Section 5.4. We explain in the sequel the most important elements of the proposed security framework.

5.1 Security Agents

Modern MPSoC systems represent very complex, heterogeneous systems with highly dynamic interactions among applications. Moreover, security related interactions (e.g. attack propagation) introduce another degree of nondeterminism in system behavior. Therefore, due to high complexity, dynamism and heterogeneity of the considered system, adoption of the multi-agent system (MAS) concept (Wooldridge [1997]; Jennings [2001]) appears to be a promising concept to tackle the listed issues. MAS has been proven as successful instrument in dealing with similar problems in a number of different fields (Borselius [2003]; Xiao et al. [2007]; McArthur et al. [2007]; Dimeas and Hatziargyriou [2011]) as well as in security related ones (Bonhomme et al. [2010]; Feltus et al. [2010]).

The Security Framework that we propose is based on a logical entity named *Security Agent*. Security agents are logical elements (built in hardware or software or with a mix of hardware and software) that encapsulate all the security related issues at different levels of design. Their functionalities depend on which level they are employed at (i.e. which purpose they serve, see Figure 5.1); they include attack specific-protection, communication with other agents and countermeasures, as detailed in Section 5.2. We devise security agents according to requirements listed in Section 3.2. Therefore, the agents are conceived as flexible, modular structures that can be easily extended with extra components for additional functionalities (i.e. handling some newly discovered attacks) by straightforward addition of an extra module. Once a new module or agent is inserted in the system it 'checks-in' to the security framework and the systems gets aware of its presence and role.

More precisely, agents and supporting structures are developed to be:

- **Comprehensive** - by implementing the agent structure in a flexible fashion we provide high autonomy to the front-end agents attached to the individual cores. In turn, each front-end agent can be tuned to support specific protection needs of a particular core (please refer to Local Security Agent in Section 5.2). On the other hand, inter-agent communication and interaction is standardized and unified. In this way, we build a security framework capable of meeting the requirements of heterogeneous MPSoC composition as well as the ones related to a wider spectrum of security aspects that may arise from such structure.

- **Modular** - the agents are built as easily extendable modular structures. In fact, the agents' architecture is designed to support in a flexible way possible changes in the number of employed agents and as well as to facilitate

insertion of new attack specific protection modules. For this reason each agent has an updated list (actually *coordination table*) of its directly subordinated agents (which belong to a lower layer in the hierarchy). Upon insertion in the system newly employed agent checks-in to the *coordination table* and the information is subsequently propagated to higher layers' agents so that the security framework as a whole is aware of the new situation. This feature enables facilitated and undisturbed extension of the protection system to different types of attacks.

- **Scalable** - the security framework can be also easily extended to accommodate the addition to the system of new cores. This is again enabled by properties of the modular structure of the proposed architecture. In fact, insertion of a new core implies addition of a newly assigned agent (attached to the core) to the system. Practically, only the *coordination table* needs to be updated through the check-in process and the entire security framework is adjusted to support the change accordingly.

- **Portable** - we encapsulate the architecture-specific issues in the communication interface of the front-end agents (i.e. attack specific agents - which are actually at the same time 'core specific', as explained in Section 5.2). These interfaces have to be customized to adopt the communication standards of the specific cores. This fact represents the limitation of the solution and at the same time it introduces the need for additional design efforts in case of deployment of the solution on other systems. Nevertheless, 'translation' of the solution to other architectures requires intervention only in this part (front-end attack specific agents), beyond this point the entire structure is fully architecture independent and portable.

5.2 Hierarchical Structure of Security Agents

We define four different security levels and we assign one type of security agent to each layer. These four types of agents are organized in a hierarchical fashion as follows (the general system structure is represented in Figure 5.1):

- An Attack Specific Agent (ASA) is a dedicated software or hardware structure that handles a specific attack (e.g. denial of service, code injection etc.). The corresponding security level is defined at Attack Specific Protection layer. Particular implementations at this level are described in details in Section 6.3.

- A Local Security Agent (LSA) takes care of securing the individual core, of aggregating and coordinating all ASAs deployed on the core and of communicating with other Security Agents that are performing specific actions as countermeasures. There is one LSA per core. It logically encapsulates all security related activities defined at core protection layer. The detailed description of the structure and implementation of LSA is given in Section 6.4).

- A Cluster Security Agent (ClSA) is a module that coordinates LSAs assigned to cores inside a cluster in those MPSoC architectures designed as interconnections of autonomous clusters (ClSAa correspond to the cluster protection layer). For simplicity reasons, but without loss of generality, we have not implemented this type of Agent in our experimental setup (the FPGA-based architecture used for the experimental validation is a single-cluster MPSoC).

- A Central Security Agent (CSA) represents a central point in the system security. It operates at the system level protection layer. It practically executes security policies (i.e. updates trusting tables, performing countermeasures etc.) and coordinates all the security activities in the system through LSAs or hierarchically through ClSAs for cluster-based architectures (structure and implementation of the solution is detailed in Section 6.5).

All the agent types are implemented in modular and scalable fashion in the form of easily extendable finite state machines.

5.3 Communication among Security Agents

As stated in Section 1.2 we consider an MPSoC as a structure composed of three basic building elements - processing cores, memory units and communication medium. As it will be shown in Section 6.3, memory units and processing cores have been secured by assigning proper Local Security Agents. Still, these agents need to communicate, in order to provide system level protection. Therefore, sensitive security related data have to be protected and possibly kept out of an attacker's reach.

In order to build an efficient and reliable security system, an appropriate communication structure must be designed with the goal of providing safe and secure communication among all security agents. To support such requirement,

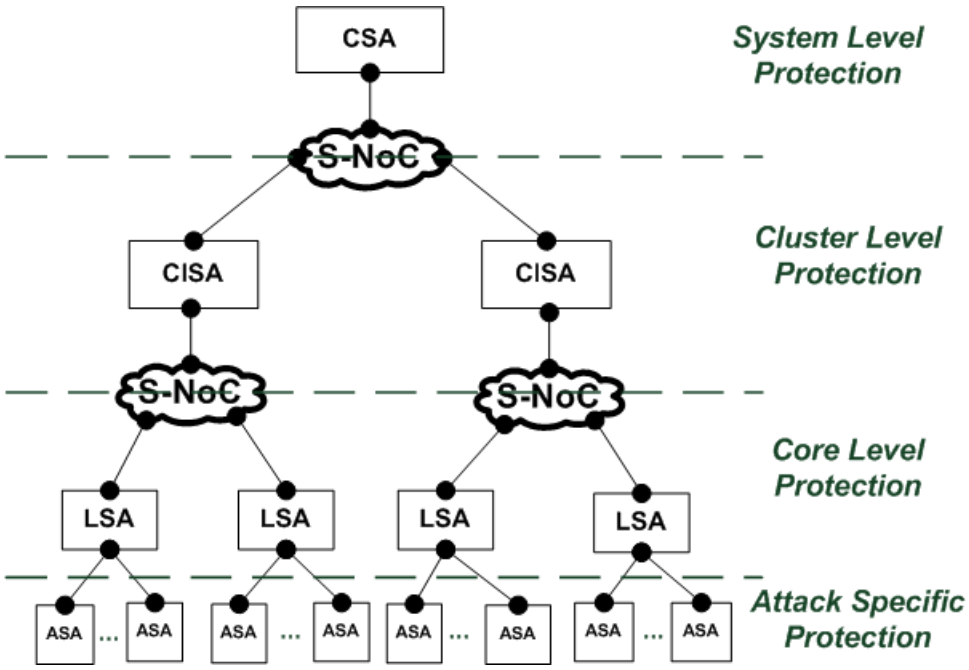

Figure 5.1. Hierarchical structure of the security system

we have introduced a *Secure* NoC (S-NoC) parallel to the 'regular' NoC that supports the standard application (SNoC implementation is described in details in Section 6.6). In this way, the security-related traffic is separated from the regular data traffic, thus providing a further degree of protection since the security-related data are kept fully isolated in the dedicated architecture. As a consequence, all security related components and services are encapsulated in a single independent structure. An alternative for autonomous security-related data routing would be utilization of virtual channels in the regular NoC, but apart from higher risk of data interception, the design efforts as well as cost of implementation in that case would be even higher (this statement especially holds for MPSoCs with limited numbers of cores as shown by Yoon et al. [2010]).

The communication protocol on the secure NoC is designed to be rather straight-forward. In the initial phase, the communication is started and established by the CSA; afterwards, the communication is effected in event based fashion (the UML like *sequence diagram* in Figure 5.2 shows the communication protocol among CSA and LSAs). In order to simplify the diagram, ClSAs are not presented in the figure since they are not implemented in our experimental set-up. Since, in the current implementation, ASAs are implemented as modules directly embedded in the LSAs, no explicit communication between these two

is presented in the diagram (internal communication is also event based). Nevertheless, CSA performs initialization of all LSAs and CSA (e.g. setting initial parameters of trusting table). Once the initialization phase is over, the system is run and its operation moves into regular execution (LSAs send reports and CSA updates the trusting table accordingly). When a violation of security policies is detected the system moves into another operation mode executing attack analysis. Upon violation is resolved the CSA reprograms the system and restores its regular operation mode.

5.4 Security policy

The security policy in the framework is based on three elements:

- Evaluation of incidents (i.e. disruptions) and security alert severity calculations
- Trusting relationships - trusting values calculations and assignment to each MPSoC
- Designing the security domains and deciding the thresholds between them; determining accordingly the degree of threat/violation and proper countermeasures

All the above issues are combined together to form an efficient protection system. They are determined statically (at design time) requiring that some basic information on the attack characteristics must be known in order to optimally customize the security policy elements. For instance, application-specific traffic statistics should be known so that protection mechanisms for some specific attacks (such as Denial-of-Service) can be optimally tuned.

5.4.1 Disruptions, security alerts and positive feedback

We consider as a *disruption* any kind of unexpected event in the system. This includes all types of unauthorized requests for resources or services. It must be noted anyway that not all disruptions come from malicious attacks, as many may be caused by various software bugs or system failures.

We consider in general two kinds of attack detection:

- Deterministic attack detection - enabled through customized logic that identifies specific violations according to a given set of rules

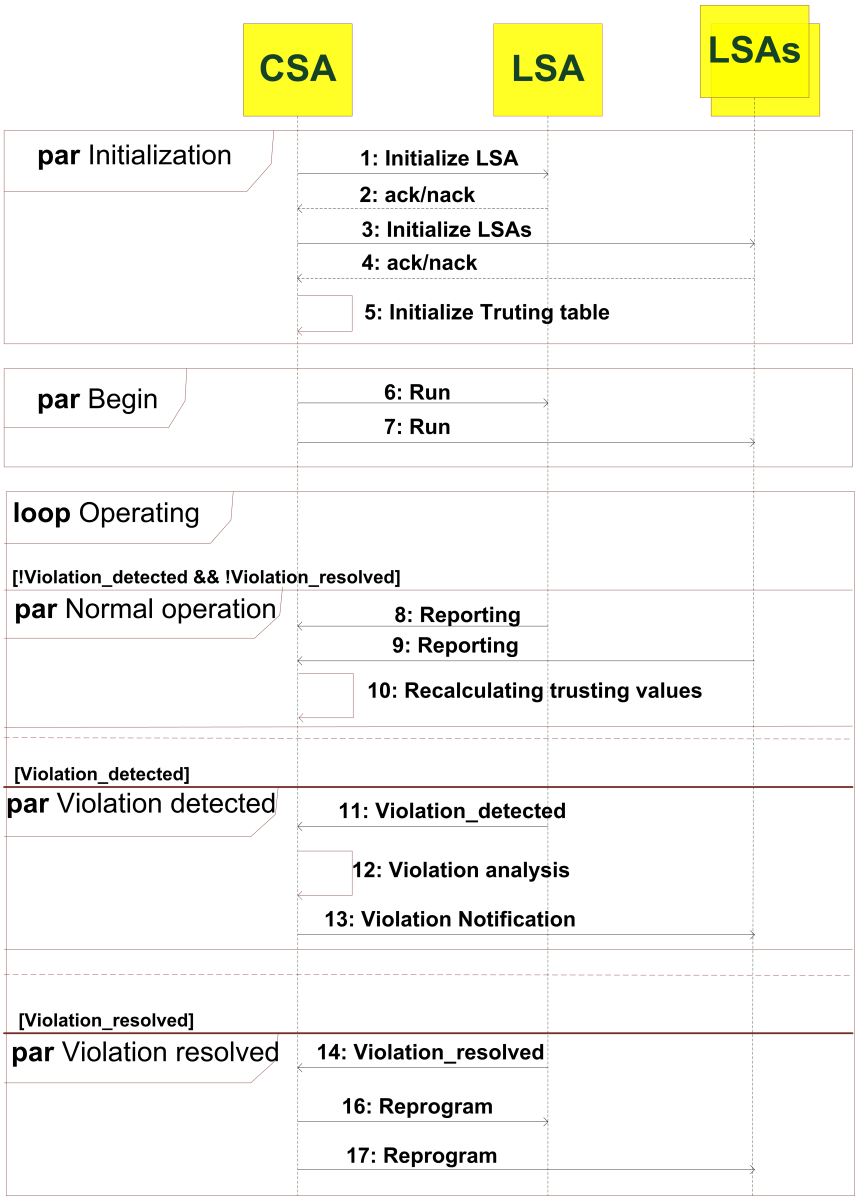

Figure 5.2. Communication protocol among security agents

- Statistical attack detection - achieved by setting up suitable 'misbehavior' patterns and accordingly detected violation conditions based on system monitoring statistics

Obviously for the second type of attack detection techniques, actually distinguishing between malicious disruption and a regular application execution incident is a very challenging task. In order to distinguish between intentional incidents and malicious attacks a customized solutions for each type of attack has to be made (e.g various types of applications have very different communication patterns and this in turn requires customization of attack detection parameters for DoS detection). Such kind of solutions are integrated in Local Security Agents (LSAa) who are in charge of making the decision on alert severity levels.

Alert severity levels can be given in various resolutions, but for simplicity reasons and without loss of generality we stick to three levels, i.e., low, medium and high level. Such division enables us to group and process disruptions (calculating and then forwarding the appropriate alert level) locally in LSA. This way, we transmit less often information on the SNoC, thus unburdening it from traffic and saving the energy at the same time.

Positive feedback

As already mentioned above, some attack detection techniques that rely on application behavior monitoring might be misled by unexpected irregularities and raise a false alert. In particular this applies to Denial-of-Service attack as described in Section 2.3.1. In order to prevent the possibility that such false alerts cause permanent trusting value decrease we introduce and implement *positive feedback* mechanisms. These mechanisms must be tailored to specific applications as well. Nevertheless, their purpose is (once activated) to monitor appearance of specific events in defined time windows (usually after an alert has been raised or when the core gets into specific security domain as defined in Section 5.4.3). If none event has been recorded during the monitored period the positive feedback signals is sent to the CSA and this results in incrementing the trusting value for the considered core.

5.4.2 Trusting policy

We implement a centralized trusting policy. All trusting related matters are managed by CSA. It performs the following functionalities:

- Defines levels of trustiness which correspond to *security domains*
- Defines threshold values between different security domains
- Assigns initial trusting values to the cores
- Updates trusting values in the trusting table

At system initialization time all cores are given highest trusting value. Definition of both the exact trusting value as well as its initial rate can be decided by the system designer at design time. Upon reception of each alert (or positive feedback) the trusting is recalculated. The assigned value is decreased/increased according to severity of the alert (or value of the positive feedback) and once it gets zero the CSA performs proper countermeasures (e.g. cutting the access to the NoC to the attacked core).

Considering in particular DoS detection, trusting value can also be increased if no alert has been detected in the given time period after the first detected alert. Once the alert is fired (a medium-severity alert pushes the core into medium security domain) the counter in LSA is started and if no other alert arrives in the specific time-window a '*positive-feedback*' signal (which increases the trusting value) is sent to CSA. The alerts can be of low, medium and high severity (decreasing the trusting value, in our specific implementation, for 5, 10, 15 respectively). Positive feedback signals correspond to a trusting value increment of two points in our specific implementation.

Equation 5.1 shows how the trusting value (TV) for the specific core has been calculated. In this equation $TV(i)_T$ represents the trusting value of the specific core in some instant of time. $TV(i)_0$ is the initial trusting value and in our implementation it is the maximum value as well. The trusting value for the specific core is calculated as deduction of all reported alerts for the core with addition of all the positive feedbacks for the core. We have two sums in this case:

- All the types of alerts which correspond to all the kinds of implemented attack specific agents for the specific core
- All the positive feedbacks which again may come from all the different attack specific agents

In general, this means that for a specific core, all reported alert types from all existing cores (as well as positive feedbacks) are summed up to give a proper trusting value, as indicated in equation 5.1 (where m denotes all the types of

alerts and n is the total number of the cores; $Alert(X_i)$ represents the alert of the specific attack type for the given core):

$$\begin{aligned} TV(i)_T &= TV(i)_0 - Alert(X_i) + PF(X_i) = \\ &= TV(i)_0 - \sum_{j=0}^{m}(\sum_{i=0}^{n} Alert(X_i)) + \sum_{j=0}^{m}(\sum_{i=0}^{n} PF(X_i)) = \\ &= TV(i)_0 - \sum_{j=0}^{m}(\sum_{i=0}^{n} Alert(X_i) + \sum_{i=0}^{n} PF(X_i)) \end{aligned} \tag{5.1}$$

Alert detections (as well as positive feedbacks reporting) can be consequences of differently implemented protection mechanisms. Some of them are implemented in the form of attack specific agents that monitor the core they are attached to and report alerts on violation detection coming from the core they are assigned to (e.g. code injection). On the other hand, there are attack specific agents which detect attacks to the cores they are attached to that are coming from other cores (e.g. unauthorized memory access). In some cases both types of agents can be employed in the same core (i.e. embedded in the same local security agent). Therefore, every LSA may report different alert types for all cores in the system (including the one the LSA is attached to). That's why in the sum in equation 5.1 we consider all the cores and all the alert (actually protection) types. Thus, the trusting value for each core is calculated based on alerting/feed-backing signals from LSAa at all cores in the system.

Such an approach enables to combine a wide diversity of attack specific protections for an unlimited number of cores. We establish thus in a simple and elegant way a structure that correlates all the types of protections into a single system level security strategy. Still, alert severity weighting and design of positive feedback mechanisms must be customized for each attack itself fairly related with other protection types and integrated into a single security matrix (in which each alert for every core is weighted properly).

5.4.3 Security domains, violation detection and countermeasures

In order to optimally manage system operation from security aspect, we have introduced *security domains* inspired by those proposed by (Inoue et al. [2005]; Hiroaki et al. [2008]; Porquet et al. [2009]). In general, the concept of security domains is based on confining application execution in specific environment according to importance and trusting of the application and the core that is to execute it (as explained in Section 2.3). These solutions normally requires implementation of *virtualization* of execution environment. As we have on available

limited MPSoC resources, such a solution (fairly costly in terms of processing power and memory) common in general purpose computing would not be applicable. In our specific solution we consider mapping of individual cores into proper security domains rather then applications or threads. Thus we have lower granularity of instances that we are processing but still without loss of generality of the solution.

Different *security domains* are defined at the design time, and the cores in the system are assigned to appropriate domains in run-time. We have established - high, medium and low security levels corresponding to actual level of the trusting value of the core. Definition of **thresholds** between the different levels (i.e. trusting values which define specific security domains) is another challenge and it should be left to the designer of the system based on particularities of the system and possible attacks. In our concrete implementation we have taken uniform distribution of threshold values, meaning that cores with trusting values in the upper third (i.e. 11-15) are considered as highly trusted and medium or low trusted cores are accordingly grouped.

Depending on its current trusting value, a core may belong to one of the defined security domains and it would be threated accordingly. In our implementation belonging to a security domain impacts on the way a core is monitored, the different access rights to shared memories it may have and finally on the access to the NoC itself. More specifically there are three possible options:

- High security domain - all the resources of the system are available to the core and it can execute all the application types. The security monitoring system (which involves activation of positive feed-backing mechanism) is switched off. In other words only the basic alert detection mechanisms are active for such a core. This way we avoid unnecessary energy dissipation, as if a core is considered as highly trusted there is no need for improvement of its trust value or some other additional checks.

- Medium security domain - security monitoring for the core in the domain is activated (i.e. positive feed-backing is enabled); also some restrictions to specific memory regions may apply (this depends on specific system implementation and it is defined at system design time)

- Low security domain - additional restrictions to shared memory access are applied. All the cores are sent warnings on the potential threat coming from this core

Once the trusting value reaches zero we consider that security violation is definitely detected. The core is cut off from NoC access until the cause of the vi-

olation has been resolved. More exact countermeasures go beyond our research as they are attack and technology depended. We implement very basic countermeasures that are aimed towards attack isolation and preventing its propagation through the system.

Part II

Implementation and Validation of the Proposed Solutions

Chapter 6

From the General Approach to Actual Architectural Design

In this Chapter we present the approaches and instruments used in the process of final design implementations. We discuss performance and limitations of the existing instruments as well as proposed enhancements applied to the existing SoC design flow. We show implementations of different types of security framework components, actually agents - Attack Specific Agents (ASAs), Local Security Agent (LSA) and Central Security Agent (CSA). The structure and functionalities of each of the agents is explained in details.

6.1 SoC Design Flow and Fast Prototyping Strategies

SoC platform design in general requires hardware-software co-design decisions to be made. It requires evaluating trade-offs among different merits and constraints such as area costs, power consumption, performance, flexibility, time to market, design time etc. Once the system structure is defined, the design flow involves parallel development of hardware and software components of the system (this process is represented in Figure 6.1). When it comes to design synthesis, CAD tools generate hardware components (through place and route process) first, and only after the hardware platform of the system is built SW blocks are integrated by software development environment (in our particular case we have used for hardware generation the Xilinx provided tools - ISE, EDK, SDK etc. (Xil [2005]) - as explained in Section 6.1.1)

In order to use available resources more efficiently, designers of NoC-based MPSoC architectures need tools allowing reliable and fast validation and testing strategies for system debugging and development. Automation of the design

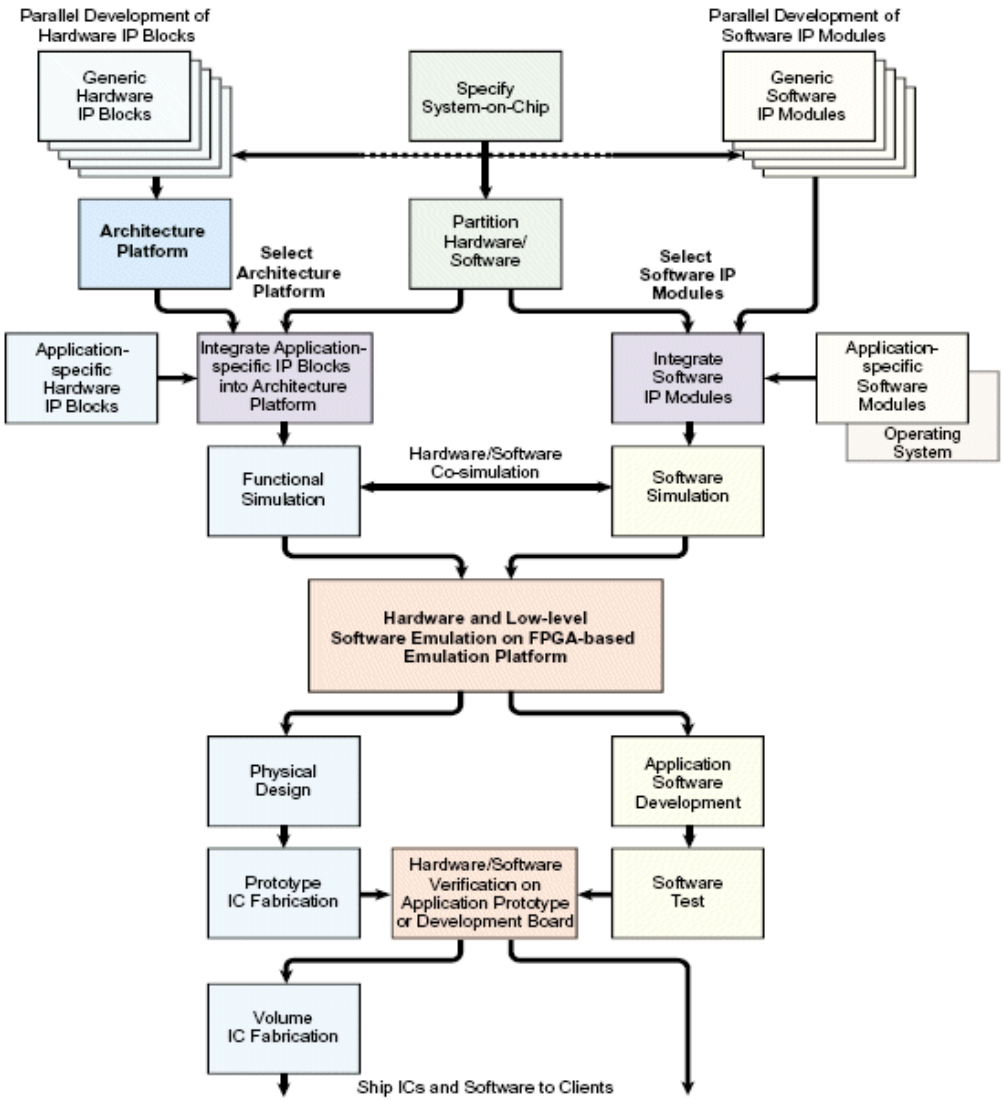

Figure 6.1. Standrad SoC hardware-software design flow (taken from Bishop [n.d.])

flow is the solution to reduce the time spent in these activities, in particular in the case of reconfigurable NoC-based multiprocessor architectures.

