Horizon 2020 Program (2014-2020)

Cybersecurity, Trustworthy ICT Research & Innovation Actions
Security-by-design for end-to-end security
ICT 32-2014

Secure Hardware-Software Architectures for
Robust Computing Systems [†]

Deliverable D3.2: Design specification of the SHARCS hardware techniques

Abstract: This deliverable describes the design specification for all SHARCS hardware techniques, which will be used in the WP5 applications covering their hardware security requirements as analyzed in D3.1. It further explains how these techniques are integrated in the system architecture of each application to provide, together with the software techniques (D4.2) a complete solution for security.

Contractual Date of Delivery	Month 24
Actual Date of Delivery	Month 24
Deliverable Dissemination Level	Public
Editor	Ioannis Sourdis
Contributors	CTH, FORTH, NEU, ONAPP, EAB
Quality Assurance	Thomas Kamm (EB), Kaveh Razavi (VUA)

[†] The research leading to these results has received funding from the European Union Horizon 2020 Program (2014-2020) under grant agreement n° 644571.

The SHARCS Consortium

Foundation for Research and Technology – Hellas	Coordinator	Greece
Vrije Universiteit Amsterdam	Principal Contractor	The Netherlands
Chalmers Tekniska Högskola	Principal Contractor	Sweden
Technische Universität Braunschweig	Principal Contractor	Germany
Neurasmus BV	Principal Contractor	The Netherlands
OnApp Limited	Principal Contractor	United Kingdom
IBM - Science and Technology LTD	Principal Contractor	Israel
Elektrobit Automotive GmbH	Principal Contractor	Germany

Document Revisions & Quality Assurance

Internal Reviewers

1. T. Kamm (EB)
2. K. Razavi (VUA)

Revisions

Ver.	Date	By	Overview
0.0	25/10/2016	*Editor* I. Sourdis (CHAL)	Outline of the document created.
0.1	23/11/2016	*Editor* E. Vasilakis (CHAL)	Chalmers contributed to Section 3.
0.2	24/10/2016	*Editor* I. Sourdis (CHAL)	Introduction text added.
0.3	28/11/2016	*Editor* G. Christou (FORTH)	FORTH contribution in Section 3 added.
0.4	1/12/2016	*Editor* T. Kamm (EB)	EB contributions in Section 2 added.
0.5	2/12/2016	*Editor* J. Thomson (OnApp)	OnApp contributions in Section 2 added.
0.6	7/12/2016	*Editor* C. Strydis (NEUR)	Neurasmus contributions in Sections 2 and 3 added.
0.7	8/12/2016	*Editor* E. Vasilakis (CHAL)	Minor fixes, document ready for internal review.
0.8	18/12/2016	*Editor* T. Kamm (EB), K. Razavi (VUA)	Internal review completed.
1.0	28/12/2016	*Editor* I. Sourdis (CHAL)	Addressing internal review comments, preparing final version.

Contents

List of Figures

1 Introduction

This deliverable (D3.2) is the result of our activities in Tasks T3.2 Secure processors, T3.3 Secure memory systems, and T3.4 Secure communication. It primarily describes the design specifications of the hardware security mechanisms developed in the above tasks.

1.1 Outline

D3.2 is structured as follows: the WP3 hardware security techniques are listed in Chapter 2 and briefly mapped to the SHARCS use cases. The details of how WP3 techniques are adapted and integrated to the SHARCS systems are provided in WP5 deliverables. Subsequently, Chapter 3 provides the design specifications of each WP3 technique. It is worth noting that although WP3 activities have largely been determined by the requirements of the applications described in SHARCS, they are generic and may be additionally applied to other systems, consequently they are described here as such.

1.2 Connection to other SHARCS activities

The WP3 hardware security requirements (D3.1) were used as a starting point for D3.2. In turn, the design specifications of the WP3 techniques provide input to our further activities in Tasks T3.2, T3.3, and T3.4 which will produce their final implementations in year 3 of the project. D3.2 further provide input to WP5 where the hardware security techniques (as well as the software ones) will be adapted and used in the SHARCS applications.

2 Overview of the SHARCS hardware techniques and their use in the SHARCS Systems

This Chapter provides an overview of the SHARCS hardware techniques and how they are mapped and used in the systems considered in the WP5 of the SHARCS project, namely in the Implantable Medical Device (IMD), the automotive Electronic Control Unit (ECU) and the cloud computing. Together with the software techniques designed in WP4, they will provide a complete solution for security for the SHARCS applications. The detailed plan of integrating a subset of these techniques to each of the SHARCS applications and systems is described in the WP5 deliverables.

In WP3, the following SHARCS hardware techniques are designed:

- An Instruction-Set Randomization (ISR) mechanism for detecting code injection.
- A Control-Flow Integrity (CFI) mechanism for detecting and preventing code reuse.
- A custom security microprocessor (Smart-Implant Security Core (SISC)) for secure Implant-SoC communication.
- A Memory-Encryption Unit (MEU) for SoC.
- A Data-Integrity mechanism for memory blocks.
- An Instruction Read-Only Memory (ROM) for storing a fall-back default binary of a core used for mitigating code injection and code reuse attacks.
- A secure Memory Protection Unit (MPU) is designed to control memory accesses in a SoC.
- A lightweight, secure data-exchange protocol for embedded systems.

- A lightweight, secure key-exchange protocol for embedded systems.
- A FPGA-accelerated Network Intrusion Detection System (NIDS) for high throughput and low-latency network security of large networking systems.

In the remainder of this section we describe how the above hardware techniques will be mapped to the SHARCS use cases.

2.1 Use of the SHARCS hardware techniques in the Implantable Medical Device

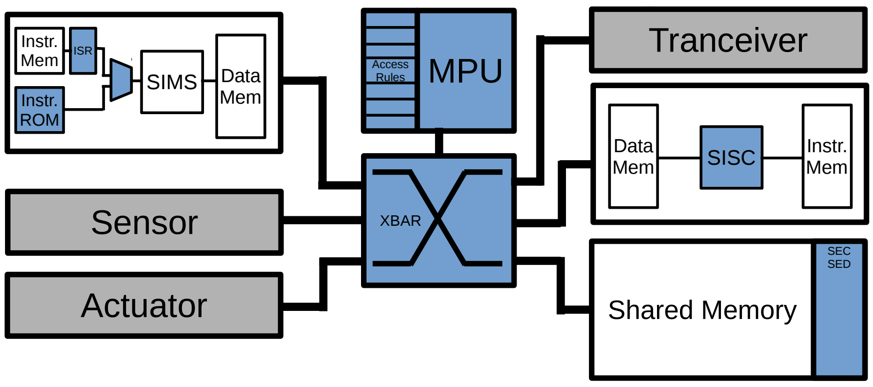

Figure 2.1: The IMD SoC with the SHARCS hardware techniques.

The design specifications of the implant use-case, as identified during the first project year, have driven the design and development of the secure implant SoC shown in Figure 2.1. The initial IMD SoC, originally introduced in deliverable D2.1, consisted of a number of building blocks: the main-functionality core (SiMS), a shared-memory block, sensor and actuator blocks, a transceiver block and – finally – a bus interconnect block (instantiated as an AMBA interface on the FPGA prototyping platform).

Of the hardware-security mechanisms identified in the introduction, a subset has been deemed suitable and selected for porting onto the implant SoC. These techniques are highlighted as blue-colored boxes in Figure 2.1, which illustrated the SHARCS version of the IMD SoC.

First off, a secondary Instruction Memory (IM) in the form of a ROM block, and a low-power version of Instruction Set Randomization (ISR) are shown. The secondary IM (see section 3.2.4) can be used generally as a standard method for safely updating the implant firmware. However, the backup IM is included for an additional reason: It can be combined with the

ISR mechanism to implement a defense against code injections and code reuse in the critical SiMS core. The ISR mechanism is implemented in the implant case as a XORing operator (see section 3.1.1) to perform instruction (de)randomization in the IM. Once a mismatch between the expected and the actually executed instruction is detected, a warning flag is raised and the SiMS program counter toggles to the backup IM which is guaranteed to hold an uncompromised SiMS binary.

Moreover, a security coprocessor (SISC) has been added to the implant SoC (see section 3.1.3), as originally conceptualized and proposed in [44]. The SISC core is the primary security gateway for the whole system. It essentially is an Application-Specific Instruction Processor (ASIP) optimized for running lightweight security protocols (see sections 3.3.1 and 3.3.2), well-suited for implantable devices.

The above components have already been added to the implant SoC and their correct functionality has been demonstrated during the first project review. More details on the current version of the implant platform are given in deliverable D5.1. As Figure 2.1 suggests, two more components are planned to be added to the overall design within the next year: a secure MPU, and a data-integrity mechanism for protecting the shared-memory unit of the implant. The two mechanisms are respectively detailed in subsections 3.2.2, and 3.2.3.

Finally, it should be mentioned that the blocks in Figure 2.1 highlighted in grey color are out of the scope of SHARCS and have been prototyped as memory-mapped I/O peripherals under the direct control of a UART port. More details on their implementation and prototyping details will be given in deliverable D5.1 and are out of the scope of this deliverable.

2.2 Use of the SHARCS hardware techniques in the Automotive application

In a modern premium car there are up to 80 different Electronic Control Units (ECU) installed. Each ECU is responsible for a specific function. You can find for example ECUs for engine control, braking, window lift and the infotainment system. A typical ECU (see figure 2.2) consists of an automotive microcontroller and different sensors (e.g. speed) as well as actors (e.g. engines). Furthermore the ECUs are interconnected with automotive bus systems like CAN to communicate with each other.

Through SHARCS, the Electronic Control Units (ECUs) will be enhanced with secure hardware and software to prevent attackers from manipulating functionality or gaining unauthorized access to those ECUs. The main threat, which must be prevented at all events, is the remote exploitation of a possible vulnerability in a large number of cars. New security mechanisms like Instruction Set Randomization (ISR) and Control Flow Integrity

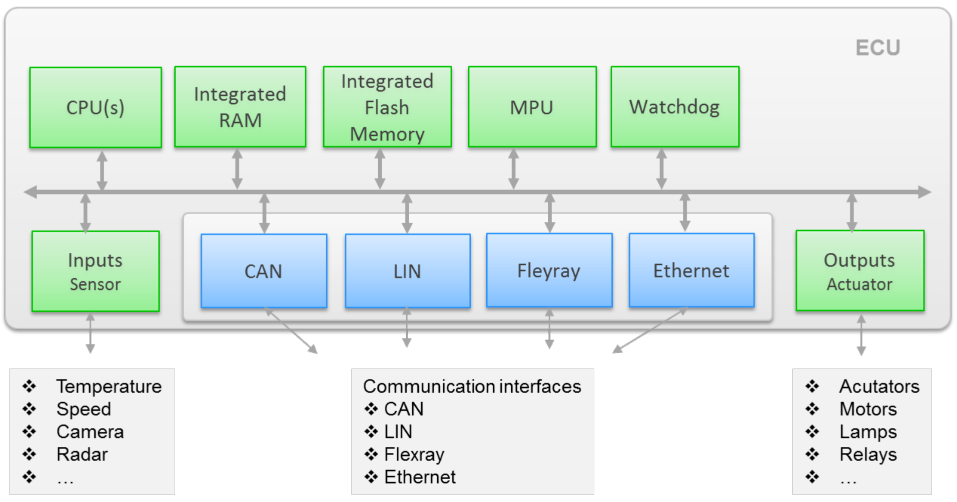

Figure 2.2: Typical ECU Hardware overview

(CFI) might be appropriate to achieve this and to guarantee the safety and security of passengers on the road.

The most secure approach is to harden the hardware layer as proposed by the SHARCS clean slate approach. However, in a typical Electronic Control Unit (ECU) an automotive qualified microcontroller is used. Enhancing these controllers with hardware security features is not feasible in the frame of the SHARCS project. We are therefore using a hardware prototyping platform instead to show the effectiveness as well as the usability in an automotive environment. If the solution is considered to be effective and robust future automotive microcontrollers might also support the newly introduced security mechanisms.

The hardware prototype emulates an Electronic Control Unit with communication interfaces and actors. In our demonstrator we are controlling some LEDs. We intentionally implemented a very general use case because the new security feature shall be applicable to a variety of automotive applications (e.g. infotainment, door locker). The prototype is shown in figure 2.3. The integrated Instruction Set Randomization (ISR) is highlighted in green color.

More details about the automotive use case as well as the prototype can be found in deliverable D5.3.

