KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Lightweight Roots of Trust for Modern Systems-on-Chip

Pieter Maene

Supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

October 2019

Lightweight Roots of Trust for Modern Systems-on-Chip

Pieter MAENE

Supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Examination Committee:
Prof. dr. Adhemar Bultheel
Chair
Prof. dr. ir. Yves Willems
Chair
Prof. dr. ir. Frank Piessens
Prof. dr. ir. Marian Verhelst
Prof. dr. ir. Bjorn De Sutter
UGent, Belgium
Prof. Dr.-Ing. Felix Freiling
FAU Erlangen-Nürnberg, Germany

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

October 2019

Uitgegeven in eigen beheer, Pieter Maene, Kasteelpark Arenberg 10 bus 2452, B-3001 Leuven (Belgium)

§ Preface

Ten years ago, I arrived in Leuven to study at this university and obtain my engineering degree. The week before, I had joined the student organisation's welcome weekend, where I not only met some truly amazing people that would become my friends, but also remember saying that I would never consider getting a PhD. However, I believe I can now say with some confidence that eighteen-year-old me did not really know what he was talking about, and am happy to have decided differently when I graduated.

First and foremost, I therefore want to thank Prof. Ingrid Verbauwhede for extending me this incredible opportunity, and allowing us to freely explore our ideas, readily giving direction when we need it. Over the past five years, I have learned more about cryptography and software security, computer architectures, and the technologies enabling these, than I could ever have imagined.

I also want to express my sincere gratitude to Prof. Felix Freiling for his thoughtful insights, the many invitations to visit his chair in Germany, and the warm welcome every time we were there. My thanks also go out to Prof. Frank Piessens for his close collaboration, considering our solutions from a different perspective and sharing his expertise. I am also grateful to the additional members of my jury, Prof. Bjorn De Sutter and Prof. Marian Verhelst, for their valuable feedback. Finally, I want to thank the chairs of my preliminary and public defences, Prof. Yves Willems and Prof. Adhemar Bultheel.

None of this would have been possible without the many incredible co-authors I had the privilege of working with, and I am thankful to all of you for your experienced help and for looking at my proposals with a critical eye. Additionally, I want to thank our trusted computing colleagues from DistriNet for the uniquely interactive CosiNet meetings, which therefore consistently overran. I am particularly grateful to Job Noorman for his enthusiasm in helping me contribute to his projects, as well as Jan Tobias Mühlberg and Jo Van Bulck for the interesting discussions. I also want to express my gratitude to Jens Hermans

and Roel Peeters for trusting me to come along on their spin-off adventure. Furthermore, I am indebted to Johannes Götzfried for showing me a wonderful time on every visit to Erlangen, for introducing me to climbing and taking me snowshoeing, but especially for the many warm conversations.

Over the years, I've had the privilige of sharing an office with some genuinely great people. I would therefore like to acknowledge Kimmo Järvinen and Anthony Van Herrewege for their kind and helpful pointers when I was just starting out. I also want to thank Furkan Turan and Toon Purnal for their delightful company and productive discussions. However, I particularly want to mention Ruan de Clercq, who moved along with me through the different offices, and thank him for his friendship and all the exciting climbing trips.

It has also been an absolute pleasure to be a part of COSIC, and I could not have wished for a friendlier group of colleagues. I would like to recognise the members of the Hardware Group for the informative presentations and appreciated feedback, but just as much everyone else at the group, who were always ready to answer questions or give advice. I will continue to remember the entertaining lunch conversations, the deliciously generous cake events, adventurous weekends, and the relaxing evenings, sharing a beer on the castle's lawn. This group could not do without the care and assistance of its amazing technical and administrative people. In particular, I want to express my gratitude to Wim Devroye for his expert handling of our travel arrangements and equipment purchases, and especially to Péla Noé for making us feel at home.

This thesis would not have been written without the financial help of different organisations, notably the KU Leuven and the InvasIC project funded by the German Research Foundation. I also want to acknowledge the Research Foundation - Flanders for offering me an SB PhD fellowship.

After first discovering VTK during that weekend before my first year, I later joined this organisation, and I am incredibly thankful to everyone there and at Student IT for patiently bearing with me while I learned new skills. I also want to extend my appreciation to my friends, and I hope we will continue to gather regularly for many years to come. Although they are no longer all around, I am grateful to my grandparents for passing on their passions and believing in me. Furthermore, I want to thank my little sister Liesbeth for lending her ear whenever needed and showing me other points of view. Thank you, Mom and Dad, for believing in me and for always being there unconditionally, ready with help and guidance. Lastly, to my girlfriend Delphine, I could not have hoped for someone kinder by my side, and I am grateful for your gentle support.

Pieter Maene
Leuven, October 2019

§ Abstract

ELECTRONIC devices have become indispensable parts of our homes and businesses. They help us stay in touch with each other and events around the world, as well as facilitate us by controlling our appliances and driving complex manufacturing installations. Consequently, they interact with physical processes, and while they can operate stand-alone, these increasingly capable devices are commonly remotely accessible. Combined, these evolutions not only lead to advanced automation, but also expose our homes and factories to security risks. Indeed, attackers have exploited software vulnerabilities, causing fridges to send out spam, the shutdown of multi-national companies, and even the disruption of nuclear enrichment facilities.

Various defence strategies have been proposed in response, one of which is trusted computing. The core concept behind it is to ensure that an attacker cannot elicit undefined behaviour from the device. Due to the limitations of software-based approaches, countermeasures that can protect against strong adversaries have to be rooted in hardware, guaranteeing their functional integrity. Crucially, the extraction of critical functionality into self-contained modules enables many of the protection mechanisms introduced by trusted computing.

In this thesis, we present novel lightweight building blocks that contribute to the realisation of this goal at different levels of the design hierarchy. First, various hardware-based trusted computing mechanisms presented in literature are surveyed, leading to definitions of common security properties and a comparison of all considered designs. Since cryptographic primitives are at the core of these solutions, often impacting performance, we also evaluated seven block ciphers with respect to their area and latency. Our third contribution is a key distribution service for Systems-on-Chip, allowing secure management of symmetric device secrets through tight integration of efficient hardware and software components. Finally, we discuss the design and evaluation of two processor-based mechanisms, which rely on cryptographic units to protect sensitive code and data.

§ Samenvatting

Elektronische apparaten verbinden ons niet alleen met elkaar, maar vereenvoudigen ook onze levens door zowel huishoudtoestellen als complexe industriële installaties aan te sturen. Hoewel ze alleenstaand kunnen werken, zijn deze steeds krachtigere apparaten doorgaans ook vanop afstand bereikbaar. Deze evoluties hebben niet alleen gezorgd voor verregaande automatisatie, maar maken onze woningen en fabrieken ook kwetsbaar voor veiligheidsrisico's. Zo hebben aanvallers zwakheden in software uitgebuit om koelkasten spam te laten uitsturen, de werking van multinationals plat te leggen en zelfs nucleaire verrijkingsprocessen te verstoren.

Daarop zijn verschillende verdedigingsstrategieën voorgesteld, waaronder betrouwbare berekening, met als kernidee dat de aanvaller geen ongedefinieerd gedrag mag kunnen uitlokken van het apparaat. De beperkingen van oplossingen op basis van software vereisen dat tegenmaatregelen die beschermen tegen sterke aanvallers, verankerd zijn in hardware, waar hun functionele integriteit gegarandeerd is. Hierbij maakt het onderbrengen van kritieke functies in autonome modules verscheidene van deze beveilingsmechanismes mogelijk.

In deze thesis presenteren we nieuwe goedkope bouwblokken die bijdragen aan de realisatie van dit doel op verschillende niveaus van de ontwerphiërarchie. We bouwen eerst op een overzicht en vergelijking van verscheidene hardware-gebaseerde ontwerpen uit de literatuur om definities op te stellen van gedeelde beveiligingseigenschappen. Gezien deze oplossingen steunen op cryptografische primitieven, waar ze prestaties kritisch beïnvloeden, evalueren we de oppervlakte en wachttijd van zeven blokcijfers. Onze derde bijdrage is een oplossing voor sleutelverdeling op systemen-op-chip, die veilig beheer van symmetrische apparaatgeheimen mogelijk maakt door middel van doorgedreven integratie van efficiënte hardware- en softwarecomponenten. Tot slot bespreken we het ontwerp en de evaluatie van twee processoraanpassingen die gebruik maken van cryptografische eenheden om gevoelige code en gegevens te beschermen.

§ Contents

§ List of Abbreviations

AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AEX	Asynchronous Enclave Exit
AIK	Attestation Identity Key
ARC	Address Range Comparator
ASIC	Application-Specific Integrated Circuit
ASLR	Address Space Layout Randomization
CA	Certificate Authority
CAN	Controller Area Network
CFA	Control Flow Attestation
CFG	Control Flow Graph
CFI	Control Flow Integrity
CMP	Compartment Metadata Page
CMV	Physical Page Compartment Membership Vector
CPSR	Current Program Status Register
CPT	Compartment Page Table
CPU	Central Processing Unit
CRL	Certificate Revocation List
CT	Compartment Table
DDoS	Distributed DoS
DEP	Data Execution Prevention
DMA	Direct Memory Access
DoS	Denial-of-Service
DPA	Differential Power Analysis
DRAM	Dynamic Random-Access Memory
DRoT	Dynamic RoT
EA-MPU	Execution-Aware MPU

ECB	Enclave Control Block
ECC	Elliptic Curve Cryptography
EID	Enclave Identity
EK	Endorsement Key
ELF	Executable and Linkable Format
EM	Execution Monitor
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
EPID	Enhanced Privacy Identifier
ETB	Embedded Trace Buffer
EtM	Encrypt-then-MAC
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HMAC	Keyed-Hash Message Authentication Code
HSM	Hardware Security Module
I/O	Input/Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IDT	Interrupt Descriptor Table
IoT	Internet of Things
IP	Infrastructure Provider
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
IU	Integer Unit
IV	Initialization Vector
KDF	Key Derivation Function
KDM	Key Derivation Mechanism
KDS	Key Distribution Service
LFSR	Linear Feedback Shift Register
LLC	Last-Level Cache
LPC	Low Pin Count
LRU	Least Recently Used
LSB	Least Significant Bit
LUT	Look-Up Table

MAC	Message Authentication Code
MAL	Memory Access Logic
MCU	Memory Control Unit
MEE	Memory Encryption Engine
MitM	Man-in-the-Middle
MMIO	Memory-Mapped IO
MMU	Memory Management Unit
MPU	Memory Protection Unit
MRID	Memory Region ID
NoC	Network-on-Chip
NS	Non-Secure
OS	Operating System
PAR	Place and Route
PC	Program Counter
PCR	Platform Configuration Register
PL	Programmable Logic
PLC	Programmable Logic Controller
PMA	Protected Module Architecture
PMR	Protected Memory Region
PRM	Processor Reserved Memory
PRNG	Pseudo-Random Number Generator
PROM	Programmable ROM
PS	Processing System
PTBR	Page Table Base Register
PTM	Program Trace Macrocell
PTR	Private and Authenticated Tamper-Resistant Environment
REE	Rich Execution Environment
RNG	Random Number Generator
ROM	Read-Only Memory
ROP	Return-Oriented Programming
RoT	Root of Trust
RTM	RoT for Measurement
RWP	RAM Write Pointer
SE	Secure Executable
SEID	Secure Executable ID
SEM	Security Enforcement Module
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extensions

SK	Security Kernel
SM	Software Module
SME	Secure Memory Encryption
SoC	System-on-Chip
SP	Software Provider
SPID	Secure Process ID
SPM	Scratchpad Memory
SSBL	Second Stage Boot Loader
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TE	Tamper-Evident Environment
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TOCTOU	Time-of-Check Time-of-Use
TP	Trusted Partition
TPM	Trusted Platform Module
TRNG	True Random Number Generator
TXT	Trusted Execution Technology
UDP	User Datagram Protocol
UP	Untrusted Partition
VPN	Virtual Private Network
WSN	Wireless Sensor Network

List of Figures

§ List of Tables

1 Introduction

Security and privacy increasingly gain importance as electronic devices become more present in our daily lives. Attackers are always looking for ways to exploit the software running on these devices in an attempt to obtain sensitive information. All of our devices are also more and more networked, with remote access growing the opportunity for attack considerably. This might seem innocuous when it is just about fridges sending spam, but can have significant ramifications when critical infrastructure is breached.

Discovered in 2010, Stuxnet is perhaps one of the most well-known worms, as it was responsible for disrupting Iran's nuclear program [1]. It contained a sophisticated payload for a specific Siemens Programmable Logic Controller (PLC) series, which were installed in a uranium enrichment facility and controlled centrifuges. The attackers reprogrammed the PLCs in such a way that they would vary the frequency of these centrifuges, driving down the yield of this process. However, the malware would still report that everything was operating as intended, hiding its manipulation and confusing the plant's engineers.

More recently, a power grid operator in Ukraine was locked out of his computer and saw his mouse move across the screen, flipping high-voltage breakers in local substations [2]. The attackers also compromised two other distribution centres, cutting power to several regions of the country. They even went as far as to disable the backup power supplies for the control centres themselves, leaving their operators in the dark.

Finally, the global operations of shipping giant Maersk ground to a halt as ransomware spread through the company's systems, encrypting their hard drives [3]. With the global network offline, no new cargo could be accepted or released at any of Maersk's shipping terminals. It took the company several days to restore basic functionality and employees could only resume work after two weeks. NotPeya, as the malware was called, affected several other multinational companies as well, causing an estimated ten billion dollar in damages.

In each of these attacks, electronic devices were manipulated to make them deviate from their normal, intended behaviour. This brings us to the subject of trusted computing, which ensures that a device can be trusted if it cannot be made to misbehave. Given the complexity of today's software and the growing sophistication of the attacks perpetrated against it, this is not straightforward to attain. Therefore, trusted computing solutions build on Roots of Trust (RoTs), inherently trusted components that enable this trust relationship. This thesis evaluates the cryptographic primitives that are at their core, introduces novel solutions that enable the use of these strong ciphers, and proposes protection mechanisms that build on them.

1.1 Problem Statement

Attackers commonly target memory vulnerabilities to exploit software, abusing this weakness to access its code or leak its secrets. The former are referred to as code-reuse attacks, where execution is redirected to realise malicious functionality without adding new code. These vulnerabilities can have far-reaching effects, giving attackers a foothold onto the device, that could be combined with other attack vectors and leveraged to gain privileged access or compromise secrets. In both cases, the adversary manages to make the program behave in ways that were not intended by its developers, either through the introduction of new behaviour or by leaking sensitive information.

Modern embedded and mobile devices no longer consist of discrete Integrated Circuits (ICs), but are instead built around a System-on-Chip (SoC). This single chip combines a processor with additional on-chip blocks, such as a Graphics Processing Unit (GPU), and peripheral connectivity in a single package, reducing the area and power requirements. Its tight integration has also influenced the hardware design process, as blocks from external vendors can now be combined with custom hardware in the same package. This has simplified introducing new hardware features, as all components are typically connected to a single system bus, through which they can be accessed by software or communicate with each other.

Today's programs are complex, often comprising hundreds of thousands or even millions of lines of code, which has made it increasingly difficult to guarantee that they are free of exploitable bugs. This observation was first made by McCune et al. in 2008 [4], leading to the concept of Protected Module Architectures (PMAs). They proposed to extract a program's critical functionality, such as implementations of cryptographic algorithms and protocols. These smaller modules are much more contained and can even be proven to be free of bugs.

Furthermore, isolation can protect these modules from other software running on the system, exposing only their public interface. Finally, PMAs also make it possible for third parties to obtain proof that the module's code was not tampered with through attestation.

These isolation and attestation mechanisms form the foundation of trusted computing, which guarantee that software cannot be made to misbehave. However, they require functionality offered by RoTs, which are inherently trusted components that validate the trust assessment. If any of them were to be compromised, the security of the system can no longer be guaranteed. RoTs can be implemented in hardware and software, but the former is more common, because the intrinsic immutability of hardware implies that their functionality cannot be tampered with. While they are essential to trusted computing architectures, RoTs themselves are also made up of multiple building blocks. We first consider these cornerstones of trusted computing, before moving up through the design hierarchy and introducing architectural security features, building on novel RoTs.

1.2 Background

Having outlined the motivation for designing trusted computing architectures in the previous section, we will now introduce related concepts that will return throughout this thesis. First, Section 1.2.1 discusses fundamental notions from computer architecture, such as the memory hierarchy and protection rings. Second, security solutions always make assumptions about the capabilities of the attacker they protect against, which we describe for trusted computing architectures in Section 1.2.2.

1.2.1 Computer Architecture

Hennesy and Patterson note that the design of modern electronic devices has become incredibly complex, making it hard to exactly define the term computer architecture [5]. Therefore, an architect has to determine the computer's important attributes and realise a fast and energy-efficient design subject to cost, power, and availability constraints. Trusted computing mechanisms have been designed for a wide range of architectures, from lightweight microcontrollers to high-end multi-core processors. Consequently, each of our solutions will outline its required architectural features, and the following presents several foundational concepts that are relevant to this thesis.

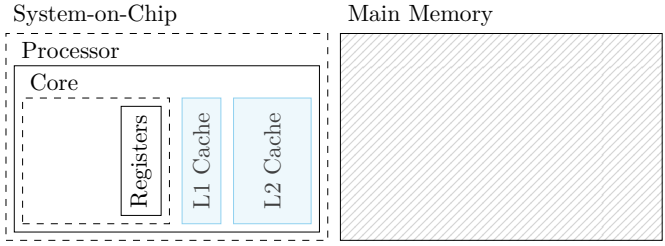

Figure 1.1: The inverse relationship between memory capacity and response times has resulted in a complex memory hierarchy, giving the processor access to a large storage area, seemingly at the latency of its internal registers.

Today's devices not only consist of hardware, but also feature system software, which is an important part of their design and integral to their operation. We therefore consider a system to comprise both the hardware and this basic software layer. Returning to the description of an SoC (Section 1.1), the hardware includes the processor, any on-chip components supporting the Central Processing Unit (CPU), and the communication bus interconnecting all of these components. Since main memory is not part of the SoC, the system boundary is found at the interface to external peripherals. Due to the heterogeneity of the platforms discussed in this thesis, we define the system software as any code that is not part of an application.

A fundamental computer architecture concept is the memory hierarchy, which is crucial to meet the performance requirements and power constraints of modern processors (Figure 1.1). Registers are the fastest type of memory and sit closest to the core's computing units, but they are also expensive in terms of area and power. Computers therefore rely on a large main memory, which is external to the CPU and connected to the system bus. However, since its response is much slower than that of the internal registers, CPUs also have one or more levels of cache. These introduce smaller and faster internal storage, which retain recently-accessed code and data. As registers, caches, and main memory are all volatile, most devices also include persistent storage (e.g., flash).

While the previous discussion focused on the main memory, it generally does not require all available addresses, especially not on 64-bit systems. Other resources, like peripherals, can therefore also be assigned a segment through which they are exposed to software. However, developers need to be aware of what resource is mapped where, as their accesses could pass through the cache, requiring manual flushing or marking these regions as non-cacheable, if supported. Finally, some architectures feature fast on-chip Scratchpad Memory (SPM) rather than caches, which is directly accessible from software instead.

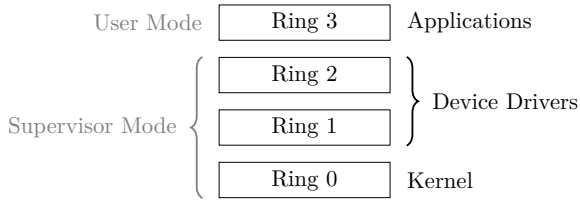

Figure 1.2: Processors have different privilege levels at which software can run, with associated capabilities. System software runs with more rights in supervisor mode, while applications are more restricted in user mode.

Rather than allowing software to access memory and peripherals through their physical addresses, computer architectures often rely on virtual addresses. Here, each application runs within its own virtual address space, isolating it from all other processes running on the system. When a virtual address is accessed, a Memory Management Unit (MMU) inside the processor will translate it to the corresponding physical one. This MMU is configured by the system software, which can typically also set the different access rights for ranges of memory. In contrast to the applications it manages, the system software therefore does have direct access to the physical address space.

As evidenced by the operation of an MMU, processors need some way to limit what software can or cannot do. To this end, most architectures implement protection rings or privilege levels, which allow the processor to control what operations are available. The number of levels and terminology varies across architectures (Figure 1.2), but CPUs generally offer at least a supervisor mode for system software and user mode for applications. Intel processors feature four rings, with the lower rings being the more privileged ones. The system software therefore runs in rings zero to two and applications are assigned to ring three [6]. In contrast, Arm cores only have two exception levels separating the system software from applications, respectively denoted by EL1 and EL0 [7].

Upon reset of the processor, a chain of software components is executed before the applications become active, initializing the system and its peripherals. Especially on more complex systems, it is common to have multiple of these so-called boot loaders, with the later stages including richer and richer functionality. This approach ensures that the boot process remains flexible, as the initial stage has to be stored in Read-Only Memory (ROM). The final boot loader will start the system software, which is in turn responsible for managing the applications. Trusted computing architectures make trust assessments about the state of these components, which is defined as their code and static data. This definition results from the requirement that this assessment should be verifiable, and can therefore only include elements that are known to the verifier.

1.2.2 Attacker Model

When building security mechanisms, designers always make some assumptions about the attacker's capabilities, and their solution will only be secure as long as the adversary stays within this model. In case of trusted computing architectures, these follow from considering an attacker targeting applications running on a remote system that he might have physical access to or even manage. Since he has full control over the system's configuration and the software running there, trusted computing mechanisms therefore need to be based on trusted hardware, which is significantly harder to tamper with than code. Furthermore, having physical access to the device means that he could directly connect to main memory and the bus interface that exposes it. Finally, this also gives him the possibility to manipulate the device's network channel.

All architectures discussed in this thesis include an attacker model in their description that follows from these considerations. While each design might introduce additional assertions to cover specific design decisions or account for the protection mechanisms it introduces, the attacker models of trusted computing solutions have several elements in common, which we will now list.

1. The attacker has access to all untrusted code, including large parts of the system software. This is therefore a very privileged attacker who can break traditional security assumptions (e.g., he can control the virtual address map). He can also manipulate existing code or introduce new software to the system.

2. It is assumed that the attacker controls the device's network channels, and can therefore intercept and modify messages. While he is generally considered to have the ability to tamper with the security protocols securing this channel, the adversary does follow the Dolev-Yao model [8]. This model states that the cryptographic primitives used in these protocols, such as the encryption algorithms or hash functions, cannot be broken by the attacker.

3. Denial-of-Service (DoS) attacks, which compromise device availability, are also not considered. In fact, many trusted computing solutions even introduce DoS vectors, as the processor will respond to security violations and take action against the offending software, e.g., by resetting the device. Additionally, these attacks are also out of scope at the network level.

4. We also exclude physical attacks on the device from the model. The adversary therefore cannot access or tamper with the device's internal signals. Furthermore, this assumption also implies that he cannot read or write data on the system bus directly or to main memory.

Finally, the attacker is also assumed not to mount any side-channel attacks, neither in hardware nor software. This class of attacks observes implementation-specific metadata and uses it to infer confidential information, that would otherwise have been protected. For example, if the secret were to determine whether an expensive operation is performed or not, the attacker could measure execution time or monitor the power drawn by the device to determine if it was active, recovering the secret bit by bit [9]. At the hardware level, physical effects are measured, such as power usage [10] or electromagnetic radiation [11], while a software-based attacker observes effects of code execution, like memory access latency [12].

Recently, microarchitectural attacks have been introduced, which exploit fundamental design flaws of the computer architecture from software [13–16]. Both hardware and software implementations can be protected from these types of attacks through various countermeasures, but microarchitectural vulnerabilities require particular attention, as they should be taken into account during the architecture's design process. We therefore view side-channel attacks as orthogonal to the issues addressed by trusted computing, as their respective protection mechanisms are generally composable.

Given this attacker model, we can now also identify the extent of the system's trust boundary, which encloses all components that are security-critical, as they will have direct access to sensitive data. In addition to any trusted hardware introduced by the specific architecture, the core processor functionality, its registers, and the cache hierarchy are typically included. In contrast to the system boundary defined earlier, the main memory and system bus are also within this boundary, unless there is a mechanism protecting the bus transactions and memory contents.

1.3 Outline

This thesis presents different contributions that we made to the field of lightweight trusted computing. Our work covers the design spectrum of hardware-based trusted computing architectures, from basic cryptographic building blocks (Chapters 3 and 4) to novel architectural protection mechanisms (Chapter 5). We now give a short overview of each chapter, sketching the structure of this text.

Chapter 1 – Introduction We first describe typical attack vectors that are used to exploit applications, before considering the hardware-based protection mechanisms that can protect against them. Next, an outline of this thesis

is presented and summaries of our publications that are not included in full conclude this chapter.

Chapter 2 – Trusted Computing Background For the past decade, both academia and industry have actively contributed to the field of trusted computing. In this chapter, we define the terminology that is used to describe these architectures, identify a shared attacker model, and introduce the security properties offered by them. Furthermore, we summarise thirteen existing hardware-based architectures, that offer either or both isolation and attestation mechanisms. Finally, the considered architectures are compared with respect to the considered security properties as well as traditional architectural features.

Chapter 3 – Single-Cycle Implementations of Block Ciphers Cryptographic primitives form a core building block of trusted computing architectures. Symmetric ciphers are particularly important for the design and implementation of lightweight architectures, as these algorithms can be implemented more efficiently in hardware. We built hardware implementations of six lightweight block ciphers and AES, in order to evaluate their latency and area requirements. Most block ciphers have an iterated design and we fully unrolled their internal structure to analyse single-cycle performance.

Chapter 4 – Eleutheria: Lightweight Key Distribution Service Part of the design complexity when using symmetric cryptographic algorithms arises from the fact that all communicating parties require access to a shared secret. This key distribution problem is typically solved by running a protocol based on asymmetric primitives first, which require significant hardware resources.

In this chapter, we present a key distribution mechanism which is rooted in hardware and relies on a network service implemented in software, securing the communication through asymmetric algorithms. Application-specific keys can be requested remotely and are derived in hardware from a unique device key, which is never extracted from the device or exposed to software.

Chapter 5 – Hardware-Based Memory Protection Mechanisms Applications often implement specific algorithms, which are valuable intellectual property, or process sensitive data. This information should therefore be secure from attackers, preventing them from obtaining the application's binary or retrieving its input and output.

To this end, we present two architectural extensions that introduce hardware-based mechanisms protecting either or both code and data in untrusted main memory. First, Atlas introduces an encryption unit in the memory hierarchy that transparently decrypts or encrypts any code or data that is read or written, including the application's identity in the encryption context, so that no code or data leaks to other software. Furthermore, the choice of cipher was driven by the observations from Chapter 3, as the memory access latency impacts system performances. Second, we designed an extension for Sancus enabling confidential loading of modules. This additional functionality required little extra hardware, as Sancus already implemented a flexible cryptographic primitive.

Chapter 6 – Conclusion We close this thesis with an overview of the presented solutions, highlighting their contributions, but also considering their limitations. Since our work focused on specific cryptographic primitives and trusted computing features, we also discuss possible avenues for future research.

1.4 Other Publications

Aside from the work discussed in this thesis, we also co-authored other publications. These papers are listed here in reverse chronological order, unless there were multiple papers on the same topic. A summary of their contributions is also included.

In *Soteria: Offline Software Protection within Low-cost Embedded Devices* [17], we extend Sancus with a software-based solution to protect code confidentiality. Application binaries are decrypted by a special loader module, whose identity is included in the derivation of the encryption key. Since Sancus defines the identity of a module as the combination of its memory layout and code, any modifications to the loader module will result in the derivation of an incorrect key and decryption will consequently fail. After the module has been loaded and its plaintext code is stored in memory, Soteria relies on Sancus' hardware-based isolation mechanism to ensure that it does not leak.

> J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling and I. Verbauwhede, "Soteria: Offline Software Protection within Low-cost Embedded Devices", in *Proceedings of the 31st Conference on Computer Security Applications*, 2015

SOFIA: Software and Control Flow Integrity Architecture [18, 19] is an architecture offering a strong Control Flow Integrity (CFI) mechanism. Control

flow attacks manipulate the original execution order of an application in an attempt to run arbitrary code. SOFIA's protection mechanism is two-fold. First, it encrypts the application's binary in such a way that it is bound to its control flow, preventing code injection and reuse attacks. Second, it introduces an integrity mechanism that dynamically calculates a Message Authentication Code (MAC) in the processor's pipeline and prevents execution from continuing if verification fails.

> R. de Clercq, R. De Keulenaer, B. Coppens, B. Yang, K. De Bosschere, B. De Sutter, P. Maene, B. Preneel and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", in *Proceedings of the 19th Conference on Design, Automation and Test*, 2016
>
> R. de Clercq, J. Götzfried, P. Maene, D. Übler and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", *Computers & Security*, vol. 68, no. 7, 2017

In *Hardware Acceleration of a Software-Based VPN* [20], we present an accelerator to improve the performance of Virtual Private Network (VPN) applications, which provide an encrypted network tunnel. Specifically, our architecture includes efficient hardware implementations of the Salsa20 stream cipher and Poly1305 MAC function. These hardware blocks were tightly integrated with SigmaVPN, a lightweight VPN application that runs on Linux. Our co-design for the Xilinx Zynq-7010 FPGA improves TCP and UDP bandwidth by a factor of 4.36 and 5.36 respectively.

> F. Turan, R. de Clercq, P. Maene, O. Reparaz and I. Verbauwhede, "Hardware Acceleration of a Software-Based VPN", in *Proceedings of the 26th Conference on Field Programmable Logic and Applications*, 2016

SOFIA's strong guarantees stem from tight integration with a processor's pipeline. However, modern SoC design typically integrates third-party building blocks, so-called IP cores. In order to facilitate more straightforward adoption of the concepts developed by SOFIA, but with weaker security guarantees, we designed *SCM: Secure Code Memory Architecture* [21]. In this paper, we present an IP core that connects to the SoC's main memory bus and aliases part of the system's main memory. When the processor is executing from this aliased range, the introduced hardware block will decrypt the code that is being read and verify MACs that are interleaved with it. If this integrity verification fails, the processor will be reset.

R. de Clercq, R. De Keulenaer, P. Maene, B. De Sutter, B. Preneel and I. Verbauwhede, "SCM: Secure Code Memory Architecture", in *Proceedings of the 12th Conference on Computer and Communications Security*, 2017

Finally, we present a device tracking system based on Internet of Things (IoT) technology in *A Privacy-Preserving Device Tracking System Using a Low-Power Wide-Area Network (LPWAN)* [22]. Our design relies on the presence of Bluetooth beacons, which continuously broadcast a unique identifier. Our tracker will periodically scan for these broadcasts and store them in a local buffer. At fixed intervals, this buffer will be encrypted and uploaded to a server through LoRa, which is a low-power networking technology. This server also hosts a list with the geographical coordinates of every beacon. The tracker's owner can link his device to a mobile application, which downloads and decrypts the messages from the server. Finally, the tracker's location history is displayed by combining the identifiers from the decrypted messages with the geographical coordinates of the beacons.

T. Ashur, J. Delvaux, S. Lee, P. Maene, E. Marin, S. Nikova, O. Reparaz, V. Rožić, D. Singelée, B. Yang and B. Preneel, "A Privacy-Preserving Device Tracking System Using a Low-Power Wide-Area Network (LPWAN)", in *Proceedings of the 16th Conference on Cryptology and Network Security*, 2017

1.5 Conclusion

Recent attacks have clearly shown the consequences of exploitable software vulnerabilities on personal devices as well as the systems of large companies and even critical utilities infrastructure. When applications are exploited, they act in ways that the user does not expect nor as intended by the developer. The goal of trusted computing is to address exactly this behaviour, by guaranteeing that software cannot be made to misbehave.

The remainder of this thesis will focus on hardware-based trusted computing, but any changes to the architecture also require modifications to the software it executes. We will first provide more extensive background on trusted computing, defining terminology and surveying existing hardware-based architectures for isolation and attestation. We then discuss the design and implementation of such architectures, starting at the bottom of their design hierarchy by analysing the performance of block ciphers and designing a novel key distribution mechanism. Finally, we close this thesis by introducing two architectural solutions protecting the confidentiality of code and data.

2 Trusted Computing Background

Before detailing the design and implementation of cryptographic building blocks and novel architectures in the next chapters, we must first explain what is understood by trusted computing and define the related terminology that will be used throughout this thesis. Trusted computing has been an active field of research over the past ten years, and several architectures have been proposed for devices ranging from lightweight embedded systems to high-performance desktop and server processors.

This chapter gives an overview of all major trusted computing designs making hardware changes to the underlying architecture from the academic community as well as industry, and offering either or both isolation and attestation. The former protects applications from other software, while the latter allows a third party to get proof that the software was not tampered with. Architectures which do not meet these requirements, being software-based designs or providing different functionality, are not included.

Content Source

P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation", *IEEE Transactions on Computers*, vol. 67, no. 3, 2017

Contribution: Main author together with Johannes Götzfried and Ruan de Clercq. Responsible for paper structure, comparison, and sections on AEGIS, Iso-X, Sancus, SecureBlue++, SMART, and TIMBER-V.

2.1 Introduction

The goal of trusted computing is to develop technologies which give users guarantees about the behaviour of the software running on their devices. More specifically, a device can be trusted *if it always behaves in the expected manner for the intended purpose* [24]. This means that even when an attacker gains control of the system, he cannot make it misbehave. This is a complex concept, covering many aspects, resulting in a wide range of solutions based on software, hardware, or co-design of both. Industry has also been active in this field, adding security mechanisms to their products, some of which can be found in millions of devices.

After defining trusted computing, the difference between it and *trustworthy computing* [25] should be pointed out. In trusted computing, users are asked to trust a set of components, and the security of the system is no longer guaranteed if any of its components are breached. Users are given no guarantees that the trusted components will not breach their security policies. On the other hand, trustworthy computing provides users with proof that its trusted components will not violate security [26]. Its focus is more on improving software engineering processes [27], rather than modifying the hardware architecture.

Although software-based trusted computing architectures with interesting results have been proposed [28–32], they can typically only be used in limited settings, nor are they able to give the same security guarantees as hardware-based architectures. An important part of trusted computing is to protect against attackers who have full control over the system, i.e., any application could have been exploited, as well as the Operating System (OS). Many hardware-based architectures shield applications from a malicious OS. No software-only solution can provide these guarantees, as an attacker can always manipulate software if the OS is not trusted. It is much harder for an attacker to modify hardware functionality, to the extent that hardware is considered to be immutable. Therefore, a user's trust is said to be rooted in the hardware, which is also why we only consider hardware-based architectures.

In the remainder of this chapter, we first define basic trusted computing terminology (Section 2.2) and expand the general attacker model introduced in Section 1.2.2 (Section 2.3). Next, we detail the trusted computing properties and architectural features that were taken into account for the comparison in Section 2.4. Section 2.5 then summarises the design of the selected hardware-based architectures. Finally, we compare them with respect to the defined properties and features in Section 2.6.