Automation in the generation of processing nodes in multiprocessor SoC is discussed in Section 2.5. We have developed a light-weighted framework to perform automatic generation of MPSoCs based on on-chip networks (Lukovic and Fiorin [2008]), in particular addressing FPGA-based design of such systems. Compared to previous related work, the framework described in this paper extends for the first time the functionalities offered by the Xilinx EDK 7.1i tool-chain, in order to give support for the fast automatic generation of NoC-based MPSoCs employing the processing elements provided by Xilinx. The framework provides the system designer with the possibility to quickly develop prototypes that can be significantly helpful to test, validate and debug the type of system addressed, thereby reducing the time for the development of a project.

6.1.1 The Standard FPGA design flow

In this Section we briefly present the characteristics of the Xilinx EDK design flow, describing its advantages and disadvantages in the case of NoC-based MPSoCs design.

We refer to EDK versions 7.1i and later, used for the development of our work. The tool-chain involves hardware and software design flows, integrating a complete platform generation in one framework. It is composed of many different tools, with functionalities ranging from IP-core insertion wizards to programs exploiting complex algorithms to perform the place-and-route of the design. The EDK functionalities are presented to the end-user through a graphical user interface - the Xilinx Platform Studio (XPS).

The entire EDK system generation relies on information written into several project-specific description files. Platform-related information is written in the Xilinx Microprocessor Project (XMP) file, representing a central point for the entire project. Specifications of system hardware components for each IP instance are described in the Microprocessor Hardware Specification (MHS) file, and software properties (e.g. specific drivers) of each of them are given in the Microprocessor Software Specification (MSS) file. The MHS and the MSS files are at the top of system description hierarchy. Features and characteristics of each kind of IP included in the project are described in the Microprocessor Peripheral Description (MPD) file.

Each IP instance in the system, described by means of a parameterizable MHS file, receives the adequate parameter value from a dedicated description record in the MHS and MSS files, where all components of the system are listed.

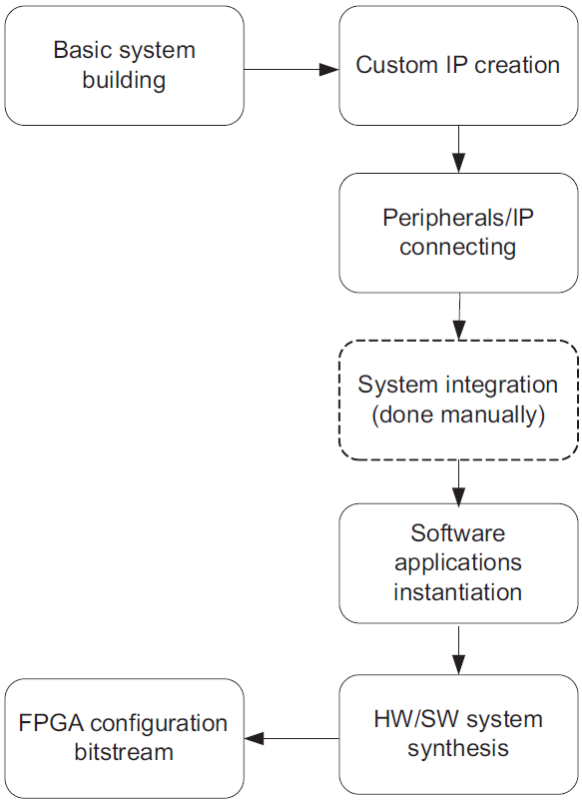

Figure 6.2. The Xilinx EDK design flow

The external input-output connections of the system are defined in the User Constraint File (UCF) Xil [2005].

EDK merges the creation of the software and hardware parts into one integrated system (as presented in Figure 6.2). Its real power lies in the clear representation of the entire HW/SW system it gives to the user, as well as in the significant number of IPs provided by the FPGA vendor. Insertion of provided IP-cores as well as of the custom created ones is facilitated by tools for IP integration.

While very comfortable for automatic generation of a system based on the provided processors and IPs, EDK is not well-suited for multiprocessor platforms based on custom IPs, in particular if connected through non-standard interfaces. In fact, for non trivial architectures including a mix of custom modules and Xilinx IP blocks adopting a complex communication system, EDK often requires designers a time-costly manual modification of the system files, with possible generation of subtle errors during the procedure. Therefore, in the next section we focus on possibilities for enhancing the existing EDK design flow for

automatic generation of NoC-based MPSoC systems, proposing and describing the integrated automation framework we implemented on top of the EDK tool-chain.

6.1.2 Network-on-Chip adjusted FPGA design flow

The integration of multiple instances of the same IP and their connection through not-standard interfaces requires additional efforts from a designer as all the links (i.e. ports, buses etc.) among the components have to manually inserted. This represents a significant problem in the generation of NoC-based MPSoCs on FPGA and it reflects at increasing the prototyping time for the system. An efficient approach for cores interconnection can be developed by knowing the exact number of inserted IPs and their communication interface. Based on such approach, we apply a set of rules for IP connections in order to automate the manual work that should be otherwise performed. In this way, we fill up the gap in existing design flow between the connection of peripherals/IPs and the instantiation of software applications (as show in Figure 6.2).

The automation engine we propose frees system designers from taking care of internal system interconnection as well as from the need of knowing the EDK tool-chain and the format of system files by providing an efficient components integration instrument (Lukovic and Fiorin [2008]). These improvements bring more efficiency in system design facilitating testing and validation phases.

The automation engine we present here is conceived as a three-layer structure (see Figure 6.3). Each layer corresponds to one phase of the design flow. The core of the solution relies on a Tcl language script that creates the required directory structure, and designs the specific project files and makefile. The fundamental benefit brought by this approach is the fact that it enables system designers to be aware of just system functionalities without needing knowledge of its internal structure. The enhancement of existing design flows enables the full integration of the several components involved in system generation, reducing the knowledge about the specific tool-chain needed by users. All the needed specific description files of the project as well as the makefile that drive the process of synthetization are generated in few minutes, after defining the characteristics of the system to implement.

System Initialization

The outer layer - the system initialization - provides direct interaction with the system architect in the form of text user interface. Moreover, relying on

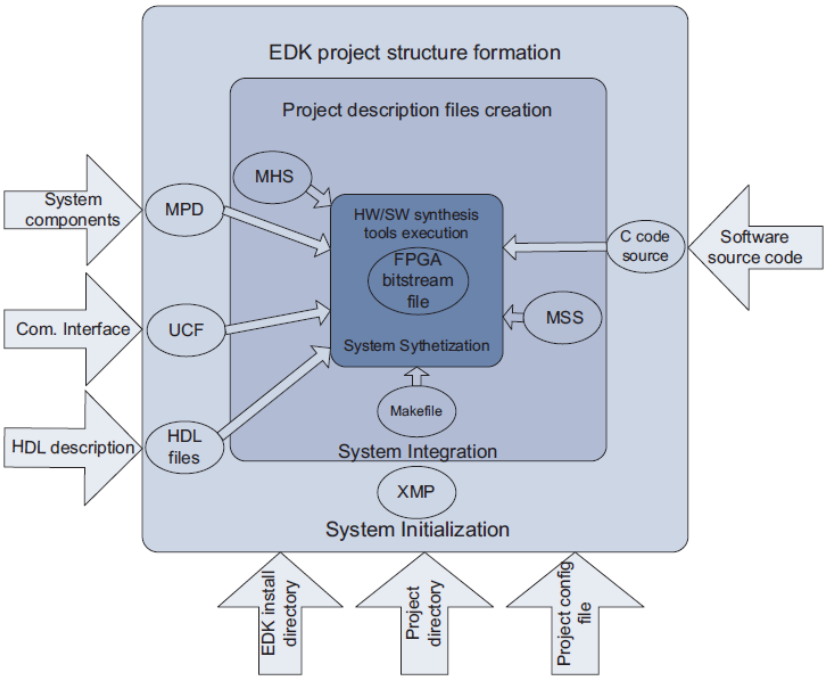

Figure 6.3. The layered structure of design flow

Xilinx EDK, our framework has to provide full compatibility with appropriate structure in sense of directories tree and system description files. Therefore, the task this layer performs is the fast automated creation of project folders and files based on system designer preferences. The framework takes as inputs - destination paths to EDK install directory, project directory, paths to HDL source code of custom IP and C source code of software application, as well as the number and type of system components and type of interaction between the host and platform. Additionally, debugging information in form of signals to be monitored could be provided. All these data are used for creating appropriate MPD and UCF files and binding of custom IPs' HDL and applications' C source files to the adequate points in the structure. The project configuration file - written in the XMP file - carries information on the hardware platform used. This process directly corresponds to 'Basic system builder' and 'Create/Import Peripherals' in XPS design flow (see Figure 6.2).

System Integration

After the creation of the required project structure, the design flow moves onto the next stage - the system integration layer. The designer's preferences are now presented in EDK understandable form, which enables the process of components interconnection/integration. As the automation engine at this stage knows all the components of the system, it creates the proper number and types of NIs and appropriately associates them to the given components. The customizable routers are also configured according to the total number of inserted IPs. All IPs are assigned an identification number depending on their instantiation. The same number is labeled to the associated NI and router input-output port. The framework connects the suitable components following the assigned numbers. The IP's are labeled according to the order of instantiation, in a way allowing the user to know exact 'address' of each component in the system.

Components' interfaces in MHS files are now assigned according to described rules for NoC interconnections. In this way, the NoC is instantiated among the Xilinx and custom IPs. This step represents an essential enhancement to EDK design flow as it grants designers full freedom from interconnecting issues.

System Synthesis

Once the system is fully integrated, synthesis begins. The makefile created in previous phase drives the execution of HW/SW sinthetization tools of the EDK design flow. Hardware flow is run first. After system Netlists creation, the implementation flow is executed. Then, the bitstream file is generated and then the software flow is run. This phase consist of three steps - adding a software application to the desired processor, generation of custom libraries and compilation and linking of source code. Once both hardware and software flows are executed, the bitstream file is initialized with BRAM data (for initialization of data instruction memories attached to processing units).

The final result of the automation engine is a configurable bitstream file which is directly downloaded to the attached platform.

6.2 Network-on-Chip as a medium to accommodate security related enhancements

Being modular, scalable and technology independent, Networks-on-Chips represent a solid ground to provide MPSoCs with extra services (in addition

to inter-core communication). In fact, NoCs represent a solid basis to meet requirements of a security framework listed in Section 3.2. The NoC interconnect comprises two types of components, namely: *Network Interfaces* (NIs) and *Routers*. We consider in the present work reference NoC architecture as detailed in Section 4.1. Naturally, NIs as front-end components of NoCs, attached directly to the cores, represent the optimal point for hosting additional services.

6.2.1 Enhancing Network Interfaces

As discussed in Section 4.1.2, the Network Interfaces (NIs) act as adapters of communication standards between the various IP cores and the NoC. The NI module, acting as an interface between the IP and the interconnecting structures, encapsulates all the issues related to a reliable and efficient transmission of data through the network, freeing the IP of any communication related concerns. The NI is in charge of structuring data as packets or flits, and it also manages transmission of necessary control flow information.

NIs represent a gateway to the NoC and due to their modular and flexible design as well as to close interaction with cores they are attached to, they are an ideal point for deployment of modules enabling services other than communication related ones. We embed Local Security Agents (that actually host Atack Specific Agents) in NIs in such manner that regular functionalities are not disturbed.

6.3 Attack specific protection

The combination of various MPSoC design decisions and operating environment may expose the system to critical combinations of security risks as shown in Section 3.1. Attack techniques may be easily merged and coordinated, a fact representing an additional challenge for security aware design.

Among the most widespread types of attacks we consider:

- Code injection
- Denial of services
- Side channel attacks

We show in the sequel protection solutions we implemented (against the above listed attacks) and their integration in wider security framework.

6.3.1 Attack Specific Agent - Data Protection Unit

We introduce a concept (named as 'MemPROT') for protecting shared memories from unauthorized access. It is based on the Data Protection Unit (DPU) which is a hardware module that enforces access control rules specifying the way in which a component connected to the NoC can access the blocks in which a memory can be divided to allow separation between sensitive and non-sensitive data of different processors (Bjerregaard and Mahadevan [2006]). The module is integrated in the Local Security Agent (as shown in Section 6.4.3) embedded in the Network Interface of the target memory (or of the memory-mapped peripheral) to supply services similar to those offered by a classical 'firewall' in data networks. The Network Interface receives packets coming from several initiators requesting access to the target memory. While processing the packet, the information contained in the header is passed to the DPU. The protection module looks up the access rights for the requesting packet and checks if the requested operation is allowed, granting or denying the access of the data to the memory block. The internal structure of the DPU is given in Figure 6.4. One of the most relevant parts of the DPU is represented by the lookup table. In hardware this element is commonly implemented combining a typical Content Addressable Memory (CAM) (Guz et al. [2006]), used in associative memories and data networks routers, and a RAM storing the access rights (load, store, both or none). It is important to note that coupling the DPU with the NI guarantees that no additional latency is associated with the access right check since, as we will show better later, the protocol conversion and the DPU access are performed in parallel.

Figure 6.4 shows the architecture details when the DPU is embedded at the target NI (attached to memory). For this architecture, the DPU checks the header of the incoming packet to verify if the requested operation is allowed to access the target. This access control is done mainly by using a look-up table (LUT), where entries are indexed by the concatenation of the *SourceID*, the type of information $\overline{D} = /I$, and the starting address of the requested memory operation MemAddr. The number of entries in the table depends on the number of memory blocks to be protected in the system, as well as on the number of initiators. In the implementation shown in Figure 6.4, we assume 4 Kbytes as the size of the smallest memory block to be managed for the access rights.

This means that all data within the same block of 4 Kbytes have the same rights (corresponding to the 12 LSB in the memory address) and that we use only the 20 most significant bits of the *MemAddr* field for the lookup. The LUT of the DPU is the most relevant part of the architecture and it is composed of

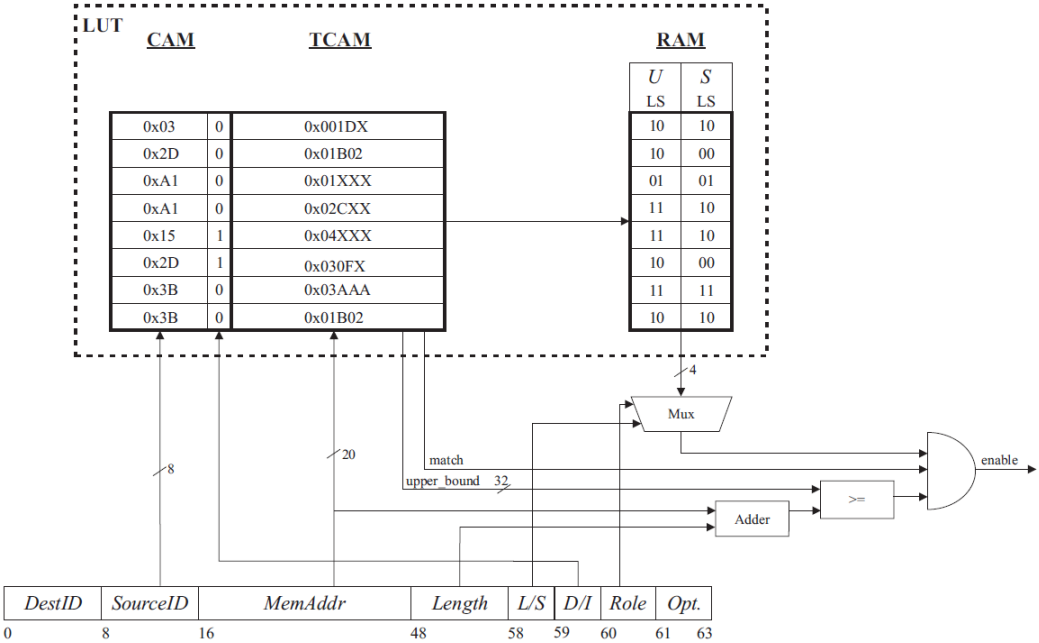

Figure 6.4. Data Protection Unit - internal structure

three parts:

- A Content Addressable Memory (CAM) (Radulescu et al. [2005]) used for the lookup of the *SourceID* and the type of data $\overline{D} = /I$,
- A Ternary CAM (TCAM) (Radulescu et al. [2005]) used for the lookup of the MemAddr. With respect to the binary CAM, the TCAM is useful for grouping ranges of keys in one entry since it allows a third matching state of 'X' (Don't Care) for one or more bits in the stored datawords, thus adding more flexibility to the search. In our context, the TCAM structure has been introduced to associate to one LUT entry memory blocks larger than 4 Kbytes.
- A simple RAM structure used to store the access right values

Each entry in the CAM/TCAM structure indexes a RAM line containing the access rights $allowed/\overline{notallowed}$ for user load/store and supervisor load/-store. The type of operation $\overline{L}/S$ and its role $\overline{U}/S$ taken from the incoming packets are the selection lines in the 4:1 multiplexer placed at the output of the RAM. Moreover, a parallel check is done to verify that the addresses involved in the data transfer are within the memory boundary of the selected entry. If

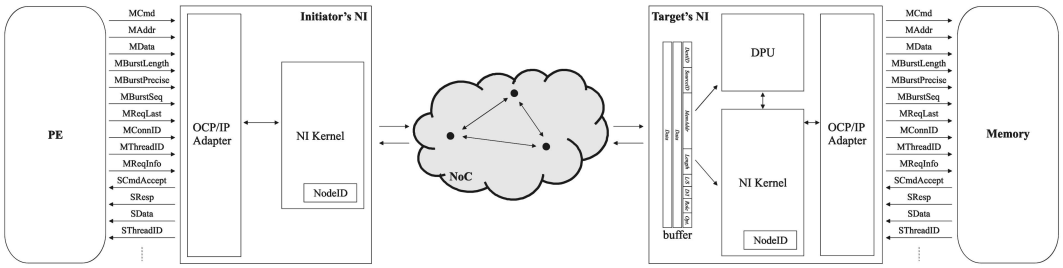

Figure 6.5. Overview of the whole NoC-based architecture including the Data Protection Unit at target Network Interface

the packet header does not match any entry in the DPU, there are two possible solutions, depending on the security requirements. The first is more conservative (shown in Figure 6.4), avoiding access to a memory block not matching any entry in the DPU LUT by using a match line. The second one, less conservative, also enables the access in the case when there is no match in the DPU LUT. This corresponds to the case when a set of memory blocks does not require any access verification. The output enable line of the DPU is generated by a logic AND operation between the access rights obtained by the lookup, the check on the block boundaries, and, considering the more conservative version of the DPU, the match on the LUT. Given the complexity of the protocol conversion to be done by the NI kernel, we can assume that the DPU critical path is shorter than the critical path of the NI kernel (as confirmed in the results reported in Section 7). Under this assumption, integrating the DPU at the target NI guarantees that no additional latency is associated with the access right check since, as shown in Figure 6.5, the protocol conversion and the DPU access are done in parallel.

6.3.2 Attack Specific Agent - Stack Protection Unit

We introduce a protection mechanism against 'code insertions' attacks named InjectPROT. The solution is based on a Stack Protection Unit (SPU).

The basic concept of the ASA/SPU is rather straightforward. Every time a function is called, the return address of the caller function is stored on the stack but it is also replicated in the ASA/SPU which keeps track of the return addresses of the nested functions and reports any attempt at overwriting that particular address (simply comparing stored value with one obtained from the stack). Practically, the task of the ASA/SPU is to restrict the stack accesses to the current stack frame. The concept is presented in Figure 6.6.

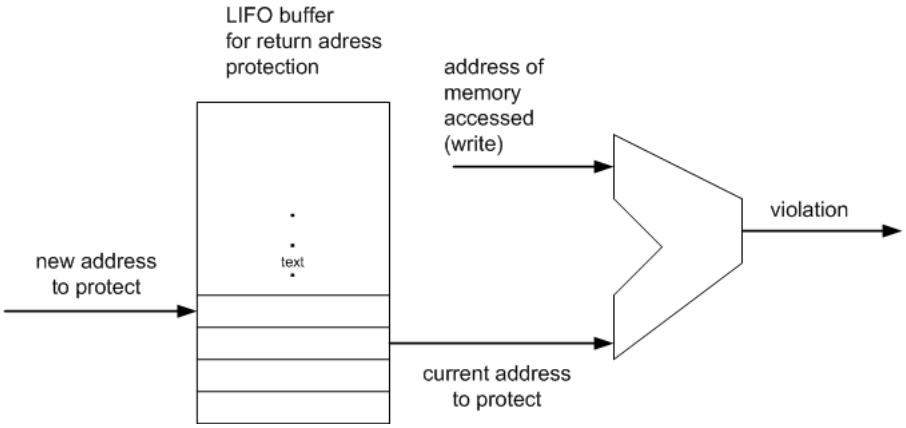

Figure 6.6. Basic concept of Attack Specific Agent - Stack Protection Unit

ASA/SPU customization for MicroBlaze processor

The ASA/SPU has to keep track of return addresses and it should store them on each process context switch or process execution mode switch (*user-kernel* execution mode shift). While the concept is totally general, the implementation is strongly CPU-specific, as the exact machine instructions and event sequences devised for function call and return must be matched step-by step by the ASA/SPU. The scenario of context switching and instruction that handle it in the MicroBlaze architecture are shown in Figure 7.5. The ASA/SPU should follow such scenario and on a context switch (that may occur for different reasons such as function nesting, interrupt handling, exception management and so forth) it should store the return address as well as verify return address correctness on function return.

Using a *brlid r15*, instruction the current value of the Program Counter (PC) is placed in the *r15* (pointer link register), and the PC is replaced with the address of A function. In the A function the instruction $addikr1, r1, -x$ is introduced to prepare the current stack frame (i.e. the stack frame space used by the function). After the stack pointer is decreased, the *r15* register is stored at the beginning of the current stack frame. This is the return address for the function A. These are the necessary operations to enable arbitrary nested function calls. In order to return from the called function, the instruction to restore the *r15* register is issued. Instruction *lwi r15, r1, 0* , restores the program counter to the instruction next to the return address (*rtsd r15,8*) and cleans up the stack frame by $addikr1, r1, x$. The SPU must follow this sequence and store the nested return addresses.

The ASA/SPU has been realized as a Finite State Machine (see Figure 6.7).

Upon initialization the SPU goes to *Idle* mode (or state) in which it waits for the instruction from the Kernel. The command which signals that context switch has occurred or that execution mode has been changed drives the ASA/SPU in an appropriate state.

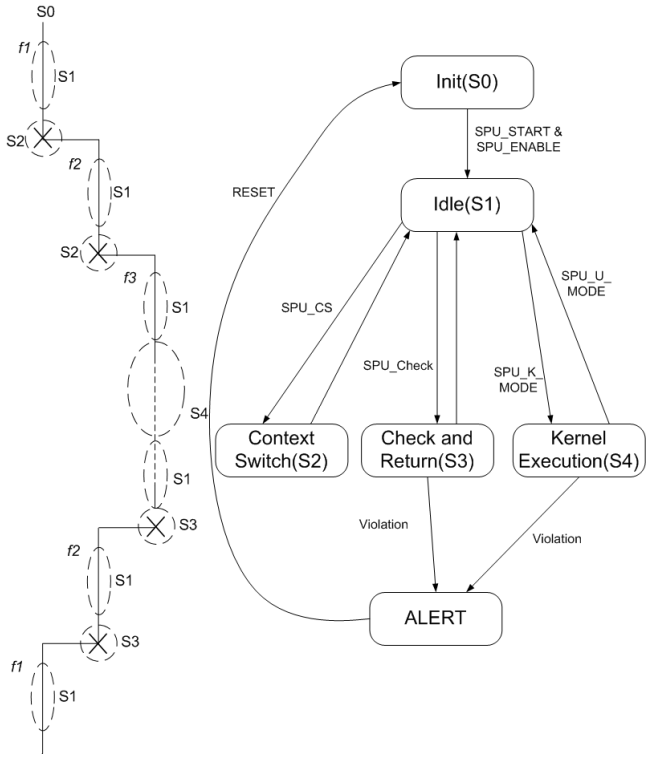

Figure 6.7. Functions' nesting and Stack Protection Unit as finite state machine that follows the scenario

The internal structure of the ASA/SPU is given in Figure 6.8. In case of a call (i.e. process context switch) the Kernel (actually the *Scheduler*) requests from SPU (actually from *User Stack History Updater*) the history of the nested path calls of the previous process and then sends to the SPU the history of the nested calls of the current process in order SPU to update the set of protected return addresses of the running process. The call is detected by the *Stack Pointer Tracker* which checks every executed instruction searching for *addik r1, r1, -x*. Then it takes the value of the SP from the `Trace_New_Reg_Value`, saves it and increments the address of the user/kernel address space in SPU where the new SP value will be stored. In case it encounters *addik r1, r1, x* it means that the return from a call is detected. It decrements the address of the user/kernel address space, so as to point to the SP of the previous function.