2.3 Use of the SHARCS hardware techniques in the Cloud pilot

The Network Intrusion Detection System (NIDS), is an FPGA based network traffic monitoring solution. The management and storage traffic are seen by

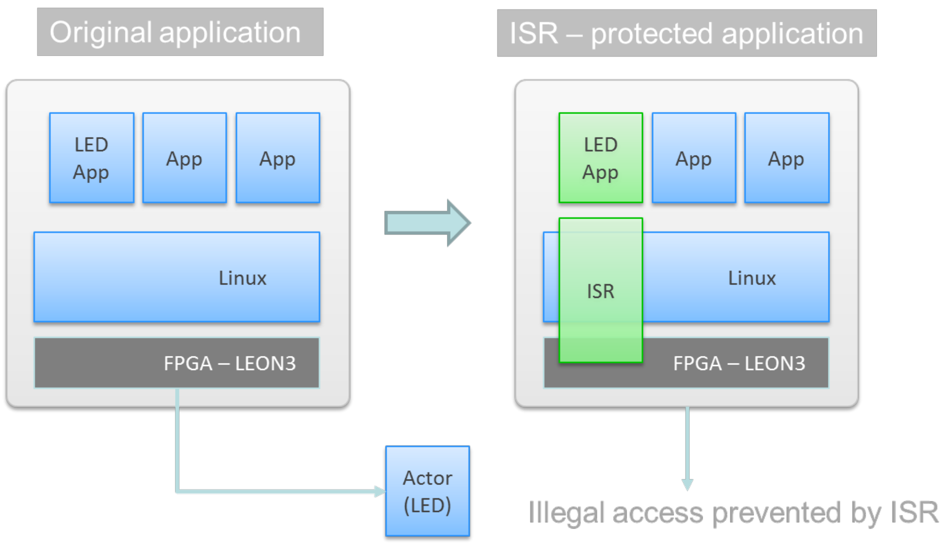

Figure 2.3: Overview of the automotive hardware prototype. The added SHARCS features are highlighted in green colour

the Backup Server as indicated in Figure 2.4 and so it makes sense for the NIDS device to be located there. The Network Interface Card (NIC) on the Backup Server can be set to promiscuous mode to monitor all traffic flowing in the network and the switch can be used to mirror and expose traffic that is directed at other ports to the Backup Server, for further processing by the NIDS hardware. The NIDS hardware will be able to flag warnings and errors to the Control Panel server by means of an API that will be developed specifically for this purpose.

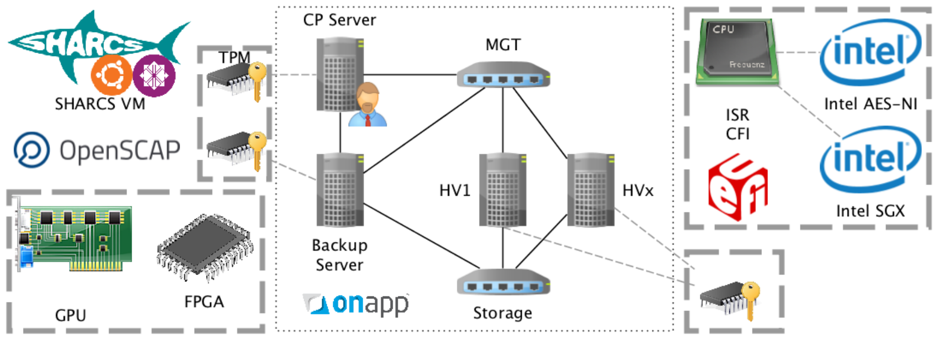

Figure 2.4: Hardware techniques that will be used in the Cloud pilot

Given that each of the Hypervisors in the OnApp Cloud deployment can host multiple tenants it will be important to attribute the actions of particular Virtual Machines (VMs) to malicious network actions. One way that this could be done is by providing applications with their own specific VLANs or monitoring based on a public IP that is associated with a VM. Each VM has a private IP address as well as potentially a public address and so it is important to detect any malicious traffic to or from such VM in order to carry out further actions. A simple step after detection could be to limit the network activity (rate limit) of that particular VM through modification of the hypervisor network for a given time if it is found to be causing interference. Further actions might be to deactivate the VM for a period of time. These actions though will all be defined by the Control Panel server to allow an administrator to decide what course of actions should be taken at either a Cloud or specific user level.

Experimentation will be needed to determine the right sampling intervals, the types of traffic to monitor and how to best highlight and respond to malicious flows. This will require collaboration between CTH and ONAPP with an experimental system to determine how to initially configure all the variables and determine how to best make use of the additional security hardware.

3 Design specifications of the SHARCS hardware techniques

In this section, we describe the design specifications of the SHARCS hardware techniques. In particular, for the processor security covered in Task T3.2, we describe the ISR and Control Flow Integrity (CFI) mechanisms designed by FORTH to detect and prevent malicious code injection and reuse. In addition, the custom SISC processor designed by Neurasmus and Chalmers is detailed that supports secure SoC communication, handling encryption/decryption tasks more efficiently, and security protocols. For memory security (Task T3.3), we explain our solution for encryption and data integrity offered by FORTH and Chalmers, respectively. Moreover, a ROM is integrated by Chalmers for storing a default binary of a core so to mitigate code injection and code reuse attacks. A secure MPU is designed to control memory accesses in a SoC. Related to Task T3.4, we describe the design specifications of the secure data- and key-exchange protocols designed by Neurasmus and Chalmers; these techniques are suitable for embedded systems that can use biometrics. Finally, an FPGA accelerated NIDS is designed by Chalmers[1] to provide low-latency network security for large networking systems. Details of the above follow in the rest of this chapter.

3.1 Secure microprocessor architectures

ISR, CFI, and the customized RISC processor for secure communication (SISC) are the three techniques we work on within T3.2. Their design specifications are described below.

[1]Software-only NIDS implementations are designed in WP4, however, depending on the throughput and latency constraints of a system, the FPGA-based NIDS is expected to deliver better performance.

3.1.1 Instruction-Set Randomization

Instruction-Set Randomization (ISR) is a promising technique capable of mitigating code injection attacks. The main principle behind ISR is to encrypt the instructions of executables residing in the main memory and decrypting them just before execution. This simple principle is enough to prevent the execution of arbitrary code injected to a running program by an attacker. Even if an attacker successfully injects code and diverts the control flow to the injected code, the result will be an execution exception, since the injected code will not translate to any meaningful machine instruction.

ISR is part of the SHARCS clean-slate approach and integrated to each use-case regarding the requirements. The automotive use-case consists of a POSIX compliant Operating System and System on Chip with a variety of peripherals. The modifications in order to enable ISR for user-level applications each layer is enhanced with security, from processor core to compilation and linking of the application as overviewed in figure 3.1.

Specifically, the application source code must be compiled with a new header (assembly file with new ELF header) which defines the encryption algorithm of ISR and instructs the ELF loader to enable ISR for the loaded application. Additionally, during compilation certain optimizations must be disabled in order to avoid data residing inside the code segment (e.g., jump tables). Finally, the linker using a linking script ensures that data and text section reside in different memory pages. Those modifications are minimal and most applications can be recompiled with ISR without any change in the source code, in other words, applications are transparently protected.

The operating system is modified in three key parts. The ELF loader recognizes ISR binaries and generates a unique random encryption key on each run. Also, the generated keys are stored in the ISR application Process Control Block. The scheduler is responsible to toggle the ISR functionality bit for ISR/non-ISR applications and set the appropriate key every time an ISR application is scheduled. Finally, the page fault handler encrypts the faulting page with the application key if the owner application is ISR enabled and the page fault originates from text segment (i.e. the faulting page contains code).

In the processor core the additional hardware consists of the key registers, where the scheduler loads the decryption key for the application, the ISR toggles bit in order to enable/disable ISR for applications with/without ISR and hardware decryption units. Before an instruction populates the instruction cache, it is decrypted using the loaded application key.

When an injection occurs, the execution of the incorrectly decoded instructions will most probably result in an illegal instruction exception which can be handled from the user-level.

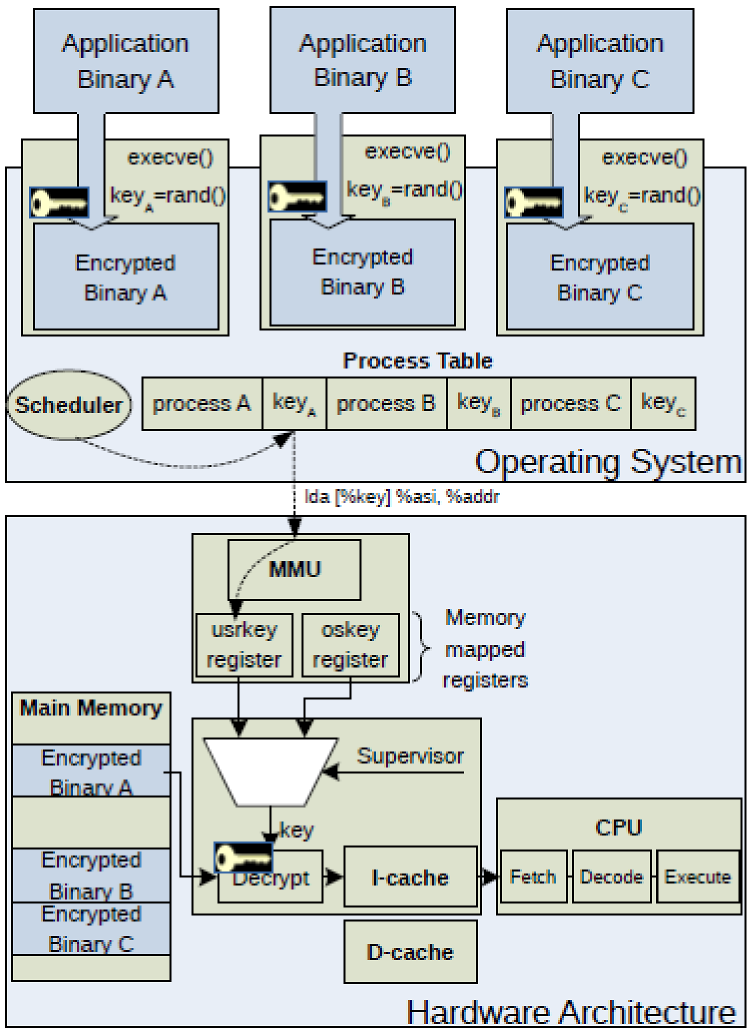

Figure 3.1: Instruction Set Randomization overview.

3.1.2 Control-Flow Integrity

Code reuse is one of the most prevalent forms of exploit in use today. To defend against such an attack, Control-Flow Integrity (CFI) was proposed [1]. Unfortunately, most CFI implementations to date either introduced prohibitive amounts of complexity to the application being defended, inducing extreme overhead, or reduced the complexity by simplifying the security scheme.

To address this problem, we proposed HCFI [12]; a CFI implementation designed as an architectural extension to CPUs. Our proposed scheme

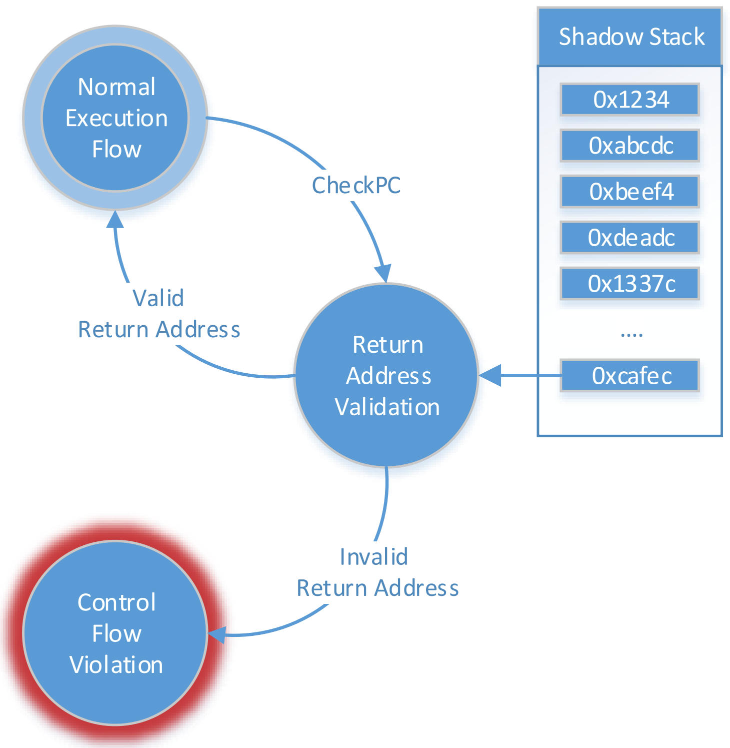

Figure 3.2: Return States.

offers a policy-agnostic, fine-grained, CFI extension to the Instruction Set Architecture (ISA).

In layman's terms, we added new instructions to an already existing ISA, namely the SPARC Architecture, that allow for a developer to define his own principles for CFI instrumentation, without compromising the security of the scheme, or the overall performance.

The additions to the ISA consist of 6 new instructions that operate on three new memory elements. Particularly:

A pair of instructions (SetPC, CheckPC) that, with the assistance of a hidden *shadow stack*, validate all return instructions allowing for perfect *backward-edge* CFI. SetPC pushes the current *Program Counter* to the shadow stack when a function call takes place, while CheckPC pops the top value from the shadow stack, and compares it to the address of the next instruction executed, typically the first instruction after a return. A Finite State Machine (FSM) describing this functionality is shown in figure 3.2.

