UNIVERSITA POLITECNICA DELLE MARCHE
FACOLTA DI INGEGNERIA

Dottorato di Ricerca in Ingegneria dell'Informazione - XVIII ciclo
Curriculum ingegneria elettronica

Protezione e sicurezza per comunicazioni on chip e Mixed-Critical Cyber-Physical Systems

Protection and safety framework for on-chip communications and Mixed-Critical Cyber-Physical Systems

Relatore:

PROF.ING. MASSIMO CONTI

Autore:

IOANNIS CHRISTOFORAKIS

Anno Accademico 2019-2020

Indice

Sommario

L'Internet of Things (IoT), una rete globale emergente di dispositivi elettronici embedded identificabili in modo univoco all'interno della rete Internet, sta trasformando il modo in cui viviamo e lavoriamo aumentando la connessione di persone e cose su una scala che un tempo era inimmaginabile. Oltre a una maggiore efficienza di comunicazione tra gli oggetti connessi, l'IoT comporta anche nuove sfide in termini di sicurezza e privacy. La specifica della sicurezza deve essere implementata in dispositive IoT che hanno il vincolo di memoria limitata, middleware vincolato, bassa potenza di calcolo e basso consumo. La sicurezza è uno degli aspetti fondamentali che differenzia l'IoT dai generici dispositivi embedded.

L'implementazione hardware su circuito integrato di diverse funzionalità di sicurezza migliora la protezione di un sistema implementando il controllo degli accessi a risorse critiche, il rilevamento di manomissioni e guasti, la protezione dei canali laterali e la protezione contro il reverse engineering e il furto di IP.

La presente tesi ha come obiettivo la definizione di una metodologia di progettazione hardware che garantisca sicurezza e protezione. Questa metodologia è stata implementata in un framework sviluppato per estendere le capacità del sistema al controllo delle minacce alla sicurezza del sistema attraverso una protezione a livello hardware. In questa tesi presentiamo il framework dell'architettura hardware, che combina la TAMMU (Translation and Allocation Memory Management Unit) utilizzata in SoC eterogenei che supportano la virtualizzazione integrata con un'architettura di protezione hardware (MSU).

Questi miglioramenti hardware si concentrano sull'isolamento dei compartimenti della memoria fisica applicando regole di accesso. Il framework, pertanto, consente l'applicazione di politiche dinamiche di sicurezza sull'hardware per la protezione da componenti hardware o software non affidabili.

D'altra parte, nelle Networks-on-Chip la velocità di iniezione del traffico è gestita principalmente impiegando tecniche complesse. Questo lavoro propone un Traffic Shaper Module che supporta sia il monitoraggio che il controllo del traffico sull'interfaccia di rete su chip o sul controller di memoria. Il vantaggio di questo Traffic Shaper Module è che garantisce una larghezza di banda di memoria alle applicazioni critiche limitando il traffico di attività non critiche.

Il sistema è sviluppato hardware Xilinx ZYNQ7000 system-on-chip, mentre le misurazioni sono state acquisite su una scheda di sviluppo Xilinx Zed-board. Abilitando il Traffic Shaper nella nostra architettura, abbiamo raggiunto il controllo della larghezza di banda con un sovraccarico trascurabile, fornendo allo stesso tempo una larghezza di banda dello 0,5-5 per cento in meno rispetto alla larghezza di banda teorica specificata.

L'architettura TAMMU proposta offre funzionalità innovative uniche che supportano più istanze di macchine virtuali (VM) simultaneamente attive con commutazione del contesto con latenza zero e abilitando i servizi di traslazione degli indirizzi per un massimo di mille domini virtuali mentre servono più dispositivi. Allo stesso tempo, il progetto proposto consente di soddisfare più richieste di traslazione di indirizzi in parallel e una in invalidazione per dominio del Translation Look-aside Buffer (TLB). L'architettura proposta è innovative rispetto allo stato dell'arte in quanto combina i servizi di enabling address translation con una larghezza di banda di memoria garantita e protesione della memoria. L'implementazione sulla piattaforma programabile FPGA Xilinx XC5VLX110T ha permesso il confronto con architetture alternative.

La tesi è organizzata nel seguente modo: lo stato dell'arte è presentato nell'introduzione.

Le caratteristiche del framework proposto sono descritte nella Sezione 2. La viosione generale del Sistema è presentato nella Sezione 3. La Sezione 4 riporta il test del sistema con la valutazione delle performances e le risorse richieste. Un confronto con le prestazioni di altre architetture è presentato nella Sezione 5.Un esempio applicativo sull'healtcare è presentato nella Sezione 6. Infine, la Sezione 7 riporta le conclusioni.

Abstract

The Internet of Things (IoT), an emerging global network of uniquely identifiable embedded computing devices within the existing Internet infrastructure, is transforming how we live and work by increasing the connectedness of people and things on a scale that was once unimaginable. In addition to increased communication efficiency between connected objects, the IoT also brings new security and privacy challenges. The security requirements for the huge base of connected embedded devices are distinct on account of their limited memory, constrained middleware, and low computing power. Security is the new differentiator for embedded and IoT devices.

At the on-chip level, several security features enhance the protection of a system by implementing access control to critical resources, by tamper and fault detection, by side-channel protection, and by protection against reverse engineering and IP theft.

The thesis targets the design of on-chip system by implementing a methodology that ensures safety and security by design. This methodology is enabled by a framework developed to extend system capabilities so as to control the concurrent effects of security threats on the system behavior focusing on hardware level protection. We present the hardware architecture Framework, that combines Translation and Allocation Memory Management Unit (TAMMU) utilized in heterogeneous SoCs that support full virtualization integrated with a hardware protection architecture (MSU). These hardware enhancements focus on isolating physical memory compartments by applying access rules; thus, we allow dynamic security policies to be enforced at the hardware for protection against untrustworthy hardware or software components.

On the other hand, Networks-on-Chip manage the traffic injection rate mainly by employing complex techniques; either back-pressure based low-control mechanisms or rate-control of traffic load (i.e. traffic shaping). This work proposes such a Traffic Shaper Module that supports both monitoring and traffic control at the on-chip network interface or the memory controller. The advantage of this Traffic Shaper Module is that proposed security framework provides guaranteed memory bandwidth to the critical applications by limiting traffic of non-critical tasks.

The system is developed in the Xilinx ZYNQ7000 System-on-Chip while the measurements were captured on a Zed-board development board. By enabling the Traffic Shaper in our architecture, we achieved ne-grain bandwidth control with negligible overhead, while providing bandwidth of only 0.5-5 percent less than the theoretical specified bandwidth

The proposed TAMMU architecture offers unique innovative features supporting multiple concurrently active virtual machine instances (VMs) with zero-latency world-context switching and enabling address translation services for up to a thousand virtual domains while serving multiple devices. At the same the proposed design allows for serving multiple address translation requests in parallel and per domain Translation Look-aside Buffer (TLB) invalidation.

Proposed architecture is innovative in relation to the state of the art as it combines enabling address translation services with the capability that the proposed security framework provides guaranteed memory bandwidth and memory protection. The combination of these two complex features is not supported in previous systems. We prove that despite the increased need for hardware, our design manages to keep resource utilization at least at the same level as other known technologies implemented in modern systems. Significant differentiation, favorable to our architecture, is also achieved in performance compared to the state of the art. The need for comparisons with alternative architectures made it necessary to integrate our system into the Xilinx XC5VLX110T FPGA platform as well.

The thesis is organized as follows. An overview of state of the art is given in Introduction section. The techniques that our framework include and its features are described in Section 2, followed by full system overview in Section 3. In Section 4, the testing of Framework and the performance and resource requirements are discussed. In Section 5, comparison with the State of the Art presented. A healthcare example is given in Section 6. Finally, Section 7 concludes the thesis.

1 Introduction

Internet-of-Things (IoT) products like smart home controllers, wearables and smart metering devices bring many benefits such as improved convenience, better health or environmental savings. However, the internet connectivity and massive data collection that enable these benefits also bring threats to the reliable functioning of these products and to the privacy of their users.

When architecting such a system, trade-offs have to be made. This is especially true with security, as there is no one magic solution that makes a system “secure”. The trade-offs to be made are complex, interdependent, and span multiple disciplines. Hardware versus software, throughput versus area and energy consumption, and security level versus cost are some of the decisions to be made. But although the magic IP block that protects a full system does not exist, securing an IoT system is made significantly easier by using building blocks that were designed with secure systems in mind. IoT systems security should always be considered end-to-end. But since there is a significant data size, speed and power variation across the different devices in the chain, as well as different security threats, technical solutions for the groups of devices differ as well. Connected IoT devices may be accessed or controlled by malicious network nodes. The authors in [1] demonstrated how it is possible for an external party to gain control over every connected device within a ZigBee network by taking advantage of security flaws in ZigBee standard, one of the most popular wireless communication standards used by IoT devices. Attackers may compromise IoT devices and build a botnet to launch cyber-attacks, including sending spam, spreading viruses and worms, and running denial-of-service attacks. In 2014, Proofpoint uncovered the first proven IoT-based cyber-attack, which involved more than 750,000 phishing and spam emails launched from more than 100,000 compromised IoT devices, including home-networking routers, connected multimedia centers, televisions and at least one refrigerator [2]. IoT devices infected with malware may disclose sensitive data (e.g., contactless payment information) to adversaries.

Communication between IoT devices or between IoT device and trust center (i.e., the core device who is responsible for joining activities within a local area network) may suffer from eavesdropping. Data captured by sensors connected to IoT devices may be altered maliciously during communication. Credentials of IoT devices may be stolen by hackers to perform further cyber-attacks [3]. Credentials assigned to one IoT device may be replayed by another device after its lifetime. On the other hand, user privacy of IoT devices is also at risk especially when current smart devices could collect much private information such as blood pressure and heart rate.

A lot of solutions have been proposed to defend IoT devices against cyber threats. By optimizing communication standards, improving device security configuration, upgrading firmware, setting strong passwords, installing patches, etc., the vast majority of cyber-attacks can be prevented. Symmetric encryption algorithms such as Advanced Encryption Standard (AES) [4] are widely adopted to prevent eavesdropping on the communication between two resources constrained network devices. The keyed-hash message authentication code (HMAC) [5] can be used to simultaneously verify both the data integrity and the authentication of a message. To prevent credentials of one dead

IoT device from being reused by another device, credentials need to be tied to lifetime. To preserve user privacy in a participatory sensing network, a Hot-Potato-Privacy-Protection algorithm (HP3), in which data is delivered to the next hop until some user-defined threshold is reached before being uploaded to the server, has been proposed in [6]. To limit access to information content, an access control technique implemented within a network device was presented in the patent [7]. It determines whether to forward client requests for processing by comparing client source information against a database of Uniform Resource Locators (URLs), IP addresses, or other resource identification data.

However, hardware threats are rarely touched which are also critical to IoT security. The IoT devices may contain untrusted components. For example, counterfeit integrated circuits (ICs) (e.g., recycled or remarked ICs) or ICs containing hardware Trojans may have been mounted on the PCBs of IoT devices intentionally or unintentionally by the system integrators before they enter the supply chain. Authentic IoT devices may be mixed with clones or fakes during their distribution by untrusted supply chain partners. IoT devices may be lost or stolen during distribution or even after deployment. IoT devices may even be physically tampered by rogue elements who may have access to them after being provisioned.

To detect hardware Trojans contained in the ICs, a series of techniques have been developed using side-channel signal analysis, functional test, etc [8].

Different from Internet, the IoT has its own unique properties that increase difficulty to security and privacy maintenance. The properties of IoT are summarized as follows:

Heterogeneity: The IoT exhibits much higher level of heterogeneity than Internet, as objects with totally different functionality and originated from various technology and application fields will belong to the same communication environment. IoT device types range from small RFID tags with limited processing power to large connected servers running so-phisticated operating systems. Hence, corresponding security and privacy measures should be interface-friendly and compatible with various types of IoT hardware.

Specificity: Vast majority of current IoT devices (e.g., SmartBand) are designed for a particular use and could collect sensitive personal information (e.g., blood pressure, heart rate, living habit, etc.), in which case how to effectively protect user privacy will be a big concern. In addition to consumer electronics, IoT devices are more and more used in industrial and agricultural automation. For example, IP surveillance cameras are widely used to monitor asset status in the inventories. Compromised IoT devices could disclose significant trade secrets.

Resource Constrained: Most IoT devices are low-cost hardware with constrained resources in terms of computing, communication, and storage capabilities, which requires corresponding security and privacy measures to be lightweight and cost-effective. For example, passive RFID tags used to track and trace commodities in the supply chain are usually equipped with simple read/write operations, XORing with random numbers, and cyclic redundancy check (CRC) capabilities. Wireless sensor network (WSN) sensors are usually equipped with low-cost microcontrollers with small bit width.

Wireless: A large number of IoT devices are equipped with wireless communication modules (e.g., WiFi, Bluetooth, ZigBee, etc.) and have the capability to communicate with neighboring devices or network nodes through the air channel. As a result, malicious readers or network nodes could easily intercept those packets being communicated between IoT devices or between IoT device and trust center without being noticed. Here, we refer to the core device (e.g., the core router in the home security network) who is responsible for joining activities within a local area network as the trust center.

Infectivity: Since most of the time IoT devices are connected to the network and usually share the same network key or group key within a seemingly trusted area (e.g., theme parks, music concerts, sports games, etc.), if one device is compromised, the adversary could easily hack its neighboring devices with the deciphered network/group key. For example, IoT devices within the same ZigBee network will encrypt packets using the shared network key after authenticating themselves to the trust center with their link keys [59].

Mobility: Many IoT devices (e.g., smart phones) are mobile and would move together with their users. As a result, their communication neighborhood will be transformed a periodically. Dynamic communication neighborhood will be a challenge to authentication methods based on fixed IP addresses or interaction with neighboring devices.

Scalability: The number of IoT devices on the earth have been growing exponentially. The management of such a huge number of IoT devices will be a big challenge. Furthermore, the number of IoT device types is also on the rise, which raises a new challenge to device authentication.

1.1 Characteristics and foibles of embedded and IoT devices

Many of the inherent characteristics of embedded and IoT devices have direct impact on security-related issues. We discuss some of their implications on vulnerabilities in embedded systems, having regard to the categorization made in [73]:

1.1.2 Characteristics

Embedded devices are used for special purpose applications where conventional workstation computers cannot cope with functionality, cost, power requirements, size, or weight. The specificity of the embedded device often comes with one or more disadvantages of the following type:

• Limited processing power usually indicates for an embedded system that it cannot have applications used to attack conventional computer systems (eg virus detector, intrusion detection system).

• Limited power available is one of the main limitations of embedded systems. Many of these systems operate on batteries and increased power consumption reduces system life. Therefore, the embedded system can provide limited energy resources to provide system security.

• Physical exposure is typical of embedded systems that are developed without direct control by the owner or operator (eg public location, client requirement). Thus, embedded systems are vulnerable to attacks that exploit the attacker's physical proximity.

• Remote operation and unmanned operation are necessary for embedded systems deployed in inaccessible locations (eg hard environment). This limitation means that deploying updates and paches as with conventional workstations is difficult and needs to be automated. These automated mechanisms provide potential targets for attacks.

• Network connectivity through wireless or wired access is increasingly common for embedded systems. This access is required for remote control, data collection, updates. In cases where the embedded system is connected to the Internet, vulnerabilities can be remotely exploited from anywhere.

These characteristics lead to a unique set of vulnerabilities that need to be considered in embedded systems.

1.1.3 Foibles

Embedded and IoT device are vulnerable to a range of abuses that can aim at stealing private information, draining the power supply, destroying the system, or hijacking the system for other than its intended purpose. Examples of the foibles in embedded devices as presented in [73]:

• **“Energy drainage (exhaustion attack)”:** In embedded systems, the limited battery power makes it vulnerable to attacks that discharge this resource. The best possible energy consumption can be achieved by increasing the computational load, reducing sleep cycles or increasing the use of sensors or other peripherals.

• **“Physical intrusion (tampering)”:** The proximity of embedded systems to a potential intruder makes them vulnerable to attacks when physical access to the system is necessary. Examples are power analysis attacks or snooping attacks on the system bus.

• **“Network intrusion (malware attack)”:** Web-based systems are vulnerable to the same type of remote exploits common to workstations and servers, such as buffer overflow attacks.

