ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE

Ciclo 38

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

HARDWARE-SOFTWARE CO-DESIGN METHODS TO UNLEASH OPEN HARDWARE CYBERSECURITY AND TRUST IN RISC-V ARCHITECTURES

Presentata da: Alberto Musa

Coordinatore Dottorato

Davide Dardari

Supervisore

Andrea Acquaviva

Co-supervisore

Francesco Barchi

Esame finale anno 2026

Abstract

The increasing criticality of embedded systems, particularly within the rapidly expanding open-source RISC-V ecosystem, demands advanced defense mechanisms against a wide range of threats. These include sophisticated software exploits, hardware-based attacks, and the emerging challenges posed by quantum computing. OpenTitan, as a robust open-source silicon Root-of-Trust (RoT), provides a transparent and verifiable foundation for secure computing, offering essential capabilities such as secure boot, hardware-based isolation, and dedicated cryptographic accelerators. However, effectively harnessing OpenTitan's powerful capabilities and seamlessly integrating its advanced security features throughout the rapidly evolving RISC-V ecosystem presents a critical challenge due to a persistent gap between sophisticated hardware advancements and the maturity of corresponding software stacks. This disparity leads to difficulties in fully leveraging security features, performance bottlenecks, and limited developer accessibility.

This thesis introduces a holistic hardware-software co-design approach to address these critical gaps, culminating in a secure, high-performance, and future-proof embedded computing paradigm. The central contribution is TitanSSL, a meticulously engineered software stack designed to not only transparently offload cryptographic operations to OpenTitan's dedicated hardware accelerators, but also to extend its RoT benefits, such as secure key storage and system integrity, to the broader RISC-V ecosystem. TitanSSL unites three synergistic components: a custom OpenSSL engine for application integration, optimized OpenTitan firmware for managing cryptographic workloads, and a robust Linux kernel driver. An advanced, secure communication protocol inextricably links these elements, meticulously orchestrating data exchange, memory protection, and resource arbitration between the host processor and OpenTitan. A rigorous performance evaluation, conducted on a CVA6 application core running Linux and an OpenTitan instance on a Xilinx VCU118 FPGA, demonstrates substantial speedups for typical cryptographic workloads compared to software-only implementations. Specifically, for larger payloads, such as 128 KiB, the framework ensures consistent outperformance, reaching speedups of 10.50x for SHA-256 and 2.96x for AES-256-CBC, although accelerator utilization reaches only 9.35% of the theoretical peak due to memory-bound constraints.

This observed performance gap stems from inherent data movement bottlenecks in early OpenTitan integrations that limited peak utilization. To effectively address these limitations and fully leverage OpenTitan's capabilities, architectural enhancements have been implemented, focusing on Direct Memory Access (DMA) and Tightly Coupled Data Memory (TCDM). By offloading data transfers from the CPU and providing low-latency, high-bandwidth memory close to the accelerators, these modifications substantially reduce CPU overhead and improve accelerator utilization. This led to significant improvements in cryptographic throughput, achieving up to 2.2x speedup over the baseline TitanSSL without DMA and raising hardware utilization to 20.56% for large payloads.

Beyond cryptographic acceleration, this work significantly enhances system integrity through TitanCFI, a novel Control-Flow Integrity enforcement mechanism. Leveraging OpenTitan's RoT, TitanCFI provides robust runtime protection against sophisticated code-reuse attacks (e.g., Return-Oriented Programming) with minimal performance impact, by streaming control-flow instructions and securing metadata using OpenTitan's cryptographic accelerators. Finally, anticipating future cryptographic challenges, the thesis explores the integration of Post-Quantum Cryptography (PQC) algorithms, including the development of OpenSSL providers for NIST-standardized PQC schemes like ML-KEM. This demonstrates the feasibility and benefits of custom Instruction Set Extensions (e.g., B and V extensions) for accelerating PQC operations on RISC-V platforms, showing speedups of up to 1.66x for ML-KEM encapsulation and 1.78x for decapsulation compared to software implementations.

In summary, this research advances the state-of-the-art in secure RISC-V System-on-Chip design by providing a comprehensive, integrated solution for cryptographic offloading, system integrity, and quantum readiness. The contributions establish a foundation for developing highly secure, high-performance, and resilient embedded systems, offering critical insights for both academic research and industrial applications.

Acknowledgements

Alla mia Mamma e al mio Papà.

Ad Auri e Leo.

List of Publications

1. M. Ciani, E. Parisi, **A. Musa**, F. Barchi, A. Bartolini, A. Kulmala, R. Psiakis, A. Garofalo, A. Acquaviva, and D. Rossi. *"Unleashing OpenTitan's Potential: a Silicon-Ready Embedded Secure Element for Root of Trust and Cryptographic Offloading."* **ACM Transactions on Embedded Computing Systems (TECS)**, vol. 24, no. 5, Art. 67, September 2025. doi: 10.1145/3690823

2. **A. Musa**, E. Parisi, L. Barbierato, E. Patti, A. Acquaviva, and F. Barchi. *"Integrating OpenTitan as a Security Controller for Cryptographic Tasks in RISC-V SoCs."* In **ITASEC/SERICS**, CEUR Workshop Proceedings, 2025. URL: https://api.semanticscholar.org/CorpusID:279077465

3. **A. Musa**, F. Volante, E. Parisi, L. Barbierato, E. Patti, A. Bartolini, A. Acquaviva, and F. Barchi. *"TitanSSL: Towards Accelerating OpenSSL in a Full RISC-V Architecture Using OpenTitan Root-of-Trust."* In **Computer Safety, Reliability, and Security (SAFECOMP 2024)**, pp. 169–183, Springer, 2024. doi: 10.1007/978-3-031-68606-1_11

4. E. Parisi, **A. Musa**, S. Manoni, M. Ciani, D. Rossi, F. Barchi, A. Bartolini, and A. Acquaviva. *"TitanCFI: Toward Enforcing Control-Flow Integrity in the Root-of-Trust."* arXiv: 2401.02567, 2024.

5. E. Parisi, **A. Musa**, M. Ciani, F. Barchi, D. Rossi, A. Bartolini, and A. Acquaviva. *"Assessing the Performance of OpenTitan as Cryptographic Accelerator in Secure Open-Hardware System-on-Chips."* In **Proceedings of the 21st ACM International Conference on Computing Frontiers (CF 2024)**, pp. 172–179, 2024. doi: 10.1145/3649153.3649213

6. **A. Musa**, E. Parisi, L. Barbierato, A. Acquaviva, and F. Barchi. *"End-to-End Integration of OpenTitan Security Features in a Pure RISC-V SoC."* In **20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2024)**, pp. 1–4, 2024. doi: 10.1109/SMACD61181.2024.10745397

Contents

List of Figures

List of Tables

Acronyms

ABI Application Binary Interface.

AES Advanced Encryption Standard.

AP Application Processor.

API Application Programming Interface.

CFI Control-Flow Integrity.

DMA Direct Memory Access.

DRAM Dynamic Random-Access Memory.

DTS Device Tree Source.

HMAC Hash-based Message Authentication Code.

IOCTL Input/Output Control.

IoT Internet of Things.

IP Intellectual Property.

ISA Instruction Set Architecture.

LLC Last-Level Cache.

ML-DSA Module-Lattice-Based Digital Signature Standard.

ML-KEM Module-Lattice-Based Key-Encapsulation Mechanism Standard.

MMU Memory Management Unit.

NIST National Institute of Standards and Technology.

OS Operating System.

OTBN OpenTitan Big Number.

PMP Physical Memory Protection.

PQC Post-Quantum Cryptography.

RoT Root of Trust.

RSA Rivest–Shamir–Adleman.

RTL Register-Transfer Level.

SC Security Controller.

SHA Secure Hash Algorithms.

SoC System-on-Chip.

TCDM Tightly Coupled Data Memory.

TEE Trusted Execution Environments.

TL-UL TileLink Uncached Lightweight.

TPM Trusted Platform Modules.

Chapter 1

Introduction and Context

1.1 The Evolving Landscape of Embedded System Security

The extensive deployment of embedded systems and Internet of Things (IoT) devices across critical infrastructures, ranging from industrial control and healthcare to autonomous vehicles and smart cities, has elevated security from a peripheral issue to a fundamental design requirement [1–4]. This escalating reliance on embedded devices for sensitive operations, often involving personal data or critical control functions, necessitates robust end-to-end protection, especially for resource-limited microcontroller units and their communication networks [5]. These systems frequently operate in potentially hostile or unmonitored environments, rendering them susceptible to a sophisticated and ever-evolving array of threats [6, 7].

Modern System-on-Chip (SoC) designs, characterized by their increasing complexity and heterogeneity, inherently present a significantly expanded attack surface compared to traditional computing architectures [8, 9]. The intricate interplay of diverse components, Intellectual Property (IP) blocks from various vendors, and complex software stacks creates numerous potential entry points for malicious actors. These challenges manifest across multiple layers:

- *Software Vulnerabilities*: Exploits targeting software vulnerabilities remain a primary concern. These include classic vulnerabilities such as buffer overflows, format string bugs, and code injection attacks, which allow attackers to compromise Operating Systems (OSs) or applications to gain unauthorized privileges or execute arbitrary code [7, 10]. The firmware itself, which forms the operational core of many embedded devices, often exhibits vulnerabilities that can be exploited remotely [10]. A particular challenge arises in embedded systems utilizing complex OSs like Linux, where studies indicate that many firmware images omit crucial attack mitigations commonly found in desktop or server environments, leaving these devices significantly more vulnerable to memory corruption and control-flow hijacking attacks [11, 12].

- *Hardware-Based Attacks*: These attacks exploit physical characteristics, design flaws, or manufacturing-time vulnerabilities within the hardware itself. Examples include side-channel analysis (e.g., power analysis, electromagnetic emissions) to extract sensitive cryptographic keys or data, fault injection (e.g., voltage glitches, clock glitches, laser attacks) to disrupt device operation or bypass security mechanisms, and the insertion of malicious hardware Trojans during the manufacturing or supply chain process [7, 13, 14]. The diverse and often globalized supply chain for chip components further amplifies the risk of introducing undetectable vulnerabilities or backdoors [3]. Physical access to IoT devices facilitates these attacks, enabling sophisticated techniques like JTAG exploitation or memory dumping[7, 11].

- *Communication Insecurities*: The growing interconnectedness of embedded systems, both internally (e.g., between chiplets or IP blocks within an SoC) and externally (e.g., via wireless or wired networks), introduces new vectors for attack. Communication between various system components, typically over high-speed interfaces, may be susceptible to unauthorized monitoring, data manipulation, replay attacks, and denial-of-service attacks if not properly secured [3, 15]. Furthermore, the limited computational and energy resources of many IoT devices hinder the implementation of robust authentication and encryption mechanisms, making device authentication a weak point in the overall security framework [16].

The adoption of complex OSs like Linux in embedded systems, while offering unparalleled flexibility and advanced functionalities, simultaneously expands the attack surface and elevates security requirements [15, 17]. The Linux kernel, responsible for managing hardware resources and enforcing system policies, becomes a critical component in the security chain [18]. Without rigorous security-by-design approaches and robust integration of security features into the kernel and its drivers, embedded Linux systems can become targets for large-scale cyber-attacks due to insufficient security procedures, infrequent updates, and challenges in managing hardware-software interactions securely [10, 18].

To counteract these pervasive threats, various existing hardware security solutions have been developed, including Trusted Platform Modules (TPM), Trusted Execution Environments (TEE), and hardware Root of Trust (RoT)[19, 20]. TPMs, for instance, are designed to provide a secure environment for cryptographic operations, secure key storage, and platform attestation [21, 22]. However, traditional TPMs typically offer predefined functionalities, limiting their flexibility for custom security applications, and can be expensive, rendering them unsuitable for all resource-constrained IoT devices [22]. Similarly, TEEs, such as ARM TrustZone or Intel SGX, offer strong isolation for sensitive code and data within secure enclaves [23, 24]. Although powerful, TEEs focus on protecting specific enclaves rather than establishing a system-wide RoT, which means they might still be susceptible to side-channel or physical attacks if their isolation is not perfectly implemented or if the underlying secure boot process is compromised [24, 25]. These limitations underscore a continuous need for more adaptable, transparent, and comprehensive security frameworks that can be tailored to the diverse requirements of modern embedded systems.

Adding a significant layer of future complexity to this threat landscape is the emerging field of quantum computing. Future large-scale quantum computers, leveraging algorithms such as Shor's (for integer factorization) and Grover's (for search optimization), are projected to compromise the security of many currently prevalent public-key cryptographic algorithms, including Rivest–Shamir–Adleman (RSA), while also significantly reducing the effective security strength of symmetric ciphers like Advanced Encryption Standard (AES) [26–29]. This necessitates a proactive and urgent transition to Post-Quantum Cryptography (PQC), which comprises cryptographic schemes designed to be resilient against attacks from quantum computers [26, 27, 30]. However, integrating PQC into resource-constrained embedded and IoT devices presents considerable challenges due to the high computational complexity, larger key sizes, and increased memory requirements of these new algorithms, demanding optimized implementations and often hardware acceleration to meet performance, energy, and latency constraints [27, 31–33].

Rising embedded system criticality, sophisticated threats, constrained security mechanisms, and the potential impact of quantum computing underscore a complex and evolving security landscape. This environment demands innovative approaches to protect embedded devices throughout their lifecycle, from design and manufacturing to deployment and ongoing operation.

1.2 Open-Hardware and RISC-V as Foundations for Trust

In the contemporary landscape of embedded system security, a crucial paradigm shift is underway, moving towards open-hardware methodologies and transparent architectural designs to promote inherent security and integrity. The escalating complexity of modern SoC designs, coupled with the imperative to defend against increasingly sophisticated hardware and software threats, has profoundly underscored the need for verifiable trust [34]. Open-source hardware addresses this challenge by providing transparency, allowing extensive public auditing and review of designs that would otherwise remain inaccessible if proprietary [35]. This inherent openness dismantles the "black-box" nature of traditional hardware components, thereby facilitating formal validation of security-critical properties, supporting tailored modifications to enhance resilience against attacks, and permitting low-level instrumentation for precise security monitoring [35, 36]. Such transparency is not merely an academic ideal but a practical necessity for building and sustaining trust throughout the entire lifecycle of computing systems, particularly given the increasing complexity of integrated circuits [34, 36].

The RISC-V Instruction Set Architecture (ISA) stands as a pivotal force within this growing open-hardware movement, rapidly establishing itself as a fundamental cornerstone for constructing trustworthy and secure computing platforms [35]. Its open-source, modular, and highly flexible nature has fueled its accelerated adoption across an expansive range of applications, spanning from everyday IoT devices to demanding high-performance computing environments [35]. Unlike proprietary ISAs, RISC-V offers a distinct advantage by enabling robust security guarantees through the direct implementation of novel extensions and mechanisms at the hardware level [37]. This high level of flexibility not only supports academic research and industrial collaboration in processor design and validation, but also enables the broader community to collaboratively develop and continuously improve reliable security mechanisms through public review and iterative enhancements [38, 39]. The global RISC-V community is actively engaged in pioneering security solutions designed to protect the integrity and confidentiality of sensitive information processed on RISC-V based devices [38].

Within this evolving RISC-V ecosystem, the concept of a "Root of Trust" has emerged as a fundamental element for ensuring system security. A hardware RoT is a perpetually trusted component or set of components within a system that acts as the immutable foundation for all security operations. It serves as the initial, unfalsifiable source of trust, guaranteeing the integrity and authenticity of following boot stages, firmware, and software. This foundational element is critical for establishing a secure boot process, enabling remote attestation, and securely managing cryptographic keys [19, 22]. Without a reliable RoT, the entire chain of trust can be compromised, rendering all subsequent security measures vulnerable [19].

OpenTitan epitomizes this commitment to open-source hardware security by offering a transparent and verifiable silicon RoT [40]. Designed as the first truly open-source silicon RoT, OpenTitan's objective is to provide a secure, flexible, and transparent foundation for building trustworthy hardware systems through its freely available and meticulously engineered architecture [41]. Its core purpose is to significantly enhance trust and transparency within the semiconductor industry, thereby enhancing the security of hardware platforms deployed in critical, security-sensitive scenarios [41]. These capabilities are enabled by a secure microcontroller, hardware cryptographic accelerators, and attestation mechanisms, which together allow sensitive operations to be executed in a hardened and fully verifiable environment. Since OpenTitan is designed to operate independently, integrating it into broader SoC designs requires additional extensions and interfaces to preserve its security guarantees. This strong commitment to openness and verifiability positions OpenTitan, along with other RISC-V-based open-hardware initiatives, as a key foundation for the development of secure and trustworthy embedded systems. However, the full realization of their potential in terms of both security and performance is intrinsically dependent on their effective integration with a robust and well-designed software stack.

1.3 Bridging the Gap: Software-Hardware Co-Design for Security

Embedded systems are becoming increasingly complex. At the same time, they face a wide range of threats. These challenges demand a shift toward solutions that tightly combine software and hardware to ensure effective security. While advanced hardware features, such as cryptographic accelerators and RoT, provide powerful security primitives, their full potential can only be realized through a robust and cohesive software stack [18]. This critical interface, often referred to as software-hardware co-design, is essential to bridge the conceptual and functional gap between hardware capabilities and their practical application in real-world scenarios. The absence of such integration can lead to underutilized security features, performance bottlenecks, and an increased attack surface, ultimately compromising the overall system's integrity [38]. Effective co-design guarantees that security is not simply an afterthought, but is fundamentally integrated into the system architecture from the outset. This approach enables software to utilize hardware-level protections both efficiently and securely [18].

However, making these sophisticated hardware accelerators accessible and efficient for applications presents significant challenges. Dedicated security hardware, while offering superior performance and protection against various attacks, often remains isolated or difficult to access for higher-level software. One significant obstacle is the development of standardized and efficient software interfaces that can translate complex hardware operations into manageable and secure Application Programming Interfaces (APIs) for developers [10]. Without such interfaces, developers may rely on custom, often insecure, solutions or neglect hardware acceleration entirely, opting for less efficient software implementations [10]. This problem is compounded in heterogeneous embedded systems, where different hardware components might require distinct integration approaches, leading to fragmentation and increased development overhead [42].

Furthermore, exposing hardware functionalities to software applications, particularly in multi-layered operating environments like Linux, requires careful consideration of privilege separation, memory management, and data transfer efficiency [10]. The overhead associated with data movement between the CPU, memory, and specialized accelerators can negate the performance benefits of the hardware itself [43]. For instance, CPU-managed transfers for cryptographic operations can saturate memory buses, thereby limiting the effective utilization of accelerators [43]. Developers face the challenge of designing kernel drivers and user-space libraries that efficiently orchestrate data flow to and from accelerators, manage their state, and ensure that sensitive information remains protected within trusted boundaries [18]. The goal is to provide a seamless and consistent interface for developers, allowing them to utilize hardware security features transparently, without requiring deep knowledge of the underlying hardware intricacies [10]. This approach aims for end-to-end cryptographic enablement, ensuring that the enhanced security features provided by hardware can be fully utilized at the application level in the Linux user space, often through standard cryptographic mechanisms like OpenSSL [18].

The necessity for tightly integrated hardware and software solutions is therefore paramount to leverage the full spectrum of security features effectively, ensuring that hardware advancements translate into tangible, deployable security benefits for embedded systems. This integration must overcome challenges related to accessibility, efficiency, and the secure exposure of hardware capabilities to applications, forming a critical area of focus in modern embedded system security research.

1.4 Overview of Research Contributions and Thesis scope

This thesis addresses the critical need for enhanced security and efficiency in modern embedded systems, particularly in the context of open-hardware platforms and the RISC-V ecosystem. The increasing deployment of embedded systems across critical infrastructures and their growing role in sensitive operations amplifies the necessity for robust security solutions [11, 44]. Their inherently complex and heterogeneous nature expands the attack surface, making them vulnerable to sophisticated software, hardware, and communication-based threats [38, 44]. Open-hardware platforms and the RISC-V ecosystem have emerged as critical enablers of trust, offering transparency, verifiability, and flexibility in processor design [38, 45]. However, fully realizing advanced hardware security features, such as those provided by silicon RoT like OpenTitan, requires robust software-hardware co-design. Without this integration, capabilities remain underutilized, and performance bottlenecks persist [46].

The primary focus of this research centers on enabling and optimizing cryptographic services and establishing a robust RoT environment through the development of a comprehensive software stack, referred to as "TitanSSL". This framework is engineered to seamlessly bridge the gap between advanced hardware security features, such as those provided by a silicon RoT like OpenTitan, and their efficient utilization by Linux-based applications [18]. This includes not only accelerating cryptographic operations but also enabling critical RoT functionalities like secure boot, secure key storage, and memory protection mechanisms within the system. The necessity for tightly integrated software and hardware solutions to effectively leverage security features is well-established, as the absence of such integration can lead to underutilized capabilities and performance overheads [18]. TitanSSL delivers this end-to-end cryptographic enablement and hardware acceleration by seamlessly integrating with the Linux OS and leveraging cryptographic frameworks like OpenSSL, allowing applications to securely and efficiently perform operations within the embedded environment. Challenges in making hardware accelerators both accessible and efficient for software applications are addressed directly, aiming to provide a standardized and consistent interface for developers[47].