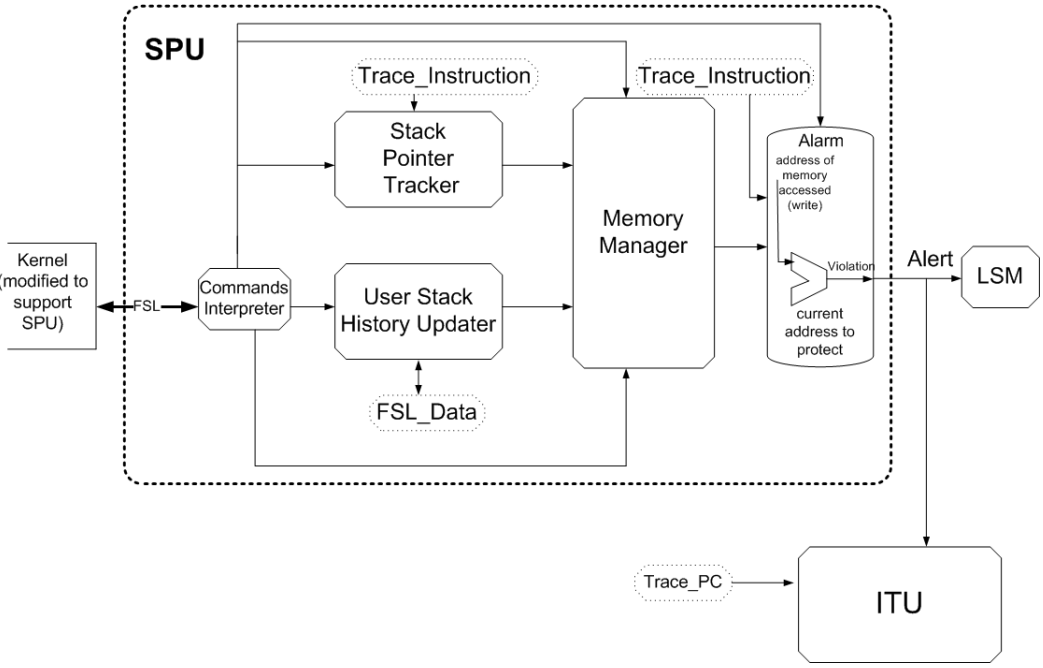

Figure 6.8. Internal structure of Attack Specific Agent - Stack Protection Unit

The operational scenario is presented in Figure 7.5. The *Command Interpreter* takes through FSL instructions from the Kernel (as it can be seen in Figure 6.8), translates them and uses to drive the SPU. In the case of context switch (SPU is in *Context Switch* state accordingly), *User Stack History Updater* points to the location to which newly arrived return address (to be stored) is going to be written and *Stack Pointer Tracker* provides the exact value to be written (i.e. return address itself obtained from *Trace bus*). The *Memory Manager*, based on command from the *Interpreter*, places the provided return address into the storage dedicated for user or kernel mode stacks. On the other hand in the case of function return (*Check and return* state) the return address stored previously by *Memory Manager* is compared with one that processor wants to return at (from `Trace_Instruction` provided by MicroBlaze Trace bus), in case these two differ, the *Alert* signal is set.

ASA/SPU interaction with the processor and Kernel

The overall NoC system with PPS deployed in Network Interfaces (NIs) attached to MicroBlazes is represented in Figure 1.2.

In the case of stand-alone applications running on the CPU (without operating system installed) the ASA/SPU shown in Figure 6.6 works very effi-

ciently(which is verified by testing results reported in Chapter 7). In this case no modifications are required on the CPU side. Deployment of an operating system (OS) on the microprocessor requires some modifications to the Kernel of the OS so as to support ASA/SPUs to be supported. Actually, the Kernel structures must be modified so as to be able to inform (issue commands through FSL connection) the ASA/SPU about changes in execution of processes running at the core. Therefore, realization of the ASA/SPU solution requires 'patching' of the Kernel.

In order to provide some basic secure environment for correct execution of the applications, the OS needs to work in at least two modes (i.e. *user* and *kernel* mode). These modes are supported by any present-day CPU and usually this means that in *user* mode all resources are accessible and all instructions are executable, while in the restricted, *kernel* mode memory regions and special register accesses are allowed only partially or not allowed at all.

The OS creates a separate *User Stack* for every newly created process (see Figure 6.9). Most of the time processes run in *user mode*. Processes can access services offered by the kernel (e.g. printing to the console) using a system call. System calls are usually accessed through wrapper functions offered by libraries (i.e. standard *libc*). The wrapper function prepares the parameters for the desired function and raises a *debug trap* via a dedicated instruction (i.e. *brki* for MicroBlaze, invokes both *syscall* and debug trap). Upon the trap instruction is set, OS works in kernel mode and doesn't consider the user stack. The kernel uses a dedicated space in kernel memory, one for every process, to store the kernel stack (see Figure 6.9), and in the case of uClinux to store the process data structure, needed to handle the process, too. Also in the case of external interrupts or exceptions OS moves into kernel mode execution. Nevertheless, since MicroBlaze does not fully support different working modes we had to introduce dedicated instructions in order to inform the ASA/SPU when a switch between these two working modes occurs. The code for handling interrupts, exceptions and system calls is architecture specific and it is located in two assembly files under the specific architecture directory (`entry.S` and `hw_exception_handler.S`). A dedicated control word that informs the SPU when the system enters or leaves kernel mode is inserted in the appropriate structure in OS. The information between MicroBlaze and SPU is communicated through FSL.

Apart from execution modes of a single process, the OS must also switch between processes. When a context switch occurs the scheduler must save the context of the current process in order to restore it next time it runs. Once the current context is saved the context of the next process scheduled to run is restored. This gives to the process the illusion that nothing has happened in the meantime. During this procedure the OS updates the information that it uses to

manage processes as well. In a similar way the OS must also perform a context switch of the SPU to store the data of the current process and to reprogram the SPU memory with the data of the next process that will run. The OS asks to SPU the history of the current user stack (in order to verify correctness of the return address) and also initiates reprogram of the SPU with the data of the newly activated process. This procedure, in charge of return address check and SPU update, is performed using only two control words through FSL, and the data is stored in the process data structure every time a context switch occurs. Once this procedure is over, a new process is ready to restore its context and get executed.

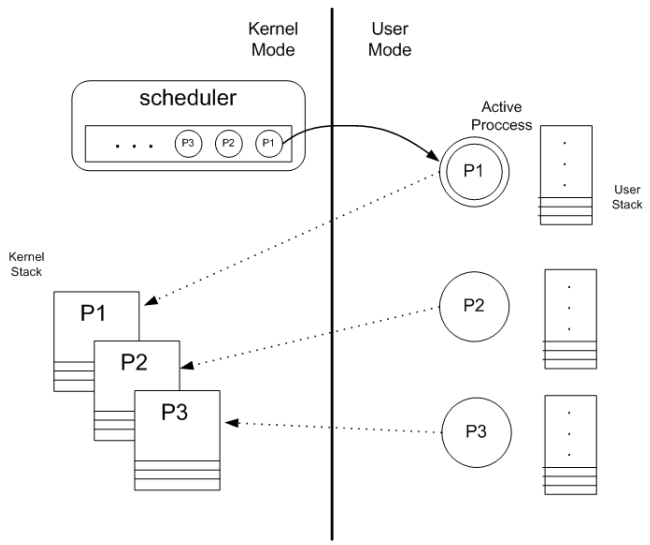

Figure 6.9. Processes and stacks considering Operating System modes

The interaction between OS and ASA/SPU is needed at process creation and whenever a process swap is performed by the scheduler. In order to temporarily save the data of the SPU traced history between process switch we added an array in the *task-struct* structure `task_struct->stack_history[]`. In the *schedule()* function we added the instructions to save the current SPU history in the array and to reprogram the SPU with the traced history of the next process. The inserted array is accessed within the *schedule()* function and rewritten with external data (i.e. this from SPU).

The modifications presented here are MicroBlaze specific and porting of the solution to other architectures would require their customization.

6.3.3 Attack Specific Agents - DoSPROT

We now introduce a protection against Denial-of-Service types of attack named DoSPROT, consisting in practice of two different types of attack specific agents. Actually, we implement attack specific protection for detecting *vulnerable* and *flooding* DoS attacks in the form of two independent Attack Specific Agents (ASAs). They work fully in parallel to each other as well as to the NI kernel so that no performance degradation is introduced into the system. We show in the sequel the role, place, functionalities and implementation details of both of them.

Attack Specific Agent - VDoS

A *Vulnerable DoS* attack is usually manifested by sending random packets to the network including among the destinations also non-existing cores. Even though these packets can not reach the destination, as they are not routed (NoC discards them), their presence is an important indicator of the security threat. VDoS therefore represents straight-forward implementation of 'out-of-boundary' check logic integrated in a LSA in form of an ASA (as shown in Section 6.4). It is matched to raise a high severity alert upon detection of addressing space violation as such an event is commonly clear proof of presence of malicious application. The implementation of VDoS/ASA is presented in Figure 6.10. The *address map* of the system contains addresses of memory mapped system cores. The *control logic* module extracts the destination address from the appropriate control flit and verifies if it is within the valid address range.

Attack Specific Agent - FDoS

FDoS/ASA is based on the module that performs the CUSUM algorithm; we have adapted, and enhanced the algorithm to our specific needs. Algorithm parameters have been customized to the given input traffic and special attention has been paid to prevention of false alerts. In particular, this is reflected in trusting calculation for the core for which FDoS alert is determined (we have assigned medium level severity to such alerts so that practically at least three alerts in a raw should be raised in order to consider the core as attacked). The block scheme of the FDoS/ASA is given in Figure 6.11. The exact implementation of the module is presented in the sequel.

Flooding DoS attack detection algorithm

We have adopted and applied the non-parametric CUSUM (Basseville and Niki-

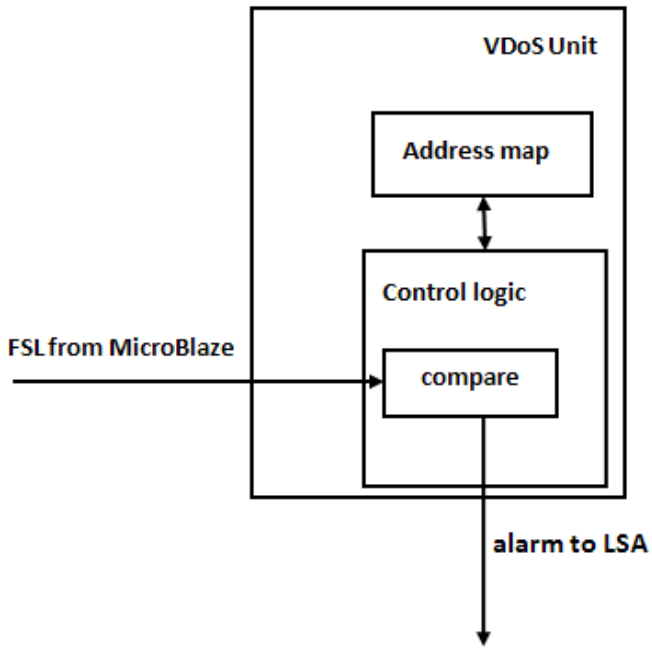

Figure 6.10. Architecture of the Vulnerable DoS - Attack Specific Agent

firov [1993]) method for Flooding DoS attack detection. Assume that variable X represents the number of packets that are sent through the network and $E(X) = c$ is the mean value of random variable X that is calculated as the average number of the flits sent in the fixed time interval t_0 through the network. We chose the step parameter a (which implies that cumulative value of the function decreases) to be fairly greater than c and define new random variable $\tilde{X}$ as: $\tilde{X} = X - a$. During the normal traffic flow the mean value of the new random variable $\tilde{X}$ will be less than zero. If an attack happens or a sudden increase of the network traffic occurs, the mean value of the $\tilde{X}$ will be positive since the mean value of the X will be greater than step parameter a. CUSUM value is calculated and updated during for each observed period t_0 as the cumulative sum of all previous mean values of the random variable $\tilde{X}$ (as shown in Equation 6.1). If $\tilde{X}$ is greater than zero in several consecutively repeated measuring periods, as a result, the cumulative value of CUSUM function will increase. Obviously this may happen due to irregularities in regular traffic or due to an DoS attack.

$$CUSUM = y_n = y_{n-1} + E(\tilde{X}_n) = \sum_{i=0}^{n} E(\tilde{X}_i) = \sum_{i=0}^{n} E(X_i - a) \qquad (6.1)$$

During normal traffic flow the CUSUM value will be on average negative with small fluctuations when the packet burst on the network occurs. But in the case of flooding attack, in every considered interval t_0 the mean value of the random variable $\tilde{X}$ will be positive which will increase CUSUM. When the value of CUSUM exceeded the threshold h (defined in design time according to expectable application traffic type) the attack is detected and the appropriate alarm is sent to the CSA (see equation 6.2).

$$alert = \begin{cases} 0 & \text{if} \quad y_n < h \\ 1 & \text{if} \quad y_n \geq h \end{cases} \tag{6.2}$$

In order to implement this algorithm, a module which captures the time series of the transmitted flits in the considered time period t_0 has been developed. This unit is also equipped with the local memory for the purpose of saving the values of algorithm parameters and various temporal values (such as mean value of the transmitted flits $E(X) = c$, upper bound a, CUSUM value, alert threshold h etc.).

The accuracy of the algorithm as well as the attack detection time, depends on the parameters a and h. These parameters require design time definition. These values should be chosen in such a way to optimize attack detection in terms of time while minimizing the number of the false alarms. However, these goals cannot be simultaneously achieved, so the appropriate parameters are chosen by analyzing specific input traffic characteristics, in order to make the best trade-off between objectives.

In order to further improve the algorithm we introduce a *saturation value* (the minimal possible CUSUM value so that it can not go more negative than that; the value is related with the *'a'* step parameter and it is determined for each application traffic input separately) and also we reset the CUSUM value upon detection of the alert. This increases efficiency in distinguishing false from real alerts. The CUSUM alert is of medium severity which in our case means decrease of core trusting value for ten. This practically means that three consecutive alerts will cause the core to be cut from the NoC. The CUSUM value is reset to saturation level upon the alert and the counter is started. If in specified time (defined as 25 CUSUM periods) upon alert issuing no new alerts arrive, LSA sends 'positive-feedback' signal to LSA.

The architecture of the solution and its integration with LSA and NI is given in Figure 6.11

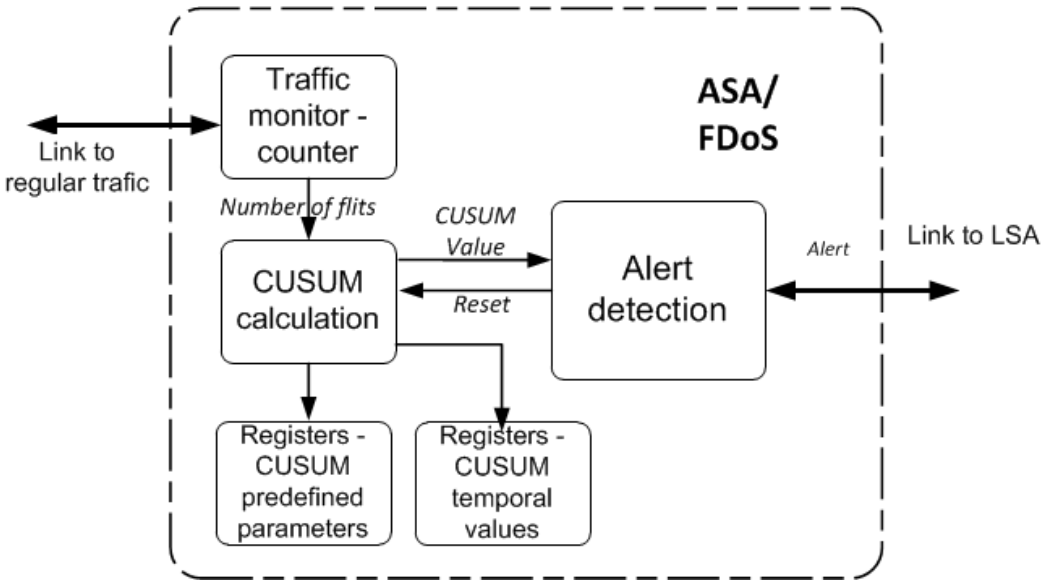

Figure 6.11. Architecture of the Flooding DoS - Attack Specific Agent

6.4 Local Security Agent

In this section we detail the implementations of the Local Security Agent (LSAs) as a module incorporating all the ASAs at an individual core and integrating them with the security framework.

The core functionalities of LSA are encapsulated in two modules:

- *The Local Security Manager* performs the key operations that are the same for all LSAs. It evaluates reported security events and assesses them forming appropriate alerts (maps alert to different severity levels). It implements positive feedback for different kinds of attacks (each type of the attack has its own ASA and in some cases associated 'supporting modules' such as positive feedback mechanism e.g. for FDoS - CUSUM). In some cases, the LSA may contains a module dedicated to attack analysis purposes (e.g. for SPU/ASA an Instruction Tracing Unit (ITU) that serves for debugging purposes is designed). Finally the LSA maintains the *ASA Portfolio* which keeps track of all ASAs integrated in the LSA with all associated 'supporting modules'. The detailed description of LSM structure and functionalities is given in Section 6.4.1.

- *Communication interface* adopts communication standards of SNoC to LSA. It implements the protocol presented in Figure 5.2 from LSA side. The details on this module are given in Section 6.4.2.

It should be noted here that even though in general the structure of all LSAs is the same, practically two different kinds of LSA implementations can be identified in our system - LSA for processing core and shared memory block. Nevertheless, the only difference between them lies in types of ASAs they employ and accordingly assigned supporting modules of LSM (refer to Section 6.4.3 for details). While LSAs for processing cores employ SPU/ASA, VDoS/ASA and FDoS/ASA; an LSA assigned to shared memory accommodates DPU/ASA. We explain later how these differences reflect to the practical LSA implementation (e.g. positive feedback mechanisms).

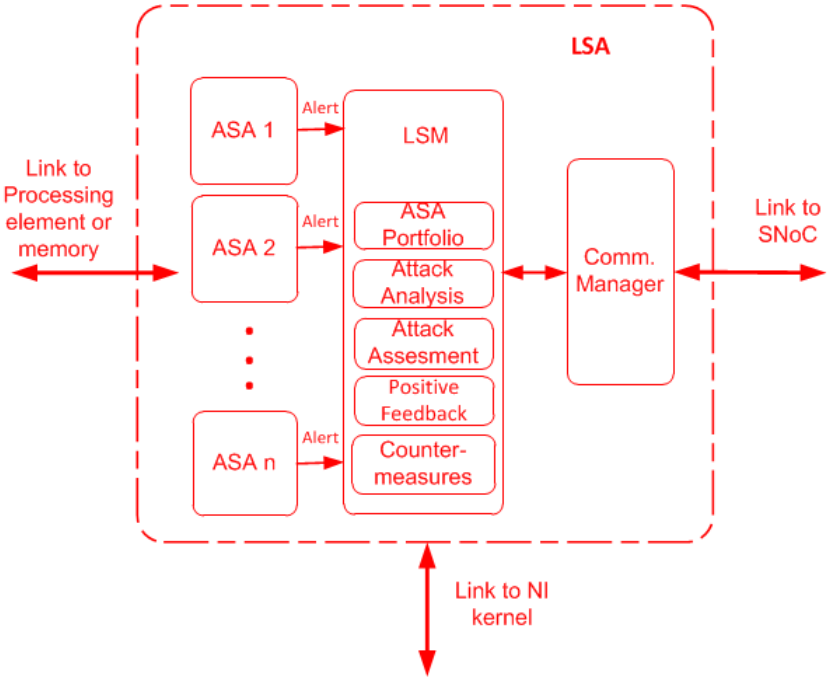

Figure 6.12. General Local Security Agent architecture (details explained in Figure 6.4.1)

A brief description of LSA modules as well as specific LSA solutions developed for our security framework are given in the sequel. The general architecture of LSA is presented in Figure 6.12 and explained in details in Section 6.4.1.

6.4.1 Local Security Manager

The Local Security Manager (LSM) represents a core component of LSA. It manages all the security related issues relevant for the core it is attached to. These consider following functionalities:

- Evaluation of detected violations reported by ASAs. This concerns processing of the of the incident and calculating of the alert severity accordingly. For instance, as unauthorized memory access may originate from different reasons, an alert is raised only if a defined number of such violations is received in given time (both, the number of detected violations and time window are statically decided in design time). These operations are performed by the *violation assessment* module and are optional, depending on tha type of the attack.

- Positive feed-backing is mechanism that is specific for each attack type and it is aimed at prevention of false alerts. It relies on expectancy that detected violation will be repeated in case of the malicious code presence in the system. It is realized in form of an alert monitor which operates in a given time-window. The counter (that represents this monitored period) is triggered by violation detection. This concept is optional and applicable only for attacks that may have some uncertainties in the detection process (such as memory access validation or DoS detection) so that additional checks are needed. Detection of some of attacks such as 'code injection' is highly reliable (deterministic) and no other proofs are needed. These attacks are given high severity level.

- Collection of information (if applicable) on the alert that might be useful for precise threat identification, countermeasures or debugging process is also optional. This information may be available for some types of attacks such as for instance 'code injection' (appropriate ASA may get some debug information provided by the processor it is attached to, as the case with Instruction Trace Unit explained later). All operations associated with evaluation of the attacks are encapsulated in a dedicated *attack analysis module*.

- ASA portfolio represent the list of all employed ASAs and provides information on all the supporting elements assigned to specific ASA. Not all the ASAs have the same supporting elements, some of them (such as DoS related ASAs) have assigned positive feedback modules) while others (such as 'code injection' SPU/ASA) may have associated violation analysis modules or for instance violation assessment modules (which calculate alert severities as for instance memory access violation related DPU/ASA)

- Attack countermeasures - performed by *countermeasures module* which directly manages reactions of the framework to the discovered threats. These

are mostly oriented towards problem isolation, for instance, the access to the attached processing unit to the regular data NoC can be disabled on request from CSA. The access to the regular NoC to the blocked processing unit can be reestablished only if CSA informs LSA (actually *Countermeasure* module) that the attached unit can be unblocked.

Attack analysis module - Instruction Trace Unit

The attack analysis module is in charge of providing as much as possible information on detected attack. The information can be obtained from different sources in different ways and it is practically attack and technology specific. Nevertheless, the 'report form' is the same for all the attack types. In our work we rely on debug instruments enabled by MicroBlaze processor.

MicroBlaze provides the so called *trace bus* which enables an external view to the instructions that are being executed. Based on this property we built an Instruction Trace Unit (ITU) that can be used for debugging purposes. ITU is implemented in form of inferring shift registers which keep tract of executed instructions. The number of these registers determines the number of instructions that are able to be stored, the only limitation in this case is the available area.

ITU works in two modes:

- Tracing instructions - in this mode it only memorizes the value of PC or instruction code of each newly executed instruction by writing it into the first shift register
- Alert (hold) mode - if the SPU has provided an alert signal the ITU freezes tracing and provides the kept track of instructions memorized in its registers to Local Security Manager, through dedicated custom bus

The structure of ITU is represented in Figure 6.13. The obtained information which represent a set of latest instructions can be locally processed or sent for further processing to CSA. In our implementation this information is transmitted to CSA where it is stored and could be used for further attack elaboration and decision on countermeasures.

6.4.2 Communication interface

The communication interface adapts LSA communication to SNoC standards. It actually interprets and packs properly information from Local Security Manager modules (such as *alert assessment*, *alert analysis* etc.) and forwards it to

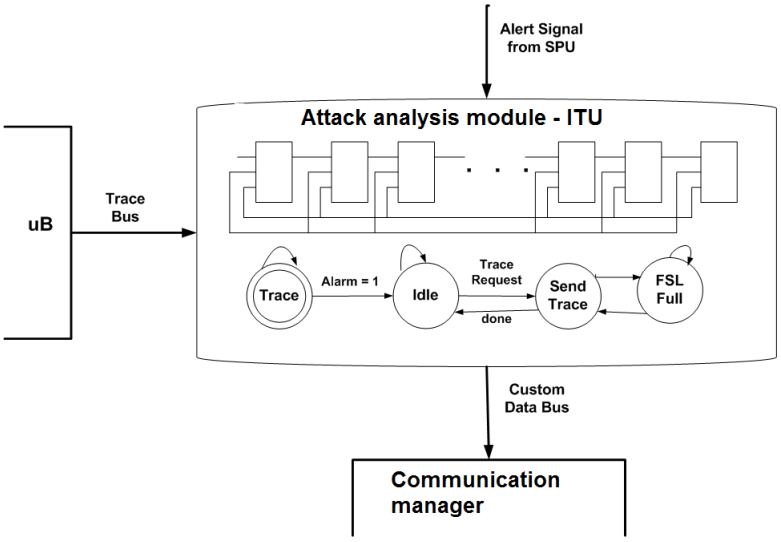

Figure 6.13. Instruction Tracing Unit structure

the CSA according to specified protocol. Conversely, it translates messages from CSA and drives LSA modules accordingly (for instance disables access to NoC in case of request from CSA).