Two instructions used in *forward-edge* CFI (SetPCLabel, CheckLabel), essentially validating all indirect branches and calls, using another of the memory elements, a *Label Register*. SetPCLabel stores a label, which it car-

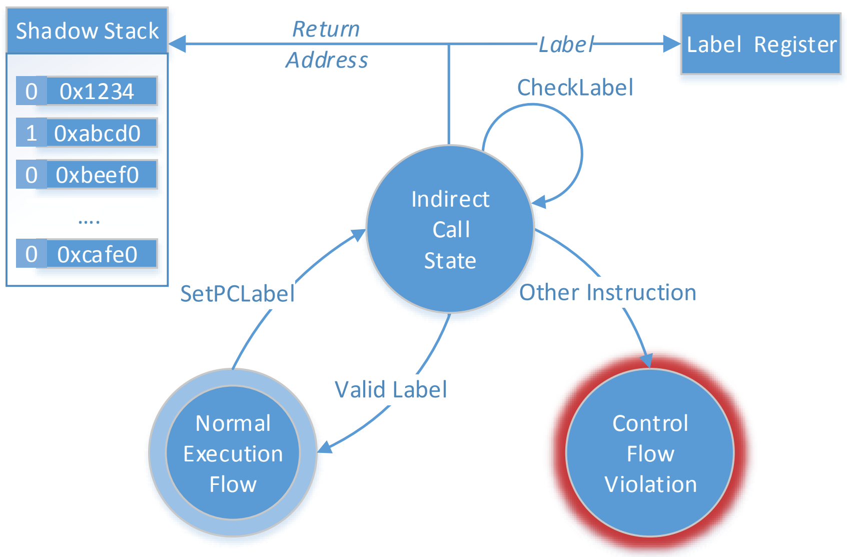

Figure 3.3: Indirect Call States.

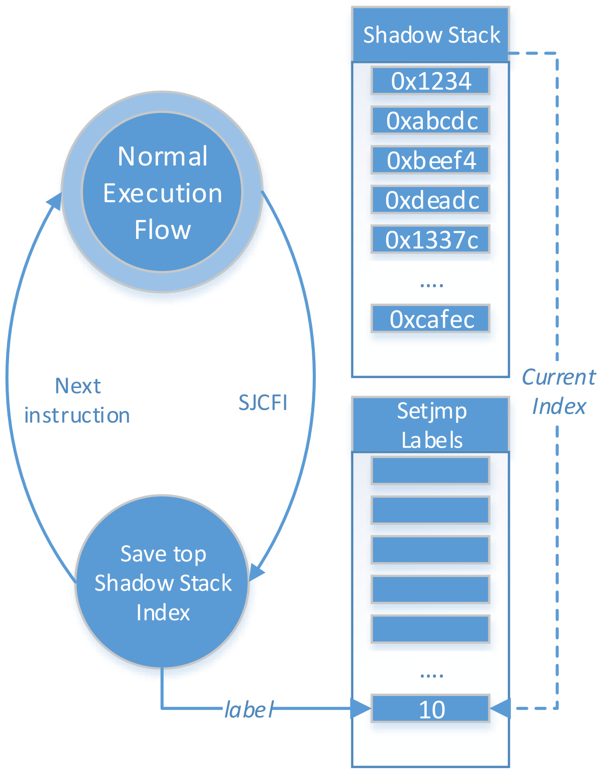

Figure 3.4: Setjmp Finite State Machine.

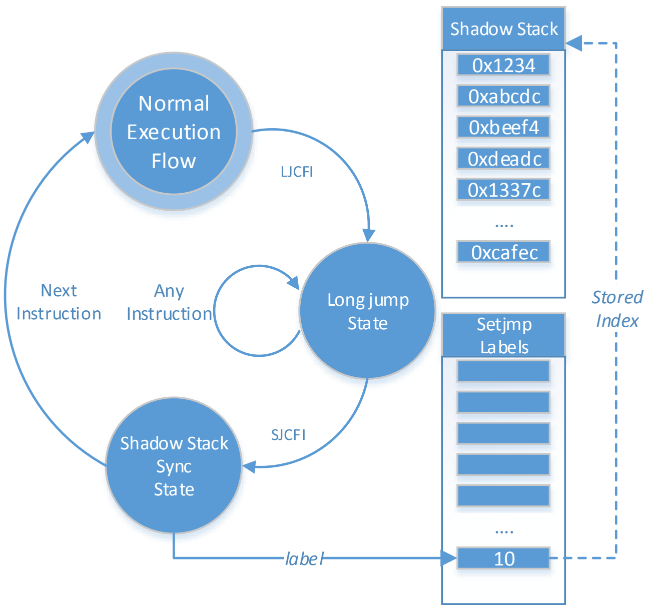

Figure 3.5: Longjmp Finite State Machine.

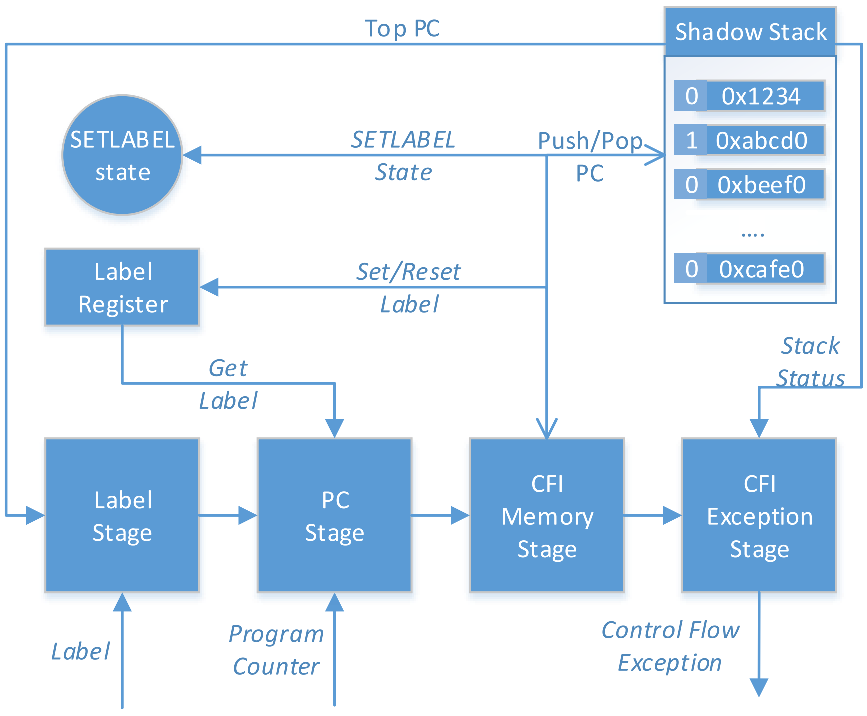

Figure 3.6: The CFI pipeline.

ries as an argument, to the label register, while also executing the functionality of SetPC. Finally, it sets a rule that only CheckLabel instructions are legal until a CheckLabel with the correct label is executed. CheckLabel compares the label it carries with the register. The granularity of the policy enforced is controlled by the developer and the effort of analysis they perform on the application code. The granularity, of the policy enforced, can be as fine as required. The FSM of the forward edge functionality is detailed in figure 3.3.

The final pair of instructions(SJCFI, LJCFI), assisted by the last memory element, allow for the implementation of workarounds for edge-cases (e.g. setjmp/longjmp). An SJCFI instruction stores the shadow stack's current height to the memory (shown in figure 3.4), while an LJCFI sets the current height to the one stored in memory, as long as the value is lower than the current (figure 3.5). Those can override the shadow stack's current height to lower it to a previous value; essentially jumping over stack-frames.

Those instructions are executed in a separate pipeline (figure 3.6), parallel to the original core's pipeline. The resulting spacial overhead and power requirements are very low, making this design suitable for small form-factor devices.

3.1.3 SISC: customized processor for secure communication

In this subsection, we concisely present the SISC architecture, a custom-designed, security ASIP. By definition, an ASIP is more power-consuming but also more *flexible* compared to an ASIC through changing the binary code it executes. Thus, an ASIP strikes a better trade-off between optimized performance/power and design flexibility.

The use of processor-based implant controllers has been demonstrated in various designs in the past [43, 17, 15, 50, 46, 48, 19] whereby in-vivo reprogramming of the implant has been successfully performed. Although these cases focused on reprogramming the main implant processor, in this work we extend the reprogramming capabilities of the implant security core. We believe this feature to be pivotal for the design of realistically safe and secure future implants for the following reason: IMDs comprise a narrow market niche and, as such, have to this point attracted only limited (malevolent) attention. As IMD use is becoming more widespread over time, more systematic and versatile security attacks are bound to be mounted. Unfortunately, under such conditions, the incorporated security of many future commercial IMDs will be compromised. The only eventual safeguard against this will be the in-vivo reprogramming of such devices for adding extra phases in the security scheme, changing the protocol altogether, and so on. Such major updates are impossible under an ASIC approach and device replacement through surgery will be needed, which is an action to be taken only as a last resort.

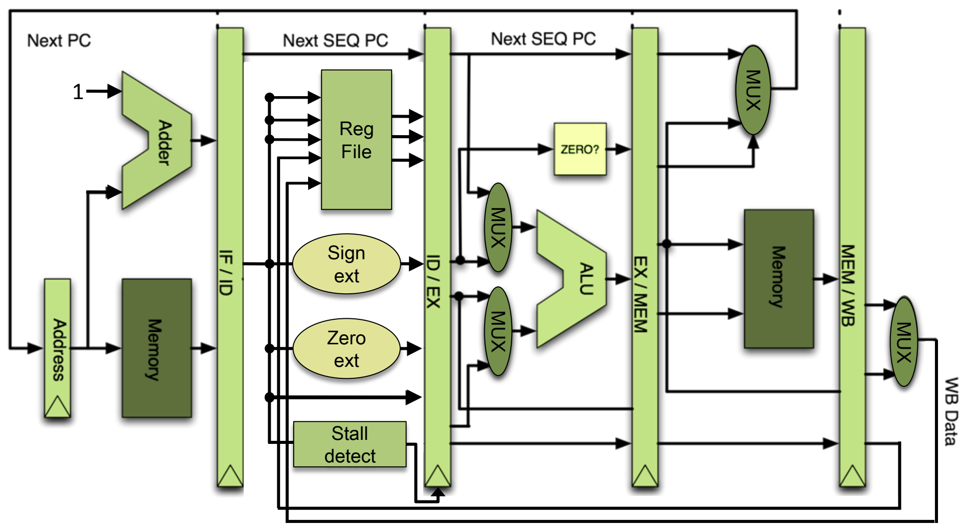

Figure 3.7: SISC-architecture schematic diagram.

Since our SISC design needs to be low-energy, but also perform non-trivial computations (see sections 3.3.1 and 3.3.2), we have chosen to implement an integer, in-order RISC-style architecture. We have selected to design a typical 5-pipeline-stage processor which consists of Instruction-Fetch, Decode, Execution, Memory and Write-Back stages. It consists of a 16-bit instruction set along with 16 registers of 32 bits each. In our case, the use of 32-bit registers is important because the encryption and decryption algorithms of our design benchmarks are operating on 32-bit quantities (i.e., MISTY1). When run on a standard, in-order RISC core, the encryption algorithm generates a significant amount of read-after-write hazards. In order to reduce the instruction binary size – or, equivalently, the implant memory needs – in our SISC design, NOPs are removed from the binary at compile time and are dynamically inserted at runtime whenever a hazard is detected. With this optimization, the IMEM size is $24\ KB$ and the DMEM size is $16\ KB$. The architecture schematic is shown in Figure 3.7.

The SISC ISA consists of 25 instructions listed in Table 3.1. Profiling of the SISC security-protocol traces (to be discussed later on) have yielded various potential optimizations. One such optimization is the replacement of the very frequent *mov-and-beqz* group by an instruction extension (*mandb*), highlighted in grey in the table. With this optimization introduced in the baseline core, the execution time has been reduced by 34.31% and the energy cost by 34.26% at a scant power increase of 0.73%. The SISC processor and C compiler have been designed by using the Synopsys Processor-Designer and Compiler-Designer tools.