Building upon this foundational work, the development and rigorous evaluation of the TitanSSL framework exposed inherent architectural bottlenecks in existing hardware/software interaction models, particularly in data movement strategies that can limit the effective utilization of cryptographic accelerators. These findings motivated architectural enhancements aimed at maximizing cryptographic throughput. The research therefore investigates optimized data transfer mechanisms and memory architectures to offload transfers from the main CPU, reduce latency, and provide efficient local memory access, significantly improving the performance and efficiency of cryptographic workloads.

Furthermore, this work introduces integral mechanisms to strengthen system integrity, acknowledging the persistent and evolving threat landscape that embedded systems face. While traditional attacks often involve injecting malicious code, modern adversaries increasingly exploit sophisticated code-reuse techniques, such as Return-Oriented Programming [48–50]. These attacks subvert program execution by chaining together legitimate code snippets already present in the system's memory, making them particularly insidious and challenging to detect with conventional security measures. In resource-constrained embedded environments, where extensive software-based defenses may be impractical or introduce unacceptable performance overheads, robust runtime protection against such control-flow hijacking is paramount. Therefore, this research investigates Control-Flow Integrity (CFI) as a crucial defensive measure. Specifically, by leveraging a strong hardware-anchored RoT, such as OpenTitan, this work develops novel approaches, exemplified by contributions like "TitanCFI", to enforce CFI. This methodology aims to provide comprehensive runtime protection against unauthorized control transfers, ensuring that program execution adheres to its intended flow, all while striving for minimal performance overhead and maximum resilience against advanced persistent threats [39, 51].

Finally, looking towards the future of embedded security, this thesis includes a crucial exploration into PQC. Recognizing the imminent threat posed by quantum computers to current cryptographic standards, the secure deployment of PQC schemes in resource-constrained embedded and IoT devices has become a critical safeguard [21, 52]. These new algorithms, however, typically present significantly higher computational complexity and memory requirements compared to their classical counterparts, demanding highly optimized implementations and robust hardware acceleration to meet performance, energy, and latency constraints inherent to embedded systems [21, 52].

This research addresses these challenges by investigating the feasibility and performance of PQC schemes on an accelerated RISC-V platform with custom ISA Extensions[53]. This includes enabling PQC through standardized OpenSSL interfaces, acknowledging that platforms such as OpenTitan are also actively engaged in implementing National Institute of Standards and Technology (NIST) PQC standards alongside classical cryptography [21, 52]. A key contribution is the comparative analysis of leading NIST-standardized PQC schemes[52, 53]. This comparison specifically evaluates implementations integrated within OpenSSL against highly optimized versions that exploit advanced hardware capabilities through cus-

tom ISA extensions[53]. This forward-looking investigation explores how optimized software strategies, by leveraging hardware acceleration and custom ISA extensions, can efficiently support the integration of quantum-resistant algorithms, preparing the developed ecosystem to face the cryptographic challenges of the post-quantum era. Collectively, these contributions demonstrate how robust software-hardware co-design can establish a secure and performant open-hardware platform for embedded systems.

1.5 Thesis Structure

This thesis is structured to guide the reader through the foundational concepts, proposed solutions, architectural enhancements, and forward-looking considerations in securing embedded systems using open-hardware principles.

- Chapter 1 - Introduction and Context: This chapter sets the stage by introducing the evolving landscape of embedded system security, highlighting the escalating threats and the growing need for robust protection. It then presents open-hardware and RISC-V as foundational elements for trust, discusses the necessity of software-hardware co-design for security, and provides an overview of the research contributions and thesis scope.

- Chapter 2 - Embedded System Security Foundations and Robust System Design: This chapter delves into the fundamental architectural components and background security concepts essential to this research. It begins by providing an overview of the underlying cryptographic algorithms employed in modern embedded systems. Subsequently, it describes the OpenTitan and RISC-V ecosystems for secure computing, details advanced security concepts such as CFI and PQC, and outlines the software stack requirements for hardware security modules, including the Linux Kernel and OpenSSL framework.

- Chapter 3 – TitanSSL: A Comprehensive Software Stack for End-to-End Cryptographic Offloading to OpenTitan: This chapter introduces TitanSSL, a specialized software stack designed to enable and optimize cryptographic services, and to integrate OpenTitan's RoT capabilities. It covers the motivation and design principles behind TitanSSL, details its architecture and data flow, and explains its implementation aspects, including the OpenSSL interfaces and Linux kernel driver, before presenting its performance evaluation and analysis.

- Chapter 4 – Architectural Enhancements for Optimized Cryptographic Performance: This chapter focuses on addressing the performance limitations highlighted by the development and evaluation of TitanSSL. It explores a co-design approach to implement architectural enhancements, primarily targeting data movement bottlenecks, and evaluates their significant impact on cryptographic throughput and efficiency.

- Chapter 5 – Enhancing System Integrity and Forward-Looking Cryptography: This chapter explores two critical areas for future-proofing embedded system security. It first details methods for strengthening system integrity through CFI, presenting a RoT-based approach that leverages OpenTitan's features for runtime protection against control-flow hijacking attacks with minimal overhead. Subsequently, it investigates the challenges and opportunities of integrating PQC on accelerated RISC-V-based platforms, discussing the quantum threat, standardization efforts, and early explorations into leveraging custom ISA extensions for PQC algorithms.
- Chapter 6 – Conclusions: This final chapter summarizes the key contributions of the research, discusses its broader impact and significance, and outlines potential future research directions stemming from this work.

Chapter 2

Embedded System Security Foundations and Robust System Design

This chapter explores the fundamental pillars of embedded system security, tracing the evolution from theoretical cryptographic principles to the hardware architectures that enforce them, and finally to the software layers that orchestrate and manage these protections. It begins with the cryptographic algorithms forming the mathematical backbone of digital security, including an overview of Post-Quantum Cryptography (PQC) algorithms. Then examines resilient hardware architectures, focusing on the RISC-V platform and the OpenTitan Root of Trust (RoT) as foundational elements. Furthermore, it explores advanced security paradigms, including robust Control-Flow Integrity (CFI) mechanisms and the critical need for PQC resilience, which are essential for safeguarding modern embedded systems. Building upon this foundation, the discussion turns to the software domain, highlighting the central role of the Linux kernel in managing hardware security features and coordinating cryptographic resources. Finally, it highlights user-space libraries and frameworks that interface with these hardware protections, illustrating how embedded systems achieve resilience and trustworthiness through the integration of hardware and software.

2.1 Foundational Cryptographic Principles

This foundational section establishes the essential cryptographic algorithms that support modern secure communications and data protection. It systematically explores the well-established conventional primitives that have historically secured digital interactions, detailing their core mechanisms and critical applications. These algorithms, having undergone extensive cryptanalysis and widespread deployment, remain indispensable for ensuring confidentiality, integrity, and authenticity in current systems [38, 54]. Despite their proven robustness against classical attacks, the advent of quantum computing necessitates a forward-looking perspective, leading

to the introduction of PQC [21]. This emerging field represents a crucial evolution, driven by the potential for quantum computers to compromise existing cryptographic schemes, thus demanding new cryptographic paradigms [21, 32, 53]. Grasping these core principles, whether well-established or emerging, is essential for comprehending the full scope of security mechanisms, modern hardware security approaches, and the challenges discussed in this thesis. It highlights not only what has secured our digital world but also what is required to maintain that security in the face of evolving computational threats.

2.1.1 Core Conventional Cryptographic Algorithms

Conventional cryptography includes a set of algorithms that have been rigorously studied, formally analyzed, and widely implemented. These algorithms ensure the core security properties of confidentiality, integrity, authentication, and non-repudiation in digital data [38, 55]. They are also considered robust against all known classical computational attacks, establishing the current global standard for cybersecurity. Architectures like OpenTitan RoT are specifically designed with hardware cryptographic accelerators to efficiently support and leverage these core functions, forming the foundation of a secure computing environment [46].

To fulfill these fundamental security functions, conventional cryptography employs various types of algorithms, each meticulously designed for specific objectives and operations [54, 55]. Symmetric encryption, for instance, guarantees data confidentiality through the use of a single secret key [38]. Hashing functions, on the other hand, ensure message integrity and authenticity by generating unique digital fingerprints [38]. Finally, public-key cryptography enables secure exchanges and identity verification in untrusted environments, utilizing pairs of public and private keys [38]. Each of these categories plays a distinctive and complementary role in the architecture of digital security[55]. This is because they address different aspects of digital security, and their combined use creates a robust, multi-layered defense system that is more efficient and secure than relying on any single type alone [38, 55].

No single cryptographic primitive can provide all desired security properties across every scenario [55]. The main approaches can be summarized as follows:

- *Symmetric-key cryptography* excels at providing confidentiality for large amounts of data due to its high speed and efficiency [38]. In such schemes, once a shared secret key is established, the underlying symmetric primitives enable rapid encryption and decryption of bulk data[56].
- *Asymmetric-key cryptography*, on the other hand, is generally slower but is crucial for establishing secure channels and verifying identities. It solves the critical problem of secure key exchange (allowing two parties to agree on a secret key without prior shared knowledge) and provides authentication and non-repudiation through digital signatures [38, 56].

- *Cryptographic Hashing* are primarily used to ensure data integrity and authenticity. They create a unique, fixed-size fingerprint of a message, allowing recipients to verify that the data has not been tampered with and often confirming the sender's identity when combined with public-key signatures[38].

To achieve efficient and comprehensive security, these distinct cryptographic types are often combined in hybrid systems [54, 55]. For instance:

1. Public-key cryptography is first used to securely exchange a session key, which is a symmetric key [38]. This solves the challenge of distributing symmetric keys over insecure channels.

2. Once the symmetric session key is securely established, symmetric-key cryptography is then used for the high-speed encryption and decryption of the actual bulk data, leveraging its efficiency [56].

3. Throughout this process, hashing functions can be used to generate message digests, which are then signed using public-key cryptography (creating digital signatures) to ensure the integrity and authenticity of the data being transmitted [38]. This ensures that the message hasn't been altered and confirms who sent it.

Therefore, symmetric encryption provides fast confidentiality, public-key cryptography manages key distribution and digital signatures, and hashing ensures data integrity. Together, they form a synergistic security architecture, with each type compensating for the limitations of the others to provide a robust and practical solution for digital security [38, 54]. The following subsections explore selected algorithms from each of the core categories, examining their mechanisms and significance.

AES

The Advanced Encryption Standard (AES) stands as the global benchmark for symmetric-key block ciphers. AES operates by transforming fixed-size blocks of plaintext into ciphertext using a single, shared secret key for both encryption and decryption. Its design, based on the Substitution-Permutation Network structure, involves a series of substitution (non-linear byte transformations) and permutation (bit shuffling) layers, interspersed with key additions. AES supports key lengths of 128, 192, and 256 bits, offering varying levels of security commensurate with the sensitivity of the data protected. It can be operated in various modes (e.g., CBC, CTR, and GCM) which define how successive plaintext blocks are combined and encrypted to ensure confidentiality and, in some cases, integrity. Its widespread adoption spans numerous applications, from securing network communications (e.g., in TLS/SSL) to encrypting data at rest on storage devices. In resource-constrained embedded systems, efficient hardware implementations of AES are vital for enabling high-throughput secure communication and ensuring data confidentiality without significantly impacting overall system performance or power consumption.

SHA

Secure Hash Algorithms (SHA) comprise a family of cryptographic functions designed to generate a fixed-size output, known as a message digest or hash value, from an arbitrary-length input message. The critical cryptographic properties of SHA functions include:

- *Preimage Resistance*: It is computationally infeasible to find the original input message given only its hash value.
- *Second Preimage Resistance*: Given an input message and its hash, it is computationally infeasible to find a different input message that produces the same hash value.
- *Collision Resistance*: It is computationally infeasible to find any two different input messages that produce the same hash value.

SHA algorithms, such as SHA-256 and SHA-3, are indispensable for ensuring data integrity (detecting unauthorized modifications), authenticating messages, constructing digital signatures, and deriving cryptographic keys. They also serve as the underlying primitive for the Hash-based Message Authentication Code (HMAC) construction. HMAC combines a cryptographic hash function, like SHA, with a secret key to provide both message integrity and authenticity. This allows two parties that share a secret key to verify that a message has not been tampered with and that it originates from a trusted source. Within embedded systems, hash functions are fundamental for verifying firmware integrity, validating code execution, protecting stored data, and establishing trust in cryptographic protocols, such as those employed during secure boot processes.

RSA

Rivest–Shamir–Adleman (RSA) is a cornerstone algorithm in public-key cryptography, a paradigm that revolutionizes secure communication by employing distinct but mathematically linked keys: a public key for encryption (or signature verification) and a corresponding private key for decryption (or signature generation). The security of RSA is predicated on the assumed computational difficulty of factoring the product of two large prime numbers. This asymmetry allows for secure communication over untrusted channels without the necessity of prior shared secrets and provides the foundation for digital signatures, which offer strong authentication, integrity, and non-repudiation. While generally more computationally intensive than symmetric algorithms, RSA is indispensable for critical functions such as secure key exchange, digital certificate validation (e.g., in Public Key Infrastructure), and establishing secure connections (e.g., during the handshake phase of TLS/SSL protocols).

2.1.2 The Post-Quantum Paradigms

The rapid theoretical and practical advancements in quantum computing present a profound and impending threat to the foundations of modern digital security and cryptographic systems. As a result, many widely deployed cryptographic schemes, including public-key, symmetric-key, and hash-based algorithms, may become vulnerable once sufficiently powerful and stable quantum computers are realized [32, 57]. This existential threat necessitates the development and swift adoption of PQC, a new class of cryptographic primitives specifically designed to remain secure against attacks mounted by both classical and future quantum computers [53]. In particular, the transition to PQC is considered critical and urgent, especially due to the risk of "harvest now, decrypt later" attacks where currently encrypted communications could be stored and later deciphered by quantum adversaries [53]. Consequently, the National Institute of Standards and Technology (NIST) initiated a public standardization process to define the next generation of quantum-resistant cryptographic primitives [53].

At the core of this paradigm shift lies the emergence of powerful quantum algorithms that directly threaten the mathematical foundations of classical cryptography. Two notable examples illustrate the extent of this threat:

- *Shor's Algorithm*: This algorithm can efficiently factor large integers and compute discrete logarithms, tasks that are computationally intractable for classical computers [21]. Its implications are severe, as it directly undermines widely deployed public-key cryptosystems such as RSA and Elliptic Curve Cryptography, which rely on the hardness of these problems. If Shor's algorithm becomes practical, the security of digital certificates, secure communication protocols (e.g., TLS/SSL), and key exchange mechanisms would be effectively compromised.
- *Grover's Algorithm*: While it does not completely break symmetric-key cryptography, it can significantly speed up brute-force attacks on symmetric ciphers (e.g., AES) and hash functions, effectively reducing their security strength by half [21]. For instance, a 128-bit symmetric key would only offer 64 bits of effective security against a quantum adversary.

This growing quantum threat has accelerated global research efforts toward PQC, aiming to design new cryptographic primitives that can resist both classical and quantum attacks.

Post-Quantum Cryptography Landscape

In response to the emerging quantum threat, PQC schemes have been developed to provide security against both classical and quantum adversaries. The NIST has been at the forefront of a multi-year standardization process to identify and select a suite of PQC algorithms suitable for widespread adoption. This process has led to the emergence of several promising families of PQC algorithms:

- *Lattice-based Cryptography*: These schemes derive their security from the perceived hardness of certain problems in mathematical lattices. Examples include Kyber (for key encapsulation) and Dilithium (for digital signatures). They are generally efficient and offer strong security guarantees[58].

- *Hash-based Cryptography*: These rely on cryptographic hash functions. They provide strong, well-understood security but often have larger key sizes or stateful properties, making them more suitable for specific applications like firmware signing. XMSS and SPHINCS+ are prominent examples.

- *Code-based Cryptography*: Based on error-correcting codes, the classic example is the McEliece cryptosystem. While offering long-standing security, they typically suffer from very large public keys.

- *Multivariate Polynomial Cryptography*: These schemes use systems of multivariate polynomial equations over finite fields. Rainbow is an example, though some algorithms in this category have faced attacks.

- *Isogeny-based Cryptography*: Based on walks in graphs of elliptic curve isogenies, these offer relatively small key sizes but can be computationally intensive. SIKE was a candidate in the NIST process but has recently been broken.

These PQC schemes rely on mathematical problems that are presumed to be hard for even quantum computers to solve, such as those found in lattice-based, hash-based, and multivariate cryptography [31, 59]. Among these, Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) and Module-Lattice-Based Digital Signature Standard (ML-DSA) have emerged as leading candidates in the post-quantum landscape and are the focus of further discussion in this work.

ML-KEM: A Post-Quantum Key Encapsulation Mechanism

ML-KEM, formerly known as Kyber, has been selected by the NIST as a primary algorithm for standardization as a Post-Quantum Key Encapsulation Mechanism [52]. ML-KEM is a lattice-based cryptographic algorithm, whose security relies on the computational hardness of problems over lattices (discrete, regularly repeating arrangements of points in space), specifically the conjectured difficulty of the Learning With Errors problem and its ring variant. As a Key Encapsulation Mechanism, ML-KEM enables two parties to securely establish a shared secret key over an insecure communication channel, providing forward secrecy and resistance against adversaries with quantum capabilities. Its selection by NIST highlights its critical role in securing future digital communications against the emerging quantum threat.

To efficiently implement these security guarantees, ML-KEM relies on a set of core cryptographic primitives. The Number Theoretic Transform is used for fast polynomial multiplication in modular domains, which is a fundamental operation in lattice-based schemes. Keccak serves as the basis for both hashing and pseudo-random function operations, instantiating SHAKE-128 and SHAKE-256, which are

crucial for key derivation and other symmetric operations within ML-KEM. Additionally, Keccak is employed for polynomial sampling, a critical step in generating secure lattice elements. Together, these primitives ensure both the security and correct operation of the ML-KEM scheme.

ML-DSA: A Post-Quantum Digital Signature Scheme

ML-DSA, is a primary algorithm selected by NIST for standardization as a Post-Quantum Digital Signature Algorithm [53]. Based on the CRYSTALS-Dilithium scheme, it provides secure digital message authentication through three core operations: KeyGen, Sign, and Verify. Similar to ML-KEM, ML-DSA is a lattice-based cryptographic algorithm, whose security relies on the computational hardness of problems within mathematical lattices [53]. Minor modifications from the original Dilithium scheme target performance improvements while maintaining algorithmic robustness.

ML-DSA shares several computational primitives with ML-KEM, most notably the Number Theoretic Transform and the Keccak permutation. Number Theoretic Transform accelerates polynomial multiplications in modular arithmetic, representing a key efficiency bottleneck in Sign and Verify operations[53]. Keccak is used extensively for hashing, key derivation, challenge generation, and polynomial sampling, forming an essential component of the scheme's cryptographic operations [53]. These primitives underpin both the efficiency and quantum resistance of ML-DSA, supporting secure and performant digital signature generation and verification.

2.2 OpenTitan and RISC-V Ecosystem for Secure Computing

The contemporary landscape of embedded system security is characterized by a rapid evolution, with increasing emphasis on open-source hardware and transparent architectures as foundational elements for building trustworthy platforms. Within this context, the synergy between RISC-V and OpenTitan emerges as a pivotal solution for addressing the complex security challenges inherent in modern System-on-Chip (SoC).

2.2.1 The RISC-V Architecture: Principles and Security Extensions

The RISC-V Instruction Set Architecture (ISA) has established itself as a significant driving force within the open-hardware movement, serving as a fundamental cornerstone for constructing secure and reliable computing platforms[38, 45]. Its open, modular, and highly flexible design has driven its accelerated adoption across a broad spectrum of applications, ranging from low-power Internet of Things (IoT)

devices to demanding high-performance computing environments [38, 60]. In contrast to proprietary ISAs, the inherent openness of RISC-V facilitates extensive public auditing and scrutiny of its architectural design, thereby fostering the development of secure platforms and enhancing hardware resilience against cyber threats [38, 58]. This transparency creates unparalleled opportunities for innovation in chip security solutions [45].

At the heart of RISC-V's design lie its fundamental principles and advantages, which include openness, modularity, and a collaborative ecosystem. Together, these characteristics enable the development of secure, verifiable, and highly adaptable processor architectures.