2.2 Preliminaries

This section introduces basic trusted computing terms which are most widely used by the different architectures. We assume that the reader is familiar with symmetric and asymmetric encryption algorithms, hash functions, and MACs. Otherwise, an introduction can be found in the Handbook of Applied Cryptography [33].

Protected Module Architectures (PMAs) Software has become incredibly complex, making it almost impossible to prove that an application does not contain bugs. Furthermore, attackers are always looking for vulnerabilities they can exploit to gain access to a system. As introduced in Section 1.1, McCune et al. therefore came up with the concept of Protected Module Architectures (PMAs) [4], where security-critical components are separated into smaller *protected modules*. Since they are much smaller, it is easier to verify their correctness. These modules are then isolated (Section 2.4.1) from any other software on the system, so that they cannot be tampered with. It has been shown that PMAs can be implemented at any level of the architecture, from the hardware to the OS kernel [34].

Throughout this chapter, we will adopt the terminology used by the original authors when referring to a specific paper. Consequently, protected modules will also be referred to as *Software Modules (SMs)*, *Secure Executables (SEs)*, *enclaves*, *secure tasks*, or *trustlets*.

Trusted Computing Bases (TCBs) are the sets of hardware and software components which are critical to their architecture's security. The careful design and implementation of these components are paramount to the overall security. The components of the Trusted Computing Base (TCB) are designed so that, when other parts of the system are exploited, they cannot make the device misbehave. Ideally, a TCB should be as small as possible in order to guarantee its correctness.

Measuring is used to verify the authenticity of software components. This is done by calculating a hash or MAC of its code and data. Some designs also include other identifying information, like the memory layout. The measurement result can then be used to attest (Section 2.4.1) the component's state. Since a hash or MAC value for a given input is probabilistically unique, it also identifies the state of the software component at that time.

Trust Chains are formed by verifying each component's validity from the bottom up. For software, this can be done by measuring each component in the chain before its execution.

2.3 Attacker Model

The general attacker model outlined in Section 1.2.2 also applies to the trusted computing architectures discussed in this chapter, but architectures often introduce additional assumptions to account for design decisions or novel protection mechanisms. We therefore repeat the attacker's main capabilities and provide additional detail related to the included designs.

1. The attacker is assumed to be in complete control of all software on every device in the system, except for the software that is part of the TCB. This means that he can tamper with the OS, or even deploy malicious software components. Some architectures use software modules that are part of the TCB, and it is assumed that the attacker cannot change these.
2. The communication channel to the device is considered to be controllable by the adversary. He is therefore capable of sniffing and modifying network traffic, or mounting Man-in-the-Middle (MitM) attacks on these channels. The abilities listed here are important when considering attestation.
3. The Dolev-Yao model [8] is applied, where the attacker is assumed to be unable to break cryptographic primitives, but can try and find weaknesses at the protocol level.
4. None of the architectures are capable of providing availability guarantees, and therefore cannot protect against DoS attacks.
5. Architectures which do not secure main memory consider physical attacks on the hardware out of scope, as they do not include any mechanisms preventing them. This means that the adversary does not have physical access, cannot probe the memory, and cannot disconnect components. However, architectures which do protect sensitive information when it is stored in main memory, consider the attacker capable of performing physical attacks on off-chip memory, but not on any hardware components which are part of the TCB, such as the CPU. In addition, none of the architectures include protection against hardware side-channel attacks, which are therefore not considered here.
6. In the context of this chapter, software side-channel attacks are considered to exist where untrusted software or malicious modules can extract secrets by observing memory access patterns. Architectures without protection against this therefore do not allow adversaries to monitor cache accesses or to observe page fault addresses. Note that other categories of software side channels exist, where the novel class of microarchitectural vulnerabilities was mentioned explicitly in Section 1.2, but these are all out of scope.

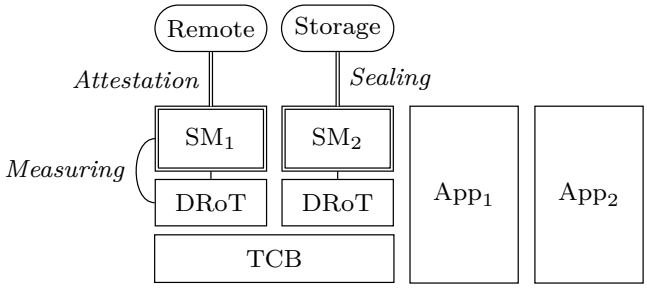

Figure 2.1: In general, a Protected Module Architecture (PMA) runs multiple Software Modules (SMs) side by side, along with one or more unprotected applications. The Trusted Computing Base (TCB) ensures that the state of the SMs is isolated from any other software running on the system (indicated by the double border). The measurement of the SM establishes a Dynamic Root of Trust (DRoT), and its result can attest the state of the module to a remote verifier. By sealing data, the SM can send it securely to untrusted storage.

2.4 Properties

Our work discusses and compares the different architectures with respect to a set of security properties (Section 2.4.1) and architectural features (Section 2.4.2). The former are the result of security mechanisms which were added specifically to provide users with strong guarantees about the software executing on their system (Section 2.1). The latter are features commonly found in current microcontroller and general-purpose architectures, but which require special attention in the context of trusted computing. Figure 2.1 gives a schematic overview of a PMA, also illustrating some of these security properties.

2.4.1 Security Properties

The following seven security properties were selected to facilitate this chapter's discussion, each of which is offered by at least one architecture described in Section 2.5. Since we focus on architectures which provide isolation and attestation, these were included first. All other properties are the result of new functionality introduced by the discussed architectures, and were added to enable a comparison between all designs. The first five are fundamental features provided by the trusted computing architectures discussed in this chapter, while the last two are also more widely used in security research.

Isolation denotes a hardware-based architectural mechanism that provides access control for software and its associated data. By placing code and data inside a protected module, no software outside it can read or write its runtime state or modify its code. Execution of code inside such a module can only be started from at least one predefined location. Such an *entry point* ensures that attackers cannot reuse the module's own code to extract secrets or implement malicious behaviour, as is done in Return-Oriented Programming (ROP) [35]. Current architectures allow for one or more modules, and some even support running them concurrently. Protected modules are used to store confidential information like secret keys, as other software cannot access its state. Writes into them are also prevented, protecting the integrity of their code and data.

Attestation is the process of proving to an authorised party that a specific entity is in a certain state. In order to give strong security guarantees, an architecture which supports attestation should guarantee integrity of the attested state as well. Trusted computing architectures may provide *local* and *remote* attestation. The former refers to one software module attesting its state to another running on the same platform, while the latter attests to a remote party residing outside the trusted system.

A common way to implement attestation is to measure (Section 2.2) modules during their initialization, while preventing later modifications by means of isolation. It can then be used to authenticate a challenge sent by the authorised party, as the measurement uniquely identifies the module's state. Since it could only have resulted from measuring a specific software module in a certain configuration, the authorised party knows it communicates with this module.

Sealing wraps confidential code or data in such a way that it can only be unwrapped under certain circumstances. Code or data are wrapped by binding it either to a specific device, a certain configuration of the device, the state of a software module, or a combination of these. It can then only be unwrapped when the binding conditions are met, e.g., on the same device or one which runs the same configuration. Sealing is usually based on encryption, and relies on similar mechanisms as software attestation, i.e., the key for encrypting confidential code or data is typically derived from the software module measurement taken during initialization.

Dynamic Roots of Trust (DRoTs) In order to keep the TCB (Section 2.2) as small as possible, most trusted computing technologies build trust chains. However, these chains always need to be anchored in a component that is inherently trusted, which are referred to as Roots of Trust (RoTs). A Dynamic RoT (DRoT) is established for a software module at runtime, measuring the application's state right before execution starts. It is typically combined with isolation to protect against Time-of-Check Time-of-Use (TOCTOU)

vulnerabilities as well, where an attacker changes the module's code after it has been measured.

Code Confidentiality A trusted computing architecture which guarantees code confidentiality ensures that sensitive code or static data cannot be obtained by untrusted hardware, software, or any other unauthorised entity covered by the attacker model. This property usually requires both isolation and encryption. Isolation is used to protect against software attackers after modules have been loaded. Encryption is needed to protect confidential information before loading, and to prevent physical attacks. Sealing can be used to ensure that only a certain software component can obtain some intellectual property.

Side-Channel Resistance A trusted computing architecture is called side-channel resistant with respect to software attackers if no software module, including privileged software like an OS, is able to deduce information about the behaviour of other modules apart from their in- and output. In the context of this chapter, particularly information flowing through untrusted channels, such as caches, or revealed by page faults cannot leak to untrusted software. An architecture with side-channel resistance should take care to flush caches during context switches. Leakage due to page faults, for example, can be prevented by giving each software module the ability to handle its own page faults.

As mentioned in Section 1.2.2, none of the architectures we consider, include micro-architectural side channels in their attacker model and therefore do not protect against them. These software-based attacks exploit fundamental architectural flaws to extract secret information from other applications running on the system, and PMAs have been shown to be vulnerable to them [15, 16].

Memory Protection specifically refers to protecting the integrity and authenticity of data sent over system buses or stored in external memory from physical attacks. We consider both *passive* (e.g., bus snooping) and *active* (e.g., fault injection) attacks. First, this means that data has to be encrypted, to prevent sensitive data from leaking. Second, it also has to be integrity-protected, for example, using a MAC. Third, replay attacks, where previously valid memory contents are restored, have to be prevented as well. These operations have to be performed when data is sent to or fetched from external memory.

2.4.2 Architectural Features

Designers have to make numerous decisions when integrating the security mechanisms needed for trusted computing in complex modern processor architectures. We selected seven basic features they typically take into account.

The first, targeting a lightweight architecture or not, is special as it will also influence the design of the other features.

Lightweight We define an architecture to be lightweight when it does not use an MMU. Lightweight embedded systems have very simple memory hierarchies, and therefore do not need complex memory management. Furthermore, they only run a limited number of applications, which share the memory space cooperatively, not requiring virtual addressing to map the memory (Section 1.2.1).

Coprocessor Many trusted computing architectures require security mechanisms to be implemented in hardware. In case of a coprocessor, this functionality is added as a separate chip or module which interfaces with the main processor through the bus. Alternatively, the functionality can be integrated inside the processor. Trusted computing architectures that are integrated in a processor can typically provide stronger security guarantees than coprocessor-based designs, because data does not have to leave the CPU. In addition, some functionality (e.g., isolation) cannot be implemented in a coprocessor, unless it is tightly coupled to the CPU or can access its internal signals.

Hardware-Only TCB It is typically better for the TCB (Section 2.2) to rely on hardware, as this provides stronger security guarantees, such as protection from an untrusted OS. In this chapter, we only discuss architectures that have a hardware-only TCB or architectures that have one consisting of hardware and software components.

Preemption When preemption is supported, the system can suspend running tasks at any time, without first obtaining permission from the task. This makes it possible to handle interrupts, but also allows preemptive scheduling of multiple protected modules. Preemption mainly impacts the context switching logic, since the architecture now has to ensure that no sensitive information can leak between modules, as this would violate the isolation primitive. Without support for preemption, applications have to run cooperatively, i.e., they need to call each other after finishing a task.

Dynamic Layout A static layout is often used when all software shares the same address space, and no MMU is present to provide virtual memory for different applications. It has the disadvantage that one trusted entity, e.g., the hardware or software manufacturer or a system integrator, needs to compile all software and fix the layout before deployment to the target device. With a dynamic layout, however, applications can be loaded to locations that do not need to be known at compile time.

Upgradeable TCB Architectures which have a HW-only TCB are not upgradeable, because its components can no longer be changed after being manufactured. However, some designs include trusted software components,

typically to implement functionality which would be too expensive in hardware. These components are then protected by a hardware mechanism, such as Programmable ROM (PROM). This not only results in more design flexibility, but also enables upgrading the TCB at a later time, e.g., when a bug has been discovered or to add new functionality.

Backwards Compatibility When adding features, an important consideration for industry is whether legacy code runs on the modified architecture without any changes, possibly after recompilation. Since these applications do not use the introduced protection mechanisms, they typically do not receive any of the associated security guarantees.

2.5 Architectures

This section gives detailed descriptions of thirteen isolation and attestation designs, making hardware modifications to their target platform. Architectures which are implemented entirely in software or provide other functionality were therefore not included. A wide range is covered, from lightweight designs for the IoT to desktop and server architectures. The selection was not limited to academic research, but also includes industry efforts. All architectures were ordered chronologically by year, and then alphabetically.

2.5.1 AEGIS

Suh et al. designed AEGIS [36] in 2003 already, making it one of the oldest trusted computing architectures. It provides programs with a Tamper-Evident Environment (TE), where any memory tampering, either from software or physical, is detected. Even stronger guarantees are provided by Private and Authenticated Tamper-Resistant Environments (PTRs), which also protect the confidentiality of code and data. The CPU itself is considered to be trusted, placing external memory and peripherals outside of the TCB, where they are vulnerable to software and hardware attacks. In a hardware-supported implementation, the OS can be malicious, but when a Security Kernel (SK) is used to implement AEGIS' functionality, parts of the OS need to be trusted.

Since the system can also run legacy code, the protection mechanisms have to be enabled by calling the `enter_aegis` instruction. This will calculate a hash of the program and store it in a protected area. The program hash will also be used later as an identifier. Each program also includes some code which will measure any other code or data it depends on. Finally, it should also ensure

that it is running in the expected environment (e.g., the current CPU mode). After `enter_aegis` has been called, the program is isolated, and any memory tampering will be detected. Since the on-chip caches are considered trusted, the hardware only needs to prevent applications from writing to locations they do not have access to. This is done by tagging the cache entries with a Secure Process ID (SPID), which is assigned when the program is started.

However, a more complex mechanism is needed to protect off-chip memory. Whenever data is read into the cache, its integrity needs to be verified. When a cache block is evicted, the corresponding leaf node of a hash tree is updated with the new contents. Of course, this means that all internal nodes also have to be updated. Since the depth of the tree increases with the memory size for a given block size, this could result in a large number of additional transfers. The internal nodes are therefore also cached in the L2 cache, performing updates first in the cache, and not directly in memory. The authors distinguish between non-blocking and blocking instructions. For the former, the integrity verification can be delayed, as long as the CPU is eventually notified it was working with tampered data. When the AEGIS mode is set to PTR, the CPU also needs to guarantee confidentiality. This is done by encrypting the blocks with AES in the CBC mode of operation, using 32-bit random Initialization Vectors (IVs) to guarantee uniqueness of the ciphertexts. The architecture uses separate keys for static and dynamic data. The former is used to decrypt the binary's code and data, while the latter protects any data generated at runtime.

The remote attestation mechanism hashes the provided data together with the program digest, and asymmetrically signs the result with a private key specific to the CPU. This binds the data to the code of the program, as well as the specific processor it is executing on. In case AEGIS' features were implemented in a software SK, the hash of the kernel is also included in the attestation result. This functionality is provided through the `sign_msg` instruction.

2.5.2 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) version 1.2 [37] was specified by the Trusted Computing Group (TCG) in 2011. It is a coprocessor on the motherboard, which is capable of storing keys and performing attestation. It is a passive piece of hardware, meaning that software can interact with the TPM, but needs to do so explicitly. To give a local or remote party guarantees, any software that runs on the target device, i.e., the boot loader, OS, and applications, needs to be measured successively by the TPM, and is consequently part of the TCB. Code manipulation is only detected during measurement, and all parts of the software are considered trusted after loading.

The TPM only provides limited protection against physical attacks, because not only the CPU package, but the TPM chip and all connecting buses are part of the TCB as well. For instance, a physical attacker can compromise software integrity by tapping the Low Pin Count (LPC) bus between the TPM and CPU [38].

The design principles published by the TCG in version 1.2 specify the minimum functionality and cryptographic primitives that are required for TPMs, but a TPM manufacturer is allowed to extend the hardware module with additional functionality. Each TPM has to be equipped with a Random Number Generator (RNG) as a randomness source and an RSA implementation with at least 2048-bit keys. The minimum requirement for software measurement is the SHA-1 hash algorithm. At manufacturing time, the Endorsement Key (EK) is generated and written to persistent memory within the TPM. The EK is unique to every TPM chip, not known to the user, and serves as master key for all operations provided by the TPM. In addition to the EK, Attestation Identity Keys (AIKs) and storage keys can be generated on the fly and stored in volatile memory within the TPM. AIKs are used for all operations directly involving digital signatures, whereas the storage keys are used for encryption and decryption of data. Finally, a TPM contains a certain number of Platform Configuration Registers (PCRs), which are capable of storing successive hash values for code or data that is sent to the TPM and are important for remote attestation. The TCG also specified TPM version 2.0 [39] in 2014, supporting a larger variety of cryptographic algorithms, multiple banks of PCRs, and three hierarchies instead of only one. For simplicity, we focus on version 1.2 in the following discussion.

With the minimum functionality required by the TCG, each TPM is able to perform at least three different operations. First, it is able to bind code or data to a given device by encrypting it with one of the storage keys. The encrypted code or data can then only be decrypted on the same platform, because the particular storage key cannot be extracted from the TPM's volatile memory.

Second, a TPM is able to attest to another party that the given device is currently running a certain software configuration (Section 2.4.1). To this end, code or data sent to the TPM is hashed together with values of specific PCRs, and the result is again stored in the same PCRs. This way, each software component running on the device is able to successively extend a measurement over all components running on the device. The PCRs are initially set to a fixed value before starting the system, and because the hash function is irreversible, it is not possible to set the PCRs to a user-defined value. The resulting hash values from the PCRs can then be cryptographically signed by an AIK, and this signature can later be verified by a second party. When this party receives a correctly signed value, it can be sure that the system runs a certain software

configuration, because this signed message could not have been created without going through the software measuring process.

Third, code or data can be sealed to a given device in a certain software configuration (Section 2.4.1). In general, sealing works similar to binding, i.e., code or data is encrypted and decrypted, but it is additionally ensured that sealed code or data is only decrypted if the platform configuration has not changed in between. To check against changes of the platform configuration, the PCR values are saved together with the sealed code or data and checked against the current PCR values during unsealing.

Attestation and sealing only behave as intended if the platform configuration is measured from the earliest boot step, up to the currently running software component, because otherwise malicious software could potentially exclude itself from the measurement. This restriction is the biggest disadvantage of the standalone TPM over other solutions that support DRoTs (Section 2.4.1).

To overcome the restriction of all software having to be part of the TCB, Intel introduced its Trusted Execution Technology (TXT) [40], which also uses the TPM chip, but allows dynamically establishing a new RoT for software running in a virtualised environment separate from the normal stack. TXT ensures that the virtualised software has exclusive control over the device by suspending any other running code, i.e., the OS and all applications. When switching to trusted TXT software, the CPU essentially performs a warm reset and initialises a certain subset of PCRs with a new value. The TXT software can then extend this measurement and attest to a second party that it has not been modified before being loaded. Since it monopolises all resources once it has been loaded, the integrity of the TXT software is guaranteed over its entire runtime.

Although TXT can be used to overcome the restriction of all software having to be part of the TCB, it still has some issues. Suspending all other applications on the device for the TXT software to run negatively impacts performance, or might even lead to losing interrupts depending on its size. Since the TXT code has to run exclusively, it cannot easily use functionality of untrusted software and needs to perform expensive context switches. Finally, all physical attacks that succeed for a standalone TPM, e.g., LPC bus tapping, also succeed for TXT. Fides [31], Flicker [4], and TrustVisor [41] are examples of architectures which build on the functionality offered by TXT.

2.5.3 TrustZone

GlobalPlatform wrote an industry standard for security architectures called the Trusted Execution Environment (TEE) [42, 43]. The TEE is a secure area of the

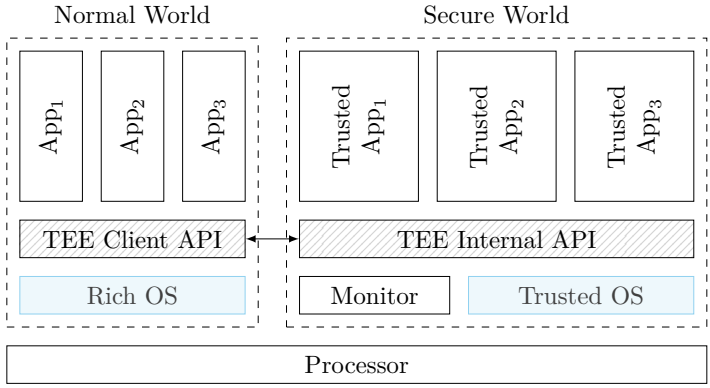

Figure 2.2: TrustZone is a hardware-based security architecture by Arm that creates two execution domains, the normal world and secure world. These are also referred to as the Rich Execution Environment (REE) and Trusted Execution Environment (TEE), respectively. Any resources assigned to the secure world cannot be accessed from the normal world. Since both worlds share the processor, the monitor performs the context switch between them.

main processor, and provides isolated execution, integrity of trusted applications, as well as confidentiality of trusted application resources. The TEE is isolated from the Rich Execution Environment (REE) where the untrusted OS runs. The REE resources are accessible from the TEE, while the TEE resources are not from the REE, unless explicitly allowed. Therefore, only trusted resources can access other trusted resources. The standard does not specify how manufacturers should implement it, with TrustZone being an implementation of this standard by Arm.

TrustZone [44] (Figure 2.2) is a hardware-based security architecture for an SoC that is currently used in a large number of smartphones. The TEE, also called the *secure world*, provides protection for trusted hardware and software resources. Hardware-based mechanisms ensure that resources in the REE's untrusted OS, or *normal world*, cannot access secure world resources. This is done by two main hardware features. First, the SoC's AXI bus ensures that secure world resources cannot be accessed from normal world resources. Second, a TrustZone-enabled processor core uses time-slicing to execute code in either the secure or normal world.

To enforce isolation between trusted hardware resources on the bus, a control signal known as the Non-Secure (NS) bit was added to the AXI specification. This bit is used to communicate the security state of a master component to a slave component. The bus or slave logic uses this bit to ensure that the security

separation is not violated. When an untrusted master attempts to access a secure slave, the transaction should fail and an error may be raised.

A TrustZone core can switch between security states at runtime. When the processor core is in the secure state, it generates AXI transactions with the NS bit set to zero, allowing it to access resources in both security domains. However, a processor core in the normal world can only access normal world resources. The processor's L1 and L2 caches use a bit to store the security state of the transaction that accessed the memory. The cache controllers are then assumed to be responsible for ensuring that only secure masters can access memory that was fetched from a secure slave. Extending the cache removes the need for a flush when performing a context switch between security domains, and further allows software to efficiently communicate from the non-secure to the secure world.

To perform a context switch to the other world, the processor first has to pass through a new mode called *monitor mode*, which serves as a gatekeeper that manages context switches between the two worlds. This is done by saving the state of the current world and restoring the state of the world being switched to. Monitor mode exists in the secure world, and both privileged and user mode exist in each world. Normal world entry to monitor mode is only possible via an interrupt, external abort, or explicit call of the `smc` instruction. Secure world entry to monitor mode can additionally be invoked by writing into the Current Program Status Register (CPSR). Arm recommends to execute monitor code with interrupts disabled. The address mappings in the MMU can be configured independently for each world. This allows the OS in each security domain to enforce its own memory management. Inter-Process Communication (IPC) with small messages can be done by placing the message inside registers and then invoking `smc`. For larger messages, it is possible to use shared memory.

Interrupts can be serviced in either security domain. When a context switch is required to handle an interrupt, the processor traps directly into monitor mode. A different exception vector table is used to specify the interrupt service routine addresses for normal world, secure world, and monitor mode respectively. Each of the vector table base addresses can only be updated from its respective mode. This enables secure interrupt sources that cannot be manipulated by normal world software.

During the boot process, a chain of trust is formed by verifying the integrity of the trusted Second Stage Boot Loader (SSBL) and trusted OS before execution. The TrustZone processor starts in secure world when it is powered on. The firmware of the first stage boot loader is implicitly trusted, and is typically located in ROM. It initialises critical peripherals, such as memory controllers, and further performs an integrity check of the SSBL, which is stored in flash.

If this check passes, the SSBL is executed. It in turn verifies the integrity of the secure world OS and boots it, after which the normal world OS is started without performing an integrity check. Some implementations of the secure OS also verify the integrity of trusted applications before loading them. TrustZone uses a signature scheme based on RSA. A vendor uses its private key to sign the code. The firmware then uses the public key to verify the signature at runtime. In order to support different vendors, the architecture supports the use of several public keys.

Since any trusted component can violate the system's security, it is important to respect the principle of least privilege and restrict the access of each component in the TCB. TrustZone does not adhere to this design principle, as applications from different vendors run in the same secure world. Furthermore, when multiple secure master devices from different vendors are placed on a TrustZone SoC, least privilege is violated as every secure master has access to all memory.

2.5.4 Bastion

Bastion [45] is a combined hardware-software architecture, which relies on a trusted hypervisor together with a modified processor to ensure confidentiality and integrity for security-critical software modules. Physical attacks on all hardware components but the CPU package are allowed, i.e., Bastion provides memory protection. Only single-core processors are currently supported.

Since everything apart from the microprocessor and the hypervisor is considered untrusted, including firmware and code needed during booting, Bastion first protects the state of the hypervisor. Afterwards, the hypervisor is able to protect any number of software modules. To this end, the Bastion hypervisor calls `secure_launch` from its initialization routines, which computes a cryptographic hash over the state, i.e., code and data, of the hypervisor and stores the result in a CPU register. The `secure_launch` routine also generates a new key which is used to encrypt and integrity-protect all data belonging to the hypervisor with the help of an on-chip cryptographic engine. The hash value later serves as the identity of the hypervisor and is, for example, needed to unseal permanently stored data. The implementation of `secure_launch`, as well as the register contents, cannot be modified from software.

After the hypervisor has been loaded, software modules can invoke a new `secure_launch` hypercall for initialization, which activates runtime memory protection for all module pages and computes a hash of the module's initial state, including the virtual memory layout, that serves as the module's identity. For instance, this identity is used to seal data that needs to be permanently stored on disk.

A modified CPU ensures that the hypervisor is invoked for each Translation Lookaside Buffer (TLB) miss. The hypervisor checks whether the virtual address responsible for the access corresponds to the one associated with the physical page and a specific software module. For these checks, modules are identified by a unique identifier (usually eight to 20 bits), which is assigned during the `secure_launch` call. All untrusted software not belonging to a specific module is treated as a module with identifier zero, ensuring that untrusted software cannot access code or data from security-critical modules.

To invoke a function of a secure module, a special `call_module` hypercall is added, which takes the hash of the destination module and the entry point as parameters, because direct transitions would trigger a memory violation. Similarly, on returning, the `return_module` hypercall is needed. When a module needs to be preempted, for example, due to a timer interrupt, the hypervisor takes care of saving all state information, such as register contents, and wiping sensitive data before calling the interrupt handler. When returning from the interrupt handler, the hypervisor also takes care of restoring all state information and handing back control to the secure module.

2.5.5 SMART

Eldefrawy et al. designed SMART [46] to establish a DRoT in remote embedded devices (Section 2.4.1). The architecture is also minimal, requiring only the smallest possible set of hardware changes in order to implement remote attestation, which was later formalised by Francillon et al. [47]. It was one of the first designs using hardware-software co-design to build a lightweight trust architecture. Prototypes to demonstrate the feasibility were built on open-source clones of the ATmega103 and MSP430. The attacker model specifically assumes that adversaries do not tamper with ROM. Peripherals which can directly access memory should be disabled while SMART is executing.

In general, SMART provides remote attestation of a memory range $[a, b]$ specified by the verifier. It then calculates a MAC over this memory region and sends the result back. The verifier calculates the same MAC over the expected contents and compares both. This process dynamically establishes a RoT. In addition, an address x can be given, where execution will continue atomically after SMART has completed. By choosing $x = a$, the verified code is started.

Support for SMART requires four features: attestation ROM, secure key storage, MCU access controls, as well as reset and memory erasure. The ROM stores the attestation code which is invoked when a verification is requested. This program disables interrupts, measures the specified memory region by calculating a SHA-1 HMAC [48], and reports the result. When x is set, interrupts remain

disabled and control jumps to that address, but otherwise they are re-enabled and execution continues.

Secure key storage is added to the microcontroller for the symmetric key used to calculate the HMAC, and the MCU access controls ensure that it is only accessible when the CPU's Program Counter (PC) is in the ROM region containing the attestation code. In order to prevent code reuse attacks, the MCU also enforces a single entry point into the ROM code, only allowing access from the initial instruction, and disabling exits from any instruction other than the last. When an invalid memory access is detected by either mechanism, the processor is reset immediately.

The attestation code is carefully written to ensure it cleans up any sensitive data after it has finished. However, when the processor is reset during its execution, this cleanup might be skipped. Therefore, all memory is erased by the hardware when the processor boots or after a reset.

SMART and TrustLite (Section 2.5.10) were later used to prototype a scalable attestation mechanism for large swarms of small embedded devices [49]. Furthermore, Eldrefawy and Tsudik contributed to a formally verified remote attestation scheme called VRASED [50]. While its design relies on similar mechanisms as SMART, it is based on abstract models that enable formal verification of its security guarantees. This is realised by first considering different sub-properties in isolation before proving the end-to-end security of their composition. These seven sub-properties are split into two groups, covering key protection and safe execution of the attestation code. VRASED's implementation consists of a hardware module and software component, where the former enforces memory access control and the latter produces a SHA-256 HMAC measurement. Finally, the authors prototyped their architecture on the MSP430, which satisfies the required microcontroller and compiler axioms.

2.5.6 Sancus

Sancus [51] is a hardware-only PMA designed by Noorman et al. for lightweight embedded devices, like wireless sensor nodes. In addition to isolating an application's code and sensitive data, it also has support for remote attestation. It adds secure linking functionality as well, enabling applications to verify modules they depend on. While a follow-up publication [52] introduced additional functionality and improvements to the prototype implementation (Section 5.3), this section details the original design.

The architecture was designed for small embedded devices, which are typically deployed in large swarms. These nodes are managed by an Infrastructure

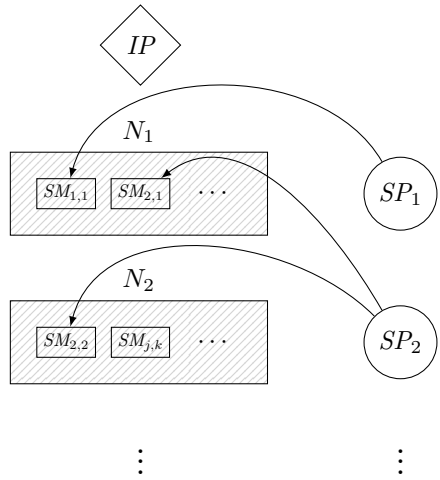

Figure 2.3: Sancus considers multiple nodes that are managed by a single Infrastructure Provider (IP). Different Software Providers (SPs) want to run their Software Modules (SMs) on one or more of these nodes.

Provider (IP), and they share a fixed key K_N with it, which is etched into the silicon. When Software Providers (SPs) want to load a protected application onto a node, they have to go through the IP. Each SP is assigned a unique public identifier, which is used to derive a key $K_{N,SP} = kdf(K_N, SP)$ for the SP. The *kdf* is implemented in hardware to realise a zero-software TCB. Since the IP manages the node key K_N, it knows all other keys used in the system, and SPs therefore have to trust it to behave as intended.

In Sancus' system model (Figure 2.3), protected applications are called Software Modules (SMs). Each Software Module (SM) consists of a text section, which contains code and constants, and a protected data section storing sensitive dynamic data. Additionally, an SM can include unprotected sections, which makes it possible for developers to keep the size of the sensitive code as small as possible. Each SM is also assigned an identity, consisting of the contents of its code section, and its layout. The latter are the start and end addresses of its protected code and data sections, making it possible for two modules where these sections are identical to exist on the system at the same time. Similar to $K_{N,SP}$, the SM's identity is used to derive a third-level key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$.

When an SM uses functionality from another protected module, it can use caller authentication to ensure that the other module was not tampered with. To this end, SM_1 stores a MAC with its own key K_{N,SP,SM_1} of SM_2's identity in an unprotected section. It can issue a special instruction at runtime to verify

the MAC. The hardware also enforces a single entry point. This single physical entry point is not a limitation, as it can be multiplexed to multiple logical ones.

The memory access control mechanism is program counter-based, i.e., the access rights depend on the current value of the processor's program counter. The protected text section is always readable, but only executable when the module is running. The only exception is the entry point, which is always executable. The text section has to be readable for the caller authentication to work, as it is included in the MAC. Similarly, the protected data section is only read- and writeable when the program counter is in the module's text section.

These protection mechanisms are enabled by calling the special `protect` instruction, which has the layout information and SP identity as parameters. This instruction also derives the node key $K_{N,SP,SM}$ and stores it together with the layout in a *protected storage area* (Figure 2.4), inaccessible from software. Conversely, invoking `unprotect` will disable the isolation primitive. Memory violations are handled by resetting the CPU, at the cost of availability, which is excluded from the attacker model (Section 2.3). Three additional new instructions are introduced for the remote attestation and secure linking functionality respectively. First, `MAC-seal` can be used to calculate a MAC with $K_{N,SP,SM}$ over a given memory range. Second, calling `MAC-verify` will calculate the MAC of the specified module, with the current module's key, and verify that it matches the MAC stored in memory. Third, since the MAC calculation is expensive, a module is assigned a monotonically increasing ID by the CPU at load-time, which can be queried with `get-id`. This ID is used to securely link to the same module at a later time, by storing it and checking that it still matches the one returned by `get-id`.

A prototype was developed based on the openMSP430, an open-source implementation of Texas Instruments' MSP430 processor. The most important changes are the addition of the on-chip protected storage area, the Memory Access Logic (MAL) circuitry, and a hardware implementation of the HMAC algorithm based on SPONGENT [53], a lightweight hash function. The number of SMs which can be loaded concurrently is fixed at synthesis time, and determines the size of the protected storage area. The MAL circuit is fully combinational, so it does not need additional cycles to perform its checks. Furthermore, it is shown not to be on the processor's critical path, meaning that it doesn't impact the clock frequency. In addition to the hardware changes, an LLVM-based toolchain was built to compile SMs, which allows developers to easily use the new functionality through code annotations, inserting stubs which handle stack switching, secure linking, and entry point multiplexing. Evaluation of the prototype showed that the main performance overhead is found in instructions which use the hashing functionality, i.e., `protect`, `MAC-seal`, and `MAC-verify`. The duration of the hash calculation depends on the size of the input data.