• **“Information theft (privacy)”:** Data stored in an embedded system is usually vulnerable to unauthorized access, as the embedded system can be deployed in a hostile environment. An example of data to be protected is cryptographic keys or electronic currency on smart cards.

• **“Introduction of forged information (authenticity)”:** Malicious input of incorrect data (either by system sensors or by direct memory recording) also makes the embedded systems vulnerable. Examples are incorrect video feeds to security cameras or replacement of metering data to an electricity meter.

• **“Confusing/damaging of sensor or other peripherals”:** As with malicious data entry, embedded systems are vulnerable to attacks that cause improper sensor or peripheral operation. An example is the violation of the calibration of a sensor.

• **“Thermal event (thermal virus or cooling system failure)”:** Embedded systems must be capable of operating in smooth environmental conditions. Due to the highly exposed operating environment of embedded systems, there is a potential vulnerability to attacks that overheat the system.

• **“Reprogramming of systems for other purposes (stealing)”:** Embedded systems are general-purpose systems, but are often intended to be used for a specific purpose. These systems are vulnerable to unauthorized reprogramming for other uses. One example is the game console rescheduling to run Linux.

In order to defend embedded and IoT devices from these attacks, it is necessary to consider different types of attacks and countermeasures in more detail.

1.2 Attacks and threats on Embedded and IoT devices

IoT is not a single system or type of systems and it is therefore impossible to provide a comprehensive list of potential threats that apply to all. The threats to a smart utility metering system are typically different from the threats to a wearable consumer or healthcare device, which differ again from the threats to a sensor network for environmental monitoring or to an Internet connected refrigerator that autonomously orders groceries to keep stocked.

From a high-level perspective, the IoT threats are about privacy, information theft, (device) identity theft or impersonation, device cloning or counterfeiting and denial-of-service. A good security practice is to not re-invent the wheel, but to build as much as possible on well-known, proven solutions. Since IoT systems combine aspects from multiple existing domains, threat models from these domains provide a good starting point. One domain to look at is traditional Internet connected devices; another domain is bank cards and payment terminals, a domain where security requirements and practices are considered highly evolved. Typical security requirements for IoT devices will fall between these two domains, though the trend is that bank card grade security is rippling through to IoT devices as well.

1.2.1 Software attacks

For today's IoT systems, most security breaches reported are on software/ end-to-end system level and often due to negligence. For example, 70% of the devices investigated in a recent study [9] used unencrypted network services. Attacks that could exploit such security issues include eavesdropping and man-in-the-middle attacks that intercept and modify the data being communicated.

Other examples of attacks are software buffer or stack overflow attacks that feed a device with out-of-spec inputs like the OpenSSL Heartbleed bug. These types of attacks can leak secure information or could enable running of malicious applications and rooting, obtaining higher level privileges than a user normally would be entitled to.

1.2.2 Side Channel attacks

Physical attacks can be split into two categories: invasive and non-invasive. Non-invasive attacks use regular interfaces of a device like USB ports, or relatively easily accessible JTAG debug interfaces that provide access to and control of the IoT device processor and memories.

Another type of non-invasive attacks are side-channel attacks. This exploit information leaked from a device to reconstruct protected secrets by measuring variations in power consumption of a device or variations in the electromagnetic radiation to reconstruct the computations performed and to extract private keys from these. Invasive attacks go a step further by opening up the IoT system and potentially altering it physically. This can be at board level e.g. by tapping into the bus interface between processor and external memory, but also at chip level. Depending on the budget of the attacker, attacks can go as far as de-capping an IC package, removing top layers and then inspecting structures, monitoring signals (by attaching microprobes), or even altering structures with a Focused Ion Beam (FIB) device.

Given the diversity of all of the potential security threats described above, it is important to perform a specific threat analysis for a specific type of IoT device. In addition to the technical side of the attacks, the likelihood and potential impact of different attacks should also be estimated. Although privacy is at stake when data is leaked from a wearable fitness band, successfully hacking a smart utility meter is likely a more profitable activity and consequently the smart meter would require better protection mechanisms. Good security analysis enables trading off security versus cost, performance, and energy when selecting the potential countermeasures that are described in the next section.

1.2.3 Countermeasures against software attacks

There are several countermeasures proposed in the literature to defend against code injection attacks performed by exploiting common implementation vulnerabilities. These can be divided into nine groups based on: the system component where the proposed countermeasure is implemented; and the techniques used for the countermeasures.

As return addresses of functions are the most attacked target of buffer overflows, there are many hardware/architectures assisted countermeasures that aim to protect these addresses. Some of these techniques are described in [10, 11, 12]. Another technique to counter code injection attack is to ensure code integrity at runtime. The authors in [13] have proposed a microarchitectural technique to ensure program code integrity at runtime and thereby preventing code injection attacks. An embedded monitoring system to check correct program execution is proposed in [14].

Safe languages such as Java and ML are capable of preventing some of the implementation vulnerabilities discussed here. However, everyday programmers are using C and C++ to implement more and more low- and high-level applications and therefore the need for safe implementation of these languages exists. Safe dialects of C and C++ use techniques such as restriction in memory management to prevent any implementation errors. Examples of such methods are shown in [15, 16].

Static Code Analyzers, analyze software without actually executing programs built from that software [19]. In most cases the analysis is performed on the source code and in the other cases on some form of the object code. The quality of the analysis performed by these tools ranges from those that only consider the behavior of simple statements and declarations, to those that include the complete source code of a program in their analysis. The information collected by these analyzers can be used in a range of applications, starting from detecting coding errors to formal methods that mathematically prove program properties. Examples of static code analyzers are shown in [17, 18].

In dynamic code analysis, the source code is instrumented at compile time and then test runs are performed to detect vulnerabilities. Even though performing dynamic code analysis is more accurate than static analysis (more information of the execution is available at runtime compared to compile-time), dynamic code checking might miss some errors as they may not fall on the execution path while being analyzed. Some well-known dynamic code analyzers are shown in [20].

Behavior-based anomaly detection compares a profile of all allowed application behavior to actual behavior of the application. Any deviation from the profile will raise a flag as a potential security attack [20]. This model is a positive security model as this model seeks only to identify all previously known good behaviors and decides that everything else is bad. Behavior anomaly detection has the potential to detect several type of attacks, which includes unknown and new attacks on an application code. Most of the time, the execution of system calls is monitored and is recorded as an anomaly if it does not correspond to one of the previously gathered patterns. A threshold value for the number of anomalies is decided a priori and when the threshold is reached, the anomaly can be reported to the system and subsequent action, such as terminating the program or declining a system call can be taken. On the negative side, behavior anomaly detection can lead to a high rate of false positives. For instance, if some changes are made to the application after a behavior profile is create behavior-based anomaly detection will wrongly identify access to these changes as potential attacks. Some examples of this technique are described in [21, 22].

Sandboxing is a popular method for developing confined execution environments based on the principle of least privilege, which could be used to run untrusted programs. A sandbox limits or reduces the level of access its applications have to the system. Sandboxes have been of interest to systems researchers for a long time. Butler Lampson, in his 1971 paper [25], proposed a conceptual model highlighting properties of several existing protection and access-control enforcement mechanisms. Other examples are given in [23, 24].

Compilers play a vital role in enabling the programs written via language specifications to run on hardware. The compiler is the most convenient place to insert a variety of solutions and countermeasures without changing the languages in which vulnerable programs are written. The observation that most of the security exploits are buffer overflows and are caused by stack-based buffers, has made researchers propose stack-frame protection mechanisms. Protection of stack-frames is a countermeasure against stack-based buffer overflow attacks, where often the return address in the stack-frame is protected and some mechanisms are proposed to protect other useful information such as frame pointers. Another commonly proposed countermeasure is to protect program pointers in the code. This is a countermeasure which is motivated by the fact that all code injection attacks need code pointers to be changed to point to the injected code. Since buffer overflows are caused by writing data which is over the capacity of the buffers, it is possible to check the boundaries of the buffers when the data is written to prevent buffer overflow attacks. Solutions proposed as compiler support for bounds checking are also discussed in this section. Some examples of the techniques are given in [26**].**

Safe library functions attempt to prevent vulnerabilities by proposing new string manipulation functions which are less vulnerable or invulnerable to exploitations. In [27] the authors propose alternative string handling functions to the existing functions which assume strings are always NULL terminated. The new proposed functions also accept a size parameter apart from the strings themselves. In [28], another safe string library is proposed as a replacement to the existing string library functions in C.

Operating system based solutions, use the observation that most attackers wish to execute their own code and have proposed solutions preventing the execution of such injected code. Most of the existing operating systems split the process memory into at least two segments, code and data. Marking the code segment read-only and the data segment non-executable will make it harder for an attacker to inject code into a running application and execute it [29].

1.2.4 Countermeasures against side channel attacks

There are several countermeasures against side channel attacks. These have been divided into six categories:

To mask code execution and to confuse an adversary, noise can be injected during code execution. Examples of masking techniques are presented in [30]. Substitution Boxes (SBOXes), often used in cryptology can also be masked in the execution. Some examples for the SBOX masking techniques are presented in [31, 32].

A window method can be applied in Public Key Cryptosystems to prevent power analysis based side channel attacks. In the window method, a modular exponentiation can be carried out by dividing the exponent into certain sizes of windows, and performing the exponentiation in iterations per window by randomly choosing the window [33].

Dummy instructions can be placed to provide random delays. This confuses the adversary when attempting to correlate the source implementation with the power profile. Chari et al. [34] claimed that countermeasures involving random delays (i.e., dummy instructions used to provide random delays in an execution) should be performed extensively, otherwise they can be undone and re-ordered, causing a successful attack. Several dummy instruction approaches are presented in [35].

Public Key Cryptosystems like RSA and ECC have been severely attacked using Simple Power Analysis (SPA), mainly because of the conditional branching in the encryption. Such vulnerabilities in the program can be prevented by modifying the implementation or replacing with a better new algorithm to perform the same task. Key code modification techniques to prevent power analysis are explained in [34].

The software code can be modified in such a way that complementary events are coded to negate the effects of the actual computations. Examples of such code balancing techniques are presented in [21]. Evidently, balancing at the gate level is the most appropriate solution to prevent power analysis, since the power is consumed/dissipated depending on the switching activities in gates. Hardware balancing is primarily performed by placing two gates in parallel, one complements the other when switching. Various hardware balancing techniques are given in [37].

Some of the other techniques include **signal suppression circuits**, which can be used to reduce the Signal-to-Noise Ratio (SNR) to prevent the adversary from differentiating the power profile. Examples for the suppression circuits are given in [38, 39]. Software level current balancing approaches are performed by modifying the source and inserting naps to keep the current constant [40].

May et al. [41] proposed **a non-deterministic processor design**, where the independent instructions are identified and executed out-of-order in a random choice by the processor. This infringes the conventional attack rule removing the correlation between multiple executions of the same program, thus preventing the adversary from comparing different runs for power analysis. Several other improved versions of the non-deterministic processor architecture are proposed in [42].

Randomizing the clock signal [43] for the secure processor to confuse the adversary is another countermeasure proposed to prevent power analysis. This prevents the adversary from analyzing the clock signals to identify certain significant instruction executions in the power profile. More examples on handling the clock signal to prevent power analysis are presented in [36, 44, 45].

Power analysis can also be prevented by designing special instructions whose power signature is difficult to analyze [32] or whose power consumption is data independent [46]. Several examples of creating extensible instructions are given in [47, 48]. Such extensible instruction designs can also be adapted to prevent power analysis attacks.

1.3 Protection mechanisms of embedded and IoT devices

This part of the thesis details a number of the IoT security countermeasures from the previous section and describes some key architectural decisions to be made. The topics are structured according to the architecture levels of a product: system, processor, and software, and architectural examples and mechanisms for each level. The overview we present in this work is based on corresponding works such as [74] and [75].

1.3.1 System-level protection mechanisms

At the system level, the main decisions to be made are what to implement in hardware and what in software, and how to guarantee strict separation of secure system resources, software and other secure information so non-authorized entities do not have access.

A first hardware-software tradeoff is the implementation of the foundation functions like cryptography algorithms. These can be implemented fully in software, use hardware to accelerate software, be fully implemented in hardware, or use a combination of those options. Depending on system requirements on throughput and latency on the one hand, and silicon area and power efficiency on the other hand, a choice can be made between different options.

1.3.1.1 Anti-Tamper Protection

Tamper [49] refers to intentional alteration or manipulation to the system such that it compromises the secrets in the system or enables unauthorized operation of the system. Designing systems that are absolutely tamper proof is often not possible due to the increased cost involved to implement countermeasures against various known and potentially unknown attacks (for example, due to constantly improving technology that increases adversaries' capabilities to carry out an attack, either in terms of time taken or reduced cost to perform a successful attack). The goal of tamper mechanisms is to prevent any attempt by an attacker to perform an unauthorized physical or electronic action against the device. Tamper mechanisms are divided into four groups: Resistance, Evidence, Detection, and Response Tamper mechanisms are most effectively used in layers to prevent access to any critical components. They are the primary facet of physical security for embedded systems and must be properly implemented to be successful. From the designer's perspective, the costs of a successful attack should outweigh the potential rewards.

Often, existing tamper mechanisms can only be discovered by attempted or complete disassembly of the target product. This may require an attacker to obtain more than one device in order to sacrifice one for the sole purpose of discovering such mechanisms. Once the mechanisms are noted, an adversary can form hypotheses about how to attack and bypass them.

1.3.1.2 Hardware Monitors

The hardware monitors used in the work are based on [50] and are illustrated in the figure 1. The hardware monitors the processor kernel operation for each command to compare the actual behavior with the one you expect. The graph contains all possible control flows between the basic blocks and a short hash of each binary command. The use of a clean version of the binary command (instead of the full binary command) is necessary to reduce the size of the tracking graph to a fraction of the binary processing.

During processing, the core processor indicates the hash value of the current function. The display compares the hash value with that stored on the hardware screen. If the hashing match continues, the comparison continues for the next command.

Assumes the following functions to be valid (since the monitor does not have a data path to calculate the option that is valid). If the processor attacks and starts executing code other than the original binary (eg after a stack crash attack), the hash values reported by the processor kernel differ from the hash values on the monitor, and the attack can to be detected. In a network based on Internet Protocol (IP), recovery from an attack can be easily accomplished by removing the attack packet, resetting the processing stack, and continuing processing the next packet.

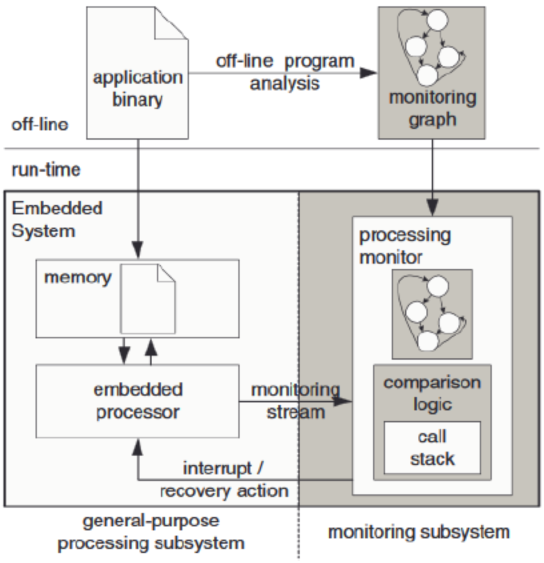

Figure 1: System Architecture for secure embedded processing as presented in [74]

1.3.1.3 TrustLite Architecture

TrustLite security architecture presented in [50] for providing trusted computing functionality on low-cost embedded systems. Their design represents a new alternative for isolating secure applications, providing trusted execution, OS interoperability and secure peripheral access.

This architecture is enabled by a generalized memory protection scheme, combining an executionaware memory protection unit (EA-MPU) with a secure exception engine that protects the task state from untrusted exception handlers. Secure Loader of presented architecture provides a simple yet flexible establishment and update of the platform's security policy, prevents memory leakage after platform reset and can be extended to act as a root of trust for remote attestation and trusted execution. Depending on the application scenario and cost constraints, TrustLite can be instantiated in several configurations, from providing a single atomic firmware security service to isolating users pace tasks in a preemptive multi-tasking environment.