- *Transparency and Verifiability*: The open nature of RISC-V allows for comprehensive public review and detailed analysis of its design, overcoming the "black-box" limitations often associated with proprietary hardware. This intrinsic transparency is crucial for formal validation of security-critical properties and supports bespoke modifications to bolster resilience against attacks [38]. This openness directly enables the construction of reliable security mechanisms [38].

- *Modularity and Extensibility*: RISC-V's modularity enables developers to integrate custom instructions or functional blocks directly into the base ISA, allowing for precise adaptation of the processor to specific application requirements [58]. This capability is particularly beneficial for embedding advanced security features or cryptographic accelerators directly within the hardware design [38].

- *Community and Collaboration*: The vibrant global community of researchers and developers actively engaged with the RISC-V ecosystem promotes continuous collaboration and innovation [38]. This collective effort leads to the rapid development and iterative enhancement of security mechanisms through public review and continuous refinement [38].

These foundational principles naturally inform the design of RISC-V's layered privilege model, which provides differentiated protection across the software stack [38]. The model includes Machine (M), Supervisor (S), and User (U) modes, each enforcing distinct access rights and responsibilities.

- *Machine Mode*: This is the highest privilege level and is considered the most trusted. Code executing in M-mode possesses low-level access to the machine's implementation details and is responsible for managing the secure execution environment within RISC-V [38]. It is the only mandatory privilege level for RISC-V hardware platforms [38].

- *Supervisor Mode*: Designed for Operating System (OS) environments, such as Linux. S-mode provides isolation between a supervisor-level OS and the Supervisor Execution Environment [38].

- *User Mode*: Representing the lowest privilege level, U-mode is where user application code is executed. It is restricted in its capabilities, preventing direct access to hardware resources or other protected memory regions, thereby ensuring application isolation [38].

This robust privileged architecture allows for flexible routing of traps to different privilege levels, contributing to overall system security [38].

Complementing the privilege model, Physical Memory Protection (PMP) provides fine-grained control over memory access. This critical security mechanism within RISC-V utilizes per-hart machine-mode control registers to define read, write, and execute privileges for specific memory regions [38, 61]. PMP checks are enforced on all memory accesses when the processor operates in S or U modes, effectively granting permissions to these modes and, conversely, revoking permissions from M-mode where necessary, thereby enforcing strict memory isolation [38]. PMP violations are precisely trapped by the processor [38]. This mechanism is fundamental for preventing unauthorized access to sensitive data or critical code [38]. RISC-V PMP allows privileged software to define memory access permissions for distinct memory regions [38].

Beyond foundational protection, the RISC-V ISA can be extended to directly incorporate hardware cryptographic functions, significantly reducing the attack surface, enhancing resistance to side-channel attacks, and strengthening control-flow integrity [38]. The RISC-V Cryptographic Extensions Task Group is actively developing proposals for both scalar and entropy source instructions, which encompass bit manipulation, scalar AES, SHA, SM3, and SM4 acceleration, as well as True Random Number Generation entropy source interfaces [38, 58]. These extensions are designed to deliver substantial performance improvements for cryptographic algorithms [38]. For instance, a simple RISC-V instruction set extension can achieve significant acceleration for lightweight cryptographic primitives [38].

Projects such as OpenTitan build upon these open and extensible hardware principles to showcase practical applications of RISC-V security.

2.2.2 OpenTitan: A Silicon Root-of-Trust for RISC-V platforms

Within the growing RISC-V ecosystem, the concept of a "Root of Trust" plays a central role. A hardware RoT functions as a perpetually trusted component or set of components within a system, serving as the immutable foundation for all security operations [44, 62]. It is the initial, unfalsifiable source of trust that guarantees the integrity and authenticity of subsequent boot stages, firmware, and software [44, 62]. This foundational element is indispensable for establishing a secure boot process, enabling remote attestation, and securely managing cryptographic keys [44, 62]. The absence of a reliable RoT can compromise the entire chain of trust, rendering all subsequent security measures vulnerable [44].

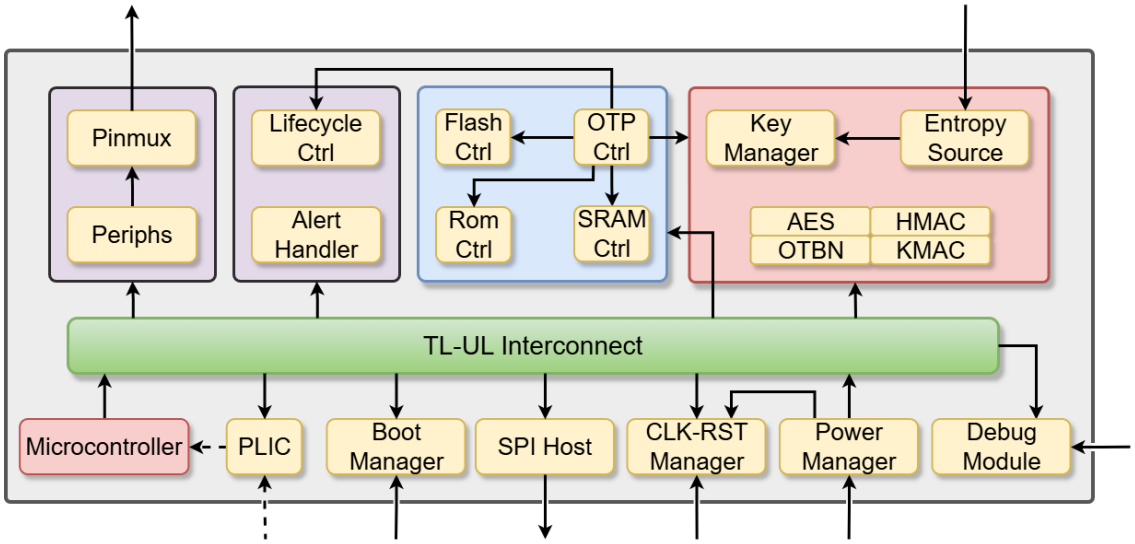

Figure 2.1: Block diagram of the `top_earlgrey` module, representing the core digital architecture of the Earl Grey SoC.

OpenTitan represents a pioneering open-source silicon RoT project, dedicated to providing a transparent, verifiable, and high-quality foundation for secure computing, particularly within RISC-V-based systems. Its core objective is to significantly enhance trust and transparency throughout the semiconductor industry, thereby fortifying the security of hardware platforms deployed in a wide array of security-sensitive applications [62]. The project's emphasis on open-source design allows for rigorous public auditing and verification, which is paramount for establishing enduring trust in computing systems [63]. OpenTitan represents a major milestone as the first open-source silicon project to achieve commercial availability, featuring validated chips based on the OpenTitan Earl Grey discrete RoT (`top_earlgrey`) [63]. The block diagram of the Earl Grey module's core digital architecture is shown in Figure 2.1.

The platform establishes a robust chain of trust, beginning with a secure boot process that ensures only authenticated software is executed [63]. This multi-stage authentication process involves cryptographic checks to validate each component, effectively safeguarding against unauthorized modifications or executions [63]. The secure boot begins with the ROM stage, responsible for initial device setup and authenticating the subsequent `ROM_EXT` stage using public keys. The `ROM_EXT`, stored in Flash, further authenticates the next boot stage, manages key transitions, and performs critical boot services. This hierarchical, multi-stage approach ensures that each component is cryptographically verified by its predecessor, leveraging hardware features and cryptographic checks to protect against tampering. OpenTitan can also extend its secure boot flow to the application processor, utilizing mechanisms like mailboxes and interrupts for further configurations.

Fundamentals of OpenTitan Architecture

The hardware architecture of OpenTitan is meticulously engineered to provide a comprehensive and robust security solution. It integrates a secure microcontroller, dedicated on-board private memory, an array of cryptographic hardware accelerators, and essential communication I/O interfaces. Several fundamental components underpin OpenTitan's design, enabling secure processing, storage, and communication within the platform.

Ibex Processor: The central processing unit within OpenTitan is the Ibex processor, a 32-bit RISC-V CPU, that wraps its data and instruction memory interfaces to TileLink Uncached Lightweight (TL-UL). This standard represents the minimal conformance level of the TileLink protocol, a lightweight, uncached interconnect designed for simple memory-mapped communication between masters and peripherals. The Ibex processor itself is optimized for embedded and power-efficient applications, and is designed to support the execution of RoT functions [64]. This microcontroller handles security control tasks and the logical and coordination operations necessary for RoT management. The Ibex core implements the RISC-V ISAs I (base integer operations), M (standard integer multiplication and division), and C (16-bit compressed instruction encoding) [64]. For enhanced reliability and fault protection, OpenTitan can instantiate the Ibex core in a dual-core lockstep configuration.

Cryptographic Hardware Accelerators: OpenTitan includes a dedicated set of cryptographic accelerators designed to safeguard data confidentiality, integrity, and authenticity. These accelerators efficiently execute compute-intensive security primitives, offloading these tasks from the Ibex core and ensuring high performance for security operations.

- *Hash Functions*: Dedicated accelerators for hash functions such as SHA-256 and SHA-3 are integrated, playing a crucial role in data integrity verification. For message authentication, OpenTitan's HMAC accelerator can be configured for both SHA-256 and HMAC message digests. Its operation involves a memory-mapped FIFO where, for SHA-256, the software pushes 64-byte data blocks; once filled, the accelerator processes the block in 80 cycles, updating its internal state. Software setup includes selecting SHA or HMAC mode and specifying endianness, with the secret key loaded for HMAC mode, followed by a continuous loop of feeding data blocks once the FIFO is empty.

- *Symmetric Encryption*: Accelerators for symmetric encryption algorithms like AES are fundamental for data confidentiality. OpenTitan's AES accelerator is designed to optimize storage by generating round keys in real-time, thereby reducing memory requirements and speeding up key-switching operations. To mitigate side-channel leakage, the device erases key and data registers with pseudo-random data upon reset or software activation. The accelerator's front-end features a 16-byte FIFO for plaintext input from the software driver. In

automatic mode, upon loading, it immediately encrypts the content and produces the ciphertext block based on the chosen operational mode. A specific OpenTitan implementation utilizing a unmasked AES accelerator with Cipher Block Chaining (CBC) mode and a 256-bit key can encrypt a 16-byte plaintext block in 16 cycles. Similar to the HMAC accelerator, software configures control registers, loads the key and initialization vector, and then enters a processing loop, awaiting completion of acceleration operations before retrieving ciphertext and feeding new plaintext blocks.

- *Asymmetric Cryptography*: The OpenTitan Big Number (OTBN) accelerator significantly enhances the performance of mathematical operations critical to asymmetric cryptography, such as RSA and Elliptic Curve Cryptography. These algorithms are essential for digital signatures and key exchange protocols. OTBN is a co-processor specialized for broad integer arithmetic, featuring a 32-bit control path and a 256-bit data path. It provides extensive control flow support, including conditional branches, unconditional jumps, and hardware loops, alongside custom instructions for common public-key cryptosystem functions like pseudo-module addition or partial-word multiplication and accumulation. RSA's security relies on the computational difficulty of factoring large prime numbers. Accelerating asymmetric encryption on OTBN involves implementing the algorithm with its instruction set, then loading the program binary and data from the Ibex secure microcontroller into OTBN's reserved memories, and finally activating computation, with OpenTitan monitoring a control register for completion

Memory and Secure Storage: OpenTitan's memory domain includes a 128KB scratchpad SRAM and embedded flash memory. These memory units are equipped with dedicated controllers that include Error Correcting Code and data-address scrambling mechanisms to enhance both security and reliability. Crucially, essential cryptographic and scrambling keys are generated by accelerators and securely stored within a tamper-proof One-Time Programmable eFuse memory. This write-once memory ensures that keys cannot be altered once programmed, serving as an immutable anchor of trust and playing a critical role in the correct and secure operation of OpenTitan. The One-Time Programmable controller, which manages these partitions, can also store secret data in a scrambled form and allows for read and write lockable data, with integrity and storage consistency checks.

Key Manager: An internal key manager is responsible for overseeing hardware identities and root keys, thereby safeguarding critical assets from software-based threats.

Hardware Life Cycle Controller: This component dynamically manages the availability of features based on the device's operational state. This ensures that certain functionalities, such as debugging or testing capabilities, are restricted in production environments, thereby reinforcing system security throughout its lifecycle.

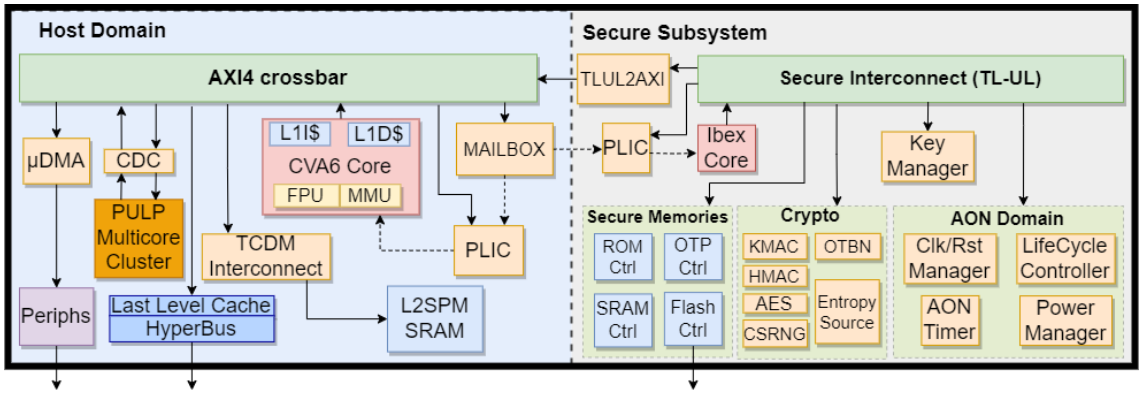

Figure 2.2: Diagram illustrating how OpenTitan is integrated as a secure subsystem within the overall architecture.

Security Peripherals: OpenTitan includes a peripheral subsystem supporting UART, USB, I2C, SPI, and GPIO for communication with the external world. It also features various protections against physical attacks, including power analysis countermeasures and tamper detection circuits.

2.2.3 Integration of OpenTitan within RISC-V System-on-Chips

A SoC integrates multiple electronic components onto a single microchip, typically including CPU cores, memory, and various input/output interfaces and specialized accelerators. The secure and efficient integration of a hardware RoT, such as OpenTitan, is a critical aspect for modern embedded systems, particularly in architectures designed for high-assurance applications. This section details the specific design of an existing RISC-V based SoC that incorporates OpenTitan, illustrating its function as a dedicated security controller and outlining the established communication mechanisms with the host processor[63]. The SoC is shown in Figure 2.2.

This specific SoC design meticulously integrates OpenTitan to establish a robust and isolated security subsystem, working in conjunction with a host RISC-V application processor. This architecture ensures that critical security functions are handled by a dedicated, trusted component, significantly bolstering the overall security posture of the embedded system. The core principle of this integration leverages OpenTitan as a specialized security coprocessor that offloads sensitive cryptographic and attestation operations from the main RISC-V host core. This approach not only enhances the performance of compute-intensive cryptographic tasks but also profoundly improves system security, flexibility, and portability by isolating the security domain.

The SoC design integrating OpenTitan features three primary components:

- An *Application Subsystem*: It leverages the high-performance CVA6 RISC-V application-class core to execute complex workloads and support secure operations. This is a powerful open-source 64-bit RISC-V processor, specifically

engineered for running complex OSs like Linux. Its architecture is built around a six-stage, in-order pipeline, and it offers comprehensive support for the RV64GC instruction set, which includes standard integer and floating-point arithmetic, along with extensions for multiplication, division, atomic operations, and compressed instructions. Furthermore, it integrates hypervisor capabilities, enabling advanced virtualization. The CVA6 features an advanced 64-bit virtual memory subsystem, complete with a Memory Management Unit (MMU) that supports various privilege levels to effectively manage memory access and enforce robust isolation. Designed for high instruction throughput while maintaining timing predictability, the core also includes private, split L1 instruction and data caches, an intelligent branch prediction unit, and a reorderable commit stage, ensuring its efficient deployment across ASIC and FPGA platforms. This host core is primarily responsible for executing the system's main application logic and managing peripheral interactions.

- A *Programmable Multi-Core Accelerator*: This component consists of a cluster of specialized cores, optimized to accelerate computationally intensive tasks such as Digital Signal Processing or machine learning, thereby enhancing the overall system capabilities.
- *OpenTitan*: Integrated within the same silicon die, OpenTitan acts as the hardware SoC, providing a physically isolated and hardened environment for critical security operations.

Beyond the architectural overview, three key aspects define how OpenTitan operates within this SoC: its role as a dedicated Security Controller (SC), the memory architecture that supports both performance and security, and the mechanisms that govern its communication with the host processor.

Dedicated Security Controller and Isolation: OpenTitan functions as an autonomous security element, primarily dedicated to RoT operations, while the host RISC-V core handles general application software. This high-level separation provides the context for the subsequent memory architecture and communication mechanisms. This clear and physical separation of responsibilities is fundamental for containing threats. Any compromise of the host software or OS is prevented from affecting the critical security functions and secrets managed by OpenTitan. This independence means OpenTitan's isolation mechanisms operate irrespective of the host processor's OS or specific architectural features, making it a highly versatile and adaptable dedicated SC.

Memory Architecture and Organization: The SoC features a hierarchical memory architecture that balances performance, security, and consistency between the host processor and OpenTitan.

- *CVA6 Core Memory*: The CVA6 application processor features a hierarchical cache system, including configurable first-level (L1) instruction and data caches, with an option for a second-level cache.

- *OpenTitan Internal Memory*: OpenTitan integrates on-board private memory. This includes a secure One-Time Programmable eFuse memory used for permanently storing lifecycle states and cryptographic seeds. The internal flash memory was reduced to 64 KB, configured with two 32-KB banks to support A-B partitioning for firmware updates.
- *System-Level Memory Hierarchy*: The SoC's memory hierarchy features a Last-Level Cache (LLC) shared between the main CVA6 core and the OpenTitan microcontroller (Ibex), enabling both processors to access frequently used data efficiently. Accesses to external Dynamic Random-Access Memory (DRAM) are mediated through a 512 KB last-level cache managed by an open-source HyperRAM controller, which coordinates memory transactions and accelerates data retrieval. To ensure memory consistency, the CVA6 flushes its L1 caches before initiating operations on OpenTitan. This allows OpenTitan to write results directly to memory in a zero-copy manner, guaranteeing that the host processor always accesses the most recent data and preventing conflicts in shared-memory scenarios. These measures preserve data integrity even when the host and OpenTitan operate asynchronously.

Communication Interfaces and Security Boundaries: The communication between the host RISC-V application processor and OpenTitan is a critical aspect of the SoC design, carefully engineered to maintain security while ensuring efficient data exchange.

- *System Interconnect*: The internal system-level interconnect within OpenTitan, and often the primary interface for its integration into larger SoCs, is based on the TL-UL protocol.
- *Host-to-OpenTitan Bridging*: Given that the application domain of the host platform (e.g., a CVA6 running Linux) employs a different interconnect protocol, such as AXI4, a crucial component for successful integration is the TileLink-AXI4 bridge (e.g., TLUL2AXI). This custom bridge is essential to reconcile protocol differences, allowing OpenTitan to interface with the host's memory map. The bridge connects as a slave to the TL-UL interconnect within OpenTitan and exposes AXI4 master interfaces to the outside, enabling OpenTitan to control the primary interconnect through a dedicated master port and gain visibility over the entire memory map of the SoC.
- *Mailbox Communication for Secure Interaction*: To preserve the host core from tampering with the content of the secure subsystem, OpenTitan connects to the rest of the SoC primarily through its master port and does not make any slave ports accessible. Communication between the main processor and the secure component is deliberately restricted to message exchange via a mailbox system facilitated by interrupts.
  - *Mailbox Structure*: The mailbox itself is typically integrated into the SoC domain, accessible by both the application processor and OpenTitan, but crucially positioned outside OpenTitan's protected perimeter.

It comprises a set of general-purpose memory-mapped registers intended for data sharing.

- *Doorbell and Completion Registers*: This system utilizes dedicated registers, such as `Doorbell` and `Completion` registers, to signal communication. The CVA6 host processor initiates communication by writing a command and metadata to the mailbox, following SCMI standards, and then asserting the `Doorbell` register, which triggers an interrupt to the Ibex core within OpenTitan. OpenTitan then reads and decodes the command, retrieving any necessary payload from external memory.
- *Interrupt Handling*: OpenTitan's Programmable Interrupt Controller is extended to accommodate the new interrupt line from the mailbox. This external interrupt serves as the exclusive mechanism for the host to directly interact with OpenTitan, whose Platform Level Interrupt Controller (PLIC) configuration ensures such interrupts are masked unless specifically configured for secure interaction.