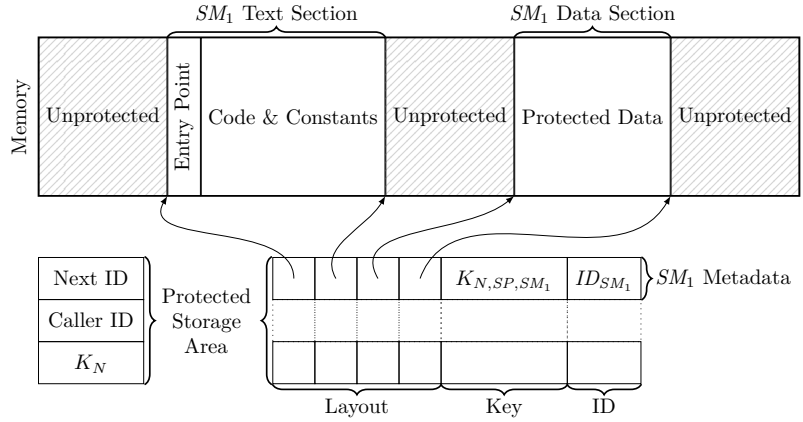

Figure 2.4: Sancus introduces several new hardware registers, that are inaccessible from software and only read or written by special instructions. The layout, $K_{N,SP,SM}$, and ID of each module are stored in these registers.

Soteria [17] is an extension of Sancus which takes advantage of the architecture's functionality to add code confidentiality. This is done by creating a loader module SM_L, which has the module key K_{N,SP_L,SM_L}. This loader module atomically reads the encrypted binary from the node's memory, and writes the decrypted code back to memory, after which it calls `protect` on the decrypted SM. Each module is assigned an identity $\widetilde{SM_E}$, which is used to derive the encryption key $E_{SM_E} = kdf(K_{N,SP_L,SM_L}, \widetilde{SM_E})$. Although the encryption algorithm is implemented in software, it is not part of the TCB, because the key derivation includes the module's text section. Therefore, any changes to SM_L would result in a different E_{SM_E}, which would be detected during authenticated decryption. Due to the way the derivation of E_{SM_E} is implemented, SM_L and SM_E mutually authenticate each other. Additionally, Sancus' hardware is modified to deny other SMs access to the text section of SM_E.

Following the software-based design presented by Soteria, hardware support for code confidentiality was also added to Sancus 2.0 (Section 5.3), along with other architectural and implementation improvements. The former includes support for interrupts, while an example of the latter is the unification of all cryptographic operations required by Sancus in a single hardware unit based on SpongeWrap [54].

2.5.7 SecureBlue++

SecureBlue++ [55, 56] is an early PMA from IBM, which isolates Secure Executables (SEs) from each other, and protects the confidentiality and integrity of their code and data. The main architectural changes involve a Memory Protection Unit (MPU), using different mechanisms at each level of the memory hierarchy. It also protects against physical memory attacks, and prevents writes to the SE's code region, as well as execution from its data memory.

A SE binary consists of a cleartext section containing a loader, which copies the cleartext integrity tags of the compiled binary into memory, and then calls the new `esm` instruction to start decryption of the encrypted sections and jump to the SE's entry point. The loader is followed by system call pass-through buffers and the *executable key*, which was used to encrypt the sections that are stored after it, namely the metadata for the call to `esm`, and the application's code and data. This key is itself encrypted asymmetrically under a public key bound to the CPU that the application will run on. The private key is installed in the factory, and the manufacturer signs the public part to generate a certificate asserting its validity.

The MPU protects an application's code and data at all levels of the memory hierarchy. Encryption is used to protect data in external memory, automatically decrypting cache lines and verifying their integrity when they are read. Similarly, evicted lines are encrypted on the fly, also updating the integrity protection tree. A tree is used because replay protection requires a nonce, which in turn needs to be stored and MACed. All integrity information is kept in a dedicated memory region, but only the root node and its metadata require expensive on-chip storage.

The caches store everything in plaintext, and therefore also need to enforce access control. This is done by adding a label to each cache line with the ID of the SE it belongs to. The CPU stores the current Secure Executable ID (SEID) in a special register and compares it to the line's label. Instead of storing the SEID directly, a Memory Region ID (MRID) is used. This is an index into the Protected Memory Region (PMR) table holding the metadata of a specific page, like the owner's SEID. This table also manages shared memory regions, adding a second SEID for the first sharer. Any additional SEs requiring access are given the same secret, which needs to be present at a specific memory location.

To avoid leaking the values stored in registers after a context switch, SecureBlue++ stores the registers protected in the cache, meaning they would also be encrypted automatically when evicted to memory. A new privileged instruction, `RestoreContext` can be used by the OS to restore the registers and wake the previously active SE, which is indicated in a special register.

Since the isolation mechanism prevents even the OS from accessing an SE's data, two approaches for handling system calls are presented. The first is modifying *libc* to wrap the system calls, not requiring any hardware support. The second is to change the behaviour of the system call instruction `sc`. Before transferring control to the OS, the CPU checks whether it is in SE mode. In this case, the call is redirected to a small wrapper inside the SE before invoking the `sesc` instruction, which bypasses the security check and immediately calls the OS's system call handler.

2.5.8 Software Guard Extensions (SGX)

In 2013, Intel announced Software Guard Extensions (SGX) [57], which enables establishing dynamic RoTs inside regular applications, without monopolising the system like TXT (Section 2.5.2). SGX supports protecting an application's code as well as data [58], is able to guarantee integrity, and provides local and remote attestation [59]. In addition, SGX includes physical attacks on communication channels and main memory in the attacker model.

In SGX, the protected parts of an application are placed within so-called *enclaves*. An enclave can be seen as a protected module within the address space of a given process, and enclave accesses obey the same address translation rules as those to process memory, i.e., the OS and the hypervisor are still in charge of managing the page tables. This has the advantage that SGX is fully compatible with existing memory layouts, usually configured and managed by an MMU, and also works well in a multi-core environment. Although an enclave resides in the process address space, there are certain restrictions in enclave mode. For example, system calls and instructions that would cause a trap into the OS or hypervisor, such as `cpuid`, are not allowed and it is necessary to leave enclave mode before dispatching them. Furthermore, this mode can only be entered from user mode, which essentially means enclaves can be used within applications, but not the OS [60].

An SGX-enabled CPU ensures in hardware that non-enclave code, including the OS and potentially the hypervisor, cannot access enclave pages. Specifically, a region called the Processor Reserved Memory (PRM), which contains the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM), is protected against all non-enclave accesses by the CPU. The EPC stores enclave pages, i.e., enclave code and data, while the EPCM stores state information about the pages currently held within the EPC. This state information consists of the enclave page access rights and the page's virtual address when the enclave was created, amongst others. For each (non-cached) access of an EPC page, the current access rights and virtual address are checked against the state stored

within the EPCM, and if a mismatch is detected, access is denied. The caching of state information is necessary, because all software, including at system level, is considered untrusted, and therefore attacks such as enclave layout changes through remapping have to be prevented directly in hardware. If the capacity of the EPC is exceeded, enclave pages might be written out to a memory region outside the PRM by the OS, but are then transparently encrypted by a hardware Memory Encryption Engine (MEE) [61], which is inside the CPU package.

Before an enclave can be used, it has to be created and initialised by untrusted software. The hardware ensures that an enclave's pages can only be modified until the initialization is finished. All page contents, including code and static data, are measured during initialization. As this measurement depends on all contents of the enclave, and later modifications are prevented, it can be used as a basis for local or remote attestation. All operations involved in the management of an enclave, e.g., enclave creation, initialization, and destruction, are performed by system software, while entering and leaving the enclave is done by the application software. The latter is implemented similarly to system calls, i.e., an enclave has its own execution context and dedicated instructions need to be called. SGX enclaves are again entered through an entry point, but its predefined location can be varied for each enclave thread. Upon leaving the enclave, the context of the current thread is saved within an EPC page and all registers are cleared. The appropriate context is loaded again when the enclave is entered. If an interrupt occurs during enclave execution, an Asynchronous Enclave Exit (AEX) is performed by the CPU, which also saves the current enclave execution context and ensures that no data leaks to the untrusted system software handling the interrupt.

Within an enclave, other features are provided in addition to confidentiality and integrity of code and data. One enclave can attest to another that it has been loaded as intended by sending a *report*, which includes information about the enclave (the measurement) or its author. This process is called local attestation. With the help of a trusted Quoting Enclave provided by Intel, the report can be wrapped into a *quote*, converting the local attestation to a remote attestation by signing the quote with an asymmetric attestation key, which is part of Intel's Enhanced Privacy Identifier (EPID) group signature scheme [62]. The quote can be verified by a remote party with the corresponding verification key provided by Intel. Besides local and remote attestation, data produced within an enclave can also be sealed to it and, e.g., written to memory outside the PRM. Sealed data can serve as permanent storage and retains information during different runs of an enclave. Local attestation, remote attestation, and sealing all rely on the non-forgeability of the initial enclave measurement.

Intel uses dedicated enclaves for complex functionality which would be expensive to implement in hardware, like the asymmetric cryptography needed for remote

attestation. It also provides a Launch Enclave required to start executing any other enclave, a Provisioning Enclave to initially provision asymmetric keys for attestation to end-user devices, and the previously mentioned Quoting Enclave to cryptographically sign the attestation quotes. The downside of this approach is that Intel has a de facto monopoly regarding enclave signing and remote attestation, as they decide which enclaves are allowed to run and everybody who wants to verify quotes needs to have an agreement with Intel.

More details about SGX can be found in an exhaustive summary by Costan [63]. Since its introduction, SGX has been used to secure different applications. For instance, Open Whisper Systems relies on it to enable private contact discovery for their encrypted messaging application Signal [64]. Haven [65] and VC3 [66] are two academic solutions which use it in an untrusted cloud context. However, neither solution used real hardware, but relied on an emulator instead. Finally, AMD has also presented security extensions for their processors called Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [67]. The former adds memory encryption at page granularity to protect data in memory, while the latter relies on this unit to isolate virtual machines from each other as well as the hypervisor.

2.5.9 Iso-X

Iso-X [68] is an isolated execution architecture where memory can be assigned dynamically to Untrusted and Trusted Partitions (UPs and TPs), which contain *compartments*. These compartments are essentially protected modules, and a developer can indicate which parts of his code should be compartmentalised. The architecture also includes a remote attestation mechanism in hardware, which is based on asymmetric signatures.

A static memory region is allocated and protected during boot to store hardware-maintained management information. The Physical Page Compartment Membership Vector (CMV) is a bit vector tracking whether a memory page was already assigned to a compartment or not, while the Compartment Table (CT) records the compartment's characteristics, like its base address and size. In addition to these static structures, each compartment is also given a Compartment Page Table (CPT), which maps virtual to physical addresses, since the OS's page tables cannot be trusted. Finally, each compartment is assigned a Compartment Metadata Page (CMP) as well, which keeps track of any other data, like the compartment's public key.

Six new instructions are added to the processor, which are called either by the OS or applications to manage and use compartments. First, a new system call can be invoked by an application when it wants to start a compartment. It looks for

an unused compartment identifier, and then executes the `COMP_INIT` instruction, signalling the hardware to prepare its internal data structures. Similar to SGX (Section 2.5.8), memory pages are only added to the compartment after its initialization. This is done through the `CPAGE_MAP` instruction, which also adds the page's hash to the compartment's measurement. Note that this also includes the virtual page number and permission bits. Analogously, `CPAGE_REVOKE` removes a page from the compartment, which is considered destroyed once no more pages belong to it. Finally, a compartment can be started from its entry point through `COMP_ENTER`.

`COMP_ATTEST` generates a certificate that can be used to prove the compartment's integrity to an external verifier. This certificate is signed with the CPU's private key, which is generated in the factory. Finally, `COMP_RESUME` restores the compartment's state after a context switch, copying CPU registers back and returning to compartment mode.

Additionally, the authors present a secure swapping mechanism to support memory management. Before giving the OS access to a page, `COMP_SWAP_PREP` hashes it and overwrites the corresponding CPT entry with the result, also resetting the valid bit. The page is then encrypted symmetrically, and the corresponding CMV bit is cleared. Afterwards, the OS uses `COMP_SWAP_RET` to copy the page back to memory, which not only decrypts it and verifies the hash, but re-enables isolation as well.

2.5.10 TrustLite

TrustLite [69] is a generic PMA for low-cost embedded systems which was developed by the Intel Collaborative Research Institute for Secure Computing. A *trustlet* isolates software components, providing confidentiality and integrity guarantees for both its code and data. The architecture provides OS-independent isolation of trustlets, attestation of them, trusted inter-process communication, secure peripherals, and supports interrupts. It was implemented as an extension to the Intel Siskiyou Peak research platform. The attacker model from Section 2.3 is used, with the assumption that the trustlets and bootstrapping routine behave correctly.

When the TrustLite device is booted, the first software to execute is the Secure Loader, which is stored in PROM as part of the SoC. The Secure Loader is responsible for loading trustlets and their data regions into on-chip memory. It also configures an MPU to enforce isolation of each trustlet's memory regions, which can include Memory-Mapped IO (MMIO) peripherals. The configured regions are also recorded in a Trustlet Table for use by individual trustlets or attestation routines. The untrusted software, such as the OS, is allowed

to execute after the Secure Loader has configured all trustlets. The Secure Loader is only active during initialization, and the MPU is also used to protect it at that time. As the initialization code configures the memory access control regions on platform reset, there is no need to clear the main memory as in SMART (Section 2.5.5).

The MPU uses registers to store multiple different protection regions for each trustlet. The architecture uses program counter-based isolation. The memory regions of a trustlet are only accessible when the PC is in its code region. When the PC is outside of a specific trustlet, the regions specified in the MPU are not accessible. The processor raises an exception on an access violation. In addition, it invalidates currently executing instructions, and flushes the processor's pipeline stages.

To support interrupting an executing trustlet, the architecture needs to ensure that no information leaks. It does this by storing the current state of the processor on the stack of the interrupted trustlet, saving the stack pointer in the trustlet table and clearing the general-purpose registers, after which the OS stack pointer is restored, followed by execution of the Interrupt Service Routine (ISR). A return from an interrupt is performed by jumping to the entry point, and restoring the trustlet's stack pointer in software.

Each trustlet uses an entry vector to specify the addresses which can be called by other tasks or trustlets. The trustlet itself can execute its entire code section, but other tasks or trustlets can only execute the addresses listed in the entry vector. The entry vector should be carefully programmed to avoid information leakage or other exploits.

Signalling and sending short messages are done by calling the entry point of a trustlet and passing the arguments in CPU registers. Large messages can be communicated by signalling with a pointer to a shared memory region, which needs to be inside an MPU region. Protected communication between trustlets is performed by means of a simple handshake protocol. The handshake requires that the initiator verifies the platform state, and that each party attests the other trustlet's state by checking the correctness of the relevant entries in the trustlet table and MPU registers. The initiator may additionally perform an integrity check of the responder's program code to ensure that it was not maliciously modified. After attesting each other, subsequent messages can be authenticated by means of a cryptographic session token.

2.5.11 TyTAN

TyTan [70] is an architecture for lightweight devices which provides isolation between tasks, secure IPC with sender and receiver authentication, and has real-time guarantees. Its TCB consists of both hardware and software components.

A core trusted hardware component of TyTAN is its Execution-Aware MPU (EA-MPU), which enforces program counter-based isolation. The EA-MPU ensures that each isolated task can only access its assigned memory regions. In addition, these tasks can only be invoked at a dedicated entry point.

Several static secure software components are part of the TCB. The Secure Boot task is invoked at boot, and is responsible for loading all other trusted software components. Each of these components is isolated from the rest of the system by the EA-MPU. The EA-MPU supports loading and unloading of secure tasks at runtime by means of a driver. The RoT for Measurement (RTM) task can attest other tasks. This task calculates a cryptographic hash of the binary code of each created task, which serves as its identity id_t. The Remote Attest task uses a MAC to prove the authenticity of id_t to a remote verifier.

Secure IPC is done by means of the IPC Proxy task. This task is responsible for forwarding a message m from the sender S to the receiver R. For short messages, the sender invokes the proxy with the receiver's identity id_R and message m as parameters, which then copies m into R's memory. Since the EA-MPU ensures that only the proxy can write to R's memory, this implicitly authenticates m and id_S. For large messages, the proxy sets up a shared memory region that is accessible only by the communicating tasks.

The Secure Storage task seals data by storing it encrypted in non-secure memory. It is encrypted with a task key that is derived from id_t. Tasks communicate with the secure storage via secure IPC. Finally, a trusted Interrupt Multiplexor (Int Mux) task is used to securely save the context of an interrupted task to its stack and clear the CPU registers before control is passed to the interrupt handler. Different interrupt handlers can be specified in the Interrupt Descriptor Table (IDT), which is protected by the EA-MPU.

TyTAN was implemented as an extension to Intel's Siskiyou Peak architecture, and uses the FreeRTOS real-time OS. The FreeRTOS preemptive scheduler was modified to support secure tasks. All secure software tasks were designed to be interruptible, or to have an upper bound on execution time. To support dynamic loading of tasks, FreeRTOS was extended with an ELF loader.

2.5.12 Sanctum

Sanctum [71] combines minimal hardware modifications with a trusted software component to offer an isolation primitive which is in many ways similar to SGX (Section 2.5.8). Like SGX, Sanctum only allows enclaves to run at user level. In contrast, it does not provide memory protection, as there is no MEE to encrypt code or data before being written out to memory.

In Sanctum, each enclave controls and manages its own page tables and handles its own page faults, whereas the former are managed by the OS or hypervisor in SGX. Furthermore, Sanctum ensures that each enclave is assigned a separate DRAM region corresponding to distinct sets in the shared Last-Level Cache (LLC). These two measures allow Sanctum to protect enclaves against software side-channel attacks where a malicious application or OS tries to learn information from an enclave's memory access pattern (Section 2.4.1). In SGX, a potentially malicious OS can observe the accesses of any enclave at page granularity by reading the page table's *dirty* and *accessed* bits. Additionally, enforcing distinct cache sets per enclave protects against cache timing attacks.

Instead of implementing trusted functionality in microcode like SGX does, Sanctum uses a trusted software component called the *security monitor*. When booting a Sanctum system, measurement code within ROM is executed and calculates a hash of the security monitor, which is included in all further measurements, before giving control to it. The security monitor then provides an API for enclave management, e.g., for creating and destroying enclaves. It also manages transitions into and out of enclaves, i.e., special monitor calls need to be used to enter and exit an enclave. In case of an interrupt, the security monitor takes care of saving the enclave's current state. After handling the interrupt, however, the enclave is entered at its entry point, and has to restore its state on its own. The environment within an enclave is also restricted, so that enclaves need to be exited for system calls and Input/Output (I/O).

Sanctum modifies the MMU in such a way that there are two Page Table Base Registers (PTBRs), one for untrusted code and one for the currently running enclave. Only the security monitor is able to change the contents of these registers. Furthermore, the modified MMU ensures that only certain pages can be referenced by enclave page tables. In more detail, metadata indicating valid pages is saved during enclave creation, and checked against after each page table walk. This metadata cannot be changed from software after enclave creation, during which the security monitor checks for overlapping pages or other invalid mappings, and writes it accordingly. Although the initial mappings are created by the OS, and copied to the enclave's page tables, the enclave is able to verify them by inspecting its own page tables and aborting if necessary.

2.5.13 TIMBER-V

By combining tag-based memory isolation with an MPU, TIMBER-V [72] realises a flexible and fine-grained lightweight isolation mechanism, which is managed by a trusted software component. The latter also implements remote attestation and sealing functionality, and is a trusted party in a shared memory scheme which implicitly provides local attestation. The design follows the general attacker model outlined in Section 2.3 and does not provide memory protection nor side-channel attack countermeasures. Interestingly, it even features a very specific CFI mechanism to protect calls from trusted to untrusted code. The authors evaluate their design on a cycle-accurate RISC-V simulator, indicating an average cycle overhead of 25.2%, which drops to 2.6% when tags are cached.

Together with the RISC-V's supervisor and user modes, the tag-based isolation mechanism of TIMBER-V provides four security domains, similar to TrustZone (Section 2.5.3). All untrusted memory is N-tagged, with the OS running in supervisor mode and applications at user level. On the trusted side, user-level *enclave* memory should hold a TU tag, and code executing in trusted supervisor mode should be TS-tagged. The processor switches the security context transparently based on the tag value of the fetched instruction, where an enclave's entry points require a TC tag, meaning that multiple physical entry points can be specified. In order to store these different tags, two metadata bits are added to every memory word, which are set using special *checked* instructions. These tag operations are subject to an update policy, preventing privilege elevation. Since the tags only indicate the different security domains, an MPU is used to isolate different processes within them, which can be configured by the system software running in either world. This is enabled by adding two flags to each slot. While the untrusted OS cannot manipulate slots marked with the TS flag, TU slots are shared between both domains. The TU flag is automatically cleared when the slot is overwritten by the untrusted OS and can only be asserted by the trusted system software. This combined approach allows interleaving of trusted and untrusted memory ranges, reducing fragmentation and obviating the need for trusted dynamic memory management.

An important part of the TCB is a software-based trust manager called TagRoot, which runs in trusted supervisor mode and manages enclaves and offers trusted services. Crucial to security, it is assumed to be loaded through a secure boot process, and protected by the isolation tags afterwards. Its enclave management functionality is invoked by the untrusted OS to instantiate and load enclaves. Similar to SGX (Section 2.5.8), the untrusted system software initializes enclaves by informing TagRoot of the enclave's memory regions and entry points, storing the enclave's configuration in an Enclave Control Block (ECB) allocated in protected TS-tagged memory. During this process, the enclave is measured

by calculating a SHA-256 hash, where the finalised measurement is used as the Enclave Identity (EID). The EID is used as part of an HMAC-based key derivation together with an identifier and secret platform key, supporting remote attestation and sealing functionality. Among other services, enclaves can request generation of keys by issuing a system call, which is handled by TagRoot. Although we will not discuss it in detail here, TagRoot is also central to TIMBER-V's preemption support and implements secure shared memory. Since the latter mutually authenticates the involved enclaves based on their EIDs, it also enables local attestation.

Since security domains are crossed transparently, the architecture does not feature instructions to control the isolation mechanism. As mentioned, it does provide checked variations of RISC-V's load and store operations, which are tag-aware and can both verify and manipulate the memory word's tag bits. For instance, `swct` has the same semantics as `sw`, but takes an expected and new tag as additional arguments, trapping if the current tag does not match the expected value, and updating it otherwise. Given that TIMBER-V uses two-bit tags, which are encoded as immediates, the address offset parameter is reduced to ten and eight bits respectively, necessitating code rewrites to address possible overflows. Additionally, a special `ltt` instruction can be used to test whether the current tag has a specific value.

2.6 Comparison

This section provides a detailed comparison of all architectures discussed in this chapter. Table 2.1 compares all of them with respect to the security properties and architectural features given in Section 2.4. In addition, it is indicated for each architecture whether it was published by academic researchers, if its source code is public, and what Instruction Set Architecture (ISA) it was based on.

Except for the TPM and SMART, which were specifically designed for attestation, all architectures provide some isolation mechanism, which protects applications from each other and even the OS. In general, lightweight architectures include program counter-based access control in the memory controller, verifying each access. A common approach is to use a set of boundary registers which indicate the memory regions for a pre-defined number of protected modules. Since they already include an MMU, complex architectures extend it to include access control. At cache line or page granularity, the isolation is much coarser here than it is for lightweight architectures, where each memory access is checked.

Table 2.1: Overview of all hardware-based trusted computing architectures detailed in Section 2.5. They are compared with respect to the security properties and architectural features they support. We also list whether they are open source, were developed by academia or industry, and which ISA was targeted.

Architecture	Security Properties							Architectural Features							Other		
	Isolation	Attestation	Sealing	Dynamic RoT	Code Confidentiality	Side-Channel Resistance[1]	Memory Protection[2]	Lightweight	Coprocessor	HW-Only TCB	Preemption	Dynamic Layout	Upgradeable TCB	Backwards Compatibility	Open Source	Academic	Target ISA
AEGIS [36]	●	●	●	●	●	○	●	○	○	●	●	●	○	●	○	●	–
TPM [37]	○	●	●	○	●	–	◐	○	●	●	–	–	○	●	○	○	–
TXT [40]	●	●	●	●	●	●	◐	○	●	●	○	●	○	●	○	○	x86_64
TrustZone [44]	●	○	○	●	○	○	○	○	○	○	●	●	○	●	○	○	Arm
Bastion [45]	●	○	●	●	●	○	●	○	○	○	●	●	●	●	○	●	UltraSPARC
SMART [46]	○	●	○	●	○	–	○	●	○	○	–	–	○	●	○	●	AVR/MSP430
Sancus [51]	●	●	○	●	○	●	○	●	○	●	○	○	○	●	●	●	MSP430
Soteria [17]	●	●	○	●	●	●	○	●	○	●	○	○	○	●	●	●	MSP430
Sancus 2.0 [52]	●	●	○	●	●	●	○	●	○	●	◐	○	○	●	●	●	MSP430
SecureBlue++ [55]	●	○	●	●	●	○	●	○	○	●	●	●	○	●	○	○	POWER
SGX [57]	●	●	●	●	●	○	●	○	○	○	●	●	●	●	○	○	x86_64
Iso-X [68]	●	●	○	●	○	○	●	○	○	○	●	●	●	●	○	●	OpenRISC
TrustLite [69]	●	●	○	○	○	●	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
TyTAN [70]	●	●	●	●	○	●	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
Sanctum [71]	●	●	●	●	●	●	○	○	○	○	●	●	●	●	●	●	RISC-V
TIMBER-V [72]	●	●	●	●	○	○	○	●	○	○	●	●	●	●	●	●	RISC-V

● = Yes; ◐ = Partial; ○ = No; – = Not Applicable

[1]Resistance against software side-channel attacks targeting memory access patterns only.

[2]Protection from physical attacks, both passive (e.g., probing) and active (e.g., fault injection).

It is interesting to see how different architectures implement remote attestation. Some architectures add a simple attestation protocol in hardware, based on symmetric primitives. These are cheaper than asymmetric algorithms in terms of computation and resource requirements. For example, Sancus (Section 2.5.6) uses an HMAC based on SPONGENT. However, other designs instead opt to move their attestation functionality to software, relying on hardware mechanisms to protect it from the rest of the system. In this case, local attestation is used to secure the on-chip communication with the application being attested. This approach is especially attractive for complex protocols, like the one from SGX (Section 2.5.8), which is based on a group signature scheme.

The same approach can be followed for other components of the TCB, especially for features which are expensive to implement in hardware. It has the additional advantage that the TCB can be upgraded, since it is partially implemented in software. The only exception is SMART, where the SW is stored in non-programmable ROM. In contrast, having a HW-only TCB implies that it cannot be upgraded. This software will be part of the design's TCB, though, requiring users to trust that a potential attacker has no way of modifying its operation. All architectures have at least part of their TCB implemented in hardware, which is assumed to be immutable by attackers. Any software component which is part of the TCB relies on these features. HW-only TCBs can generally give much stronger guarantees, because no part of the architecture is vulnerable to software-level attackers. However, if carefully designed and implemented, some of its components can be moved to software, speeding up development and increasing flexibility. TrustZone (Section 2.5.3) is the only architecture where a large amount of software is part of the TCB, because its isolation mechanism only supports two domains, and therefore includes the secure world OS.

All isolation architectures have similar attacker models (Section 2.3), and consequently protect against the same types of vulnerabilities. There are two high-level categories of software attacks: code injection and code reuse attacks. The isolation mechanism protects against the former, since an attacker outside the module can no longer modify its code. Furthermore, attestation enables detection of any changes to the module at the time the measurement is taken. An entry point prevents external adversaries from performing the latter (Section 2.4.1). However, neither mechanism can secure against vulnerabilities found inside the module itself. Software side channels are a third category, but only Sanctum (Section 2.5.12) addresses a specific instance of this attack class.

The trust boundaries typically extend to the CPU package, but in some cases external memories and peripherals are also included (Section 1.2.2). This is the case for the TPM, which is a coprocessor connected to a shared system bus. Any other components on the same bus are therefore part of the TCB. However, when external memories and peripherals are not included in the TCB, there is

also protection against physical memory attacks. For example, SecureBlue++ (Section 2.5.7) transparently encrypts and decrypts cache lines when they are evicted or fetched from memory. Therefore, attackers who probe the memory or snoop the bus cannot obtain sensitive information. The hardware also maintains an integrity tree of all entries, defending against active memory attacks.

All discussed architectures modify the processor architecture itself, except for the TPM (Section 2.5.2). SECA [73] is another instance where the security mechanisms are integrated outside of the CPU package. Instead, this architecture enforces configured *security contexts* at the bus level through a Security Enforcement Module (SEM). This hardware component monitors the bus traffic and interrupts the CPU when violations are detected. Different contexts can be configured by a secure kernel running on the processor, and it can update the currently active context at any time. For example, a security context can specify access rights to a specific memory range to isolate it.

Sancus (Section 2.5.6) is an example of a cooperative architecture without preemption (Section 2.4.2). Such designs often rely on static memory layouts where all applications are stored in pre-defined memory locations, so that they know where to call each other. Since only one application is running at the same time, monopolising all resources, no software side channels exist in these architectures (Section 2.4.1).

All included designs either give programmers the choice to integrate their security mechanisms, or enable them transparently. Both approaches result in fully backwards compatible architectures, and with dynamic loading, the original binaries can even be used. However, legacy applications remain just as vulnerable as before in most designs. TrustLite (Section 2.5.10) is one exception, as its MPU always provides isolation transparently once it has been configured by the Secure Loader, even for untrusted code.

The goal of isolation is to protect modules from any other software running on the system. However, these components sometimes need to be able to communicate with each other, or even with untrusted applications, like the OS. This IPC is typically implemented in two ways. The fastest is to use processor registers for passing smaller messages. Larger messages are sent through shared memory regions. Some architectures even support secure shared memory, where modules can selectively allow others to access a memory region (e.g., SecureBlue++ and TIMBER-V).

2.7 Conclusion

The goal of trusted computing is to protect applications and users from malicious software. It has increasingly gained interest in recent years, both from academia and industry, resulting in a variety of new mechanisms. We presented detailed descriptions of thirteen hardware-based architectures, focusing on attestation and isolation designs, and compared them with respect to their security properties and architectural features. Our comparison shows that all architectures offer strong guarantees, but very few support all possible trusted computing mechanisms. The main differences are the size of the TCB, which sometimes contains software, and where the trust boundaries extend to. Furthermore, not all architectures support certain architectural features.

This chapter shows there has been a lot of work in this area, but researchers do not have widespread access to these technologies. Academic architectures are rarely open sourced, making it harder to extend the work of other researchers. However, this may be changing thanks to the the rising popularity of the open-source RISC-V ISA. While mature isolation and attestation functionality is available for both low- and high-end architectures, not all related issues have been solved. For instance, possible avenues for future work include bringing these solutions to multi-core embedded systems or studying their interaction with other system components, like Direct Memory Access (DMA) controllers. Furthermore, the general decision to not consider software side channels as part of the attacker model has led to the compromise of trusted computing architectures, including Sancus [16] and SGX [15]. Finally, while these two mechanisms significantly reduce the attack surface, we noted that some vulnerabilities remain exploitable. They therefore do not fully realise the goal of trusted computing, leading to the development of additional solutions (e.g., CFI).

3 Single-Cycle Implementations of Block Ciphers

Cryptographic primitives are at the core of all trusted computing architectures discussed in Chapter 2. Ciphers are one type of primitive which can be used to protect the confidentiality of data. Both symmetric and asymmetric encryption algorithms exist, with the former relying on a single secret that is shared among all parties and the latter using key pairs consisting of a public and private key. While the private key should be stored securely, the public key can be shared with anyone. However, as we already mentioned, asymmetric algorithms have a higher cost, making them less suited for use in resource-constrained devices.

Because of this, symmetric encryption algorithms are an important building block for hardware-based trusted computing architectures. Block ciphers are one instance of such algorithms, which work on fixed-length data elements and typically have a round-based structure. This chapter gives synthesis results for unrolled implementations, meaning that its rounds were synthesised as a single combinational circuit, so that a data block is processed in one cycle. We consider six families of lightweight ciphers, where the same approach is used for all of them. Whenever possible, the results are grouped by block and key size to make a fair comparison with regard to the security they offer.

Content Source

P. Maene and I. Verbauwhede, "Single-Cycle Implementations of Block Ciphers", in *Proceedings of the 4th Workshop on Lightweight Cryptography for Security and Privacy*, ser. Lecture Notes in Computer Science, 2015
Contribution: Main author.

3.1 Introduction

Software applications have always been vulnerable to attacks from malicious actors. As introduced in Section 2.5.8, SGX automatically encrypts and decrypts sensitive data when it leaves or enters an enclave, which requires a fast cipher. Finding suitable low-latency cryptographic algorithms is one of the biggest challenges when bringing such memory protection functionality to area-constrained embedded systems.

Additionally, smaller silicon technology nodes make it possible to place more and more transistors on a single die, and modern SoCs have become many-core devices. High-bandwidth, packet-switched Networks-on-Chip (NoCs) have replaced slower buses [75]. Protection of these networks is an open research question. The underlying ideas of security mechanisms for traditional networks can be used, but will require fast and efficient cryptographic primitives.

In both these applications, data should be processed as fast as possible and it is not necessary that the cipher has high throughput. Additionally, design constraints often limit the clock frequency of these circuits. Therefore, only a limited number of cycles will be available to finish the encryption within a given delay and in some cases a single-cycle implementation will be the only alternative. One approach to achieve this, is by unrolling existing iterative block ciphers. However, this results in long combinational paths, which have a high associated delay. As will be shown in this chapter, they can only operate at such low clock frequencies, the operating speed of the architectures they are integrated with will be limited. Of course, introducing pipeline registers would increase the throughput and maximum clock frequency, but at the cost of additional latency. Another advantage of fully combinational implementations is that they can be easily integrated with existing designs, because of the lack of control logic.

The different algorithms discussed in this chapter are AES [76], KATAN [77], PRESENT [78], PRINCE [79], RECTANGLE [80], SIMON [81] and SPECK [81]. These ciphers were chosen because they cover a wide range of algorithm types and possible design choices. A similar analysis was done by Knežević et al. in 2012 [82]. This chapter includes some of the same ciphers, but also adds results for several recent designs that were introduced since. A short summary of the best known cryptanalysis results is given for each algorithm, showing their current security bounds. Section 3.2 first introduces some general concepts and terminology. Synthesis results for FPGA and ASIC are given in Section 3.3. Finally, Section 3.4 compares our results, followed by a conclusion in Section 3.5.

3.2 Preliminaries

Before discussing the selected designs, we provide some background on the structure of block ciphers and some common design strategies (Section 3.2.1). Section 3.2.2 and 3.2.3 respectively define logic depth and fan-out, which are two important circuit properties that influence its latency.

3.2.1 Block Cipher Structure

A block cipher (Definition 3.1 [33]) is a basic cryptographic building block offering confidentiality of data. It is used in a wide variety of applications, from protecting communication to generating pseudo-random numbers.

Definition 3.1. *An n-bit block cipher is a function $E : V_n \times \mathcal{K} \to V_n$, such that for each key $K \in \mathcal{K}$, $E(P, K)$ is an invertible mapping (the encryption function for K) from V_n to V_n, written $E_K(P)$. The inverse mapping is the decryption function, denoted $D_K(C)$. $C = E_K(P)$ denotes that ciphertext C results from encrypting plaintext P under K.*

Algorithm designers typically use established design techniques when creating new algorithms. Most current block ciphers are iterated ciphers (Definition 3.2 [33]). Feistel ciphers (Definition 3.3 [33]) are a special instance with a particular structure.