6.4.3 Specific Local Security Agent solutions

In our implementation we have employed two different types of cores (processing elements and memories) which practically resulted in two different implementations of LSAs that have been realized. In fact, the differences came from different types of attacks that have been addressed by ASAa integrated in these LSAs. Actually, the structure of both LSAs fully complies with general LSA structure exposed in Section 6.4 but practically the two differ in optional modules inside Local Security Manager that are implemented. More precisely:

- LSA associated to memory core - involves only DPU/ASA which requires *attack assessment* and *positive feedback* module while *attack analysis* module has been realized as a trivial structure (as no additional information on the attack, apart of what is the access initiator core, can not be obtained). Realization of the LSA hosting DPU/ASA is shown in Figure 6.16.

- LSA associated to processing core - involves SPU/ASA (shown in Figure 6.14), VDoS/ASA and FDoS/ASA (both shown in Figure 6.15). While the first one requires *attack analysis*, VDoS/ASA practicaly does not require

any of these optional modules; FDoS requires only two other modules, namely - *attack assessment* and *positive feedback*.

The architecture of the LSAs, together with inter- and intra- connections among components is presented in Figure 6.14 (for processing core involving ASA/SPU) and for LSA involving DoS protection in Figure 6.15. In Figure 6.16 we present integration of ASA/DPU with LSA into corresponding NI (for memory blocks).

6.5 Central Security Agent

The CSA represents the central point in managing security policies of the framework. It is in charge of correlating all security related information from different sources, establishing a system level protection mechanism. It practically monitors and controls LSAs, maintains trusting table, performs attack countermeasures and so forth. It is designed in modular and scalable manner. CSA is composed of three main components - Communication interface, Security Policy Manager and Trusting Table. The structure of CSA is represented in Figure 6.17

The Communication interface translates SNoC protocol to drive internal operations and vice versa. It actually performs operations that are mirrored to those done by LSA communication interface. Trusting table actually is a block of RAM memory that keeps track of trusting values of each core (i.e. LSA) in the system. The heart of the CSA is indeed Security Policy Manager that is composed of following modules:

- LSA portfolio - keeps track of all cores (actually LSAs) in the system. If an additional core is to be inserted it should be only checked out in the module
- Trusting Policy Manager - executes the valid trusting policy (in our case one defined in Section 5.4.2), counting alerts and positive feedbacks for each core
- Security domains - defines the number and 'borders' (in sense of trusting values) between different levels of security. It also defines the way how cores in specific domain are treated.
- Attack analysis - collects information obtained from counterparts in LSAs, creates logs, history and statistics of attacks per core

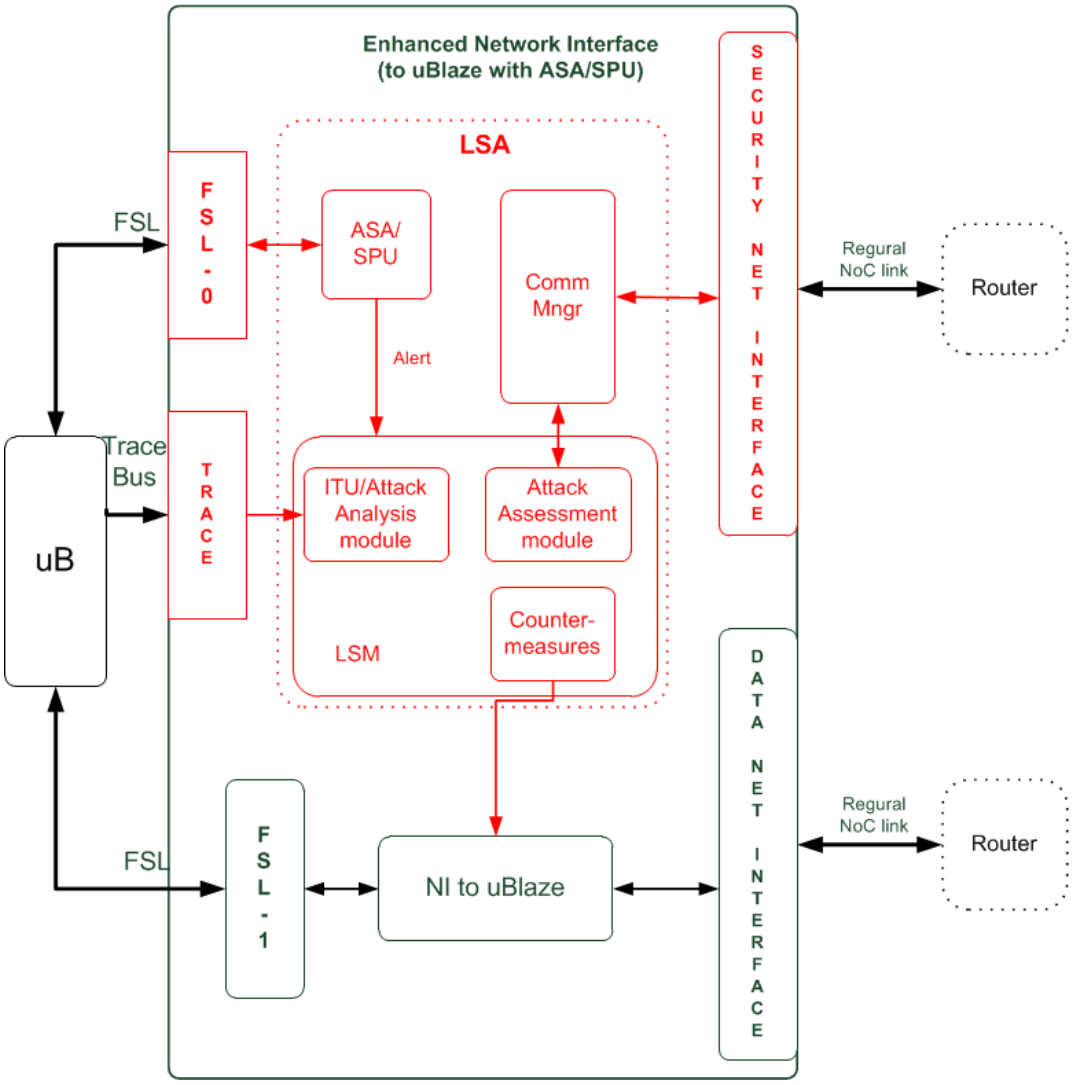

Figure 6.14. Enhanced NI to microBlaze with Local Security Agent involving Attack Specific Agent - Stack Protection Unit (security related elements are denoted in red color)

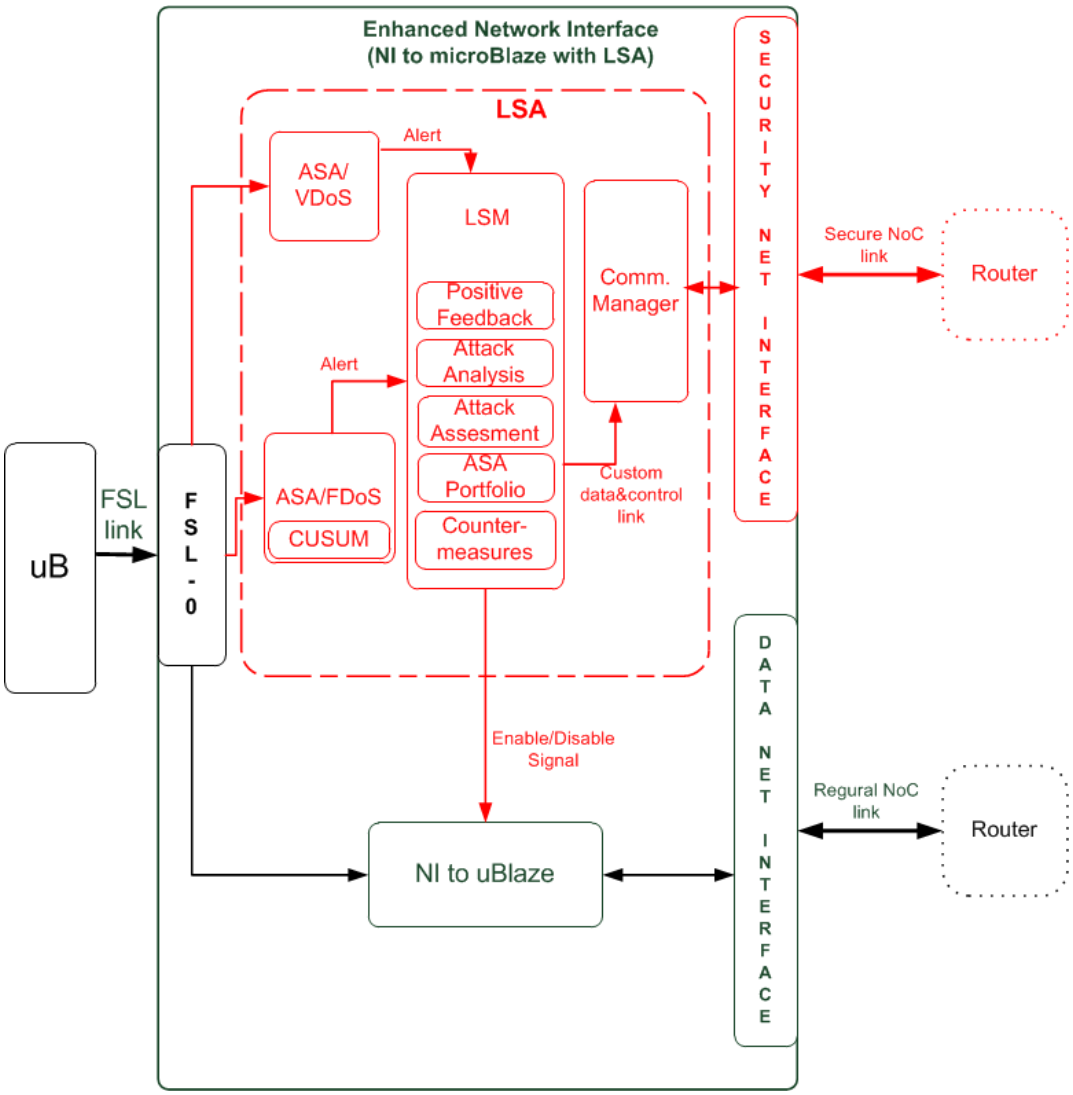

Figure 6.15. Enhanced NI to uBlaze with Local Security Agent involving ASA/VDoS and ASA/FDoS (security related elements are denoted in red color)

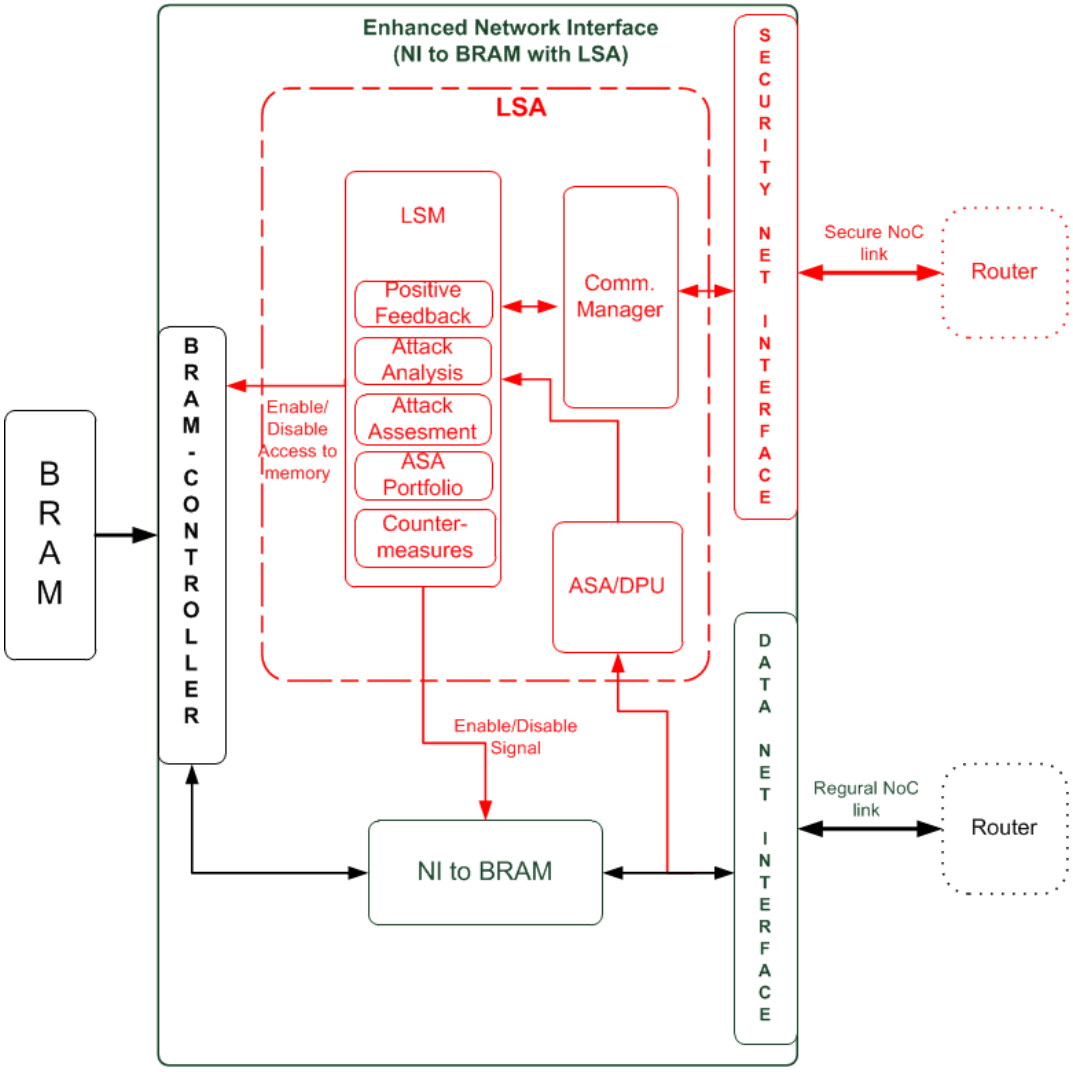

Figure 6.16. NI to memory with Local Security Agent for involving Attack Specific Agent - Data Protection Unit (security related elements are denoted in red color)

- Countermeasures - performs specified actions against detected attacks

A CSA is a fairly 'light' component of the system as most core functionalities are distributed and left to LSAs (e.g. positive feed-backing) which unloads communication on SNoC and also saves energy at the same time. Nevertheless, some operations are still centralized as CSA is the only pure hardware part of the system (LSA kernel is also built in hardware but contains some ASAs that have software components as well) and accordingly is considered as the safest part of entire security framework. We purposely build the unit in hardware as this allows claiming the highest level of robustness and resistance to attacks.

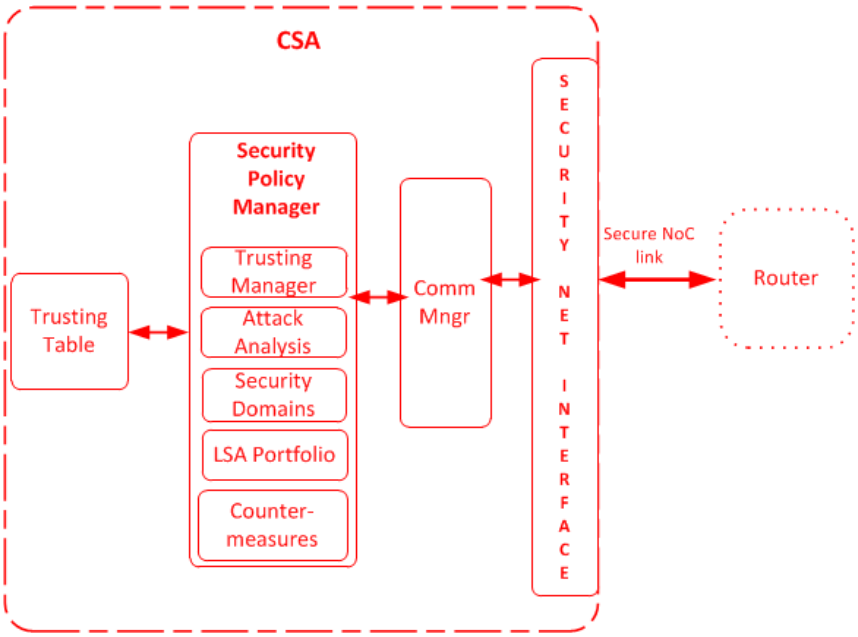

Figure 6.17. Structure of Central Security Agent

6.6 Secure NoC

The Secure NoC (SNoC) represents an interconnecting medium among security related components. In other words, it is a fully NoC compliant structure both in terms of architecture and communication protocols. Nevertheless, SNoC is built fully in parallel to existing NoC that routs regular data. Both networks practically use the same NIs for adopting data transmission to given communication protocols - NI kernel considers regular data while communication interface of LSA handles security related data. From this point onwards the two networks

are fully separated. This way, we encapsulate the security related communication in a dedicated structure and achieve full separation of regular and security related system components and their communication.

The structure and position of S-NoC inside the security framework is shown in Figure 1.2. The communication among agents attached to S-NoC is performed according to the protocol presented in Figure 5.2.

Chapter 7

Testing and Validation

The final system has been implemented on a Xilinx Virtex-5 ML510 board using EDK 10.1 tool-chain while several components have been previously implemented and tested on a Virtex-II Pro board. The proposed solutions have been tested against various attack scenarios. The area and power consumption as well as performance impact are studied and presented in this Chapter.

7.1 Validation approach

We have implemented, tested and validated the proposed solutions in a two phase process:

- Firstly, attack specific solutions have been developed, tested and validated. These include Data Protection Unit, Stack Protection Unit and Denial-of-Service (both vulnerable and flooding DoS) protection modules. It should be mentioned that both implementation and validation of these framework elements represents technically the most demanding part of the work.

- In the second validation phase the architecture of the 'core' security framework which integrates diverse attack specific solutions has been developed and verified. This includes Local Security Agents, Central Security Agent and Secure NoC. Also efficiency of security policies implemented in the security framework has been tested and validated.

We show in details approaches and procedures adopted for the purpose of validation of the proposed solutions. Validation of attack specific solutions includes four different units:

- Attack Specific Agent - Data Protection Unit - presented in Section 7.2
- Attack Specific Agent - Stack Protection Unit - presented in Section 7.3
- Attack Specific Agent - Vulnerable DoS - presented in Section 7.4
- Attack Specific Agent - Flooding DoS - presented in Section 7.5

The details on concrete implementation and validation of 'core' of the security framework and implemented correlating security policies are described in Section 7.6

7.2 Attack Specific Agent - Data Protection Unit - implementation and validation

The conceptual solution as well as the implementation details of Data Protection Unit (DPU) have been presented in Section 6.3.1. Additional enhancements of the Network Interface connected to the shared memory (i.e. Block RAM - BRAM) are needed in order to adapt the solution to the architecture of the system.

7.2.1 Integration with NI to BRAM

As mentioned above, the ASA/DPU is embedded in the NI that interfaces to the interconnection network with the BRAM used as shared memory in our system.

As shown in Figure 7.1, the NI embeds the controller of the BRAM, which is in charge of handling write/read accesses to memory. The BRAM controller works in parallel with the DPU controller. Both controllers are implemented as state machines that take the same input and process it in parallel. The UML activity diagram for an access in memory is shown in Figure 7.2. Upon the arrival of a new first flit containing the header of the packet, access information is passed by the NI to both controllers. While the DPU controller checks the access rights of the request, the BRAM controller sets the memory block in the read/write access operation mode. In case of a request satisfying the access rules specified in the DPU, data are read or written to memory and an appropriate acknowledgment packet is sent to the initiator of the transaction. If the request is not accepted, the packet is discarded and a packet of negative acknowledgment is sent to the initiator.

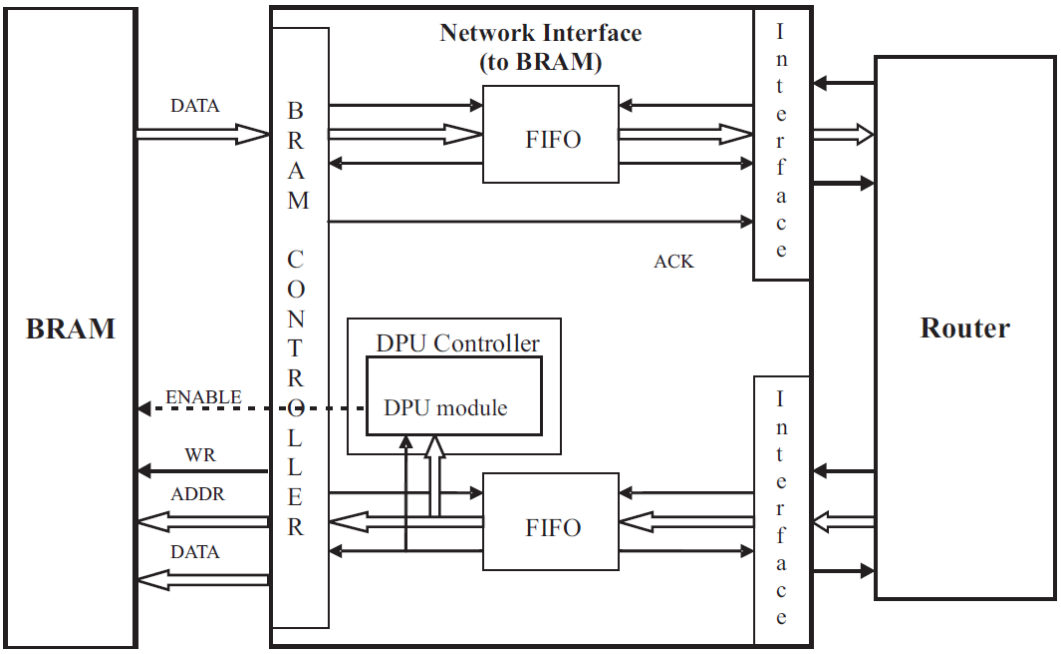

Figure 7.1. Internal implementation of the Network Interface to BRAM embedding the ASA/DPU

7.2.2 Reconfiguration of the DPU table

In order to make the proposed solution for data protection more flexible and more efficient, we designed the DPU module to be reconfigurable. In fact, in a general software environment the required memory access rights can change dynamically with the evolution of the applications (and/or the system). Moreover, access rights assigned to cores in the system may change due to decreased trusting so that reconfiguration of shared memory access rights as a countermeasure is required. This requires an update of the data protection characteristics in order to satisfy the security requirements of the changing applications.

To enable this feature, we added a write port and another memory unit (called *shadow memory*) to the basic DPU architecture. Actually, this module has been integrated in the LSA that embeds ASA/DPU in the form of a countermeasure module (as represented in Figure 6.16). The additional write port is used to update the DPU table, while the memory module is used to store the new DPU values.Actually, the *shadow memory* stores the necessary information to reconfigure the DPU that allows to satisfy the security requirements of the following application scenario. When it is necessary to let the system switch to a new scenario, the *shadow memory* is updated with the new information. Only

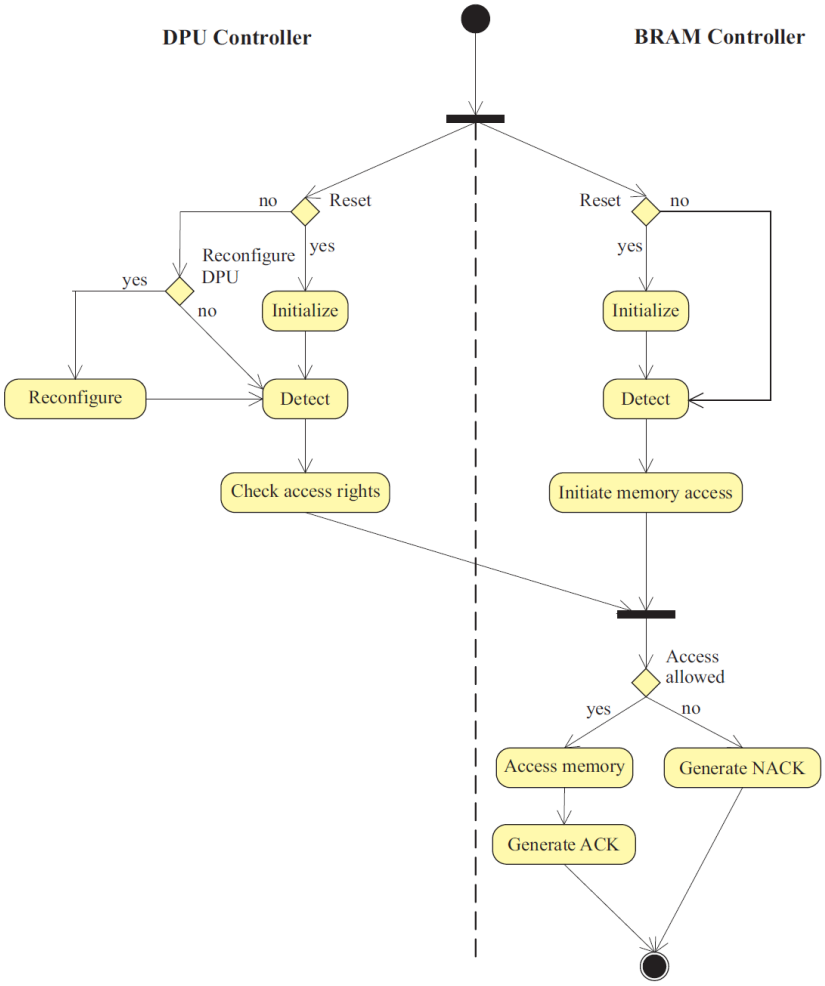

Figure 7.2. Activity diagram of the NI embedding the DPU: access to memory and reconfiguration

Table 7.1. FPGA resources consumption - System composed of two Microblazes, shared BRAM and NoC (implemented on a Xilinx Virtex II board)

System with DPU (4 lines)	**Used/Available**	**% Used**
Slice Flip Flops	2,465/27,392	9
4 input LUTs	5,671/27,392	20
Occupied Slices	3,371/13,696	24
Block RAMs	104/136	76
MULT18X18s	6/136	4
System with DPU (8 lines)	**Used/Available**	**% Used**
Slice Flip Flops	2,610/27,392	10
4 input LUTs	5,891/27,392	21
Occupied Slices	3,527/13,696	25
Block RAMs	104/136	76
MULT18X18s	6/136	4

when the *shadow memory* has received all the new table values from the CSA that initiated the table update (the controller of the overall security system) and the reconfiguration signal has been issued, the values on the shadow memory can be committed to the DPU table.