This sophisticated integration strategy ensures that OpenTitan can effectively serve as a robust hardware RoT and security controller, providing physically isolated cryptographic acceleration and secure services while maintaining efficient yet controlled interaction with the host RISC-V application processor within the same SoC. By emphasizing openness and verifiable design, this integration positions OpenTitan as an indispensable foundation for architecting the next generation of secure and trusted embedded systems.

2.3 Advanced Security Concepts in Embedded Systems

Embedded systems, increasingly interconnected and deployed in critical applications, face a sophisticated and evolving threat landscape. While foundational security measures remain paramount, advanced techniques are essential to protect these systems from both current and emerging threats. This section explores two such critical advanced security concepts: CFI and PQC. CFI addresses runtime attacks by ensuring that program execution follows its intended path, guarding against common exploitation techniques. Concurrently, PQC tackles the future threat posed by quantum computers, which are poised to break many of today's widely used cryptographic algorithms. Together, these advanced concepts are vital for building resilient and future-proof embedded systems.

2.3.1 Control-Flow Integrity Threats and Hardware Solutions

In the complex landscape of embedded system security, CFI stands as a critical defense mechanism against a prevalent class of software attacks [65]. These attacks, often exploiting memory corruption vulnerabilities, aim to subvert the intended execution path of a program, leading to arbitrary code execution or system compromise

[66, 67]. This section delves into the fundamental principles that underpin CFI, examining how it enforces legitimate program behavior, and subsequently explores the primary attack vectors, such as code-reuse attacks, that CFI is designed to mitigate. Understanding these concepts is essential for appreciating the necessity and design considerations of robust security solutions in modern embedded environments.

Principles of Control-Flow Integrity: The core principle of CFI involves dynamically or statically analyzing the program's intended control flow and then enforcing this pre-computed model during runtime. This enforcement can be implemented through various architectural extensions:

- *ISA Extensions*: Techniques that introduce custom opcodes for security checks during program execution. These typically have minimal runtime overhead but require specific toolchains and modifications to the processor pipeline [68].
- *Hardware Monitors*: Custom-designed Intellectual Propertys (IPs) integrated with the processor pipeline to observe control-flow instructions and implement security features like shadow stacks or jump tables directly in silicon [68]. These are transparent to the executed software but lack flexibility in dynamically updating the CFI enforcement policy.
- *Programmable Co-processors*: These rely on a dedicated (semi-)programmable core that enables the implementation of security policies in software, operating in parallel with the main core [69]. This category is particularly suited for integration within hardware RoT, enabling flexible yet isolated enforcement mechanisms [69].

The Threat of Control-Flow Hijacking: Control-flow hijacking attacks are a significant threat to embedded systems, particularly those that lack robust memory protection mechanisms or frequently interact with untrusted inputs [70]. These attacks aim to divert the program's execution path to attacker-controlled code or to legitimate code snippets (gadgets) in an unintended sequence.

- *Return-Oriented Programming*: A prevalent code-reuse attack where an attacker chains together small sequences of existing machine instructions, called "gadgets," which typically end with a RET instruction [67]. By manipulating the stack to point to these gadgets, the attacker can execute arbitrary code by effectively "programming" the target's existing code [67]. Return-Oriented Programming attacks are particularly challenging to detect because they do not inject new code but rather reuse existing legitimate code, making them difficult to distinguish from normal program execution [67].
- *Jump-Oriented Programming*: Similar to Return-Oriented Programming, Jump-Oriented Programming leverages existing code gadgets, but these gadgets typically end with `JMP` or `CALL` instructions rather than `RET`. Attackers construct their malicious logic by orchestrating a sequence of these jump-oriented gadgets, again subverting the legitimate control flow [68, 71].

These attacks are especially dangerous in embedded systems due to several factors:

- *Resource Constraints*: Many embedded systems operate with limited memory and processing power [37]. Software-only CFI solutions often introduce substantial runtime overhead and may rely on static policies, posing a significant challenge for real-time embedded systems [65].
- *Lack of Operating System Protection*: Simpler embedded systems may not run a full-fledged OS, thus lacking the advanced memory protection and process isolation features available in larger systems [38]. Furthermore, even when present, security metadata stored in virtual memory pages protected by it may lack authenticity guarantees if the OS is breached [72].
- *Firmware Vulnerabilities*: Firmware, which is often less scrutinized than application software, can contain exploitable vulnerabilities that attackers can use to gain initial control and then launch CFI attacks[11].

Therefore, CFI is crucial for embedded systems, providing a robust layer of defense against these sophisticated code-reuse attacks by ensuring that program execution remains predictable and within the bounds of legitimate behavior [73]. Leveraging dedicated security hardware, such as a RoT like OpenTitan, can significantly enhance the effectiveness of CFI implementations [38, 46]. The integration of control-flow verification within a trusted hardware boundary provides stronger isolation. The RoT, operating as an autonomous security element, can:

- *Strengthen the Application*: A RoT can monitor control-flow events and validate them against an embedded policy database [73]. Deviations can trigger immediate actions, thereby preventing malicious code execution.
- *Ensure Metadata Integrity*: The RoT can utilize its cryptographic primitives to authenticate CFI metadata, storing sensitive information in a protected silicon region isolated from the untrusted host [37, 46]. This ensures that compromise of the application or OS cannot bypass CFI enforcement.
- *Act as a Trusted Arbiter*: A programmable co-processor within the RoT can execute custom CFI policies in software, taking advantage of the RoT's tamper-proof storage and cryptographic accelerators [74].
- *Support Adaptive Mechanisms*: While fixed hardware monitors may lack flexibility in updating CFI policies dynamically, RoT-based solutions can introduce adaptability through firmware-configurable rules, allowing policies to be updated post-deployment [69]. Advanced approaches, such as machine learning-based CFI implemented within the secure perimeter of a RoT, can enable adaptive and field-updatable policies, supporting both real-time and offline enforcement modes.

In summary, hardware-backed CFI mechanisms are essential in embedded systems, where software-only defenses can be vulnerable to sophisticated attacks and resource constraints often preclude complex security features [11]. For instance, ISA extensions typically introduce negligible runtime cost, while dedicated hardware monitors ensure continuous verification of control flow with limited impact on execution speed. When combined with a RoT, hardware-based CFI solutions further enhance system resilience by isolating critical control-flow metadata and enforcement logic from the main software domain [74]. This isolation makes them more resistant to compromise, even in the presence of OS breaches or malicious firmware. Although some hardware monitors implement fixed policies that cannot be modified after deployment, RoT-based or programmable hardware solutions can retain adaptability through firmware-configurable security rules [68]. Altogether, these characteristics position hardware-assisted and RoT-integrated CFI as a cornerstone for building secure and trustworthy embedded platforms [38, 62].

2.3.2 Hardware Accelerated Post-Quantum Cryptography

The advent of quantum computing poses a significant threat to current public-key cryptographic schemes, necessitating a proactive shift towards PQC [32, 52, 53]. The NIST has led a global effort to standardize quantum-resistant algorithms, selecting initial PQC standards [52]. These include lattice-based schemes like the Key Encapsulation Mechanism CRYSTALS-Kyber (standardized as gls mlkem) and the Digital Signature Algorithm CRYSTALS-Dilithium (standardized as ML-DSA) [53, 59]. These two schemes currently represent the cornerstone of standardized lattice-based PQC algorithms. Implementing these new PQC algorithms, especially in resource-constrained embedded systems, presents several challenges related to performance and efficiency [75]. PQC schemes often involve complex mathematical operations that are computationally intensive when executed purely in software [21, 53]. Overcoming these limitations typically involves a combination of optimized software implementations and dedicated hardware acceleration [21, 59]. This has motivated extensive research exploring multiple techniques of acceleration to meet the performance demands of high-volume cryptographic operations [53].

General Approaches to PQC Acceleration in Embedded Systems: To address these challenges, several strategies are employed to accelerate PQC operations within embedded systems, ranging from software optimizations to dedicated hardware support.

- *Instruction Set Extensions*: Custom instruction set extensions can be added to general-purpose processors (like RISC-V cores) to accelerate specific, computationally intensive operations common across PQC schemes [58]. These extensions preserve flexibility as the software utilizing them remains updateable, allowing for adaptability to new schemes or slight changes in existing ones [52]. This approach combines the flexibility of software with the efficiency of hardware, often targeting mathematical operations useful for both PQC and

pre-quantum cryptography[52]. Examples include extensions for number theoretic transforms, polynomial arithmetic, and efficient modular arithmetic [31, 76].

- *Dedicated Hardware Accelerators*: For maximum performance and energy efficiency, specific hardware accelerators can be designed for entire PQC algorithms or their critical components. These can be integrated as co-processors alongside the main CPU [57, 59]. While offering high speedups, pure hardware solutions might require new silicon manufacturing runs for any changes or updates in PQC schemes [52].
- *Hardware/Software Co-design*: This balanced approach combines the flexibility of software with the efficiency of hardware. It often involves using flexible hardware accelerators for computationally intensive operations while leaving the overall algorithm control to software running on the main CPU [57, 77]. This method has proven essential for enabling secure PQC adoption in embedded platforms, especially for critical functionalities like secure boot and firmware updates [77].

These design principles have guided a wide range of practical implementations and research efforts over the past few years. A variety of approaches have been explored to accelerate PQC algorithms such as ML-KEM (Kyber) and ML-DSA (Dilithium) [53]. Most early efforts focused on pure hardware accelerators implemented on FPGA or ASIC platforms, often using high-level synthesis techniques to optimize number theoretic transform, sampling, and hashing operations [27, 31, 59]. Later, hardware/software co-designs emerged, leveraging reconfigurable architectures and embedded processors to balance performance and flexibility [21, 26, 29]. More recent work has explored unified and parameterized accelerators that support multiple PQC schemes, along with integrated countermeasures against side-channel attacks such as masking and fault-tolerant computation [26, 29, 30]. While these designs demonstrate impressive throughput and energy efficiency, they often rely on dedicated hardware, limiting portability and programmability. As a result, attention has shifted toward architectural-level and ISA-based acceleration, exploiting RISC-V instruction set extensions as a means to integrate PQC performance enhancements directly within general-purpose processors [77].

RISC-V ISA Extensions for PQC Acceleration

The RISC-V architecture, with its modularity and extensibility, offers a compelling platform for integrating PQC acceleration directly into the processor's ISA. Rather than relying solely on custom hardware, leveraging standard or custom RISC-V ISA extensions can provide substantial performance benefits while maintaining flexibility. For instance, specific bit manipulation (B) and vector (V) extensions, such as those integrated into the Sargantana RV64GBV core, have been shown to significantly accelerate ML-KEM and ML-DSA operations [53].

The Sargantana RV64GBV core, a single-issue in-order RISC-V processor, serves as a notable experimental platform for evaluating PQC algorithms in realistic embedded contexts [78]. It features a 128-bit wide SIMD unit that supports RISC-V Vector Extension version 1.0. This enables efficient execution of vector operations critical for lattice-based cryptography [78]. Furthermore, the bit manipulation (B) extension significantly enhances performance by reducing multiple instructions into single, more efficient operations. This is particularly beneficial for cryptographic primitives like Keccak, which constitutes a substantial portion (over 50%) of the ML-KEM and ML-DSA execution cycles[78].

Empirical evaluations on the Sargantana core have shown that exploiting the standard RISC-V extensions (B and V) can considerably accelerate post-quantum algorithms such as ML-KEM and ML-DSA compared to baseline implementations. Hand-optimized scalar and vector code demonstrated notable performance advantages, with manual bit-manipulation and vectorization strategies outperforming compiler-generated routines [78]. These results highlight that the integration of standard RISC-V extensions, when manually exploited, can bridge the performance gap for PQC algorithms on general-purpose processors, reducing the need for purely custom hardware accelerators [78]. The Sargantana core, therefore, provides a flexible and powerful environment to test the overheads and benefits of PQC algorithms in both optimized software and with hardware acceleration [78]. This proactive approach to PQC integration, combining versatile experimental platforms and software frameworks, is essential to future-proof embedded systems against the impending threat of quantum adversaries [57].

2.4 Software Stack for Hardware-Based Security in Embedded Linux Systems

Modern embedded systems are increasingly complex, often featuring powerful processors and running sophisticated OSs like Linux [56]. They are deployed in critical and interconnected environments, ranging from industrial IoT to advanced cyber-physical systems, and are no longer confined to isolated functions. This evolution demands robust security measures to protect sensitive data and operations [38]. In such complex landscapes, dedicated hardware security mechanisms, including RoTs modules and cryptographic accelerators, provide a strong foundation for secure key storage and cryptographic function offloading [46]. However, their full potential can only be realized through a well-designed and standardized software stack [56, 79].

This software stack bridges the gap between high-level application requirements and the low-level intricacies of hardware security modules or cryptographic accelerators. It encompasses multiple layers, from kernel drivers that manage direct hardware access to user-space cryptographic APIs that simplify secure application development [56]. By effectively managing these layers, the software stack ensures that the robust security provided by the hardware is fully leveraged by the OS and applications [37, 79].

A prominent example is the OpenSSL library, widely used in embedded Linux systems to interface with hardware security [80]. OpenSSL offers a comprehensive suite of cryptographic functions and can integrate with hardware accelerators to enhance the performance of security-critical operations, such as those in SSL/TLS protocols [56]. By leveraging dedicated hardware, embedded systems can achieve substantial speed gains in computationally intensive cryptographic tasks, improving both security and overall system efficiency [56].

The following sections detail the essential components and considerations for developing a comprehensive and performant software architecture that fully exploits hardware-based security in embedded Linux systems.

2.4.1 Kernel-Level Management of Hardware Security in Embedded Linux

The Linux kernel stands as the central arbiter of system security and resource management within embedded systems. It establishes foundational security mechanisms such as privilege separation, access control, and process isolation [38, 81]. These mechanisms define and enforce boundaries, ensuring that system components operate with restricted privileges and controlled access to resources such as memory and peripherals [38, 66]. The kernel mediates all interactions between user-space applications and physical devices, establishing a secure environment that also ensures the proper integration of hardware security modules and cryptographic accelerators.

The Role of Kernel Drivers in Hardware Integration

Kernel drivers are the indispensable software components that provide the controlled interface between user-space applications and the underlying hardware. They operate within the privileged kernel space, a protected execution mode that grants them direct access to hardware resources while shielding these resources from potentially malicious or faulty user-space code [82]. This privileged position allows drivers to manage the intricate details of hardware communication, handle interrupts, maintain synchronization, and control access to shared hardware resources [82, 83].

A fundamental aspect of hardware integration in embedded Linux is the Device Tree Source (DTS), which serves as a comprehensive data structure providing the Linux kernel with a detailed description of the hardware components present in the system [84]. This includes peripheral devices, memory regions, CPU cores, interrupt lines, and their specific configurations, such as addresses and frequencies [37, 84]. Drivers abstract away the complexity of physical devices, presenting a standardized interface to higher layers of the software stack. In the context of external cryptographic hardware, this abstraction is facilitated by the DTS, which promotes portability and reusability of drivers across different hardware platforms. Drivers rely on the information provided by the device tree to discover, configure, and operate the hardware they manage, ensuring correct interaction with the specific memory regions and registers assigned to a device [84]. This abstraction is often realized through

character devices, a common type of device file in Unix-like systems. User-space applications interact with these character devices as if they were regular files, using standard file operations such as `open()`, `close()`, `read()`, and `write()`. However, for more complex, device-specific commands, drivers typically expose functionality through the Input/Output Control (IOCTL) interfaces [85].

Device-Specific Control through IOCTL

In Linux-based embedded systems, user-space applications operate with limited privileges and cannot directly access specialized hardware components [86]. This separation between user-space and kernel-space, which contains the OS and device drivers, is fundamental for system stability and security [38]. While user-space can perform standard input and output operations, such as reading from or writing to files, executing non-standard hardware-specific commands requires a controlled interface.

The IOCTL mechanism provides this interface, allowing user-space applications to communicate device-specific instructions to kernel-space drivers [87]. Unlike generic system calls, which handle standard data streams, IOCTL is designed to convey customized commands and associated parameters to hardware devices. When an application invokes the `ioctl()` system call, it typically provides a file descriptor identifying the target device, a unique command code specifying the operation, and optionally a pointer to a data buffer containing input parameters or receiving output [87]. The kernel-space driver interprets the command code, performs any necessary memory management, including pinning memory pages and translating virtual addresses to physical addresses, and executes the operation on the hardware component [23]. Drivers also handle interrupts and synchronize access to shared hardware resources when needed. Upon completion, the driver collects any resulting data or status information and returns it to the calling application through the same IOCTL interface. This mediated process ensures that hardware access is controlled, validated, and safe, preventing unauthorized manipulation or accidental corruption of critical resources [87].

Additionally, IOCTL allows offloading of computationally intensive operations, such as cryptographic functions, to dedicated hardware accelerators, thereby improving system performance while maintaining a secure execution environment. Robust error handling in IOCTL calls ensures that any hardware or driver faults are correctly reported back to user-space applications, further enhancing reliability and safety.

2.4.2 User-Space Libraries and Frameworks for Security

Security in embedded systems extends beyond kernel-level management and direct hardware interaction, where a substantial portion of implementing security policies and cryptographic operations occurs in user-space through dedicated libraries and frameworks. These tools offer standardized interfaces for developers, abstracting the underlying complexity of cryptographic algorithms and facilitating integration with hardware security modules. They are crucial for building secure applications and ensuring that critical data is protected throughout its lifecycle.

OpenSSL: A Foundational Cryptographic Library

OpenSSL stands as one of the most widely adopted and robust open-source cryptographic libraries [80]. It is extensively utilized to implement security protocols such as SSL/TLS across a diverse range of applications and systems, including embedded devices [56].

Its architecture is primarily structured into two core components. The `libcrypto` is the fundamental library that provides a comprehensive suite of cryptographic functionalities. It encompasses implementations of symmetric algorithms, such as AES, asymmetric algorithms including RSA, hash functions such as SHA, key generation mechanisms, and routines for digital certificate management and large-number arithmetic [56]. The `libssl`, on the other hand, focuses on the implementation of SSL/TLS protocols, which are essential for establishing secure communications over untrusted networks. It relies heavily on `libcrypto` for all the underlying cryptographic operations required by these protocols [56]. In addition to these core libraries, OpenSSL provides a rich set of Application Programming Interfaces (APIs) for application developers, enabling seamless integration of cryptographic functions into custom software. Among these, the high-level EVP (Envelope) interface abstracts cryptographic operations such as symmetric encryption, hashing, and digital signatures, allowing developers to switch algorithms without changing the application logic. It also includes a command-line interface tool, which allows users to perform cryptographic operations, certificate management, and testing directly from the shell.

OpenSSL's versatility stems from its capacity to support not only software-based cryptographic algorithms but also to interface with dedicated cryptographic hardware. This capability enables the offloading of computationally intensive operations, which is particularly critical in embedded systems where CPU resources are often constrained and cryptographic performance is a key consideration. To integrate hardware accelerators and support custom cryptographic functionalities, OpenSSL has historically employed an extensible architecture based on Engines. More recently, with OpenSSL 3.0, it introduced a modern architecture centered around Providers. These extension mechanisms are vital for tailoring OpenSSL to specific hardware and security requirements in embedded contexts.

OpenSSL Engines: OpenSSL Engines are dynamic modules designed to extend the library's cryptographic capabilities. They allow seamless integration of specialized hardware, proprietary algorithms, and secure key management into the OpenSSL framework[18, 56]. Engines integrate with `libcrypto`, intercepting calls to standard OpenSSL functions. When an operation configured for an Engine is invoked, execution is redirected to the underlying hardware or custom implementation.

The primary applications of OpenSSL Engines include:

- *Hardware Acceleration*: Engines delegate cryptographic operations to specialized hardware modules such as GPUs or dedicated security controllers. This improves performance and reduces the processing load on the main CPU [38].
- *Custom Functionality*: Engines enable the integration of proprietary cryptographic algorithms or security mechanisms that are not part of the standard OpenSSL distribution [88].
- *Secure Key Management*: Engines can interface with hardware security modules to securely store and manage cryptographic keys, ensuring private keys never leave the protected hardware environment [38].

Technically, an Engine operates through a sequence of well-defined steps:

1. *Engine Registration*: Upon loading, the Engine's initialization routine (often a `BIND` function) registers the Engine. It defines a unique identifier, declares its capabilities, and registers the cryptographic algorithms it supports.
2. *Function Hooking*: For each supported algorithm, the Engine provides pointers to its custom implementation. The `libcrypto` maintains internal function pointers for cryptographic operations, and the Engine overrides them as needed, redirecting calls from the default software implementation to the Engine's optimized or hardware-accelerated functions [56].
3. *Operation Execution*: When an application invokes an OpenSSL API function (e.g., AES encryption or SHA-256 hashing), `libcrypto` checks if an active Engine has registered an override. If so, the Engine's Perform Operation function is called This function communicates with the underlying hardware accelerator or secure element, often through kernel-space drivers or dedicated hardware interfaces, to execute the task.
4. *Result Return*: After the operation is completed, the Engine collects the results and returns them to `libcrypto`, which passes them back to the calling user-space application.