Definition 3.2. *An iterated block cipher is a block cipher involving the sequential repetition of an internal function called the round function. Parameters include the number of rounds r, the block bit-size n, and the bit-size k of the input key K from which r subkeys K_i (round keys) are derived. For invertibility (allowing unique decryption), for each value K_i the round function is a bijection on the round input.*

Definition 3.3. *A Feistel cipher is an iterated cipher mapping a $2t$-bit plaintext (L_0, R_0), for t-bit blocks L_0 and R_0, to a ciphertext (R_r, L_r), through an r-round process where $r \geq 1$. For $1 \leq i \leq r$, round i maps $(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$ as follows: $L_i = R_{i-1}$, $R_i = L_{i-1} \oplus f\,(R_{i-1}, K_i)$, where each subkey K_i is derived from the cipher key K.*

Hardware implementations of iterated block ciphers usually have logic for a single round and a controller that manages the round function iterations. Consequently, several clock cycles will be required before the result is ready. It is important to note that the number of clock cycles needed to encrypt a

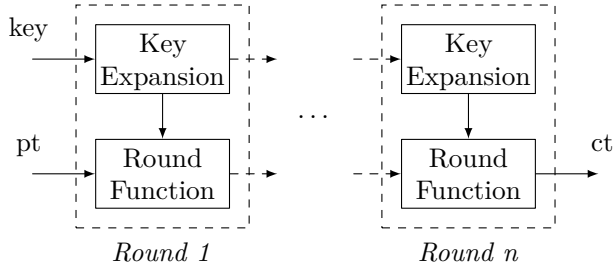

Figure 3.1: Structure of unrolled block ciphers (pt: plaintext, ct: ciphertext).

block is a property of the implementation. One way to reduce the number of cycles is by unrolling the iterations and increasing the amount of combinational logic (Section 3.2.2). When all rounds are fully unrolled, we obtain single-cycle implementations with the basic structure shown in Figure 3.1.

It can be seen from Definition 3.2 that each round has two components: the key expansion and round function. The former generates the subkeys K_i based on the original key, a previous one or a combination of both. The latter transforms the input data using the key. In general, the function is identical for each round, but some algorithms introduce small variations (e.g. a different constant could be added in each round). The total number of rounds depends on the algorithm and can vary widely. An operation is sometimes applied to the plaintext before using it as an input to the first round. The last round's output can be similarly modified before using it as the ciphertext.

3.2.2 Logic Depth

The logic depth [83] of a path is defined as the number of combinational gates between input and output. Since each level of the path has a specific delay associated with it, the logic depth will be linked to the latency of the circuit. However, some operations will have a longer intrinsic delay than others, so that a deep circuit of low-latency gates will have a lower delay than a shallow circuit with high-latency gates. The logic depth is a property of the implementation, which is influenced by the design.

Section 3.3 will give the logic depth of the critical path on FPGA for each algorithm. The critical path of a circuit is the path for which it takes the longest for the output to stabilise [84], i.e., the one with the longest delay.

3.2.3 Fan-Out

The fan-out denotes the number of load gates that are connected to the output of the driving gate [84]. When the fan-out of a gate is large, it will deteriorate performance because the load on that gate will be very high. This impacts its dynamic performance and slows down the circuit. The fan-out of a gate is influenced by the design of the algorithm and how it is implemented. Therefore, a designer should be careful not to reuse a single intermediate result in a next step too often. While the intermediates of such high-load gates can be buffered on ASIC to reduce their negative effect on latency, they will typically increase the critical path on FPGA.

3.3 Synthesis Results

We now discuss the design criteria and specifications of each block cipher, as well as its most important results. The best cryptanalysis results known to us are listed as well. An overview of the properties of all discussed algorithms is given in Table 3.1. Table 3.2 and Table 3.3 give an overview of all FPGA and ASIC results respectively. A diagram of the critical path for each cipher is also drawn. Note that these figures do not show the algorithm's full data flow, but rather a simplified version for clarity.

The regular structure (Section 3.2.1) of block ciphers makes it possible to use a generic approach for unrolling each algorithm. Only the encryption mode of each cipher was implemented. The area cost of adding decryption will depend on the design: this requires less overhead compared to encryption for some than others. Both FPGA and ASIC results are listed, because although most real-world applications will eventually be produced as ASIC, FPGAs are sometimes introduced in products, e.g., because they can be upgraded in the field. They are also heavily used in the development of new hardware.

The FPGA results were obtained after Place and Route (PAR) on a Xilinx Virtex 6 device in Xilinx ISE. More specifically, the configuration of the Xilinx ML605 development board was selected (`xc6vlx240t-2ff1156`). All syntheses for ASIC were done with UMC's 130 nm technology in Synopsys Design Vision.

3.3.1 AES

In 1998, Daemen and Rijmen submitted their Rijndael algorithm [76] to the Advanced Encryption Standard (AES) competition, organised by NIST. Three

Table 3.1: Properties of all implemented algorithms.

Cipher	Key Size	Block Size	Rounds	Type	Characteristics
AES	128	128	10	SP Network	8-bit S-box
KATAN	80	32 64	254		Non-Linear Boolean Functions (`AND` and `XOR`)
PRESENT	80 128	64	31	SP Network	4-bit S-box
PRINCE	128	64	12	Unrolled	4-bit S-box, Matrix Layer
RECTANGLE	80	64	25	SP Network	4-bit S-box
SIMON	64 128	32 64	32 44	Feistel	`XOR` and Left Cyclic Shift
SPECK	64 128	32 64	22 27		`XOR`, Addition, and Cyclic Shift

years later, the design won and it is now known as AES. The implementation criteria for the AES contest were high throughput, low memory requirements, as well as hardware and software suitability [85]. It is used for confidentiality in a wide range of applications, such as the protection of Wi-Fi connections, to secure web traffic, or hard drive encryption. The Rijndael family can accommodate any block and key size from 128 to 256 bits, with steps of 32 bits. NIST fixed the block size at 128 bits, but the key size can be chosen depending on the required level of security (128, 192, or 256 bits) [86].

The algorithm has the following three basic operations: SubBytes, ShiftRows, and MixColumns. SubBytes substitutes a state byte with the result of an S-box look-up. ShiftRows cyclically shifts the state's rows. MixColumns applies an invertible linear transformation to each column. AES was not specifically designed as a low-area or low-latency hardware cipher, but it is included here as a reference because its algorithm is well-understood and generally known.

The best known shortcut attack that works on the full versions of AES is a biclique attack from 2011 [87]. It breaks all 10 rounds of AES-128 with a time complexity of $2^{126.18}$ and data complexity of 2^{88}. These numbers are still high enough to have no practical value.

Our implementation for 128-bit keys uses 8,984 LUTs and has a 24.7 ns combinational delay. On FPGA, logic is responsible for 21.94% of the delay and routing for 78.06%. These ratios will be very different on ASIC, where interconnects are more efficient in terms of delay and gates can be placed more flexibly to minimise routing delay [88, 89]. The logic depth (Section 3.2.2) of the critical path consists of 52 levels. The S-box look-up of each round accounts for three levels, or 30 for our design (10 rounds). A diagram of the critical path for one round is shown in Figure 3.2. The S-box look-up and finite field multiplication are the most expensive components in terms of delay. However,

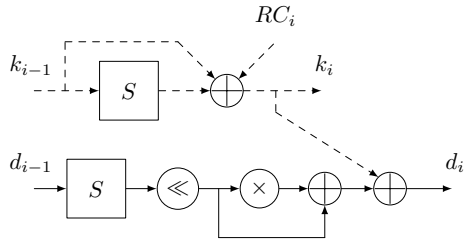

Figure 3.2: Critical path diagram of one unrolled AES round (RC_i: round constant, $\ll$: circular shift, $\times$: finite field multiplier). The dashed part is the key expansion, which does not impact the cipher's latency.

note that the multiplication can be implemented efficiently and without a full multiplier. Although the key expansion for each round is done in parallel with the calculations of the round itself and therefore does not appear on the critical path, it is shown to give an idea of its cost.

The big difference between the logic and routing delay has two causes. First, the main operations on the critical path are look-ups in big 8-bit S-boxes, which have long delays. They incur a total delay, i.e., including both logic and routing, of 11.2 ns, or 45.24%. Second, the input signal to each round has a large fan-out, slowing down the circuit. This is not caused by a design decision here, but rather an effect of how the S-box was synthesised.

All S-boxes were implemented with 8-bit to 8-bit Look-Up Tables (LUTs). This explains the large ASIC area, because LUTs do not map well to ASIC. Note that implementations which rely on composite field arithmetic yield significantly better area results, especially in ASIC [90, 91], but longer critical paths.

3.3.2 KATAN

De Cannière et al. designed KATAN and KTANTAN [77] to be used in RFID tags. Their goal was to build an algorithm with an efficient hardware implementation, while still achieving reasonable throughput. The family of ciphers has a fixed key size of 80 bits, but the block size is a parameter (32, 48 or 64 bits). KATAN uses a Linear Feedback Shift Register (LFSR) for the key expansion. Encryption is done by splitting the state into two parts of different length and applying a non-linear function to each in every round of the algorithm. The only difference between KATAN and KTANTAN is that the latter has a hard-coded key.

Bogdanov and Rechberger [92] first broke the KTANTAN family of ciphers with a meet-in-the-middle attack that has a time complexity of $2^{75.170}$ and data

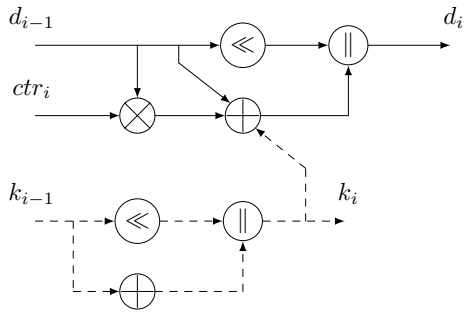

Figure 3.3: Critical path diagram of one unrolled KATAN round (ctr_i: LFSR round counter, $\ll$: regular shift). The dashed part is the key expansion, which does not impact the cipher's latency.

complexity of 3. So far, there are only known attacks against reduced-round versions of KATAN, the best of which was found through multi-dimensional meet-in-the-middle cryptanalysis by Rasoolzadeh and Raddum [93]. It breaks 206 out of 254 rounds of KATAN-32 with a time complexity of 2^{79} and requires three known plaintexts.

Two versions of KATAN were built: KATAN-32 and KATAN-64 use 32-bit and 64-bit blocks respectively. The former requires 1,064 LUTs and has a critical path of 41.2 ns. Although it has a very small area, its practical use is limited by the long delay, due to the large number of rounds. The results for the latter are similar, with 2,550 LUTs and 47.3 ns. On FPGA, 91.0% of the delay is caused by routing, and 9.0% by logic for both variations. The logic depth consists of respectively 62 and 72 levels for the 32- and 64-bit states.

Figure 3.3 shows a diagram of the critical path. The signal runs in parallel through the paths with the left shift, and `XOR` and `AND` gates respectively. Since it does not cost much to implement a shift in hardware, only the latter will be in the critical path. Both the key expansion and LFSR round counter (ctr_i, which is not shown) can be calculated in parallel and are therefore not part of the critical path. Although the round function has a small delay, the large number of rounds explains why a combinational implementation of the overall algorithm is slow.

The `XOR` gates have a nine to one delay ratio of routing to logic. In the Virtex 6 FPGA, they are implemented with 6-input LUTs, which have a constant look-up time of 0.061 ns (the logic delay). The routing delay accounts for the time needed to get the result to the next LUT. Contrary to the constant logic delay, it varies slightly depending on the fan-out (Section 3.2.3) and placement of the design.

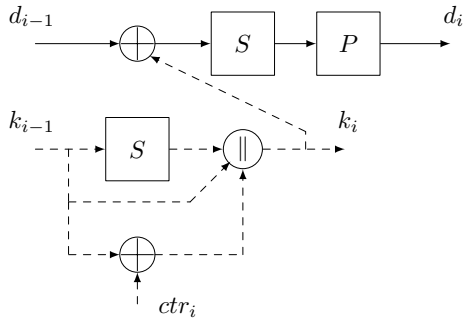

Figure 3.4: Critical path diagram of one unrolled PRESENT round. The dashed part is the key expansion, which does not impact the cipher's latency.

3.3.3 PRESENT

Like KATAN (Section 3.3.2), PRESENT [78] was created as a lightweight block cipher for constrained environments. They have very similar characteristics, but PRESENT has a higher throughput with lower area. In each encryption round, the state's nibbles are run through a 4-bit S-box. This is followed by a permutation layer which moves bits to different positions. The block size is fixed at 64 bits, but both 80- and 128-bit keys can be used. The variation with 80-bit keys takes up 2,089 LUTs and has a 29.2 ns delay. The one with 128-bit keys uses 2,203 LUTs and has a critical path of 32.6 ns. Increasing the key size has a small impact on area and latency.

Full-round attacks exist on both the 80-bit and 128-bit variation, using biclique cryptanalysis. The best result for PRESENT 64/80 was introduced by Faghihi Sereshgi et al. [94] and has a time complexity of $2^{79.34}$ and data complexity of 2^{22}. PRESENT 64/128 was broken by Changhoon Lee [95] with a time complexity of $2^{127.81}$ and data complexity of 2^{19}.

On FPGA, 9.0% of the delay is caused by logic and 91.0% by routing for both key sizes. A diagram of the critical path for one round is shown in Figure 3.4. In each round, it first passes through the `XOR` with the subkey, followed by the S-box look-up and finally the permutation layer. The latter is a very cheap operation in hardware, as it only requires reordering wires. The `XOR` gates have the same characteristics that were mentioned earlier, but the smaller 4-bit S-boxes have a logic and routing delay similar to other gates. The critical path of the 80-bit variation has a logic depth of 48 levels, while the 128-bit version comes in at 52 levels.

3.3.4 PRINCE

PRINCE [79] is the first lightweight block cipher design that focuses on reducing latency. Traditional block ciphers are iterated algorithms with almost identical round functions (Section 3.2.1). This similarity is a big advantage to build compact multi-cycle algorithms, but becomes problematic when the ciphertext needs to be ready in a single cycle. By deciding on an unrolled structure from the start, the design space greatly increases, as there is no need for each round to be identical. An additional requirement for PRINCE was negligible overhead for the decryption mode.

The algorithm has a symmetric design about a centre matrix multiplication. Aside from the addition of the expanded key and round constants, the rounds have two basic operations: a 4-bit S-box and matrix multiplication. The latter is constructed so that every output bit is influenced by three input bits and is implemented as an `XOR` of the selected bits. Three different matrices are used, but only construction of the symmetric matrix M' is given in the original paper. The matrix M is derived from M' by first shifting the input state similarly to AES' ShiftRows before the multiplication. Both the block and key size are fixed to 64 and 128 bits respectively. The 128-bit key input is expanded to 192 bits, so that three different 64-bit keys are available. k_0 and k'_0 are used for pre- and post-whitening respectively, and k_1 as the round subkey.

The key k_0 and a round constant are added first. Then, there are five rounds in which the S-box is applied to the state, followed by a multiplication with M, and again the addition of a round constant and the key k_1 (Figure 3.5). The centre part of the algorithm applies the S-box, multiplies the result with M', and applies the inverse S-box. This is followed by five inverse rounds, where the order of the operations is reversed, and the inverse S-box and M^{-1} are used. The final step is again the addition of a round constant and key k'_0.

Since its publication, the resistance of PRINCE against different attacks has been investigated. The most recent ones are due to Morawiecki [96], Derbez and Perrin [97], Canteaut et al. [98] and Zhao et al. [99]. The best known attack so far is the one by Morawiecki [96]. His meet-in-the-middle approach compromises 10 out of 12 rounds with an online time complexity of 2^{68} and data complexity of 2^{57}. When the reflection parameter α can be chosen, the cipher core, i.e. the algorithm without the pre- and post-whitening keys, is fully broken with a time and data complexity of 2^{41} [100].

PRINCE only needs 1,244 LUTs and has a short critical path of 16.4 ns. It first passes through the three initial `XOR`s, which are combined in a single LUT. In the five regular rounds that follow (Figure 3.5), the S-box look-up and matrix multiplication are also synthesised to a single LUT, as well as the two remaining

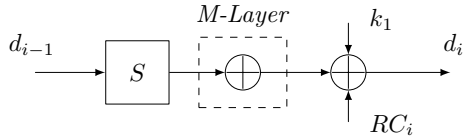

Figure 3.5: Critical path diagram of one regular PRINCE round (RC_i: round constant). The state is multiplied with a matrix in the M-Layer.

XORs. The signal then runs through another S-box look-up and the central matrix multiplication. The remainder of the path is symmetric, due to the cipher's design. On FPGA, routing is responsible for 91.0% of the delay and logic for 9.0%, which can again be explained by the general gate characteristics given earlier. The logic depth of the critical path is 26 levels.

The absence of a complicated key expansion does not impact the critical path, as it can be processed in parallel with the data processing. This was also observed for the other algorithms, where the key expansion never shows up in the critical path. However, it does lower the area requirements of the cipher.

3.3.5 RECTANGLE

Published in 2014, RECTANGLE [80] is the most recent cipher discussed here. It was designed to have good hardware and software performance. The round function is very simple: first, there is an XOR with the round subkey, followed by the application of a 4-bit S-box substitution to the state's columns and a cyclic shift of its rows over different offsets. The key expansion also has these two operations, where the S-box is only applied to the zeroth column of the key state, as well as the addition of an LFSR-generated round constant. The block size is fixed at 64 bits, but there are two possible key sizes (80 and 128 bits).

Since its introduction, few analyses have been published on RECTANGLE. Currently, there is only one report about the variation with 80-bit keys by Shan et al. [101]. Their differential attack breaks 19 out of 25 rounds with a time complexity of $2^{67.42}$ and data complexity of 2^{62}.

The variation with 80-bit keys takes up 1,682 LUTs and has a 19.4 ns delay, while the one with 128-bit keys requires 1,730 LUTs and has a critical path of 19.3 ns. Notice that the latencies for both key sizes are almost identical, confirming that the key expansion is not part of the critical path. For each round, the critical path runs through the XOR with the round key, S-box look-up and circular shift (Figure 3.6). The key expansion can be done in parallel and is only shown to give an idea of its cost. On FPGA, one LUT combines the

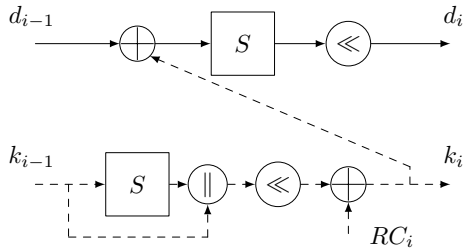

Figure 3.6: Critical path diagram of one unrolled RECTANGLE round ($\ll$: circular shift, RC_i: round constant). The dashed part is the key expansion, which does not impact the cipher's latency.

`XOR`, S-box look-up, and shift. However, the synthesis cannot merge the three operations in some cases, probably due to placement constraints. The final component is the `XOR` with the last subkey (not shown on Figure 3.6). On FPGA, 8.2% of the delay is caused by logic and 91.8% by routing, which is expected given the general characteristics of the gates. The logic depth of the critical path is 41 levels.

3.3.6 SIMON

The designers of SIMON and SPECK (Section 3.3.7) [81] focused on flexibility. Most lightweight block ciphers have a small number of possible block and key sizes. This can make it hard to find a suitable algorithm for a specific application. In contrast, the parameters of SIMON and SPECK give rise to 10 variations. The block size ranges from 32 to 128 bits and the key size from 64 to 256 bits.

SIMON is a Feistel cipher (Section 3.2.1) where the cipher's state is split in half and in each round, the upper part of the input is left unchanged and becomes the lower part of the output. The round function is applied to the lower part and assigned to the upper part of the output. SIMON's round function is very straightforward: it has just three cyclic shifts, three `XOR` gates, and one `AND` gate. The key expansion is slightly more complicated, but uses similar building blocks as the round function.

SIMON and SPECK have been analysed for mathematical weaknesses using a variety of techniques [102], but none have broken the full cipher so far. Note that some publications are limited to a set of specific parameter pairs. The best result for SIMON 32/64 at this time is a linear super-trail attack by Ashur [103] which breaks 24 out of 32 rounds with a time complexity of $2^{63.57}$ and data complexity of $2^{31.57}$.

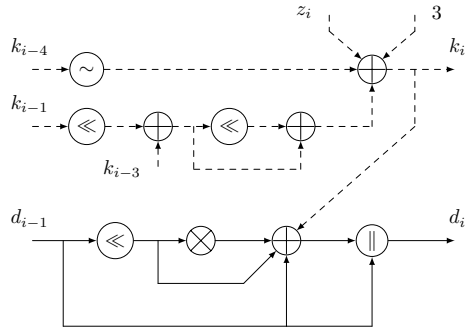

Figure 3.7: Critical path diagram of one unrolled SIMON round ($\sim$: inverter, $\ll$: circular shift, z_i: bit from a predefined constant vector). The dashed part is the key expansion, which does not impact the cipher's latency.

We implemented two parameter pairs: one with 32-bit blocks and 64-bit keys and one with 64-bit blocks and 128-bit keys. The former needs 960 LUTs and has a critical path of 20.4 ns. The latter uses 2,688 LUTs and the output is ready after 27.3 ns. The critical path runs through a circular shift, `AND`, and `XOR` gate (Figure 3.7). Again, the key expansion is not part of the critical path, but is only included in the diagram to show its cost. The `XOR` and `AND` operations in each round are combined in a single LUT. On FPGA, 90.0% of the delay is caused by routing and 10.0% by logic for both variations, which is the ratio we have seen for the other designs as well. The logic depth of the smallest variation consists of 34 levels and 46 levels for SIMON 64/128.

3.3.7 SPECK

SPECK was published together with SIMON (Section 3.3.6), and although both perform well in general, SIMON was optimised for hardware implementations and SPECK for software. The state is also split in half in SPECK's design, but it is not a Feistel cipher, so both halves change in each round. The round function has even fewer operations than SIMON's, but an important difference is the use of one adder. Although trivial in software, this design decision has a significant impact on hardware performance, as can be seen from the results.

The cryptanalysis overview by the authors of SIMON and SPECK [102] also lists results for different variants of SPECK. The best known attack against SPECK 32/64 breaks 15 out of 22 rounds with a time complexity of $2^{61.41}$ and data complexity of $2^{29.41}$ [104].

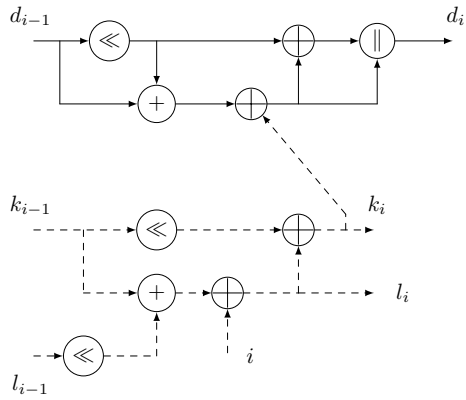

Figure 3.8: Critical path diagram of one unrolled SPECK round ($\ll$: circular shift, i: round counter). The dashed part is the key expansion, which does not impact the cipher's latency.

Implementations were built for the same two parameter pairs as were used with SIMON (Section 3.3.6). SPECK 32/64 requires 1,513 LUTs and has a 40.3 ns delay. SPECK 64/128 uses 3,594 LUTs and has a critical path of 50.3 ns. The components of the critical path differ between the rounds depending on the possible optimizations after placement. In general, it runs through the circular shift, adder chain, and finally the `XOR` gates (Figure 3.8). Comparing the delay for both variations, we can clearly see the impact of the adder. On FPGA, logic is responsible for 33.0% and the wiring for 67.0% for both variations. This is due to the adders introducing longer logic delays than the basic gates that were used in all other algorithms. The critical path of SPECK 32/64 has a logic depth of 124 levels, while SPECK 64/128 comes in at 197 levels. The total delay caused by the adders is 26.4 ns (65.53%) and 32.8 ns (65.28%) respectively.

3.4 Comparison

Table 3.2 summarises all FPGA results from the previous section, grouped by block and key size. Looking at the ciphers with 32-bit blocks, SIMON 32/64 has the best performance both in terms of area and throughput. An important disadvantage are the 64-bit keys, offering only very short-term protection against small organizations [105]. While KATAN 32/80 uses stronger keys and has similar area requirements, its large number of rounds results in high latency.

Among the algorithms with 64-bit blocks and 80-bit keys, RECTANGLE is the smallest and has the shortest latency too. PRESENT has similar characteristics

because they use the same techniques. The difference between the two is only caused by their S-box design and permutation layer. Although KATAN's area is still quite small for these parameters, its latency is the second-highest of all implementations. The reason for the higher throughput is the bigger block size.

Comparing the results for the last parameter pair (64-bit blocks, 128-bit keys), PRINCE's performance really stands out. It is by far the smallest in its category and not even that far off SIMON 32/64. The latency is the lowest of all implemented ciphers, which confirms its main design requirement. The numbers for PRESENT and SIMON are similar, with PRESENT having a slightly smaller footprint and SIMON being a bit faster. However, as the area of the latter increases with the parameter size, the variations with small parameters are most interesting. The circuit is compounded by a large number of additional rounds when the size of the parameter goes up. SPECK's results don't make it an attractive alternative. The critical path is particularly long because of the adders in its design.

Looking at the different lightweight ciphers, the performance of AES is surprisingly good. It has a very large area because of the big S-boxes (8-bit to 8-bit), but its latency is competitive, given the small number of rounds and efficient permutation layer. Combined with the 128-bit blocks, this results in high throughput.

Most ASIC results are in line with the expectations from FPGA. The biggest surprise is SPECK's area being smaller than SIMON's, both for 32- and 64-bit blocks. A possible explanation for this difference is that the adders can be mapped better on ASIC than FPGA. Also note that the latency for SPECK 64/128 is very high on ASIC.

It is now possible to make some observations on the design of lightweight ciphers. Unrolling the rounds of an iterated cipher places all data operations of the round function on the critical path. Therefore, when an algorithm has more rounds, the critical path will often be longer as well (Figure 3.9). This is clear from the results for KATAN, which has a very large number of rounds. It is well known that regular arithmetic does not perform well in hardware, especially in terms of latency. SPECK's performance is a clear indication of this. Big S-boxes are also expensive, and as can be seen from the AES implementation, they have a large area requirement, especially in ASIC. Additionally, because they don't map well to the FPGA fabric, they have very long delays. The number of S-boxes used in the round function is of less importance, as they are working in parallel. Depending on the platform, using multiple-input gates could also negatively impact the latency (e.g. a four-input `XOR` can be implemented in a single LUT on FPGA, while it will result in a cascade of three `XOR`s in ASIC).

Table 3.2: Area, critical path latency, and throughput of the full cipher on FPGA (italics: best result in a security class, bold: best result overall). All results are given for an unrolled implementation on a Xilinx Virtex 6 device in Xilinx ISE.

	Cipher	Area [LUTs]	Latency [ns]	Throughput [Gbit/s]
32/64	SIMON	**960**	*20.4*	1.46
	SPECK	1,513	40.3	0.74
32/80	KATAN	1,064	41.2	0.72
64/80	KATAN	2,550	47.3	1.26
	PRESENT	2,089	29.2	2.04
	RECTANGLE	*1,682*	*19.4*	3.08
64/128	PRESENT	2,203	32.6	1.83
	PRINCE	*1,244*	**16.4**	3.64
	RECTANGLE	1,730	19.3	3.08
	SIMON	2,688	27.3	2.18
	SPECK	3,594	50.3	1.19
128/128	AES	8,984	24.7	4.82

Table 3.3: Area, critical path latency, and throughput of the full cipher on ASIC (italics: best result in a security class, bold: best result overall). All results are given for an unrolled implementation and were obtained for UMC's 130 nm technology in Synopsys Design Vision.

	Cipher	Area [GE]	Latency [ns]	Throughput [Gbit/s]
32/64	SIMON	8,432.00	*29.6*	1.00
	SPECK	**5,893.25**	82.1	0.36
32/80	KATAN	11,939.50	61.2	0.49
64/80	KATAN	24,766.50	75.8	0.79
	PRESENT	22,063.50	39.4	1.51
	RECTANGLE	*18,160.75*	*34.87*	1.71
64/128	PRESENT	23,005.75	38.1	1.57
	PRINCE	*9,522.75*	**22.9**	2.60
	RECTANGLE	18,935.00	34.68	1.72
	SIMON	23,584.00	41.7	1.43
	SPECK	16,371.00	182.4	0.33
128/128	AES	126,571.00	61.6	1.93

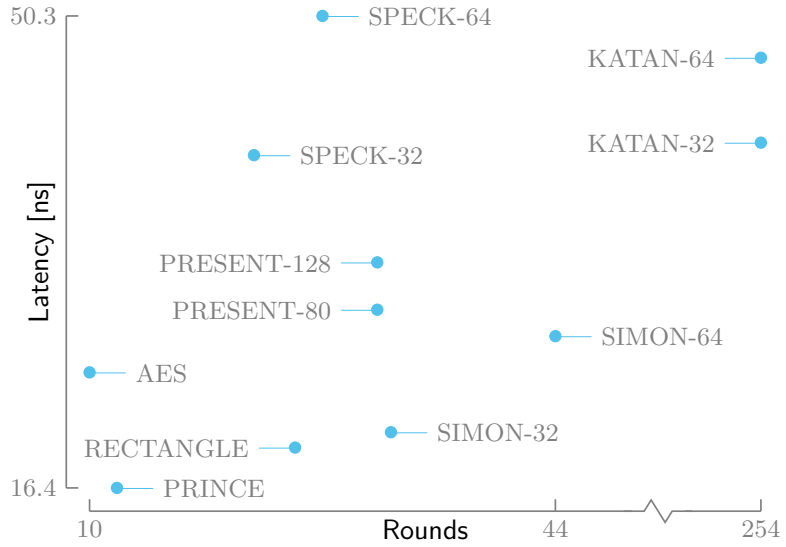

Figure 3.9: The critical path latency of unrolled block cipher implementations on FPGA, in function of their number of rounds. In general, algorithms with fewer rounds will have lower latency, as unrolling an iterated cipher will result in the critical path running through the data operations.

Finally, recommendations for the design of low-latency algorithms follow from these remarks. When focusing on low latency, having an unrolled design, like PRINCE, gives significantly better results. Iterated SP networks also perform well: the delay of small S-boxes is not very high and the permutation layer can essentially be implemented for free. The number of rounds should be as low as possible, while still maintaining an acceptable level of security. Small S-boxes are an interesting component, as they have low latency as well as good area performance. Lastly, the general design rule to use boolean operations in hardware designs also applies here.

3.5 Conclusion

In this chapter, we have given synthesis results for unrolled implementations of six families of lightweight block ciphers, along with AES for reference. It was shown that PRINCE, the only cipher specifically designed to have low latency, is the fastest of all implemented algorithms, and also has a very competitive area. For smaller block sizes, which are useful for some applications, SIMON

has the smallest area and offers good throughput. However, the latency of most ciphers is too high to be useful in practice. For example, PRINCE runs at 61.039 MHz on the Virtex 6, which is fast compared to the other ciphers, but is suitable only for small embedded applications. The attainable speed in a microcontroller will be even lower once it is integrated with other components that add to the critical path.

The search for new lightweight low-latency ciphers therefore remains an important future research topic. Furthermore, ciphers are rarely used in isolation, but rather as a building block in a certain mode of operation, and the design of this mode will consequently impact the overall performance. In this context, it is interesting to note that the currently-running NIST Lightweight Cryptography competition required submissions to be either an Authenticated Encryption with Associated Data (AEAD) primitive, which simultaneously encrypts and authenticates messages, or a hash function.

4 Eleutheria: Lightweight Key Distribution Service

Trusted computing architectures often rely on symmetric ciphers like those discussed in Chapter 3 and therefore require shared secrets. Consequently, many designs generate a unique key for each device at manufacturing time, which is then fused into the hardware. However, one important issue is how to manage and distribute this key, as symmetric secrets need to be shared with all communicating parties. To this end, architectures typically store a copy of these device keys with a remote party, where they could be compromised, requiring all devices to be replaced.

We present an alternative hardware-based solution, where the device key is never stored remotely. Instead, the device runs a network service that can be used by SPs to request the derivation of an application-specific key. This key derivation is performed in hardware, directly accessing the device-specific key, which is never exposed to software. The derived key is transmitted securely to the SP through asymmetric cryptographic algorithms that are implemented in software. Combining hardware implementations of symmetric ciphers for common operations with infrequent calls to software-based asymmetric algorithms, enables high-performance secure applications on embedded devices.

Content Source

P. Maene and I. Verbauwhede, *Eleutheria: Lightweight Key Distribution Service for Networked Embedded Devices*
Contribution: Main author.

4.1 Introduction

With the advent of the IoT, previously unconnected devices are being given network access. For example, Wireless Sensor Networks (WSNs) monitor environmental parameters throughout large areas, relying on efficient networking technologies to interconnect the nodes. Furthermore, modern cars feature tens of microcontrollers which share information and drive actuators through a Controller Area Network (CAN). However, networking such devices also exposes them to remote attackers, no longer requiring physical access to the software running on them. In August 2016, the Mirai malware infected millions of IoT devices, like IP cameras, home routers, and printers, turning them into a botnet to launch large-scale DDoS attacks [107]. Attackers have also managed to take control of cars [108] and even disrupted industrial control systems (Chapter 1).

In response, researchers and industry have been working on security mechanisms for lightweight devices, as detailed in Chapter 2. The protection offered by these architectures is based on hardware implementations of one or more cryptographic primitives. Designs for resource-constrained devices generally rely on symmetric algorithms, as asymmetric cryptography is too expensive for regular operations on resource-constrained platforms. However, this means that they need some way to share a key with users of the platform. One approach is to program it in hardware during manufacturing and share it with the device's owner. As mentioned, copies of these device keys are typically stored with a remote third party, where they could be compromised by attackers.

In this chapter, we present Eleutheria (Ancient Greek for *freedom*), a hardware-based key distribution service where no copies of the device key are stored after it has been programmed into the device, simplifying key management at the same time. Our solution consists of a software-based network service receiving key requests and a hardware component which derives application-specific keys, reducing the risk of compromise when fixed symmetric keys are used in embedded devices. Eleutheria relies on infrequent use of asymmetric cryptography implemented in software to simplify key management, allowing application keys to be requested after deployment. The software component of our service and the application for which the key is requested are included in the key derivation, implicitly authenticating both. Attackers therefore cannot modify either of them without changing the resulting key. This software component can be verified during the initial key request through attestation.

First, we give a more detailed problem statement in Section 4.2 and discuss the design of Eleutheria (Section 4.3). Next, Section 4.4 details our implementation for the Zynq, followed by the evaluation in Section 4.5. Finally, we list related work (Section 4.6) and conclude the chapter (Section 4.7).