This method avoids a transient behaviour of the DPU during the updating since committing from the shadow memory is faster than a remote update. Packets that arrive from the NoC during the reconfiguration wait in the input queue of the NI until the end of the process before being analyzed.

Another important issue is that the reconfiguration phase for the DPU can be performed only by the CSA, since otherwise the reconfiguration of the *shadow memory* can be used as base for attacks. CSA communicates with the appropriate LSA which further performs ASA/DPU reconfiguration through LSM and 'countermeasures module'.

ASA/DPU has been proved to be an efficient firewall-like protection mechanism which filters unauthorized memory access attempts according to a predefined set of rules. Its efficiency depends on the rest of the security system (access rights are calculated according to the trusting which is calculated by CSA) so that we can consider it as an useful complementary part of overall security solution.

Table 7.2. Area consumption (occupied slices) as relative ratio between components

DPU (8 lines)/Component	Consumption ratio
DPU/Router	1.06
DPU/Microblaze	0.25
DPU/NI (with DPU)	0.83

7.2.3 Costs of implementation

In this section we present synthesis results of the multiprocessor system previously similar to the one shown in Figure 1.2. The test architecture is composed of two microprocessors (MicroBlazes), two shared on-chip memory block (BRAM) of 64 KB, and the interconnection NoC system, in which we implement our module for data protection. The whole architecture was developed on a Xilinx Virtex-II Pro XC2VP30-FF896 board, by using the Xilinx Embedded Development Kit version 8.2. The system was implemented to work at the operation frequency of 100 MHz. For test and debug purposes, an RS232 interface was used for communication between the board and the host computer.

Table 7.1 shows FPGA resources utilization of the overall system, in the case in which a DPU with respectively 4 and 8 lines is implemented. Numbers reported in the tables refer to an implementation of the elements of the system as shown in Figure 1.2 In the first case, the total number of equivalent gates is equal to 6,757,418, while in the second case it is 7,057,675. A DPU with 4 entry lines is able to protect a memory divided into 2 protection regions from all the possible types of access request (*load*/*store*) issued by the two processing elements in the system. In fact, the number of entry lines is given by the product of the number of PEs and the number of protection regions in which the memory is divided. A DPU with 8 entry lines can protect the same system with a memory with up to 4 protection regions.

Table 7.2 shows the ratio between the dimension of a DPU with 8 entry lines and those of the other components of the system. As it is possible to notice, the dimension of the DPU is almost equivalent to the dimension of the NoC router and it represents a significant part of the NI to BRAM.

In Figure 7.3 we show, expresses as number of occupied slices, the area of different configurations of the DPU. As also shown in (Fiorin, Palermo, Lukovic and Silvano [2007]), the number of slices occupied by the implementation of the data protection module increases linearly with the number of entry lines, and it represents the dominant part in the area overhead of the NI to BRAM.

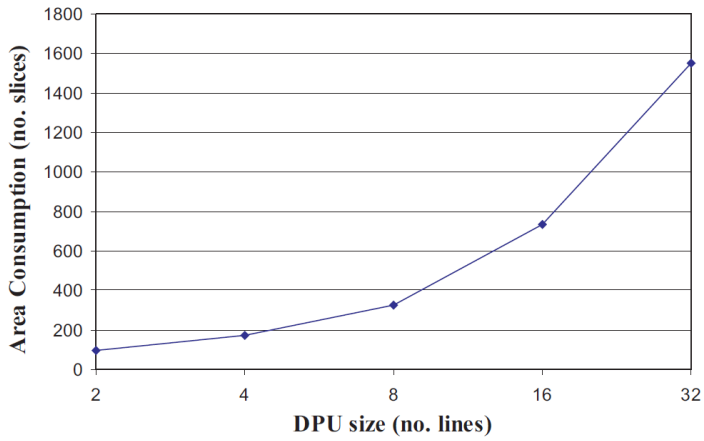

Figure 7.3. Area of the DPU for different numbers of entry lines

7.3 Attack Specific Agent - Stack Protection Unit

We present here the implementation of the Stack Protection Unit explained in the Section 6.3.2. In the prototype presented we have used Xilinx provided MicroBlaze 'soft core' processor. MicroBlaze implements stack conventions as represented in Figure 7.4.

Whenever a function calls another procedure (a branch in the instruction flow occurs) the PC of the caller is saved in the link register (R15). The stack pointer must be decreased by as many positions as the width of the stack frame is wide, saving the context of the caller function and opening a new stack frame for the called function.

At the return from the callee function, the return address is retrieved from the stack and saved in the Link Register; then the return instruction updates the PC with this value. The stack is updated (increased) with the instruction following the return instruction.

This analysis can reveal a potentially critical problem: by writing an array to the stack without checking its upper bound (e.g. using the C function *strcopy()*) one can overwrite the data of the other stack frames. Very simple attacks can be performed on architectures in which the instruction and data memories are shared, consisting in injecting machine code directly on the stack and modifying the return address to point at malicious code. Even if the stack memory is not executable, and/or separated, overwriting the return address of the caller and also overwriting saved registers is still possible. This means that the attacker can redirect the control flow by giving the corrupted values to the targeted return function.

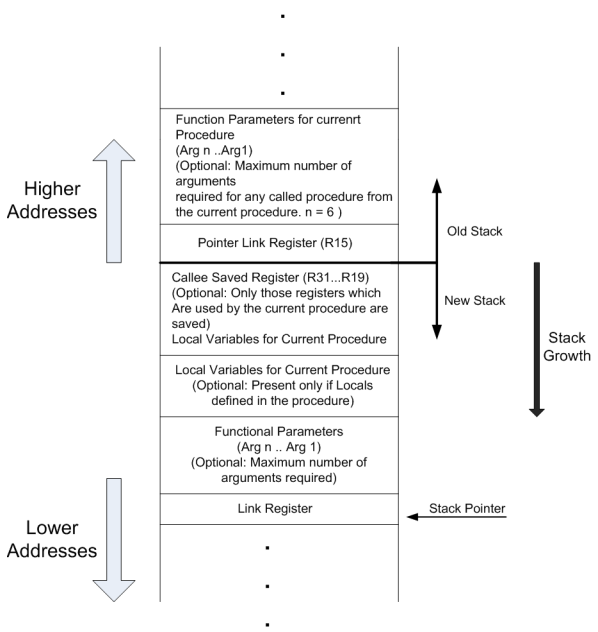

Figure 7.4. MicroBlaze stack organization

The basic concept of the SPU is rather simple. Every time a function is called, the return address of the caller function is stored on the stack and also replicated in the SPU, which keeps track of the return addresses of the nested functions and reports any attempt at overwriting the particular address (simply by comparing the stored value with one obtained from the stack). Figure 6.6 shows the concept of the SPU. Practically, the task of the SPU is to restrict the stack accesses to the current stack frame.

Considering the context switch scenario as presented in Figure 7.5 and the simplified attack scenario:

```
void return_input (void){
        char buffer[30];
        gets (buffer);
        printf("%s\n", buffer);
}
main() {
        return_input();
        return 0;
}
```

In the *main* the *return_input* function is called. Calling the function saves the

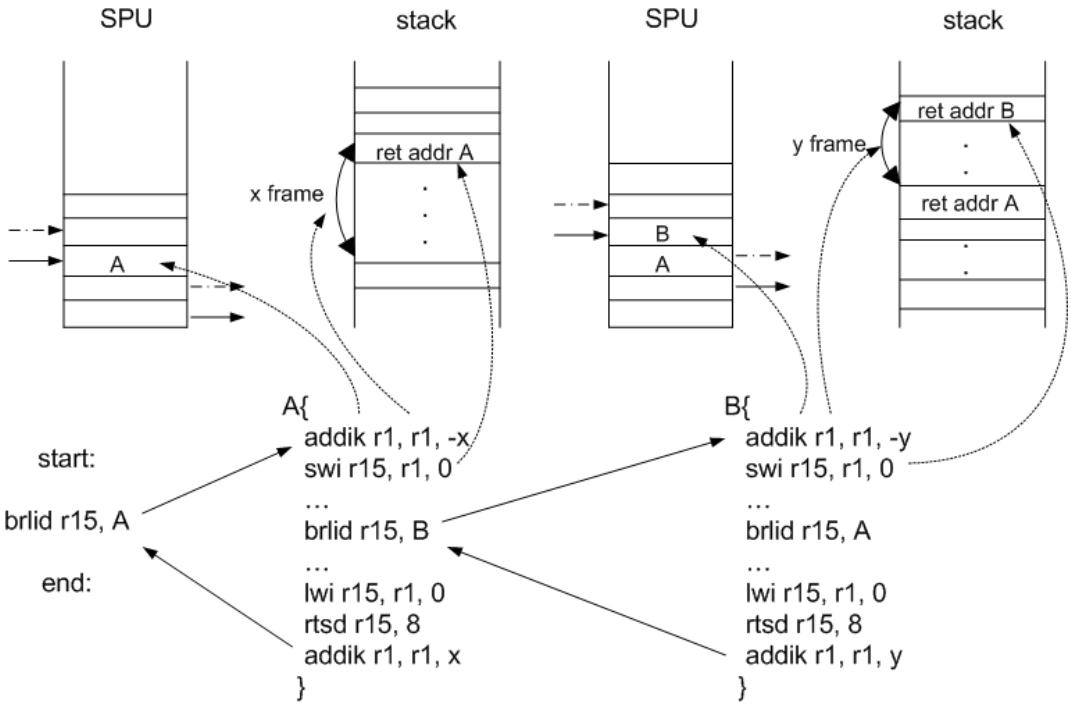

Figure 7.5. MicroBlaze and SPU context switch scenario

return address onto the stack and the space for the array is reserved, however there is no check if the input can fit in the array size.

ASA/SPU appeared to be an efficient protecting instrument in all considered scenarios. It other words, the SPU raises the *alert* signal whenever the attack is performed, ITU (which is actually Attack Analysis Module of its LSA, see Figure 6.14) moves into *hold* mode and Local Security Manager of the LSA takes and transmits those data to the CSA disabling at the same time to the attacked processor access to the NoC.

7.3.1 Costs of implementations

The entire system prototype (based on a scheme shown in Figure 1.2) is composed of two shared memory blocks two MicroBlazes (running uClinux) with LSA and NoC (including NI to memory with embedded DPU), occupies 17,338 slice LUTs which is around 22% of total available area on the board (total available on ML510 board: Slice Registers - 81920; Slice LUTs - 81920).

The detailed cost of the overall implemented system of its (separately evaluated) components in terms of area consumed is given in Table 7.3. It can be

Table 7.3. Area consumption (in number of occupied slices) per component - implemented on Xilinx Virtex-II board

Component	**Area (slice Regs)**	**Area (slice LUT)**
Total System	16504	17338
ASA/SPU	186	360
ASA/DPU	334	314
LSA/uB	213	481
LSA/DPU	361	435
CSA	45	64
SNoC	558	1284
Tot Security Framework	1751	3180

seen that the proposed security framework takes about 10% of Slice Registers and about 18% of Slice LUTs consumed by the entire system design. These results are slightly better compared to those provided in (Patel et al. [2010]) which is also dealing with system level protection against buffer overflow.

It turns out that the LSA for MicroBlaze (uB) represents about 77% of the enhanced NI with respect to area consumed. Such result had to be expected as the LSA components consume RAM for storing data while the part of NI that is interfaced to MicroBlaze mostly consists only of control logic. Nevertheless, the ASA/SPU represents the most consuming component within LSA.

Analysis of the synthesis results shows that area overhead is inevitable in solutions like this as deployment of security concepts necessitates commitment of additional system resources. On the other hand, the costs introduced are still acceptable and they even represent an improvement compared to previously proposed solutions.

7.4 Attack Specific Agent - Vulnerable DoS - validation

The ASA/VDoS module is implemented according to the detailed description given in Section 6.3.3. It actually verifies correctness of the destination addresses of packets traveling on the NoC and raises alerts in case of any detected violation. Due to simplicity of its functionalities its realization has been rather simple and not demanding both in terms of technical realization as well as with respect to resources consumption. Accordingly, the validation process for the module has been straight-forward. The module is tested against different sets of

packets transmissions and it is proved to be efficient in detecting address space violations, reporting alerts in all attack situations.

The module operates fully in parallel to the rest of the system so that no performance degradation has been introduce by ASA/VDoS.

Due to its extreme simplicity (address comparator and a look-up table) consumption of area and power compared to the rest of the design is negligible (far less then 1%).

7.5 Attack Specific Agent - Flooding DoS - validation

Design and validation of ASA/FDoS has represented one of technically most requiring tasks for the present work. Moreover, due the nondeterministic nature of this type of attack, protection against the attack itself represents a very relevant test for our security framework.

The algorithm adapted for the purposes of FDoS detection as well as the appropriate ASA/FDoS module are described in Section 6.3.3. Due to complexity of the testing of the proposed solution, validation has been carried out in two phases. We have built a simulation environment in Matlab which fully corresponds to the experimental setup given in Figure 7.6. Processing cores are simulated as sources of traffic according to the distribution presented in Section 7.5.1. In the first phase, we have tested only efficiency of the enhanced CUSUM algorithm considering values of the parameter a for the given traffic inputs. In a subsequent stage we tested efficiency of the security policy that combines all the protection mechanisms in the system according to the defined trusting policy (refer to Equation 5.1). For these purposes we used the simulation environment provided by EDK Xilinx which fully emulates the developed hardware design enabling much better insight in results.

In practice we performed:

- Simulation of the algorithm for the given system considering the predefined traffic patterns. This step aims firstly at validating the algorithm itself and afterwards at tuning its parameters for the specific input traffic pattern. Simulations are performed in Matlab simulation environment.

- FPGA prototyping of the solution and its integration in the existing experimental NoC based MPSoC as the final validation step

We present our approach to testing and validation in details in the sequel.

7.5.1 Simulation of FDoS detection by CUSUM algorithms and parameters tuning

In order to credibly validate the proposed solution we have to determine the relevant application communication patterns and set algorithm parameters accordingly.

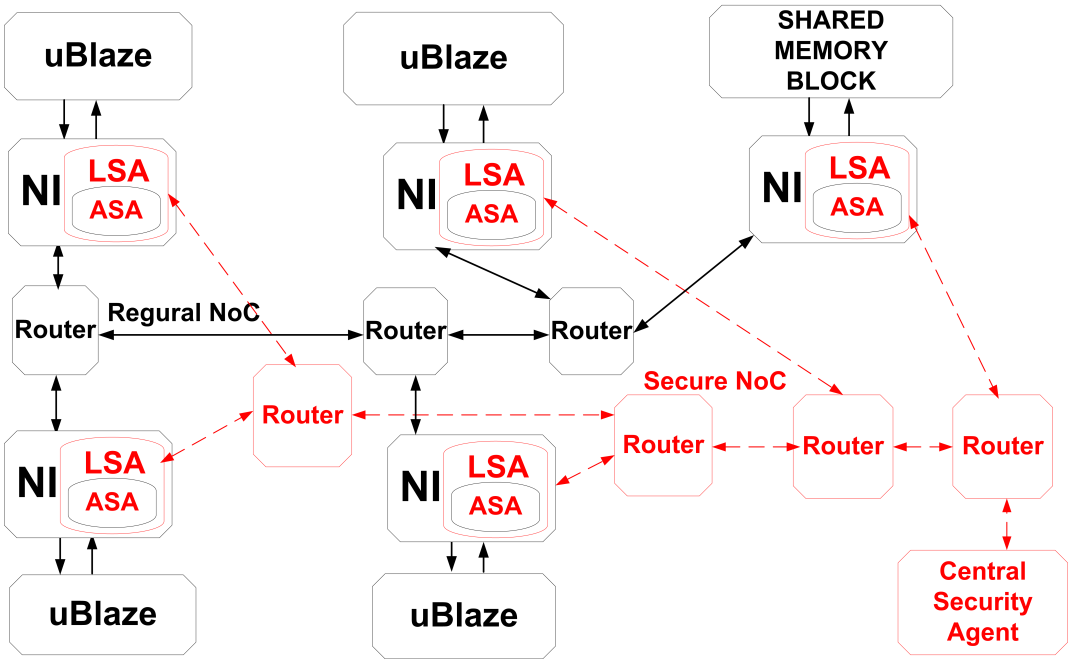

Figure 7.6. Experimental MPSoC setup used for final system validation

The input traffic patterns have been used as defined in the sequel.

Packet Distribution of the test Applications

Since the protection from DoS attacks requires the observation and analysis of the network traffic, in order to properly simulate the attack, the normal data packet flow for different type of on-chip applications has to be taken into account. For the sake of testing the CUSUM algorithm we model the traffic on the network. There are several possible ways to characterize the traffic behavior on the NoC-based MPSoC among which the following are the most important (according to Santi et al. [2005]):

- *uniform distribution of the packets or constant injection rate* - assumes that the source constantly sends data to the destination at the same rate

- *packets' distribution using the probabilistic functions* - describes the traffic by the probabilistic functions
- *trace methods* - monitoring traffic to detect behavioral patterns

These approaches may be tailored to different applications. Using the constant injection rate in modeling the NoC traffic can be suitable for streaming multi-media applications (as stated by Tedesco et al. [2006]). Probabilistically modeled applications can be classified in two groups: application for which the average traffic rate is not known a priori - Unknown Rate (UR) applications - and applications for which rates are previously defined, named Known Rate (KR) models. The algorithm has been tested on these two traffic simulation types, the detailed description and explanation of these methods will be presented.

Constant Injection Rate: According to this model packets are injected at the fixed rate, in the defined time periods (Tedesco et al. [2006]). This model does not reflect very well real word situations, since usually the applications' data transmission rates vary over time. Still, the model is very efficient for media and streaming applications.

Probabilistic Models : There are two classes of such models, namely - Unknown Rate (UR) models, and Known Rate (KR) models. The first class includes On-Off processes, while normal and exponential distributions are instances of the second class.

Unknown Rate (UR) Model: The applications that are fairly well described by this model should inject fixed-length packets, but with variable times of activity and inactivity. These kinds of applications are usually called On-Off applications and the size of the packet burst as well as the duration of the idle period vary. During the active period the traffic source produces the fixed-length packets at regular intervals while during inactive period there is no packet generation. Figure 7.7 presents the On-Off traffic rate model (Tedesco et al. [2006]).

This model can be described with different probabilistic distributions among which the Pareto distribution is the most widely used. The model that uses Pareto distribution suitable for describing the applications like MPEG-2 video and Internet traffic (according to Pande et al. [2005]). Another unknown rate model is based on Markov Chains and is known as Markov Modulate Process (MMR). The current state of a traffic source in a Markov Chain specifies data generation at a rate r. The function that describes state changes is an exponential (Tedesco et al. [2006]). This model is usually adopted for modeling Internet traffic (Clegg [2007]).

The Pareto On-Off distribution, which is of particular interest for our work, is described as:

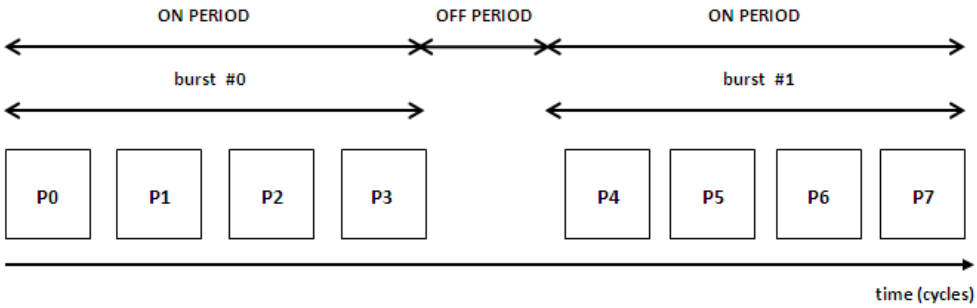

Figure 7.7. On-Off Traffic Model of the test application

$$t_{ON} = (1-r)^{\frac{-1}{\alpha_{ON}}} \tag{7.1}$$

$$t_{OFF} = (1-r)^{\frac{-1}{\alpha_{OFF}}} \tag{7.2}$$

In this equation r is the randomly chosen value in the [0:1] range allowing to dynamically generate the size of the 'On' and 'Off' periods. α is a parameter that is specific for every application.

Known Rate (KR) models are based on discretized versions of continuous probabilistic functions, such as the normal and exponential distributions. Given probabilistic distributions, and given the amount of packets, together with a set of transmission rates, the process mounts a packet transmission table indicating the number of packets for each defined rate. Equations 7.3 and 7.4 present the normal and exponential distribution, respectively, and they compute the probability p for each rate. In these equations, μ corresponds to the average injection rate and σ the injection rate standard deviation.

$$P(rate) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(rate-\mu)^2/2\sigma^2} \tag{7.3}$$

$$P(rate) = \frac{1}{\mu} e^{-rate/\mu} \tag{7.4}$$

Contrary to UR models, the global average injection rate for the KR models is directly obtainable from the particular distribution equation together with other parameters.

According to (Tedesco et al. [2006]), performance requirements of HDTV and MPEG-2 video are two relevant workloads for SoC research groups. These applications comprise large data blocks and must respect strict time intervals. This type of traffic can be efficiently modeled with probabilistic functions and Pareto ON-OFF.

Determining optimal CUSUM parameters

The enhanced CUSUM algorithms used in our solution is explained in details in Section 6.3.3. We tested the algorithm's efficiency against traffic input as shown in Section 7.5.1. Considering that different applications have different communication requirements it is expected that the detection algorithm parameters will have to be tuned accordingly.

Once the test application communication patterns have been set, the values of the attack-detecting algorithm's parameters a and h parameters (refer to Section 7.5) must be decided. This is where the real challenge comes. Parameter a should be greater than the bit rate captured in one time frame for the sake of having negative CUSUM value. Also, parameter a should not be greater than the maximum bit rate, in order to be able to get a positive value of the CUSUM when the attack occurs (see the equation 7.5).

$$CUSUM = \sum_{i=0}^{n} E(X_i - a) \tag{7.5}$$

If a is greater than maximum bit rate value this sum will never be positive and the proposed algorithm cannot be implemented. Thus, in the case of constant injection rate, the parameter a should fall in the following interval:

$$a = (\frac{max_{throughput}}{2}, max_{throughput}) \tag{7.6}$$

Where the maximum throughput is calculated as the product of the maximum number of flits in one time frame multiplied by the maximum number of bits in one flit, divided by the number of clock cycles per time frame (as shown in equation 7.7).

$$\begin{aligned} max_{throughput} &= \left\lfloor \frac{X * Y}{No.of\ clk\ cycles\ per\ timeframe} \right\rfloor \\ &= \left\lfloor \frac{75 * 34}{80} \right\rfloor = 31 \quad bits\ per\ clock\ cycle \end{aligned} \tag{7.7}$$

Where: $X = max\ No.of\ flits\ per\ timeframe\ (design\ specific);\ Y = max\ No.of\ bits\ per\ flit$

Parameter h will determine how fast we will detect the alarm and it directly influences the number of false alerts. In order to decrease the number of detected false alarms, parameter h should not be close to the value of the parameter a. Also, it should not be much greater than parameter a for the sake of detecting the alert in reasonably time. For the purposes of this work we have fixed its value to the $2a$.

Performing different simulations we have observed that the parameter a is the one which has the greatest impact on the algorithm execution. The algorithm is very sensitive to changes of this parameter since it is directly involved in the calculation of the CUSUM. Small fluctuations of this parameter can make CUSUM value extremely negative and this can slow down the detection of the attack (as it may be seen in Figure 7.8). On the other hand the value of the parameter also impacts on false alerts detection as the higher the value of parameter a lowers the possibility of false detections. Obviously a trade-off between number of false alerts and detection latency must be found. A number of simulations for the specific test application settings has been performed and based on the results resented in Figure 7.9 the value of the parameter a has been set to ten.