This modular framework allows OpenSSL to offload computationally intensive operations to specialized hardware, achieving significant performance gains while maintaining a high level of security. By combining dynamic Engine loading, secure key handling, and hardware acceleration, Engines ensure that sensitive operations can be executed efficiently and safely, without exposing critical cryptographic material to the main CPU or user-space[56].

OpenSSL Providers: Introduced with OpenSSL 3.0, the Providers architecture represents a significant evolution beyond the Engine framework, offering enhanced modularity and flexibility in integrating algorithms and hardware accelerators. Providers encapsulate the implementations of cryptographic algorithms and services, providing a structured approach to extending OpenSSL and enabling more granular management of cryptographic functionalities.

Building upon this modular foundation, this modern design allows developers to integrate new algorithms, including PQC, and dynamically select between different implementations (e.g., software versus hardware-accelerated) at runtime [77]. This capability highlights the Providers' adaptability across a broad range of use cases, from general-purpose cryptography to specialized hardware-based acceleration. Providers therefore offer a cleaner interface and a more robust extension mechanism than Engines, while retaining the central concept of offloading operations to hardware.

The technical advantages of Providers, stemming from their modular design, include:

- *Modular Implementation*: Providers encapsulate specific algorithms and services, creating a clear separation of concerns. Different Providers can manage distinct sets of operations, such as symmetric ciphers, hashing algorithms, or dedicated hardware accelerators. This enhances maintainability and supports independent development of cryptographic modules.

- *Flexible Deployment and Selection*: Providers can be dynamically loaded or unloaded at runtime. OpenSSL can be configured to use specific Providers for particular algorithms, or it can automatically select the most suitable implementation (e.g., preferring a hardware-accelerated version over a software implementation) based on system capabilities and configuration [87]. This flexibility allows embedded systems to tailor their cryptographic backend to available resources and performance requirements.

- *Post-Quantum Cryptography Support*: Leveraging this modularity, Providers are particularly suited for integrating emerging cryptographic standards such as PQC. Libraries like OpenSSL are actively implementing PQC finalists, including CRYSTALS-Kyber, and CRYSTALS-Dilithium, within the Provider framework. This architecture facilitates adoption of new algorithms without extensive changes to the core library or existing applications, making OpenSSL more future-proof against quantum threats.

In essence, while Engines provided a way to extend OpenSSL, Providers offer a more elegant, modular, and forward-looking architecture. Their design not only simplifies integration and maintenance but also ensures smoother coexistence of classical and post-quantum algorithms within the same cryptographic framework. They are particularly effective in dynamic environments and for the seamless adoption of cutting-edge cryptography, including PQC, while maintaining efficient integration with hardware accelerators [86].

GlobalPlatform API and Secure Elements

Beyond the OpenSSL framework, other standardization efforts contribute to the software stack supporting hardware-based security. The GlobalPlatform specifications define standards for managing applications and security domains within secure elements and smart cards [89]. A secure element is a tamper-resistant hardware component designed to securely store sensitive data and execute cryptographic operations. Such components provide a protected environment for operations such as cryptographic processing, device authentication, and secure boot, thereby strengthening system trust and isolation mechanisms [63, 83]. GlobalPlatform APIs define standardized communication interfaces that allow applications to interact with these secure elements in a consistent and portable manner [87]. While not directly integrated into OpenSSL, the principles and interfaces established by GlobalPlatform are essential for building an interoperable and secure ecosystem around hardware security modules in embedded devices, complementing the role of cryptographic libraries and kernel-level drivers.

Together, these components, the robust Linux kernel, the extensible OpenSSL framework with its Engine and Provider mechanisms, and the standardized interfaces for secure elements, constitute a cohesive and layered software stack. This stack enables embedded systems to leverage dedicated cryptographic hardware effectively, ensuring that security-critical operations are executed efficiently and within trusted hardware boundaries, from low-level device interactions to high-level application services.

Chapter 3

TitanSSL: A Comprehensive Software Stack for End-to-End Cryptographic Offloading to OpenTitan

The growing complexity and critical importance of embedded systems call for robust security solutions that seamlessly integrate both hardware and software components. Open-source hardware initiatives such as OpenTitan provide a solid foundation for secure computing [40]. In this work, OpenTitan has been integrated into a RISC-V–based SoC, as described in Section 2.2, to extend its security capabilities to embedded platforms. However, the full potential of such architectures can only be unlocked through a well-designed software stack that efficiently exposes their cryptographic capabilities to applications. This chapter introduces TitanSSL, a comprehensive software framework developed to bridge this gap by enabling Linux-based applications to securely and efficiently offload cryptographic operations to OpenTitan's dedicated hardware accelerators. TitanSSL ensures that performance-critical cryptographic workloads benefit from hardware acceleration while maintaining the highest security standards guaranteed by a hardware RoT.

3.1 Motivation and Design Principles of TitanSSL

The increasing integration of advanced hardware security modules into embedded SoC platforms highlights a critical need for efficient software frameworks that can fully exploit these security features. OpenTitan, a leading open-source silicon RoT, provides a secure and isolated execution environment that includes dedicated cryptographic accelerators and protected memory for storing critical assets [87]. However, this architecture introduces a considerable area overhead, which makes it essential to take full advantage of OpenTitan's hardware capabilities, particularly its cryptographic acceleration, in order to justify the design cost [85].

TitanSSL is primarily motivated by the following research question: 'To what extent can the memory isolation and cryptographic acceleration features of an OpenTitan-based SoC, as described in Section 2.2, be leveraged to implement a secure backend for OpenSSL?' [87]. Existing solutions in the literature, such as EdgeLock Secure Enclave and Keystone, present several limitations that TitanSSL aims to address [90, 91]. EdgeLock Secure Enclave is a proprietary security subsystem integrated into NXP processors. It employs hardware-level isolation and cryptographic protections to ensure data integrity and confidentiality. Despite these advantages, its proprietary nature significantly restricts its adaptability. Replicating or porting its functionality to custom architectures is challenging, which limits its usefulness in open and flexible SoC designs. Keystone, on the other hand, is an open-source framework designed to create secure software environments on RISC-V platforms. It achieves physical memory isolation through RISC-V Physical Memory Protection (PMP). However, Keystone relies on a software-based Secure Monitor, which introduces performance overhead because enclaves share the same processor with other system components. This architectural choice can cause performance degradation in security-critical workloads that involve frequent context switches. Even though the GlobalPlatform specifications allow Trusted Execution Environments (TEE) and Rich Execution Environments to execute on physically separated hardware resources within an SoC, many Keystone-based designs reported in the literature run both environments on the same application processor, which compromises the intended separation [87].

TitanSSL seeks to overcome these limitations by offering a distinct approach that leverages OpenTitan's inherent hardware isolation and accelerators more effectively. A key advantage of this approach is that the isolation mechanisms are independent of any specific OS or architectural features of the host processor. This independence allows OpenTitan to be integrated with a variety of processors, enabling it to function as a dedicated Security Controller (SC). By leveraging its hardware cryptographic accelerators, OpenTitan can significantly improve the performance of cryptographic tasks while maintaining a high degree of security, flexibility, and portability.

The design principles of TitanSSL are centered on maximizing performance, ensuring robust security, and promoting usability through standardization:

- *Efficient Exposure of OpenTitan's Cryptographic Capabilities*: TitanSSL is conceived as a secure software stack dedicated to offloading cryptographic operations to OpenTitan. This ensures that the powerful hardware accelerators within OpenTitan are fully utilized.
- *Standardized Application-Facing Interface*: A core design principle is the use of OpenSSL as the primary application-facing interface. This choice ensures that developers can access OpenTitan's security features through a widely adopted and consistent API, simplifying integration into Linux-based applications.

- *Robust Security Guarantees*: TitanSSL leverages OpenTitan's fundamental security features, including memory isolation, secure boot, and RoT capabilities. It securely manages secrets and cryptographic keys by storing them within OpenTitan's private memory, preventing their exposure in main memory. The kernel driver component of TitanSSL is designed to mitigate threats related to Broken Access Control and Broken Authentication.
- *Modular and Extensible Architecture*: The TitanSSL framework is composed of an OpenSSL backend (implemented as an OpenSSL engine), a Linux kernel driver for communication, and custom firmware for OpenTitan's Security Controller. Communication between these components is facilitated by a custom Application Binary Interface (ABI), ensuring driver independence from specific cryptographic operations and promoting extensibility. The architecture also complies with GlobalPlatform specifications for secure execution environments.
- *Optimized Data Flow*: Early works highlighted communication overheads and inefficiencies caused by memory access latencies and the lack of transparent data movement mechanisms. TitanSSL's design addresses these issues by optimizing data transfers and providing seamless support for application-level virtual memory through the SC.

In essence, TitanSSL's motivation arises from the need to fully unlock the secure and performance-enhancing capabilities of OpenTitan within RISC-V SoCs, providing a secure, performant, and user-friendly cryptographic backend that outperforms software-only solutions and addresses the rigidities and overheads of alternative security frameworks.

3.2 OpenTitan Hardware Performance Characterization

Before integrating OpenTitan into a full SoC environment, a foundational step in understanding its capabilities for cryptographic offloading is to rigorously characterize its raw hardware performance. This initial assessment of OpenTitan's cryptographic accelerators, performed in isolation, is crucial for establishing a baseline for expected performance gains and identifying potential architectural bottlenecks. The aim is to thoroughly evaluate how effectively OpenTitan can execute cryptographic workloads using its dedicated hardware, thereby justifying its significant area overhead within an SoC design and maximizing its computational utility. This preliminary characterization serves to illuminate the intrinsic efficiency of OpenTitan's security features before considering the complexities of system-level integration. It is a critical first step in bridging the gap between advanced hardware features and their efficient utilization, ensuring that the powerful security primitives provided by OpenTitan are fully understood in isolation, prior to the challenges of software-hardware co-design within a larger system. A detailed description of OpenTitan's architecture and structure, as well as the SoC environment in which it was integrated, is provided in Section 2.2.

Table 3.1: Definition and purpose of the opcode and semantic labels used to annotate benchmark execution traces. Opcode labels describe the type of instruction executed by the Ibex core, while semantic labels capture the functional role of each instruction within the cryptographic operation, enabling fine-grained correlation between software behavior and hardware accelerator activity.

Type	IP	Label	Description
OpCode	All	ALU	Arithmetic and logical operations between data stored in registers.
		CTL	Control instructions: branch, jump, call, and return.
		Memory-RoT	Load/store instructions accessing the OpenTitan private scratchpad or the accelerator FIFO.
		Memory-RAM	Load/store instructions accessing the main system memory.
Semantic	HMAC	Config	Configure the accelerator in SHA-256 or HMAC mode, loading the secret key if required.
		Digest	Load data from memory and push it into the accelerator FIFO to prepare it for processing.
		Wait	Wait for the accelerator to be ready to accept additional data.
		Final	Pad the message, finalize digest computation, and copy the result back to memory.
	AES	Config	Initialize the accelerator, setting the mode of operation, key, and initialization vector.
		Cipher	Push the next data block into the accelerator while writing the previous block back to memory.
		Wait	Wait for the accelerator to be ready to accept additional data.

3.2.1 Evaluation Setup

The performance characterization focused on assessing the effectiveness of three key OpenTitan cryptographic accelerators, HMAC, AES, and the OTBN accelerator, across five critical cryptographic workloads: SHA-256, HMAC, AES-256, RSA-512, and RSA-1024. These primitives were chosen due to their central role in major cryptographic libraries and transport layer security protocols like TLS and DTLS [92]. The evaluation was conducted on a cycle-accurate Register-Transfer Level (RTL) simulator of the OpenTitan design, specifically targeting the `top_earlgrey` architecture, enabling full visibility into the execution of every instruction and hardware event.

Table 3.2: Instruction count and cycle breakdown for OpenTitan HMAC and AES accelerators when processing a 4 KiB payload. The table reports the number of ALU, control (CTL), and memory instructions accessing either the private scratchpad (RoT) or system DRAM, alongside the percentage of total cycles consumed by each type of operation.

	Operation	Instructions [#]					Cycles [%]				
		ALU	CTL	Memory		TOT	ALU	CTL	Memory		TOT
				RoT	DRAM				RoT	DRAM	
SHA-256	Config	5	–	6	–	11	0.02	–	0.10	–	0.12
	Digest	230	210	1090	1024	2554	0.92	0.68	21.06	73.24	95.90
	Wait	74	74	74	–	222	0.46	0.23	2.48	–	3.17
	Final	4	–	13	8	25	0.02	–	0.20	0.60	0.82
	TOT	**313**	**284**	**1183**	**1032**	**2812**	**1.42**	**0.91**	**23.84**	**73.83**	**32260**
HMAC	Config	5	–	23	–	28	0.02	–	0.40	–	0.42
	Digest	230	210	1090	1024	2554	0.91	0.67	20.89	72.64	95.11
	Wait	86	86	86	–	258	0.53	0.26	2.87	–	3.66
	Final	4	–	13	8	25	0.02	–	0.20	0.59	0.81
	TOT	**325**	**296**	**1212**	**1032**	**2865**	**1.48**	**0.94**	**24.36**	**73.23**	**32526**
AES-256	Config	28	9	64	–	101	0.05	0.01	0.54	–	0.61
	Cipher	1281	256	2306	2048	5891	4.60	0.39	18.89	72.42	96.30
	Wait	258	257	257	–	772	0.77	0.39	1.93	–	3.10
	TOT	**1567**	**522**	**2627**	**2048**	**6764**	**5.43**	**0.79**	**21.36**	**72.42**	**66461**

Table 3.3: Memory access cost statistics for OpenTitan SoC, measured in cycles. The table reports the minimum, maximum, and average cycle counts for accessing the private scratchpad (RoT) and the system DRAM.

Memory	Min.	Max.	Avg.
RoT	5.0	12.0	5.7
DRAM	23.0	27.8	23.4

To ensure an accurate measurement of OpenTitan's intrinsic capabilities, the benchmarks were developed in C and programmed directly on the accelerators, bypassing the official OpenTitan software stack's accelerator tests. This approach minimized potential overheads from unoptimized software and unnecessary memory accesses, allowing for a more precise understanding of the hardware's performance limits. Any situation where software needed to wait for the accelerator (such as when input FIFOs were full or when accelerators required reconfiguration) was specifically managed through polling to avoid additional latencies caused by inefficiencies in interrupt handling and wake-up delays.

The performance analysis combined information extracted from RTL execution traces produced by ModelSim and Verilator simulators with a hand-annotated disassembly of the benchmark software. Each assembly instruction was labeled with two dimensions: a semantic label and an opcode label. The type and meaning of both opcode and semantic labels are described in Table 3.1. Opcode labels indicate the nature of the operation (e.g., arithmetic, control flow, or memory access), helping to identify the prevalence of computational versus memory-bound instruc-

Table 3.4: Memory access bandwidth for OpenTitan SoC accelerators, measured in cycles per byte. The table reports the number of read/write accesses, total cycles, and resulting cycles per byte for HMAC and AES accelerators using the private scratchpad (RoT) and system DRAM.

Accelerator			HMAC		AES	
			RoT	RAM	RoT	RAM
Block Size [B]			64		16	
Accesses [#]	Accel.	Read	–	–	4	4
		Write	16	16	4	4
	RoT	Read	16	–	4	–
		Write	–	–	4	–
	RAM	Read	–	16	–	4
		Write	–	–	–	4
Cost [cycles]	Accel.		80	80	40	40
	RoT		80	–	40	–
	RAM		–	368	–	184
	TOT		160	448	80	224
Bandwidth [cycles / B]			2.5	7.0	5.0	14.0

tions. Semantic labels specify the purpose of executing a given instruction within the cryptographic operation (e.g., configuration, data digestion, waiting for accelerator, or finalization), enabling fine-grained correlation between software operations and hardware activity. This labeling and trace analysis facilitated cycle-level performance profiling, revealing which operations dominate execution time on each accelerator and identifying bottlenecks. By combining RTL traces, hand-annotated instruction semantics, and direct accelerator programming, the study provides a high-fidelity characterization of OpenTitan's cryptographic accelerators, independent of software overheads or OS effects.

3.2.2 Performance Analysis and Bottleneck Identification

The detailed characterization revealed significant insights into the operational efficiency and existing bottlenecks within OpenTitan's cryptographic accelerators. A meticulous analysis of the cycle consumption for HMAC and AES accelerators, utilizing a 4 KiB payload, unequivocally demonstrated that a dominant proportion of execution cycles are dedicated to memory operations. As summarized in Table 3.2, no less than 93.78% of the total cycles are consumed by memory accesses, with over 72% of these cycles specifically required for retrieving data from the system DRAM. In stark contrast, only a minimal fraction, at most 3.66%, of cycles are spent waiting for the accelerator to complete its computation before the next data chunk can be pushed into its FIFO. This in-depth analysis confirms that despite its hardware acceleration, OpenTitan's performance for these specific workloads is severely constrained by memory access latency and data movement overheads.

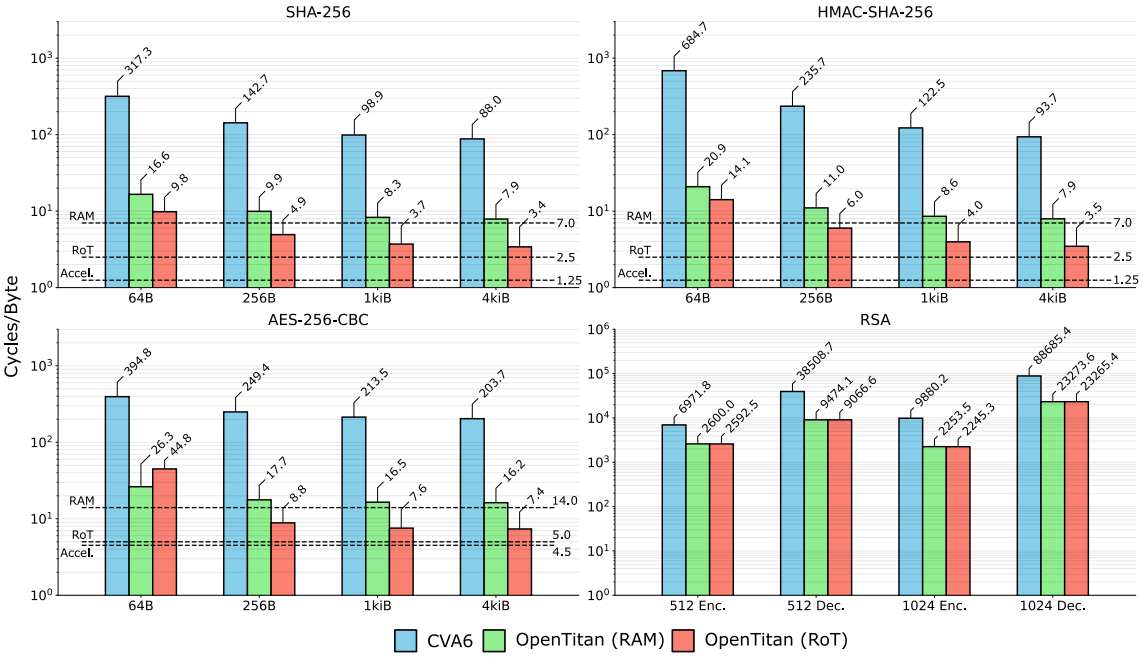

Figure 3.1: Characterization of the *Cycles/Byte* achieved by OpenTitan hardware accelerators for payloads from 64 bytes to 4 KiB. Multiple curves represent the performance of different cryptographic operations (SHA-256, HMAC, AES, RSA-512, RSA-1024) and data sources (system DRAM vs. private scratchpad). Horizontal lines indicate the theoretical maximum bandwidth of each accelerator and the estimated memory bandwidth limits, highlighting memory-bound scenarios and the impact of data movement overheads. A baseline curve for a software implementation on the CVA6 core is also included for comparison.

This pronounced dominance of memory accesses as a performance limiter is directly attributable to the varying costs associated with each memory operation. Table 3.3 provides a statistical overview of the cycle cost incurred by OpenTitan's Ibex microcontroller for a single memory access to either its private scratchpad or the system DRAM. Accessing the OpenTitan private scratchpad or any accelerator FIFO incurs an average cost of 5.7 cycles. Conversely, accessing the system DRAM, even with a LLC hit, approximately costs 23.4 cycles. The minimum and maximum cycle counts shown represent the architectural cost of these memory accesses, with higher counts typically observed following branch or control-flow transfers. To determine the maximum achievable bandwidth, the minimum memory access cost was considered, as it reflects the platform's peak potential.