A number of other optimizations is possible and constitutes part of the ongoing effort of NEU. For instance, the current version of the C compiler

Table 3.1: SISC Instruction-Set Architecture: supported types and syntax are listed. An implemented instruction extension is highlighted in grey.

type	**instruction format**			
rr	op [4]	reg [4]	reg [4]	funct [4]
rrr	op [4]	reg [4]	reg [4]	reg [4]
ri	op [4]	reg [4]	imm [8]	
jump	op [4]	imm [12]		

name	**type**	**assembly**	**action**
jr	rr	jr rd	branch to addr[rd]
and	rr	and rd,rs	rd $\leftarrow$ rd and rs
lw	rr	lw rd,rs	rd $\leftarrow$ mem[rs]
sw	rr	sw rd,rs	mem[rs] $\leftarrow$ rd
mov	rr	mov rd,rs	rd $\leftarrow$ rs
not	rr	not rd,rs	rd $\leftarrow$ not rs
or	rr	or rd,rs	rd $\leftarrow$ rd or rs
xor	rr	xor rd,rs	rd $\leftarrow$ rd xor rs
sub	rr	sub rd,rs	rd $\leftarrow$ rd-rs
add	rr	add rd,rs	rd $\leftarrow$ rd+rs
lb	rr	lb rd,rs	rd $\leftarrow$ mem[rs] (byte)
se	rr	se rd,rs	if(rd==rs) rd$\leftarrow$1 else rd$\leftarrow$0
sgt	rr	sgt rd,rs	if(rd>rs) rd$\leftarrow$1 else rd$\leftarrow$0(signed)
sgtu	rr	sgtu rd,rs	if(rd>rs) rd$\leftarrow$1 else rd$\leftarrow$0(unsigned)
beqz	rr	beqz rd,rs	if(rd==0) branch to addr[rs]
subi	ri	subi rd,imm	rd$\leftarrow$rd-imm
addi	ri	addi rd,imm	rd$\leftarrow$rd+imm
li	ri	li rd,imm	rd$\leftarrow$imm
sftl	ri	sftl rd,imm	rd$\leftarrow$rd<<imm
sftr	ri	sftr rd,imm	rd$\leftarrow$rd>>imm(sign extension)
sftru	ri	sftru rd,imm	rd$\leftarrow$rd>>imm(zero extension)
cb	rrr	cb rd,rs1,rs2	exchanges the $rs2^{th}$-byte of rd by the LSB of rs1
mandb	rrr	mandb rd,rs1,rs2	if (rd and rs1==0) branch to addr[rs2]
j	jump	j imm	branch to addr[imm]
jal	jump	jal imm	branch to addr[imm] and r15$\leftarrow$PC+1

performs efficient scheduling of the program code but, other than that, it performs no additional optimizations, e.g., for low power. However, this version of SISC (and compiler) is intended as a proof-of-concept security core for demonstrating efficient, system-wide security.

3.2 Secure Memory Architectures

In T3.3 we work on the following techniques related to secure memory hierarchy of a system. The first two approaches provide data protection through encryption and error-detection codes. The third approach enforces particular memory access rights to different SoC nodes via a secure MPU. Finally, as means to mitigate system code injection attacks, an instruction ROM is used to store a basic default code of a processor. The design specifications of the above are detailed below.

3.2.1 Encryption of shared-memory data

Numerous threats like DMA attacks, bus monitoring etc. pose a serious threat regarding the confidentiality of sensitive data. The state of the art protection of sensitive data residing in addressable memory is encryption. In order for encryption to be an effective countermeasure, the algorithm used must meet all the encryption standards. The most suitable encryption algorithm is AES which is resistant to known cipher-text known plain-text attacks and the only way to acquire the original data is by using brute-force methods (i.e., try all the possible keys). Since, AES is very expensive when implemented in software, modern processors include hardware accelerators (i.e, AES-NI), in order to minimize the cost of cryptographic operations. The memory encryption scheme consists of a simple principle, data are decrypted before being used for computation (i.e., loaded to registers) and encrypted when stored back to memory. However, even using the existing hardware cryptographic accelerators by placing AES instructions before loads and stores the execution time of a whole memory encryption protected application is at least tenfold the execution time of the same application running without memory encryption. Thus, a clean-slate approach and a classification method for sensitive memory regions should be considered in order to amortize the massive overhead of this security technique.

3.2.2 Memory-Protection Unit for embedded SoCs

There exist many embedded SoCs which are bare-metal not having any operating system support. That may be motivated by a variety of reasons such as cost, energy-efficiency, and low latency. One such example is the IMD SHARCS use case. Such systems may not demand any sophisticated OS support for their main functionality, however, the lack of an operating system leaves a gap in the memory protection that would normally be covered by memory-access rules imposed by an OS and corresponding mechanisms such as a Translation Lookaside Buffer (TLB) and page tables.

Especially in multi-core embedded systems, all the above cases blur the lines between the responsibilities of the different cores and dictate that there must be a mechanism in place so that data and functionality sharing is done in a controlled and safe manner. To exemplify, we can consider the IMD example depicted in Figure 3.8. SiSC is the core responsible for the communication with external readers, through which data logs are exchanged or even firmware updates are performed. In case SISC becomes compromised by a successful code injection or code reuse attack, given the current state of the system where everything is connected together via a shared bus, the attacker could have access to all system memory and peripherals without compromising the SiMS core. To prevent such scenarios, we propose the use of a Memory Protection Unit (MPU) (Figure 3.9). The MPU will sanitize all shared memory and peripheral read and write requests with a set of predefined and also dynamic access rules enforced by hardware.

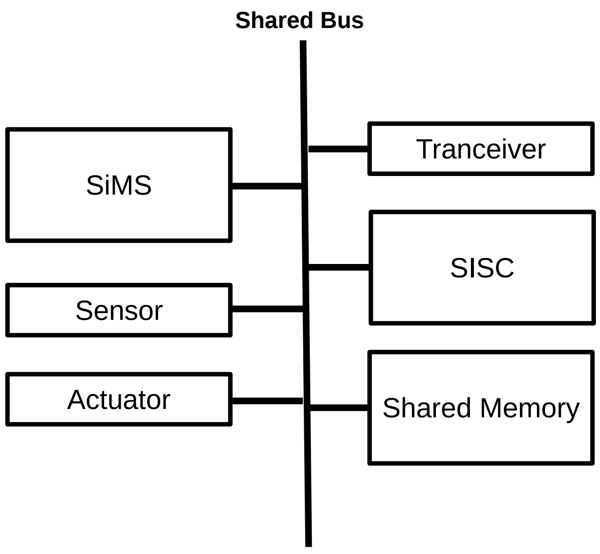

Figure 3.8: IMD baseline system with no memory protection

The MPU will be implemented as a security mechanism along with a crossbar rather than a shared bus to further protect against Denial of Service (DoS) attacks. The crossbar will use fixed priorities, in the IMD case favoring SiMS which runs the main implant functionality. Using the MPU with a crossbar will ensure that even with a compromised part of the SoC, DoS attacks are not possible thought the shared bus.

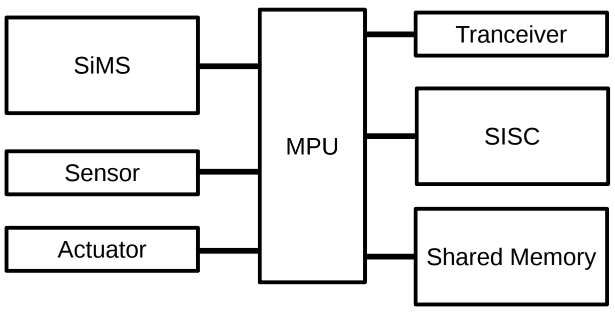

Figure 3.9: IMD baseline system with memory protection unit (MPU)

3.2.2.1 MPU design

The access rules enforced by the MPU will be encoded and will only be updated by authenticated users through the SiSC core during a firmware update, in a similar manner the binary of a core is updated. More precisely, the MPU will implement a basic authentication protocol based on a shared secret key that is not known to the attacker. This means that even if an attacker manages to perform a successful attack on a SoC component, they will be unable to change the memory protection rules. Because of this it is not possible to gain further control of the system and get access to memory regions for which the compromised SoC component has no access rights.

Figure 3.10 shows the MPU protection table, which has the following fields:

- Starting address
- End address
- Permissions for SoC component-1
- Permissions for SoC component-2
- ...
- Permissions for SoC component-X

The *start* and *end* address fields are inclusive and can be the same so that even single memory locations can be specified in the protection table. The permissions for each SoC component are two bit entries that specify read/write attributes for the respective memory ranges. The protection table is integrated to the crossbar that interconnects the SoC components. During each access from any to any component, the memory operation is checked against the protection table. If the access is allowed based on the specified rules then the request is routed to the memory or peripheral target. In the opposite case, where the memory operation is not allowed, the request is not forwarded and a flag is raised indicating that a request to a non-permitted region was attempted. Such flag may be used to notify the requesting component or a central security manager of the SoC.

The size of the protection table depends on the SoC size (number of components) and the granularity of the protected memory regions. For more fine-grain protection the table requires more entries, while for more components it requires wider rows to store their permissions. It is noteworthy that all memory regions (table entries) need to be accessed in parallel for each simultaneous connection. That poses additional complexity in the implementation of the protection table.

Protection Table updates: The protection table may need to be dynamically updated for a number of different use cases related to the application

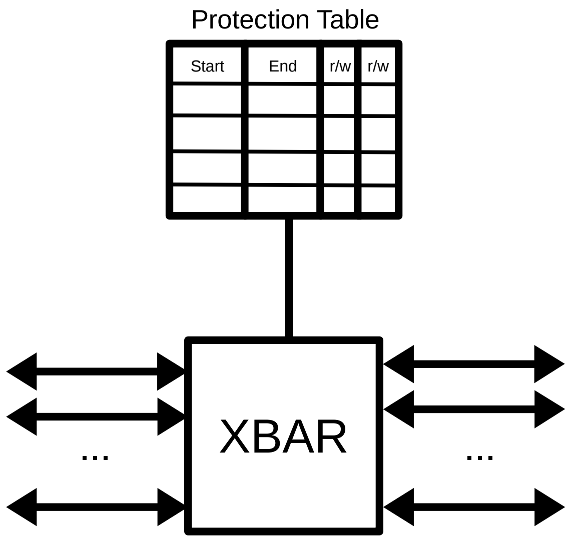

Figure 3.10: MPU Protection Table and crossbar.

of a SoC. In the IMD case, for example, one such scenario could be that a physician wants to have temporary direct control of the actuator to administer some treatment. To facilitate this, the protection table will be itself memory mapped, and updating it should be performed with an additional security protection mechanism. For this reason, the MPU will implement a basic authentication protocol based on a shared secret key that is not known to the attacker. This means that even if an attacker manages to perform a successful attack to some part of the SoC, they will be unable to change the memory protection rules without the secret key. This extra authentication will be implemented in hardware in the MPU. For any change of the protection table, the user will have to provide an authentication token that is based on a shared secret key programmed in the hardware at fabrication time.

3.2.3 Data-Integrity mechanisms

Integrity of data stored in on-chip memory blocks is a very important factor for mission-critical SoCs. This technique provides a hardware mechanism for detecting and correcting unintentional modifications of data stored in a memory block. This mechanism considers that normal memory reads and writes are protected by other security mechanisms as described in this Section (e.g. MPU, memory encryption, or ISR for ensuring secure processor memory accesses). Consequently, this technique protects memory contents against unintentional alteration of data caused by other types of attacks such as Electro-Magnetic Interference (EMI). Such a mechanism should be present in safety-critical systems for fault tolerance reasons as well.

There is a large amount of related work on Error Correcting Codes (ECC) in memory, such as the well-known 2D ECC [23]. However, there is a trade-off between coverage and cost of such codes: the stronger the error detection and correction the higher the additional number of bits required for protecting a memory row.

In this task, data integrity is intended to be applied in small embedded SoC's. Consequently, the overhead of ECC should be low. To balance the cost and the effectiveness of such a technique we will implement a Single Error Correction/Single Error Detection (SECSED) code for on-chip memory blocks. Since data is read and written at 32-bit granularity SECSED codes will be applied for every 32-bit word in the protected memories. Such a protection scheme allows for the detection of single bit errors and the correction of such errors with an area overhead of 6 bits for every 32-bit word which is acceptable for the IMD use case. The computational complexity of SECSED is in the order of a few gate delays which can be easily tolerated by the IMD application since performance is not critical.

The most common implementation of SEC codes is by using parity H-matrix with r rows and n columns [3] [11]. The encoding and decoding circuitry requires four modules: Parity generator, Syndrome generator, Error locater, and Error corrector.

Parity Generator: Each parity check-bit can be constructed by modulo-2 sum of particular data-bits. The encoding process for a (38,32) SEC code requires generating 6 parity check-bits. Placing parity check-bits in power-of-two position makes it possible for the parity generator module to compute the parity bits independent of each other [49]. The parity check-bit generation can be implemented with a 3-deep Xor logic tree. The incurred delay is negligible considering the intrinsic delay of a 3-port Xor gate (i.e., in 28nm ST technology is around 40ps).

Syndrome Generator: On the receiver side, the same parity generator module can be employed to reconstruct the parity check-bits which form a syndrome vector. For a given code and error position, it is a common practice to precompute and store syndromes in a 32x6 syndrome table. If all the bits in the generated syndrome are equal to zero, there has been no fault.