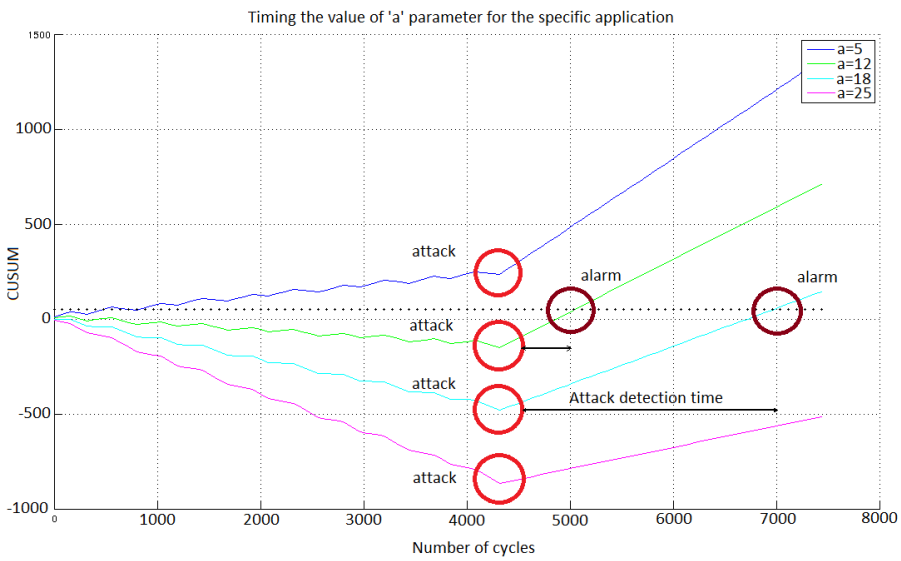

Figure 7.8. Impact of step parameter 'a' on attack detection latency

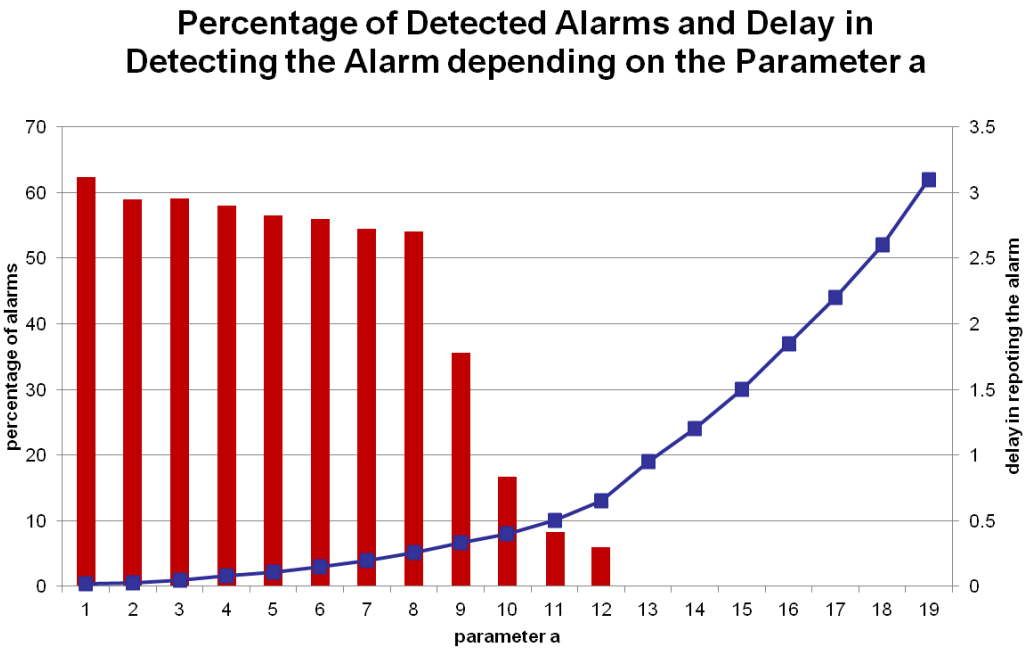

Figure 7.9. Detection latency vs. false alerts trade-off - optimal value of a parameter may be 10 or 11 for the specific case

7.5.2 Costs of the implementation

Considering now the impact on performance, both units ASA/VDoS and ASA/FDoS work fully in parallel to the rest of the system and do not introduce any overhead and do not require any additional code instrumentation. Power overhead is also negligible (less than 1%)

Table 7.4 represents the synthesis results for the area occupied by the whole design. The area is estimated both through the number of slice Regs and the number of slice LUTs.

It can be seen that proposed ASA/FDoS implementation takes around 10% of Slice Registers and around 13% of Slice LUTs consumed by entire system design.

7.6 Validation of the overall security framework

As shown in Section 5.1 attack specific protection solutions (that are realized in form of Attack Specific Agents) are integrated in Local Security Agents and further through Secure NoC into wider (system level) security framework. We consider the system as shown in Figure 7.6. The integration and correla-

Component	Area (Slices)	LUTs
Total System	5751	9802
ASA/VDoS	4	10
ASA/FDoS	63	220

Table 7.4. Resource Utilization in terms of area (implementation on Xilinx Virtex-V FPGA)

tion of different security solutions into one system-level protection mechanism is detailed in Chapter 5.

In order to demonstrate the effectiveness of the proposed solution we have focused on protection against Denial-of-Service attacks, considering that it tests most thoroughly our approach (as it is a nondeterministic detection method whose performance varies with different applications based on their expected and actual behavior). We consider several different aspects of detection efficiency. These include:

- Rate of successful detection of real attacks - percentage of reported alerts considering total number of attacks
- Rate of false alerts notification - percentage of reported (false) alerts considering total number of regular traffic bursts (that may be mistakenly presented by CUSUM as attacks)
- Detection latency - defined as the time needed to raise an alert from the moment when the attack occurred

Our goal is to show how our system level security framework improves attack detection in all these aspects (and at different levels staring from attack specific protection layer to the system level protection layer).

In this Section we present results of DoS attack detection simulations in the case of input traffic described by statistics given in Section 7.5.1 with the addition of a burst traffic event. We have run several types of experiments:

- Detection using basic standalone CUSUM and introducing saturation value (minimal CUSUM value below which it is not possible to go)
- Detection using improved standalone CUSUM detection algorithm and proper security policies favoring multi-level detection (several attack detections cause an alert - depending on defined alert severity level)

- Detection using CUSUM detection algorithm and correlating security policy (i.e. CUSUM complemented with other protections strategies in particular unauthorized memory detection)

As a communication model we consider an multi-media like application characterized with input traffic pattern as defined in Section 7.5.1. In addition to that we simulated another application with burst communication pattern. We relied on CUSUM detection algorithm parameters as previously defined and obtained in Section 7.5.1. Test input traffic is shown in Figure 7.10 where the red arrow denotes malicious traffic. The appropriate calculated value of the two CUSUM versions (basic and saturated) is given in Figure 7.11. The corresponding trusting value (considering only CUSUM detection) is represented in Figure 7.12.

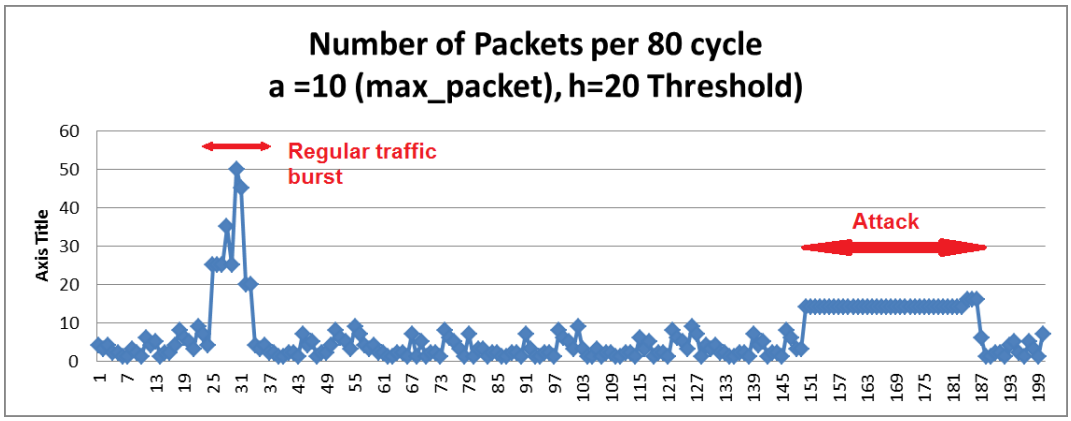

Figure 7.10. Input traffic sent to the NoC by the monitored core (the attack starts around slot 150)

In this case we may notice that for the given traffic inputs and algorithm parameters, the basic CUSUM algorithm detects none of the events while saturated CUSUM detects both false and real one. The reason for such behavior is in the cumulative nature of the CUSUM algorithm. If the value of the CUSUM goes so negative that it would take too long to detect an attack, on the other hand if the saturation value is low, than there is the possibility that even regular traffic bursts would cause false attacks. Considering the nature of FDoS attacks (Blazek et al. [2001]) which must last in time in order to take an effect, it is clear that once detected attack will be detected very soon again (even with CUSUM reset). For this reason we introduce the CUSUM reset value (assigned to CUSUM upon detection of the alert, and equal to half minimal-saturation value) and we assign medium severity level to the CUSUM (i.e. FDOS) alert. In addition to that, a positive feed-backing mechanism is started immediately upon attack detection.

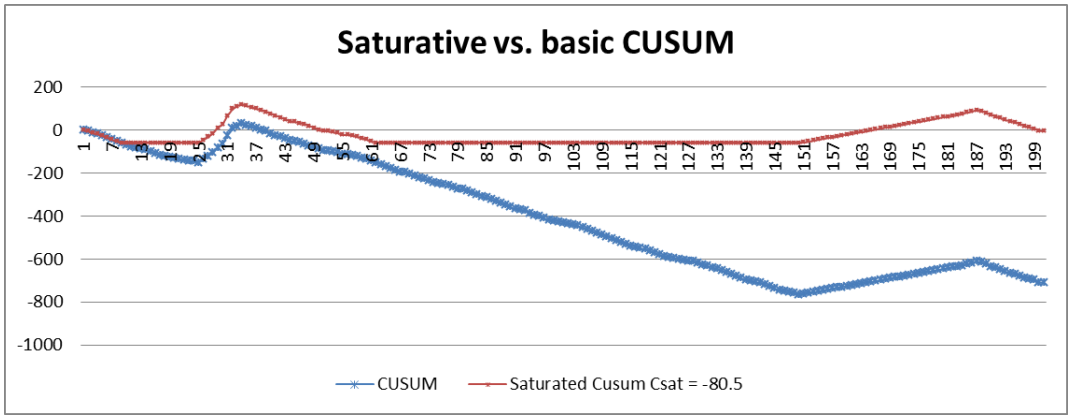

Figure 7.11. Measured CUSUM values - for the basic and saturative algorithms

This actually means that in order to declare presence of a FDoS attack in the system three CUSUM alerts (of medium level) must be detected in a pre-defined time (otherwise trusting value would be increased by the positive feed-backing mechanism). All these enhancements are introduced in appropriate LSA which represents second (i.e. core) level protection. The trusting value change with enhanced CUSUM is presented in Figure 7.12.

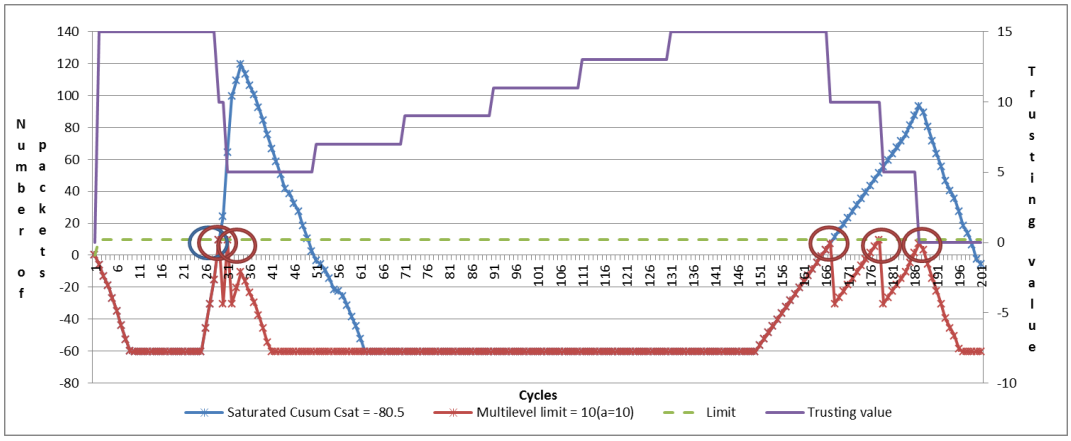

Figure 7.12. Core trusting value changing for basic and enhanced CUSUM algorithm

The analysis of the attack so far has been made considering only statistical elaboration of the traffic produced by the monitored core. If we take a look from another perspective, namely, from the cores that are exposed to the malicious traffic, then we might be able to complement and improve detection of FDoS at the core which implements ASA/FDoS. In fact, FDoS attacks are commonly

targeted at the communication infrastructure in the sense that packets are randomly sent through the network (Blazek et al. [2001]) in order to congest it. It is reasonable to assume that a portion of such packets would eventually hit shared memory cores installed in the system as well. In that case, our ASA/DPU protection mechanism would promptly report access violations (unless the attacker in some manner manages to present false memory access protocols and access rights, which is highly improbable) causing alerts which additionally decrease the trusting value of the attacking core. The value for which the core trusting decreases (considering alerts for ASA/DPU) is calculated according to number of unauthorized memory accesses in defied time window (we consider each such access as an incident and we count the alert level according to the number of incidents in defined time slot, which is equal to five CUSUM observing slots in this case).

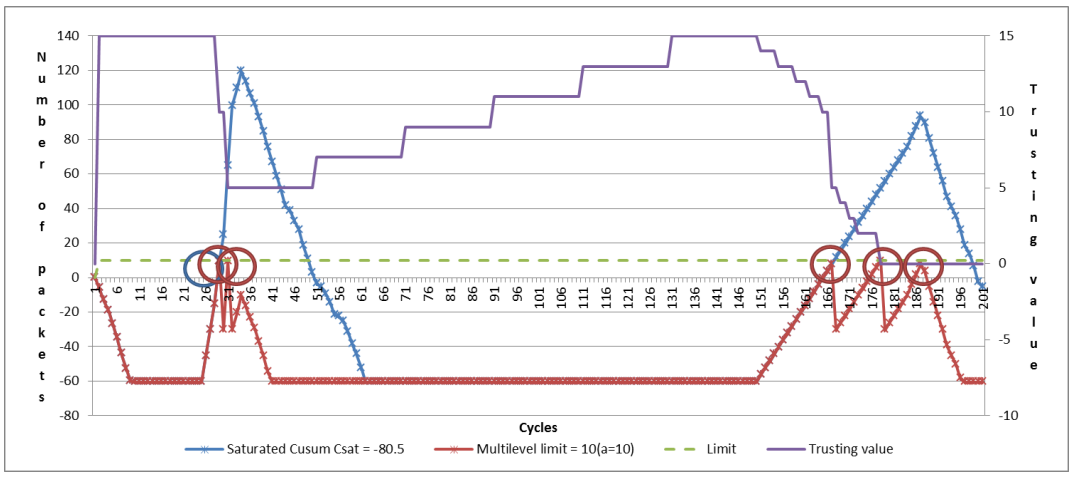

Figure 7.13. Total trusting value (including CUSUM and DPU alerts and positive feed-backing)

Finally figure 7.13 represents trusting values change in case when we consider (i.e. correlate) all the local security agents (meaning all the specific attack agents) simultaneously. These figures clearly show improvements in false alerts removal (by introducing core protection level - LSA modules) as well as in decreasing of detection time latency for a FDoS attack detection. This is achieved, at one side, by improving the CUSUM algorithm provided by the ASA/FDoS with extensions done at LSA (i.e. saturation value, resetting and positive feed-backing). On the other hand, additionally combining security related information from different sources at the system level (i.e. CSA and appropriate security policies) improves significantly latency of the detection and also positively im-

pacts real attacks detection.

7.6.1 Overall security framework efficiency

We have previously shown what are the methods we used to validate our solution. We have also defined efficiency aspects that we use as a reference to compare protection efficiency at different levels. In fact, we compare percents of real and false attacks detection as well as latency in detection at three levels:

- Attack specific protection level - which practically means that we measure efficiency of exact detection mechanism for specific attack. In considered case of FDoS attack, it practically means that we consider only basic ASA/FDoS which implements CUSUM algorithm.

- Core protection level - locally at LSA we process to some extent information obtained from ASAs. In observed FDoS case it means that we, on top of CUSUM value provided by ASA/FDoS, add other mechanisms such as saturation, multiple level alert (by assigning proper severity level to the specific alert), positive feed-backing etc.

- System level protection - at this stage we implement security policy which correlates security information from different sources in order to improve the overall attack detection mechanisms.

We have simulated the input traffic based on replication of the pattern provided in Figure 7.14. Each time slot has 80 clock cycles (which is CUSUM time window as defined in Section 6.3.3) and inside slots there are as many packet transactions over the network as defined in 7.5.1. For the simulation purpose this pattern has been replicated hundred times. There are several parameters that are changing in this pattern in order to provide relevant results: the number of bursts and attacks per slot; intensity of bursts and attacks; duration of both of them.

In the Figure 7.15 we may see that first level (CUSUM based ASA/FDoS) has quite low detection rate with still fairly good latency. Nevertheless, the second protection level (based on LSA enhancements) introduces additional latency in the detection. This effect comes from the introduction of saturation and reset values as well as for the reason that severity level of the alert is decreased to medium so that three alerts from CUSUM result in a final FDoS alert. At the third (i.e. system) level, introduced security policy brings correlation with

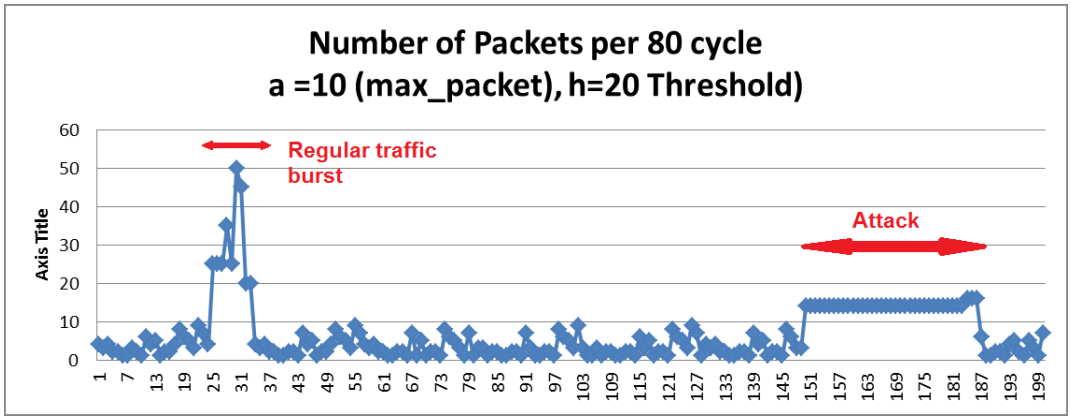

Figure 7.14. Input traffic pattern

other detection mechanisms which complements other FDoS detection strategies. From the figure we may notice that third (actually system wide) protection level brings higher detection rate, with remarkable latency decrease as well.

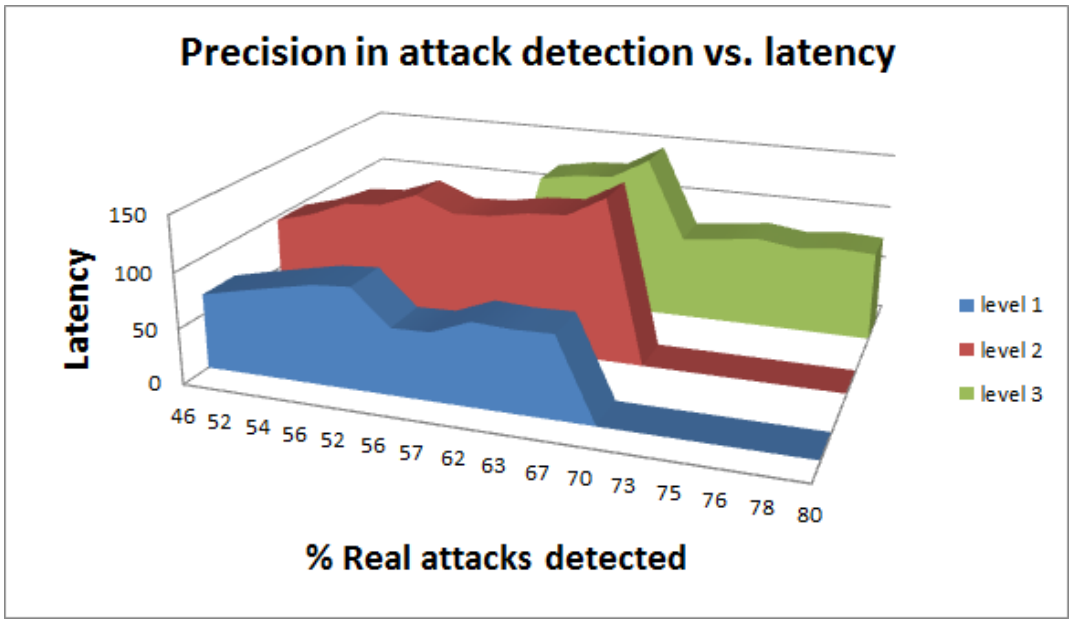

Figure 7.15. Real alerts detection efficiency

Another aspect of the efficiency, as previously discussed, relates to false alerts detection. In fact, regular traffic (especially bursts of regular applications' communication) may be wrongly interpreted by CUSUM as DoS attacks. This effect has been a motivation for introduction of features such as CUSUM reset and pos-

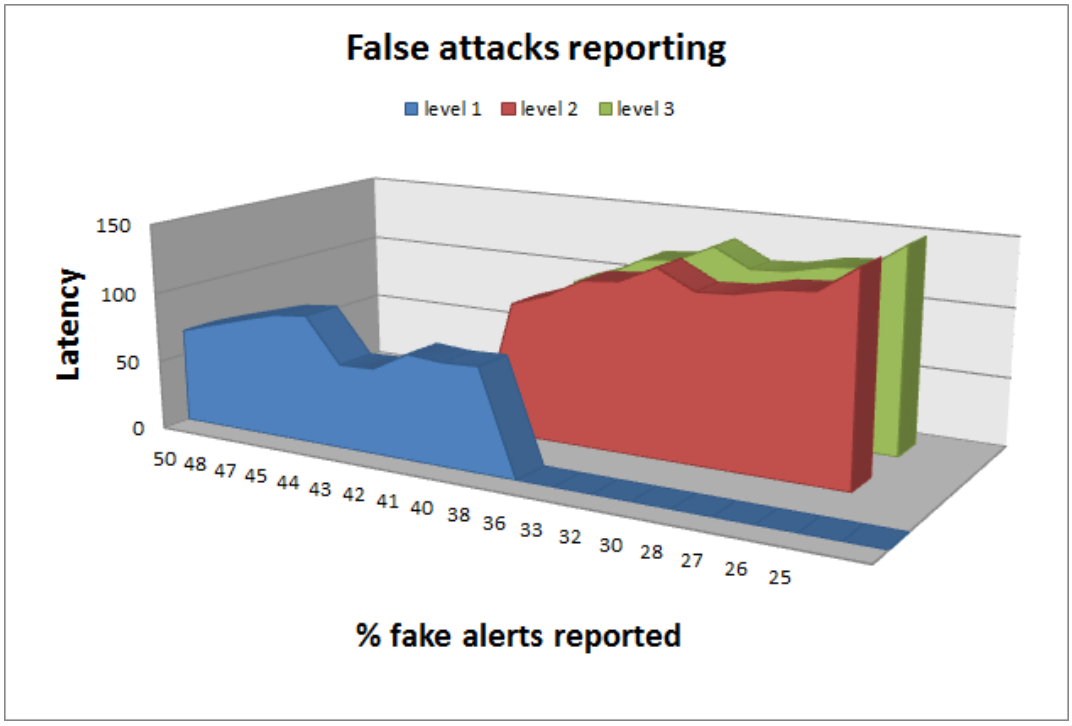

Figure 7.16. False alerts detection - as a consequence of regular traffic bursts

itive feedback (both implemented at second protection level i.e. in LSAs). We may clearly notice from Figure 7.16 that in this case second security level (i.e. LSA) improves the efficiency of the detection considering the decreased percents of false alerts notifications. Still at the third (i.e. system) protection level we do not have any contribution from other agents (in this case ASA/DPU) as false alerts are caused by bursts of the regular traffic which do not cause any memory violations and no alerts from ASA/DPU are raised in that case. As a result the graph looks the same for the second and the third level. Even though the latency is an irrelevant factor in this case it can be still noticed that those fake alerts are reported earlier at the second protection level, compared to the first (ASA) level. Nevertheless, there are no changes at the third (system) level as no contribution from other ASAs is present (again due the fact that only bursts of regular traffic are considered so that no security violations are present in the system).

Finally, considering the security efficiency aspects we defined in Section 7.6 we may state that compared to original specific attack protection mechanisms (first level protection in our case) our added security levels behave as follows:

- Second (i.e. core) level protection layer does not impact precision of real attacks detection but at the same time considerably decreases the rate of

reported false attacks. In turn, it increases detection latency by additional processing

- Third (i.e. system) level protection layer improves detection precision of real attacks while it remarkably improves latency of detection by simultaneous correlation of security related information from different sources. Nevertheless, it does not impact in any way detection of false alerts so that for this aspect of security it is in fact useless.