Further extending the bandwidth analysis, Table 3.4 presents an estimation of the minimum cycles per byte achievable by the HMAC and AES accelerators. This estimation is derived by calculating the Ibex core's memory accesses required to collect input data, multiplied by the respective memory access cost. This analysis vividly demonstrates the maximum attainable throughput for these accelerators, highlighting that the process of pushing data into the accelerator FIFOs is fundamentally constrained by Ibex's ability to fetch words from memory. For instance, the HMAC accelerator necessitates fetching and pushing 16 4-byte words per compression loop

into its input FIFO, leading to a minimum bandwidth requirement of 2.5 CpB (Cycles per Byte) if data is sourced from the OpenTitan scratchpad, or 7.0 CpB if read from system DRAM. Similarly, a single AES-256 encryption or decryption iteration requires loading 4 words from memory, pushing them into the plaintext FIFO, reading back the previously ciphered 16-byte block, and writing it back to memory.

These quantitative data are visually reinforced by Figure 3.1, which illustrates the performance of OpenTitan's cryptographic accelerators across payloads ranging from 64 bytes to 4 KiB, including HMAC, AES, and RSA-512/1024. The figure compares data loaded from system DRAM and the private scratchpad against a software implementation on the CVA6 host core, with multiple curves representing each algorithm and data source. Horizontal lines indicate the theoretical maximum bandwidth of the accelerators and estimated memory limits. Overall, the graph highlights substantial speedups over software-only implementations, particularly when using the private scratchpad.

For instance, SHA-256 and HMAC achieve maximum speeds of 7.88 and 7.94 CpB respectively when data is loaded from system DRAM, improving to 3.41 and 3.47 CpB when using the private scratchpad. AES reaches a peak of 16.23 CpB from DRAM and 7.36 CpB from the scratchpad. RSA decryption, being more computationally intensive, requires approximately $9,100$ CpB and $23,300$ CpB for 512-bit and 1024-bit keys, respectively. These results clearly demonstrate that, although OpenTitan accelerators provide significant speedups compared to software, their throughput is often limited by memory access, particularly when data resides in system DRAM.

3.2.3 Summary of Experimental Results

In summary, the experimental results conclusively demonstrated the significant potential of OpenTitan as a cryptographic accelerator. Compared to a pure software implementation running on a modern RISC-V application-class core like CVA6, OpenTitan's hardware accelerators provided substantial speedups for various cryptographic functions. Specifically, for a 4KiB payload, leveraging OpenTitan results in the following improvements for symmetric algorithms:

- SHA-256: an 11.1× speedup,
- HMAC: an 11.8× speedup,
- AES-256: a 12.5× speedup.

For asymmetric cryptography, in particular RSA decryption, the speedups are:

- RSA-512: a 4.3× speedup,
- RSA-1024: a 3.8× speedup.

These results unequivocally indicate that harnessing OpenTitan's hardware capabilities dramatically enhances computational performance for cryptographic workloads when compared to software-only approaches. However, the characterization also revealed that the cryptographic accelerators were not fully saturated by the tested workloads. For HMAC and SHA-256, only 16% of the theoretical bandwidth was utilized, increasing to 28% for AES when data was loaded from system DRAM, improving to 37% for HMAC and 61% for AES when using the private scratchpad. This underutilization is primarily due to memory access latencies and the secure microcontroller's responsibility for transferring data between the system memory and the accelerator FIFOs.

To complement these findings, a preliminary assessment of the maximum memory bandwidth achievable by the Ibex secure microcontroller was conducted. This analysis quantified how efficiently Ibex can feed data to the accelerators in memory-bound scenarios, providing insight into the practical throughput limits and motivating the architectural optimizations discussed in subsequent chapters.

3.3 Architecture of the TitanSSL Software Framework and Data Flow

Following the isolated characterization of OpenTitan's intrinsic hardware performance, which established its potential for significant hardware acceleration, the subsequent and equally critical step is to seamlessly integrate these powerful features into a usable and efficient system. This transition from raw hardware performance to practical application within a SoC (Section 2.2) operating a Linux environment necessitates the development of a sophisticated software framework. This section introduces TitanSSL, a comprehensive software stack meticulously designed to bridge the gap between OpenTitan's powerful cryptographic accelerators and Linux-based applications running on a host processor. TitanSSL's core function is to securely and efficiently offload cryptographic tasks, such as SHA-256, RSA, and AES, from the Application Processor (AP) (the host processor) to the OpenTitan core, which functions as a dedicated Security Controller (SC). This redirection ensures that the robust security and performance benefits inherent in OpenTitan's hardware are fully leveraged by user-space applications without compromising the overall system's integrity or performance.

3.3.1 Overview of the TitanSSL Software Stack

The TitanSSL software stack is organized into two main operational domains, carefully designed to enforce a robust security boundary: one on the AP (host, specifically a CVA6 core running Linux) and the other within the SC (the Ibex core embedded in OpenTitan). This multi-layered architecture enables secure communication between untrusted user-space applications, which typically use the OpenSSL library, and the isolated, trusted environment of the SC, while maintaining clear separation of responsibilities and extensibility for additional cryptographic primitives.

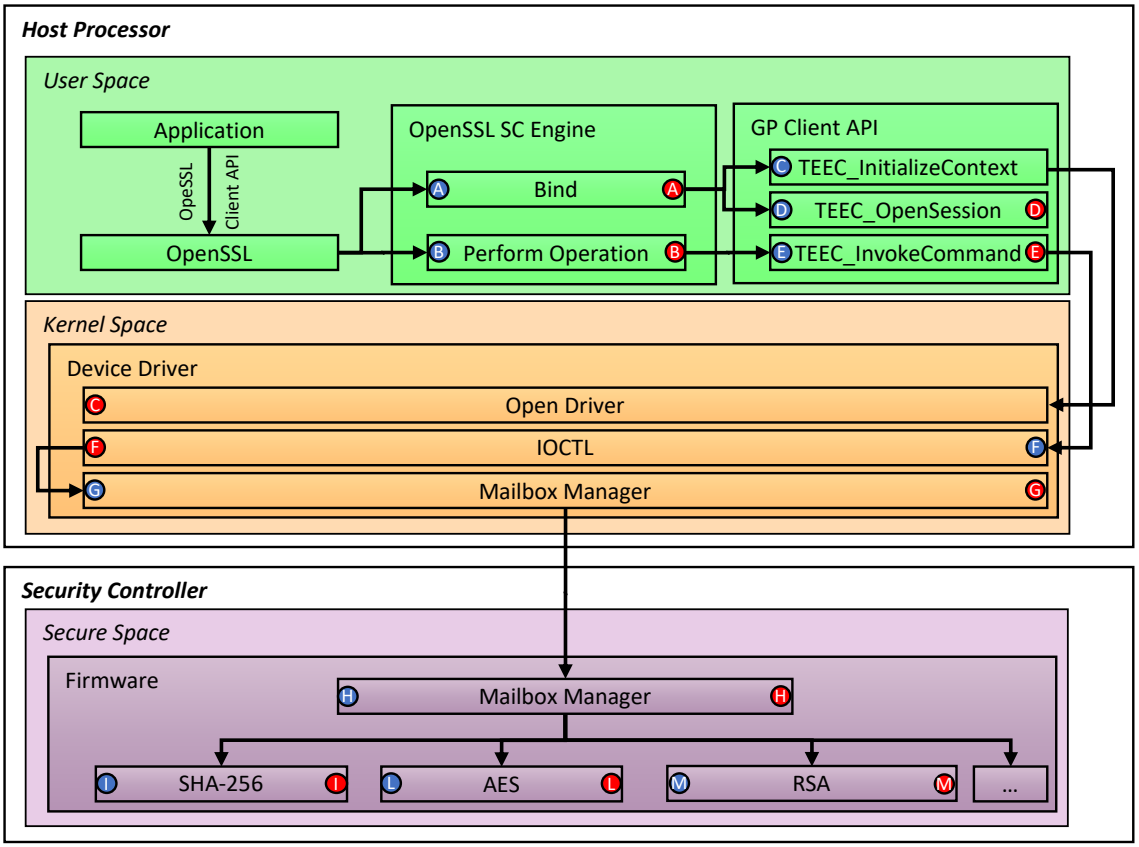

Figure 3.2: Illustration of the TitanSSL software stack, showing the separation between the Application Processor (User and Kernel Space) and the Security Controller, including the OpenSSL engine, TitanSSL driver, firmware, and hardware accelerators.

Figure 3.2 provides an illustration of this architecture, showing both the AP and SC domains. On the host, software is divided into User Space (depicted in green) and Kernel Space (shown in yellow), reflecting their privilege levels and responsibilities under Linux. The User Space hosts a custom OpenSSL Engine that offloads selected cryptographic operations to secure hardware, while the Kernel Space includes the TitanSSL Driver, managing communication with OpenTitan through a mailbox interface. Within the SC, dedicated firmware running on the Ibex core orchestrates cryptographic requests and interacts directly with hardware accelerators (e.g., AES, HMAC, RSA), ensuring isolation and secure execution. By combining domain separation with a well-defined communication channel, the TitanSSL stack achieves both robustness and scalability in secure embedded environments.

3.3.2 Component Breakdown and Responsibilities

This architectural overview provides a detailed blueprint of the TitanSSL software stack, highlighting the roles and responsibilities of each component from the AP down to the SC.

- **Application Processor – User Space:**
  - *Applications*: Represent any user-defined software that initiates cryptographic requests.
  - *OpenSSL Engine*: Intercepts standard OpenSSL cryptographic operations and redirects requests (e.g., SHA-256, AES, RSA) to TitanSSL, allowing applications to leverage OpenTitan hardware accelerators without modification.
  - *GlobalPlatform Client API Implementation*: Acts as an abstraction layer that manages secure communication with the SC. Responsible for initializing contexts and sessions as per the GlobalPlatform standard, providing a uniform interface for cryptographic device management.
- **Application Processor – Kernel Space:**
  - *TitanSSL Driver*: Facilitates communication between the user space OpenSSL Engine and the SC firmware. Operates in kernel space using a mailbox mechanism for bidirectional data exchange, translating high-level requests into commands understandable by the SC and handling the responses.
- **Security Controller – OpenTitan:**
  - *TitanSSL Firmware*: Custom firmware running on the Ibex core, orchestrating the execution of cryptographic requests received via the mailbox. Interfaces directly with OpenTitan hardware accelerators (HMAC, AES, OTBN, etc.), ensuring secure and efficient processing.
  - *Hardware Accelerators and Memory Management*: Handles cryptographic operations and manages internal memory structures such as private scratchpads, providing isolation and secure execution within the SC domain.

In addition to these components, the *Mailbox* serves as the critical communication interface between the Kernel Space driver and the OpenTitan SC. It enables asynchronous data transfer and synchronization between the host CVA6 core and the Ibex core, ensuring reliable and secure communication across the two domains. Moreover, the mailbox facilitates coordination between software and hardware components, allowing efficient dispatch of cryptographic requests and responses throughout the TitanSSL stack.

3.3.3 Data Flow and Interaction Mechanisms

Figure 3.3 provides a detailed illustration of the data flow and interactions across the TitanSSL software stack, highlighting how a cryptographic request propagates from the application layer down to OpenTitan hardware and back. The sequence shown in the figure (steps ①–⑥) corresponds to the following operational flow, ensuring both efficient hardware acceleration and strong security isolation:

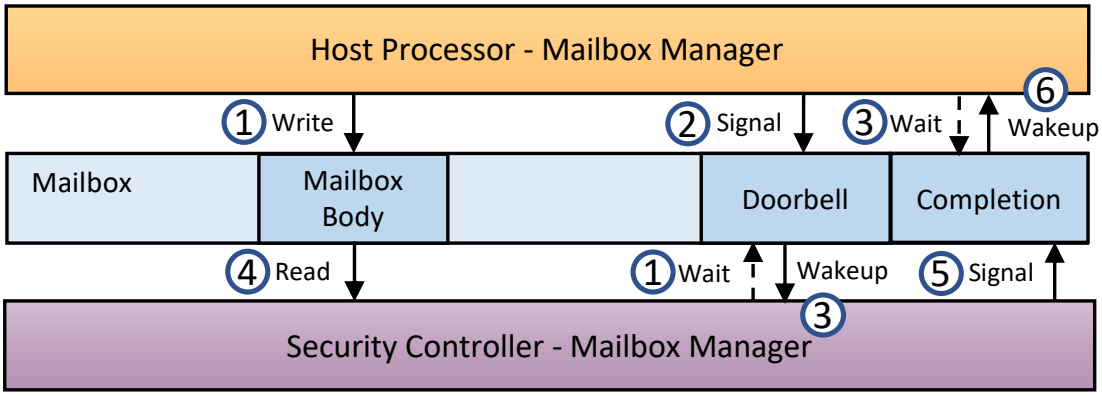

Figure 3.3: Detailed view of the mailbox communication mechanism between the host and Security Controller, illustrating how the TitanSSL Driver and Firmware coordinate data transfer, interrupts, and synchronization through the Mailbox Manager using the `Doorbell` and `Completion` registers.

1. *Application Request*: A user-space application running on the CVA6 processor invokes a standard cryptographic function such as hashing, encryption, or signature generation. The application interacts with OpenSSL as usual, remaining unaware that the operation will be offloaded to secure hardware.

2. *OpenSSL Engine Redirection*: The TitanSSL OpenSSL Engine intercepts the request and, using the GlobalPlatform Client API if necessary, packages the cryptographic task, its input data, and metadata into a standardized structure ready for kernel-level transmission.

3. *Kernel-Space Mediation*: The TitanSSL Driver receives the request through a system call and performs the necessary preprocessing. It translates virtual to physical memory addresses, pins the relevant pages to prevent swapping, and constructs the command and session information according to the ABI that defines communication conventions with OpenTitan. This includes the use of Master Pages, which organize memory regions for different source modules, and a Header Page, which stores metadata about memory layout and access rights.

4. *Mailbox Communication*: The driver writes the assembled message into the mailbox body, comprising the command, algorithm identifier, and physical memory references for input and output buffers ①. Once the message is ready, it triggers the `Doorbell` register ②, generating an interrupt to the Ibex core within OpenTitan. The Host Mailbox Manager in the TitanSSL Driver then enters an idle state ③, awaiting completion notification.

5. *Security Controller Firmware Processing*: Upon receiving the interrupt, the Mailbox Manager within the SC wakes up and reads the mailbox contents ④. The TitanSSL Firmware running on the Ibex core interprets the command, retrieves the necessary input data directly from physical memory via bus-master access, and delegates execution to the appropriate OpenTitan hardware

accelerator (e.g., HMAC, AES, or OTBN). Sensitive intermediate data are processed within private scratchpads, ensuring isolation and confidentiality.

6. *Result Return and Completion Signaling*: After computation, the firmware writes the results back to the designated output memory region and updates the `Completion` register ⑤. This action triggers an interrupt to the host processor, waking the Host Mailbox Manager ⑥ to signal that the operation has finished.

7. *Delivery to Application:* The TitanSSL Driver in the Kernel Space retrieves the results from the output buffer, unpins the associated memory pages, and passes the processed data back to the OpenSSL Engine. The Engine transparently completes the original API call and returns the result to the user-space application.

This integrated mechanism ensures that all sensitive cryptographic computations occur entirely within the secure confines of OpenTitan, leveraging its dedicated hardware accelerators for performance and isolation. Meanwhile, user-space applications experience a seamless, standards-compliant interface identical to conventional cryptographic APIs. The mailbox and its synchronization registers, `Doorbell` and `Completion`, enable reliable, asynchronous communication between the AP and SC, ensuring efficient coordination without exposing sensitive data outside the secure domain.

3.4 Implementation Details and Security Guarantees

The implementation of TitanSSL is meticulously designed to provide robust security guarantees across the entire software stack while simultaneously maximizing performance by offloading cryptographic workloads to dedicated hardware. This sophisticated architecture necessitates precise coordination between software components running on the host application processor and the custom firmware executing within the OpenTitan SC. A paramount aspect of this implementation is its strict adherence to a strong separation between trusted and untrusted domains. This fundamental principle ensures that critical cryptographic assets, particularly private keys and other sensitive data, never leave the secure boundaries of OpenTitan, thereby providing a hardware-anchored RoT for all security operations. This section delves into the specific implementation nuances of each TitanSSL component, detailing how cryptographic tasks are executed, how communication is managed, and the inherent security measures safeguarding the system against various threats. It also touches upon key management strategies, secure storage mechanisms within OpenTitan, and the overarching design philosophy guiding the development.

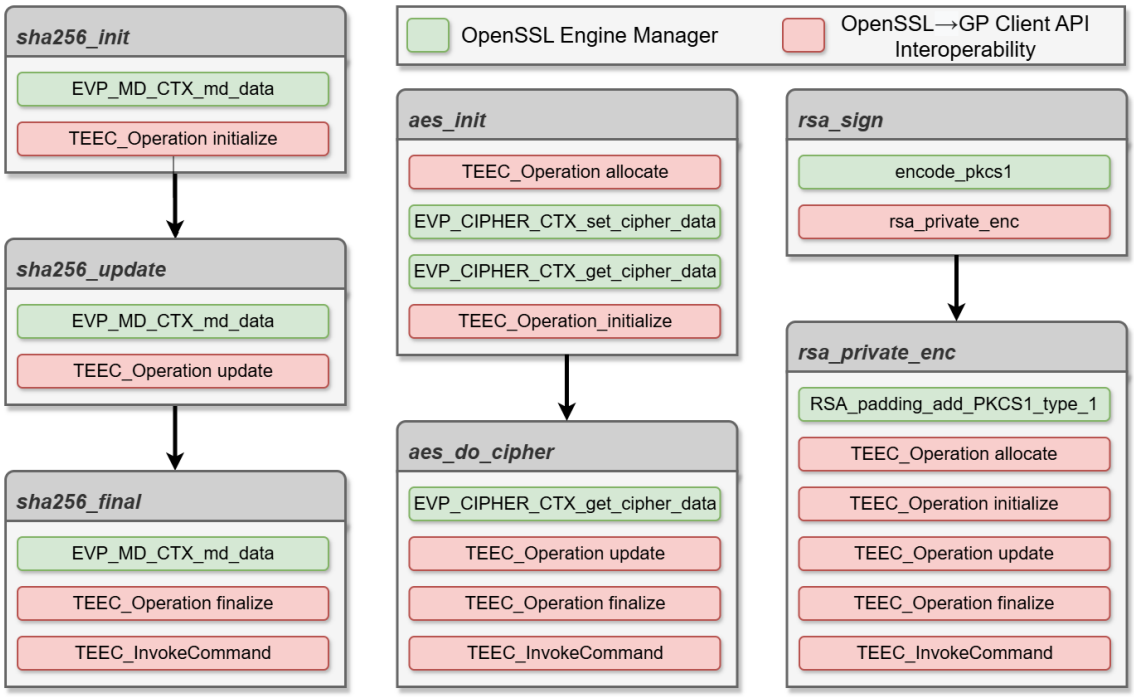

Figure 3.4: Illustration of the function flows within the TitanSSL OpenSSL Engine for cryptographic operations. Subfigure (a) shows the SHA-256 initialization, update, and finalization steps; (b) depicts AES key setup, encryption/decryption, and cleanup; (c) represents RSA signing and encryption operations, highlighting how each algorithm is processed and offloaded to OpenTitan hardware.

3.4.1 The TitanSSL OpenSSL Engine Implementation

The TitanSSL OpenSSL Engine is a crucial user-space component of the TitanSSL framework. Its primary role is to intercept calls to standard OpenSSL cryptographic functions (such as SHA-256, AES, and RSA) and redirect them to OpenTitan's hardware accelerators. This redirection is achieved by dynamically substituting the default software implementations with custom functions provided by the engine, leveraging OpenSSL's extensible architecture. The engine integrates OpenSSL functionalities with the GlobalPlatform standard through the implementation of the GlobalPlatform Client API, represented by the `TEEC` component within the overall software stack.

The engine's responsibilities include initializing communication contexts (`TEEC_Context`) and sessions (`TEEC_Session`), as defined by the GlobalPlatform Client API, and preparing cryptographic requests for the lower layers of the system. To achieve this, the engine manages specific data structures, notably the `TEEC_Operation` structure, which is central to implementing the GlobalPlatform standard for subsequent steps in the cryptographic operation. This structure allows the transfer of operation-specific parameters and data between the OpenSSL engine and the underlying driver.

To maintain driver agnosticism and support various cryptographic algorithms, the engine encapsulates all operation-specific information within dedicated meta-information data structures. For instance, AES algorithms require initialization vectors or keys, while RSA operations demand exponents and modulus values to handle asymmetric keys. Each cryptographic algorithm that requires such supplementary information is represented by a dedicated data structure, such as `titanssl_meta_<operation>_t`. The specific function flows within the OpenSSL Engine for key cryptographic algorithms like SHA-256, AES, and RSA are visually represented in Figure 3.4. These diagrams illustrate how each operation is processed from initialization through to execution.