Error locator: If the syndrome generated from the received codeword is not equal to zero, then the syndrome table will locate the position of the error.

Error corrector: Corrects the detected error.

3.2.4 Instruction ROM

Instruction-Set Randomization (ISR) is a technique that can protect against code injection attacks even at extremely resource constrained devices such as an IMD. In full-fledged systems, mitigation of such attacks is provided by the runtime or operating system. However, bare-metal embedded systems that have tight constraints on area and power require a different mechanism for mitigating code injection attack. In order to provide a response to an ISR-detected code injection attempt on a core of an embedded bare-metal SoC, we add an instruction ROM (I-ROM). The I-ROM is preloaded with a default failsafe functionality for the protected core (Figure 3.11) and when an ISR violation is detected, instead of trapping to the operating system as full-systems would do, the protected core is reset and switches to the fail-safe binary stored in the I-ROM (rather than the normal instruction ROM). Switching back to the normal operation can only be achieved through a firmware update. In other words, when the normal instruction memory of the core is updated with a new executable, the core goes back to its full functionality until another possible ISR violation occurs.

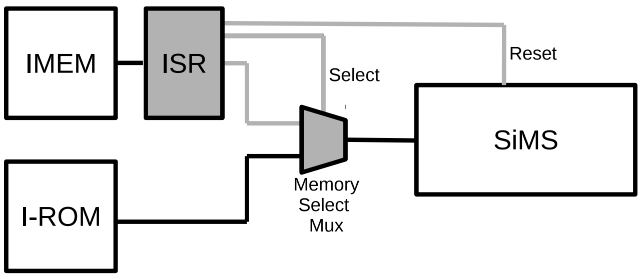

Figure 3.11: IMD instruction ROM

3.3 Secure communication

In T3.4 we work on three techniques for secure communication. The first two are complementary and involve (a) a lightweight, secure data-exchange protocol (and supporting hardware), and (b) a lightweight, secure key-exchange protocol. Both protocols are targeting implantable devices but can – largely – be used in other types of energy-constrained devices. The third one is a hardware-accelerated Network Intrusion Detection. The design specifications of both are described below.

3.3.1 Lightweight, secure-communication protocol

Achieving security for an implant is a challenging endeavor that hinges on various interworking system aspects. Roughly speaking (and adhering to the strategy described in Deliverable D2.1), it can be divided into two parts. The first part, present in every secure system, is to make the system resilient to attacks. The second part – not so common in most other secure systems – pertains to the IMD requirements of being power- and energy-efficient and the need for communication during medical emergencies. In this section we will focus on security protocols suitable for implantable systems, with other (coinciding) hardware and software aspects left for the rest of this Deliverable (D3.2) and Deliverable D4.2.

To justify our protocol design choices, we have to introduce our adversarial model and security threats (recounted from Deliverable D2.1).

3.3.1.1 Adversarial Model

The goal of Adversary A is to obtain leverage over a patient which can be used to A's advantage. We design our security solution for an active adversary, i.e., A has full control of the channel and may eavesdrop, modify, drop and replay messages exchanged between Reader R and Implant I, in addition to forging his own messages. We note that this is a strong assumption (it is unlikely that A is present when R and I are communicating given the limited communication range), yet, we consider IMD security important enough to assume such a worst-case model. Finally, we assume that R is a trusted entity without malicious intentions.

3.3.1.2 Security attributes and attacks

Based on our adversarial model, we next outline the security attributes that should be guaranteed by our security solution and exemplify the security risks without these attributes in place.

- **Confidentiality:** The confidentiality of messages ensures that only R and I can read the messages exchanged between them. Given the strong privacy of the data exchanged between R and I (treatment parameters, private patient data, etc.), any messages exchanged between R and I should be confidential under both US and EU legislation [2, 13]. Without confidentiality (or, if confidentiality is broken through e.g. identifying weaknesses in a cipher), A could eavesdrop on these messages, which compromises sensitive patient data. In turn, the leakage of private patient data could lead to a variety of repercussions for the patient, ranging from *blackmail* to *social segregation* (for instance, if an employer can identify that a potential employee carries an IMD, he might feel inclined not to hire him).

- **Authentication:** Authentication proves that R and I are who they claim they are. Without authentication (or, if it is compromised using e.g. a replay attack) could lead to I believing that A is a trusted entity. In turn, this may give A the ability to make any modifications that R is allowed to make: For example, A could modify the firmware to include ransomware (i.e. prevent treatment until a ransom is paid), or give a deadly shock to the patient [24]. Alternatively, A may pretend to be I and exchange malicious messages with R: For example, A could transmit incorrect/old data-logs to R, leading to a wrongful diagnosis and, indirectly, to a malicious modification of I's firmware. Since spoofing either R or I could result in harm for the patient, *mutual authentication* is required.

- **Integrity:** The integrity of a message ensures that the data exchanged between R and I cannot be changed without R and I noticing these changes. Similar to bypassing authentication, message alternation may be used to inject malicious and potentially dangerous commands to the IMD. The integrity of messages sent by both R and I should, thus, be verifiable.

- **Availability:** Availability pertains to both I being capable of performing its regular functionality and R being able to communicate with I. The often life-critical functionality performed by I requires a high availability throughout its lifetime. We distinguish three types of attacks on I's availability:

  - **Function DoS**: Function Denial-of-Service (DoS) occurs when A blocks the main functionality of I without compromising its confidentiality, authentication or integrity attributes. For instance, if I always serves a communication request whenever it is made, A may prevent I from performing its main functionality by flooding it with (invalid) communication requests.
  - **Battery DoS**: Battery DoS occurs when A depletes the battery of I. For instance, if A repeatedly requests a specific operation from the implant, I will repeatedly run the same authentication protocol for analyzing the request and, eventually, deplete its power source even if the request itself is not valid.
  - **Jam DoS**: This type of DoS occurs when A blocks the communication channel of I by constantly sending it valid or invalid messages. In normal cases, I and R will frequently communicate non-mission-critical information in a protected environment, making jam DoS a non-issue. It may be possible for a A to launch jam-DoS attacks during medical emergencies (if proximal to I and R). In such cases, the only possible solution is to *shield* the

communication between I and R or increase their *signal strength* to be well above that of A. We consider such solutions outside the scope of our work and, as such, assume that a jam-DoS attack shall *never* occur.

3.3.1.3 Protocol Description

In terms of *availability* and without loss of general applicability, the protocol proposed here can ensure **function DoS** use of an IMD system architecture which isolates the security functionality from the main implant functionality by using a dedicated security co-processor (as described as early as Deliverable D2.1). The system-architecture implementation details are given in *Deliverable D5.1* which should be read in conjunction with the protocol discussion in this section.

Another type of DoS, **battery DoS**, is tackled in a different manner: The protocol has been designed so that the mutual-authentication phase (to be elaborated next) is so energy-efficient that it can be done completely on harvested RF energy, if the system (here: the implant) supports it. Only if mutual authentical is successful, does the protocol permit toggling to battery power and proceeding with normal communication. This strategy ensures that no adversary can exhaust the implant battery by mounting continuous authentication requests that will eventually fail, yet threaten to deplete the battery by simply causing the implant run (and drop) authentication checks.

The remaining security attributes (*confidentiality*, *authentication* and *integrity*) of the communication between R and I will be covered through actively designing a *security protocol* that complements our proposed system architecture. There are two approaches to designing a security protocol: (i) design a protocol from scratch; or (ii) design a protocol based on an international standard. The former may lead to novel (e.g., more efficient) solutions, yet might suffer from security vulnerabilities even when a rigorous security analysis is conducted together with the protocol proposal. For instance, over the last years many RFID authentication protocols followed this approach and a significant number of them have been partially or completely broken [47, 53, 7]. The advantage of using standards is that they have been deeply scrutinized by the community before being part of a regulation, which provides some guarantees regarding their security and a lower chance of latent errors [9, 6]. As our primary goal is to propose fully secure IMDs, we have based our security protocol on standards and established international recommendations. Under this approach, the construction of messages is set by the standard(s) but the message content and various extra fields are dependent on the application context. Last but not least, the usage of a standard security scheme for implants has the added benefit of narrowing the gap between research and actual, commercial IMDs.

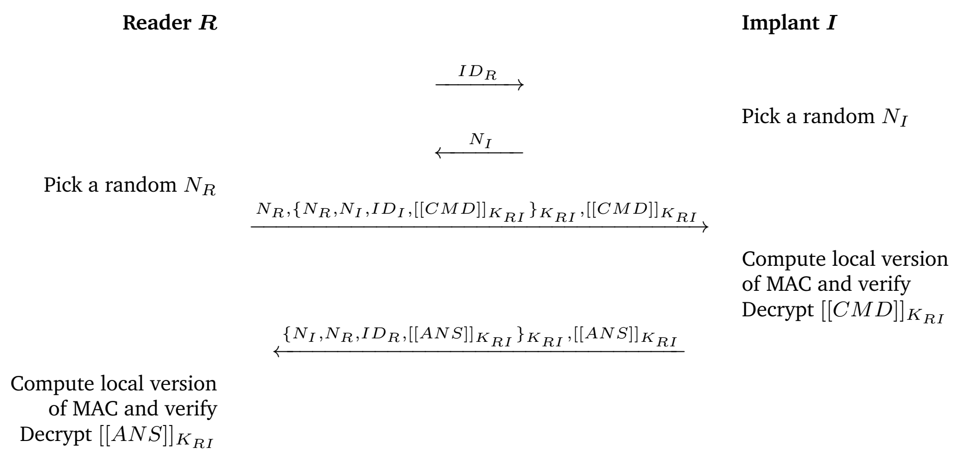
Reader R
Implant I
ID_R
Pick a random N_I
N_I
Pick a random N_R
N_R,$\{N_R,N_I,ID_I,[[CMD]]_{K_{RI}}\}_{K_{RI}}$,$[[CMD]]_{K_{RI}}$
Compute local version of MAC and verify
Decrypt $[[CMD]]_{K_{RI}}$
$\{N_I,N_R,ID_R,[[ANS]]_{K_{RI}}\}_{K_{RI}}$,$[[ANS]]_{K_{RI}}$
Compute local version of MAC and verify
Decrypt $[[ANS]]_{K_{RI}}$

Figure 3.12: The proposed IMD security protocol conforming to ISO/IEC 9798, which guarantees mutual authentication between a reader and IMD. Minimal effort is required from the IMD prior to reader authentication, allowing the IMD to operate on RF-harvested energy.

In the context of medical implants and with the previous system choices in mind, we have drawn final decisions on how our protocol should facilitate the confidentiality, integrity and authentication security attributes. First, we have decided to use symmetric-key cryptography for message encryption (confidentiality) and for Message-Authentication-Code (MAC) (integrity, authentication) calculation. The reason is that symmetric-key cryptography is often regarded as considerably less energy-hungry than public-key cryptography [26, 25]. The main downside of symmetric-key cryptography is that each entity pair (R, I) requires a unique key K_{RI} that should be kept secret by both entities and stored on I. Consequently, as the number of pairs increases, it becomes increasingly harder to distribute these keys, ensure that they are kept secure and, additionally, IMD-storage requirements increase. The assumption is made here that only a small number of valid readers exist: A typical example would include the patient himself, a few close relatives and his treating physician[2]. This leads to a small number of $R - I$ pairs, making symmetric keys a viable (and preferred) option. In this work, we assume that these keys are pre-distributed to the different readers by a manufacturer and do not change over time. In practice, keys could be replaced, invalidated or created by an authorized reader using our security protocol discussed below, which implicitly facilitates such operations.

Our proposed solution is based on ISO/IEC 9798 [18], which specifies six schemes based on symmetric-encryption algorithms that provide various degrees of authentication: unilateral authentication, mutual authentication and authentication with key establishment using a third-entity

[2]For more details, refer to Deliverable D2.1, Section 2.1.3: "User Roles".

(server). Since we identified that mutual authentication is required for IMDs in Section 3.3.1.2, our proposed scheme is based on the fourth (mutual-authentication) protocol specified in the standard and is depicted in Figure 3.12. Here, the entities share a key K_{RI} and its identifiers are ID_R and ID_I respectively, Nx symbolizes a random number generated by entity X, $[[M]]_K$ represents the encryption of message M with key K to provide confidentiality and $\{M\}_K$ symbolizes a message-authentication code (MAC) to provide authentication and integrity through encrypting M with K. In practice, both $[[M]]_K$ and $\{M\}_K$ are calculated using the same symmetric cipher: The difference is that the MAC includes an identifier and random *nonces* (*number-once*: freshly generated unique numbers generated by both R and I) to ensure authentication using a new message (i.e., so as to prevent *replay attacks*). The messages may include a command field (CMD) or answer field (ANS), which dependend on the intended data exchange (e.g., readout data-logs or modify treatment parameters). In detail, our protocol in Figure 3.12 facilitates secure communication through the following steps:

1. $R \rightarrow I$: ID_R. R wakes up I and shares its identifier, which is used by I to choose the correct K_{RI}.