4.2 Problem Statement

As introduced in Chapter 2, security architectures for lightweight devices often rely on symmetric cryptography to realise their functionality, because it is more efficient and requires fewer resources than asymmetric algorithms. However, this also means that the symmetric key generally needs to be shared with a remote third party (Section 4.2.1). Next, Section 4.2.2 lists our assumptions regarding all involved parties, followed by a description of the attacker model we consider in Section 4.2.3.

4.2.1 Symmetric Device Keys

Since hardware implementations of asymmetric cryptographic algorithms are still expensive in terms of efficiency and resources, most hardware-based security architectures for embedded systems rely on symmetric ciphers. For example, SMART (Section 2.5.5) is a lightweight remote attestation mechanism, allowing an external third party to obtain evidence about the device state (Section 2.4.1).

During its attestation protocol, SMART calculates an HMAC in software using the SHA-1 hash function, which is symmetrically keyed with $\mathcal{K}$ stored in protected memory. This key should not leak, as this would compromise any authenticity guarantees given by the attestation. On the device, this attestation key is therefore protected by dedicated memory access control logic, ensuring that it can only be read by the attestation routine. However, it still needs to be known by the remote verifier to check the HMAC.

While the authors extensively consider the security of the on-device key storage, they do not discuss the implications of the verifier having to store these keys too. Of course, this is justified, as such devices are typically installed in locations where adversaries have easy physical access, whereas the verifier can save the keys on a server in a restricted location. However, if this server were to be compromised, either by local or remote attackers, all keys stored there would leak. This would require all devices to be replaced or reprogrammed, as the protocol's security hinges on the confidentiality of this shared secret. Particularly in larger deployments, this could be a very expensive operation.

Recall that Sancus is a PMA for embedded devices (Section 2.5.6). Its cryptographic unit is also based on the HMAC algorithm using the SPONGENT hash function. Recall that each node is again given a key K_N, which is fixed in hardware. Modules are assigned unique keys by calculating two subsequent key derivations as defined by Equation 4.1.

$$K_{N,SP} = kdf(K_N, SP)$$
$$K_{N,SP,SM} = kdf(K_{N,SP}, SM) \tag{4.1}$$

Here, *SP* is the identity of the SP and *SM* the one of the software module. Sancus' system model assumes that all nodes are owned by a so-called IP. In order to distribute keys to SPs, this IP again needs to extract the node keys K_N and store them in a database. Similarly to SMART, the security of all nodes would be compromised if an adversary were to ever obtain this database.

SMART and Sancus are two examples of architectures where our solution could improve the management and security of the key distribution. However, Eleutheria is not limited to either, and could be applied to any hardware-based security mechanism relying on a fixed symmetric secret which needs to be shared with an external party (e.g., the CFI architecture SOFIA [18]).

4.2.2 System Model

The goal of Eleutheria is to increase the security of key distribution for embedded devices relying on a fixed, unique key, also improving key management at the same time. Similar to Sancus, we consider multiple SPs who want to run applications on a distributed infrastructure (Figure 4.1). They use symmetric cryptography to communicate confidentially with their deployed applications. At a high level, there are therefore two parties involved in this scenario. First, the SPs who want to run their applications on the provided nodes. Second, all devices are owned and managed by an IP.

The IP installs the devices throughout an unrestricted area and connects them to a network, enabling remote access. This is also the only way for SPs to interact with the device, as they are never given physical access. Each device has a fixed, unique key K_D which is initialised during manufacturing. In contrast to existing solutions, K_D is never extracted. SPs require a unique key per application to exchange secrets with it, which is derived from K_D. Authorised SPs can communicate directly with the devices to request application keys. Finally, there should also be a communication channel between the SPs and IP.

We assume that the devices consist of typical modern SoCs featuring an embedded processor, memory, and flash storage. They can range from lightweight embedded systems with a low-frequency microcontroller, such as a wireless sensor node, to more powerful platforms. Note that our proposed solution is not specific to a single architecture and can be adapted to others.

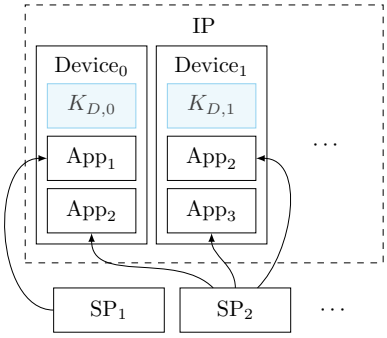

Figure 4.1: The Infrastructure Provider (IP) deploys and manages all devices, providing remote access to them. Each device has a fixed, unique key K_D, generated during manufacturing. The Software Providers (SPs) can have their applications deployed by the IP and request a unique key to be derived for each.

4.2.3 Attacker Model

Eleutheria protects against an attacker who wants to learn confidential data communicated between the SPs and the node, by compromising either the fixed key K_D or any derived application keys. The common model that was identified in Section 1.2.2 is also applicable here, with this section reiterating and expanding the relevant assumptions.

Regarding the cryptographic primitives, we assume the Dolev-Yao model, i.e., an attacker is allowed to perform protocol-level attacks but cannot break the used algorithms [8]. At the system level, the adversary is therefore allowed to attack secure channels established between the different parties, both actively and passively. However, both the SPs and IPs are assumed to have access to a limited amount of secure storage which cannot be breached by the attacker.

He also has full control over any software running on the node, but the architecture should feature memory access control. Section 4.5.3 discusses this limitation in more detail. Additionally, we assume that its processor does not allow data execution, e.g., through separate code and data memory ranges or Data Execution Prevention (DEP).

Finally, the attacker cannot tamper with the device's hardware. For instance, he is not allowed to attack the CPU physically, tap the bus, or dump the contents of its volatile memory. We also consider side-channel attacks on the algorithm implementations out of scope, in hardware or software, e.g., Differential Power Analysis (DPA), fault attacks, or cache timing attacks. In addition, we do not consider DoS attacks, neither at the network level nor locally.

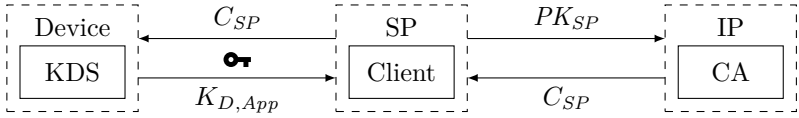

Figure 4.2: There are three parties in Eleutheria's system model, each of them using different components. The Infrastructure Provider (IP) issues authorization certificates to Software Providers (SPs), which can then be used to negotiate a secure channel with the Key Distribution Service (KDS) and request the derivation of $K_{D,App}$.

Although the IP is central to the system model, he is regarded as a third party by the SP, and assumptions made about his behaviour should be specified. We model the IP as honest-but-curious, following the specification without acting maliciously, but he does try to learn as much information as possible. Similarly, we assume that the generated device keys are discarded after programming them into the hardware.

4.3 Design

Figure 4.2 shows the three parties involved in Eleutheria together with their respective components. Next, Section 4.3.1 discusses how the IP authorises SPs to request application keys. We then explain the preparation of binaries for deployment, and how SPs communicate with the hardware-based Key Distribution Service (KDS) running on the device (Section 4.3.2). The design of its software and hardware components are detailed in Section 4.3.3 and Section 4.3.4 respectively.

4.3.1 Infrastructure Provider

Aside from providing the device infrastructure, the IP also authorises SPs to request keys. One approach would be to include SP identities directly in the KDS deployed to each device. However, this would not scale to large device swarms, as each device would have to be updated whenever privileges are added or revoked. Rather, the IP acts as a Certificate Authority (CA) and generates a certificate to give an SP access to the KDS. The certificate's metadata can include additional information to further restrict the grant's validity. For instance, the SP could be asked to include an application identifier in the certificate requests, only allowing requests for that application, or access could

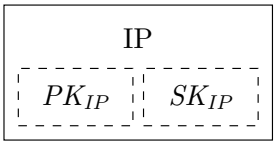

Figure 4.3: The Infrastructure Provider (IP) authorises Software Providers (SPs) to request keys by creating certificates signed with SK_{IP}. The public key PK_{IP} is included in the Key Distribution Service (KDS) running on the device.

be restricted to specific devices. Furthermore, if an SP were to be compromised, its certificate can be revoked, ensuring that an attacker cannot impersonate the SP and request its keys or deploy malicious applications.

In order to sign certificates, the IP has his own asymmetric key pair (PK_{IP}, SK_{IP}). The private key SK_{IP} should be stored securely (e.g., in an HSM), while the public key PK_{IP} is included in the KDS deployed to the device. Since this is a long-term key, the devices would only need to be updated infrequently. When an SP wants authorization to access the KDS, it sends its public signature key PK_{SP} to the IP (Section 4.3.2). The IP then creates the certificate C_{SP} for the received key and signs it with his private key SK_{IP}. As an attacker could send forged certificate requests, they should be authenticated by the SP, e.g., by signing them with a different long-term key pair shared with the IP. However, the communication between the IP and SP does not need to be secure, as no secrets are exchanged. Alternatively, certificate requests could also be sent out-of-band.

4.3.2 Software Provider

Requesting a unique key $K_{D,App}$ from the device, for instance to share confidential information with the application, requires three steps. First, the SP generates an asymmetric key pair (PK_{SP}, SK_{SP}), sending the public key to the IP for certification. As discussed in Section 4.3.1, IP verifies the identity of the SP and creates the certificate C_{SP}. This certificate can be verified by the KDS, authenticating the SP to the device. Second, the SP needs to sign his application binary with the certified signature key. Third, he sets up a secure session with the KDS to request the derivation of $K_{D,App}$.

After receiving the certificate C_{SP} from the IP, the SP uses his private key SK_{SP} to sign the application binary that will be deployed to the device. There are two main design reasons for including this signature $SIG_{SP}(App)$ in the application's binary. First, the KDS will verify it before starting the key derivation when the

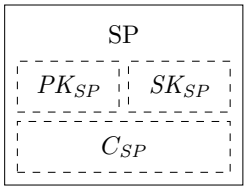

Figure 4.4: The Software Provider (SP) generates an asymmetric key pair, sending the public key PK_{SP} to the Infrastructure Provider (IP), which then creates the certificate C_{SP}. This key pair is used to sign application binaries and to authenticate SP to the Key Distribution Service (KDS).

SP requests a key. Receiving the application key attests the binary's integrity at the time of the derivation. Second, it allows multiple SPs to run the exact same application, as the signature will be unique across providers, therefore resulting in different binaries and unique keys $K_{D,App}$ (Section 4.3.4).

After deriving the application key, the device needs to send it back to the provider. Since this key should be kept secret, they have to communicate over a secure channel. At the start of a session, the KDS and SP therefore go through a key exchange protocol to establish a session key that will be used to protect the confidentiality and integrity of the channel. During this protocol, both parties should mutually authenticate themselves. To this end, the SP will send C_{SP} to the KDS, which verifies it using PK_{IP}. In addition, the device is also provisioned with a key pair certified by the IP, as will be discussed in Section 4.3.3. This device certificate C_D is used during the key exchange to authenticate the device to the SP.

4.3.3 Key Distribution Service

In order to enable SPs requesting application keys, all devices run a service which can be connected to over the network. This Key Distribution Service (KDS) is provisioned by the IP when setting up the infrastructure and the SP can initially verify its integrity through remote attestation. Aside from a software part, the KDS relies on two hardware components. First, each device is assigned a unique key K_D, generated during manufacturing and fixed in hardware. This generated key is discarded afterwards and cannot be read by the IP or SP. Second, the Key Derivation Mechanism (KDM) derives the application key $K_{D,App}$.

To keep the hardware cost low, the network service which handles remote key requests from SPs was designed as a software component. As mentioned in

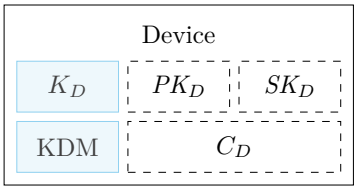

Figure 4.5: The Key Distribution Service (KDS) running on the device consists of software (dashed) and hardware components. The former is responsible for establishing secure sessions with Software Providers (SPs), while the Key Derivation Mechanism (KDM) performs the actual key derivation in hardware. The device key K_D is generated by the manufacturer.

Section 4.3.2, a secure channel is established between the KDS and the SP. An asymmetric key pair (PK_D, SK_D) is therefore generated by the IP for each device before deployment. The public key is also certified by him, resulting in C_D which is used to authenticate the device to the SP in the key exchange protocol. The KDS also verifies the certificate C_{SP}, which authorises the SP to request keys (Section 4.3.1), closing the connection upon failure. After completing the protocol, during which PK_{SP} was received as part of C_{SP}, the KDS first checks the signature $SIG_{SP}(App)$ of the application for which the key request was issued. If the verification passes, the service invokes the hardware-based KDM to derive $K_{D,App}$. Otherwise, the connection to the SP is closed. Finally, the derived key is securely transmitted back to the SP. This approach has the advantage that only PK_{IP} needs to be included in the KDS. SPs are authenticated through their certificate, and PK_{SP}, needed to verify $SIG_{SP}(App)$, is received as part of the key exchange protocol.

Deployed applications should also have access to their key $K_{D,App}$ when encrypting or decrypting data sent to or received from the SP. Rather than calling the KDS as well, they invoke the hardware directly. The KDM ensures that applications can only request their own key and not that of others. Note that the binary's signature is not verified before this derivation.

Once the device has been provisioned with its key pair (PK_D, SK_D) and the KDS, the IP first connects to the device and requests an attestation key $K_{D,KDS,SP}$. After the device has been deployed, this key can be used to attest that the original binary is still running. This only needs to be done when an SP initially requests an application key, as any modifications afterwards would result in a different $K_{D,App}$. One possible approach would be for the IP to encrypt $K_{D,KDS,SP}$ asymmetrically using PK_{SP}. The SP's identity is included in the derivation of $K_{D,KDS,SP}$, ensuring its uniqueness. Before requesting $K_{D,App}$ for the first time, the SP runs a challenge-response protocol, sending a nonce to the

device, which encrypts it using $K_{D,KDS,SP}$ and sends the ciphertext back [109]. Since the SP received $K_{D,KDS,SP}$ from the IP, it can also encrypt the nonce and compare the result to the received ciphertext, asserting the integrity of the KDS. The memory access control protects the KDS after this attestation protocol has been completed, but before the SP's key request is received.

4.3.4 Key Derivation Mechanism

The actual key derivation is done in hardware by the KDM, which itself performs two main functions. First, it calculates $K_{D,App}$ from K_D using a hardware implementation of a Key Derivation Function (KDF). Second, its Execution Monitor (EM) tracks which application is currently running on the processor to restrict access to itself. Doing this in hardware ensures that K_D can be wired directly to the KDF, never exposing it to software. Furthermore, the EM prevents software attackers from requesting unauthorised keys. To prevent access glitches, it is important that the EM is started before the KDF.

Key derivation enables the computation of a set of new keys from older keys [110]. Different key derivation methods have been proposed, based on symmetric encryption algorithms as well as one-way functions. The latter approach is used in Eleutheria, relying on a hash function to derive unique application keys from K_D. In addition to the device key K_D, the application binary and its signature by the SP are included in the derivation, as well as the KDS binary:

$$K_{D,App} = kdf(K_D, App, SIG_{SP}(App), KDS) \tag{4.2}$$

The application itself and its signature are included to ensure that the derived key is unique for each application. Even when identical binaries would be deployed by different SPs, recall that adding the signature to the hash will result in a unique key, because each binary was signed with a different key pair (PK_{SP}, SK_{SP}). Furthermore, if an attacker tampers with the application, its binary will change and $K_{D,App}$ will not be calculated correctly. Consequently, the attacker would not be able to decrypt any secrets protected with this key, nor could he send forged data to the SP, who would still be using the key from the initial request. Recall that the application's signature was verified at that time, proving to the SP that it had not been tampered with. Similarly, the KDS is included so that the derived key would change when an attacker tampers with the service's binary. The key derivation therefore implicitly authenticates both the application and the KDS, requiring the original binaries to be running on the device for the correct key to be derived.

To prevent an attacker from requesting keys for any application, it should only be possible to derive $K_{D,App}$ when either the application or the KDS are

running. Therefore, the EM should monitor that the processor is executing code from their respective memory ranges. Consequently, when a malicious application tries to request a key for any application other than itself, access to the hardware should be denied. The KDS range is monitored separately so that keys can be derived when requests from SPs are serviced.

4.4 Implementation

Eleutheria was prototyped as an IP core for Xilinx's Zynq architecture, which features an Arm Cortex-A9 processor combined with FPGA fabric, referred to as the Processing System (PS) and Programmable Logic (PL) respectively. All device code is running on the Arm and was implemented in C, while the software components for the IP and SP were written Go.

Although we built an IP core prototype, our design can be trivially adapted to integrate directly with the processor (e.g., Sancus' modified openMSP430). Compared to an instruction set extension, the hardware of our implementation is more complex, in particular the EM. However, it is much easier for third parties to integrate an IP core in an SoC than to modify the processor's architecture.

This section first gives some background on the cryptographic protocols and algorithms used in our prototype (Section 4.4.1). The software running at the IP, SP, and on the device are discussed next in Sections 4.4.2, 4.4.3, and 4.4.4 respectively. Finally, the hardware implementation of the KDF and execution monitor are detailed (Section 4.4.5). All interactions between the different parties (Section 4.3) are again shown schematically in Figure 4.6.

4.4.1 Background

Throughout our design, software implementations of asymmetric cryptographic algorithms and protocols are used. The SIGMA key exchange protocol was selected to establish secure communication channels. Additionally, efficient public-key algorithms based on Elliptic Curve Cryptography (ECC) were chosen, as our solution is designed for lightweight devices. For the same reason, we rely on a lightweight block cipher-based MAC algorithm. Eleutheria also features an FPGA-based True Random Number Generator (TRNG) to generate high-quality randomness, which is not readily available on many embedded platforms.

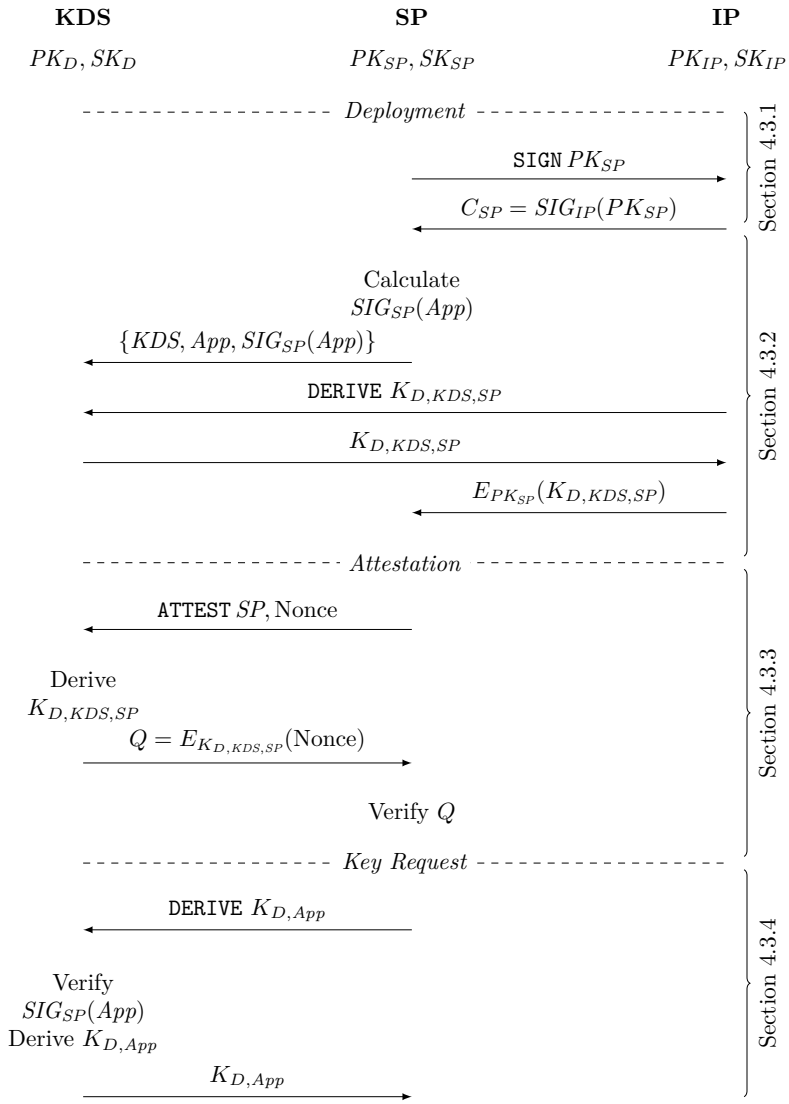

Figure 4.6: Overview of the interactions between the parties involved in our design during the deployment, attestation, and key request stages. Without loss of generalization, only one application is deployed in this figure. Additionally, the steps performed during the SIGMA key exchange protocol (Section 4.4.1) have been left out.

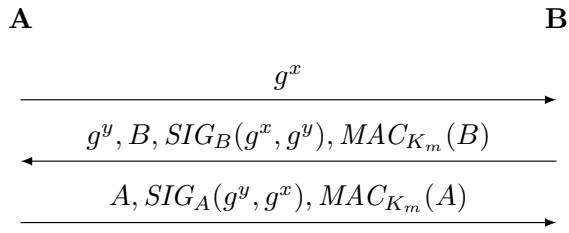

Figure 4.7: Messages exchanged during the basic SIGMA protocol [111]. Here, A and B are the participants' identities, g^x and g^y are the Diffie-Hellman exponentials, SIG is an asymmetric signature, and MAC_{K_m} is an authentication tag with key K_m.

SIGMA

The SIGMA family of protocols by Hugo Krawczyk [111] are authenticated Diffie-Hellman key exchanges which provide perfect forward secrecy. The motivating reason for their design was prevention of identity-misbinding attacks. Eleutheria relies on the basic variant of the protocol without identity protection to exchange session keys when establishing secure channels.

Note that this exchange involves several cryptographic mechanisms. Furthermore, rather than using the Diffie-Hellman value g^{xy} directly, a key K_m is derived from it to calculate MACs, along with a session key K_s. Curve25519 [112] is used for the Diffie-Hellman exchange in our implementation because of its performance as well as its short key sizes. The related Ed25519 digital signature scheme [113] protects the exponentials from modification. LightMAC [114] is a lightweight and efficient authentication primitive based on block cipher calls. SHA-256 [115] handles deriving K_m and K_s by hashing the concatenation of g^{xy} and a key identifier. Finally, certificates are used as the identities A and B, again relying on Ed25519 for the authority signatures.

LightMAC

LightMAC [114] is a mode of operation for block ciphers to generate authentication tags. This gives application designers a lot of flexibility, as the cipher can be trivially changed depending on the constraints of the target platform. At a high level, the message is split into blocks of size $n - s$, where n is the cipher's block size and s is the length of a counter. The value of the counter is prepended to each message block before it is encrypted with the key K_1. After each block cipher call, the ciphertext is XORed with the previous

one. After processing the last block, the result is encrypted using a second key K_2 and the resulting ciphertext is truncated to the desired tag length t.

Eleutheria's implementation uses AES-128 [116], as it is well-established and can be implemented efficiently in software. Considering that the transmitted messages are reasonably short and that counter-message pairs need not be unique across the key, a 16-bit counter is used. Therefore, 112 message bits can be processed during each block cipher call, limiting the overhead. Straightforward big-endian encoding is applied to the counter value before it is prepended. The final ciphertext is not truncated, retaining the full 16-byte tag. SHA-256 is again used to derive the two required keys K_1 and K_2 from a single 128-bit key.

True Random Number Generator

One approach to generating random numbers is the use of a Pseudo-Random Number Generator (PRNG). As its name implies, this is a deterministic algorithm which output can be predicted if the seed is known. Seeding a PRNG in an embedded system is difficult due to the lack of entropy sources. In contrast, TRNGs are typically based on physical noise and generate truly unpredictable bit sequences. One class of TRNGs are based on sampling ring oscillators, which have been shown to perform well on FPGA [117]. We have included a basic ring oscillator-based TRNG in our design to generate randomness for the cryptographic primitives.

4.4.2 Infrastructure Provider

As discussed in Section 4.3.1, the IP authorises SPs to request keys by creating a certificate for the SP's public key. To this end, we built a basic CA based on fast and short Ed25519 signatures in Go, using Adam Langley's open-source implementation [118]. Before deploying devices, the IP first generates the authority key pair (PK_{IP}, SK_{IP}). Since this is an Ed25519 key pair, the public and private key are respectively 32 and 64 bytes long. Recall that this public key is also included in the KDS so that it can verify the certificate's signature and the implied authorization grant.

After the SP has generated the key pair (PK_{SP}, SK_{SP}), the IP creates C_{SP} by signing the public key. Our prototype does not make use of certificate metadata (e.g., expiration time or application identifiers), but any additional information would also need to be included in the signature, if it were added. Finally, the public key and signature are JSON-encoded, representing byte sequences with hexadecimal characters, and sent to the SP.

4.4.3 Software Provider

Before an application is deployed, the SP has to sign it with his private key SK_{SP} (Section 4.3.2). Currently, the KDS and any deployed applications are compiled into a single bare-metal binary running on the Arm. This binary is an ELF file with the KDS residing in the text section and all code of each application assigned to specific sections. One of the tools provided by Xilinx for their FPGAs is a software IDE. Its GCC-based toolchain for Arm processors first compiles all source files to object files before linking them together in the final binary. Since the linker modifies the compiled objects (e.g., to fix up symbol addresses), application signatures can only be calculated after the link step. However, they cannot be added to the final binary, as this would shift its contents, breaking memory addressing. We therefore first generate a binary object with a 64-byte section for each Ed25519 application signature. These generated objects are then linked together with the original source objects into a single bare-metal binary. Next, the contents of each application section are read to calculate $SIG_{SP}(App)$. Finally, the allocated signature sections in the ELF file are updated with $SIG_{SP}(App)$.

Another advantage of manipulating the ELF file is that the KDS source code can use its symbols to avoid hard-coding memory addresses in C, which would have to be changed whenever the source is updated. As specified in Section 4.3.3, the KDS verifies $SIG_{SP}(App)$ in software, which requires the application and its signature to be read from memory. Additionally, the key derivation includes the memory contents of the application and the KDS.

Once its application is running on the device, the SP can connect to the KDS running there, establish a secure session, and request the derivation of $K_{D,App}$. This client was also developed in Go, implementing the cryptographic primitives discussed in Section 4.4.1. All messages exchanged between the client and server are JSON-encoded. The KDS uses the device certificate C_D as its identity, so that the SP can verify whether it is connecting to a genuine device with the IP's public key PK_{IP}. Similarly, the SP's identity is its own certificate C_{SP}, which contains PK_{SP}. This key is later used by the KDS to verify the application's signature. Any protocol errors cause the SP to close the connection to the KDS.

After going through the key exchange, the SP and KDS share a symmetric session key K_s which is used to encrypt and authenticate messages sent between them. We reused already implemented primitives to perform Authenticated Encryption (AE), following the Encrypt-then-MAC (EtM) approach [119]. Messages are first encrypted with AES-128 in counter mode [120], after which the ciphertext is processed by LightMAC (Section 4.4.1) to calculate the authentication tag. Note that unique keys should be derived for both operations, which are obtained by

applying a software-based SHA-256 KDF to K_s, which results in an encryption key $K_{s,e}$ and MAC key $K_{s,m}$ (Section 4.4.1).

4.4.4 Key Distribution Service

On the device side, a software service implements all mechanisms involving asymmetric cryptography and the actual key derivation is calculated in hardware (Section 4.4.5). Most of the service's technical details are related to the key exchange and secure session, which were discussed in the previous section. However, they were implemented in C here, using open-source implementations of AES [121], Curve25519 [122], Ed25519 [123], and SHA-256 [121]. The randomness required to generate the session key pair (y, g^y) is sourced from the TRNG instantiated in the FPGA (Section 4.4.1). Note that Xilinx supports TCP/IP communication on the Zynq through the lwIP library [124] with drivers for its Ethernet subsystem.

The KDS first verifies $SIG_{SP}(App)$ (Section 4.3.3), reading the application's code and signature from memory. Next, it configures the KDM with the start address and size of both the application and the KDS, triggering the derivation of $K_{D,App}$. Once the calculation has finished, the derived key is read from the hardware and sent to the SP in an encrypted message. If an error occurs at any point during this process, the device again closes the connection. Our prototype also implements the proposed attestation mechanism, showing that it can be realised without any additional hardware or software.

4.4.5 Key Derivation Mechanism

At the heart of our solution is the hardware responsible for the actual key derivation (Figure 4.8). The KDM itself implements two main features (Section 4.3.4). First, it calculates the key derivation by processing K_D and the memory contents of the application as well as the KDS. Second, it monitors the processor's execution to ensure that either the requesting application or the KDS is currently running (Section 4.4.5).

Key Derivation Function

$K_{D,App}$ is derived by hashing the device key, the application, its signature, and the KDS (Equation 4.2). The KDF in our prototype is based on SPONGENT [125], a lightweight hash function which can be efficiently implemented in hardware (Figure 4.9). Furthermore, its design parameters can be tuned depending on

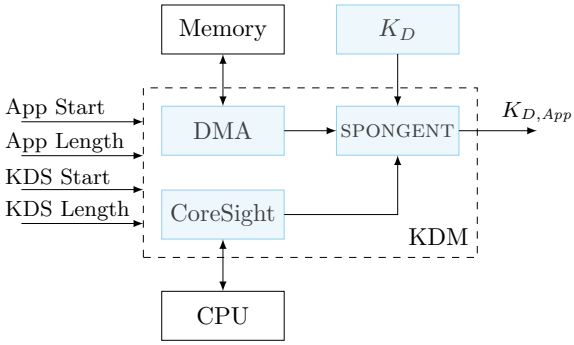

Figure 4.8: The Key Distribution Service (KDS) writes the start address and length of the application as well as itself to configuration registers. In addition to K_D, the Key Derivation Function (KDF) based on SPONGENT processes the memory contents received via DMA. The configuration values are also used to setup CoreSight for execution monitoring.

the design's optimization constraints (e.g., area or throughput). SPONGENT is a hash function based on a sponge construction. This is an iterated design which *absorbs* the variable-length message at a rate of r bits. These message blocks are processed by XORing them with the first r bits of the b-bit internal state and then applying the permutation π_b to the state. Its size b is the sum of the rate r and the sponge's capacity c, which is determined by the rate r and the size n of the hash. After the absorption phase, the hash is obtained by *squeezing* the sponge, repeatedly releasing the first r bits of the state and permuting it until n bits have been output.

Since our prototype works with 128-bit keys, the SPONGENT-128/256/128 variant was selected, which has a 128-bit output. Furthermore, it was also chosen for its 128-bit rate, which has much higher throughput than the 128/128/8 variant, at the cost of larger area due to the increased state size. Its larger capacity also results in better security.

The derivation is started when the source address and length of the application and the KDS are written to the KDM's memory-mapped configuration registers. The signature $SIG_{SP}(App)$ resides next to the application in memory, so it is not necessary to specify its address, adding its length to the application's instead. Before calculating the KDF, the KDM first starts the EM using the configured source addresses and lengths (Section 4.4.5). Once it is running, the device key K_D, which is wired directly to the hash function, is the first block to be absorbed by SPONGENT. Next, the hardware writes the application's source address and length to the DMA configuration registers.

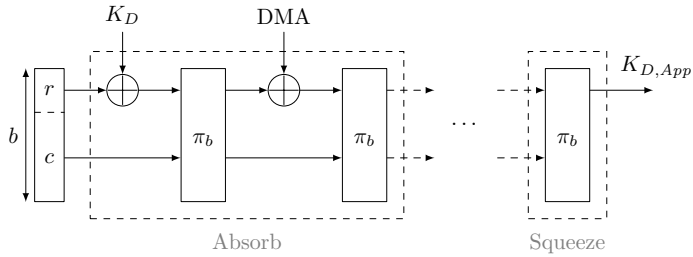

Figure 4.9: The b-bit state of sponge-based hash functions consists of the rate r and capacity c, respectively determining the input size and security, and is processed by the permutation π_b. When the key derivation starts, K_D is initially absorbed by the sponge, followed by the Direct Memory Access (DMA) stream, squeezing out $K_{D,App}$ once the last block has been received.

The KDM connects directly to an instance of Xilinx's AXI DMA LogiCORE IP block. The processor's DDR is memory-mapped to the DMA through an AXI interconnect with the AXI-Stream output of the DMA wired back to the KDM. Since the DMA reads 32 bits at a time, four words are buffered before SPONGENT is called. Once the application and its signature have been processed, the sequence is repeated with the source address and length of the KDS. Finally, $K_{D,App}$ is squeezed from the sponge and stored in a memory-mapped register, from where it can be read unless the EM signals that the KDM should be reset.

Execution Monitor

The final part of our implementation is an IP core-based monitor to observe the execution status of the processor and prevent the key from being accessed by unauthorised code. This is needed, as an attacker could otherwise request the derivation of a key for arbitrary memory regions (Section 4.3.3). Although execution monitoring is straightforward to implement when the KDM is tightly integrated into the processor architecture, it is more complex from an external IP core which does not have direct access to the CPU's internals.

CoreSight [126] is Arm's debug and trace technology for its processors. Several additional components are added to the processor, enabling low-level debugging of code and even SoC components, giving developers powerful tools going beyond traditional software debuggers. Furthermore, in contrast to the latter, these components do not interfere with program execution due to their hardware-based nature. The Zynq's Cortex-A9 includes the CoreSight debugging components (Figure 4.10). First, the Program Trace Macrocell (PTM) traces the occurrence

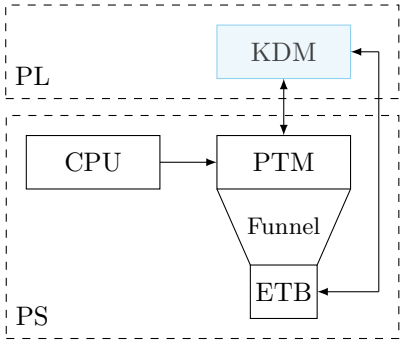

Figure 4.10: CoreSight is an on-chip debug and trace unit for Arm processors. Two of its components are used to implement our Execution Monitor (EM). The Program Trace Macrocell (PTM) can trigger tracing when the processor is executing in certain memory regions. The Key Derivation Mechanism (KDM) monitors the Embedded Trace Buffer (ETB) to verify whether tracing is active.

of certain types of instructions, referred to as waypoint instructions (e.g., branches), which are sufficient to later reconstruct the program's control flow [127]. Since developers are typically interested in tracing a specific application, it is possible to trigger the PTM conditionally. While its event system supports building complex conditions, Eleutheria only uses the Address Range Comparators (ARCs). Second, the Embedded Trace Buffer (ETB) is a small on-chip memory which can store the trace output. In the context of CoreSight, the PTM is also referred to as a *source* and the ETB as a *sink*.