SHA-256 Implementation

For SHA-256 operations, the default OpenSSL functions `SHA_Init`, `SHA_Update`, and `SHA_Final` are used to initialize the hash context, process input data in chunks, and finalize the computation to produce the digest. These functions are replaced by the engine's custom implementations `sha256_init`, `sha256_update`, and `sha256_final`. The flow of these functions is illustrated in Figure 3.4a.

- The `sha256_init` function retrieves a `void` pointer (`md_data`) from the `EVP_MD_CTX` structure, a classic OpenSSL Engine management approach. This pointer is then cast as a `TEEC_Operation` structure pointer to facilitate GlobalPlatform standard implementation. It initializes the `Started` and `ParamTypes` fields of `TEEC_Operation` and allocates its `Params` field to a `hash_struct_t` pointer.
- The `sha256_update` function similarly retrieves the `md_data` pointer and populates the fields within the `hash_struct_t` with the input buffer address and its size.
- Finally, the `sha256_final` function processes the last chunk and triggers the `TEEC_InvokeCommand` to execute the operation on OpenTitan.

AES Implementation

For AES operations, the engine replaces the default OpenSSL functions `AES_init`, `AES_do_cipher`, and `AES_cleanup`. These functions respectively initialize the AES key schedule, perform encryption or decryption of the input data, and release any allocated resources. The engine uses its custom implementations `aes_init`, `aes_do_cipher`, and `aes_cleanup` instead. The flow of these functions is illustrated in Figure 3.4b.

- The `aes_init` function allocates a `TEEC_Operation` structure and uses `EVP_CIPHER_CTX_set_cipher_data` to link the `TEEC_Operation` pointer with the `cipher_data` pointer of the `EVP_CIPHER_CTX` structure. It then retrieves this `cipher_data` pointer (cast as `TEEC_Operation`) and initial-

izes the `Started` and `ParamTypes` fields, allocating the `Params` field to an `aes_struct_t` pointer.

- The `aes_do_cipher` function retrieves the `TEEC_Operation` pointer and uses `TEEC_Operation_update` and `TEEC_Operation_finalize` to populate the structure with input and output buffer addresses and their sizes. Similar to SHA-256, the `TEEC_InvokeCommand` is ultimately called to dispatch the operation to OpenTitan.

It is important to note that within OpenSSL's EVP framework, the `do_cipher()` function can be invoked multiple times for a single encryption or decryption operation. Intermediate calls process successive data chunks, while the final call handles block alignment. This behavior ensures that streaming or non-contiguous data can be processed correctly while maintaining proper block structure and padding integrity. The actual addition or removal of padding for the last block is managed by the underlying AES implementation (i.e., by the SC), ensuring that the input length matches the block size (16 bytes). From the user-level perspective, this always requires one additional `do_cipher()` call to finalize the operation and produce correctly aligned ciphertext or plaintext.

RSA Implementation

For RSA algorithms, specific operations like `rsa_sign` and `rsa_private_enc` are handled by the engine. These operations internally rely on the lower-level functions described below. The integration of these functions is illustrated in Figure 3.4c.

- The `TEEC_Operation_allocate` function allocates the `TEEC_Operation` structure required for storing RSA parameters.
- The `TEEC_Operation_initialize` function populates the `Started` and `ParamTypes` fields and allocates the `Params` field to an `rsa_struct_t` pointer.
- The `TEEC_Operation_update` function populates the `rsa_struct_t` with the input buffer address and its size.
- The `TEEC_Operation_finalize` function populates the `rsa_struct_t` with the output buffer address and its size. The `TEEC_InvokeCommand` is then invoked with the configured `TEEC_Operation`.

In this way, high-level RSA operations such as signing and verification utilize these lower-level helper functions to manage memory, parameters, and communication with the secure environment.

After the `TEEC_Operation` structure is prepared, the `TEEC_InvokeCommand` function manages the switch to select the correct algorithm and performs checks on parameter types before ultimately calling the Device Driver's IOCTL. This mechanism ensures that the cryptographic request, along with its metadata, is securely and efficiently passed down to the Kernel Space for further processing by OpenTitan.

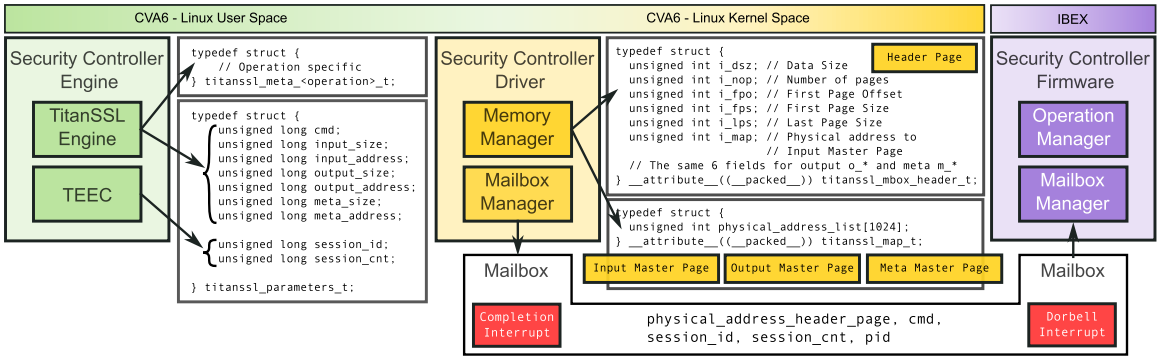

Figure 3.5: Illustration of the complete TitanSSL software stack, showing the flow of cryptographic requests from the CVA6 host processor through the OpenSSL Engine and Kernel Driver, across the mailbox interface, and into the OpenTitan Ibex secure environment. Key data structures are highlighted to demonstrate driver-agnostic communication and secure coordination between software and hardware components.

3.4.2 The Linux Kernel Driver for OpenTitan Cryptographic Accelerators

The TitanSSL Driver is a critical kernel-space component responsible for managing requests directed to the OpenTitan SC and facilitating secure communication between the AP and OpenTitan. This component exposes its functionalities through a character device, making it accessible to the user-space TitanSSL OpenSSL Engine. User-space applications interact with this device through standard file operations and, more importantly, via the IOCTL interface for device-specific commands and data exchange.

To provide a clear understanding of the overall communication flow and the interaction between the software stack components and OpenTitan, Figure 3.5 illustrates the architecture. This figure visually represents how the TitanSSL OpenSSL Engine integrates OpenSSL functionalities and how the `TEEC` component interacts with the Kernel Space. It also illustrates how the TitanSSL Driver manages communication via the mailbox and how TitanSSL Firmware on OpenTitan processes requests, leveraging its cryptographic accelerators and secure memory. Data structures are highlighted to ensure driver agnosticism.

The TitanSSL Driver's primary responsibility is to handle the low-level interactions with OpenTitan, which includes translating virtual addresses from user space to physical addresses that OpenTitan can utilize. This translation process is performed in two essential steps within the kernel:

- *Pinning memory pages*: This prevents data corruption by locking user-space memory pages into physical DRAM, ensuring they are not swapped out or moved during the operation. Kernel functions, such as `get_user_pages_fast`, are typically employed for this purpose to ensure the memory remains resident and accessible.

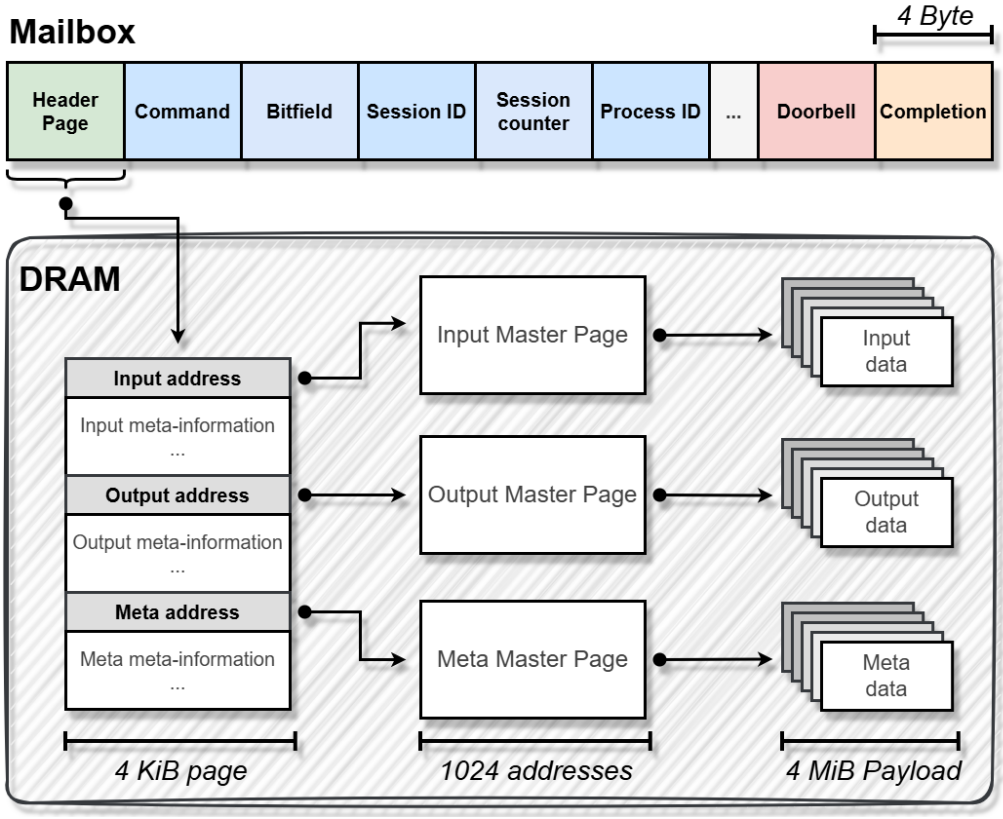

Figure 3.6: Illustration of the mailbox structure and its memory layout, including register fields and the paging scheme used for communication between the CVA6 host and OpenTitan, enabling secure and asynchronous request handling.

- *Translating virtual addresses*: The virtual addresses provided by the user-space application are converted into corresponding physical addresses that OpenTitan can directly access, as OpenTitan operates as a bus master in the architecture. The Linux kernel function `page_to_phys()` is used for this translation, obtaining the physical address by adding the offset at which the specified buffer in the page is located.

Communication between the CVA6 host core and OpenTitan is mediated by the custom mailbox, which is a memory-mapped peripheral with a 40-byte shared memory region in the SoC. The structure of this mailbox is illustrated in Figure 3.6. This shared region is designed for data sharing and includes two specialized 32-bit registers: `Doorbell` and `Completion`. The least significant bit of these registers is connected to the interrupt controllers of Ibex and CVA6, respectively, enabling an asynchronous acknowledgment system between the two cores. The Mailbox Manager component within the TitanSSL Driver is responsible for orchestrating this data exchange.

When the CVA6 wants to communicate with OpenTitan, the driver writes a message into the general-purpose registers of the mailbox, containing the command to be executed and payload metadata. It then sets the `Doorbell` register, triggering an interrupt request to the Ibex core within OpenTitan. Conversely, upon completing an operation, OpenTitan signals completion by asserting the `Completion` register in the mailbox, triggering an interrupt to the CVA6 core.

A critical implementation detail addresses the lack of cache coherence between OpenTitan and the CVA6 host core. To ensure that CVA6 utilizes the most recent results from OpenTitan's computations, especially when memory locations written by OpenTitan might be cached by CVA6, a robust memory fence (or barrier) instruction mechanism is employed. After OpenTitan completes writing the operation output to shared memory, a memory `fence.i` instruction is executed. This fence ensures that all memory writes performed by OpenTitan are completed and made visible to other bus masters, including the CVA6 processor, before any subsequent memory accesses by CVA6 to that region are initiated. This guarantees data consistency and prevents CVA6 from reading stale cached data, effectively optimizing memory synchronization by ensuring proper memory ordering and visibility across the system.

The ABI designed for sending information to the OpenTitan firmware consists of several key data structures, explicitly highlighted to ensure driver agnosticism. This agnosticism means the driver remains independent of the specific cryptographic operations executed by the OpenSSL engine, supporting extensibility and flexibility. The organization of the mailbox and its fields, showing how data is structured for communication between the CVA6 and OpenTitan, is illustrated in Figure 3.6. The ABI includes:

- **Three Master Pages**: These store the physical addresses of the input, output, and meta-information memory pages, as provided by the application and engine.
- **A Header Page**: This contains navigation information for the physical input, output, and meta-information pages, including the physical address of the Master Pages, the number of pages, offsets, and page sizes.
- **Data for the mailbox**: This includes the physical address of the Header Page, the requested command, session information, and the process ID (`pid`) of the application invoking the operation.

When transmitting data from the CVA6 to OpenTitan, metadata concerning the cryptographic operation, along with its payload, are passed through the mailbox. The payload itself is transmitted as a sequence of physical memory addresses. This design allows OpenTitan to freely access every location in the SoC's address space using the provided physical memory addresses, enabling efficient memory access for the Ibex microcontroller.

OpenTitan Security Controller Firmware Implementation

The TitanSSL Firmware in the SC is a custom firmware developed in C to run on OpenTitan's Ibex processor, specifically designed to handle cryptographic requests by capitalizing on the available hardware accelerators within OpenTitan. This firmware operates within the secure and isolated environment provided by OpenTitan, performing critical cryptographic operations delegated by the host processor. It was developed following the guidelines and reference documentation provided by the OpenTitan project [40]. The development of this custom firmware is integral to enabling OpenTitan's role as a SC. Unlike the original OpenTitan design, which might target a standalone implementation with external I/O, this integrated SoC design adapts OpenTitan to communicate primarily with the host processor via the custom mailbox, rather than external peripherals.

From an implementation perspective, the firmware has been designed to operate in close integration with the mailbox communication mechanism, which serves as the fundamental interface between the host processor and the SC. The design focuses on ensuring efficient data handling and secure memory access while maintaining compliance with OpenTitan's privilege and isolation model. At its core, the Mailbox Manager within the SC continuously monitors the `Doorbell` register for incoming requests from the CVA6 host. When a signal is received, the TitanSSL Firmware retrieves the operation parameters and the Header Page memory address from the mailbox, enabling an asynchronous and event-driven processing model. By leveraging OpenTitan's capability as a bus master, the firmware accesses the system's physical memory through the addresses specified in the Header and Master Pages, retrieving the complete data structures required for the cryptographic operation. This implementation ensures that all communications and data exchanges between the AP and OpenTitan occur through secure, well-defined interfaces, preserving system integrity and minimizing latency. Moreover, all sensitive data are handled exclusively within the secure memory space of the OpenTitan environment, preventing exposure to untrusted software layers and ensuring strong data confidentiality.

Once the request parameters and memory addresses have been retrieved, the firmware proceeds with the execution of the cryptographic operation. The cryptographic computation is then carried out using OpenTitan's hardware accelerators, which are natively integrated within the secure subsystem. The firmware coordinates these accelerators through their memory-mapped registers, ensuring strict control over access privileges and execution order. The Key Manager block is responsible for the management and protection of cryptographic keys.

For SHA-256 operations, the firmware interfaces with the HMAC accelerator through its memory-mapped FIFO buffer. The `HMAC_CFG` register is configured to select SHA-256 mode and set the endianness of input words (`CFG.endian_swap`) and digest output (`CFG.digest_swap`). Message data are sequentially written into the FIFO in 64-byte blocks, with the firmware monitoring the `STATUS.fifo_depth` and `STATUS.fifo_full` fields to avoid overflows. If the FIFO becomes full, writes are paused until sufficient space is available, ensuring correct and deterministic processing. Once all data blocks are processed, the 256-bit digest is read from `DIGEST_0`–`DIGEST_7`, completing the SHA-256 computation securely and efficiently.

AES encryption and decryption are managed through the dedicated AES hardware unit, which is initialized only when the module is confirmed to be in an idle state by polling the `STATUS.IDLE` flag. Initialization begins by configuring the `CTRL_SHADOWED` register, which defines the operational mode, key length, and automatic reseeding. Once the configuration is complete, the initial key material is written into the `KEY_SHARE0_0`–`KEY_SHARE0_7` and `KEY_SHARE1_0`–`KEY_SHARE1_7` registers, corresponding to the two additive key shares whose XOR combination forms the effective AES key. All sixteen 32-bit key registers are explicitly written regardless of the selected key length (AES-128, AES-192, or AES-256) to guarantee deterministic initialization and proper internal state randomization. When operating in CBC, CFB, OFB, or CTR modes, the initialization vector is subsequently loaded into the `IV_0`–`IV_3` registers, also ensuring that the AES block remains idle before each write. The firmware enforces these steps to maintain synchronization between hardware and software layers and to prevent invalid register writes during active encryption or decryption. This strict initialization procedure ensures that the AES unit operates deterministically, securely, and in full compliance with the OpenTitan hardware specification.

For operations requiring large integer arithmetic or specialized cryptographic routines, the firmware utilizes the OTBN subsystem. Program binaries and input data are loaded into its dedicated instruction and data memories, both isolated from those of the Ibex core. Execution is triggered via a control register, and upon completion, OTBN signals an interrupt to the firmware, which then retrieves the results from its local data memory. The OTBN architecture enforces multiple layers of protection, including the absence of speculative execution and caching, as well as complete inaccessibility of its scratchpad memory to the host core. These measures collectively provide a hardened and deterministic execution environment, ensuring both operational reliability and strong resistance to side-channel or fault-injection attacks.

This precise interaction, orchestrated through dedicated device interface functions and carefully configured registers, ensures that sensitive cryptographic operations are executed securely and efficiently within OpenTitan's hardened environment. By leveraging its hardware accelerators and isolated memory, the firmware maintains strong data confidentiality and integrity while providing a well-defined and controlled communication channel with the host CVA6 processor.

3.4.3 Security Guarantees and Threat Model

The TitanSSL implementation incorporates several security guarantees, which have been thoroughly analyzed through a comprehensive threat model covering the entire system architecture. This analysis assumes an adversary capable of manipulating input data to gain unauthorized privileges or extract sensitive information, but without physical access to the system. The threat model utilizes the STRIDE framework to assess potential software vulnerabilities and data flow risks [93]. The specific classification of OWASP Top 10 [94] and CWE Top 25 [95] vulnerabilities in relation to the STRIDE threat model is presented in Table 3.5, providing a detailed mapping to understand the nature of potential attacks.

Table 3.5: Classification of system threats according to the STRIDE model [93], showing the corresponding OWASP 2017 Top 10 [94] and CWE 2023 Top 25 [95] vulnerabilities, grouped by type to highlight potential security risks and areas requiring mitigation.

STRIDE	OWASP	CWE	CWE Groups
Spoofing	A2:2017 - Broken Authentication	CWE-287, CWE-798 CWE-862, CWE-306	Broken Authentication
Tampering	A1:2017 - Injection	CWE-787, CWE-20 CWE-416, CWE-476 CWE-190, CWE-119 CWE-362	Wrong Memory Management
Repudiation	A10:2017 - Insufficient Logging & Monitoring	-	-
Information Disclosure	A3:2017 - Sensitive Data Exposure, A6:2017 - Security Misconfiguration	CWE-20, CWE-125 CWE-78, CWE-476 CWE-119	Wrong Input Validation, Buffer Overflow
Denial of Service	A9:2017 - Software with known vulnerabilities	CWE-787, CWE-20 CWE-416, CWE-476 CWE-190, CWE-362 CWE-400	Wrong Memory Management
Elevation of Privilege	A5:2017 - Broken Access Control, A6:2017 - Security Misconfiguration	CWE-78, CWE-287 CWE-798, CWE-862 CWE-306, CWE-94	Broken Access Control

Components within the system are assigned criticality levels ranging from 0 (not evaluated) to 9 (handling sensitive data), where sensitive data include cryptographic secrets and critical physical addresses. Software components are analyzed for vulnerabilities related to Spoofing, Repudiation, and Elevation of Privilege, while data flows are examined for risks of Tampering, Information Disclosure, and Denial of Service. The application itself and its data flow are generally considered outside the scope of this analysis, as the software stack is designed to remain transparent to pre-existing applications.

Specific mitigations have been implemented across the software stack. The OpenSSL Engine is rated with a low criticality level, as it does not store sensitive data and operates with minimal privileges; thus, Spoofing and Repudiation are less relevant, and Elevation of Privilege is handled by subsequent authentication layers. The GlobalPlatform Client API has a higher criticality due to its role as the access point to Kernel Space, and employs strong authentication through Session and Context mechanisms to prevent Spoofing and Elevation of Privilege. Session tracking additionally provides logging and monitoring to prevent Repudiation.