1.3.2 Processor level protection mechanisms

At the processor level, several security features enhance the protection of a system by implementing access control to critical resources, by tamper and fault detection, by side-channel protection, and by protection against reverse engineering and IP theft. Architecture trade-off decisions for these features are partly determined by the system-level choices — for example combining trusted and normal code on a single processor or using a separate secure processor — and partly determined by security versus performance and area requirements — for example applying memory and pipeline encryption or not.

For controlling access to critical or security-sensitive processor and system resources, a processor can support multiple execution privilege levels and a memory protection unit as introduced in the previous section. The privilege levels control access to certain CPU control registers by only allowing access from higher levels. In combination with software, two privilege levels are enough for building secure systems. When a processor supports more than two privilege levels, more efficient, fine-grain differentiation in access control is possible. The MPU controls access to memory and memory mapped peripherals by checking and enforcing the access attributes that are defined for specific address regions. The combination of both protection mechanisms can be used to securely shield trusted software from non-trusted software.

1.3.2.1 Multi-compartment Architecture

In [52] authors presented a proposal for securely co-hosting several protection domains (compartments), called multi- compartment. By representing logical entities at the hardware level, the protection policy turns out to be much more flexible, lightweight and coherent with respect to embedded systems aspects, in particular heterogeneous multi-processing. Compared with other approaches, compartments are able to run in physical address space and enjoy direct access to security-critical initiator devices, such as DMA devices, while remaining protected from one another.

1.3.2.2 AEGIS secure processor

Edward Su et al. [53] proposed the AEGIS processor architecture that can be used to build a secure computing system. Physical random functions are used to reliably create, protect, and share secrets within processing architecture. The processor also provides four modes of operation which enable new applications for secure computing. The suspended secure mode of execution is a new contribution which allows the trust base of an application to be reduced, and improves performance. They have also shown a programming model which allows programmers to use our secure functionality with high level languages. AEGIS processor has been implemented on an FPGA, and they have shown that the overhead for secure computing is reasonable.

1.3.2.3 M-MAP: Multi-Factor Memory Authentication for Embedded Processors

Spatio-temporal vulnerabilities and memory integrity attacks pose serious challenges in designing secure embedded systems. Conventional memory safety schemes provide protection against either spatio-temporal vulnerabilities or memory integrity attacks. Solutions that provide complete memory safety guarantees come with substantial architectural modifications and overheads, making them infeasible for embedded systems. In [54] paper authors examine key theoretical and practical implications of implementing the conventional protections in a comprehensive secure processor design. Based on these implications, they proposed a holistic memory authentication framework, called M-MAP, for complete memory safety. M-MAP implements hardware-based memory integrity verification along with software-based bounds checking in order to keep a balance between hardware modifications and performance. They proposed to implement M-MAP on top of a lightweight out-of-order processor which delivers complete memory safety with a modest overhead of 32% on average. Authors argue that this enables a low-cost solution geared towards secure embedded devices.

1.3.3 Software-level mechanisms

Some of the protection mechanisms at the software level have already been discussed or introduced in the previous sections, like MPU protection. In this section a comprehensive list of software technologies that complement the hardware mechanisms is presented, as well as a more detailed description of the ones not covered so far. The 'software' is interpreted in a broad sense, including not only run-time software that is part of a final product, but also a number of software tools that enhance the security of IoT devices.

1.3.3.1 Embedded Encryption

The confidentiality and integrity of sensitive information is to a large extent implemented by the use of symmetric key algorithms, such as the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) [54].

Unfortunately, many web-based systems do not have reliable encryption to protect sensitive information. This may be due to various constraints such as resources, costs or planning. Extending an old system to an open network such as Ethernet could also create security gaps in the system.

Whatever the reason, the lack of strong encryption can lead to catastrophic consequences. Intruders or malicious administrators can read, watch, modify or remove communications. If proprietary wireless

RF interfaces are involved, the risk is further enhanced. Anyone with the right equipment can attack the system, probably from a considerable distance on a high-performance antenna.

Inadequate cryptographic protection can lead to compromises, many of which are not obvious when designing the system. A careful designer should examine the effects of deleted, modified, and false information from all elements of a networked system and take appropriate measures to protect the system from such attacks.

The current embedded operating devices provide support for various networking protocols and wireless security - WEP, WPA, and WPA2. The algorithms incorporated are particularly optimized for operations under resource- constrained environments of the embedded systems.

1.3.3.2 Elliptic Curve Cryptography

The National Institute of Standards and Technology (NIST), USA [58] has published its forecasts of adequate security for the next thirty. These recommendations are based on AES-128-bit symmetric security and on predicting the microprocessor's ability to break asymmetric encryption.

As the stronger symmetric algorithms, such as triple DES (3-DES) and AES, become more popular, the corresponding asymmetric encryption mechanisms are inadequate. Many systems use 128-bit AES or 256-bit AES for symmetric encryption that requires RSA public key sizes, however, these systems rely solely on asymmetric 1024-bit encryption. Thus, these systems achieve a level of security almost equivalent to a system that uses 80-bit keys for symmetric encryption. While 80-bits provide substantial security, systems incur the cost of 128 or 256-bit keys, without this level of security. A typical 1024-bit RSA asymmetric key is as secure as an 80-bit symmetric key, however, AES key sizes range from 128 bits to 256 bits. In order to provide security that is equivalent to that provided by the AES, the RSA public key sizes would be too large to maintain a standard embedded material while ensuring reasonable performance levels.

One extremely controversial solution to the problem of key size inequality is the family of asymmetric algorithms known as Elliptic Curve Cryptography (ECC). To provide the same level of security, the ECC uses much smaller key sizes and has the capability of providing higher levels of security compared to asymmetric techniques. For larger key sizes, the benefits are more significant: a 256-bit compact key must be protected by an asymmetric RSA or 15,000-bit DH key, while an asymmetric ECC key size of just 512 bits provides equivalent security. The reduced ECC key size results in significant cost savings.

Using a smaller key size allows the design of more compact applications. This is related to faster cryptographic operations that run on smaller brands or more compact software. This results in reduced heat output and reduced power consumption - which are particularly advantageous for limited resource systems.

As ECC is an emerging cryptographic technique, embedded implementations of ECC are now being designed and incorporated into systems. While several standard security protocol implementations do support ECC; RSA, is still more widely deployed. However, this will change as ECC gains momentum following its standardization.

1.3.3.3 Trusted Computing

It is obvious that there is a need for a more secure computing environment across multiple platforms, peripherals and devices, without compromising user integrity, user rights. The Trusted Computing Team [55] aims to establish a methodology and define open standards that can create a reliable and secure computing environment. This led to the implementation of the Trusted Platform Module (TPM). TPM is a standalone secure processor, located separately from the host CPU and handles digital certificate verification, storage and management. Its basic function is to control the loading of all software from the boot level. Thus, when fully implemented, all executable and software data must be digitally signed and verified by the TPM prior to loading and further processing on the central CPU.

TPM will be at the forefront of developing next-generation embedded systems. Taking into account the costs associated with defining and maintaining the necessary certificates, it is even more important to exploit them. Validation issues will also play a vital role, as political decisions will be taken regarding the implementation of a trust root or public key infrastructure. OEMs, content providers, and software developers should make prudent decisions about security policy. They will need to evaluate the compatibility of their products with applications where security policies are in place.

1.3.3.4 Authentication Techniques

To ensure the target runs only authorized software, firmware needs to be authenticated. This process – a digital signature – verifies that a piece of software is genuine and approved. The software loaded during manufacturing must be signed digitally and this process should apply to each firmware update. A digital signature enables trust during the device's lifetime.

A strong digital signature must be computed by a public and well proven cryptographic algorithm. If system firmware is authenticated using an elliptic curve digital signature algorithm (ECDSA) and RSA, both combined with SHA, users can have a high level of trust.

Organizations are looking for more secure authentication methods for data access, physical access, and other security applications. The use of biometrics in identification management is drawing attention across markets, even as organizations and individuals demand more reliable, highly accurate and efficient methods of confirming a person's identity.

IBM has started trial runs of a device that could ensure new levels of security to on-line banking. Named the zone trusted information channel (ZTIC) [56], the prototype device resembles a memory stick with an integrated display. The technology effectively moves all the cryptographic and critical user-interface processes away from a consumer's PC onto the ZTIC device, creating a trusted communication endpoint between the banking server and the user.

1.3.3.5 Secure Boot

The purpose of Secure Boot [57] is to bring the system to a known and trusted state. Because CPU software image resides as plaintext on external flash, an attacker can easily modify the code and cause the CPU to run code which was not intended to run.

Secure Boot, works by calculating a hash value of the software image. The hash value is then signed using the software manufacturer private key, to create a unique signature which an attacker cannot forge without access to the private key. The unique signature is then loaded to the flash alongside the software image.

The software manufacturer's public key is also provisioned to the system. The Secure Boot routine is a ROM-based routine which performs the following steps at boot time: reads the software image from flash, calculates the hash value of the software image and uses the software manufacturer's public key to verify that the new hash just calculated matches the old hash present on flash and signed by the software manufacturer's private key. Even a single bit modification of the software image will lead to a different hash and as a result will lead to Secure Boot failure.

The Secure Boot function must be ROM-based, so that an attacker cannot intercept the procedure. Additional features are required in order to provide a complete Secure Boot solution. These include the ability for software update at any point in time. Given that the software is correctly signed with the software manufacturer private key, the Secure Boot routine should pass successfully. If the software image is constructed of several components, the Secure Boot routine should be able to accommodate multiple software vendors and multiple keys used to sign the image. If a private key has been compromised, the Secure Boot routine should support revocation of a compromised private key and discard the appropriate public key. Finally, a Software Version Revocation mechanism is able to advance the system to a new version of the software image and prevent roll-back to an older version.

2 Techniques of security Architectures

Initially, we will present the techniques we implemented in hardware to ensure a truly secure and safe system. The combination of these techniques creates a protection and safety framework for on-chip communications that we implemented and presented on [76].

2.1 Traffic Monitoring Architecture

We target embedded devices that can host applications with different requirements in terms of bandwidth either of system memory or of other on-chip components. We assume a Network-on-Chip viewed as an interconnect intellectual property module with conventional read and write semantics; the network interface offers read/write and block transfers. Our architecture is based on ZYNQ System-on- Chip of Xilinx. We utilize both the Processing System that consists of the dual ARM Cortex-A9 CPU [60], as well as the Programmable Logic (PL) area, wherein we integrate the Traffic Shaper Module (TSM). More precisely, our architecture involves the following components and attributes:

- It utilizes the Processing System (PS) Direct Memory Access (DMA) controller that is used for burst transactions.
- The CPU routes accesses to the memory through the GP port to the PS DDR3 memory. A central interconnect is located within the PS that comprises multiple switches in order to connect to the system resources by using the AXI point-to-point channels for communicating addresses, data, and response transactions between master and slave ports [60]. This ARM AMBA 3.0 interconnect implements a full array of the interconnect communications capabilities and overlays for QoS, debug, and test monitoring. The interconnect manages multiple outstanding transactions and is architected for low-latency paths for the ARM CPUs and for the PL master controllers. The accesses from the PS DMA cross only one GP Port and HP port (HP0), as shown in Figure 2.
- The device integrates a Microblaze soft-processor [61] that is used to monitor resources that are used by the main processor (A9 CPU) and to collect information from the AXI performance monitor. Even though monitoring and controlling of system resources, i.e. NoC and memory bandwidth must be implemented with customized hardware to offer fine-grain control and response times, a software-oriented approach (using MicroBlaze) offers a coarser-grain solution but more flexible.

The device uses a custom Address Remap Block (ARB), to remap addresses that arrive through the PS and redirects them to the DDR. Custom circuitry is integrated at RTL level, which provides the ability of monitoring, control and supplying guaranteed bandwidth to secure the availability for critical applications.

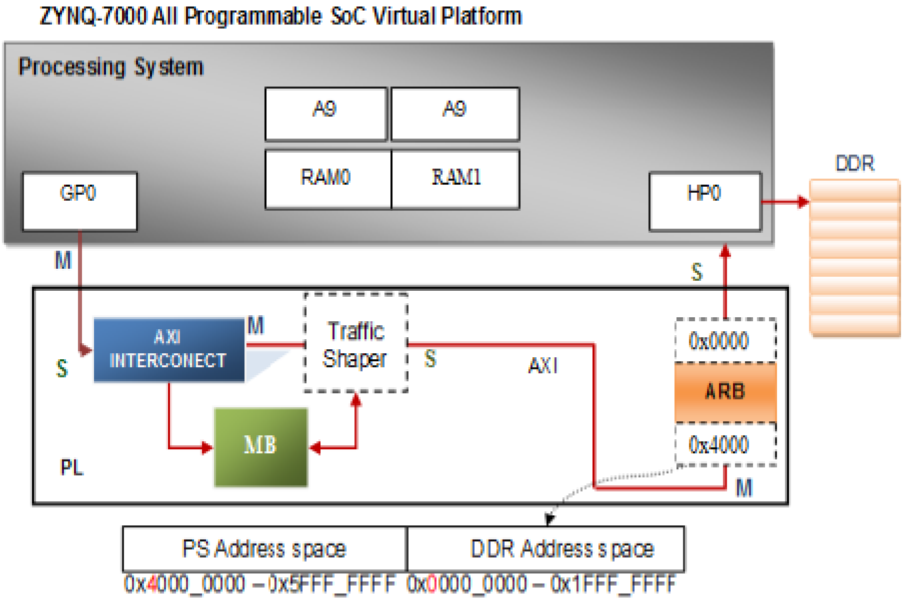

Figure 2: Architecture layout supporting precise bandwidth shaping through the use of the hardware Traffic Shaper and embedded microcontroller manager in a dedicated MicroBlaze soft-processor (MB); memory traffic flows through the General Propose (GP) Port to the DDR3 memory while Traffic Shaper monitors and controls at byte granularity

2.1.2 Traffic Shaper Module (TSM)

Shaping is a QoS (Quality-of-Service) technique that we can use to enforce prespecified bitrates, different than what the physical interface is capable of. TSM is a hardware block that implements control of incoming data traffic which actually consists of read and write transactions, following the AXI4 protocol [62]. More precisely, the developed TSM controls the number of accesses based on configured/programmed bandwidth and regulates the flow of traffic, in order to apply restrictions on the consumed bandwidth. Figure 2 shows the prototype system that integrates the TSM. The proposed architecture contains the registers that are listed in Table I. The designer's options are, a) to attach it on a Master Interface and control it via the main CPU, b) to connect the TSM control port and establish the overall management of the TSM to another independent core, which thus offloads the burden to dynamically monitor and evaluate QoS per connection or per process; the main CPU (ARM Cortex-A9) communicates with that core only to provide configuration parameters. In this work we chose the second method.

TSM utilizes a separate AXI-lite interface for configuration. The TSM main components are shown in Figure 3. The TSM configuration is done via the MicroBlaze. The programming of registers determines the maximum data transfer bytes to transfer in a time window. The AxLen [62] signals, AxSize [62] and AxValid [62] are responsible for indicating a new transaction and whether this is a single word or a burst. The DMA inside the PS supports AXI burst lengths of 1 to 16 transfers, for all burst types [3]. However, the AXI4 protocol extends the burst length support for the INCR burst type to support 1 to 256 transfers. The burst length for AXI3 is defined as, Burst_Length = AxLEN[3:0] +

1, while the burst length for AXI4 is defined as, Burst_Length = AxLEN[7:0] + 1, to accommodate the extended burst length of the INCR burst type in AXI4.