The analysis presented in this Section clearly shows improvements brought by the proposed multi-level hierarchical approach to MPSoCs security. Nevertheless, it may be assumed that combination (i.e. system level correlation) of more attack specific protection mechanisms would additionally improve overall protection efficiency.

Chapter 8

Assessment and Comparisons with Other Approaches

Different kinds of overheads are inevitable for all types of security solutions. In this Chapter we show what are the costs introduced by the presented implementation compared to other relevant solutions - costs taken into account include performance degradation, area and power overheads. We also discuss Non-Recurring Engineering (NRE) design costs compared to other solutions and we comment on achieved portability and scalability of the solutions.

8.1 General considerations and comparisons with other approaches

Our solution is intended to be very flexible - integrating and correlating many different approaches. The security framework embraces various attack-specific solutions combining them through defined security policy into an efficient system level security protection - which is to the best of our knowledge the first complete implementation of such a protection strategy for MPSoCs.

It is clear that for a single attack protection it does not make sense to build the complete framework as it is intended for protection against multiple threats and its purpose is actually to combine diverse attack specific approaches into one robust protection mechanism. In fact, with increasing of numbers of specific attacks, from which the system is protected, the overhead in relative terms (percentage of area dedicated to 'core' security framework) is decreasing while the efficiency is increasing (the higher the diversity of sources of security relevant information, the better overall system level protection) as shown in Section 8.2.

As mentioned in Section 2 solution most similar to ours is the one described in (Patel et al. [2010]). The architectural framework presented in that work is actually based on behavioral monitoring and verification by a dedicated security processor. Monitoring of application execution is enabled by code instrumentation - adding special dedicated functions at the level of block of instructions. This fact represents the main drawback of the approach as it requires remarkable engineering efforts for adapting to specific application. Furthermore, all the applications to be executed on the platform must be all known in advance by the monitoring security processor. Moreover, it is very difficult to develop the 'permissible behavior' graph for complex applications. Considering all the mentioned facts we may say that the proposed solution has very limited portability and scalability and it may be suitable for embedded systems executing rather simple and straight-forward dedicated applications. Overheads comparisons with this solution are provided in the relevant Sections 8.2, 8.3 and 8.4. As for efficiency of the solution, (Patel et al. [2010]) reports detect approximately 70% of bit flip errors in the control flow instructions (CFIs) which may be caused by whatever security attack. This is comparable with the results which we obtained at the third level protection as our rate (concerning DoS attacks which are far most difficult to detect) varies in range 70-95% for different attack patterns. On the other hand our framework guaranties full protection against buffer overflow as well.

Other approaches rely on application execution monitoring such as the one presented in (Mao and Wolf [2007]; S. and Wolf [2010]) which describes hardware support for security. This solution relies on verification of the real-time created monitoring with a 'monitoring graph' that is statically produced (i.e. in design time, according to several different patterns such as address pattern, control flow pattern, hashed pattern etc.). The verification is performed by a dedicated security structure called 'processing monitor'. This approach is very limited due to two facts:

- Application behavior must be known in advance, in other words the solution might be efficient only in the case of embedded systems executing pre-defined applications with very predictable execution implications. In our approach most attack specific protection mechanism do not require any knowledge of specific behavior pattern apart of VDoS protection - that requires communication requirements of the application that are still far less difficult to determine.

- On the other hand, determining 'permissible' behavior of the complex applications and building the appropriate 'monitoring graph' may require

considerable resources (in terms of memory, area, power consumption etc.) as well as design time. Considerable processing power may be required for run-time performance of verification algorithms. Our solution requires no additional monitoring mechanism; elementary memory is required for storing trusting tables and temporal values of variables needed for different services such as positive feed-backing; all the security related processing is performed in parallel to regular system (by the dedicated hardware built structures) so that minimal performance overhead is burden to the main system.

The above listed issues represent the key differentiation points between the two approaches. Authors claim in (S. and Wolf [2010]) that additional logic and memory correspond to roughly one-tenth of the application binary sizes which is comparable to our solution. Apart from that, we are not able to compare area and power consumption as these data are not provided.

Some sort of collaborative monitoring for embedded systems security has been introduced by (Wolf et al. [2006]). In this work 'collaborative monitoring logic' has been introduced to correlate multiple concurrent events in run-time. They employ a 'processing monitoring system' in parallel to a 'thermal monitoring system' to supply the information to the 'collaborative monitoring logic' which then makes different decisions based on collected information. The 'processing monitoring system' is practically the same as the one described in (Mao and Wolf [2007]; S. and Wolf [2010]) and the 'thermal monitoring system' collects the data on temperature at specific points on the chip. The authors rely on the assumption that disorders in system operation can reflect on heating in some predictable pattern. The solution has been tested through SimpleScalar simulator and authors claim that it introduces 13-15% memory overhead. The main shortcomings of this solution are the same as for aforementioned implementation provided by (Mao and Wolf [2007]; S. and Wolf [2010]) and moreover the correlation between temperature changes and application activities is not satisfactorily proved.

Nevertheless, our solution is still, to the best of our knowledge, the only one fully implemented and verified in FPGA technology. Furthermore, it is the only solution that utilizes multi-agent systems structured in a hierarchical fashion which implements security policy that correlates different attack specific detection approaches.

8.2 Area overhead

Considering that our solution is mostly implemented in hardware, the cost in terms of area (i.e. logic cells) used represents the major concern. According to synthesis results information shown in Section 7 we have presented the security framework area consumption data in Table 8.1.

Design	**Area (Slice Regs)**	**Slice LUTs**
Clean design	6295	7614
Sec. framework + ASA/SPU	7775	9727
Sec. framework + ASA/SPU+FDoS	8339	10768
Sec. framework + ASA/SPU+FDoS+DPU	8700	11344
Total system (with all 4 ASAs)	8704	11354

Table 8.1. Resource utilization of the components of the system for the Xilinx Virtex V FPGA on ML510 board

In fact, two main contributors to the area consumption can be distinguished:

- Costs introduced by attack specific protection solutions (implemented in form of Attack Specific Agents in our system). These mostly represent already developed solutions adapted for specific problem and technology used in our setup.

- Costs introduced by 'core' structures of security framework. These include Local Security Agents that coordinate all Attack Specific Agents assigned to a core; Secure NoC and finally Central Security Agent.

It should be noted that the area consumption of the entire security framework increases with addition of attack specific agents for two reasons: due to addition of ASAs and due to extension of supporting 'core framework' modules such as LSA and CSA (e.g. ASA portfolio in LSAs, Security Policy Manager in CSA etc.) to accommodate newly added ASAs.

Let us now consider the protection for the given system from a set of given specific set of attacks. We may compare two approaches:

- Implementation of independent standalone solutions for each attack protection separately

- Implementation of the attack specific protections (in the same manner as above) with the addition of their mutual integration and correlation in a system level security framework

It is clear that implementation of all attack specific solutions to be integrated in a wider security framework would have the same cost as the sum of all independent-standalone solutions. Still, in the first listed case, the system security strategies would not be coordinated and the overall efficiency of the protection mechanism would be as strong as the weakest attack specific protection implemented. On the other hand, our solution would in addition to the area consumed by attack specific modules (which would be actually the same as a sum of aforementioned standalone solutions) have some extra costs introduced by the 'core' of the security framework which manages security at system level (i.e. executes the security policy correlating all the implemented protection mechanisms). The area overhead in absolute terms is increasing with addition of each attack specific agent (see Figure 8.1) but the percentage of the area consumed by the core framework inside total security overhead is decreasing in relative terms (meaning that percentage of the area consumed by the core framework compared to the area consumed by the total security framework is decreasing with addition of ASAs).

In other words, comparing the two cases (with and without core security framework), we may say that the differences in cost of implementation decrease with increasing numbers of specific attack protection types that are to be implemented in the system. From this analysis we may conclude that our proposed security framework pays-off for greater numbers of ASAs. Considering that modern systems are commonly defended against a large number of possible attacks, it is reasonable to assume that our solution would be appropriate for the most modern systems.

Comparison with other solutions in terms of area consumption is fairly difficult to be performed. The main reason for that is that solutions are not fully compatible. For instance, the most relevant system level protection solution provided by (*Hardware-software design methods for security and reliability of MPSoCs* [2009]; Patel et al. [2010]) considers only the buffer overflow type of the attack. For that case area overheads for different types of the security monitor are reported to be between 22-39%. It can be seen that our proposed counterpart security framework takes around 10% of Slice Registers and around 18% of Slice LUTs consumed by entire system design (for the solution which considers buffer overhead protection only). These result is slightly better compared to those provided in (Patel et al. [2010]). Still, our fully built framework with

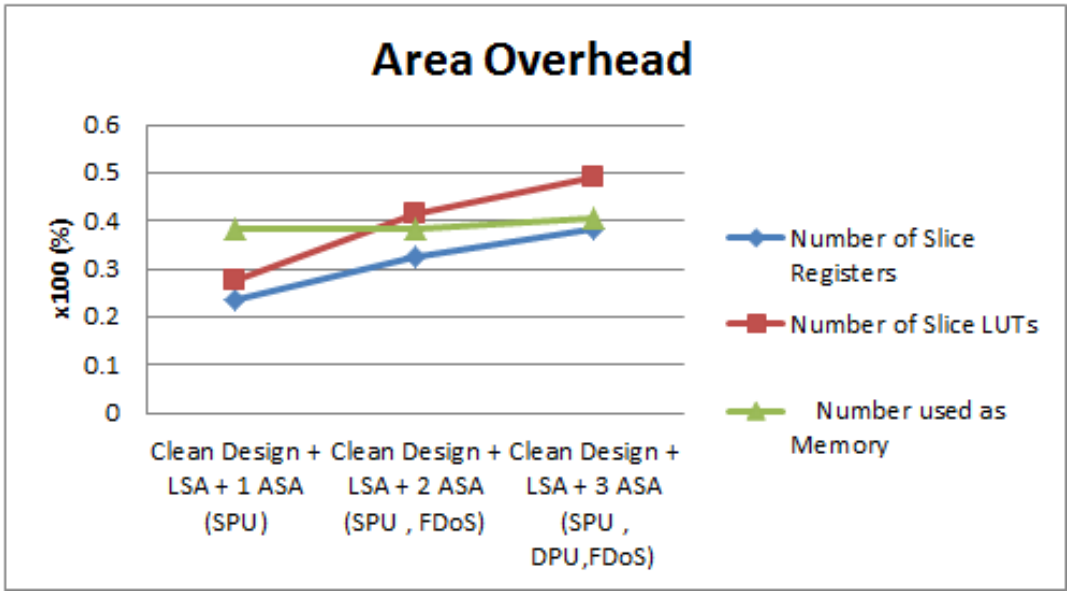

Figure 8.1. Absolute increase of area consumption with addition of core framework (i.e. LSA+CSA+SNoC) and extra ASAs with respect to original design

four different protection mechanisms (i.e. four ASA types) and support for the second and the third security levels consumes around 40% of area taken by the original design. Area overhead reported by (Sepulveda et al. [2011]) is 26.7%.

8.3 Power consumption overhead

Power consumption may be static (quiescent) and dynamic. We have collected information on power consumption via a tool available in the Xilinx toolbox, namely XPower Analyzer which operates at layout level. According to information provided by Xilinx (XilinxPower) power is considered and calculated according to the following definitions (*XPower Analyzer* [2011]):

- Quiescent power (i.e. static power) is the power drawn by the device when it is powered up, configured with user logic and there is no switching activity. In XPower Analyzer, the value reported for Total Quiescent Power is composed of these quiescent power components:
  - The device static power - represents the power consumed by the device when it is powered up without programming the user logic. Any change affecting the device operating environment will affect this power.

  - Design static power - represents the power consumed by the user logic when the device is programmed and without any switching activity. For instance, depending on the device family and resource configuration, some blocks used in a design (such as clock management, I/Os, and Multi-Gigabit Transceivers) will consume a set amount of power regardless of activity.

- Dynamic power represents the fluctuating power as the device runs. It represents the amount of power generated by the switching user logic and routing.

Having defined power consumptions like this, we have run the power analyzing tool (which emulates layout hardware) on our design executing the applications with communication patterns used for experiments as defined in Section 7.5.1 as benchmark. In this analysis we consider only dynamic power consumption as the differences in static power are negligible.

The implemented security framework has basically three sources of additional power consumption: signals (representing the power consumed by all routing structures in the device that connect logic elements, IOs and dedicated blocks) BRAM/memories and logic. It turns out that signals and logic consume the greatest part of dynamic consumption after memories. Consumption in BRAMs is mostly due to components that contain look-up tables as it is the case with CSA (containing trusting policy tables) and DPU (containing access permissions policy table).

The Table 8.2 shows the consumption of the entire design starting from basic design with step by step addition of security framework components, at first core components (which include SNoC, CSA and LSA placeholder - as no ASA are present) and later one by one ASA.

The Figure 8.2 shows increase in power consumption with addition of ASAs, one by one. It may be noticed that the greatest growth in power consumption is due to the core of the security framework (this actually represents the basic cost introduced by our solution and it is still lower then 6% of total consumption of the system). Adding attack specific protection as well as elements for their integration (e.g. LSA modules, CSA upgrades etc.) introduces additional costs as may be seen from the same figure.

Nevertheless, it is very difficult to make comparisons with other solutions concerning power consumption as technologies utilized are different. Still, in (Patel et al. [2010]) the authors claim that the security monitor consumes between 12-19% of total power consumption for the implementation on Xtensa

Table 8.2. Power consumption of the design with subsequent addition of an ASA measured for the Xilinx Virtex V FPGA

Design	Power consumption in mW
Clean design	152.41
Added core security framework	160.25
Added ASA/SPU	162.64
Added ASA/FDoS	169.07
Added ASA/DPU	175.74
Added ASA/VDoS	180.87

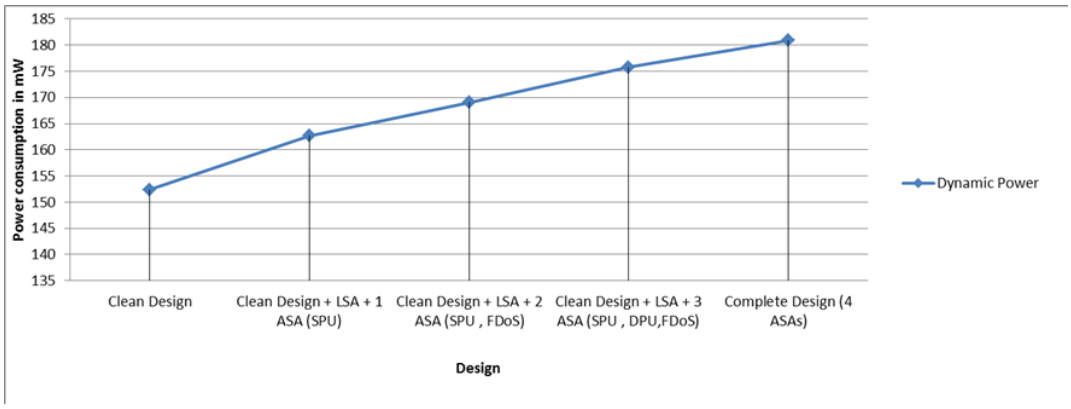

Figure 8.2. Absolute increase of power consumption with addition of an extra ASA with respect to original 'clean' design

processor from Tensilica. It varies for different benchmarks. Our solutions introduces increase of power consumption (for the test application we used as defined in Section 7.5.1) from 6% (for one ASA solutions) to 19% for four-ASA solution. On the other hand in (Sepulveda et al. [2011]) report 19.6% power overhead and only 7.6% in (Sepulveda et al. [2012]), nevertheless these results are given for SystemC-TLM simulation framework and their comparison with data provided form hardware implementation is not fully relevant.

From the analysis above we may conclude that costs in terms of power consumption caused by the proposed solution are acceptable and moreover as well as previously shown for area overheads, the solution pays if a large number of specific attack protection mechanisms implemented.

8.4 Impact on performance

The major portion of our system is implemented in hardware. Therefore, execution of most attack specific algorithms is performed in parallel to regular system execution. This concerns for instance memory access protection (i.e. ASA/DPU) as well as DoS protection mechanisms (both ASA/VDoS and ASA/FDoS-CUSUM algorithms are executed in parallel to regular system operation). Furthermore, core security framework (i.e. LSAs, SNoC and CSA) is fully independent from the main system. In this way we do not burden the system with consumption of its neither computational nor computational resources.

Nevertheless, implementation of buffer overflow protection (i.e. ASA/SPU) requires interaction with operating system at certain times. Since the SPU works by tracking memory accesses that originate from the core and by detecting instructions that could cause the violation (stack manipulation), it does not introduce any additional delays to the pipeline or to the functional units of the processor. The only impact on performance could come from the fact that the SPU takes six cycles to adapt to the new stack frame when the context switch occurs.

We executed several experiments to evaluate the impact on performance. The test consisted of executing several benchmarks from the MiBench suite (Guthaus et al. [2002]) (encoding and decoding of *jpeg* images, *dijkstra* and *patrizia* network algorithms and *bitcount* from automotive suite) and of compressing and decompressing a video using the MPEG2 standard (benchmark from MediaBench Lee et al. [1997]). All programs were executed on the system with and without the SPU, and we found that the overhead introduced by the security framework is negligible (less than 1% in all cases) and comparable with the solution presented in (Patel et al. [2010]).

Chapter 9

Conclusions and Future Work

Security of MPSoCs represents a novel challenge which attracts wider attention of research community. Adoption of common well-accepted security strategies developed in general purpose computing and computers networks are not always suitable for such systems with limited resources. On the other hand MPSoCs are getting used for widest range of applications being exposed to threats coming also from Internet. As a response to these trends we have proposed an agent based hierarchical structure as a solution for securing these systems at multiple levels.

The solution and implementation shown, represent an innovative approach to securing NoC based MPSoCs. The proposed framework does not require any additional application code instrumentation or any change of the functionalities of existing IP cores. Still, proposed framework requires minimal intervention in customizing the attack-specific solutions for specific cores and technology. Nevertheless, the modular and scalable design of the architecture enables easy system upgrades which requires only 'check-in' of the additional module into the existing structure.

Although the presented implementation is limited to the MicroBlaze softcore, it can be easily adapted to other processors. If for instance, the specific processing core has the tracing capabilities (like MicroBlaze has) ASA/SPU can be simply attached to the trace signals. In other cases, SPU can monitor the addresses that are coming out of the processor, and the instruction decode needs to be changed to signal instructions that manipulate the stack. No other complex interventions on the datapath or on the micro-architecture are required for this specific attack protection solution.

The code injection attacks - in particular 'buffer overflow' based ones, considered in the implemented prototype - represent a growing threat for MPSoCs

especially Internet enabled ones that are exposed to the danger of downloading malicious applications. It has been shown in the testing and evaluation phase that the system has been fully protected from such kind of attacks whether they are targeted to exploit software vulnerabilities in application run-time execution or to tamper the shared memory tampering.

On the other hand nondeterministic, statistic based, detection algorithm for Denial-of-Service has been used as a demonstrator of the efficiency of the hierarchical approach which bring wider integration and correlation among different protection mechanisms at system level. It is shown that such approach improves efficiency of DoS attack detection. DoS related solutions are fully portable and independent from technology utilized.

The synthesis results and tests performed show that our solution introduces minimal time overhead; area overhead as well as increased power consumption are fairly remarkable but still comparable with similar solutions proposed. Nevertheless, costs introduced may be considered in limits of acceptable keeping in mind that security solutions always require considerable resources.

9.1 Concluding evaluations and remarks

The fundamental benefits provided by the presented solution lies in introduced system level security strategy which:

- Improves efficiency of each specific attack protection mechanism itself
- Coordinates and manages all the security related mechanisms and actions in the system enabling early warning, intrusion detection system and preventing fault propagation
- NoCs proved as useful medium for deployment of auxiliary services other than communication. By hosting security related components which further profit from services provided by Network Interfaces, NoCs enable building of fully parallel security structure which is not burdening existing computational and communicational resources of MPSoC itself
- The developed structures enable easy upgrade and extension of the security strategies by adding new attack specific protection mechanisms

All the aforementioned benefits have been brought to the system with acceptable area consumption achieved with low power consumption overhead and without code instrumentation. In the sequel we comment on the one of the most important properties of the solution - its portability and scalability.

9.1.1 Portability and scalability of the solution

Our solution may be considered as composed from two parts - attack specific solutions (which is composed of ASAa) and core security framework (which includes LSA, SNoC and CSA). All the architecture specific elements are encapsulated in the attack specific protection layer. In fact, ASAs are those elements who must be tailored to adopt to the exact architecture, communication standards and protocols of the core they are attached to. These elements are embedded into NIs of the network-of-chip and they are built to rely on these structures. The core security framework is fully portable, architecturally and technologically independent solution. The concept can be adopted for any NoC based MPSoC.

Thanks to modular and flexible agent based structure, the solution is very scalable both in terms of supporting increased number of cores in the system as well as number of specific attacks the system is protecting from. Therefore, the solution is capable of coping with foreseen increase in number of cores and at the same time to adapt to growing security threats.

9.2 Future work

The solutions presented here represents the solid foundations for security framework implementation in terms of architecture and system level security policy. Still, remarkable improvements can be done mostly on the system security policy level. Our current efforts are focused on implementing proper security policies that would enable construction of enhanced trust relationships in the system, based on centralized approach. Clearly establishing the trust relationships in the system, not only from an architectural perspective, but also from a policy perspective would mean that the secure hierarchy should be able to make a distinction between the expectation and the exception.

As future work in the scope of this framework we also consider adding more Attack Specific protection solutions as well as increasing the role of the CSA in terms of improved attack analysis and countermeasures performed. A more detailed analysis of performance overhead will be performed as well (e.g. we will consider complex multithreaded and multiprogramming workloads).

Bibliography

Ahmad, B. and Arslan, T. [2005]. Dynamically reconfigurable NoC for reconfigurable MPSoC, *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pp. 277–280.

Ahmad, B., Erdogan, A. and Khawam, S. [2006]. Architecture of a Dynamically Reconfigurable NoC for Adaptive Reconfigurable MPSoC, *Proceedings of AHS'06*, pp. 405–411.

Alves, T. and Felton, D. [2004]. Trustzone: Integrated hardware and software security, White paper, ARM.

Alves, T. and Rudeli, J. [2007]. ARM Security Solutions and Intel Authenticated Flash - How to integrate Intel Authenticated Flash with ARM TrustZone for maximum system protection, Design Reuse. http://www.design-reuse.com/articles/16975/arm-security-solutionsand-intel-authenticated-flash-how-to-integratewith-arm-trustzone-intel-authenticated-flash-how-to-integrateintel-authenticated-flash-for-maximum-system-protection.html.

Arora, D., Ravi, S., Raghunathan, A. and Jha, N. K. [2005]. Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring, *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, IEEE Computer Society, Washington, DC, USA, pp. 178–183.

Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. [2004]. Basic concepts and taxonomy of dependable and secure computing, *Dependable and Secure Computing, IEEE Transactions on* **1**(1): 11–33.

Barford, P. [2002]. A Signal Analysis of Network Traffic Anomalies, *In Proceedings of ACM SIGCOMM Internet Measurement Workshop, ACM Press* pp. 71–82.

Bartic, T., Desmet, D., Mignolet, J.-Y., Marescaux, T., Verkest, D., Vernalde, S., Lauwereins, R., Miller, J. and Robert, F. [2004]. Network-on-Chip for Recon-

figurable Systems: From High-Level Design Down to Implementation, *Proceedings of FPL'04*, pp. 637–647.

Bartic, T., Mignolet, J., Nollet, V., T. Marescaux, T., Verkest, D., Vernalde, S. and Lauwereins, R. [2005]. Topology adaptive network-on-chip design and implementation, *IEE Proceedings - Computers and Digital Technologies* **152**(4): 467–472.

Bartic, T., Mignolet, J.-Y., Nollet, V., Marescaux, T., Verkest, D., Vernalde, S. and Lauwereins, R. [2003]. Highly Scalable Network on Chip for Reconfigurable Systems, *System-on-Chip, 2003. Proceedings. International Symposium on*, pp. 79–82.

Basseville, M. and Nikifirov, I. [1993]. *Detection of the Abrubt Changes: Theory and Application*, NJ: PTR Prentice Hall.

Benini, L. and Bertozzi, D. [2005]. Network-on-chip architectures and design methods, *Computers and Digital Techniques, IEE Proceedings* **152**(2): 261–272.

Benini, L. and De Micheli, G. [2002]. Networks on Chips: A New SoC Paradigm, *IEEE Computer* **35**(1): 70–78.

Benini, L. and De Micheli, G. [2006]. *Networks on Chips: Technology and Tools*, Morgan Kaufmann.

Benner, T., Ernst, R., Koenenkamp, I., Holtmann, U., Schaub, H.-C. and Serafimov, N. [1994]. FPGA Based Prototyping for Verification and Evaluation in Hardware-Software Cosynthesis, *Proceedings of FPL'94*, pp. 251–258.

Bishop, P. [n.d.]. Using ARM Processor-based Flash MCUs as a Platform for Custom Systems-on-Chip, Design Reuse. http://www.design-reuse.com/articles/13742/using-arm-processor-based-flash-mcus-as-a-platform-for-custom-systems-on-chip.html.

Bjerregaard, T. and Mahadevan, S. [2006]. A survey of research and practices of Network-on-Chip, *ACM Computing Surveys* **38**(1): 1–51.