2. $I \rightarrow R$: N_I. I generates a nonce N_I and sends it to R.

3. $R \rightarrow I$: $N_R, \{N_R, N_I, ID_I, [[CMD]]_{K_{RI}}\}_{K_{RI}}, [[CMD]]_{K_{RI}}$. R generates a random nonce N_R and encrypts the command using K_{RI} ($[[CMD]]_{K_{RI}}$) to provide confidentiality. R also computes a MAC to provide both authentication and message integrity, including $[[CMD]]_{K_{RI}}$ and the nonces to ensure that a unique MAC is generated (with high probability) provided that at least one of the entities is legitimate. Finally, R transmits its nonce N_R together with the MAC and an encrypted version of the command to I.

4. $I \rightarrow R$: $\{N_I, N_R, ID_R, ANS\}_{K_{RI}}, [[ANS]]_{K_{RI}}$. I validates if the message has been received correctly and was sent by R by calculating a local version of the MAC and checking its equality to the received value. If this validation fails, the implant aborts the protocol. If not, R has successfully been identified as a trusted entity and the CMD is ready to be processed. At this stage, the SISC is allowed to switch to the battery supply to execute the received command. I first decrypts CMD and, after executing the command, it exchanges the respond command (ANS) in a similar fashion to that in step (3), i.e., using a MAC and encrypting ANS.

5. R receives the message from I and calculates a local version of the MAC. If the received values and the computed values are equal, R and

I are mutually authenticated and the subsequently decrypted ANS is accepted by R. If not, I has failed to authenticate itself (or: message integrity is violated) and its reply is dropped.

3.3.1.4 Protocol Implementation Aspects

We start by fixing the bit length of the values used in the protocol by taking as reference common values used in low-cost RFID tags due to their rough similarities with implants (computational, memory, power-consumption restrictions) [16, 20]. The identifiers are 96 bits long ($|ID_R| = |ID_I| = 96$) and the bit length of random numbers is set to 32 ($|N_R| = |N_I| = 32$) [14]. Commands are 32 bits long ($|CMD| = 32$), as we expect this size to be large enough to cover all command options for the IMD, and answer codes are a multiple of 32 bits ($|ANS| = n \times 32$).

Regarding security primitives, we employ a lightweight and secure symmetric cipher (abbreviated CIPH below). From all possible candidates – and without loss of generality – we chose *MISTY1* which is a 64-bit lightweight block cipher [32] using an 128-bit long key. Our choice is based on a suitability analysis on symmetric ciphers for medical implants [45]. Out of 13 profiled ciphers, MISTY1 has been found best-suited for implants since it ranks high across most metrics such as power consumption, energy cost and encryption speed.

Aiming to reuse the selected primitive (to minimize instruction-memory overheads), we also use a MAC algorithm that is based on this symmetric-key block cipher (cipher-based MAC or CMAC). Our algorithm follows the NIST 800-38B Recommendation [30], providing guidelines for using block ciphers for our purposes. The sub-keys used in the cipher are generated according to the specification (NIST 800-38B; pages 7-8 [30]) and are generated and stored in the memory of the entities involved (e.g., reader and implant) at the key-distribution phase. Given the block size of 64 bits, a message M is divided into blocks of 64 bits for computing the MAC $T = \{M\}_K$, i.e., $M_1||M_2||\cdots M_m$, where $m = |M|/64$. The CMAC algorithm is described below:

CMAC Algorithm

```
1. $C_0 = 0$
2. For $i = 1$ to $m$, let $C_i = CIPH_K(C_{i-1} \oplus M_i)$.
3. $T = C_m$
4. Return $T$
```

Apart from the MAC, random numbers are used in our protocol. As specified in the NIST 800-38A specification [29] ("Recommendation for Block-Cipher Modes of Operation"), we use our block cipher in a counter mode, called CTR, to generate a fresh nonce. In CTR mode, a 64-bit pseudorandom number (O_j) is produced by encrypting a counter value (j) with any key K_{RI}. From this 64-bit number, we use the least-significant 32-bits as a nonce (N_j). The algorithm proposed is described below:

Block cipher - CTR Mode
1. $O_j = CIPH_k(j\,{+}{+})$
2. $N_j = \vert O_j \vert_{0 \cdots 31}$

The secure-communication protocol has, finally, been implementation in ANSI C and can be successfully compiled by the SISC C-compiler. It is made available as a SHARCS technique for lightweight communication. More details on the protocol can be found in [36, 38-41].

3.3.2 Lightweight, Secure Key-Exchange Protocol

The security protocol detailed in Section 3.3.1, above, provides secure communication between a trusted reader R and implant I based on a symmetric key K_{RI} *already* shared between them. An external reader R can, therefore, only communicate with I if a key K_{RI} is shared between the two. During emergencies, however, R and I are likely unknown to each other and therefore do not share a secret key. Our goal is, then, to transfer K_{RI} from I to R in a secure and trustworthy manner, which subsequently allows R to communicate with I using its regular security protocol (e.g. the one presented in Section 3.3.1). To do so, R and I first need a way to *establish trust* between them, after which key exchange ensues. The protocol described in this section addresses exactly this trusted key exchange between entities.

One promising way to facilitate this exchange is to use the cardiac Inter-Pulse Interval (IPI), that is, the time difference between two consecutive heartbeats [37, 34, 52, 33] as a Pseudo-Random-Number Generator (PRNG)[3]. Each IPI contains a number of random bits, which may only be obtained (with some consistency) by R and I if they simultaneously measure a cardiac signal from the same person. These characteristics essentially make IPIs time- and person-specific random numbers, which allows IPIs to be used for entity authentication [34, 37] or, in this case, key exchange [4, 10, 27]. In the latter case, I (generates and) commits K_{RI} using its own IPIs; R may subsequently decommit K_{RI} using R's IPIs. Our protocol distinguishes itself from related work through a novel way of overcoming *heartbeat misdetection* which is crucial for proper IPI-based random-number generation. It does so based on heartbeat classification and ignoring any IPIs which have been affected by misdetection, while at the same time providing an low-energy

[3]For details on random-number generation based on IPIs, refer to Appendix A, Section A.1.

solution suitable for implants (more details given below). Most studies related to IPI-based security do not consider heartbeat misdetection. However, a recent, preliminary study has shown that it may have a significant effect on inter-witness disparity [37][4].

3.3.2.1 Key Exchange through Fuzzy Commitment

During trust establishment, external reader R and IMD I use IPIs to derive freshly generated, random numbers called *witnesses* w_R and w_I, respectively, where $w_R \approx w_I$. This makes them suitable for both *entity authentication* and *confidentiality* purposes. With these properties in mind, we could try to exchange a symmetric key K_{RI} by first having I encrypt it using w_I in a similar fashion to a one-time pad (OTP) [52], i.e., using a simple xor operation ($K_{RI} \oplus w_I$). If $w_R = w_I$, R may subsequently decrypt K_{RI} using w_R (as $K_{RI} \oplus w_I \oplus w_R = K_{RI}$).

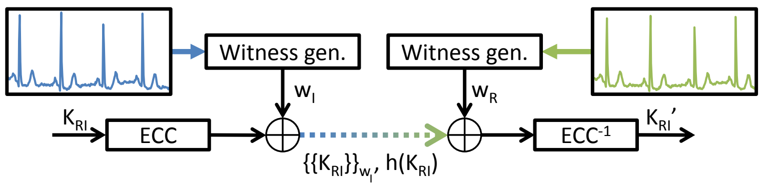

Figure 3.13: Key exchange using fuzzy commitment and heartbeats.

Unfortunately, it is common in biometrics that $w_R \approx w_I$, i.e., $w_R \neq w_I$, which prohibits R from successfully decrypting K_{RI} using only a xor operation. We can employ a so-called *fuzzy-commitment* scheme, as illustrated in Figure 3.13. This involves applying error-correcting codes (ECC) to K_{RI} prior to encryption [22] in order to achieve robust matching of witnesses. That is, I commits K_{RI} as $\{\{K_{RI}\}\}_{w_I} = ECC(K_{RI}) \oplus w_I$, where $\{\{x\}\}$ is the commitment of x. On its own, this commitment provides *entity authentication* and *confidentiality*.

To also facilitate *data integrity*, I also calculates a cryptographic hash of K_{RI} ($h(K_{RI})$), and sends the entire message to R ($I \to R : \{\{K_{RI}\}\}_{w_I}, h(K_{RI})$). R may subsequently obtain K'_{RI} through the inverse process of commitment, where $K'_{RI} = K_{RpI}$ iff $w_R \approx w_I$. To validate the correct exchange of K_{RI}, R hashes its decoded key $h(K'_{RI})$ and compares it to $h(K_{RI})$, which match iff

[4]The inter-witness disparity is determined both by the inter-sensor variability and the probability of each entity (I or R) detecting each heartbeat correctly. In turn, *inter-sensor variance* is the variance between two different sensor measurements of cardiac biosignals, caused by the variable pulse-transition time of ventricular contraction (heartbeats) to the rest of the body due to, for example, motion and pressure differences. *Incorrect heartbeat detection* (the detection of a non-existing heartbeat or not-detecting an actual heartbeat) results from, among others, sensor-movement artifacts or imperfections in the detection algorithm.

$K'_{RI} = K_{RI}$. In our protocol, we make use of this fuzzy-commitment strategy but we also demonstrate that fuzzy commitment (on its own) *cannot* tolerate the disparity resulting from heartbeat misdetection. We tackle this problem by ignoring IPI blocks which have been affected by misdetection during witness generation.

3.3.2.2 Adversarial Model

The adversarial model used is based on that described originally in Deliverable D2.1 and recounted here with specific references to the problem currently addressed. The goal of an adversary A is to obtain either K_{RI}, w_R or w_I such that he can either gain access to I or obtain private information from the (secure) communication between R and I. Our protocol is designed for an active adversary who has full control of the channel and may eavesdrop, modify, drop and replay messages sent from R to I and vice versa, in addition to forging his own messages. While this is a rather strong assumption (it is unlikely that A is present during an actual emergency), we consider IMD security important enough to assume such a worst-case model. Despite his capabilities, A is not able to measure the IPI bits which are used for witness generation: This assumption is supported by related work which has shown that remote-measuring techniques are not able to provide a statistical advantage over mere guessing [34].

3.3.2.3 Trust Establishment

We aim to establish trust between R and I through using IPIs for generating random witnesses w_R and w_I, providing a basis for trust establishment. The trust formed between R and I hinges on the assumption that $w_R \approx w_I$ iff R is physically proximal to I (i.e., capable of touching the patient), which is a common assumption for emergency-trust establishment [35]. Logically, an adversary A *could* try to gain access to the IMD by generating a witness $w_A \approx w_I$ using the same method as R. However, we expect that the risk of abusing this mechanism is minimal, as: 1) It is unlikely that a patient would not notice (or allow) A attaching a heartbeat sensor to him/her; 2) Sensors have to be fastened steadily to the patient for successful trust establishment, as there would otherwise be a significant disparity between the generated witnesses; and 3) In the unlikely event that A can satisfy both (1) and (2), it is reasonable to expect that A would have easier methods of harming the patient (e.g., physically attack the patient or use social engineering to obtain his password). That is, IMD security may not be deemed as the most crucial component for the patient's well-being in such cases.

The performance and overheads of our key-exchange protocol and the size of w_R and w_I depend largely on the expected inter-witness disparity, as a larger disparity requires stronger error-correcting codes for fuzzy commitment. To understand some of our design choices, we first exemplify the two causes of inter-witness disparity using Figure 3.14, which depicts the IPIs generated by R and I. Also highlighted in the figure are the following: 1) VAR_{IS}: When R and I obtain an IPI from the same heartbeats, a minor disparity between these IPIs occurs as R and I detect each heartbeat with slight variations due to inter-sensor variability VAR_{IS}; and 2) *Heartbeat misdetection*: It can happen that R or I fails to detect a heartbeat (or detects a fake heartbeat) due to, for example, movement artifacts or imperfections in the detection algorithm. In the example of Figure 3.14, R has failed to detect heartbeat 4, resulting in the following effects: i) R incorrectly bases IPI$_{(3,4)}$ on heartbeats 3-5, resulting in a value considerably different than its other IPIs and I's IPI$_{(3,4)}$; and ii) The remaining IPIs used for witnesses generation are misaligned as R has generated one less IPI using heartbeats 3-5 than I. That is, heartbeat misdetection introduces *order variance* between w_R and w_I.