When the key derivation is started by writing to the KDM's configuration registers (Section 4.4.5), it will first set the ETB's *Trace Capture Enable* bit and then initialise the PTM to trace the specified application and KDS ranges. This involves unlocking the PTM's configuration registers and setting the programming bit, after which these ranges are written to the first two ARCs. Next, the `ETMTECR1` register is set to include both range comparators in the trace start/stop logic. Finally, the programming bit is cleared and the PTM configuration is locked again. Once the PTM has been configured, the hardware starts monitoring whether CoreSight is tracing, resetting the key derivation immediately if it is not. Unfortunately, we could not find a status register indicating whether tracing is currently active. Therefore, the EM continuously reads the ETB's RAM Write Pointer (RWP) register [128] and verifies whether it advances to check for tracing activity. Since CoreSight only traces so-called waypoint instructions, the EM allows for up to eight consecutive reads to return the same value before stopping the derivation.

However, these are not the only steps required to setup CoreSight. When the KDS starts, it initialises the ETB and configures the funnel to source data from the PTM and sink it into the ETB. Additionally, right before the key derivation is started and after it has finished, the PTM hardware is respectively enabled and disabled. The KDS also resets the PTM's configuration when enabling it, with all security-critical registers being written by the hardware as discussed earlier. All configuration, both from hardware and software, is done through CoreSight's memory-mapped interface.

4.5 Evaluation

Having detailed the design and implementation of Eleutheria, we will now discuss the performance of our FPGA-based prototype for different application sizes (Section 4.5.1) and list area results (Section 4.5.2). Next, Section 4.5.3 will present an informal security argument for our solution.

4.5.1 Performance

We evaluated Eleutheria using Avnet's ZedBoard as the device running the KDS. This board features the XC7Z020 Zynq-7020 SoC, 512 MB of DRAM, and a Gigabit Ethernet port as one of its I/O interfaces. The IP and SP software components were installed on a machine connected to the same network as the ZedBoard. All required key pairs and certificates were generated and respectively included in the KDS or made available to the SP client.

This setup was used to benchmark the execution time required to handle local and remote key requests for differently sized applications (Table 4.1). We wrote a simple application which only invokes the KDM to retrieve its key $K_{D,App}$. Its ELF section measured 420 bytes and was padded with read-only data for the different tests. The size of the KDS remained constant at 94.56 KB, including the required cryptographic algorithms. For local key requests, the execution times vary from 9.77 ms for 1 KB to 114.00 ms for 1 MB. When requested remotely, a reply is received after 23.72 and 229.42 ms respectively. The initial key request should attest the KDS, which adds on average 10.38 ms to these times. This overhead is fixed because the KDS's size does not change.

The execution time required to service a local key request was measured on the Zynq's Cortex-A9 processor, recording the time required by the application to configure the KDM and to derive $K_{D,App}$. As expected, its execution time increases with the size of the application, as the KDF needs to process more data.

Table 4.1: Execution time needed to service local and remote key requests for differently sized applications, as well as the performance of retrieving an eight-byte message when $K_{D,App}$ is used directly and for the SIGMA key exchange with signature verification. All measurements were obtained for our basic application padded with read-only data.

Size [KB]	**Local [ms]**	**Remote [ms]**	$K_{D,App}$ **[ms]**	**SIGMA [ms]**
1	9.77	23.72	10.73	13.78
256	35.75	74.88	36.72	39.13
512	61.84	126.54	62.85	64.67
1024	114.00	229.42	115.00	115.52

The performance of a remote request was evaluated on the second networked machine, which connects to the KDS running on the Zynq to retrieve the same key $K_{D,App}$. Comparing these results to those obtained for the local requests, we can remark two things. First, there is the overhead incurred by the secure channel, taking 9.95 ms and 11.08 ms respectively on the device and networked machine. However, while this cost is fixed, the difference with the local requests continues to increase as the application grows. This can be explained by the verification of $SIG_{SP}(App)$, which is done in software and depends on the application size.

Since our scheme targets embedded devices where messages are typically very short, we also evaluated the performance of an SP requesting an arbitrary eight-byte value from the application. First, we used $K_{D,App}$ to encrypt the message measuring the case where the SP has already retrieved the application key. Second, when the established SIGMA session key is used to encrypt the response directly, the execution time averaged 11.68 ms. Comparing this average to the $K_{D,App}$ results in Table 4.1 shows that our design is slower than the SIGMA key exchange for all but the smallest application in this benchmark. However, as Eleutheria's key derivation implicitly authenticates the application binary, we should therefore also measure the performance when $SIG_{SP}(App)$ is verified after completing the key exchange but before encrypting the value. In this case, our design is on average 1.11 times faster (Table 4.1, SIGMA).

Considering that key requests will only be issued infrequently, even our prototype implementation performs realistically. In particular, remote requests will only occur after the application has been deployed, because the SP can save the retrieved key. We have also shown that even the performance of the FPGA-based prototype of our approach is comparable to traditional mechanisms giving similar security guarantees. With respect to the latter, note that the current FPGA implementation of the KDM only runs at 125 MHz, while the Cortex-A9

Table 4.2: Area utilization of Eleutheria's hardware components on a Xilinx Zynq-7020 FPGA. The requirements for our EM and KDF are also listed.

	LUTs	Registers	Slices
KDM	977	999	307
EM (CoreSight)	148	120	54
KDF (SPONGENT)	341	400	112
TRNG	26	82	32

is clocked at 667.67 MHz. In an ASIC implementation, Eleutheria's hardware would also run at higher frequencies.

4.5.2 Area

Table 4.2 lists the required area for both the KDM and TRNG. Our implementation of the former utilises 977 LUTs and 999 registers. As can be seen, the CoreSight-based EM does not require a lot of custom hardware in addition to the processor components. Furthermore, SPONGENT is responsible for a large part of the register usage. However, it is still very efficient compared to other hash functions [125]. Note that the current KDF was optimised for latency, as the 128/256/128 variant has a high rate but larger state. Finally, the TRNG only requires 26 LUTs and 82 registers, demonstrating that high-quality randomness sources can be added to embedded systems with very little cost. Compared to X25519 implementations on the same platform [129, 130], Eleutheria's components occupy respectively 70.17% and 60.39% less slices. The latter publication also lists results for an Ed25519 module, which requires 11,148 LUTs and 2,656 registers.

4.5.3 Security

The goal of our design is to improve the security and management of key distribution for embedded architectures relying on symmetric device keys. The former results from the fact that K_D is no longer extracted from the hardware, ensuring that it cannot leak if the IP is ever breached. Conversely, he is only responsible for authorising SPs to issue key requests and for authenticating the devices in his infrastructure. Therefore, SPs no longer need to trust the IP with respect to the confidentiality of any data encrypted under an application key $K_{D,App}$, which is derived from K_D. Managing the key distribution is also simplified, as SPs only need to contact the IP once to request a certificate.

When he has received the certificate, the SP can connect independently to the KDS running on the device to retrieve application keys. Furthermore, the KDS only needs to be provisioned with PK_{IP}, allowing SPs to be added and removed after deployment without any reconfiguration.

The security of Eleutheria can be analysed at the local level on the one hand and at the network level on the other. Recall that the KDF implicitly authenticates both the KDS and the application itself, meaning that an attacker cannot tamper with either, as doing so would result in a different $K_{D,App}$ being derived. Of course, this only holds if the SP initially received the key corresponding to an unmodified binary, which is guaranteed by the KDS's verification of the application signature during the remote key request. This in turn hinges on the SP remotely attesting whether the original KDS is still running on the device, right before initially requesting $K_{D,App}$ (Section 4.3.3).

The KDM combines the KDF and EM in a single hardware component, preventing an attacker from manipulating the connection between them. First, including the application and KDS binaries in the KDF ensures that the full memory ranges of both always have to be given. If an adversary were to invoke the KDM with a shorter or longer range, the function would yield a worthless key. Second, the EM restricts KDM access to the specified ranges, ensuring that only the application or the KDS can retrieve $K_{D,App}$. If they were not tightly integrated, an attacker could pass different arguments to both. Finally, $K_{D,App}$ is copied into memory, e.g., to encrypt some data with it or to send it to the SP over the secure channel. Therefore, memory access control is required to prevent an adversary from accessing this copy. Alternatively, Eleutheria can be combined with a trusted computing architecture offering an isolation mechanism (e.g., TrustZone [44] or Sancus [51]).

Given that it is crucial to our design's security, we should also consider the security of our CoreSight-based EM. Since it is only critical that tracing is active in the specified memory regions, which is enforced in hardware, most of the initial configuration can be done in software, allowing us to keep the hardware simple without compromising security. First of all, if an attacker were to tamper with the CoreSight initialisation code, the key would be derived incorrectly, as this code is part of the KDS and included in the KDF calculation. Additionally, tampering with it would likely break tracing, not giving the adversary any advantage as the key would not be derived. He could also try and interrupt the application or KDS, e.g., to program one of the free ARCs with his own memory range, but this would also halt tracing. Furthermore, an attacker could attempt to write the configuration registers through a debug interface. However, this would still require these registers to be unlocked and the programming bit to be set, which again causes tracing to stop [131]. Finally, an attacker could invoke the KDM without initialising the PTM tracing and instead advance the

RWP register manually, simulating trace activity. However, any writes to it are ignored once the *Trace Capture Enable* bit has been set [128], which is done by Eleutheria's hardware. Consequently, the KDM monitors the state of this bit and aborts if it is cleared during the key derivation.

At the network level, the key $K_{D,App}$ is sent over a secure channel, protecting it from an attacker eavesdropping on the traffic between the device and SP. However, when SK_{SP} is compromised, the SP can be impersonated when requesting $K_{D,App}$. In this case, the SP should therefore have the IP revoke the certificate, e.g., by including it in a Certificate Revocation List (CRL). Rather than having the device download the CRL and perform the lookup locally, it would be more resource-efficient if it were to send the SP's certificate to the IP for verification when it is received as part of the key exchange. Since the KDS already includes PK_{IP}, this communication with the IP can also be secured using the same protocol and algorithms. Note that this involves setting up a second encrypted channel during the session establishment, incurring the associated performance overheads.

SK_D is similarly vulnerable to extraction, and could be used to perform MitM attacks. However, $K_{D,App}$ is only transferred when requested by the SP, and as long as SK_D was not obtained before such a request, the application key would not leak. In case of such an attack, the compromised device should be replaced and the certificate corresponding to SK_D should again be listed on the CRL. Since performance is of a lesser concern at the SP than on the device, both having the SP download the CRL as well as using the approach described above are feasible. Similar to the device, the SP can obtain a copy of PK_{IP}, which can be used to establish a secure channel to the IP. In order to improve the performance of the overall key exchange, the SP could continue the protocol while the verification is pending, i.e., send the third message from Figure 4.7, closing the connection if C_D is encountered on the CRL.

Related to this is the event where the IP would either retain the private key SK_D or create a new certificate C_D. Both alternatives allow him to mount a MitM attack in order to learn $K_{D,App}$. SPs should therefore trust that the IP will act as specified, and this also explains why he was modelled as honest but curious. In this attacker model, the IP never learns $K_{D,App}$ and therefore cannot decrypt any confidential data sent between the application running on the device and its SP. If the adversary were to compromise SK_{IP}, already retrieved application keys would again remain secure, but the entire system should be reprovisioned immediately as this allows him to execute the same MitM attacks.

4.6 Related Work

Many software-based solutions exist where a central entity handles key distribution or manages authentication credentials [132–134]. For instance, the Taos OS [132] implements access control through an agent servicing authentication requests. Among its components is a secure channel manager which establishes secure connections to other nodes by participating in a key exchange protocol. This secure channel manager caches these keys, periodically flushing out old keys.

In the context of distributed wireless sensor networks, lightweight key management schemes have been studied extensively [135]. This survey clearly shows that a wide variety of schemes exists with very different approaches. For instance, some approaches rely on a network-wide master key in order to establish a pairwise link key between two nodes. A second example are probabilistic approaches, where each node is randomly assigned a set of keys from a large pool. In contrast to these schemes, Eleutheria does not perform key establishment between two devices, but enables a remote party to exchange a key with an application running on an embedded device, authenticating this application at the same time.

Job Noorman describes a possible public-key protocol to be used with Sancus in his dissertation [52]. Together with the module, the SP first sends its public key to the device, which then generates a key pair (PK_{SM}, SK_{SM}) and sends the public key to the SP. Data sent to the device can be encrypted using this key PK_{SM}, while its responses will be protected with PK_{SP}. In contrast to Eleutheria, his approach replaces the symmetric algorithms. Furthermore, the IP cannot control which SPs are given access to the device, as their keys are not certified. Soteria [17] adds support for code confidentiality to Sancus. A special loader module was developed which decrypts the protected binary. Similar to Eleutheria, the code of this module is implicitly authenticated, because the key used to encrypt applications is derived from K_{N,SP_L,SM_L}, which includes the module's code (Equation 4.1).

4.7 Conclusion

We presented the design and implementation of Eleutheria, a lightweight key distribution service for devices featuring fixed symmetric keys. Rather than extracting this secret and sharing it with external parties, our solution runs a network service which can be queried for application-specific keys. In order to issue such key requests, a certificate signed by the device owner is required,

allowing him to grant or revoke authorization to do so. This service invokes a custom hardware component performing key derivation, directly accessing the device key. We built an IP core implementation for the Xilinx Zynq-7000 series SoC, which includes a CoreSight-based execution monitor to restrict hardware accesses. The evaluation of our prototype shows that Eleutheria can be implemented in real-world systems with low performance overhead and has similar performance as traditional designs when messages are short.

5 Hardware-Based Memory Protection Mechanisms

Encryption algorithms and key distribution are essential components to build hardware-based trusted computing architectures, and protect critical information and intellectual property. The latter includes the program's code and static data that are deployed to the device. For instance, a signal processing application running on a sensor node might include proprietary algorithms with specifically tuned parameters.

In this chapter, we present two designs with novel roots of trust to protect the confidentiality of code and data by adding new hardware functionality. First, Atlas is a lightweight architecture that introduces transparent memory encryption, building on the cipher implementations from Chapter 3. Second, we extended Sancus (Section 2.5.6) to support loading encrypted binaries.

Content Sources

P. Maene, J. Götzfried, T. Müller, R. de Clercq, F. Freiling and I. Verbauwhede, "Atlas: Application Confidentiality in Compromised Embedded Systems", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2018
Contribution: Main author with Johannes Götzfried, jointly responsible for design, contributed hardware implementation.

J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices", *ACM Transactions on Privacy and Security*, vol. 20, no. 3, 2017
Contribution: Co-author, introduced confidential loading.

5.1 Introduction

Embedded systems are a core component of many products and they are increasingly networked, driven by the development of the IoT. However, this exposes them to a much larger attack surface, explaining the need for lightweight security mechanisms to protect them. For instance, modern cars rely on microcontrollers, interconnected by a CAN, for a variety of functions from controlling the brakes and engine to on-board entertainment. Driven by the increasing complexity of these devices, and in an effort to simplify architecture design and save cost, manufacturers are integrating functionality onto a smaller number of those microcontrollers [137]. This means that sensitive applications now run alongside non-critical ones, increasing the need for security mechanisms to protect confidentiality and integrity. Among others, the engine control algorithms are important intellectual property, and its parameters ensure that the car runs as designed.

However, OSs have been shown to be vulnerable in the past, leading to code and data compromise in some cases. For instance, Dirty COW [138] is a privilege escalation vulnerability based on a bug in the way Linux handled copy-on-write memory, allowing an attacker to gain write access to otherwise read-only memory. At a lower level, Google's Project Zero discovered a vulnerability in the Wi-Fi stack of Broadcom chips [139], enabling a remote adversary to execute arbitrary code on its Arm Cortex R4 running the firmware. Furthermore, this exploit eventually led to code execution in the kernel running on the host device's main processor [140]. These Wi-Fi chips run a very basic OS (HNDRTE), and while the attackers did not compromise it directly, it also does not feature many common security features, allowing memory allocation bugs to be exploited. Therefore, lightweight protection mechanisms are necessary to ensure the confidentiality of those algorithms, even when an attacker compromises the system's OS and can tamper with any software running on the device.

We present two architectures targeting embedded systems that were extended with hardware-level confidentiality protection. First, Section 5.2 details the design and implementation of Atlas, where a transparent encryption unit was added to the memory hierarchy. This unit automatically encrypts and decrypts code as well as data, including the identity of the currently running application in its encryption algorithm. Second, we present an extension to Sancus that enables loading encrypted applications. Because of the multi-functional cryptographic building blocks already present, it was possible to introduce this feature with very little additional area overhead.

5.2 Atlas: Transparent Memory Encryption

This section focuses on protecting the confidentiality of code and data against system-level attackers through transparent memory encryption. Our solution is designed to be complementary to traditional MPUs, which are configured by the OS in order to isolate the memory regions of different applications. However, when the OS has been exploited, security, and especially confidentiality of code and data, can no longer be guaranteed. We ensure confidentiality even in the event of a system compromise, which necessarily requires hardware-based solutions. Once applications start using these hardware-assisted protection mechanisms, there also needs to be a way for them to communicate reliably and securely. In addition, compared to existing trusted computing mechanisms for these lightweight processors, e.g., based on boundary registers [51], our solution has lower area overhead, which is fixed for any number of applications.

Atlas is a hardware-based security mechanism protecting application confidentiality against system-level attackers, with a fixed overhead that is independent of the number of applications running on the system. Furthermore, Atlas enables the use of shared memory as a lightweight and easy-to-use secure communication channel, without the need for a dynamic key exchange. We designed and implemented Atlas by extending the open source LEON3 processor, including a host toolchain to compile C programs for our architecture. Our FPGA-based evaluation shows that Atlas has 0.03% cycle overhead compared to an unmodified binary for a real-world signing application, at the cost of a four times slower maximal clock and 46.60% area increase.

5.2.1 Architecture

This section first presents our attacker model, followed by a discussion of Atlas' system model. We then detail the design of our architecture, giving an overview of the changes that are made to the processor's hardware. Finally, we describe the software changes which are required to enable our application-based, transparent memory encryption.

Attacker Model

In our model, we assume the attacker wants to extract confidential intellectual property (e.g., proprietary algorithms) from the application's code. Furthermore, he is also looking to obtain confidential data processed by it, which was either statically compiled or dynamically calculated at runtime. The assumptions

outlined in Section 1.2.2 are again largely applicable here, except that our solution does not take network-based attacks into account.

The attacker has system-level privileges, i.e., he can exploit any piece of software running on the device, including the OS. As long as the OS has not been compromised, an MPU ensures that applications only access their own memory. When an attacker has obtained system-level privileges, though, he can read from and write to any memory location. DoS attacks are considered to be out of scope. Following the Dolev-Yao model [8], the cryptographic primitives used in our scheme cannot be broken, but protocol-level attacks are allowed.

In contrast to the general attacker model (Section 1.2.2), our design allows physical probing of main memory. However, we assume the attacker does not have access to the CPU's internal registers or caches, excluding invasive attacks where the chip is decapsulated. This is a reasonable assumption, since such attacks require a high level of technical skill, expensive equipment, and take a long time to plan and execute. For example, Tarnovsky's attack on the Infineon SLE 66 microcontroller took six months from planning to execution [141].

System Model

Encrypting memory transparently under a single key is not sufficient to protect against a system-level attacker, as such a design could not track ownership and would return any requested data in plaintext. In order to uniquely identify applications, we define two requirements for Atlas's system model. First, all calls to any confidential application have to pass through its entry point, and applications therefore need to know each other's location. Second, an application should not be able to relocate itself to the entry point of another protected application, as this would give it access to that application's confidential code and data.

The entry point corresponds to the first instruction being executed when an application is called. Atlas satisfies the first constraint by creating a static layout of all applications running on a single device. Since decryption will fail when an attacker moves his application and because it is hard for him to generate a correctly encrypted binary himself, the code encryption mitigates the second issue. Note that applications are expected to yield control when finished, as preemption is currently not supported.

In addition to the device key K_D, the current implementation of Atlas also uses a tweak key F (Section 5.2.2). Both keys are unique for each device, and generated by the system integrator. They are hardwired in the silicon, e.g., by blowing fuses of the manufactured device.

The secure shared memory feature relies on pre-shared secrets. Since a confidential application's static data is encrypted, the communication keys can be stored securely in memory and decrypted when necessary. Generating these keys, defining the regions where the applications can read and write securely shared data, and updating the binary with these parameters are also done by the integrator.

Architecture Design

Atlas' encryption unit protects the confidentiality of applications sharing the same address space. Once memory access control mechanisms relying on software support have been compromised, applications can read from or write to any given address. However, the entry point is used as a unique IV, binding dynamically encrypted data to its application. While a system-level attacker has the ability to read any location, he will be unable to recover the correct plaintext when trying to access protected memory.

When the OS has been compromised, the MPU can no longer be trusted to protect against an attacker modifying memory. As shown earlier, code encryption prevents an attacker from relocating his code to another application's entry point. Although an adversary can now write to any memory location, data encryption cannot be configured independently and thus, code needs to be encrypted as well. This increases the attack complexity, as any instruction manipulating memory needs to be encrypted. Since the attacker does not know the encryption key, it is hard for him to obtain the instruction's ciphertext. Consequently, Atlas protects the confidentiality of code and data against all software attacks, including relocation attacks.

Encryption Unit Properties The encryption unit is considered to have the following properties: first, in order to ensure the confidentiality of different applications, it is able to identify the application to which the current memory bus request belongs. Second, as one of the design goals is to build a scalable architecture, it has to be stateless. Finally, to support secure shared memory, it should be possible to dynamically reconfigure the symmetric key used for data encryption to one that is shared among the communicating applications. We discuss the implementation of these properties in Section 5.2.2.

Hardware Architecture As shown in Figure 5.1, the encryption unit is inserted between the cache and main memory. Once it is turned on, confidential instructions are automatically decrypted when read, and data is decrypted

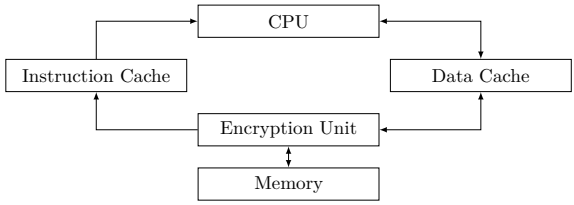

Figure 5.1: The memory hierarchy was modified to include an encryption unit. Encryption and decryption take place right before code and data enter or leave the cache, manipulating the values read from and written to memory before they are communicated over the bus.

and encrypted transparently when entering or leaving the cache. Remember that it is assumed to be impossible for attackers to read the processor's caches or internal registers (Section 5.2.1). To prevent leakage, our hardware and toolchain respectively take care of flushing both caches, and clearing all registers when the encryption unit mode is changed, e.g., when turning encryption on.

The encryption unit is controlled through custom instructions that were added to the ISA. They are executed by the application itself and can be used to turn encryption on or off, e.g., when it does not need confidentiality or in case it wants to access unprotected memory. Additional instructions are available to configure and use secure shared memory.

Software Architecture In order to decrypt encrypted code and dynamically protect data, the currently executing application has to be identifiable. An entry point is therefore created for each application, which is the very first instruction that has to be called when execution of an application is started, and takes care of setting the application's identity and switching the encryption context. Since all local and global functions are encrypted, as well as its static data, they will not be decrypted correctly unless the application was called through its entry point. During secure execution, an application can turn off data encryption, e.g., to write out a final result, but code encryption remains switched on until the application exits. Furthermore, applications are able to call unprotected code, but then any affected data will be processed in clear. Protected applications therefore cannot rely on shared libraries to handle sensitive data, but have to include the required functionality in their own binary, i.e., link against those libraries statically.

Applications are not tied to a specific region, and their code and data can be spread over the entire address space. In particular, the stack is shared between

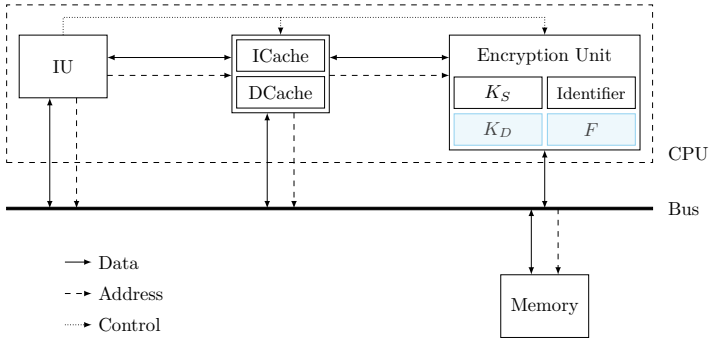

Figure 5.2: The encryption unit was added to the LEON3's cache subsystem. When encryption is turned off, the original instruction and data signals are sent to the bus; otherwise, they are routed through the encryption unit. Note that only the control signals related to the encryption unit are shown.

applications, and the registers of each application are saved to and restored from this single stack. Due to encryption context switching, stack data, including saved registers, is encrypted with a different IV for each application.

5.2.2 Implementation

We implemented Atlas by modifying the LEON3 processor from Gaisler, a 32-bit SPARCv8 architecture with a seven-stage pipeline. Furthermore, a software toolchain was developed to provide the required functionality to compile applications for our platform.

Hardware

The hardware implementation of Atlas consists of two main parts: first, a newly designed encryption unit offering the properties described in Section 5.2.1, and second, custom instructions were added to the Integer Unit (IU) to configure and control memory encryption. Figure 5.2 shows our modifications to the LEON3, which features separate set-associative instruction and data caches, with the latter applying a write-through policy [142]. Since we did not consider multi-core designs, the LEON3's L2 cache was not enabled.

Encryption Unit So far, the encryption unit was described as a building block which satisfies three properties: it can identify the currently running application,

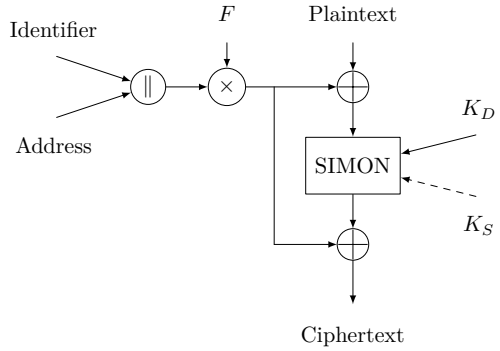

Figure 5.3: The encryption unit uses SIMON 32/64 in the LRW tweakable mode of operation. The tweak is a multiplication in the finite field GF(2^{64}) of a tweak key F and IV, which is the concatenation of the application identifier and the current memory address. The encryption key can be switched from the fixed device key K_D to a configurable pre-shared key K_S when secure shared memory is used.

encrypts data without storing state, and has a reconfigurable key (Section 5.2.1). Our implementation stores the identifier of the active application in a dedicated register, which can only be updated through a custom instruction. The device key K_D is always used, except when the encryption unit is configured to enable secure shared memory. In that case, the unit switches to the secure shared memory key K_S, which is stored in a dynamically configurable dedicated register. Note that this key is only used to encrypt and decrypt shared data, with K_D still being used to decrypt protected code.

Figure 5.3 shows a diagram of the encryption unit. The LRW tweakable mode of operation [143] is used to realise stateless encryption of a single 32-bit word. The tweak ensures that every message is unique. In this mode, the ciphertext C is calculated as follows:

$$C = E_K(P \oplus X) \oplus X$$

$$X = F \otimes I$$

where P is the plaintext, X the tweak, E_K encryption with key K (Definition 3.1), F the tweak key, and I the IV. Atlas uses the concatenation of the application identifier and the memory address that is being read from or written to as the IV. Both values are 32-bit, so therefore the tweak key F also has to be 64 bits long and the finite field used for the multiplication is GF(2^{64}). X is then truncated to 32 bits before XORing it with the plaintext and output of the cipher respectively.

Since any block cipher can be used in this mode of operation, the choice of algorithm is determined by the word size of the CPU architecture. The LEON3 is a 32-bit architecture where values are read from and written to memory at word granularity. In order to reduce the complexity of the memory controller, a 32-bit block cipher was selected. Additionally, a low-latency single-cycle implementation was used to ensure that there is no additional cycle overhead for memory accesses, and to keep the critical path as short as possible. SIMON 32/64 was shown to be the fastest and smallest algorithm with 32-bit blocks (Section 3.3). Recall that none of the alternatives with longer keys have low latencies (e.g., KATAN supports 80-bit keys, but has a two times longer critical path). Although 64-bit keys offer short term protection against small organizations [144], we recommend using PRINCE (Section 3.3.4) in case of a 64-bit architecture. It has 64-bit blocks and 128-bit keys, and is the fastest single-cycle cipher currently available, with very competitive area.

LRW is a tweakable mode of operation like XTS, which is now widely used to encrypt block devices like hard disks [145, 146]. The reason for choosing LRW over XTS was that the latter passes through the block cipher twice for each block, which would result in a longer critical path. LRW has a known weakness when the plaintext contains the tweak key F. Since the tweak key register is not accessible directly from software, this is not an issue in our design. In contrast to other modes of operation (e.g., CTR or CFB), LRW requires an implementation of the cipher's decryption function.

However, the memory is never read and written at the same time, enabling the reuse of encryption components for decryption as an optimization. SIMON is a Feistel cipher, where decryption is almost identical to encryption, except that the inputs have to be swapped and the key schedule has to be reversed. Furthermore, SIMON's key expansion is linear, thus it can also be performed in parallel to the round functions for decryption. Therefore, Atlas also includes a decryption key consisting of the last four subkeys in order to initialise the key expansion when decrypting. If SIMON were replaced with an encryption algorithm where the key cannot be expanded in parallel to the round functions when decrypting, either all subkeys should be fixed in hardware or they would have to be computed before the round functions are applied. The former would negatively impact the implementation's area, while the latter would significantly increase the critical path. In general, we suggest the use of a block cipher where encryption and decryption share functionality, and where low-latency single-cycle implementations can be built. Note that this does incur the area and latency cost of additional multiplexers where signals are driven differently when the unit is respectively encrypting or decrypting.

Custom Instructions Atlas extends the LEON3's IU with the following eight new instructions to give software developers access to the new security features.

ENCENTER stores the current value of the program counter in the identifier register and turns on encryption. It is the first instruction that has to be called at the entry point of any confidential application.

ENCEXIT clears all registers of the encryption unit and turns off encryption. It has to be invoked when there is an exit from a confidential application.

ENCPAUSE turns *data* encryption off without clearing any registers. An application which wants to write to unprotected memory needs to call this instruction first.

ENCRESUME turns *data* encryption on with the currently saved settings, resuming confidential execution of the running confidential application.

ENCSHMON turns on shared memory encryption. This instruction switches the data encryption key to K_S and zeroes out the application identifier.

ENCSHMOFF turns off shared memory encryption without clearing K_S and resumes isolated execution by switching back to K_D.

ENCSETKEY `ENCSETEKEY` and `ENCSETDKEY` are respectively used to set the encryption and decryption key for the SIMON cipher used to secure shared memory. The full 64-bit key is passed in two general-purpose registers.

To prevent data leakage, the hardware ensures that the instruction and data cache are always flushed when encryption is enabled or disabled, i.e., when `ENCENTER` and `ENCEXIT` are dispatched. The caches are not flushed during `ENCPAUSE`, `ENCRESUME`, `ENCSHMON`, or `ENCSHMOFF`, as they are executed by protected code which can be assumed to not leak confidential information. Finally, this also means that except for `ENCENTER`, these instructions will always be encrypted in the binary.

The SPARCv8 ISA defines three general formats that are used to encode operations [147]. We identified an unused value of the second format's *op2* field for all instructions defined above except for `ENCSETEKEY` and `ENCSETDKEY`. Since none of these have need of the *imm22* field, we used its three LSBs to represent the different encryption unit operations. Both `ENCSETKEY` variants were encoded using SPARCv8's third format, passing the two registers holding the key in the source register operands *rs1* and *rs2*. In order to disambiguate setting the encryption and decryption key, we could employ the LSB of *rd*, as neither operation has a return value. Finally, the LEON3's decode stage was updated to recognise these instructions and trigger the relevant functionality

in the encryption unit, replacing them with bubbles for the remaining original pipeline stages.

Software

In order to use Atlas' features, the new instructions need to be dispatched at some point. To this end, we developed a toolchain to expose the functionality to programmers as transparently as possible. With our toolchain, normal C programs can be compiled and linked for the modified core, while the programmer only needs to properly divide the functionality into confidential and unprotected code. To this end, we use ELF rewriting with relocatable object files and executables, i.e., no compiler patch is needed. Our tooling can therefore be easily combined with other existing toolchains.

Confidential Applications Code and data of a confidential application are transparently protected by the encryption unit. With our toolchain, the programmer can define which files constitute such a confidential application. The remaining functionality of all other source files is considered to be unprotected. Each application can be written in standard C code, and programmers have the ability to annotate their code with macros. These enable them to call into other confidential applications without leaking any data but the supplied parameters.

Control Flow Rewriting After each confidential application has been compiled, our toolchain parses all relocatable object files and identifies calls from unprotected code to a confidential application or vice versa. These calls are then rewritten to go through entry and exit routines, which switch the encryption context. Identifiers for the target function as well as the originating function and application are passed in registers, preserving the original control flow.

The context of a confidential application, i.e., all callee-saved registers, is saved and cleared before the context switch, and restored afterwards. Caller-saved registers which are not used for passing arguments are cleared to ensure that no data leaks.

Encryption Since our toolchain supports standard C code, we also provide built-in support for encrypting confidential applications. The code and static data of each application are both placed in separate text and data sections, except for the entry and exit stubs. After the linking step, our toolchain parses the executable file to locate these sections, and encrypts them. Furthermore, it

looks for all stubs belonging to the application in the main text section, and encrypts those as well.

One implementation aspect we would like to discuss explicitly is the encryption of the GCC integer library routines. On platforms where hardware support for certain mathematical functionality is not available, the compiler automatically inserts code implementing the missing operators. This only happens during the final link stage, and these routines are therefore only inserted into the binary once. Since this is done transparently to the programmer, they could be called from confidential code as well. One solution would be to keep these functions in unprotected code, and again perform the control flow rewriting outlined above. However, this would incur the overhead of switching the encryption context on every call and also mean that their parameters are passed in the clear. Our toolchain therefore ensures that copies of these functions are added to each protected application by partially linking its sources first. The compiled object is then encrypted like any other code in the protected application.

Atlas Library While most of our software implementation is part of the toolchain, we also provide a library for programmers. In addition to the annotation macros, we include library functions for copying data between confidential applications and unprotected code, as well as template functions for opening and accessing secure shared memory sections between different applications. Helper functions are included to set a shared precomputed key, and to copy from and to these sections. Furthermore, we provide a generator to create these routines for an arbitrary number of applications.

5.2.3 Evaluation

In this section, Atlas is first evaluated regarding performance and area on FPGA. We obtained results for the Digilent Atlys and Xilinx ML605 development boards, which have Xilinx Spartan 6 and Virtex 6 FPGAs respectively. The LEON3's default cache configuration for both these boards includes a 16 KB 2-way instruction cache with 32-byte lines and a smaller 8 KB 2-way data cache using 16-byte lines, where both rely on the Least Recently Used (LRU) replacement algorithm. Xilinx ISE 14.7 was used for synthesis, place, and route. Next, we also informally argue the security of our design.