Table 1: Description of the Traffic Shaper Module basic components

Register	Description
maxcc	Stores the Maximum time limit (cycles)
maxbw	Stores the maximum transfer bytes limit, maximum bytes to transfer in *maxcc* time interval
acc	Accumulating counter adding the new transaction bytes, should be less than maxbw limit
totalacc	Sums all the new accâĂŹs and keep the values after the reset
burst	Stores each new AXI4 transaction, burst or single word
cccnt	Clock cycle counter; in each clock cycle cccnt increases by 1 measuring total clock cycles

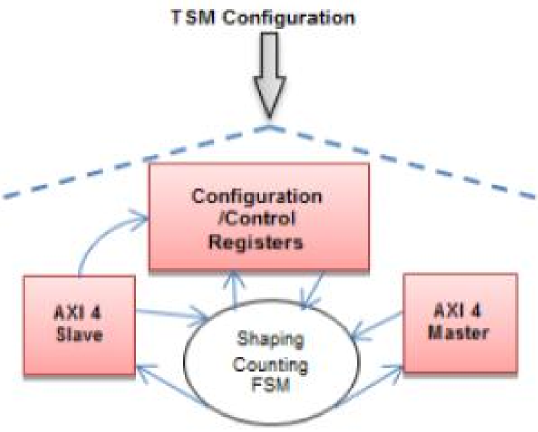

Figure 3: Traffic Shaper Module (TSM) internals as implemented in ZYNQ-7000; TSM configuration can occur on the fly and takes effect only after the current time window

2.2 Translation and Allocation Memory Management Unit (TAMMU)

The proposed TAMMU has the capacity to service simultaneous address translation requests, (a) from multiple devices, (b) from the same device (since this device can be accessed potentially by different VMs or processes) with transactions that belong to different VMs if virtualization support is included in a device.

Typically, one TAMMU component is responsible to interface a device and provide address translation services. While this option offers advantages in terms of performance and modularity, it lacks to provide optimal resource utilization.

Additionally, through increasing the size of the translation cache in a single TAMMU the performance advantage of an alternative system organization with multiple TAMMU blocks with smaller caches becomes debatable. We address both high-performance requirements and improved resource utilization by attacking the sources of inefficiency in a typical TAMMU architecture.

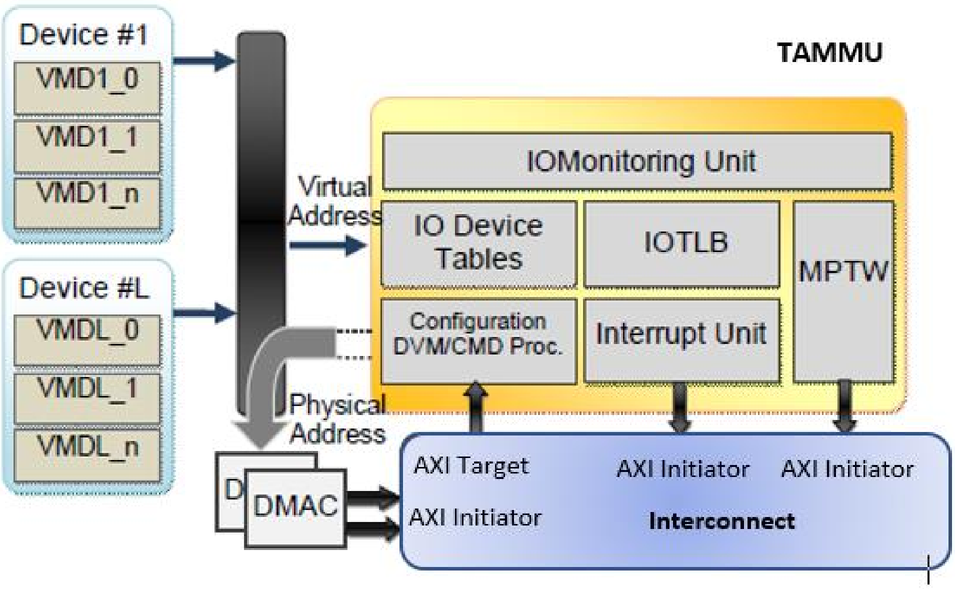

Figure 4: TAMMU block diagram; it supports many devices, efficiently virtualized along with a number of independent DMA controllers

The system organization part that integrates our proposed TAMMU is depicted in Figure 4 along with its internal sub-blocks. In particular, the developed TAMMU features an optimized translation look-aside buffer (TLB) to offer acceleration to multiple concurrent caching services for the translation process.

A TLB stores direct translations from guest virtual to system physical pages. A separate system initiator interface is integrated to accelerate page table walking. The IOTLB sub-block offers address translation caching services that have successful outcome (hit) until it encounters a miss and the request queue to the Page Table Walker is fully occupied. Each virtual entity (VM or VM-process address space combined pair) can be configured to utilize a different DMA engine bringing parallelism in performing DMA transactions, thus reducing or even eliminating stalls in the address translation process. Moreover, the DMA controller (DMAC) is activated to access the system memory only after a valid TAMMU response, so that together with the physically isolated interconnects to ensure isolation for all devices.

The proposed TAMMU architecture features support for multiple devices and multiple concurrently active VMs that access the connected devices. The consequence of serving requests from many VMs simultaneously is that the TAMMU supports identification of different virtual address spaces and pointers for all active contexts for thousands of VMID/ASID domains, both secure and non/secure. The processing engines for incoming request inside the IO Device Tables ensure device isolation by supporting protection mechanisms. Multiple isolated domains are supported by ensuring that all I/O devices are assigned to some domains, and that they can access only the physical resources (memory pages) allocated to these domains.

The Memory Page Table Walker unit (MPTW) is responsible to actually translate a virtual address when a TLB miss occurs. It receives a virtual address and discovers the corresponding physical address after traversing the page tables in the main memory. Modern processor architectures commonly support multiple virtual memory page sizes in order to efficiently map both large and small memory regions into processes' address spaces. On top, the hypervisor maintains per-VM guest-physical to machine address mapping tables, called nested page tables. Our MPTW unit is compliant with ARMv7 architecture [65] compatible, and thus it supports two stages of translation.

Stage one translation (S1) is used for translating a virtual address (VA) to intermediate physical address (IPA). The Stage two translation (S2) operates on the intermediate address to produce the physical address (PA).

The whole translation process for a virtual address requires twelve steps to identify the final IPA address, which in turn requires a final S2 translation summing to fifteen memory accesses in total. Thus, the IOTLB cache offers an invaluable performance boost to the translation process of virtual to physical addresses for ZYNQ architecture.

For improved resource utilization, the TLB is shared among all active domains using tags to identify each domain. In addition to entire TLB cache invalidation, invalidations per VMID, ASID, or virtual address are also supported for improved performance. Hence, both coherent DMA mappings and streaming DMA mappings (DMA addresses can be mapped only for the time they are actually used and unmapped after the DMA transfer) can co-exist and invalidations does not impact the translations that are already cached. Initially, as Fig. 4 shows the context of the current, validated address translation is retrieved from the IO Device Table, and after, the IOTLB unit accesses its cache to find if the translation outcome is present. A ternary context addressable memory is designed to support searching for different page sizes (4KB, 64K, 1MB and 16MB and the LPAE page sizes 2MB and 1GB).

The approach to offer near optimal cache service is to evict the least-recently-used (LRU) entry. The IOTLB embeds a true LRU eviction using one timestamp for each cache entry. A successful CAM search causes the increase of the corresponding cache line timestamp to the current time. The two most significant bits of the timestamp define four epochs. If the current time changes epoch then all timestamps that lag behind two epochs are advanced. For instance, if the most recently access cause a change from 01 to 10, then all timestamps are forced to change from 11 to 00 and from 00 to 01. This scheme ensures at least one epoch difference between the current and the LRU timestamp. The victim cache line is calculated through a tree of comparators to identify the entry with the smallest timestamp. The latency of six clock cycles to produce the next candidate victim is overlapped with the time to access the SRAM contents and the I/O handshake of the IOTLB subblock.

TAMMU features parallelized pipelined and out-of-order engines that facilitate the translation of multiple requests. To support this mode of operation the address translation requests need to be marked throughout the translation process. Thus, many DMA transactions (16 in the current implementation) from different VMs (or from the same VM) can be on the fly.

2.2.1 Functional Behavior and Synchronization

This section provides an overview of TAMMU behavior, leading to a preliminary API towards the hypervisor and guest OS components. DMA operations can be initiated by either the CPU or the device to perform reads or writes to memory.

A remote access via a DMA transaction initiated by the processor is managed by the processor's MMU. DMA transactions typically executed by a device contain virtual addresses (source and destination address fields) that must be translated by the TAMMU to the actual physical addresses. After a successful translation, the TAMMU sends the translated physical addresses (PAs or IPAs) to the device and corresponding DMA controller block to initiate the transaction. Block addresses are already registered in the IODT by the hypervisor (system software or operating system) during TAMMU setup. The TAMMU has the capacity to service simultaneous address translation requests (a) from multiple devices or (b) from the same device (if this device is potentially accessed by different VMs).

Figure 5 outlines TAMMU transactions during a typical address translation operation. The first step involves matching the incoming transaction to an existing context inside the device table. If an appropriate domain is not identified in the device table, then either a malicious access or a transaction from a yet uninitialized, recently-created VM is detected. For both cases, the hypervisor is notified, an error log is generated, and the translation is aborted.

Otherwise, matching preconfigured contexts for the address translation request results in permission to access the previously cached translations in the IOTLB. If a successful translation exists inside the IOTLB for the same domain and the same physical page, then this transaction is served immediately. Otherwise, a page table walk is initiated to perform the translation.

Based on the transaction context, Stage 1 or Stage 2 translation process is activated. Notice that many reasons for faults exist, such as non-initialized or inaccessible page and permission or security violation. TAMMU treats encountered faults depending on the pre-configured policy in two ways: " aborting the translation, generating an error event log, and notifying the initiator and/or possibly the hypervisor through an interrupt, or " interrupting the system providing related error context and stalling the current transaction until the issue is solved (TAMMU is notified to retry).

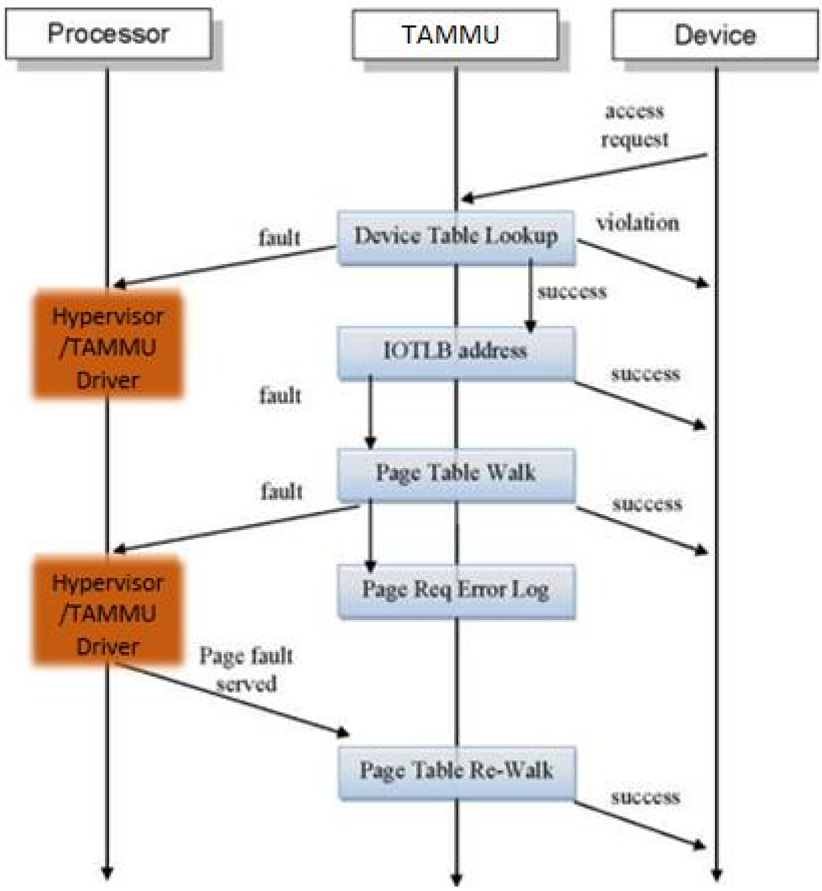

Figure 5: TAMMU functionality during address translation (transactions and protection)

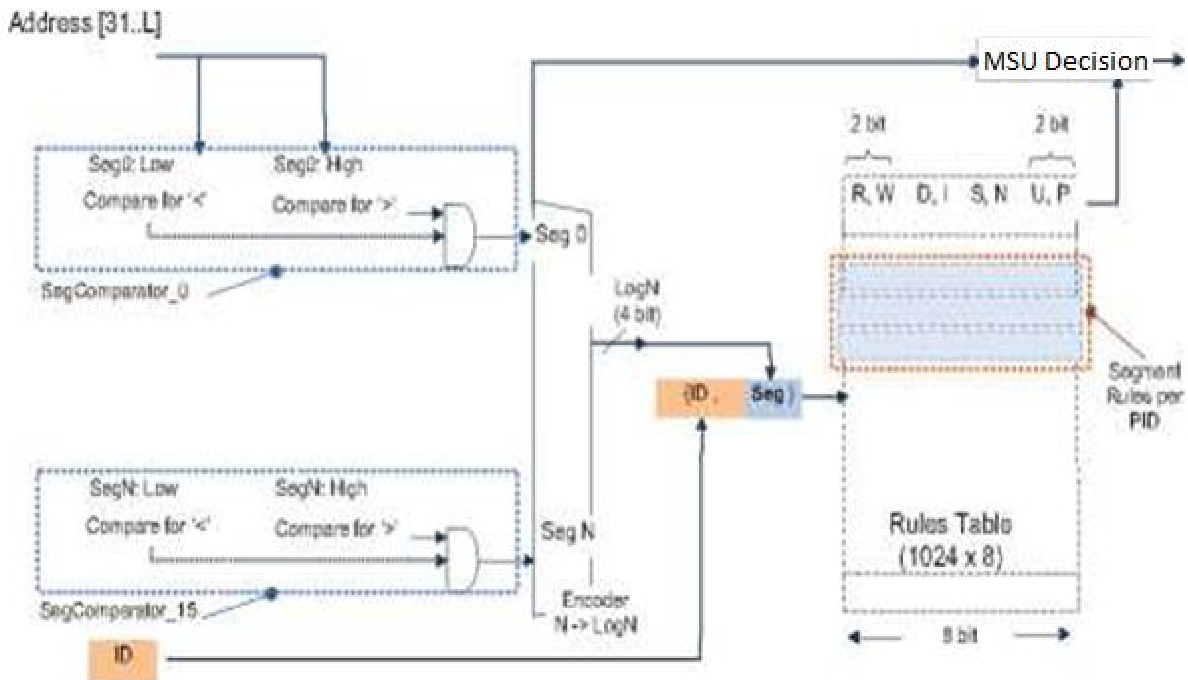

Figure 6: The MSU core integrates register-based parallel searching; one rule defines permissions for read/write, data/instruction, secure/non-secure, privileged/unprivileged access, thus matching the AxPROT field of AXI4- compliant transactions

2.3 The Memory Safeguard Unit (MSU)

The objective of the coarse grain Memory Safeguard Unit (MSU) is to control accesses to regions of system memory, or to any memory-mapped slave peripheral. On the basis of the topology, two appealing options exist for different reasons each. When integrating the MSU at the NoC initiator network interface the non-conforming transactions are not allowed to enter the NoC, thus saving throughput and energy. Alternatively, when placing an MSU after the target network interface, this offers a single point of protection for the target, thus avoiding consistency issues and synchronization overheads of updating the protection rules across different MSUs. If the access rules for a logical partition change, then all MSUs must be updated while safeguarding against transactions that may be on the fly.

The number of memory partitions, called hereafter segments, the size of each segment, and the properties or rules that are enforced for each segment are system-defined and programmable during operation. The MSU can be customized at design time to fit the requirements of a particular architecture, in terms of number of segments and their size

The MSU processes the address field of all incoming transactions, searching whether it is subject to preconfigured rules. We employed the following policy to apply access control to the incoming transaction requests:

- If there is no match in the MSU data structures, then the transaction request is allowed to proceed
- If an incoming address is in-between the preconfigured address range, then this match implies that the address belongs to a particular segment, which is subject to restrictions, and hence the rules table is accessed to decide if this transaction is compliant with the particular rules for this segment.