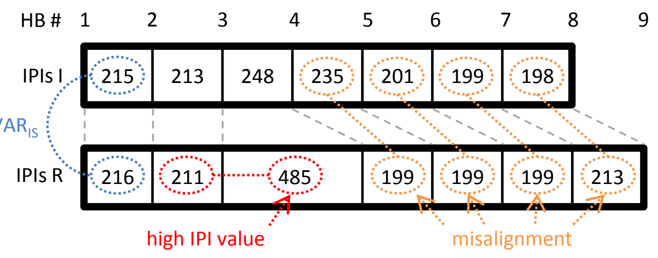

Figure 3.14: Illustration of inter-witness disparity, showing the IPIs obtained by R and I (in decimals) and annotations highlighting the disparity. In this example, R does not detect heartbeat 4.

Fuzzy commitment can tolerate the random-bit permutations stemming from VAR_{IS} effectively using error correcting codes, such as BCH codes [22]. A known limitation of the scheme, however, is that it is not capable of dealing with order variance which, in our case, is introduced by heartbeat misdetection[5]. As such, we have to ensure that R and I use the same IPIs for witness generation. To do so, we opt for having R and I first determine if a misdetection has occurred in a block of IPIs during witness generation and, if so, both entities replace the entire block for freshly obtained IPIs. While this process is further explained in Section 3.3.2.4, we first introduce here a proof-of-concept classification algorithm used by R and I to identify misdetections.

Our classification algorithm in Listing 3.1 essentially employs a double-thresholding mechanism to distinguish correct and misdetected IPIs by con-

[5]We could opt to use an order-invariant derivative of fuzzy commitment (called fuzzy vault [21]). However, as shown in [40], this scheme would be too resource-heavy for IMD application.

Listing 3.1: Heartbeat-classification algorithm (pseudo code).

```
Input: IPIb #block of IPIs
       Th_l, Th_u #classification thresholds
Output: m #misdetection flag

m = 0;
IPIm = mean(IPIb);
for i = 0: len(IPIb):
  if IPIb[i] < IPIm * Th_l or IPIb[i] > IPIm * Th_u:
    m = 1;
    return;
```

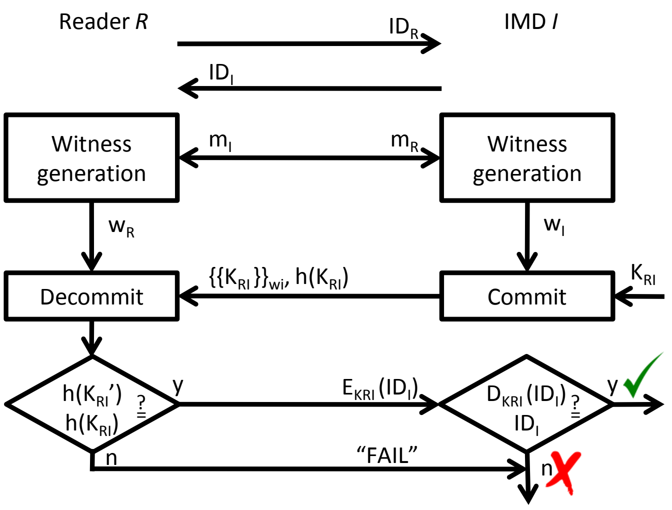

Figure 3.15: Key-exchange protocol.

sidering the substantially higher (or lower) IPI values resulting from it. The algorithm first calculates the mean of a block of IPIs and subsequently compares each IPI to this mean value and two thresholds, lower threshold Th_l ($Th_l < 1$) and upper threshold Th_u ($Th_u > 1$), to distinguish between detecting a fake and missing a real heartbeat, respectively. If a block of IPIs contains a misdetection for either R or I, they respectively set their misdetection flag (m_R or m_I, resp.) to 1. We expect that the simplicity of our algorithm favors the tight energy budget of IMDs. It should be noted that our algorithm – as with any classification algorithm – may result in both a number of false positives (i.e., a regular heartbeat is considered misdetected) and false negatives (i.e., a misdetection is not identified). These classification rates are a function of the IPI-block size, Th_l and Th_u.

3.3.2.4 Protocol Description

Our key-exchange protocol is depicted in Figure 3.15 and comprises the following four steps:

S1. *Initialization*: R and I initiate the key-exchange session. R first sends its identifier to start the session with I ($R \rightarrow I : ID_R$) after which I replies with its own identifier ($I \rightarrow R : ID_I$). These identifiers are used for binding the key K_{RI} to identifiers ID_I and ID_R upon successful exchange, allowing R and I to communicate using Is regular security protocol.

S2. *Witness generation*: R and I generate witnesses w_R and w_I from their respective heartbeat measurements following the methodology described in Section 3.3.2.3. To generate these witnesses, R and I simultaneously obtain a block of IPIs and classify if a misdetection has occurred: If so, they set their respective misdetection flags (m_R and m_I) to 1. These flags are subsequently exchanged ($R \rightarrow I : m_R$ and $I \rightarrow R : m_I$) and, if a misdetection has occurred ($m_R \vee m_I = 1$), both entities replace the block with fresh IPIs. R and I resynchronize after each block (using m_R and m_I) to prevent a misdetection from affecting multiple blocks [37]. This process is repeated until enough IPIs are obtained to generate w_R and w_I. While w_R and w_I are transmitted in plain text, this does not provide an adversary A with an advantage. First, overhearing m_I or m_R does not provide any useful information, except that a certain block of IPIs is not used. As this block is replaced with fresh and random IPI bits, the secrecy of the fuzzy commitment is not threatened. Secondly, if A were to modify, drop, replay, delay or insert its own m_R and/or m_I, the net effect would be that R and I either do not agree on which IPIs to use (i.e., introduce order variance between w_R and w_I) or continuously drop all IPIs. The security of our protocol hinges on the randomness of w_I, which is not affected by using different IPIs from I and/or R [37]. As such, malicious m_R or m_I do not provide A with any insights into w_R, w_I or K_{RI} (exchanged in the following protocol step).

S3. *Fuzzy commitment*: I generates a random secret key K_{RI} and commits it using w_I and fuzzy commitment. As both K_{RI} and w_I are fresh and random, fuzzy commitment facilitates perfect secrecy [52]. This commitment is sent to R ($I \rightarrow R :< \{\{K_{RI}\}\}_{w_I}, h(K_{RI}) >$). R subsequently decommits K'_{RI} using w_I and the inverse process of commitment. As in the original fuzzy commitment scheme, we employ BCH codes as they provide strong error-correcting capabilities to random the bit flips resulting from VAR_{IS} [22].

S4. *Key validation*: To validate the correct decommitment of K_{RI}, R computes its own hash $h(K'_{RI})$ and compares it to the hash received from I in its fuzzy commitment ($h(K_{RI}) \overset{?}{=} h(K'_{RI})$). If these hashes match, R encrypts ID_I using K_{RI} and its regular cipher and sends this to I ($R \rightarrow I :< E_{K_{RI}}(ID_I) >$). The security of this step assumes that A cannot obtain K_{RI} from this encryption even if ID_I is known, which is the case for modern ciphers. I subsequently decrypts ID_I using its cipher and K_{RI} and, if the decrypted $D_{K_{RI}}(ID_I) = ID_I$, key exchange is a success. Alternatively, if either $h(K_{RI}) \neq h(K'_{RI})$ or $D_{K_{RI}}(ID_I) \neq ID_I$, the key exchange has failed. This step guarantees mutual trust, as both R and I (implicitly) validate if $w_R \approx w_I$.

3.3.2.5 Protocol Implementation Aspects

In line with the security protocol proposed in Section 3.3.1, ID_R and ID_I are chosen to be 96 bits long. As K_{RI} is used in subsequent communication between R and I using I's regular protocol, it has to adhere to the key-length requirements of I's regular cipher. We consider the PRESENT-80 cipher well-suited for IMD cryptography, given its minimal energy footprint [8] and, as such, K_{RI} is chosen to be 80 bits long. Moreover, as PRESENT-80 has a block size of 64 bits, the encryption $E_{K_{RI}}(ID_I)$ uses the 64 least-significant bits of ID_I. The expected disparity between the IPIs obtained by R and I determines the length of the witnesses BCH codes in fuzzy commitment. Our proposed protocol uses 3 bits per IPI to generate 204-bit long witnesses and (204, 80, 37) BCH codes (a shortened version of the (255, 131, 37) BCH code). That is, the 80-bit long K_{RI} is encoded using 204 bits, which creates a Hamming-distance of 37 bits between code words. Finally, for hashing K_{RI} we rely on SHA-3, a recommended, collusion-resistant hash function [31], which emits a 224-bit long hash. This sufficiently protects the secrecy of K_{RI}, given its size and randomness.

3.3.3 FPGA-based NIDS

Server farms hosting Cloud computing require to filter incoming traffic and detect malicious incoming packets. The most effective mechanism for network monitoring is Network Intrusion Detection Systems (NIDS) [28]. The main task of a NIDS is to defend the underlying cloud network from network intrusions by detecting an attack as soon as possible based on some predefined signature rules and raising an alarm to the system administrator.

An NIDS uses a set of signature rules, each one describing a network packet suspicious for an network intrusion. Each rule describes the header and/or payload content of a malicious packet. Every incoming network packet is compared against each NIDS rule and in case of a match it is reported. NIDS processing is highly parallel as each incoming packet is compared against every NIDS rule. It further needs to be performed in line rates in order to scan all incoming traffic. Finally, the latency of the processing is important in order to report detected intrusions as soon as possible.

The hardware specifications of the FPGA-based NIDS is as listed below.

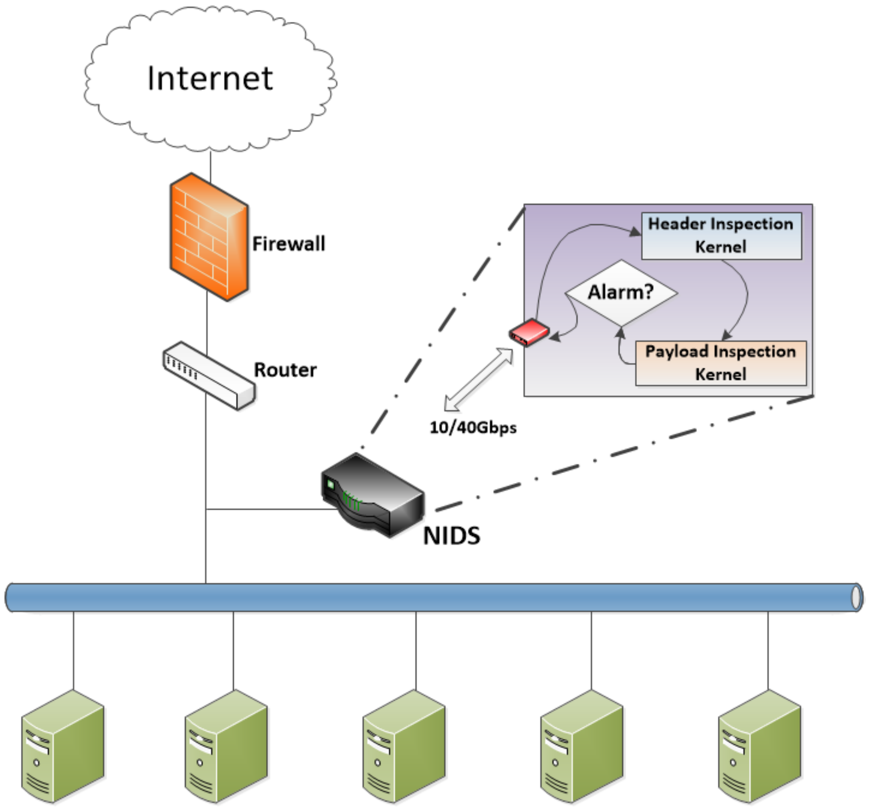

Figure 3.16: Maxeler DFE based NIDS

- **Implementation platform & NIDS kernels**: Our NIDS design, shown in Figure 3.16, is targeting a Maxeler-based Data Flow Engine (DFE) which uses reconfigurable hardware (an Field Programmable Gate Array (FPGA) device) with a direct network link operating at a line rate of 10Gbps. The DFE will be designed to have a manager instance that orchestrates the data movement between the different kernel (hardware) blocks and will be responsible for receiving and transmitting packets across the network interface. The two kernel blocks in the DFE are the following:
  - **Header inspection kernel**: This kernel compares the header description of each NIDS rule against the header of every incoming packet and reports the matches.

  - **Payload inspection kernel**: It performs deep content inspection checking for matches in the packets' payload against the predefined payload signatures. In particular, this kernel scans the payload contents of each incoming packet searching for specific payload patterns described in the NIDS rules.

  In case the kernels detect a match between an incoming packet and one or more signatures[6], the system administrator is alarmed about the attempted network intrusion.