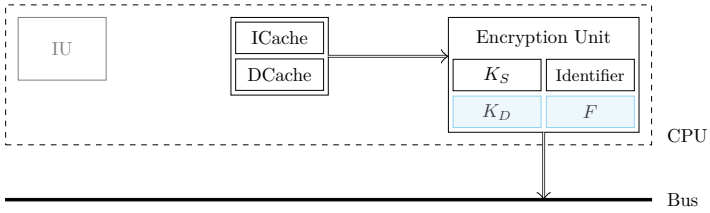

Figure 5.4: Synthesis of our prototype for two Xilinx FPGA boards confirms that the critical path of the modified core runs through the encryption unit. In detail, it starts from the cache's memory instance, passes through the block cipher implementation, and terminates at the bus.

Performance

Critical Path Since the memory hierarchy is part of a processor's critical path, and given that single-cycle implementations of encryption algorithms result in long combinational circuits (Chapter 3), the maximum clock frequency of Atlas is reduced compared to the original design. Our synthesis results confirm that the critical path runs from the memory instance of the instruction or data cache, respectively on the Atlys and ML605, through the block cipher implementation of the encryption unit, to the buffer registers at the bus interface (Figure 5.4).

On the Atlys, the original design can run at a maximum frequency of 78.57 MHz, whereas Atlas can be clocked at 19.05 MHz. We saw similar results on the ML605, where the original maximum frequency of 109.09 MHz was reduced to 31.58 MHz. However, embedded systems are typically designed for low power and therefore not clocked at the maximum possible frequency [148], so that the actual overhead therefore depends on the application. If the maximum possible frequency of a single-cycle design would not be sufficient, the cipher could be serialised or pipelined to improve performance, trading latency for delay on memory operations.

Microbenchmark Three microbenchmarks have been run on our evaluation platform to measure the performance impact of our toolchain. The first is an application which invokes a confidential one that simply returns. To show the overhead between entering a confidential application and a regular call, we compiled this application with a vanilla GCC toolchain as well as our modified one. The former finishes in 87 cycles, while the latter executes in 227 cycles. The secure context switch and cache flush, which ensure that no confidential data will leak, are responsible for this overhead.

Where the former benchmark measured the direct overhead of the context switch, our second benchmark evaluates the penalty of restarting execution with a flushed cache. To this end, we call 1,000 times into a confidential application that reads 128 bytes of data, unrolling both operations to ensure instruction cache utilization. This evaluation was run on an unmodified LEON3 with the same architectural settings as Atlas, except for two additional hardware breakpoints. Since the original GCC toolchain and processor source are used, the overhead from the first benchmark is not incurred here. We ran the evaluation binary first without any flushes, and then by setting up the debugger to break when entering and leaving the confidential application, clearing the cache before continuing. This configuration allows us to measure just the worst-case indirect overhead of Atlas' hardware-based cache flushes, revealing a 75% slowdown.

Third, we copied 1 KB of data from a confidential application to unprotected memory. This requires encryption to be switched off and on repeatedly, as each data element needs to be loaded into a register while encryption is enabled and written back to memory after it has been disabled. This operation is 4.56 times slower than `memcpy`, due to the encryption context being paused and resumed.

Macrobenchmark To demonstrate the overhead Atlas imposes on real-world programs, we wrote an example signing application, which consists of unprotected code and a confidential application with static encrypted data. A message is passed from unprotected code to the confidential application, where it is signed with an asymmetric private key stored securely in the static data section. The signed message is then passed back to unprotected code, where the signature is verified with the corresponding public key. In addition to the overhead imposed by the confidential application call, the message and signature respectively have to be copied from and to unprotected memory. The TweetNaCl [85] library is used for the signature generation and verification.

We compiled this application with both an unmodified GCC toolchain and our modified one. The combination of its write-through data cache and the write strobe functionality of its bus, allows the LEON3 to transfer only the modified bytes when partial writes are issued, breaking Atlas' encryption which requires the full word. Therefore, `stb` or `sth` cannot be used in our current prototype, and the benchmark was run with data encryption disabled. Addressing this would require modifying the cache architecture to always write all 32 bits, possibly causing a cache line to be read if that data is not present. Section 5.2.5 further discusses this point for architectures with different characteristics. However, since all other modifications to the core remained in place (e.g., cache flushes) and as the cipher implementation is fully unrolled, with the design clocked at the same frequency, the performance results are not affected.

Table 5.1: Area in terms of registers, LUTs and occupied slices of an unmodified LEON3, compared to our core on FPGA.

	Unmodified	Atlas	Overhead
Digilent Atlys			
Slices	2,496	3,659	46.59%
Registers	3,070	3,333	8.57%
LUTs	6,261	9,726	55.34%
Xilinx ML605			
Slices	5,519	7,970	44.41%
Registers	11,021	12,046	9.30%
LUTs	13,070	18,482	41.40%

For both binaries, the execution time was measured with and without copying to and from protected memory. The binary which has all Atlas features enabled imposes an overall overhead of 0.03% compared to the GCC-compiled binary without any secure copies. When secure copies are disabled in the binary compiled with our toolchain, execution takes on average 449 cycles longer than the 1,625,595 cycles of the reference binary. When compiled with GCC and secure copies enabled, the overhead is equal to 0.02%. Recall that both caches are flushed during the execution of `ENCENTER` and `ENCEXIT`, which contribute significantly to the reported overhead. For comparison, when the toolchain-compiled binary with secure copies enabled is executed on an Atlas core where these flushes were removed, the overhead drops to 0.02%.

Area

The area usage of Atlas was measured after Xilinx ISE finished place and route. An unmodified LEON3 synthesised with the same settings occupies 2,496 slices on the Atlys. Atlas occupies 3,659 slices, resulting in an overhead of 46.60% (Table 5.1). To reduce the number of required gates, the same cipher core is reused for encryption and decryption (Section 5.2.2). Although SIMON is the smallest cipher currently available, cryptographic primitives remain expensive in terms of area, especially in case of single-cycle implementations. As mentioned before, a serialised implementation could also further improve the area requirements.

Security

The goal of Atlas is to protect the confidentiality of code and data on embedded systems, even when the device's OS has been compromised. This is realised by adding an encryption unit to the memory hierarchy, which transparently encrypts any data leaving the processor and decrypts incoming transfers. The encryption unit is controlled through a set of dedicated instructions (Section 5.2.2). As discussed, `ENCENTER` is the first instruction of protected applications and the only one which is stored in plain. When called, the current value of the program counter is copied to the dedicated identifier register. Since this instruction has to be executed for that register to be set, an attacker or malicious OS cannot directly control its value. Consequently, this prevents the encryption unit from being used as a decryption oracle. Furthermore, an attacker cannot replace the code following `ENCENTER`, e.g., to read out secrets included in the binary, as this requires knowledge of K_D. Finally, note that cleartext code and data are stored in the processor's caches. Considering that an attacker cannot generate correctly encrypted code, he would first have to turn encryption off if he were to try and access cached code or data. However, recall that all caches are flushed from hardware when `ENCEXIT` is executed (Section 5.2.1).

When the secure shared memory functionality is used, the encryption unit operates differently. In particular, the application identifier is set to zero and K_S is used for encryption, which can be set dynamically. The security of this mode hinges on the fact that each application accessing the secure shared memory region includes K_S as static data, which is encrypted using K_D and the application identifier. Therefore, the attacker is not able to learn this key, as it is secured by the encryption unit.

Lastly, we also protect against some classes of physical attacks, specifically main memory probing. This relies on the fact that the encryption unit is inserted between the memory bus and caches (Figure 5.2). Code and data are therefore only decrypted within the processor's boundaries and there is no point where a probing attacker can tap cleartext from the bus or read confidential data directly from main memory.

5.2.4 Related Work

Many solutions guaranteeing code and data confidentiality have already been proposed. This section first discusses software-based memory encryption approaches, and then presents hardware-based architectures.

Software-Based Memory Encryption

Software-based memory encryption solutions [149] can be used to ensure confidentiality of code and data. This has been done at different levels of the memory hierarchy, from protecting only swap storage [150], to process memory ranges [151, 152], and even the whole RAM [153, 154]. While software-based memory encryption has the advantage of compatibility, it also negatively impacts performance and, more importantly, can only prevent memory probing attacks. Furthermore, it cannot protect applications from a system-level attacker.

CPU-based encryption is somewhat related to our work but only protects a small fraction of sensitive data. Symmetric encryption schemes range from register-based approaches as an OS patch [155, 156] to solutions relying on hypervisors [157] and caches [158]. There even exist mechanisms to protect asymmetric encryption algorithms, as it turned out that asymmetric keys can be recovered from memory as well [159, 160]. In particular, RSA implementations exist that are either register-based [161, 162] or rely on hardware transactional memory [163]. However, all these solutions just keep the encryption key and intermediate data out of memory, but not any other sensitive information, because they only have limited secure storage available. In contrast, our encryption unit is inserted directly in the memory hierarchy between the cache and main memory, ensuring that confidential code or data is protected as soon as it leaves the processor package.

As mentioned before, full-disk encryption [145, 146] uses cryptographic primitives to protect data at rest, as it addresses a related problem. However, these solutions deal with much larger storage sizes than Atlas and also have very different latency requirements. Current solutions typically rely on the XTS mode of operation [164, 165]. XTS is almost identical to LRW, its main differences being that the tweak i is first encrypted and that the second multiplicand is equal to α^j, where j is the IV. In addition, if the last plaintext block is smaller than the block size, it is padded with bits from the previous ciphertext. Finally, when applied to standard sector-level disk encryption, data units typically correspond to logical blocks [164]. Note that disk encryption solutions are length-preserving and therefore do not authenticate the encrypted data, instead relying on the fact that the ciphertext is not malleable [145]. Atlas was similarly designed to transparently encrypt and decrypt memory, but does use application-specific keys to prevent different applications from accessing unauthorised data.

Hardware-Based Memory Encryption

Intel's SGX, which was detailed in Section 2.5.8, provides a general hardware base for strict isolation of applications on x86. Recall that the MEE dynamically encrypts code and data leaving the cache to protect the confidentiality of applications in untrusted memory. In contrast to Atlas, SGX uses multiple configuration structures which are stored in memory, nor is its hardware overhead considered lightweight.

Researchers at IBM also proposed an architecture called SecureBlue++ (Section 2.5.7), protecting the confidentiality and integrity of an application's cache lines when they are evicted to main memory. Although it guarantees integrity in addition to confidentiality, an important difference compared to Atlas is that its confidentiality protection functionality relies on hardware implementations of several cryptographic primitives, thus drastically increasing the memory controller's complexity. As we have seen, binaries are encrypted using a symmetric key, which is itself encrypted asymmetrically and decrypted in hardware when entering the secure mode. While more flexible in terms of key distribution compared to Atlas, this means that an expensive hardware implementation of an asymmetric algorithm is required as well.

For embedded systems, many solutions build on the concept of PMAs. Sancus [51] is a security architecture for lightweight devices, providing isolation and attestation. Its memory access control mechanism consists of a combinational circuit which checks the current memory address against a set of boundary registers. Two pairs of registers are added, storing the start and end addresses of the text and data section respectively. Access to the memory regions specified by the registers is then restricted based on the current value of the processor's program counter. Soteria [17] further extends Sancus, protecting intellectual property at load time through encryption and during runtime with the help of Sancus' dedicated memory access logic in hardware.

All these lightweight solutions have in common that they need to maintain state per application. Furthermore, the overhead of Atlas in terms of LUTs on similar FPGAs is comparable to the fixed LUT overhead of Sancus. Our design requires significantly fewer registers, but has a greater impact on the critical path, at least until ciphers with lower latency become available. Finally, in contrast to Atlas, most solutions relying on PMAs cannot be used with more complex memory hierarchies including caches.

5.2.5 Discussion

In this section, we first consider open issues of Atlas' current prototype implementation, focusing on unsupported SPARC features and function pointers. Possible future improvements are identified next, including the impact of alternative cache designs and ways to reduce the critical path overhead.

Limitations

Atlas does not support SPARC register windows, requiring flat compilation of software. The reason is that overflowing or underflowing register windows triggers an interrupt, which currently cannot be handled by Atlas. Enabling interrupts in the current design would violate our security policy, as they circumvent the encryption context switch.

Furthermore, function pointers cannot be used currently for calls between applications. Since it is impossible to reliably determine the destination address of calculated calls at compile or link time, our toolchain would not be able to rewrite control flow to jump through the application's entry point which initialises the encryption unit. However, this could be addressed by making use of binary translation techniques which do not rely on relocation information, but rather insert code at indirect call sites to dynamically rewrite control flow to the new location [166].

Future Work

Encrypting on word granularity leads to small block sizes of 32 bits, which would change when porting our design to a 64-bit architecture, allowing stronger algorithms to be used, such as PRINCE (Section 3.3.4). Alternatively, two words could be encrypted simultaneously, but this would significantly complicate the encryption unit's design and impact performance, as reads would require both words to be fetched. Similarly, writes might incur a read, because encryption always needs to be performed with both words. Finally, the encryption unit could operate on cache line granularity in systems with write-back caches, as encryption and decryption would respectively take place when lines are flushed and loaded. Since these lines consist of multiple words, it would be straightforward to use ciphers with larger block and key sizes, nor would partial word transactions have to be accounted for.

As mentioned, serialising the cipher would improve the clock frequency overhead and further reduce the area requirements of Atlas. Alternatively, the encryption

unit could be pipelined, resulting in a shorter critical path and higher throughput. Both approaches would impose a cycle cost on each memory access, as the processor would have to wait for the encryption unit. Therefore, finding a good trade-off between both presents an interesting design challenge.

5.3 Sancus 2.0: Confidential Loading of Modules

Sancus (Section 2.5.6) is a lightweight PMA for networked embedded devices featuring isolation and attestation. It was first published in 2013 [51], but continuous contributions by different researchers resulted in a second publication [52]. Originally, Sancus' memory access logic allowed the text section of an SM to be read from unprotected code or another SM. Furthermore, modifying the memory access logic to prevent these accesses would only protect a module's code after it has been loaded. It could therefore still be read from memory beforehand, comprising its confidentiality.

Addressing this limitation, Soteria (Section 2.5.6) extended Sancus to enable loading of confidential code. While it required small hardware modifications, Soteria mainly relied on a special loader module implemented in software. This loader module decrypts the protected module, which is encrypted using AES-CCM. However, given the low frequency of the microcontrollers targeted by Sancus, this process is slow [17]. In order to speed up this operation, we added hardware support for confidential loading of modules, which required little additional resources due to the presence of a flexible cryptographic primitive.

This section first details how the original design of the `protect` instruction was modified to support confidential loading (Section 5.3.1). Next, we discuss its implementation in Section 5.3.2 and finally list evaluation results, comparing the hardware-based implementation to Soteria (Section 5.3.3).

5.3.1 Design

While Section 2.5.6 summarises the design and implementation of Sancus, we now provide additional technical detail to support our description of the confidential loading functionality. One of Sancus' core components is the protected storage area, which consists of dedicated registers holding a module's metadata (Figure 2.4). These registers are initialised when the `protect` instruction is called as follows:

`protect` *layout*, *SP*

Table 5.2: In the original design, the combinational Memory Access Logic (MAL) circuit enforces the following rights on every memory request for all enabled Software Modules (SMs). Note that the entry point and text section are readable from unprotected memory and other SMs.

From/To	Entry	Text	Data	Unprotected
Entry	`r-x`	`r-x`	`rw-`	`rwx`
Text	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected Other SM	`r-x`	`r--`	`---`	`rwx`

Recall that a module's identity includes the start and end address of both its text and data section, as well the contents of the text section and is used to calculate $K_{N,SP,SM}$ from K_{SP}. The latter is derived from the node key K_N and the SP's identity, which is the second argument to `protect`. When this instruction is issued, the processor's hardware will (i) check that the layout does not to overlap with existing modules, and register a new module by storing its layout in the protected storage area; (ii) enable memory access control (Table 5.2); (iii) derive the module key $K_{N,SP,SM}$ using the text section and layout of the actually loaded module and store it in the protected storage area.

Execution of this instruction therefore requires a hardware implementation of the KDF to calculate the module key. Furthermore, it is worth repeating that Sancus' attestation functionality uses a MAC function to calculate its measurements, which should also be available as part of the processor.

Having detailed the original design of `protect`, we now discuss how it was modified to support confidential loading of modules. The code of these modules is stored encrypted in memory up until they are enabled, when it is decrypted and protected by the processor. In order to ensure that an attacker cannot obtain a copy of the plaintext binary, these operations should happen atomically. Therefore, we now provide a second way to use the `protect` instruction:

`protect` *layout*, *SP*, *MAC*

In this form, `protect` behaves exactly the same as above except that, before calculating the module key, the module's text section is decrypted using $K_{N,SP}$. If the integrity check using the given MAC fails, the text section is cleared and the protection disabled. It should also be mentioned that the integrity check is not strictly necessary for confidential loading, since any subsequent remote attestation will also verify the module's integrity. However, it could be used as a simple form of module authentication: by disabling the non-decrypting form

of the `protect` instruction, only entities possessing a valid $K_{N,SP}$ can install modules on the system.

The memory access logic was modified to deny other SMs access to the text section. The design of the secure linking functionality (Section 2.5.6) was also adapted to support confidential code. Previously, when SM_1 wanted to securely link to SM_2, SP_2 needed to send SM_2 to SP_1 so that it could calculate a MAC with K_{N,SP_1,SM_1}. However, this is not possible for a module with confidential code, as SM_2 does not necessarily trust SM_1. This was solved by replacing the MAC with a hash of the module's cleartext code, sent by SP_1 to SP_2. A hash will not leak confidential information and it does reflect any changes to the module's functionality, still ensuring secure linking. Note that if SM_2 also wants to link to SM_1, this method creates a circular dependency between their identities. This in turn can be resolved by not including the other's identity in the text section but having the software provider securely send it after deployment and storing it in the data section.

As explained, the original design requires a hardware implementation of two cryptographic primitives: key derivation and MAC functions. Introducing confidential loading means that we now also need implementations of a cipher as well as a hash function to implement secure linking. Since Sancus is designed for embedded systems, one of the main goals is to make the implementation of these features as compact as possible. All required functionality can be realised simultaneously by an AEAD primitive, significantly reducing the additional area overhead (Section 5.3.3). Such an AEAD function returns a ciphertext and authentication tag, given the following arguments:

$$(C, T) = AEAD(K, P, A)$$

In this equation, the inputs K, P, and A respectively denote the encryption key, plaintext, and associated data. Finally, it is worth pointing out that the remote attestation functionality can now also be extended to include confidential input or output with the attestation request and measurement response respectively.

5.3.2 Implementation

The Sancus prototype is based on the openMSP430, which is an open-source clone of a Texas Instruments microcontroller by the same name. This is a straightforward processor in terms of architectural design, with three basic pipeline stages (fetch, decode, and execute) and no caching. The custom instructions introduced by Sancus were added to the core's ISA and additional hardware was included to support their operation, most notably the cryptographic primitives, memory access logic, and protected storage area.

The original Sancus implementation used the HKDF [110] mechanism as its KDF, which itself relies on the HMAC [48] construction to perform extraction or expansion from its input key material. As its name implies, an HMAC requires a hash function, with Sancus selecting SPONGENT. This is a lightweight hash function that was designed specifically to have an efficient hardware implementation. Furthermore, recall that Sancus' attestation procedure calculates a MAC tag, reusing this HMAC implementation.

However, these cryptographic primitives can also be implemented with an AEAD algorithm, and they were replaced with a single instance of the SPONGEWRAP [54] construction with SPONGENT [53] as the underlying sponge function. Given that keyed sponge functions are shown to be pseudorandom functions [167], we can reuse SPONGEWRAP to calculate MACs, and consequently for key derivation. Since its security relies on the soundness of a sponge function, it can also be used as a hash function by calling $(-, H) = AEAD(-, -, M)$ and using the authentication tag as the digest.

Furthermore, we can implement SPONGEWRAP generically, because its security is proportional to the capacity of the underlying sponge function and SPONGENT's specification lists a range of capacities. In detail, our core can be synthesised with a security parameter between 16 and 256 bits, although values less than 80 bits should be avoided. Given that this parameter influences the core's area, it offers a trade-off between cost and security. As we have seen (Section 2.5.6), all module keys are stored in hardware, making the key size an important design parameter regarding area.

A downside of SPONGEWRAP is that uniqueness of the associated data is required for confidentiality, and no security guarantees can be given when a nonce is reused. More specifically, if two ciphertext messages are captured that are encrypted with the same key and associated data, part of the `XOR` of the corresponding plaintext message may be leaked [54]. Therefore, the user of this primitive should ensure that the associated data is unique for a specific key, i.e., that it includes a nonce. Note that this is only necessary when encrypting data and there is no nonce requirement for creating MACs. In contrast, nonce-misuse resistant authenticated encryption algorithms, such as APE [168], limit information leakage about the message when the nonce is reused, but this comes at an additional implementation cost.

It is an SP's responsibility that the nonce requirement is fulfilled by its modules. In our prototypes, this is achieved by having SP send an initial counter value as nonce in its first message to a newly deployed module. For subsequent messages, modules can simply increment the counter and use that value as the next nonce. Alternatively, if an SP never wants to send messages to a module, the initial counter value can be included in the module's text section.

Because of this associated data uniqueness requirement, our implementation of confidential loading is slightly different from its design (Section 5.3.1). Since modules deployed on node N by SP are always encrypted using $K_{N,SP}$, `protect` takes an extra argument, *nonce*, to be able to fulfil the nonce requirement. This argument is used as the associated data input for the decryption routine.

Since the original HKDF and HMAC implementations had already been replaced by SpongeWrap when we started working on the confidential loading functionality, its integration was straightforward. We modified the decoding logic of `protect` to check for the new *MAC* and *nonce* arguments and trigger decryption of the module's code when they are set. This decryption is performed by initialising SpongeWrap with $K_{N,SP}$ and the provided nonce. Next, the hardware can read the module's encrypted code from memory, because the start and end address of its text section were also provided as part of the *layout* argument. The hardware will then continuously read ciphertext from memory, decrypt it, and write it back to the same location. Once all code has been processed, the authentication tag is also read from memory and compared to the one calculated by SpongeWrap. If this comparison fails, the text section is cleared and protection is disabled.

Note that this means encrypted modules are being decrypted in-place, with the ciphertext being overwritten by plaintext. Consequently, a developer of an encrypted module should never call `unprotect`, which would disable the isolation mechanism and expose the module's code. While this was not done in our design and implementation, having `unprotect` re-encrypt the module before lifting the memory access control would be one possible solution, at the cost of significantly increased execution time.

The secure linking design was also modified to require a hash rather than a MAC with the key of SP_2, so that SP_1 does not have to send the source code of a confidential module to SP_2. Given the versatility of SpongeWrap, this change simply involved calling *AEAD* without the key argument to change the MAC used in the old design to a hash.

Finally, in addition to the hardware modifications, the software toolchain was also updated to encrypt software modules. Given that our confidential loading implementation reused existing cryptographic primitives, this mainly involved extending Sancus' custom linker to encrypt the text section of a module using $K_{N,SP}$. The nonce that was used for the encryption and the resulting MAC are then stored in an additional public section that is added to the final binary.

Table 5.3: Number of cycles needed for software- and hardware-based confidential loading of modules with different sizes. The relative difference between both implementations is shown as well.

Size [B]	Software	Hardware	Difference
256	507,536	13,207	38.43x
512	951,464	25,111	37.89x
768	1,395,384	37,015	37.70x
1024	1,839,304	48,916	37.60x

5.3.3 Evaluation

We will now evaluate the performance and area requirements of confidential loading by synthesising the modified softcore for FPGA. The former considers a microbenchmark for differently sized modules and compares the results of our design with Soteria's software-based approach. Next, we list the area usage of the original openMSP430 and compare the overhead of both Sancus versions. Finally, we detail the security implications of the confidential loading support.

Performance

Since a module's code has to be decrypted before it can be run, enabling confidential loading will impact the performance of `protect`. In order to evaluate this overhead, we measured the number of cycles needed to load modules with different sizes and compare the results to Soteria's software-based approach (Table 5.3). On average, the hardware implementation is 37.91 times faster. This can be explained by the performance advantage of an encryption algorithm implemented in hardware, which has direct access to the device's memory, at the cost of flexibility and increased area and power requirements.

Area

We synthesised all designs again for a Digilent Atlys board with a Spartan 6 FPGA using Xilinx ISE, with the core running at 20 MHz. All implementations used 128-bit keys, which determines the area requirements of the cryptographic implementations and impacts the overhead of the protected storage area.

An unmodified openMSP430 microcontroller occupies 2,322 LUTs and 998 registers. The original Sancus implementation has a fixed overhead of 1,138 LUTs and 586 registers, with each SM incurring an additional 307 LUTs and

213 registers. The updated version of Sancus, where SpongeWrap is used for all cryptographic primitives and with support for confidential loading, has a fixed overhead of 995 LUTs and 720 registers. In this revised design, every additional module occupies another 366 LUTs and 212 registers.

Security

The security of confidential loading follows from two observations. First, before `protect` is called, the module's text section is encrypted using the SP's key, which the attacker does not have access to. Second, after the instruction is finished, Sancus' access rules will deny any access to the text section from outside the module. Therefore, only API-level attacks would enable an attacker to read the text section of modules that use confidential loading.

5.4 Conclusion

This chapter first presented Atlas, a scalable security architecture which provides code and data confidentiality for applications through hardware-based memory encryption. Atlas protects intellectual property against system-level attackers in the event of a complete system compromise, using unique IVs for each application. Furthermore, it has a zero-software TCB and also protects against physical attacks on main memory. Our FPGA implementation based on the SPARC LEON3 shows that an existing microcontroller can be extended to include our proposed features with negligible cycle overhead, at the cost of a reduced maximum clock frequency and increased area.

Next, we detailed the addition of confidential loading to Sancus, which was enabled by the availability of a flexible cryptographic building block implemented in hardware, SpongeWrap. Interestingly, this one AEAD primitive can be reused for all cryptographic operations required by Sancus, supporting new functionality without requiring the introduction of additional expensive hardware blocks. Our evaluation indicates that the hardware-based design outperforms a software-based implementation by an order of magnitude.

6 Conclusion

Closing this thesis, we first summarise the contributions of each chapter in Section 6.1. Next, Section 6.2 lists possible avenues for future work in the area of trusted computing architectures. We structure this discussion similarly to this thesis itself, starting with cryptographic primitives and moving up through the design hierarchy to the platform level. This section also covers additional protection mechanisms and touches on a new class of side-channel attacks that exploit fundamental functionality of modern processors.

6.1 Contributions

Chapter 1 showed the effects exploitable software vulnerabilities can have, where we focused on attacks against large public and enterprise systems, but the same holds for our personal devices. In each example, attackers managed to make devices perform in ways that were not intended by their designers nor expected by their users. We then detailed how this is addressed by trusted computing architectures and discussed the RoTs that are at the core of these solutions. Next, some basic computer architecture concepts were introduced and we described the general attacker model of trusted computing solutions, followed by the contributions this thesis made to the design and implementation of such architectures. This chapter concluded with an overview of publications that were not included in this text.

Next, we surveyed hardware-based trusted computing architectures in Chapter 2, selecting designs that offer either or both isolation and attestation mechanisms. This chapter defined basic trusted computing terminology and specified security properties and architectural features that are commonly considered. Finally, we presented summaries of thirteen architectures that fit our criteria and compared them with respect to these properties and features.

Starting at the lowest level of the design hierarchy, Chapter 3 presented a performance analysis of seven block ciphers when their structure is fully unrolled. Since cryptographic primitives are a core building block of hardware-based RoTs, the latency of their hardware implementations is an important factor. When the algorithm is fully unrolled, it can be realised as a single combinational circuit. However, given the complex structure of these ciphers, this results in a high logic depth and longer delay, which will impact the clock frequency of the overall architecture. Furthermore, this implementation approach also increases the area requirements of the implementation, as many algorithms have regular structures that allow for component reuse. Our analysis showed that cryptographers are starting to optimise their designs for low latency and we also gave recommendations on how to optimise for this.

Chapter 4 then introduced a lightweight key distribution mechanism, combining a hardware-based KDF with infrequent asymmetric operations. This work was driven by the observation that lightweight architectures typically rely on hardware implementations of symmetric algorithms, where a unique key per device is generated during production and fixed in hardware. However, a copy of this key needs to be shared with at least one party, potentially exposing it to compromise. Eleutheria derives an application-specific key in hardware, which is securely sent back to its developer over a secure channel, relying on limited use of asymmetric cryptography implemented in software. Furthermore, the application's binary is included in the KDF, attesting it at the same time. We prototyped Eleutheria as an IP core for Arm processors, demonstrating the design's flexibility and showing that the performance of our prototype is comparable to traditional approaches for protecting short messages.

Finally, we detailed two architectural mechanisms ensuring the confidentiality of code and data. First, Atlas inserted a transparent encryption unit in the memory hierarchy right before the cache. All code and data that is read or written by the processor is respectively decrypted or encrypted. Including the application identity in the encryption process ensures that other software cannot access the encrypted information, protecting its confidentiality. The design of the encryption unit was also extended to support secure shared memory communication. We implemented Atlas on the LEON3 open-source processor, where the selection of the encryption algorithm relied on the analysis from Chapter 3. Second, we extended Sancus to support loading encrypted binaries. The design and implementation of this functionality were possible with very little overhead, due to the flexibility of Sancus' custom instructions and the availability of a versatile AEAD implementation in hardware.

6.2 Future Work

This final section presents possible directions for future research related to trusted computing. We first cover low-latency cryptography (Section 6.2.1), followed by short descriptions of capability machines (Section 6.2.2) and CFI (Section 6.2.3). Section 6.2.4 introduces microarchitectural side-channel attacks, an inevitable result of the never-ending search for performance, often at the expense of security. Finally, we discuss some security challenges of hybrid CPU-FPGA platforms in Section 6.2.5.

6.2.1 Low-Latency Cryptography

Both the architectures that were surveyed in Chapter 2 and our own designs show that many lightweight solutions rely on symmetric ciphers. However, we have seen that these algorithms will often limit overall performance, as their structures result in long critical paths, reducing the design's clock frequency. Widespread adoption of trusted computing technology therefore hinges in part on the availability of compact and low-latency cryptographic building blocks.

Our analysis of PRINCE (Section 3.3.4) shows that cryptographers are becoming aware of this constraint and starting to optimise for it. Building on PRINCE, the QARMA block cipher family [169] was specifically designed to have a short critical path, and intended for new security functionality in Arm processors [170]. We believe that continuing this trend will require collaboration between cryptographers and hardware designers. In addition, this line of work might also consider investigating fast hash functions or even asymmetric algorithms, as well as the modes of operation for these primitives.

6.2.2 Capability Machines

The premise of PMAs is the isolation of sensitive application code, but this also means that they only provide coarse-grained security, as modules typically comprise larger functional blocks. While the isolation mechanism drastically reduces the attack surface, internal vulnerabilities cannot be mitigated. On the other hand, *compartmentalised execution* implements the principle of least privilege, giving programs just the rights required for their operation. To this end, programs are assigned capabilities, which are unforgeable tokens authorising them to carry out operations within an address space [171]. Furthermore, capabilities can be manipulated and delegated through controlled instructions.

The CHERI architecture [171, 172] is such a capability machine, enforcing fine-grained memory access control and protection domains. Capabilities represent permissions inside a memory range and are encoded by fat pointers, which are loaded into a special register file and provided as arguments to new memory operation instructions. This not only applies to data manipulation, but also execution flow. Maintaining backwards compatibility, legacy instructions are processed with respect to an implicit capability configurable by the program. On top of this mechanism, object capabilities were realised through OS support, enabling fine-grained compartmentalised execution, as protection domains are crossed with each object invocation.

Although capabilities enable implementation of complex security policies, they come with significant performance overhead. The authors of CHERI report overheads between 5% and 20% for the memory capability mechanism, as well as a longer critical path. This presents opportunities for further optimization, both at the conceptual and implementation level. One possible avenue for future work, which is also suggested by the authors of CHERI, is the composition of capability mechanisms with trusted computing architectures. Their combination would protect modules from memory vulnerabilities, while restricting the incurred overhead. Finally, developers should still take care during implementation, as errors could still lead to policy violations, e.g., when capabilities are leaked.

6.2.3 Control Flow Integrity

As introduced in Chapter 2, code-reuse attacks manage to make an application's control flow deviate from its intended path, realising new functionality without introducing any code. This is done by chaining together so-called gadgets, subroutines that end in a return or jump instruction. Since a module is isolated from all other software running on the system, PMAs reduce the number of gadgets that are available, but will not protect against them. Countermeasures have therefore been proposed to prevent manipulation of the control flow, which are referred to as Control Flow Integrity (CFI) solutions.

Software-based mitigations include Address Space Layout Randomization (ASLR) [173], which was introduced by the Linux PaX project in July 2011, and stack canaries [174, 175], but these have been evaded by more sophisticated attackers. Whereas isolation and attestation mechanisms are well-studied and have mature implementations, hardware-based CFI solutions have only recently gained traction. We refer to the following survey by de Clercq and Verbauwhede [176] for an overview of current hardware-based CFI designs.

One example of such an architecture is SOFIA [18, 19], which gives strong security guarantees, preventing the effects of malicious memory operations by

combining encryption and authentication primitives with the information from a precise Control Flow Graph (CFG). However, this comes with significant performance overhead, both in terms of cycles and maximum clock frequency. The latter is mainly caused by the cryptographic algorithms that are on the processor's critical path, again showing the need for low-latency primitives. Furthermore, static CFI policies like SOFIA's do not consider the stack's runtime state and allow any valid backwards edge to be taken when returning. Control-flow bending attacks exploit this limitation to redirect the execution flow along allowed paths by manipulating non-control data [177], necessitating additional countermeasures to fully mitigate memory corruption vulnerabilities.

Finally, C-FLAT [178] introduced Control Flow Attestation (CFA), which allows a remote verifier to validate the control flow path that was taken at runtime. Traditional attestation mechanisms statically prove application integrity before it is executed in an isolated environment, ensuring that the attested binary cannot be modified. We can draw a clear parallel between the attestation and isolation mechanisms discussed in this thesis, and CFA and CFI respectively. These four solutions all complement each other, so that improving each of them individually is an interesting challenge in its own right, but integrating and combining them presents unique directions for future work.

6.2.4 Speculative Execution

In early 2018, two vulnerabilities were introduced that exploit behaviour which is fundamental to modern high-performance processors. In order to increase performance, their designs have very deep microarchitectural pipelines. To keep this pipeline filled and avoid bubbles, these processors speculate which instruction path will be followed. If they guessed incorrectly, the work that was done is rolled back and execution resumes.

However, the authors of SPECTRE [13] and Meltdown [14] discovered that some effects of the speculative path remain, resulting in side channels that can be exploited from software. Nemesis [16] and Foreshadow [15] respectively demonstrated that SGX and Sancus are also vulnerable to such microarchitectural side channels, showing that this novel class of attacks should be taken into account when designing trusted computing mechanisms.

Given the complexity of modern processors, developing countermeasures for these attacks should be done from the ground up. To this end, the impact of pipelining and speculation on a shallow pipeline should be investigated first, addressing the discovered vulnerabilities. Furthermore, these smaller architectures can also be modelled, which enables methodological security evaluations through formal verification.