The coarse-grain protection implementation currently supports sixteen (16) memory segments. The maximum number of segments that can be implemented without affecting timing is technology-dependent. The 20-bit high order part of the physical address is utilized to concurrently search all defined ranges if it matches the range of each programmed segment. In this case, the corresponding enable signal for this segment is asserted. The part of the physical address that is used currently is fixed but can be easily dynamically programmed. In a multiprocess environment the OS (or a secured OS module in the TrustZone secure world for instance) can dynamically provide the 6-bit process identifier (PID) in order to access the base address of the rules defined for a particular PID. The encoded result is used to index the discovered rule inside the set of rules for this PID. Each rule in the implementation consists of eight non-encoded bits, hence rule memory is of size 1024 x 8. Each 8-bit rule in the implementation is formed by specifying the subfields: read, write, data, execute, privileged, non-privileged, secure, and non-secure domain. However, notice that not all subfields can be set independently.

As the firmware of I/O devices may contain vulnerabilities which can be exploited by attackers it is important to provide a strong, robust, and secure base for SoC designs that simply cannot be matched by software-only precautions. For example, it is demonstrated that a remote attacker can compromise a Broadcom NIC firmware through sending crafted UDP packets [67].

MMU security is an emerging technology also in embedded multicore SoCs that are used to enable usually a pass-through I/O virtualization model without risking direct memory accesses beyond the virtual machine's assigned memory. The hypervisor that manages MMUs for all I/O devices, which initiate transactions within the system, must ensure that a virtual machine has access only to its allocated resources and must prevent unauthorized accesses. However, a number of vulnerabilities have been identified circumventing the protection mechanisms introduced by MMUs. For example, via MMU protected I/O devices, a compromised or faulty guest operating system can manage to deny service to other virtual machines, especially when the guest OS is using the device in pass-through mode. It has been reported that TAMMU context registers that control the physical location of the TAMMU page table in system memory, are not properly protected from write access. Thus, during a context switch an unprivileged local attacker can use this flaw to switch to a fake page table via a specially crafted GPU command stream and access arbitrary physical memory locations.

The key feature of the MSU is the flexibility to associate the protected memory segments to the desired scope ensuring that accesses are within the allocated and legal ranges. We utilized the TAMMU that we developed at register-transfer level in order to explore the feasibility of employing the MSU for security. One instance of the MSU is integrated in the device interface of the TAMMU which accepts requests in the virtual address space of the virtual machine. Thus, by restricting the virtual memory for the virtual machine the TAMMU will never perform time and energy consuming virtual to physical translations for virtual addresses that are not allocated by the hypervisor.

In addition, we embedded one MSU component to protect the page table walking interface, since a malfunctioning hypervisor could provide erroneous context for a virtual machine to perform the virtual to physical address translation. Hence, the system can isolate the page tables to particular memory partitions building additional countermeasures to TAMMU standard functions.

2.3.1 MSU Prototype Implementation

To validate the functionality of our MSU component we developed a prototype using a hardware infrastructure on the basis of an AXI4 interconnect in the Xilinx Zynq- 7045 FPGA of a SoC-ZC706 platform. Figure 7 outlines the system architecture. The MSU component is attached to the AXI-to-AXI bridge. The core is connected to the Processing System (PS) master general-purpose port, and to the PS slave High Performance (HP) port; the latter is used to connect the Programmable Logic (PL) to the external DDR3 memory through a memory switch. The AXI-to-AXI Bridge transfers the CPU requests to the DDR3 memory controller abstracting the NoC fabric that interconnects all initiators to the system memory. All bus widths are 32-bit except the link to the HP port which is 64 bit. The PL is clocked at 100MHz.

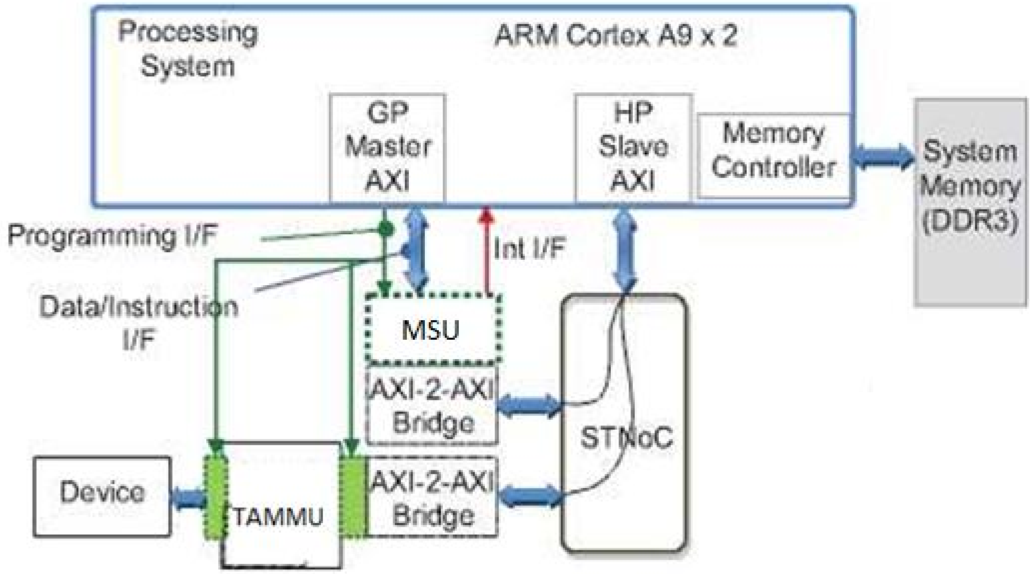

Figure 7. System-level organization on a Zynq-7000 XC7Z045 device securing initiators through using MSU cores

The CPU accesses the DDR3 memory with an average latency of 52 clock cycles (a clock cycle is 3.3 ns), while accesses through the NoC MSU and the bridge exceed a latency of 178 clock cycles; however, this effect is particular to the ZYNQ SoC communication infrastructure, since the A9 CPU can access the memory directly through the hard processing system. A malware access captured by the NoC MSU causes a CPU interrupt. The service routine needs to acknowledge the interrupt in order for the MSU to resume its operation. Typically, when investigating the incoming accesses, the MSU produces one decision every 4 clock cycles. However, there is a delay of over 350 clock cycles between the interrupt being asserted and the service routing clearing the control bit; in fact, during iterative generation of interrupts this delay reaches up to 520 cycles. Interrupt coalescing schemes can address such latencies and/or polling the MSU device to minimize interrupt delays.

In comparison to fine-grained protection environments such as Legba [68], the overheads in terms of area, complexity and in terms of overall system execution times are significantly smaller, due to its particular small size; MSU does not require hardware-walking page tables, or caching and can be customized at design time to be as much light-weight as required. Additionally, the system assets to be protected are generally not very fine-grain which translates to reduced invocation cost to re-program the memory domains and rules in the MSUs.

The cost of TAMMU operations can be as high as 4000 clock cycles for virtual address allocation in the Linux/Intel kernel as reported in [69], and hundreds of cycles for accessing the page tables in external memory for page table walking (in addition to internal TAMMU FSMs processing). Hence, an invalid access captured by the MSU improves system's energy efficiency as well. Finally, the disturbance introduced by an MSU to the common system data traffic is negligible since it is located at the initiator interfaces and configuration traffic is directed through isolated links for secure purposes.

2.3.2 MSU System-level Synchronization

The MSU mechanism must guarantee the invariant condition that all transactions in the system are processed by the correct set of rules. Thus, when the rules are dynamically updated in different MSUs, the accesses in the system should be handled by the same set of rules. To this end the MSU employs a synchronization mechanism to ensure that all accesses on the fly are completed before a rule is updated. Additionally, it is ensured that all rule programming operations will be completed so that when a new transaction is initiated it will face a coherent system. In this sense single-update atomicity is introduced to define the communication of all of the components involved. Atomicity does not define the exact instant when the rule is updated. What must be ensured is that no initiator component can ever observe a partially updated form of the atomic data that define rule information. The synchronization protocol defines the following sequence of operations.

- Initially, a sync signal is transmitted to all MSUs. The MSU component switches state to sync state in order to update the rules table. To enter the sync state any pending access request is completed and the slave interface of the AXI-to-AXI bridge becomes not ready. At the same time the programming port of the rules tables becomes unlocked allowing write operations.
- Maintenance operations on the rules table are permitted. New rules are allowed to be stored in the rule's memory. All transactions preserve the single-update atomicity principle.
- A complete signal to the MSU triggers the locking of the write port of the rules table and the update of the internal MSU status register that completion is done. However, the slave interface of the AXI-to-AXI bridge still remains not ready. No CPU core is allowed to perform any transaction until the next step is complete.
- A Read operation of the completion done bit in the status register will notify the CPU that rules updates were successful and thus the slave interface of the AXI-to-AXI bridge will become ready.

The latency to achieve a full system setup of MSUs is proportional to the number of MSUs and to the number of the segments that one new system configuration involves. In case of two idle MSUs we measured a total of 68 clock cycles to setup one rule, consistent for a single memory segment (in a baremetal environment).

2.3.3 Linux on the ZYNQ with MSU Support

On the ZYNQN FPGA, the MSU is viewed as a memory mapped device by the system. To perform read/write accesses to the 512 MB of physical memory (DDR3) through the MSU device (in the programmable logic) a special mapping is used from physical address 0x80000000 to 0xA0000000, over an AXI4 interface. The device also uses a non-shared interrupt line with number 91 to connect the device to the processing system's interrupt controller. Non shared interrupts were chosen for reasons of simplicity and since we had no issue with the total number of the IRQ lines.

2.3.3.1 Implementation of the Linux Driver

Our aim in developing the Linux drivers for the MSU device was to evaluate correctness and performance of our firewall implementation in a real system running GNU/Linux.

While kernel-space abstract the device at functional level, user-space drivers rely on the user-space I/O (uio) framework in the Linux kernel to export the device's memory-mapped registers into user-space using mmap. This can simplify driver writing, but causes implementation knowledge to diffuse into user-space, thus calling for application control of complicated system issues, such as the acknowledgment of IRQs in MSU.

Although a part of our kernel-space driver implementation is dependent on the embedded system design, many concepts introduced in this section are generic and focus on

- configuring and communicating with the MSU according to specifications,
- supporting specific commands to perform read/write requests in order to access data or instructions while considering different operating system modes, and
- handling interrupts generated by deny rules.

The Linux kernel-space driver of the MSU is in the form of a loadable kernel module which can be attached to the kernel at runtime. Since the MSU is viewed under Linux as a memory mapped device, the kernel-space Linux driver of the MSU are written as character devices and all I/O operations are memory mapped. Thus, CPU cores can access memory using virtual address pointers much more efficiently and without special-purpose instructions, while the compiler has much more freedom in register allocation and address-mode selection when accessing memory.

The basic functions used for the design of our character driver are as follows. During initialization of our kernel module, we perform the following functions:

- allocate the memory regions of the global physical address space used by our device using function struct resource *request_mem_region(unsigned long start, unsigned long len, char *name). Notice that the function allocates a memory region of len bytes, starting at start. If successful, the specified I/O memory allocation will be listed in /proc/iomem.
- map physical addresses to virtual ones, so that they can be accessed within the kernel, using two functions: a) void *ioremap(unsigned long phys_addr, unsigned long size) and b) void *ioremap_nocache(unsigned long phys_addr, unsigned long size). These functions obtain a virtual pointer to the start of the physical memory, whereas ioremap_nocache does not use the cache. This is especially important when the cache system supports write-back.
- register the interrupt using the function int request_irq (unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char *devname, void *dev_id). This function allocates interrupt resources and enables the interrupt line and IRQ handling.

During normal operation, we invoke

- the standard pair of functions unsigned int ioread32(void *addr) and void iowrite32(u32 value, void *addr) to access any specific memory mapped register (e.g. MSU (START, STOP) segment or rule definitions). The address addr is the address obtained from ioremap_* (perhaps with an integer offset). The return value of the ioread32 is what was read from I/O memory, while value is the value written to memory during iowrite32.

- the interrupt handler irq_handler_t handler(int irq, void *dev_id, struct pt_regs *regs) to clear any interrupt raised by the MSU device; notice that irq is the number of the IRQ line for the device, i.e. 91. Notice that we must have taken care of initializing the hardware and setting up the interrupt handler beforehand.

Finally, upon exiting the module, we call

- void release_mem_region(unsigned long start, unsigned long len) and void iounmap(void * addr) to free memory, and
- void free_irq(unsigned int irq, void *dev_id) to free resources related to the interrupt allocated with re- quest_irq().

2.3.3.2 Validation of the MSU Driver

Next, we describe two scenarios that have been used primarily to validate our driver. More complicated scenarios are now been considered, such as cache thrashing and performance isolation. However, describing higher-layer thread models and use cases is beyond the goal of this work which elaborates on MSU architecture and related system driver support.

In the first scenario, we have designed three kernel modules.

- The first one is a typical legitimate module (e.g. corresponding to a secure device driver component) whose accesses to memory must be protected by the MSU mechanism; for this reason, we name it protected module. In this module, we mainly call data=kmalloc(sizeof(unsigned int), GFP_KERNEL) to allocate a memory range, and use the virt_to_phys function to export the physical addresses allocated by kmalloc. Notice that unlike the equivalent vmalloc() function , the allocated memory region always forms a continuous address space, with the only restriction that the allocated address space must be below 24MB.
- The second module acts as an exploitation module, i.e. the module already knows the physical addresses of the previously allocated memory and uses the file interface of the module to perform illegitimate read or write accesses to the protected module's memory via ioread32() and iowrite32() functions.
- The third module is the Noc Firewall driver. This module initializes the memory segments and rules that enable/disable access by the exploitation module to the memory allocated by the protected module. lor this reason, the module performs ioremap() to map physical addresses of the segment and corresponding rule registers and the IRQ reset register. Then, the module calls request_irq(91, (irq_handler_t) my_irq_handler, IRQ_TYPE_EDCE_RISINC, ”nocfirewall”, NULL) to register my_irq_handler as the handler for irq 91. The module also provides standard read/write functions for capturing (resp. modifying) the status of the MSU. Notice that during writes, the segment to be protected is computed automatically from the allocated memory range

of the first module (provided as an argument), the corresponding rule(s) can be defined by writing directly to the MSU.

This above scenario relates to a memory-related bug, i.e. a corrupt device, or injected malicious code which may result in undesired behavior or even system crash, e.g. if sensitive information is overwritten. Using this scenario, the MSU driver has been validated for all rule's setup. This is based on a simple bash script which performs different actions, such as read or write accesses of data or instructions or even combinations; notice that kernel spinlocks have been used as barriers to avoid possible read/read, read/write, write/write reorders. The total number of ”isolated” unit tests is 4096 (= 16 x 256), while an even larger number of tests is needed to test for reorders (spinlock failures due); for testing two consecutive accesses the number of unique (non-symmetric) tests would rise to 65536. Next, we demonstrate only two “isolated” unit tests considering data read or data write.

- Test 1; lor validating data read requests, for each different value in the access rules of the firewall, we perform three actions: a) we write via the protected module a value to the allocated memory region, b) we read through the protected and the exploit module the data written (this access is routed via the MSU), c) we compare the two values, if they agree the read was successful. We find that for the 256 different settings of the MSU, only 16 patterns allow successful reads. These correspond to the following operating mode Read, Write, Data, Instruction, Secure, Non-Secure, User, Superuser = 1, 0/1, 1, 0/1, 1, 0/1, 0/1, 1, which sums up to an unsigned int in the set 169, 171, 173, 175, 185, 187, 189, 191, 233, 235, 237, 239, 249, 251, 253, 255.
- Test 2; lor validating data write requests, for each different value in the access rules of the firewall, we perform four actions: a) we write via the protected module a value to the allocated memory region,
- b) we (over)write via the exploitation module a different value to the same memory region (this access is routed via the MSU), c) we read once more via the protected module the data written and d) we compare with the original value written by the protected module in step (a). We find that for the 256 different settings of the MSU, only 16 patterns allow successful reads. These correspond to the following operating mode Read, Write, Data, Instruction, Secure, Non-Secure, User, Superuser = 0/1, 1, 1, 0/1, 1, 0/1, 0/1, 1, which sums up to an unsigned int in the set 105, 107, 109, 111, 121, 123, 125, 127, 233, 235, 237, 239, 249, 251, 253, 255.