- **Low latency and high throughput**: The targeted Maxeler platform provides direct network connection to the DFE (FPGA) allowing ultra low latency line-rate processing of 10Gbps network traffic. This is critical for the NIDS operation as the intrusion should be detected as soon as possible in order to lower the damage caused to the underlying cloud network.
- **Placement**: As shown in Figure 3.16, the NIDS is placed in between the firewall and the various nodes in the cloud network. This placement allows the NIDS to behave like a security camera, continuously monitoring each packet that flow to and from the cloud network and alarming the system administrator as soon as an intrusion is detected.
- **Signature-based intrusion detection**: In the cloud computing use case, signature based intrusion detection defines a set of rules, against which the incoming traffic is compared against to detect an attack. For example, the open source Snort rule database [41] can be used to monitor the traffic. A rule refers to both the header as well as the payload of the packet and can contain fields that can specify the protocol, IP address, port and content of a suspicious packet. Header rules check for equality (or range) in numerical fields. More computationally intensive is the text search of the packet payload against hundreds of patterns that must be performed at wire-speed [42]. DFEs are well suited for this application as the ultra fine grained and pipelined parallelism of the data flow architecture can be exploited to achieve high throughput. Moreover, the tight integration of network interface, memory, and compute resources may support low latency network processing.

[6]A signature rule matches, when both its header and payload part(s) of the rule have matched for a particular incoming network packet.

A Appendix

A.1 IPI-Based Random-Number Generation

IPIs are a cardiovascular biometric which is defined as the time difference between two consecutive heartbeats. Most studies conclude that the 4 least-significant bits (LSBs) of each IPI can be considered random [34, 37, 51, 38, 39]. Despite this randomness, R and I may both obtain the same IPI bits (with some consistency – minor disparities are common in biometrics) iff they simultaneously measure the same heartbeats on the same body. These characteristics allow R and I to use IPIs for generating random witnesses w_R and w_I (where $w_R \approx w_I$), providing a basis for trust establishment. The most common way of generating these witnesses (used in [5, 33, 37, 38, 39]) is depicted in Figure A.1: First, a number of heartbeats are detected from a cardiac biosignal and the time interval between consecutive heartbeats is calculated to form IPIs, i.e., $IPI_{(i,i+1)} = beat_{i+1} - beat_i$. A predefined set of (random) bits is selected from each IPI: The most-significant bits (MSBs) are typically discarded due to their inherent low entropy, while the least significant bits (LSBs) may be discarded due to inter-sensor variability (VAR_{IS}[1]). Gray coding is applied to the selected IPI bits in order to strengthen them against VAR_{IS} (reducing the number of bits affected by a small disparity between IPIs). Finally, the bits from consecutive IPIs are Gray-coded to increase information robustness and are concatenated to form a so-called *witness*.

[1]Assuming precise and non-drifting sensors, VAR_{IS} is the variance between two different sensor measurements of cardiac biosignals, caused by the variable pulse-transition time of ventricular contraction (heartbeats) to the rest of the body due to, for example, motion and pressure differences.

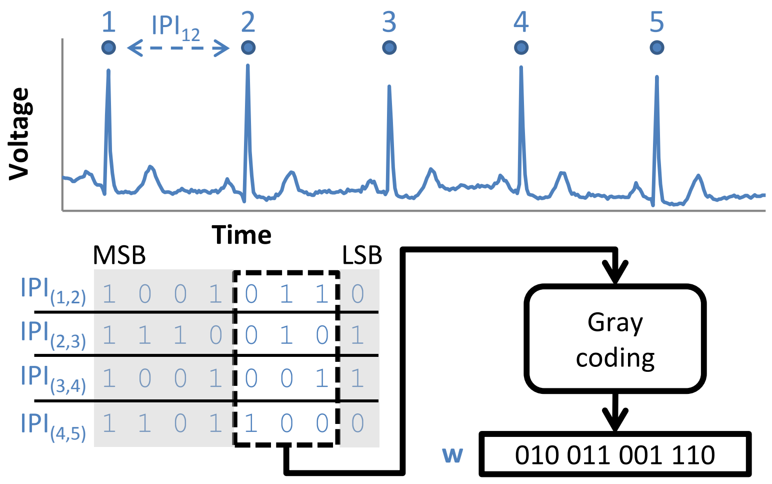

Figure A.1: Witness generation.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Accountability Act. Health insurance portability and accountability act of 1996. *Public Law*, 104:191, 1996.

[3] P. P. Ankolekar, R. Isaac, and J. W. Bredow. Multibit error-correction methods for latency-constrained flash memory systems. *IEEE Transactions on Device and Materials Reliability*, 10(1):33–39, March 2010.

[4] S.-D. Bao et al. A novel key distribution of body area networks for telemedicine. In *IEEE BioCas*, pages 1–17, 2004.

[5] S.-D. Bao et al. Using the timing information of heartbeats as an entity identifier to secure body sensor network. *T-ITB, pp. 772-779*, 12(6), 2008.

[6] A. Bauer and J. Juerjens. Security protocols, properties, and their monitoring. In *Proceedings of the fourth international workshop on Software engineering for secure systems*, SESS '08, pages 33–40, New York, NY, USA, 2008. ACM.

[7] E. A. Bogari, P. Zavarsky, D. Lindskog, and R. Ruhl. An analysis of security weaknesses in the evolution of RFID enabled passport. In *Internet Security (WorldCIS), 2012 World Congress on*, pages 158–166, Ontario, Canada, June 2012.

[8] A. Bogdanov et al. Present: An ultra-lightweight block cipher. In *CHES*, pages 450–466. Springer, 2007.

[9] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Springer Publishing Company, Incorporated, 3rd edition, 2010.

[10] F. M. Bui and D. Hatzinakos. Biometric methods for secure communications in body sensor networks: resource-efficient key management and signal-level data scrambling. *EURASIP Journal on Advances in Signal Processing*, 2008:109, 2008.

[11] S. Cha and H. Yoon. Efficient implementation of single error correction and double error detection code with check bit pre-computation for memories. *JSTS: Journal of Semiconductor Technology and Science*, pages 418–425, March 2012.

[12] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49. ACM, 2016.

[13] E. Directive. 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the EC*, 23(6), 1995.

[14] EPCglobal Inc. Class-1 Generation-2 UHF RFID protocol for communications at 860 MHz - 960 MHz. Standard, v. 1.2.0, 2008.

[15] K. Fernald, T. Cook, T. M. III, and J. Paulos. A microprocessor-based implantable telemetry system. In *IEEE Computer*, volume 24, pages 23–30, Mar. 1991.

[16] D. Halperin, T. S. Heydt-Benjamin, K. Fu, T. Kohno, and W. H. Maisel. Security and Privacy for Implantable Medical Devices. *IEEE Pervasive Computing*, 7:30–39, January 2008.

[17] C. Harrigal and R. Walters. The development of a microprocessor controlled implantable device. In *IEEE Proceedings of the 1990 Sixteenth Annual Northeast Bioengineering Conference*, pages 137–138, Mar. 1990.

[18] ISO. Information technology – Security techniques – Entity authentication – Part 2: Mechanisms using symmetric encipherment algorithms, ISO/IEC 9798-2:2008. International Standard, 2nd ed., 1999.

[19] E. Jalilian, L. Turner, G. Jullien, and M. Mitchev. Design of an implantable multichannel neurostimulator for restoring impaired gastrointestinal motility. In *Proceedings of the 9th Annual Conference of the International FES Society*, Sept. 2004.

[20] A. Juels. RFID Security and Privacy: A Research Survey. *IEEE Journal on Selected Areas in Communications*, 24(2):381–394, February 2006.

[21] A. Juels and M. Sudan. A fuzzy vault scheme. *Designs, Codes and Cryptography*, 38(2):237–257, 2006.

[22] A. Juels and M. Wattenberg. A fuzzy commitment scheme. In *ACM CCS*, pages 28–36, 1999.

[23] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 197–209, 2007.

[24] J. Kirk. Pacemaker hack can deliver deadly 830-volt jolt. *Computerworld*, 17, 2012.

[25] J. Lee, K. Kapitanova, and S. H. Son. The price of security in wireless sensor networks. *Computer Networks*, 54(17):2967–2978, 2010.

[26] S. D. Martin Feldhofer and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *Cryptographic Hardware and Embedded-Systems*, pages 85–99. Springer, 2004.

[27] F. Miao et al. A modified fuzzy vault scheme for biometrics-based body sensor networks security. In *IEEE GLOBECOM*, pages 1–5, 2010.

[28] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan. Review: A survey of intrusion detection techniques in cloud. *J. Netw. Comput. Appl.*, 36(1):42–57, Jan. 2013.

[29] NIST. Recommendation for block cipher modes of operation – methods and techniques , NIST special publication 800-38a. National Institute of Standards and Technology, 2001.

[30] NIST. Recommendation for block cipher modes of operation: The CMAC mode for authentication, NIST special publication 800-38b. National Institute of Standards and Technology, 2005.

[31] NIST. Fips 202: Sha-3 standard: Permutation-based hash and extendable-output functions. *Federal Information Processing Standards Publication*, 2015.

[32] H. Ohta and M. Matsui. A description of the misty1 encryption algorithm. *RFC2994, November*, 2000.

[33] C. C. Poon et al. A novel biometrics method to secure wireless body area sensor networks for telemedicine and m-health. *IEEE Commun. Mag.*, pages 73–81, 2006.

[34] M. Rostami et al. Heart-to-heart (h2h): authentication for implanted medical devices. In *ACM CCS*, pages 1099–1112, 2013.

[35] M. Rushanan et al. Sok: Security and privacy in implantable medical devices and body area networks. *Proceedings of the IEEE S&P*, pages 529–539, 2014.

[36] R. M. Seepers. *Implantable Medical Devices: Device security and emergency access*. PhD thesis, Erasmus University Medical Center, Rotterdam, Netherlands, December 2016.

[37] R. M. Seepers et al. Peak misdetection in heart-beat-based security characterization and tolerance. *IEEE EMBC*, 2014.

[38] R. M. Seepers, C. Strydis, I. Sourdis, and C. De Zeeuw. Enhancing heart-beat-based security for mhealth applications. *IEEE journal of Biomedical and Health Informatics (JBHI)*, –(-):1–9, 2015.

[39] R. M. Seepers, C. Strydis, I. Sourdis, and C. De Zeeuw. On using a von-neumann extractor heart-beat-based security. In *Security and Privacy in Computing and Communications (TrustCom), 2015 14th IEEE International Conference*, pages 1–8. IEEE, 2015.

[40] R. M. Seepers, J. H. Weber, Z. Erkin, I. Sourdis, and C. Strydis. Secure key-exchange protocol for implants using heartbeats. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF ’16, pages 119–126, New York, NY, USA, 2016. ACM.

[41] SNORT. Snort official web site: https://www.snort.org.

[42] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In P. Y. K. Cheung and G. A. Constantinides, editors, *Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings*, pages 880–889. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[43] L. Stotts, K. Infinger, J. Babka, and D. Genzer. An 8 bit microcomputer with analog subsystems for implantable biomedical application. In *IEEE Journal of Solid-State Circuits*, pages 292–300, 1989.

[44] C. Strydis, R. M. Seepers, P. Peris-Lopez, D. Siskos, and I. Sourdis. A system architecture, processor, and communication protocol for secure implants. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):57, 2013.

[45] C. Strydis, D. Zhu, and G. Gaydadjiev. Profiling of symmetric encryption algorithms for a novel biomedical-implant architecture. In *ACM*

International Conference on Computing Frontiers (CF'08), pages 231–240, Ischia, Italy, 5-7 May 2008.

[46] P. Valdastri, A. Menciassi, A. Arena, C. Caccamo, and P. Dario. An implantable telemetry platform system for in vivo monitoring of physiological parameters. In *IEEE Transactions on Information Technology in Biomedicine*, volume 8, pages 271–278, September 2004.

[47] T. van Deursen and S. Radomirovic. Attacks on rfid protocols. *IACR Cryptology ePrint Archive*, 2008:310, 2008.

[48] L. Wang, P. Hammond, E. Johannessen, T. Tang, A. Astaras, S. Beaumont, A. Murray, J. Cooper, and D. Cumming. An on-chip programmable instrumentation microsystem for gastrointestinal telemetry applications. In *Proceedings of the 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS)*, pages 2109–2112, San Francisco, California, USA, 1-5 September 2004.

[49] H. S. Warren. Hackers delight. 2nd., 2012.

[50] P. Wouters, M. D. Cooman, D. Lapadatu, and R. Puers. A low power multi-sensor interface for injectable microprocessor-based animal monitoring system. In *Sensors and Actuators A: Physical*, volume 41-42, pages 198–206, 1994.

[51] G.-H. Zhang et al. Analysis of using interpulse intervals to generate 128-bit biometric random binary sequences for securing wireless body sensor networks. *T-ITB*, 16(1):176–182, 2012.

[52] G. Zheng, G. Fang, R. Shankaran, and M. A. Orgun. Encryption for implantable medical devices using modified one-time pads. *Access, IEEE*, 3:825–836, 2015.

[53] X. Zhuang, Z.-H. Wang, C.-C. Chang, and Y. Zhu. Security analysis of a new ultra-lightweight RFID protocol and its improvement. *Journal of Information Hiding and Multimedia Signal Processing*, 4(3), July 2013.