6.2.5 Hybrid CPU-FPGA Platforms

Almost every FPGA vendor now offers an SoC product which combines a hardcore with reprogrammable fabric, similar to the Xilinx Zynq board that was used in Chapter 4. These give designers a powerful prototyping environment for hardware-software co-design, as the hardwired processor and fabric are interconnected, enabling fast and straightforward communication in both directions. However, these devices are also used in production settings, targeting applications that benefit from hardware-based acceleration.

The use of these hybrid CPU-FPGA platforms therefore also introduces security challenges at different levels. First, most designs support direct access to the processor's main memory, without the access control rules software is subject to. It is therefore possible to build malicious hardware blocks that leak confidential code or data. Second, the components that are programmed into the FPGA represent valuable intellectual property themselves. Their memory-mapped interfaces are accessible to any software running on the system, whereas their developer will want to ensure that it can only be queried by their applications.

Finally, cloud environments are challenging due to their multi-tenancy. Currently, instances that include access to an FPGA are assigned to a single customer. The cost of their resources therefore cannot be amortised, increasing their price. Sharing of the fabric through partial reconfiguration is possible, but brings us back to the second issue raised in the previous paragraph. Furthermore, one could envision the cloud provider offering a central store listing third-party accelerators compatible with their platform. Such a model raises complex questions about the trust relationships between customers, third-party hardware developers, and the cloud provider hosting the hybrid platform.

6.3 Conclusion

In this final chapter, brief summaries first highlighted the contributions of this thesis. We then identified avenues to advance the goal of trusted computing and prevent applications from misbehaving. Collaboration between cryptographers and hardware designers could inspire new algorithmic techniques to further reduce the latency of cryptographic primitives. Since no single solution is a silver bullet, novel architectural mechanisms can further limit the attack surface. Finally, evolving exploit strategies and growing platform complexity will continue to drive the development of hardware-based protection mechanisms.

§ Bibliography

[1] K. Zetter, *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon*. Crown Publishing Group, 2014. « Cited on p. 1. »

[2] ——, "Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid", *WIRED*, 2016. « Cited on p. 1. »

[3] A. Greenberg, "The Untold Story of NotPetya, the Most Devastating Cyberattack in History", *WIRED*, 2018. « Cited on p. 1. »

[4] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization", in *Proceedings of the 3rd Conference on Computer Systems*, 2008. « Cited on pp. 2, 15, 24. »

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Sixth Edition. Morgan Kaufmann Publishers, Inc., 2017. « Cited on p. 3. »

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, May 2019. « Cited on p. 5. »

[7] *ARM Architecture Reference Manual*, ARMv8, for ARMv8-A Architecture Profile, Apr. 2019. « Cited on p. 5. »

[8] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols", *Transactions on Information Theory*, vol. 29, no. 2, 1983. « Cited on pp. 6, 16, 69, 94. »

[9] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", in *Proceedings of the 16th Conference on Cryptology*, 1996. « Cited on p. 7. »

[10] P. Kocher, J. Jaffe and B. Jun, "Differential Power Analysis", in *Proceedings of the 19th Conference on Cryptology*, 1999. « Cited on p. 7. »

[11] K. Gandolfi, C. Mourtel and F. Olivier, "Electromagnetic Analysis: Concrete Results", in *Proceedings of the 3rd Conference on Cryptographic Hardware and Embedded Systems*, 2001. « Cited on p. 7. »

[12] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks against AES", in *Proceedings of the 8th Conference on Cryptographic Hardware and Embedded Systems*, 2006. « Cited on p. 7. »

[13] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution", *CoRR*, vol. abs/1801.01203, 2018. « Cited on pp. 7, 121. »

[14] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, "Meltdown: Reading kernel memory from user space", in *Proceedings of the 27th USENIX Symposium on Security*, 2018. « Cited on pp. 7, 121. »

[15] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution", in *Proceedings of the 27th USENIX Security Symposium*, 2018. « Cited on pp. 7, 19, 46, 121. »

[16] J. Van Bulck, F. Piessens and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic", in *Proceedings of the 25th Conference on Computer and Communications Security*, 2018. « Cited on pp. 7, 19, 46, 121. »

[17] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling and I. Verbauwhede, "Soteria: Offline Software Protection within Low-cost Embedded Devices", in *Proceedings of the 31st Conference on Computer Security Applications*, 2015. « Cited on pp. 9, 32, 43, 89, 108, 110. »

[18] R. de Clercq, R. De Keulenaer, B. Coppens, B. Yang, K. De Bosschere, B. De Sutter, P. Maene, B. Preneel and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", in *Proceedings of the 19th Conference on Design, Automation and Test*, 2016. « Cited on pp. 9, 10, 68, 120. »

[19] R. de Clercq, J. Götzfried, P. Maene, D. Übler and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", *Computers & Security*, vol. 68, no. 7, 2017. « Cited on pp. 9, 10, 120. »

[20] F. Turan, R. de Clercq, P. Maene, O. Reparaz and I. Verbauwhede, "Hardware Acceleration of a Software-Based VPN", in *Proceedings of the 26th Conference on Field Programmable Logic and Applications*, 2016. « Cited on p. 10. »

[21] R. de Clercq, R. De Keulenaer, P. Maene, B. De Sutter, B. Preneel and I. Verbauwhede, "SCM: Secure Code Memory Architecture", in *Proceedings of the 12th Conference on Computer and Communications Security*, 2017. « Cited on pp. 10, 11. »

[22] T. Ashur, J. Delvaux, S. Lee, P. Maene, E. Marin, S. Nikova, O. Reparaz, V. Rožić, D. Singelée, B. Yang and B. Preneel, "A Privacy-Preserving Device Tracking System Using a Low-Power Wide-Area Network (LPWAN)", in *Proceedings of the 16th Conference on Cryptology and Network Security*, 2017. « Cited on p. 11. »

[23] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation", *IEEE Transactions on Computers*, vol. 67, no. 3, 2017. « Cited on p. 13. »

[24] A. Martin, *The Ten Page Introduction to Trusted Computing*, 2008. « Cited on p. 14. »

[25] D. Gollmann, "Why Trust is Bad for Security", *Electronic Notes in Theoretical Computer Science*, vol. 157, no. 3, 2006. « Cited on p. 14. »

[26] C. Mundie, P. de Vries, P. Haynes and M. Corwine, "Trustworthy Computing", Microsoft, Tech. Rep., 2002. « Cited on p. 14. »

[27] S. Lipner, "The Trustworthy Computing Security Development Lifecycle", in *Proceedings of the 20th Conference on Computer Security Applications*, 2004. « Cited on p. 14. »

[28] A. Seshadri and A. Perrig, "SWATT: Software-Based Attestation for Embedded Devices", in *Proceedings of the 25th IEEE Symposium on Security and Privacy*, 2004. « Cited on p. 14. »

[29] L. Martignoni, R. Paleari and D. Bruschi, "Conqueror: Tamper-Proof Code Execution on Legacy Systems", in *Proceedings of the 7th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010. « Cited on p. 14. »

[30] R. Jayaram Masti, C. Marforio and S. Capkun, "An Architecture for Concurrent Execution of Secure Environments in Clouds", in *Proceedings of the 20th Workshop on Cloud Computing Security Workshop*, 2013. « Cited on p. 14. »

[31] R. Strackx and F. Piessens, "Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware", in *Proceedings of the 19th Conference on Computer and Communications Security*, 2012. « Cited on pp. 14, 24. »

[32] N. Avonds, R. Strackx, P. Agten and F. Piessens, "Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege", in *Proceedings of the 9th Conference on Security and Privacy in Communication Networks*, 2013. « Cited on p. 14. »

[33] A. J. Menezes, S. A. Vanstone and P. C. V. Oorschot, *Handbook of Applied Cryptography*, First Edition. CRC Press, Inc., 1996. « Cited on pp. 15, 49. »

[34] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel and F. Piessens, "Protected Software Module Architectures", in *Proceedings of the 13th Conference on Information Security Solutions*, 2013. « Cited on p. 15. »

[35] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", in *Proceedings of the 14th Conference on Computer and Communications Security*, 2007. « Cited on p. 18. »

[36] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing", in *Proceedings of the 17th Conference on Supercomputing*, 2003. « Cited on pp. 21, 43. »

[37] *TPM Main: Part 1 Design Principles*, Version 1.2, Revision 116, Trusted Computing Group, 2011. « Cited on pp. 22, 43. »

[38] K. Kursawe, D. Schellekens and B. Preneel, "Analyzing Trusted Platform Communication", in *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware*, 2005. « Cited on p. 23. »

[39] *Trusted Platform Module Library: Part 1: Architecture*, Family 2.0, Level 00, Revision 01.16, Trusted Computing Group, 2014. « Cited on p. 23. »

[40] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, First Edition. Intel Press, 2009. « Cited on pp. 24, 43. »

[41] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation", in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010. « Cited on p. 24. »

[42] *GlobalPlatform Device Technology TEE Client API Specification*, Revision 0.17, GlobalPlatform, 2010. « Cited on p. 24. »

[43] *GlobalPlatform Device Technology TEE Internal API Specification*, Revision 1.0, GlobalPlatform, 2011. « Cited on p. 24. »

[44] *Security Technology Building a Secure System Using TrustZone Technology*, 2009. « Cited on pp. 25, 43, 87. »

[45] D. Champagne and R. B. Lee, "Scalable Architectural Support for Trusted Software", in *Proceedings of the 16th Conference on High-Performance Computer Architecture*, 2010. « Cited on pp. 27, 43. »

[46] K. Eldefrawy, A. Francillon, D. Perito and G. Tsudik, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust", in *Proceedings of the 19th Symposium on Network and Distributed System Security*, 2012. « Cited on pp. 28, 43. »

[47] A. Francillon, Q. Nguyen, K. B. Rasmussen and G. Tsudik, "A Minimalist Approach to Remote Attestation", in *Proceedings of the 17th Conference on Design, Automation and Test*, 2014. « Cited on p. 28. »

[48] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", Standard, 1997. « Cited on pp. 28, 112. »

[49] N. Asokan, F. Brasser, A. Ibrahim, A. Sadeghi, M. Schunter, G. Tsudik and C. Wachsmann, "SEDA: Scalable Embedded Device Attestation", in *Proceedings of the 22nd Conference on Computer and Communications Security*, 2015. « Cited on p. 29. »

[50] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner and G. Tsudik, "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation", in *Proceedings of the 28th USENIX Security Symposium*, 2019. « Cited on p. 29. »

[51] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base", in *Proceedings of the 22nd USENIX Symposium on Security*, 2013. « Cited on pp. 29, 43, 87, 93, 108, 110. »

[52] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices", *ACM Transactions on Privacy and Security*, vol. 20, no. 3, 2017. « Cited on pp. 29, 43, 89, 91, 110. »

[53] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı and I. Verbauwhede, "Spongent: A Lightweight Hash Function", in *Proceedings of the 13th Workshop on Cryptographic Hardware and Embedded Systems*, 2011. « Cited on pp. 31, 113. »

[54] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, "Duplexing the Sponge: Single-pass Authenticated Encryption and Other Applications", in *Proceedings of the 18th Workshop on Selected Areas in Cryptography*, 2011. « Cited on pp. 32, 113. »

[55] P. Williams and R. Boivie, "CPU Support for Secure Executables", in *Proceedings of the 4th Conference on Trust and Trustworthy Computing*, 2011. « Cited on pp. 33, 43. »

[56] R. Boivie and P. Williams, "SecureBlue++: CPU Support for Secure Executables", IBM, Tech. Rep., 2013. « Cited on p. 33. »

[57] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution", in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. « Cited on pp. 34, 43. »

[58] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade and J. del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions", in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. « Cited on p. 34. »

[59] I. Anati, S. Gueron, S. P. Johnson and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing", in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. « Cited on p. 34. »

[60] "Intel Software Guard Extensions Programming Reference (329298-002US)", Intel, Tech. Rep., 2014. « Cited on p. 34. »

[61] S. Gueron, *A Memory Encryption Engine Suitable for General Purpose Processors*, Cryptology ePrint Archive, Report 2016/204, 2016. « Cited on p. 35. »

[62] S. P. Johnson, V. R. Scarlata, C. V. Rozas, E. Brickell and F. McKeen, "Intel SGX: EPID Provisioning and Attestation Services", Intel, Tech. Rep., 2016. « Cited on p. 35. »

[63] V. Costan and S. Devadas, "Intel SGX Explained", *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, 2016. « Cited on p. 36. »

[64] M. Marlinspike, *Technology Preview: Private Contact Discovery for Signal*, 2017. [Online]. Available: `https://signal.org/blog/private-contact-discovery/`. « Cited on p. 36. »

[65] A. Baumann, M. Peinado and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven", in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014. « Cited on p. 36. »

[66] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX", in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015. « Cited on p. 36. »

[67] D. Kaplan, J. Powell and T. Woller, "AMD Memory Encryption", AMD, Tech. Rep., 2016. « Cited on p. 36. »

[68] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution", in *Proceedings of the 47th Symposium on Microarchitecture*, 2014. « Cited on pp. 36, 43. »

[69] P. Koeberl, S. Schulz, A. Sadeghi and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices", in *Proceedings of the 9th Conference on Computer Systems*, 2014. « Cited on pp. 37, 43. »

[70] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices", in *Proceedings of the 52nd Conference on Design Automation*, 2015. « Cited on pp. 39, 43. »

[71] V. Costan, I. Lebedev and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation", *IACR Cryptology ePrint Archive*, vol. 2015, no. 564, 2015. « Cited on pp. 40, 43. »

[72] S. Weiser, M. Werner, F. Brasser, M. Malenko and A. Sadeghi, "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V", in *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019. « Cited on pp. 41, 43. »

[73] J. Coburn, S. Ravi, A. Raghunathan and S. Chakradhar, "SECA: Security-enhanced Communication Architecture", in *Proceedings of the 8th Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005. « Cited on p. 45. »

[74] P. Maene and I. Verbauwhede, "Single-Cycle Implementations of Block Ciphers", in *Proceedings of the 4th Workshop on Lightweight Cryptography for Security and Privacy*, ser. Lecture Notes in Computer Science, 2015. « Cited on p. 47. »

[75] W. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", in *Proceedings of the 38th Conference on Design Automation*, 2001. « Cited on p. 48. »

[76] J. Daemen and V. Rijmen, "The Rijndael Algorithm", in *The First Advanced Encryption Standard Candidate Conference*, 1998. « Cited on pp. 48, 51. »

[77] C. De Cannière, O. Dunkelman and M. Knežević, "KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers", in *Proceedings of the 11th Workshop on Cryptographic Hardware and Embedded Systems*, 2009. « Cited on pp. 48, 53. »

[78] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher", in *Proceedings of the 9th Workshop on Cryptographic Hardware and Embedded Systems*, 2007. « Cited on pp. 48, 55. »

[79] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knežević, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen and T. Yalçın, *PRINCE: A Low-latency Block Cipher for Pervasive Computing Applications*, Cryptology ePrint Archive, Report 2012/529, 2012. « Cited on pp. 48, 56. »

[80] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang and I. Verbauwhede, *RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms*, Cryptology ePrint Archive, Report 2014/084, 2014. « Cited on pp. 48, 57. »

[81] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, *The SIMON and SPECK Families of Lightweight Block Ciphers*, Cryptology ePrint Archive, Report 2013/404, 2013. « Cited on pp. 48, 58. »

[82] M. Knežević, V. Nikov and P. Rombouts, "Low-Latency Encryption – Is "Lightweight = Light + Wait"?", in *Proceedings of the 14th Workshop on Cryptographic Hardware and Embedded Systems*, 2012. « Cited on p. 48. »

[83] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 2007. « Cited on p. 50. »

[84] J. M. Rabaey, A. Chandrakasan and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, Second Edition. Prentice-Hall, Inc., 2003. « Cited on pp. 50, 51. »

[85] D. J. Bernstein, *Crypto Competitions — AES: the Advanced Encryption Standard*, 2014. [Online]. Available: `http://competitions.cr.yp.to/aes.html`. « Cited on pp. 52, 104. »

[86] NIST, "Advanced Encryption Standard (AES)", Standard, 2001. « Cited on p. 52. »

[87] A. Bogdanov, D. Khovratovich and C. Rechberger, *Biclique Cryptanalysis of the Full AES*, Cryptology ePrint Archive, Report 2011/449, 2011. « Cited on p. 52. »

[88] S. D. Brown, R. J. Francis, J. Rose and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Springer Science+Business Media, 2012. « Cited on p. 52. »

[89] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, 2007. « Cited on p. 52. »

[90] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao and P. Rohatgi, "Efficient Rijndael Encryption Implementation with Composite Field Arithmetic", in *Proceedings of the 3rd Workshop on Cryptographic Hardware and Embedded Systems*, 2001. « Cited on p. 53. »

[91] P. Hamalainen, T. Alho, M. Hannikainen and T. Hamalainen, "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core", in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, 2006. « Cited on p. 53. »

[92] A. Bogdanov and C. Rechberger, "A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN", in *Proceedings of the 18th Workshop on Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, 2011. « Cited on p. 53. »

[93] S. Rasoolzadeh and H. Raddum, "Improved Multi-Dimensional Meet-in-the-Middle Cryptanalysis of KATAN", *Tatra Mountains Mathematical Publications*, vol. 67, no. 1, 2016. « Cited on p. 54. »

[94] M. H. Faghihi Sereshgi, M. Dakhilalian and M. Shakiba, "Biclique cryptanalysis of MIBS-80 and PRESENT-80 block ciphers", *Security and Communication Networks*, vol. 9, no. 1, 2016. « Cited on p. 55. »

[95] C. Lee, "Biclique Cryptanalysis of PRESENT-80 and PRESENT-128", *The Journal of Supercomputing*, vol. 70, no. 1, 2014. « Cited on p. 55. »

[96] P. Morawiecki, *Practical Attacks on the Round-reduced PRINCE*, Cryptology ePrint Archive, Report 2015/245, 2015. « Cited on p. 56. »

[97] P. Derbez and L. Perrin, *Meet-in-the-Middle Attacks and Structural Analysis of Round-Reduced PRINCE*, Cryptology ePrint Archive, Report 2015/239, 2015. « Cited on p. 56. »

[98] A. Canteaut, T. Fuhr, H. Gilbert, M. Naya-Plasencia and J.-R. Reinhard, *Multiple Differential Cryptanalysis of Round-Reduced PRINCE (Full Version)*, Cryptology ePrint Archive, Report 2014/089, 2014. « Cited on p. 56. »

[99] G. Zhao, B. Sun, C. Li and J. Su, "Truncated Differential Cryptanalysis of PRINCE", *Security and Communication Networks*, vol. 8, no. 16, 2015. « Cited on p. 56. »

[100] J. Jean, I. Nikolić, T. Peyrin, L. Wang and S. Wu, "Security Analysis of PRINCE", in *Proceedings of the 22nd Symposium on the Foundations of Software Engineering*, 2014. « Cited on p. 56. »

[101] J. Shan, L. Hu, L. Song, S. Sun and X. Ma, *Related-Key Differential Attack on Round Reduced RECTANGLE-80*, Cryptology ePrint Archive, Report 2014/986, 2014. « Cited on p. 57. »

[102] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, *Notes on the Design and Analysis of SIMON and SPECK*, Cryptology ePrint Archive, Report 2017/560, 2017. « Cited on pp. 58, 59. »

[103] T. Ashur, *Improved Linear Trails for the Block Cipher Simon*, Cryptology ePrint Archive, Report 2015/285, 2015. « Cited on p. 58. »

[104] L. Song, Z. Huang and Q. Yang, "Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA", in *Proceedings of the 21st Conference on Information Security and Privacy*, 2016. « Cited on p. 59. »

[105] D. Giry and J.-J. Quisquater, *Keylength — ECRYPT II Report on Key Sizes (2012)*, 2014. [Online]. Available: `http://www.keylength.com/en/3/`. « Cited on p. 60. »

[106] P. Maene and I. Verbauwhede, *Eleutheria: Lightweight Key Distribution Service for Networked Embedded Devices*. « Cited on p. 65. »

[107] L. Hay Newman, "The Botnet That Broke the Internet Isn't Going Away", *WIRED*, 2016. « Cited on p. 66. »

[108] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle", *Black Hat USA*, 2015. « Cited on p. 66. »

[109] International Organization for Standardization, "Information Technology – Security Techniques – Entity Authentication – Part 2: Mechanisms Using Symmetric Encipherment Algorithms", Standard, Dec. 2008. « Cited on p. 74. »

[110] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", Standard, 2010. « Cited on pp. 74, 112. »

[111] H. Krawczyk, "SIGMA: The "SIGn-and-MAc" Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols", in *Proceedings of the 23rd Conference on Cryptology*, 2003. « Cited on p. 77. »

[112] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records", in *Proceedings of the 9th Conference on Theory and Practice in Public-Key Cryptography*, 2006. « Cited on p. 77. »

[113] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang, "High-Speed High-Security Signatures", *Cryptographic Engineering*, vol. 2, no. 2, 2012. « Cited on p. 77. »

[114] A. Luykx, B. Preneel, E. Tischhauser and K. Yasuda, "A MAC Mode for Lightweight Block Ciphers", in *Proceedings of the 23rd Conference on Fast Software Encryption*, 2016. « Cited on p. 77. »

[115] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", Standard, 2015. « Cited on p. 77. »

[116] ——, "Advanced Encryption Standard (AES)", Standard, 2001. « Cited on p. 78. »

[117] V. Rožić, B. Yang, W. Dehaene and I. Verbauwhede, "Highly Efficient Entropy Extraction for True Random Number Generators on FPGAs", in *Proceedings of the 52nd Conference on Design Automation*, 2015. « Cited on p. 78. »

[118] A. Langley, *Ed25519 for Go*, 2016. [Online]. Available: `https://github.com/agl/ed25519`. « Cited on p. 78. »

[119] H. Krawczyk, "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", in *Proceedings of the 21st Cryptology Conference*, 2001. « Cited on p. 79. »

[120] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation", Special Publication, 2001. « Cited on p. 79. »

[121] B. Conte, *Basic Implementations of Standard Cryptography Algorithms*, 2015. [Online]. Available: `https://github.com/b-con/crypto-algorithms`. « Cited on p. 80. »

[122] A. Langley, *Implementations of a Fast Elliptic-Curve Diffie-Hellman Primitive*, 2015. [Online]. Available: `https://github.com/agl/curve25519-donna`. « Cited on p. 80. »

[123] A. Moon, *Implementations of a Fast Elliptic-Curve Digital Signature Algorithm*, 2015. [Online]. Available: `https://github.com/floodyberry/ed25519-donna`. « Cited on p. 80. »

[124] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack", Tech. Rep., 2001. « Cited on p. 80. »

[125] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı and I. Verbauwhede, "SPONGENT: The Design Space of Lightweight Cryptographic Hashing", 2011. « Cited on pp. 80, 86. »

[126] *ARM CoreSight Architecture Specification*, 2017. « Cited on p. 82. »

[127] *CoreSight Program Flow Trace*, 2011. « Cited on p. 83. »

[128] *CoreSight Components*, 2009. « Cited on pp. 83, 88. »

[129] P. Sasdrich and T. Güneysu, "Efficient Elliptic-curve Cryptography using Curve25519 on Reconfigurable Devices", in *Proceedings of the 10th Symposium on Applied Reconfigurable Computing*, 2014. « Cited on p. 86. »

[130] F. Turan and I. Verbauwhede, "Compact and Flexible FPGA Implementation of Ed25519 and X25519", *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 3, 2019. « Cited on p. 86. »

[131] *CoreSight PTM-A9*, 2011. « Cited on p. 87. »

[132] E. Wobber, M. Abadi, M. Burrows and B. Lampson, "Authentication in the Taos Operating System", *ACM Transactions on Computer Systems*, vol. 12, no. 1, 1994. « Cited on p. 89. »

[133] B. C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks", *IEEE Communications Magazine*, vol. 32, no. 9, 1994. « Cited on p. 89. »

[134] M. B. Jones, J. Bradley and N. Sakimura, "JSON Web Token (JWT)", Standard, 2015. « Cited on p. 89. »

[135] M. A. Simplício Jr., P. S. Barreto, C. B. Margi and T. C. Carvalho, "A Survey on Key Management Mechanisms for Distributed Wireless Sensor Networks", *Computer Networks*, vol. 54, no. 15, 2010. « Cited on p. 89. »

[136] P. Maene, J. Götzfried, T. Müller, R. de Clercq, F. Freiling and I. Verbauwhede, "Atlas: Application Confidentiality in Compromised Embedded Systems", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2018. « Cited on p. 91. »

[137] R. Obermaisser, P. Peti and F. Tagliabo, "An Integrated Architecture for Future Car Generations", *Real-Time Systems*, vol. 36, no. 1, 2007. « Cited on p. 92. »

[138] P. Oester, *Dirty COW (CVE-2016-5195)*, MITRE, 2016. « Cited on p. 92. »

[139] G. Beniamini, *CVE-2017-6956*, MITRE, 2017. « Cited on p. 92. »

[140] ——, *CVE-2017-6975*, MITRE, 2017. « Cited on p. 92. »

[141] C. Tarnovsky, "Deconstructing a "Secure" Processor", *Black Hat DC*, 2010. « Cited on p. 94. »

[142] *GRLIB IP Core User's Manual*, Version 2019.2, 2019. « Cited on p. 97. »

[143] M. Liskov, R. L. Rivest and D. Wagner, "Tweakable Block Ciphers", *Journal of Cryptology*, vol. 24, no. 3, 2011. « Cited on p. 98. »

[144] ECRYPT II, *Yearly Report on Algorithms and Keysizes*, 2012. « Cited on p. 99. »

[145] C. Fruhwirth, "New Methods in Hard Disk Encryption", Tech. Rep., 2005. « Cited on pp. 99, 107. »

[146] N. Ferguson, *AES-CBC+ Elephant Diffuser: A Disk Encryption Algorithm for Windows Vista*, 2006. « Cited on pp. 99, 107. »

[147] *The SPARC Architecture Manual*, Version 8, 1992. « Cited on p. 100. »

[148] A. P. Chandrakasan, S. Sheng and R. W. Brodersen, "Low-power CMOS Digital Design", *IEICE Transactions on Electronics*, vol. 75, no. 4, 1992. « Cited on p. 103. »

[149] M. Henson and S. Taylor, "Memory Encryption: A Survey of Existing Techniques", *ACM Computer Surveys*, vol. 46, no. 4, 2013. « Cited on p. 107. »

[150] N. Provos, "Encrypting Virtual Memory", in *Proceedings of the 9th USENIX Symposium on Security*, 2000. « Cited on p. 107. »

[151] G. Duc and R. Keryell, "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection", in *Proceedings of the 22nd Conference on Computer Security Applications*, 2006. « Cited on p. 107. »

[152] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger and M. Backes, "RamCrypt: Kernel-based Address Space Encryption for User-mode Processes", in *Proceedings of the 11th Conference on Computer and Communications Security*, 2016. « Cited on p. 107. »

[153] P. Peterson, "Cryptkeeper: Improving Security with Encrypted RAM", in *Proceedings of the 10th Conference on Technologies for Homeland Security*, 2010. « Cited on p. 107. »

[154] J. Götzfried, N. Dörr, R. Palutke and T. Müller, "HyperCrypt: Hypervisor-based Encryption of Kernel and User Space", in *Proceedings of 11th Conference on Availability, Reliability and Security*, 2016. « Cited on p. 107. »

[155] T. Müller, F. Freiling and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM", in *Proceedings of the 20th USENIX Symposium on Security*, 2011. « Cited on p. 107. »

[156] P. Simmons, "Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption", *Proceedings of the 27th Conference on Computer Security Applications*, 2011. « Cited on p. 107. »

[157] J. Götzfried and T. Müller, "Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption", *Transactions on Information and System Security*, vol. 17, no. 2, 2014. « Cited on p. 107. »

[158] J. Pabel, *Frozen Cache*, 2009. [Online]. Available: `https://frozencache.blogspot.com`. « Cited on p. 107. »

[159] N. Heninger and H. Shacham, "Reconstructing RSA Private Keys from Random Key Bits", in *Proceedings of the 29th Conference on Cryptology*, 2009. « Cited on p. 107. »

[160] T. P. Parker and S. Xu, "A Method for Safekeeping Cryptographic Keys from Memory Disclosure Attacks", in *Proceedings of the 1st Conference on Trusted Systems*, 2009. « Cited on p. 107. »

[161] B. Garmany and T. Müller, "PRIME: private RSA Infrastructure for Memory-Less Encryption", in *Proceedings of the 29th Conference on Computer Security Applications*, 2013. « Cited on p. 107. »

[162] L. Guan, J. Lin, B. Luo and J. Jing, "Copker: Computing with Private Keys without RAM", in *Proceedings of the 21st Symposium on Network and Distributed System Security*, 2014. « Cited on p. 107. »

[163] L. Guan, J. Lin, B. Luo, J. Jing and J. Wang, "Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory", in *Proceedings of the 36th Symposium on Security and Privacy*, 2015. « Cited on p. 107. »

[164] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices", IEEE, Standard, 2008. « Cited on p. 107. »

[165] "Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices", National Institute of Standards and Technology, Special Publication, 2010. « Cited on p. 107. »

[166] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles and R. Barua, "Binary rewriting without relocation information", University of Maryland, Tech. Rep., 2010. « Cited on p. 109. »

[167] E. Andreeva, J. Daemen, B. Mennink and G. Van Assche, "Security of Keyed Sponge Constructions Using a Modular Proof Approach", in *Proceedings of the 22nd Conference on Fast Software Encryption*, 2015. « Cited on p. 113. »

[168] E. Andreeva, B. Bilgin, A. Bogdanov, A. Luykx, B. Mennink, N. Mouha and K. Yasuda, "APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography", in *Proceedings of the 21st Workshop on Fast Software Encryption*, 2014. « Cited on p. 113. »

[169] R. Avanzi, "The QARMA Block Cipher Family", *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, 2017. « Cited on p. 119. »

[170] "Pointer Authentication on ARMv8.3", Qualcomm Technologies, Tech. Rep., 2017. « Cited on p. 119. »

[171] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk", in *Proceedings of the 41st Symposium on Computer Architecture*, 2014. « Cited on pp. 119, 120. »

[172] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization", in *Proceedings of the 36th Symposium on Security and Privacy*, 2015. « Cited on p. 120. »

[173] PaX Team, "Address Space Layout Randomization", Tech. Rep., 2003. [Online]. Available: `https://pax.grsecurity.net/docs/aslr.txt`. « Cited on p. 120. »

[174] A. Baratloo, N. Singh and T. K. Tsai, "Transparent Run-Time Defense Against Stack-Smashing Attacks", in *Proceedings of the 6th USENIX Technical Conference*, 2000. « Cited on p. 120. »

[175] C. Cowan, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", in *Proceedings of the 7th USENIX Security Symposium*, 1998. « Cited on p. 120. »

[176] R. de Clercq and I. Verbauwhede, "A Survey of Hardware-based Control Flow Integrity (CFI)", *CoRR*, vol. abs/1706.07257, 2017.
« Cited on p. 120. »

[177] N. Carlini, A. Barresi, M. Payer, D. Wagner and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity", in *Proceedings of the 24th USENIX Security Symposium*, 2015.
« Cited on p. 121. »

[178] T. Abera, N Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A. Sadeghi and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software", in *Proceedings of the 23rd Conference on Computer and Communications Security*, 2016. « Cited on p. 121. »

§ Curriculum Vitae

Pieter Maene was born on September 2nd, 1991 in Duffel, Belgium. He obtained his Master's degree in Electrical Engineering from KU Leuven in July 2014, completing his thesis on the topic of online elections. He also served on the board of the student organisation affiliated with the Faculty of Engineering Science. In October 2014, he joined the COSIC research group at the Department of Electrical Engineering of KU Leuven. From January 2016 onward, his research was funded by a Strategic Basic Research Grant from the Research Foundation - Flanders. During his PhD, he collaborated closely with the IT Security chair at FAU Erlangen-Nürnberg. From June to September 2018, he was a hardware security intern at Square in San Francisco.

§ List of Publications

Journals

[1] P. Maene, J. Götzfried, T. Müller, R. de Clercq, F. Freiling and I. Verbauwhede, "Atlas: Application Confidentiality in Compromised Embedded Systems", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2018.

[2] R. de Clercq, J. Götzfried, P. Maene, D. Übler and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", *Computers & Security*, vol. 68, no. 7, 2017.

[3] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller and F. Freiling, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices", *ACM Transactions on Privacy and Security*, vol. 20, no. 3, 2017.

[4] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation", *IEEE Transactions on Computers*, vol. 67, no. 3, 2017.

International Conferences

[5] J. Hermans, R. Peeters, P. Maene, K. Grenman, K. Halunen and J. Häikiö, "n-Auth: Mobile Authentication Done Right", in *Proceedings of the 33rd Conference on Computer Security Applications*, 2017.

[6] T. Ashur, J. Delvaux, S. Lee, P. Maene, E. Marin, S. Nikova, O. Reparaz, V. Rožić, D. Singelée, B. Yang and B. Preneel, "A Privacy-Preserving Device Tracking System Using a Low-Power Wide-Area Network (LPWAN)", in *Proceedings of the 16th Conference on Cryptology and Network Security*, 2017.

[7] R. de Clercq, R. De Keulenaer, P. Maene, B. De Sutter, B. Preneel and I. Verbauwhede, "SCM: Secure Code Memory Architecture", in *Proceedings of the 12th Conference on Computer and Communications Security*, 2017.

[8] F. Turan, R. de Clercq, P. Maene, O. Reparaz and I. Verbauwhede, "Hardware Acceleration of a Software-Based VPN", in *Proceedings of the 26th Conference on Field Programmable Logic and Applications*, 2016.

[9] R. de Clercq, R. De Keulenaer, B. Coppens, B. Yang, K. De Bosschere, B. De Sutter, P. Maene, B. Preneel and I. Verbauwhede, "SOFIA: Software and Control Flow Integrity Architecture", in *Proceedings of the 19th Conference on Design, Automation and Test*, 2016.

[10] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling and I. Verbauwhede, "Soteria: Offline Software Protection within Low-cost Embedded Devices", in *Proceedings of the 31st Conference on Computer Security Applications*, 2015.

[11] P. Maene and I. Verbauwhede, "Single-Cycle Implementations of Block Ciphers", in *Proceedings of the 4th Workshop on Lightweight Cryptography for Security and Privacy*, ser. Lecture Notes in Computer Science, 2015.

Miscellaneous

[12] P. Maene, "Online verkiezingen in de praktijk: verbetering en toepassing van het Helios verkiezingssysteem", Bart Preneel (Promotor), Master's thesis, KU Leuven, 2014.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COMPUTER SECURITY AND INDUSTRIAL CRYPTOGRAPHY
Kasteelpark Arenberg 10 bus 2452
B-3001 Leuven
pieter.maene@esat.kuleuven.be
https://cosic.esat.kuleuven.be