The above two scenarios concerns validation of the MSU for read/write data protection. These scenarios can also be extended to consider other operating modes (e.g. user-level, non-secure mode). Moreover, it is interesting to mention that in order to consider instruction access we must rely on the (LDRT, STRT) class of ARM assembly instructions. These instructions enforce permission checking against the permissions that the page table specifies and can also be used to modify the privileged/user bits on the AxPROT signals on the AXI. An alternative scenario has been implemented in a single kernel module which allows for simpler manipulation and provides the possibility to examine driver performance issues. In this scenario, the main module allocates two memory regions: one freely accessed where no rule exists (alternatively, an allow rule could be in place), and another one which is denied read and/or write access by the MSU. Then, two threads are generated using kthread() function: one protected thread accessing the rule-free memory region, and another exploitation thread accessing the protected memory region (hence, it falls in interrupt mode). All accesses to memory regions are routed via the MSU and are timed by calling the ktime_get() function to enable functional profiling and obtaining performance characteristics. Results from preliminary performance profiling reveal the following.

During module entry:

- kmalloc, data initialization, virtual to physical address translation, as well as writing a segment or rule register take on the order of 100 to 1K ns
- request_mem_region, ioremap (for range and rule registers) and take on the order of 1K to 10K ns each
- ioremap for IRQregister takes 10K to 100K ns
- registering the interrupt handler takes 70K to 700K ns
- creating each kthread is delayed 100K to 1.5M ns

During normal operation:

- successful data reads and writes take 100 to 1K ns (writes are a little faster than read by almost 100 ns) • reads that cause interrupt (denied read) take 10K to 100K ns;

During module exit: freeing memory takes 100K ns.

Based on the above profiling, in relation to performance, we note that interrupt handling under Linux is very slow, taking approximately 90% of the total access time for both read and write accesses, thus ratifying our results from the bare-metal examples. High interrupt costs can be relieved by implementing a software intrusion detection system based on interrupt coalescing.

Kernel-space drivers, such as the ones discussed in this paper can be configured to provide protection of system and applications services, such as network socket-based interfaces or secure real-time monitoring hooks/actuators in heterogeneous distributed systems. Task-level protection of physical address space can rely on kernel-space drivers and processbased analysis (e.g. using Valgrind, Cachegrind and Linux pagemaps) and higher-layer security tools supporting service logs and data mining (e.g. using regular expressions).

3 Full Framework System Architecture

To validate the functionality of our Framework we developed a prototype using a hardware infrastructure on the basis of an AXI4 interconnect in the Xilinx Zynq- 7045 FPGA of a SoC-ZC706 platform. Figure 8 outlines the system architecture. We have a Network-on-Chip viewed as an interconnect intellectual property module with conventional read and write semantics; the network interface offers read/write and block transfers.

- To evaluate the impact of the TAMMU architecture to the system performance we developed DMA request generators to produce 10K virtual to physical address translation requests for each test case in order to stress the TAMMU subunits. At the same time page table initialization is created and stored in the system memory.
- The MSU component is attached to the AXI-to-AXI bridge. The core is connected to the Processing System (PS) master general-purpose port, and to the PS slave High Performance (HP) port;
- The AXI-to-AXI Bridge transfers the CPU requests to the DDR3 memory controller abstracting the NoC fabric that interconnects all initiators to the system memory.
- Traffic Shaper Custom circuitry is integrated at RTL level, which provides the ability of monitoring, control and supplying guaranteed bandwidth for critical applications. Component is attached to the AXI-to-AXI bridge and checking the transactions come from STNoC.
- We assume that Critical Application running on Processing System (PS) and VMs running on Programming Logic (FPGA).

The full functionality of three main components described in the sections before.

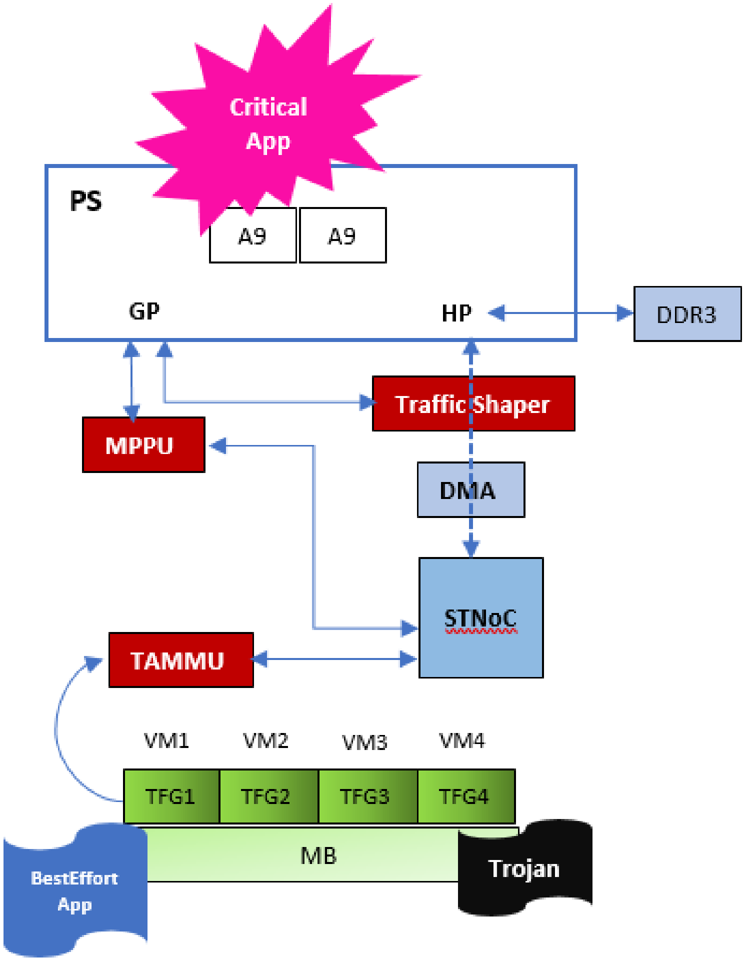

Figure 8: Prototype implementation on a Zynq-7000 XC7Z045 device

Table 2 summarizes the implementation results on the Zynq-7045 device. The MSU design consumes almost the same area as the bridge, while the delay to process an incoming transaction is four clock cycles. Comparatively, a 4x4 STNoC fabric [70] uses 24939 slice LUTs and 17983 slice registers and operates with a maximum frequency up to 129.3 MHz on the same device. The traffic shaper uses 993 slice LUTs and 1101 slice registers with a maximum frequency up to 210 MHz

TABLE 2: Summary of cost of proposed Architecture on XC7Z045 Device

Component	LUTs	Registers	RAMs	Frequency (MHz)
MSU(16 seg.)	655	1082	12	348
AXI Bridge	723	971		220.6
STNoC (4x4)	24939	17983		129.3
TAMMU	11150	9860	14+2.3	204
TRAFFIC SHAPER	993	1101	15	210

Total	3496	5661	41+2.3	

3.1 Step by step working example

In this subsection we describe clearly one by one the most important functionality steps of our architecture, in the working example:

1. Best- effort application data incoming on Microblaze. Multiple applications can be executing in parallel that also demand additional VMs.
2. TAMMU take the device request. The Memory Page Table Walker unit is responsible to actually translate a virtual address when a TLB miss occurs. STNoC checks for MPTW success message (a bit message).
3. MSU control accesses to regions of DDR. MSU integrating at the STNoC initiator network interface and control if the transaction is allowed or not to enter the DMA from NoC.
4. Assume that at the same cycle, Critical App core(A9) trigger the TSM. Traffic Shaper Module, throttles the throughput of the best effort task so as to provide guaranteed BW for the critical A9 function. With this way the incoming data traffic ensures DDR memory cell.
5. TAMMU keeps the BEApp information and via STNoC allocate it to DDR. MSU control again this transaction and allow the allocation.

In Trojan task case MSU will block the transaction. STNoC will expel out of system.

4 Experimental Results

Our TAMMU supports embedded storage for one thousand VM contexts partitioned to the number of connected devices. Hence, pre-configured VMs will cause zero latency to world switching, which is by far superior to any other solution currently known and thus we do not provide comparison in this context. To evaluate the impact of the TAMMU architecture to the system performance we developed DMA request generators to produce 10K virtual to physical address translation requests for each test case in order to stress the TAMMU subunits. At the same time page table initialization is created and stored in the system memory. The system shown in Fig. 9 is prototyped on the SoC-ZC706 platform [6] which connects to a DDR3 accesses the DDR3 memory with an average latency of 52 clock cycles (a clock cycle is 3.3 ns), while accesses through the NoC MSU and the bridge exceed a latency of 178 clock cycles It is assumed that the average latency due to non-determinism is bounded to 32 clock cycles per one page walk sequence.

In the experiments we assume that the TAMMU serves requests from three VMs: 7, 8 and 9 for devices that support requests rates equal to {RVM7, RVM8, RVM9}={80, 300, 800} clock cycles and deviation equal to {10, 40, 100}. A process generates VM requests with weights {4, 2, 1} to drive the address translation request port of the TAMMU.

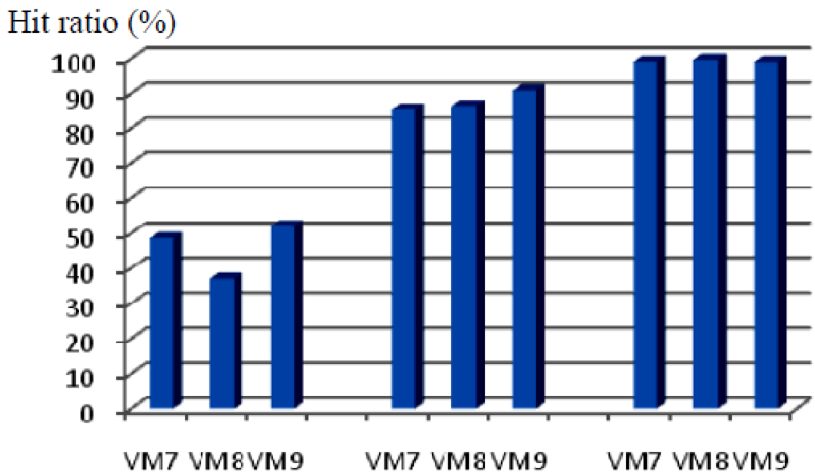

Figure 9: Cache hit ratio of the 64-entry fully associative IOTLB; three concurrent VMs issue DMA requests for new virtual addresses with a probability of 20%, 10% and 5%

Figure 9 depicts the effect of the IOTLB in three different test cases on the basis of the spatial distribution of the requests in order to spread the requests to a wide range (e.g. in situations that streaming DMA mapping vary in virtual address space and time). The process that generates the input requests creates new unique virtual addresses that have not been translated in the past with probabilities 20%, 10% and 5%. The hit ratio is rapidly increased almost exceeding 95% mainly due to the size of the IOTLB cache. Figure 10 shows the mean latency for address translation in clock cycles. When the misses are significant in the first test scenario the cost of the page table walk process is dominant. The situation gets even worse and the difference develops significantly in an actual full silicon implementation of a SoC, since the memory access time increases in terms of clock cycles.

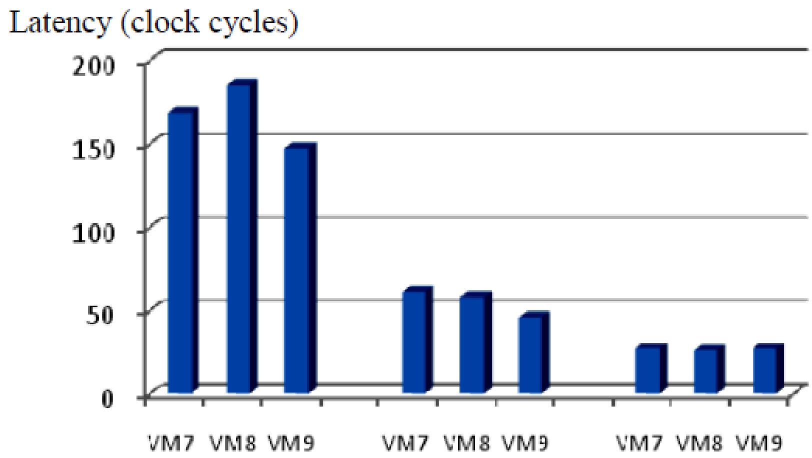

Figure 10: Latency of the TAMMU to provide the physical addresses to the devices' DMA requests; the latency is measured in clock cycles

The TAMMU is designed and implemented on a ZYNQ FPGA device. Table 2 summarizes the cost in memory per sub-block, including the internal tables in block RAMs and the interface FIFOs. In total, as the Xilinx ISE v16.2 post-placement tools report, the TAMMU occupies 11.15K lookup tables (LUTs) of the device without the RAM blocks while in comparison, the system NoC with six master and five slave interfaces follows the STNoC architecture [66] and consumes 24.95K LUTs, and the Cortex A9 (A9x2) occupies almost the full ZYNQ device . Table 3 gives the performance characteristics of the implemented TAMMU. All operations are designed to offer parallelism internal or across different sub-blocks, such as the TLB lookup is allowed while a page table walk is in progress as long as it is successful.

TABLE 3: Summary of cost of TAMMU in memory

Blocks	Freq (MHz)	Max Latency (5ns clock cycles)	Throughput
IOTLB	225	6	1 Inv/3cc, 1 Hit/6cc
IODT	368	4	1 Lookup/4cc, 1 Setup/12cc
MPTW	295	4	1 Issue/4cc
IOMON	316	3	256 events/cc
CPE	374	3	

As most of the drivers perform coherent memory mappings before they perform the streaming DMA mappings, it is likely that the coherent page frames will not be evenly distributed. When multiple devices are in use and each has its own coherent mapping, hot-spots will appear in some of the IOTLB sets, causing rapid evictions and resulting in a higher miss-rate. Amit et al. propose to locate each device's virtual I/O address space at a different offset [71]; our fully-associative IOTLB is subject to the amount of new virtual address mappings.

The cost of mapping/un-mapping in streaming type DMAs is addressed by batching or prefetching [72], in view of many I/O workloads that exhibit recurring patterns in their access sequences, often because pages are used as buffers which are reused over and over again. Our approach attacks the cost of frequent mapping first through using internal command buffers instead of letting the host software to use the system memory for this purpose [73]. Additionally, we embed more than enough contexts internal to the TAMMU to reduce also the delay to access the slow main memory. Thus, even though the number of virtual machines (VMs) is often very limited with only several or tens of VMs in a virtualized system we embed a large number of contexts to tackle the mapping/un-mapping cost in streaming type DMAs.

4.1 Evaluation Methodology

In Several different ways and methods exist, so as to monitor the activity of a processor or a module. As we presented previously this can be done by designing a controller that supports mixed-criticality service. Software techniques also exist, which support control at task level or at OS level [63].

Our methodology supports both control and monitoring on the Network-on-Chip (NoC) Interface, without requiring the re-design or tampering with the NoC routers or the memory controller [64]. The Traffic Shaper Module provides guaranteed bandwidth to the critical applications by limiting consumed bandwidth of the non-critical tasks.

We used a tightly system-coupled configuration to run our applications. Tightly system-coupled software means routines (modules or full applications) that work on only one type of system since these are dependent on each other and on the system configuration. For example, the driver of an embedded cyber-physical system device requires extensive programming changes to work in another environment, but is usually optimized to offer predictability, minimal latency and even fault-tolerance. We ran two stand-alone (baremetal) applications in A9 CPU and MicroBlaze. The MicroBlaze is responsible to monitor and control the configuration of the TSM registers. The A9 CPU can be configured to be in control of this block (start, restart, counters initialization and monitoring), but the main goal is to dedicate the processor only to the application running on it. In our case the accesses cross only one GP port (GP0) towards the slave PS interconnect (as shown in Fig. 2). If the DMA engine is utilized to perform bulk data transfers, then the DMA sequence of operations is as follows:

- DMA Configuration
- Setup DMA Interrupt
- DMA Initialization

- DMA Start Transfer
- DMA Check handler done
- DMA Stop and reset

If, we want to exploit the maximum theoretical bandwidth that manufacturers specify [60], then the NoC specificities and the memory controller properties should be carefully exploited. For instance, in the ZYNQ architecture any trans-action must use specific ports and route along the interconnection scheme to avoid conflicts. The maximum supported bus clock is 533 MHz in the DDR3 module for all speed grades, reaching a theoretical maximum bus bandwidth of 1333 Mb/s. All DMA transactions use AXI interfaces to move data between the on-chip memory, the DDR memory and the slave peripherals in the PL.