Bjerregaard, T., Mahadevan, S., Olsen, R. and Sparsoe, J. [2005]. An OCP compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip, *Proceedings of System-on-Chip, 2005. International Symposium on*, pp. 171–174.

Blazek, R., Kim, H., Rozovskii, B. and Tartakovsky, A. [2001]. A Novel Approach to Detection of Denial-of-Service Attack via Adaptive Sequential and Batch-sequential Change-Point Detection Method, *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, pp. 220–226.

Bobda, C. and Ahmadinia, A. [2005]. Dynamic interconnection of reconfigurable modules on reconfigurable devices, *Electronics and Electrical Engineering* **22**(5): 443–451.

Bolotin, E., Guz, Z., Cidon, I., Ginosar, R. and Kolodny, A. [2007]. The Power of Priority: NoC Based Distributed Cache Coherency, *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 117–126.

Bonhomme, C., Feltus, C. and Khadraoui, D. [2010]. A multi-agent based decision mechanism for incident reaction in telecommunication network, *In Proceedings of Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*.

Borselius, N. [2003]. Multi-agent system security for mobile communication, Royal Holloway, University of London. PhD Thesis.

Bulba and Kil3r [2000]. Bypassing StackGuard and StackShield, *Phrack Magazine* **10**(56).

Carl, G., Kesidis, G., Brooks, R. and Rai, S. [2006]. Denial-of-Service Attack-Detection Techniques, *Internet Computing, IEEE* (Volume 10, Issue 1): 82–89.

Cavalcante, R. C., Bittencourt, I. I., da Silva, A. P., Silva, M., Costa, E. and Santos, R. [2012]. A survey of security in multi-agent systems, *Expert Systems with Applications* **39**: 4835–4846.

Ciordas, C., Goossens, K., Basten, T., Radulescu, A. and Boon, A. [2006]. Transaction Monitoring in Networks on Chip: The On-Chip Run-Time Perspective, *Proceedings of the Industrial Embedded Systems, IES '06. International Symposium on*, pp. 1–10.

Clegg, R. [2007]. Simulating Internet traffic with Markov-modulated processes, *In Proceedings of UK Performance Engineering Workshop*.

Coburn, J., Ravi, S., Raghunathan, A. and Chakradhar, S. [2005]. SECA: Security-Enhanced Communication Architecture, *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pp. 78–89.

Collin, M., Haukilahti, R., Nikitovic, M. and Adomat, J. [2001]. SoCrates - A multiprocessor SoC in 40 days, *Proceedings of DATE'01*.

Cotret, P., Crenne, J., Gogniat, G., Diguet, J., Gaspar, L. and Duc, G. [2011]. Distributed Security for Communications and Memories in a Multiprocessor Architecture, *In Proceedings of Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 326–329.

Cowan, C., Pu, D., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S. Grier, A., Wagle, P. and Zhang, Q. [1998]. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proceedings of the 7th USENIX Security Symposium*.

D., W. W. [2000]. Enabling Reuse via an IP Core-centric Communications Protocol: Open Core Protocol, Sonics, Inc.

Dall'Osso, M., Biccari, G., Giovannini, L., Bertozzi, D. and Benini, L. [2005]. Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs, *Proceedings of Computer Design, 2003. Proceedings. 21st International Conference on*, pp. 536–539.

Dally, J. W. and Brian, T. [2001]. Route Packets, Not Wires: On-Chip Inteconnection Networks, *Proceedings of DAC'01*, pp. 684–689.

Davidson, E. M., McArthur, S. D. J., McDonald, J. R., Cumming, T. and Watt, I. [2006]. Applying multi-agent system technology in practice: Automated management and analysis of SCADA and digital fault recorder data, *Power Systems, IEEE Transactions on* **21**: 559–567.

De Bosschere, K., Luk, W., Martorell, X., Navarro, N., O'Boyle, M., Pnevmatikatos, D. Ramirez, A., Sainrat, P., Seznec, A., Stenstrom, P. and Temam, O. [2007]. High-Performance Embedded Architecture and Compilation Roadmap, *Transactions on High-Performance Embedded Architecture and Compilation* **4050/2007**: 5–29.

DeMicheli, G. and Benini, L. [2006]. *Networks on Chips*, Morgan Kaufmann.

Derin, O., Kabakci, D. and Fiorin, L. [2011]. Online Task Remapping Strategies for Fault-tolerant Network-on-Chip Multiprocessors, *NOCS '11: Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 1–8.

Diguet, J. P., Evain, S., Vaslin, R., Gogniat, G. and Juin, E. [2007]. NoC-centric Security of Reconfigurable SoC, *In Proceedings of the First International Symposium on Networks-on-Chip (NOCS07)*, pp. 223–232.

Dimeas, A. and Hatziargyriou, N. [2011]. Operation of a multi-agent system for microgrid control, *Power Syst, IEEE Transactions on* **20**: 1447–1455.

Dor, N., Rodeh, M. and Sagiv, M. [2003]. CSSV: towards a realistic tool for statically detecting all buffer overflows in c, *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 155–167.

Duato, J. [2008]. Managing Heterogeneity in Future NoCs, *Proceedings of the first Networks-on-Chip Architectures workshop (NoCArc) held in conjunction with 41st MICRO*.

Ehliar, A. and Liu, D. [2007]. An FPGA Based Open Source Network-on-Chip Architecture, *In Proceedings of Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 800–803.

Ekberg, J. and Asokan, N. [2010]. External Authenticated Non-volatile Memory with Lifecycle Management for State Protection in Trusted Computing, *Trusted Systems, LNCS* **6163/2010**: 16–38.

Ekberg, J.-E. and Kylanpaa, M. [2007]. Mobile trusted module, Technical Report NRC-TR-2007-015, Nokia Research Center. http://research.nokia.com/files/NRCTR2007015.pdf.

Faruque, M., Weiss, G. and Henkel, J. [2006]. Bounded arbitration algorithm for QoS-supported on-chip communication, *in CODES+ISSS 06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 76–81.

Feero, B. S. and Pande, P. P. [2009]. Networks-on-Chip in a three-dimensional environment: a performance evaluation, *IEEE Transactions on Computers* **58**(1): 32–45.

Feltus, C., Khadraoui, D. and Aubert, J. [2010]. A security decision-reaction architecture for heterogeneous distributed network, *In Proceedings of Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pp. 1–8.

Ferrandi, F., Ferrara, G., Palazzo, R., Rana, V. and Santambrogio, M. [2006]. VHDL to FPGA automatic IP-Core generation: a case study on Xilinx design flow, *Proceedings of IPDPS'06*.

Ferrante, A., Pompei, R., Stulova, A. and Taddeo, A. V. [2008]. A Protocol for Pervasive Distributed Computing Reliability, *Proceedings of SecPriWiMob 2008*.

Fiorin, L., Palermo, G., Lukovic, S., Catalano, V. and Silvano, C. [2008]. Secure Memory Accesses on Networks-on-Chip, *IEEE Trans. on Computers* **57**(9): 1216–1229.

Fiorin, L., Palermo, G., Lukovic, S. and Silvano, C. [2007]. A Data Protection Unit for NoC-based Architectures, *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*, pp. 167–172.

Fiorin, L., Palermo, G. and Silvano, C. [2007]. Implementation of a Reconfigurable Data Protection Module for NoC-based MPSoC, *Proceedings of the Fifth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07*.

Fiorin, L., Palermo, G. and Silvano, C. [2009]. MPSoCs Run-Time Monitoring through Networks-on-Chip, *Proceedings of the Conference on Design, Automation and Test In Europe (DATE'09)*, pp. 558 – 561.

Frantzen, M. and Shuey, M. [2001]. Stackghost: Hardware Facilitated Stack Protection, *Proceedings of the 10th USENIX Security Symposium*, pp. 681–685.

Gallery, E. and Mitchell, C. J. [2009]. Trusted Computing: Security and Applications, *Cryptologia* (Volume 33, Issue 3): 217–245.

Gehrmann, C. and Lofvenberg, J. [2011]. Trust Evaluation for Embedded Systems, Security research challenges identified from an incident network scenario, *in First International Workshop on Dependable and Secure Industrial and Embedded Systems (WORDS 2011)*.

Gong, L. and Ellison, G. [2003]. *Inside Java 2 Platform Security: Architecture, API Design and Implementation*, (The Java Series) Addison-Wesley, Upper Saddle River, NJ.

Goossens, K., Dielissen, J., Gangwal, O., Pestana, S., Radulescu, A. and Rijpkema, E. [2005]. A Design Flow for Application-Specific Networks on Chip

with Guaranteed Performance to Accelerate SOC Design and Verification, *Proceedings of DATE'05*, pp. 1182–1187.

Goossens, K., Dielissen, J. and Radulescu, A. [2005]. Aethereal Network on Chip: Concepts, Architectures, and Implementations, *Design and Test of Computers, IEEE* **22**(5): 414–421.

Grecu, C., Pande, P., Ivanov, A. and Saleh, A. [2004]. A Scalable Communication-Centric SoC Interconnect Architecture, *Quality Electronic Design, International Symposium on*, pp. 343–348.

Greenberg, S. [2011]. Intel acquire mcafee. *http://newsroom.intel.com/community/intel_newsroom/blog/*2011/02/28/.

Gu, H., Xu, J. and Zhang, W. [2009]. A Low-Power Fat Tree-based Optical Network-On-Chip for Multiprocessor System-on-Chip, *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE '09*, pp. 3–8.

Guerrier, P. and Greiner, A. [2000]. A generic architecture for on-chip packet-switched interconnections, *Design, Automation and Test in Europe Conference and Exhibition 2000*, pp. 250–256.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T. and Brown, R. [2002]. MiBench: A free, commercially representative embedded benchmark suite, *IEEE International Workshop on Workload Characterization, 2001. WWC-4. 2001*, Ieee, pp. 3–14.

Guz, Z., Walter, I., Bolotin, E., Cidon, I., Ginosar, R. and Kolodny, A. [2006]. Efficient link capacity and QoS design for Network-on-Chip, *in DATE'06: Proceedings of the conference on Design, automation and test*, pp. 9–14.

Hardware-software design methods for security and reliability of MPSoCs [2009]. The University of New South Wales. Thesis PhD Doctorate.

Harmanci, M., Pazos, N., Leblebici, Y. and P., I. [2005]. Quantitative Modelling and Comparison of Communication Schemes to Guarantee Quality-of-Service in Networks-on-Chip, *in Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS04)*, pp. 1782–1785.

Hecht, R., Kubisch, S., Herrholtz, A. and Timmermann, D. [2005]. Dynamic reconfiguration with hardwired networks-on-chip on future FPGAs, *Proceedings of FPL'05*, pp. 527–530.

Henkel, J. and Wolf, W. [2004]. On-chip networks: a scalable, communication-centric embedded system design paradigm, *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 845–851.

Hilton, C. and Nelson, B. [2006]. Pnoc: a flexible circuit-switched NoC for FPGA-based systems, *Computers and Digital Techniques, IEE Proceedings* **153**: 181–188.

Hiroaki, I., Sakai, J. and Edahiro, M. [2008]. Processor virtualization for secure mobile terminals, *ACM Trans. Des. Autom. Electron. Syst.* **13**(3): 1–23.

Ho, R., Mai, K. and Horowitz, M. [2001]. The Future of Wires, *Procedings of the IEEE* **89**(4): 490–504.

Howard, J. and Longstaff, T. [1998]. *A Common Language for Computer Security Incidents*, Sandia National Laboratories.

Hu, J. and Marculescu, R. [2004a]. Application-specific buffer space allocation for Networks-on-Chip router design, *in ICCAD 04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pp. 354–361.

Hu, J. and Marculescu, R. [2004b]. DyAD Smart Routing for Networks-on-Chip, *in Proceedings of 41st Design Automation Conference*, pp. 260–263.

Hussain, A., Heidemann, J. and Papadopoulos, C. [2003]. A Framework for Classifying Denial of Service Attacks, *SIGCOMM '03 Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* pp. 99–110.

Igure, V. and Williams, R. [2008]. Taxonomies of Attacks and Vulnerabilities in Computer Systems, *Communications Surveys and Tutorials, IEEE* **10**(1): 6–19.

Inoue, H., Ikeno, A., Kondo, M., Sakai, J. and Edahiro, M. [2005]. FIDES: an advanced chip multiprocessor platform for secure next generation mobile terminals, *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 178–183.

Jennings, N. R. [2001]. An Agent-Based Approach for Building Complex Software Systems, *Communications ACM* **44**: 35–41.

Kim, J., Nicopoulos, C., Park, D., Das, R., Xie, Y., Narayanan, V., Yousif, M. S. and Das, C. R. [2007]. A novel dimensionally-decomposed router for on-chip communication in 3D architectures, *SIGARCH Comput. Archit. News* **35**: 138–149.

Kumar, S., Jantsch, A., Soininen, J., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K. and Hemani, A. [2002]. A Network on Chip Architecture and Design Methodology, *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, pp. 105–112.

Kurian, G., Miller, J., Psota, J., Eastep, J., Liu, J., Michel, J., Kimerling, L. and Agarwal, A. [2010]. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT10)*, pp. 477–488.

Lee, C., Potkonjak, M. and Mangione-Smith, W. [1997]. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems, *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 330–335.

Lee, K., Lee, S. and Yoo, H. [2006]. Low-power Network-on-Chip for high-performance SoC design, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* **14**(2): 148–160.

Lee, R., Karig, D., Patrick McGregor, P. J. and Shi, Z. [2004]. Enlisting Hardware Architecture to Thwart Malicious Code injection, *SPC*, Vol. 2802 of *Lecture Notes in Computer Science*, Springer, pp. 237–252.

Lukovic, S. and Christianos, N. [2010a]. Enhancing Network-on-Chip components to support security of processing elements, *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS10, pp. 12:1–12:9.

Lukovic, S. and Christianos, N. [2010b]. Hierarchical multi-agent protection system for NoC based MPSoCs, *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, S&D4RCES '10, pp. 6:1–6:7.

Lukovic, S. and Fiorin, L. [2008]. An Automated Design Flow for NoC-based MPSoCs on FPGA, *Proceedings of 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP 2008)*.

Lukovic, S., Pezzino, P. and Fiorin, L. [2010]. Stack Protection Unit as a Step Towards Securing MPSoC, *Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–4.

Mao, S. and Wolf, T. [2007]. Hardware Support for Secure Processing in Embedded Systems, *Proceedings of the 44th annual Design Automation Conference*, pp. 483–488.

Marculescu, R., Hu, J. and Ogras, U. [2005]. Key research problems in NoC design: a holistic perspective, *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, pp. 69–74.

Martin, G. [2006]. Overview of the MPSoC design challenge, *Proceedings of Design Automation Conference, 43rd ACM/IEEE*, pp. 274 – 279.

McArthur, S., Davidson, E., Catterson, V., Dimeas, A., Hatziargyriou, N., Ponci, F. and Funabashi, T. [2007]. Multi-Agent Systems for Power Engineering Applications Part i: Concepts, Approaches, and Technical Challenges, *Power Systems, IEEE Transactions on* **22**: 1743–1752.

McArthur, S., Strachan, S. and Jahn, G. [2004]. The design of a multiagent transformer condition monitoring system, *Power Systems, IEEE Transactions on* **19**: 1845–1852.

Mirkovic, J. and Reiher, P. [2004]. A taxonomy of DDoS attack and DDoS defense mechanisms, *SIGCOMM Computer Communication Rev.* **34**: 39–53.

Mitchell, C. [2005]. *Institution of Electrical Engineers*, Institution of Electrical Engineers.

Mitropoulos, D., Karakoidas, V., Louridas, P. and Spinellis, D. [2011]. Countering code injection attacks: a unified approach, *Information Management and Computer Security* (Vol. 19 No. 3): 177–194.

Murgida, M., Panella, A., Rana, V., Santambrogio, M. and Sciuto, D. [2006]. Fast IP-Core Generation in a Partial Dynamic Reconfiguration Workflow, *Proceedings of VLSI-SoC'06*, pp. 74–79.

Nagata, T. and Sasaki, H. [2002]. A multi-agent approach to power system restoration, *Power Systems, IEEE Transactions on* **17**: 457–462.

Namestnikov, Y. [2011]. Kaspersky security bulletin. statistics 2011, Secureliest.

Nollet, V., Marescaux, T., Avasare, P., Verkest, D. and Mignolet, J.-Y. [2005]. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware tiles, *Proceedings of DATE'05*, pp. 234–239.

Ogras, U., Hu, J. and Marculescu, R. [2005]. Communication-centric soc design for nanoscale domain, *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 73–78.

Open Core Protocol [2000]. http://www.ocpip.org.

Palesi, M., Holsmark, R., Kumar, S. and Catania, V. [2006]. A methodology for design of application specific deadlock-free routing algorithms for NoC systems, *in CODES+ISSS 06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 142–147.

Pande, P., Grecu, C., Jones, M., Ivanov, A. and Saleh, R. [2005]. Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures, *IEEE Transactions on Computers* **54**(8): 1025–1040.

Patel, K. and Parameswaran, S. [2008]. SHIELD: A Software Hardware Design Methodology for Security and reliability of MPSoCs, *DAC '08: Proceedings of the 45th annual conference on Design automation*, New York, NY, USA, pp. 858–861.

Patel, K., Parameswaran, S. and Ragel, R. G. [2010]. Architectural Frameworks for Security and Reliability of MPSoCs, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* **99**: 1–14.

Paulauskas, N. and Garsva, E. [2006]. Computer System Attack Classification, *Electronics and Electrical Engineering* **66**(2): 84–87.

Pavlidis, V. F. and Friedman, E. G. [2007]. 3-D topologies for Networks-on-Chip, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **15**(10): 1081–1090.

Petrot, F., Greiner, A. and Gomez, P. [2006]. On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory multiprocessor soc architectures, *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO*, pp. 53–60.

Pincus, J. and Baker, B. [2004]. Beyond stack smashing: recent advances in exploiting buffer overruns, *IEEE Security and Privacy* **2**(4): 20–27.

Porquet, J., Greiner, A. and Schwarz, C. [2011]. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs, *In Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1–4.

Porquet, J., Schwarz, C. and Greiner, A. [2009]. Multi-compartment: a new architecture for secure co-hosting on SoC, *Proceedings of the 11th international conference on System-on-chip*, pp. 124–127.

Pullini, A., Angiolini, F., Bertozzi, D. and Benini, L. [2005]. Fault tolerance overhead in Network-on-Chip flow control schemes, *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pp. 224–229.

Radulescu, A., Dielissen, J., Pestana, S., Gangwal, O., Rijpkema, E., Wielage, P. and Goossens, K. [2005]. An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* **24**(1): 4–17.

Ramanauskaite, S. and Cenys, A. [2011]. Taxonomy of DoS attacks and their countermeasures, *Central European Journal of Computer Science* **1**(3): 355–366.

Rashvand, H., Salah, K., Calero, J. and Harn, L. [2010]. Distributed security for multi-agent systems - review and applications, *Information Security, IET* **4**: 188–201.

Ravi, S., Raghunathan, A., Kocher, P. and Hattangady, S. [2004]. Security in Embedded Systems: Design Challenges, *ACM Trans. Embed. Comput. Syst.* **3**: 461–491.

Richardson, R. [2003]. CSI/FBI Computer Crime and Security Survey. $http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2003.pdf$.

Rijpkema, E., Goossens, K., Radulescu, A., Dielissen, J., van Meerbergen, J., Wielage, P. and Waterlander, E. [2003]. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip, *in Proceedings of the conference on Design, Automation and Test in Europe (DATE'03*, pp. 1530–1591.

Rowen, C. and Leibson, S. [2004]. Flexible architectures for engineering successful SoCs, *Proceedings of DATE'04*, pp. 692–697.

Rushby, J. M. [1981]. Design and Verification of Secure Systems, *Proceeding SOSP '81 Proceedings of the eighth ACM symposium on Operating systems principles* (Volume 15, Issue 5): 12–21.

S., M. and Wolf, T. [2010]. Hardware Support for Secure Processing in Embedded Systems, *Design and Test of Computers, IEEE* **59**(6): 847–854.

Santi, S., Lin, B., Kocarev, L., Maggio, G., Rovatti, R. and Setti, G. [2005]. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium* pp. 2349–2352.

Schellekens, D., Tuyls, P. and Preneel, B. [2008]. Embedded trusted computing with authenticated non-volatile memory, *Trusted Computing - Challenges and Applications* **4968**: 60–74.

Seiculescu, C., Murali, S., Benini, L. and De Micheli, G. [2011]. *3D Network on Chip Topology Synthesis: Designing Custom Topologies for Chip Stacks*, Springer.

Seiculescu, C., Volos, S., Khosro, N., Falsafi, B. and De Micheli, G. [2011]. Ccnoc: On-Chip Interconnects for Cache-Coherent Manycore Server Chips, *In Proceedings of the Workshop on Energy-Efficient Design (WEED 2011)*, pp. 681–685.

Sepulveda, J., Gogniat, G., Pires, R., Chau, W. J. and Strum, M. J. [2011]. Dynamic NoC-based architecture for MPSoC security implementation, *Proceedings of the 24th symposium on Integrated circuits and systems design*, pp. 197–202.

Sepulveda, J., Pires, R., Gogniat, J., Chau, W. and Strum, M. [2012]. QoSS Hierarchical NoC-Based Architecture for MPSoC Dynamic Protection, *International Journal of Reconfigurable Computing* .

Shacham, A., Bergman, K. and Carloni, L. [2007]. The Case for Low-Power Photonic Networks on Chip, *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 132–135.

Silvano, C., Lajolo, M. and Palermo, G. [2011]. *Low Power Networks-on-Chip*, Springer.

Siripokarpirom, R. and Mayer-Lindenberg, F. [2004]. Hardware-assisted simulation and evaluation of IP cores using FPGA-based rapid prototyping boards, *Proceedings of RSP'04*, pp. 96–102.

SonicsMX SMART Interconnect Datasheet [2008]. http://www.sonicsinc.com.

Srage, J. and Azema, J. [2008]. M-Shield mobile security technology, TI White paper. $http://focus.ti.com/pdfs/wtbu/ti_m shield_w hitepaper.pdf$.

Stallings, W. [1995]. *Network and internetwork security: principles and practice*, Prentice-Hall, Inc.

Sylvester, D. and Keutzer, K. [2000]. A Global Wiring Paradigm for Deep Submicron Design, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **19**(2): 242–252.

Tamhankar, R., Murali, S., Stergiou, S., Pullini, A., Angiolini, F., Benini, L. and De Micheli, G. [1297–1310]. Timing-Error-Tolerant Network-on-Chip Design Methodology, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* (Issue 7).

Tedesco, L., Mello, A., Giacomet, L., Calazans, N. and Moraes, F. [2006]. Application Driven Traffic Modeling for NoCs, *Proceeding SBCCI '06 Proceedings of the 19th annual symposium on Integrated circuits and system design* .

Trends in IT Security Threats: Executive Summary [2007]. Computer Economics.

Trusted Platform Module (TPM) Specifications [2011]. Trusted Computing Group. https://www.trustedcomputinggroup.org/specs/TPM.

Vahid, H. [2003]. The Softening of Hardware, *IEEE Trans. on Computers* .

Verbauwhede, I. and Schaumont, P. [2007]. Design methods for security and trust, *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pp. 672–677.

Wagner, D. and Dean, D. [2001]. Intrusion detection via static analysis, *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*.

Wolf, T., Mao, S., Kumar, D., Datta, B., Burleson, W. and Gogniat, G. [2006]. Collaborative Monitors for Embedded System Security, *First Workshop on Embedded Systems Security in conjunction with EMSOFT'06*, pp. 1–8.

Wolf, W., Jerraya, A. and Martin, G. [2008]. Multiprocessor System-on-Chip (MPSoC) Technology, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **27**(10): 1701 – 1713.

Wong, H. and Sycara, K. [1999]. Adding Security and Trust to Multi-Agent Systems, *In Proceedings of Autonomous Agents Workshop on Deception, Fraud, and Trust in Agent Societies*.

Wooldridge, M. [1997]. Agent-Based Software Engineering, *Software Engineering. IEE Proceedings* **144**: 26–37.

Xiao, L., Peet, A., Lewis, P., Dashmapatra, S., Saez, C., Croitoru, M., Vicente, J., Gonzalez-Velez, H. and Ariet, M. L. [2007]. An Adaptive Security Model for Multi-agent Systems and Application to a Clinical Trials Environment, *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pp. 261–268.

Xil [2005]. *Embedded System Tools Reference Manual*. *http://www.xilinx.com/ise/embedded/est_rm.pdf*.

XPower Analyzer [2011]. *http://www.xilinx.com/support/*.

Yoon, J. Y., Concer, N., Petracca, M. and Carloni, N. [2010]. Virtual Channels vs. Multiple Physical Networks: a Comparative Analysis, *Proceedings of the 47th Design Automation Conference, DAC 2009, San Francisco, CA, USA, June 14-18, 2010. ACM 2010*, pp. 162–165.

Zhou, J. and Wu. T. snd Wu, R. [2009]. A mobile multimedia network terminal based on MPSoC, *Future Information Networks, 2009. ICFIN 2009. First International Conference on*, pp. 121–125.

Zhou, Z., Zhao, F. and Wang, J. [2011]. Agent-Based Electricity Market Simulation with Demand Response from Commercial Buildings, *Smart Gird, IEEE Transactions on* **2**: 580–588.