Initially, in evaluating the accuracy of bandwidth monitoring we used a software only approach. We used a baremetal application executing on the MicroBlaze which, with the aid of a hardware AXI performance monitor, captures the activity of the AXI interconnect. The A9 CPU drives the PS DMA to send traffic over this interconnect. As Figures 11 and 12 show, when fine-grain resolution is adopted the software monitoring delivers poor results, while when the hardware TSM is activated the mean error is negligible.

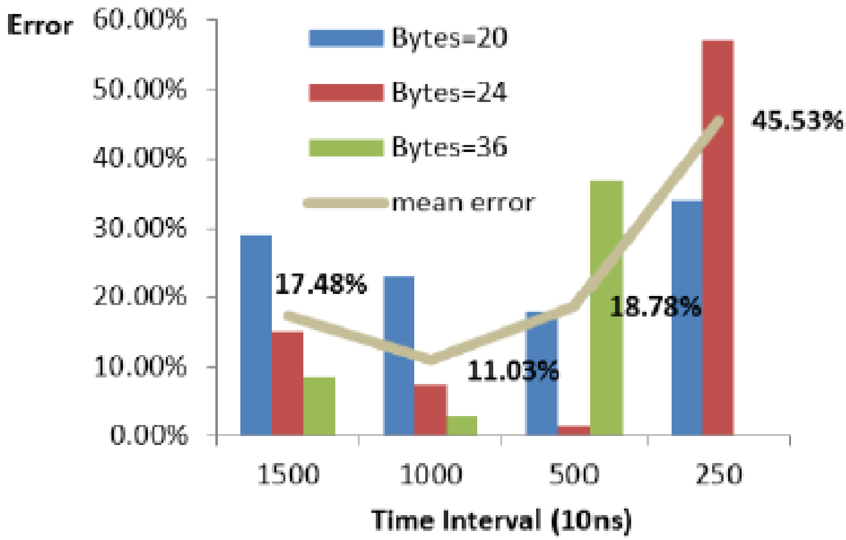

Figure 11: Comparison of accuracy when scaling the time window using varying data unit size

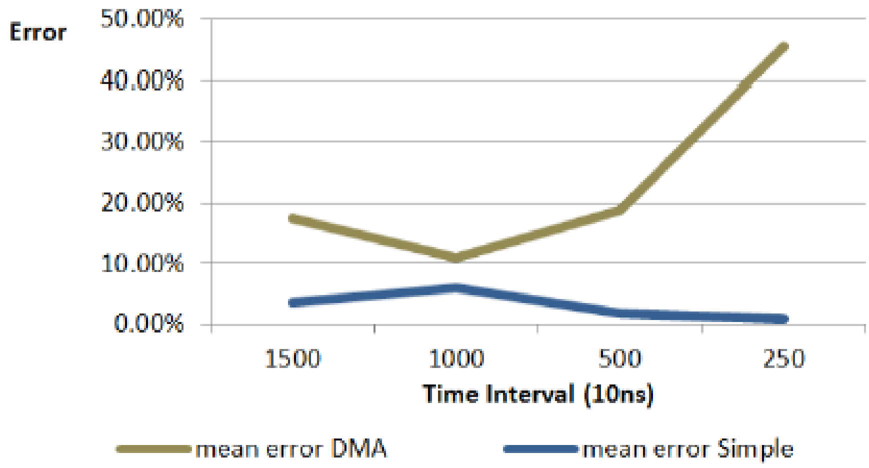

Figure 12: For four different time intervals (1500, 1000, 500, 250 cycles) we measure the error rate that raises between the theoretical and the measured tarffic bandwidth without activation of the TSM. For each interval we count three different data size transfers (20, 24, 36 bytes). For example, as we observe in the graph, for 1500 cycles the average difference arises to 17.48% of bandwidth that was measured compared to the theoretical values that were calculated. By enabling the TSM the error rate in the worst case reaches 6.07%

There are many options and routes to transfer data in an advanced SoC such as ZYNQ. Two typical DMA transaction examples include: 1) memory to memory (On-chip memory to DDR memory) 2) memory to/from a PL peripheral (DDR memory to PL peripheral). The incoming traffic arrives only from one GP Port crossing the AXI HP0 port and the S2 OCM port [60].

We developed two applications to move data using the first configuration. In one scenario, one alleged critical application runs on the A9 CPU in a baremetal fashion and the VMs enables the TSM and generates traffic. As Figure 13 shows, in the first scenario the VMs writes to the memory by using the DMA engine, thus reaching 23.5MB/sec. We also notice that the traffic from the A9 CPU towards the DDR is not affected, and this is because the TSM provides the maximum throughput to the critical application. In the second scenario in Figure 14 we clearly observe that the process on the A9 CPU that makes the Reads and Writes is influenced. The traffic generators running in the Microblaze is activated at specific times and significantly affects the bandwidth received by the A9 CPU.

As long as the Traffic Shaper Module is in a ‘disable’ mode, the DMA can reach up to a limit depending on AxLen (Burst size and Length). Moreover, as we have observed the GP Port performs poorly in throughput, as it also appears in Figure 13. When the TSM is enabled the shaper can be configured to regulate the traffic until the consumed bandwidth is less than or equal to the maximum bandwidth that can be reached when the shaper is disabled.

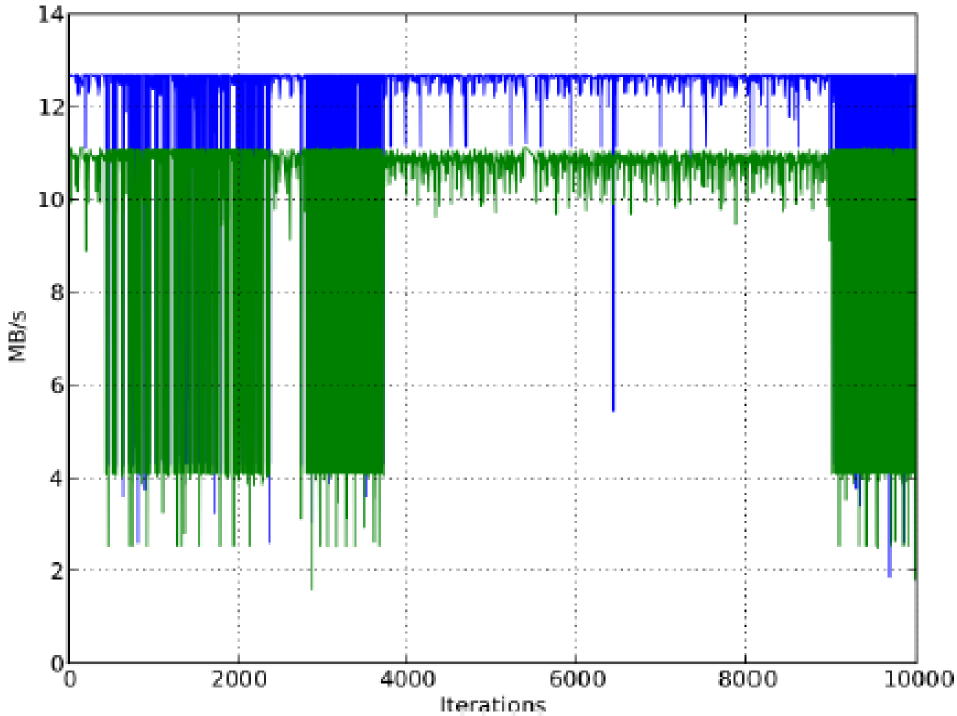

Figure 13: Nearly perfectly unaffected bandwidth served in control of the TSM

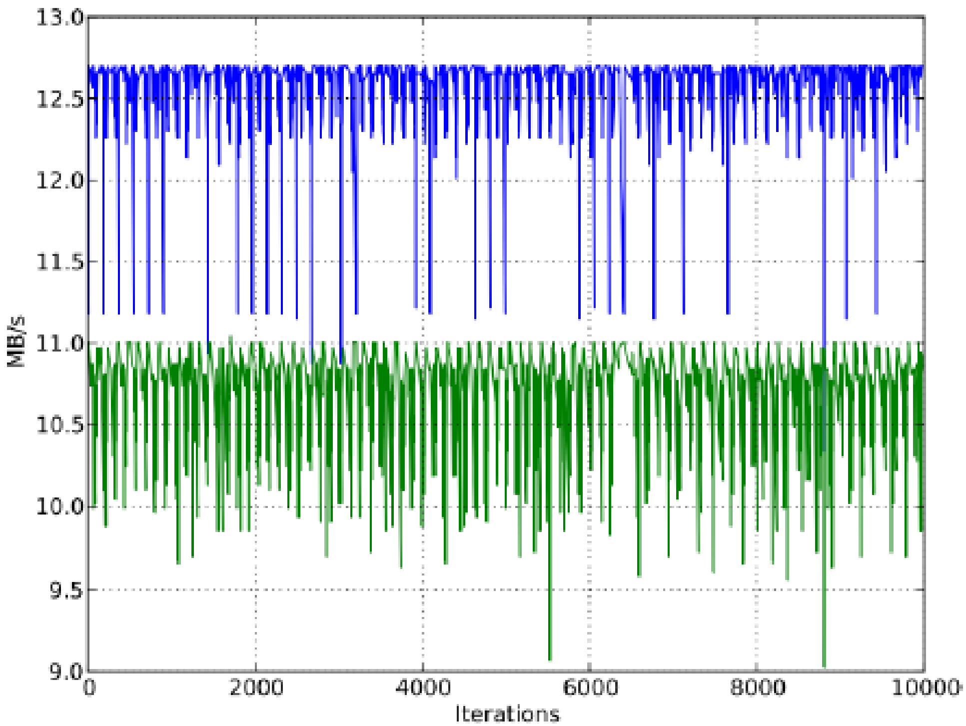

Figure 14: System impact when the TSM is deactivated; random DMAs cause irregular and uncontrolled memory bandwidth consumption

5 Comparison with the State of the Art

Compared to existing techniques, the proposed has the ability to combine dynamic memory allocation and bandwidth guarantee. However, it is questionable whether the architecture of such a complex system is beneficial in terms of resource utilization. That is, how much resources it consumes compared to alternative technologies. Another important issue is the performance of such a complex system. In this section we compare both the performance and overall utilization of our system compared to the major architectures presented.

For comparison purposes, we have integrated and synthesizable our architecture on the Virtex 5 platform with the support of Xilinx ISE Design Suite we are recording its total utilization on this platform.

TABLE 4: Resource utilization comparison between proposed and two alternatives technologies

Architecture	Protection	LUTs (%)		Registers (%)	
		XC7Z045	Virtex5	XC7Z045	Virtex5
Proposed	✓	0.5	2.4	1.3	1.9
sysAlloc [78]	×	5.5	-	1.9	-
DOMMU [79]	×	-	6	-	1
DMM [80]	×	-	47,9	-	14.4

In terms of comparing the proposed technology with those available in dynamic memory management, it is evident in Table 4 that our system has significant resource utilization advantages. The only case of no benefit is compared to the DOMMU slice registers. However, for a system that at the same time offers guarantee resource protection for critical task, the difference is insubstantial.

In Figure 15 we clearly observe the performance comparison of the proposed architecture with the widely used sysAlloc [78] and SoCDMMU [81] architectures.

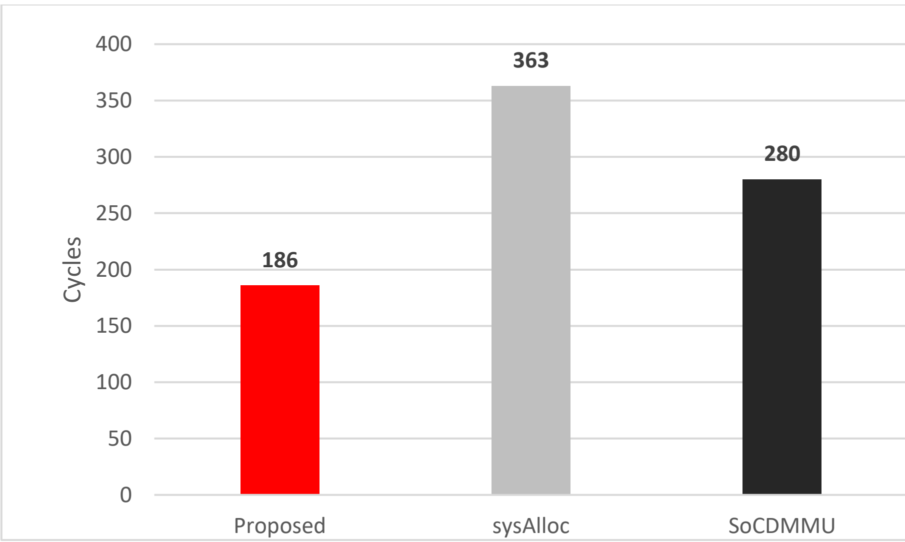

Figure 15: Performance compare: Proposed with sysAlloc and SoCDMMU architectures

The above figure shows the advantage of our architecture in relation to its allocation and de-allocation performance, along with other important works.

It is for a future work to test our system on multiple platforms to compare it to as many architectures as possible. Also, an issue to consider in the near future is its implementation in various operating systems in real-time.

6 Use case for Healthcare Application Example

Healthcare devices that integrate Bluetooth version 4.0 [77] may be found in three different locations on the patient's body, which transmit to a gateway device similar to the ZYNQ-based one that we presented previously. In this particular scenario we consider such a device to run a critical application that can process streaming data that reach 25Mbit/sec or 3 MB/sec each [77]. Consequently, we could reach up to N devices*3MB/sec theoretical aggregate throughput. The gateway device utilizes only a single Bluetooth interface to communicate with each external healthcare device.

In our use-case scenario that is depicted in Figure 16, the incoming data traffic can reach up to the maximum of 3MB/sec and the gateway forwards this traffic with at least equal speed. Consequently, the maximum aggregate bandwidth is 6MB/sec, i.e 3MB/sec incoming traffic and 3 MB/sec out going trace.

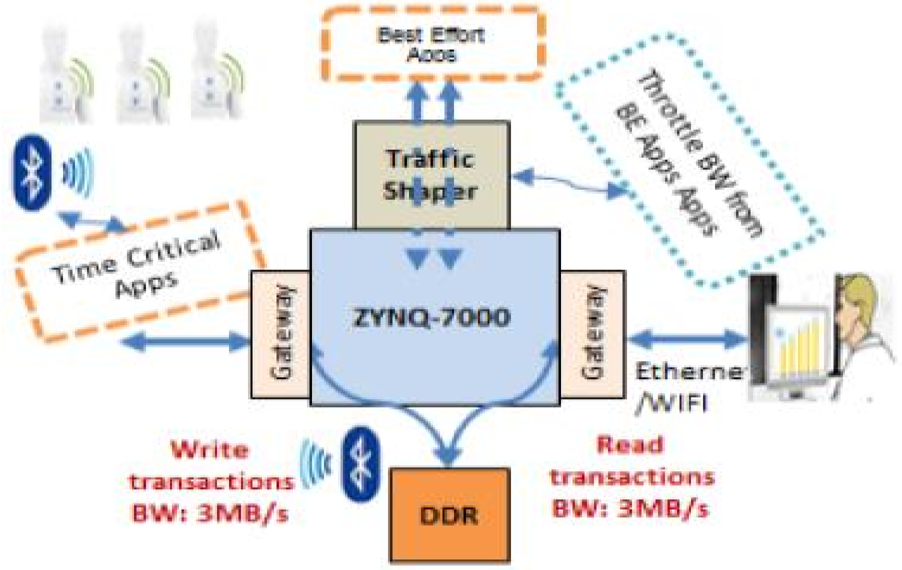

Figure 16: Healthcare use-case for a gateway of Bluetooth enabled sensors

In addition, best-effort applications can be executing in parallel that also demand additional memory throughput. As we have analyzed in the previous section, when the Cortex A9 CPU communicates through the AXI GP0/1 ports to the system memory we can achieve 10.797 MB/sec maximum throughput. The Bluetooth communicates via the ZYNQ gateway in order to feed the information to the user via a local Ethernet or WIFI connection.

When enabling the Traffic Shaper Module, we can throttle the throughput of the best effort applications so as to provide guaranteed BW for the critical function of conveying the devices data, which may contain important messages for the patients that must receive guaranteed delivery to the user. The TSM can be configured so that it can support the desired service level even to the maximum throughput (9 MB/sec).

7 Conclusion

We proposed a new effective technique that combines, secure service of critical tasks, security and safety of on-chip communications to set the ground for the development of mixed-critical embedded products. To implement an all-around approach to safety and security of embedded dependable systems, preventing and detecting potential attacks and secure critical tasks execution in on-chip communications.

Composing a secure SoC when insecure, untrusted hardware and software components are integrated remains an open challenge in computer security. This work proposed a hardware unit to leverage existing processor-centric hardware security mechanisms and OS-level software solutions.

As silicon manufacturers provide multi-core SoCs, using one designated core to execute secure configuration for different isolated worlds is one envisioned approach for our technology building a trusted execution environment.

Additional we incorporated scope, a novel hardware memory management unit (TAMMU) is introduced to map DMA virtual addresses from multiple devices to the correct VM's physical memory locations, offering enhanced protection, high performance supported by a configurable TLB and an integrated lightweight hardware monitoring unit to facilitate runtime system optimizations. Unlike similar hardware units we proposed architectural supports to offer zero latency activation of new mappings through increased memory storage for integrated hardware context structures. Additionally, we designed a number of optimizations such as parallelization of translation requests and memory accesses and domain tags inside the TLB for isolation and fast per domain invalidations.

A multicore embedded system is demonstrated in this paper, where the bandwidth is controlled by establishing a hardware Traffic Shaper Module. In such systems, in order to ensure system predictability and applications' properties such as criticality, we have to control the usage of systems resources. We achieve this by using a Traffic Shaper Module; its goal is to accurately control the number of accesses to provide the desired configured bandwidth. We presented a methodology for bandwidth management suitable for NoC interface and memory controller with the usage of custom circuitry and a minimal software monitoring application.

The system was prototyped using the Xilinx ZC706 System-on-Chip and the measurements extracted from a ZYNQ development board. By enabling the Traffic Shaper in our architecture, we achieved very fine- grain control with negligible overhead, while providing bandwidth of only 0.5-5 percent less than the theoretical bandwidth specified. The proposed architecture originates with combinations that enable address translation services with the capability that the proposed security framework provides guaranteed bandwidth and memory protection. It has been shown that despite the increasing need for hardware, our design has been able to maintain their low utilization with other well-known technologies applied in modern systems. It also proved to be capable of performance compared to other systems

Bibliography

[1] Tobias Zillner and Sebastian Strobl. ZigBee Exploited - The good, the bad and the ugly. In Black Hat USA 2015, 2015.

[2] Proofpoint. Proofpoint Uncovers Internet of Things (IoT) Cyberattack, Jan 2014.

[3] Eduard Kovacs. Attackers Use Stolen Credentials to Hack Cisco Networking Devices, August 2015.

[4] Joan Daemen and Vincent Rijmen. The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media, 2013.

[5] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-hashing for message authentication. 1997.

[6] Ling Hu and Cyrus Shahabi. Privacy assurance in mobile sensing networks: go beyond trusted servers. In Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, pages 613–619. IEEE, 2010.

[7] Steven Shannon. Access control of networked data, May 15 2001. US Patent 6,233,618.

[8] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. 2010.

[9] Hewlett Packard Enterprise (2015). “Internet of things research study, 2015.

[10] Arora D, Ravi S, Raghunathan A, Jha NK (2005) Secure embedded processing through hardware- assisted runtime monitoring. In: Proceedings of the design, automation and test in Europe (DATE’05), vol 1

[11] Barrantes EG, Ackley DH, Palmer TS, Stefanovic D, Zovi DD (2003) Randomized instruction set emu- lation to disrupt binary code injection attacks. In: CCS ’03: proceedings of the 10th ACM conference on computer and communications security. ACM, New York, pp 281–289..

[12] [Kc GS, Keromytis AD, Prevelakis V (2003) Countering code-injection attacks with instruction-set randomization. In: CCS ’03: proceedings of the 10th ACM conference on computer and communications security. ACM, New York, pp 272–280

[13] Ragel RG, Parameswaran S (2006) IMPRES: integrated monitoring for processor reliability and security. In: Proceedings of the design and automation conference 2006 (DAC’06). ACM, San Fransisco, pp 502–505

[14] [Mao S, Wolf T (2007) Hardware support for secure processing in embedded systems. In: Proceedings of 44th design automation conference (DAC), pp 483–488, San Diego, CA, June 2007.

[15] [Condit J, Harren M, McPeak S, Necula GC, Weimer W (2003) CCured in the real world. In: PLDI ’03: proceedings of the ACM SIGPLAN 2003 conference on programming language design and implemen- tation. ACM, New York, pp 232–244

[16] [12] Kc GS, Keromytis AD, Prevelakis V (2003) Countering code-injection attacks with instruction-set randomization. In: CCS ’03: proceedings of the 10th ACM conference on computer and communications security. ACM, New York, pp 272–280

[17] [Ragel RG, Parameswaran S (2006) IMPRES: integrated monitoring for processor reliability and security. In: Proceedings of the design and automation conference 2006 (DAC’06). ACM, San Fransisco, pp 502–505

[18] Mao S, Wolf T (2007) Hardware support for secure processing in embedded systems. In: Proceedings of 44th design automation conference (DAC), pp 483–488, San Diego, CA, June 2007.

[19] [Condit J, Harren M, McPeak S, Necula GC, Weimer W (2003) CCured in the real world. In: PLDI ’03: proceedings of the ACM SIGPLAN 2003 conference on programming language design and implemen- tation. ACM, New York, pp 232–244

[20] Hofmeyr SA, Forrest S, Somayaji A (1998) Intrusion detection using sequences of system calls. J Com- put Secur 6(3):151–180

[21] Forrest S, Hofmeyr SA, Somayaji A, Longstaff TA (1996) A sense of self for Unix processes. In: SP ’96: proceedings of the 1996 IEEE symposium on security and privacy. IEEE Computer Society, Washington, p 120

[22] Joglekar SP, Tate SR (2004) Protomon: embedded monitors for cryptographic protocol intrusion detec- tion and prevention. In: Proceedings on the international conference on information technology: coding and computing (ITCC’04), vol 1. IEEE Computer Society, Washington, p 81

[23] Erlingsson Ü, Schneider FB (2000) SASI enforcement of security policies: a retrospective. In: NSPW ’99: proceedings of the 1999 workshop on new security paradigms. ACM, New York, pp 87–95

[24] Evans D, Twyman A (1999) Flexible policy-directed code safety. In: IEEE symposium on security and privacy, pp 32–45

[25] Lampson BW (1971) Protection. ACM Oper Syst 8(1):18–24

[26] Austin TM, Breach SE, Sohi GS (1994) Efficient detection of all pointer and array access errors. In: PLDI ’94: proceedings of the ACM SIGPLAN 1994 conference on programming language design and implementation. ACM, New York, pp 290–301

[27] Miller TC (1999) Strlcpy and strlcat—consistent, safe, string copy and concatenation. In: 1999 USENIX annual technical conference. USENIX Association, Monterey, pp 175–178

[28] Messier M, Viega J (2005) Safe C string library. Available at http://www.zork.org/safestr/

[29] Chew M, Song D (2002) Mitigating buffer overflows by operating system randomization. Technical Re- port CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, December 2002

[30] Rostovtsev A, Shemyakina O (2005) AES side channel attack protection using random isomorphisms. Cryptology ePrint Archive, Report 2005/087 Gebotys C (2006) A table masking countermeasure for low-energy secure embedded systems. IEEE Trans Very Large Scale Integr Syst 14(7):740–753

[31] Goubin L, Patarin J (1999) Des and differential power analysis (the “duplication” method). In: CHES ’99: proceedings of the first international workshop on cryptographic hardware and embedded systems. Springer, London, pp 158–172.

[32] Nedjah N, de Macedo Mourelle L, da Silva RM (2007) Efficient hardware for modular exponentiation using the sliding-window method. In: ITNG ’07: proceedings of the international conference on information technology. IEEE Computer Society,Washington, pp 17–24.

[33] Chari S, Jutla C, Rao JR, Rohatgi P (1999) A cautionary note regarding evaluation of AES candidates on smart-cards. In: Second advanced encryption standard (AES) candidate

conference, Rome, Italy. http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm

[34] Aciiçmez O, Koç ÇK, Seifert J-P (2007) On the power of simple branch prediction analysis. In: ASI- ACCS '07: proceedings of the 2nd ACM symposium on information, computer and communications security. ACM, New York, pp 312–320

[35] Chari S, Jutla CS, Rao JR, Rohatgi P (1999) Towards sound approaches to counteract power-analysis attacks. In: CRYPTO, pp 398–412

[36] Danil S, Julian M, Alexander B, Alex Y (2005) Design and analysis of dual-rail circuits for security applications. IEEE Trans Comput 54(4):449–460

[37] Kessels J, Kramer T, den Besten G, Peeters A, Timm V (2000) Applying asynchronous circuits in contactless smart cards. In: Advanced research in asynchronous circuits and systems (ASYNC 2000), pp 36–44

[38] Muresan R, Vahedi H, Zhanrong Y, Gregori S (2005) Power-smart system-on-chip architecture for em- bedded cryptosystems. In: CODES+ISSS '05: proceedings of the 3rd IEEE/ACM/IFIP international con- ference on hardware/software codesign and system synthesis. ACM, New York, pp 184–189

[39] Muresan R, Gebotys CH (2004) Current flattening in software and hardware for security applications. In: CODES+ISSS, pp 218–223

[40] May D, Muller HL, Smart NP (2001) Non-deterministic processors. In: ACISP '01: proceedings of the 6th Australasian conference on information security and privacy. Springer, London, pp 115–129

[41] Irwin J, Page D, Smart NP (2002) Instruction stream mutation for non-deterministic processors. In: ASAP '02: proceedings of the IEEE international conference on application-specific systems, architec- tures, and processors. IEEE Computer Society, Washington, p 286

[42] Sprunk E (1999) Clock frequency modulation for secure microprocessors. US Patent WO 99/63696

[43] Benini L, Macii A, Macii E, Omerbegovic E, Pro F, Poncino M (2003) Energy-aware design techniques for differential power analysis protection. In: DAC '03: proceedings of the 40th conference on design automation. ACM, New York, pp 36–41

[44] Benini L, Micheli GD, Macii E, Poncino M, Scarsi R (1999) Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. ACM Trans Des Autom Electron Syst 4(4):351–375

[45] Saputra H, Vijaykrishnan N, Kandemir M, Irwin MJ, Brooks R, Kim S, Zhang W (2003) Masking the energy behavior of des encryption. 01:10084

[46] Fiskiran A, Lee R (2004) Evaluating instruction set extensions for fast arithmetic on binary finite fields. In: Proceedings of the 15th IEEE international conference on application-specific systems, architectures and processors, pp 125–136

[47] Großschädl J, Savas E (2004) Instruction set extensions for fast arithmetic in finite fields gf (p) and gf (2m). In: CHES, pp 133–147

[48] Steve H. Weingart, Physical Security Devices for Computer Subsystems: A survey of Attacks and Defenses. In CHES 2000 Proceedings of the Second Internationl Workshop on Cryptographic Hardware and Embedded Systems, pp 302-317

[49] Mao, S., and Wolf, T. Hardware support for secure processing in embedded systems. IEEE Transactions on Computers 59, 6 (June 2010), 847–854

[50] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, Vijay Varadharajan TrustLite: A Security Architecture for Tiny Embedded Devices, ACM, EuroSys 2014

[51] Joel P., Christian S., Alain G., Multi-compartment: A new architecture for secure co-hosting on SoC, IEEE International Symposium on Sytem-on-Chip, 2009

[52] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas, Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions, IEEE ISCA '05 Proceedings of the 32nd annual international symposium on Computer Architecture Pages 25-36

[53] Wollinger T., Guajardo J., Paar C., Cryptography in Embedded Systems: An Overview, 2003, In Proc. of the Embedded World 2003 Exhibition and Conference

[54] https://en.wikipedia.org/wiki/Trusted_Computing_Group

[55] IBM Zone Trusted Information Channel (ZTIC), IBM Global Technology Services, 2010

[56] Microsemi, Overview of Secure Boot With Microsemi Smart Fusiion 2 FPGAs, 2013

[57] Barker El. Recommendation for Key Management, NIST Special Publication 800-57 part 1, Revision 4. 2016

[58] ZigBee Alliance. ZigBee Specification, January 2008

[59] Zynq-7000 All Programmable SoC Technical Reference Manual UG585 v1.10, Xilinx, February 23, 2015

[60] MicroBlaze Processor Reference Guide, Xilinx UG984, April 2, 2014

[61] "Speci_cation Documents", Bluetooth SIG, www.bluetooth.org

[62] G. Ciocarlie, H. Schubert and R. Wahlin, A Data Centric Approach for Modular Assurance, Workshop on Real-time, Embedded and Enterprise-Scale Time-Critical Systems, 23 Mar 2011

[63] G. Kornaros, I. Papaefstathiou, A. Nikologiannis and N. Zervos, A fully-programmable memory management system optimizing queue handling at multi gigabit rates, Proceedings of the 40th annual Design Automation Conference, pp. 54{59, 2003}

[64] ARM Architecture Group. Virtualization Extensions Archi-tecture Specification, 2010, http://infocenter.arm.com/help/index.jsp? topic=/com.arm.doc.ddi0438d/CHDCHAED.html

[65] M. Coppola et al. "Spidergon: a novel on chip communica-tion network", in Proc. Int. Symp. System on Chip, 2004

[66] L. Duflot, Y.-A. Perez, and B. Morin, "What if you can't trust your network card?" in Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, ser. RAID'11, 2011, pp. 378-397.

[67] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser, "Legba: Fast hardware support for fine-grained protection," in Advances in Computer Systems Architecture, ser. Lecture Notes in Computer Science, A. Omondi and S. Sedukhin, Eds. Springer Berlin Heidelberg, 2003, vol. 2823, pp. 320-336.

[68] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "riommu: Efficient iommu for i/o devices that employ ring buffers," in Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '15, 2015, pp. 355-368.

[69] Tatas, K. Siozios, D. Soudris, and A. Jantsch, "The spidergon STNoC," in Designing 2D and

3D Network-on-Chip Architectures. Springer New York, 2014, pp. 161-190.

[70] Xilinx, 2014, security Solutions for Zynq-7000 All Programmable SoC, www.xilinx.com/products/silicon-devices/soc/zynq-7000/security.html

[71] Texas Instruments, October 2011, KeyStone Architecture, SOC Security User Guide, Literature Number: <SPRUHC3>.

[72] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, “Secure embedded processing through hardware-assisted run-time monitoring,” in Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, ser. DATE '05, 2005, pp. 178-183.

[73] S.Parameswaran and T. Wolf, “Embedded systems security – an overview” in Design Automation for Embedded Systems, 12(3), 173–183

[74] K. Hu, T. Wolf, T. Teixeira and R. Tessier, "System-level security for network processors with hardware monitors," 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2014, pp. 1-6.

[75] Jalali S., "Trends and Implications in Embedded Systems Development", http://www.jpspcb.com/upfile/2016/12/20161202213324_503.pdf

[76] Christoforakis I., Astrinaki M., Kornaros G., "Towards architectural support for bandwidth management in mixed-critical embedded systems" January 2018ACM SIGBED Review 14(4):21-26

[77] Specification Documents, Bluetooth SIG, www.bluetooth.org

[78] Xue Z, Thomas D.B.,“SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems” 2015 25th International Conference on Field Programmable Logic and Applications (FPL)

[79] Dessoukly G., Klaiber M., Bailey D., Simon S., “Adaptive Dynamic On-Chip Memory Managementfor FPGA-based Reconfigurable Architectures” 24th International Conference on Field Programmable Logic and Applications (FPL)

[80] Xydis S., Bartzas A., Soudris D., Lu Z., “Custom Microcoded Dynamic Memory Management for Distributed On-Chip Memory Organizations”, 2011, IEEE Embedded Systems Letters

[81] M. A. Shalan and V. Mooney, "A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip," International Conference on Compilers, Architecture and Synthesis for Embedded Systems, November 2000, pp. 180-186.