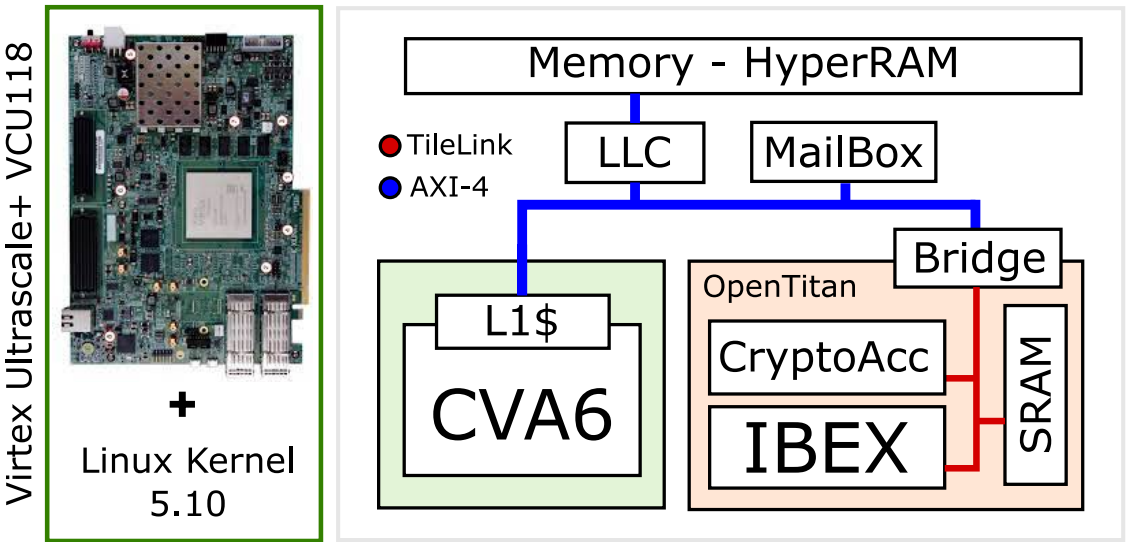

Figure 3.7: Illustration of the hardware architecture of the FPGA-based SoC used for performance evaluation, showing the CVA6 host, integrated OpenTitan Security Controller, and the testbed used to measure execution overhead of TitanSSL software stack components.

The Linux Kernel Driver mitigates threats related to Broken Access Control and Broken Authentication by leveraging Linux privileges, ensuring only authorized users can access it. This significantly reduces the risks of Spoofing and Elevation of Privilege. Repudiation, specifically Insufficient Logging and Monitoring, is addressed via kernel log messages. Data flows between the Device Driver and the Mailbox represent a critical point, as sensitive information such as physical addresses could be exposed if an attacker gains direct access to the driver. This risk is mitigated through careful software implementation and strict buffer validation, ensuring that all structures passed between components include sizes and offsets to prevent out-of-bounds access.

The overall system design further benefits from inherent security features. Critical cryptographic assets, including private keys and sensitive data, are confined within the secure boundaries of OpenTitan's hardware-anchored RoT, preventing exposure to the host processor. The mailbox mechanism between the CVA6 and OpenTitan is secured through hardware-enforced memory protection and verified boot processes, restricting write access to privileged kernel mode only. Offloading cryptographic operations to dedicated hardware accelerators reduces the attack surface and enhances resistance to side-channel attacks, while standardized interfaces, such as the GlobalPlatform API, provide a robust and interoperable framework for managing secure communication.

Together, these meticulous implementation details, the inherent security features of OpenTitan, and the application of the STRIDE threat model ensure that TitanSSL provides an end-to-end secure and efficient solution for cryptographic offloading in RISC-V based embedded systems.

3.5 Performance Evaluation and Analysis of TitanSSL

The performance evaluation of the TitanSSL software stack is a critical component of its validation, demonstrating the efficacy of offloading cryptographic operations to OpenTitan's hardware accelerators. Our analysis characterized and compared TitanSSL's performance against software-only implementations, as well as against the theoretical limits of the accelerators themselves. This evaluation confirms that the overhead introduced by the TitanSSL software stack is outweighed by the performance gains achieved through OpenTitan's dedicated cryptographic hardware.

3.5.1 Experimental Setup and Methodology

All performance measurements were executed directly on the target SoC using a lightweight micro-benchmark exercising OpenSSL's high-level `EVP` API via either the host CPU or the TitanSSL engine. The experimental setup is shown in Figure 3.7. The system under test is implemented on a Xilinx VCU118 FPGA and comprises a CVA6 application core running Linux together with an integrated OpenTitan SC. Benchmarks were conducted with the SoC clocked at 25 MHz. The harness performs the cryptographic operation (AES-256-CBC, SHA-256, or RSA), measures elapsed time using `clock_gettime(CLOCK_MONOTONIC_RAW)`, and computes throughput. Core timing is performed around the `EVP` sequence to capture end-to-end user-space latency:

Listing 3.1: Excerpt of the code region used to measure benchmark timing.

```
clock_gettime(CLOCK_MONOTONIC_RAW, &start);
EVP_MD_CTX *ctx = EVP_MD_CTX_new();
EVP_DigestInit_ex(ctx, EVP_sha256(), e);
EVP_DigestUpdate(ctx, data, filesize);
EVP_DigestFinal_ex(ctx, hash, &hash_len);
EVP_MD_CTX_free(ctx);
clock_gettime(CLOCK_MONOTONIC_RAW, &end);
```

Each workload is executed in two modes: native OpenSSL, and TitanSSL engine. For each input size, the measurement is repeated ten times, and the average elapsed time is used to compute throughput, ensuring reproducibility and statistically meaningful results.

From each run we extract the following metrics:

- *Throughput* in KiB/s, reported separately for SHA-256 and AES-256-CBC.
- *Speedup* of the TitanSSL engine relative to the CPU implementation.
- *Effective utilization of hardware limits* (OT Limit), representing the percentage of the theoretical maximum throughput of each accelerator, as specified in the OpenTitan datasheet, that is achieved by the measured performance.

Table 3.6: Measured throughput (KiB/s), speedup, and effective hardware utilization (OT Limit) for SHA-256 and AES-256-CBC across different payload sizes. Columns "OpenSSL" and "TitanSSL" show the throughput for the software-only and accelerator-offloaded implementations, respectively. "Speedup" reports the ratio between TitanSSL and OpenSSL throughput, indicating performance improvement. "OT Limit" indicates the percentage of the theoretical maximum throughput of the OpenTitan hardware accelerator achieved for each operation.

Payload	SHA [KiB/s]		Speedup	OT	AES [KiB/s]		Speedup	OT
[Byte]	OpenSSL	TitanSSL		Limit	OpenSSL	TitanSSL		Limit
16	0.87	1.43	1.65x	0.01 %	2.65	1.36	0.51x	0.01 %
32	2.22	2.78	1.25x	0.01 %	6.74	2.73	0.41x	0.01 %
48	3.33	4.39	1.32x	0.02 %	8.61	3.73	0.43x	0.02 %
64	4.36	5.98	1.37x	0.03 %	11.66	5.09	0.44x	0.02 %
112	6.89	10.07	1.46x	0.05 %	19.10	9.17	0.48x	0.04 %
128	8.27	11.79	1.43x	0.06 %	22.67	10.74	0.47x	0.04 %
240	14.82	17.94	1.21x	0.09 %	35.79	19.32	0.54x	0.08 %
256	14.50	18.26	1.26x	0.09 %	32.98	20.70	0.63x	0.08 %
496	25.41	36.71	1.44x	0.19 %	51.35	36.90	0.72x	0.15 %
512	26.50	41.97	1.58x	0.21 %	50.13	36.93	0.74x	0.15 %
1008	47.56	78.08	1.64x	0.40 %	73.68	67.79	0.92x	0.28 %
1024	45.44	84.40	1.86x	0.43 %	70.92	69.68	0.98x	0.29 %
2032	69.60	169.00	2.43x	0.87 %	93.01	122.53	1.32x	0.50 %
2048	69.61	169.03	2.43x	0.87 %	96.50	121.31	1.26x	0.50 %
4080	102.86	299.15	2.91x	1.53 %	108.40	171.03	1.58x	0.70 %
4096	96.92	273.34	2.82x	1.40 %	109.91	169.81	1.54x	0.70 %
8176	125.73	523.85	4.17x	2.68 %	112.62	247.14	2.19x	1.01 %
8192	125.67	502.89	4.00x	2.57 %	114.08	223.37	1.96x	0.91 %
16368	147.02	893.59	6.08x	4.58 %	116.90	279.76	2.39x	1.15 %
16384	147.37	847.60	5.75x	4.34 %	116.53	274.93	2.36x	1.13 %
32752	160.24	1257.49	7.85x	6.44 %	118.74	318.35	2.68x	1.30 %
32768	162.89	1205.64	7.40x	6.17 %	118.75	310.59	2.62x	1.27 %
65520	169.59	1599.08	9.43x	8.19 %	114.62	333.02	2.91x	1.36 %
65536	170.26	1604.54	9.42x	8.22 %	114.51	315.25	2.75x	1.29 %
131056	174.68	1818.38	10.41x	9.31 %	115.05	341.99	2.97x	1.40 %
131072	173.95	1826.51	10.50x	9.35 %	114.93	340.68	2.96x	1.40 %

3.5.2 Performance Results and Analysis

The experimental results, summarized in Table 3.6, demonstrate measurable performance improvements achieved by the TitanSSL framework across a variety of cryptographic workloads on the FPGA-based SoC.

For SHA-256 hashing, TitanSSL consistently outperforms the native OpenSSL implementation. Small payloads (16–512 B) already benefit from offloading, with speedups ranging from 1.3× to 1.6×. As the payload size increases, the advantage grows substantially: for 2 KiB, the speedup reaches 1.86×, and for 128 KiB payloads, the speedup exceeds 10×. The effective utilization of the OpenTitan accelerator (OT Limit) also rises with payload size, approaching 9–10% of the theoretical maximum for the largest payloads tested. These trends confirm that larger workloads amortize software overhead and allow the hardware accelerators to dominate performance.

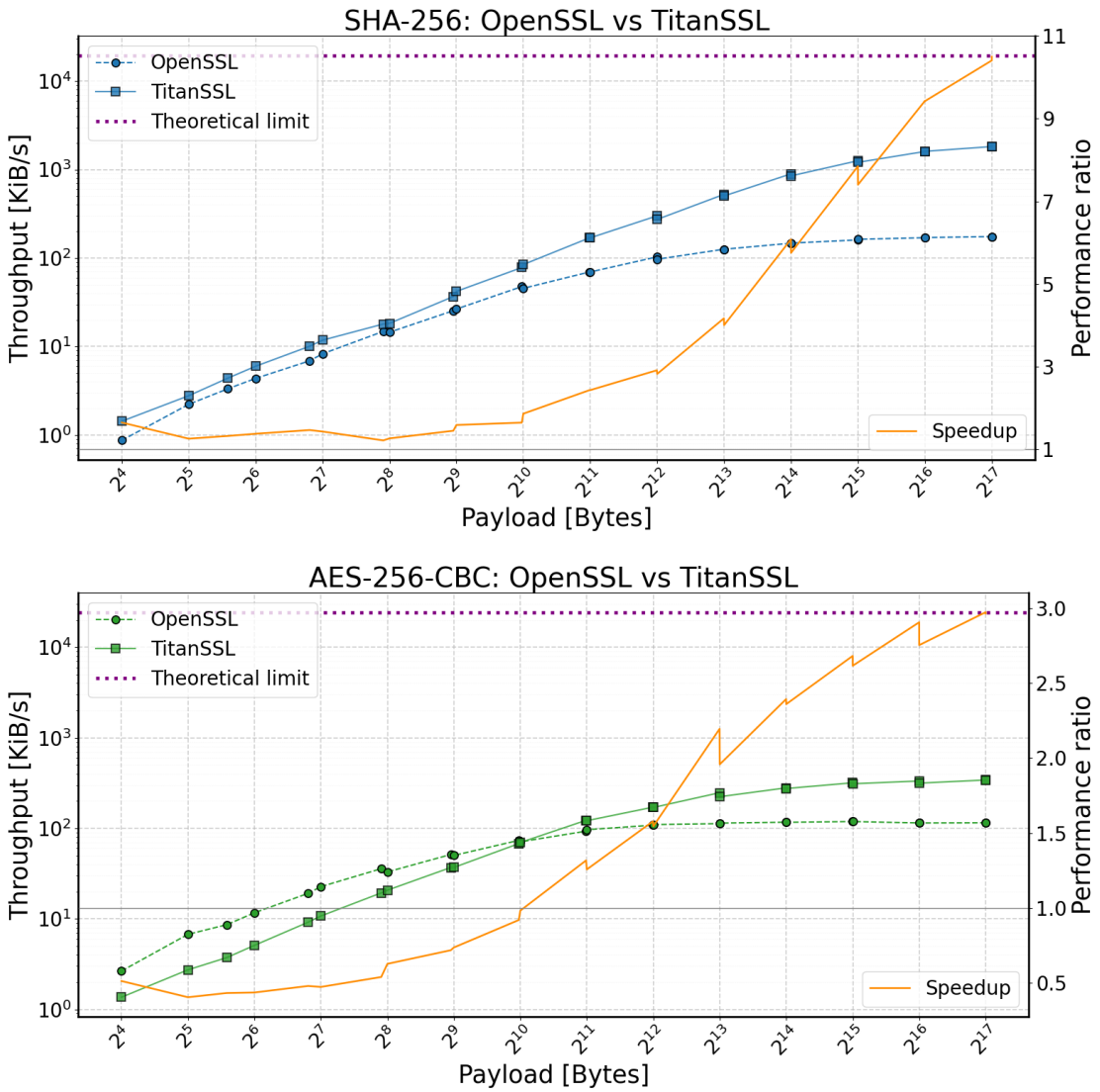

Figure 3.8: Throughput and performance comparison between TitanSSL and native OpenSSL across varying payload sizes. The upper plot reports results for SHA-256, while the lower plot shows AES-256-CBC measurements. In each plot, the left y-axis represents throughput in KiB/s on a logarithmic scale, and the right y-axis represents the performance ratio between TitanSSL and OpenSSL. The x-axis reports payload size in bytes on a base-2 logarithmic scale. The theoretical limit indicates the ideal accelerator throughput derived from datasheet specifications and scaled to the system operating frequency, highlighting both absolute performance and the acceleration achieved through offloading.

AES-256 in CBC mode exhibits similar behavior, though the baseline CPU throughput is higher due to lower per-byte hashing overhead. For small payloads (16–128 B), speedups are moderate (0.4–0.7×) due to software stack overhead. For payloads above 1 KiB, TitanSSL surpasses CPU performance, with speedups reaching 2–3× at 4–16 KiB and gradually increasing to nearly 3× for the largest payloads. The effective hardware utilization follows the same trend, achieving 1–1.4% of the theoretical AES accelerator bandwidth for the largest transfers.

RSA operations with 1024-bit keys were also evaluated and show limited acceleration: signing achieves a modest 1.4× speedup, while verification remains close to the software-only baseline. This confirms that for asymmetric operations, communication overhead within TitanSSL can offset the benefits of the accelerator for the tested key size.

These performance trends are further illustrated in Figure 3.8, which highlights the relationship between payload size and speedup. From the plots, it is evident that TitanSSL's advantage becomes more pronounced as payloads grow, with the measured throughput gradually approaching the hardware's theoretical limits for large transfers. The plots also make clear the software overhead impact on smaller payloads, particularly for AES, and show how offloading to hardware accelerators effectively improves performance beyond a certain block size threshold.

3.5.3 Overhead Considerations and Break-even Points

Although TitanSSL provides measurable speedups for symmetric and hashing workloads, the software stack introduces some overhead due to memory management and handling of the accelerator interface. This overhead is particularly noticeable for small payloads, where performance gains are partially offset. For example, AES-256-CBC achieves speedups only for payloads larger than approximately 1–2 KiB, while SHA-256 benefits from offloading even at smaller sizes.

The overhead diminishes as payload size increases, allowing the hardware accelerators to dominate and gradually approach a fraction of their theoretical maximum throughput. RSA operations remain limited by communication latency and software handling, highlighting opportunities for improvement in asymmetric workloads.

Importantly, targeted software and system-level optimizations, such as reducing memory management overhead, streamlining user-space interactions with the accelerator and improving offload paths for all cryptographic operations, could further enhance overall performance. These refinements would enable TitanSSL to make fuller use of the hardware accelerators, pushing throughput closer to the theoretical maximum while preserving OpenTitan's strong security guarantees. Although measurable speedups are already observed for symmetric and hashing workloads, the results suggest that further optimizations could unlock the full potential of the hardware, particularly for large payloads, delivering consistently robust and efficient cryptographic processing.

Chapter 4

Architectural Enhancements for Optimized Cryptographic Performance

This chapter extends the baseline performance characterization of OpenTitan's cryptographic accelerators presented in Chapter 3. The TitanSSL software stack demonstrates significant performance improvements, while also highlighting inherent bottlenecks in data transfer and memory access that limit the achievable speedup. Based on these identified limitations, this chapter investigates architectural enhancements. Specifically, we explore the integration of a Direct Memory Access (DMA) engine and Tightly Coupled Data Memory (TCDM) as key mechanisms for optimizing data paths and improving overall cryptographic throughput and efficiency, thereby overcoming these constraints. This approach represents a practical implementation of software-hardware co-design aimed at addressing the identified performance bottlenecks. Finally, we analyze how these enhancements impact system performance and demonstrate how software-hardware co-design, motivated by observed performance gaps, can optimize data paths, improve cryptographic throughput, and unlock the full potential of hardware-accelerated cryptography within OpenTitan-based systems.

4.1 Identifying and Addressing Data Movement Bottlenecks

As evidenced by the detailed performance evaluation presented in the previous chapter, the initial integration of OpenTitan's cryptographic accelerators within the TitanSSL framework, despite providing substantial speedups over software-only implementations, was significantly constrained by I/O and memory access limitations. Preliminary characterization of OpenTitan confirmed that, despite hardware acceleration, its performance on these workloads was severely limited by memory access latency and data movement overheads. Findings, summarized in Table 3.2, revealed

that a dominant proportion of execution cycles were dedicated to memory operations rather than the cryptographic computations themselves. Specifically, no less than 93.78% of the total cycles were consumed by memory accesses, with over 72% specifically required for retrieving data from system DRAM. In stark contrast, only a minimal fraction, at most 3.66%, of cycles were spent waiting for the accelerator to complete its computation before the next data chunk could be pushed into its FIFO. This disproportionate cycle consumption clearly indicates that memory operations were the primary bottleneck.

This pronounced dominance of memory accesses as a performance limiter is directly attributable to the varying costs associated with each memory operation. Accessing OpenTitan's private scratchpad or any accelerator FIFO incurred an average cost of 5.7 cycles, whereas retrieving data from system DRAM cost approximately 23.4 cycles per access. This overhead was compounded by OpenTitan's lack of a dedicated DMA engine. Consequently, the secure Ibex microcontroller was forced to handle all data transfers between system memory and accelerator FIFOs, a CPU-intensive task that further constrained throughput.

Further analysis highlighted a significant underutilization of the cryptographic accelerators. For instance, when data was sourced from system DRAM, the HMAC and SHA-256 accelerators only utilized 16% of their theoretical bandwidth. Even under ideal conditions, with data ideally located in the private scratchpad, utilization only improved to 37% for HMAC and 61% for AES, as shown in Figure 3.1. This stark discrepancy between achieved and theoretical performance is clearly illustrated by the measured cycles per byte: SHA-256 achieved a maximum of 7.88 CpB and AES 16.23 CpB when data was loaded from system DRAM. These values are significantly higher than the theoretical limits, demonstrating that these CPU-intensive data transfer mechanisms fundamentally limited accelerator utilization and throughput, preventing the hardware from reaching its theoretical maximum performance.

Building upon these findings, the performance evaluation of the TitanSSL software stack within a Linux environment, as detailed in Table 3.6, further confirmed the persistent challenges in fully exploiting OpenTitan's accelerators. While TitanSSL demonstrated substantial speedups across both symmetric and hashing workloads, the accelerators remained underutilized relative to their theoretical capabilities. For SHA-256, TitanSSL achieved speedups ranging from 1.3× to 10.5×, with the maximum observed for 128 KiB payloads, while AES-256-CBC attained speedups from 0.41× on very small payloads up to 2.97× for 128 KiB.

Despite these improvements, the effective utilization of the hardware accelerators remained modest. For SHA-256, the highest measured throughput corresponded to approximately 9.35% of the theoretical maximum, while AES-256-CBC reached at most 1.4% of its peak bandwidth. Smaller payloads, particularly those below 1–2 KiB, exhibited very low utilization, often below 0.1–0.5%, due to the dominant impact of system overheads. This overhead arises primarily from the software stack, including both the Linux kernel and OpenSSL library, whose scheduling, memory management, and internal processing introduce non-negligible latency for short-lived operations. As a result, for small payloads, the benefits of hardware acceleration are marginal and cannot be considered representative of realistic cryp-

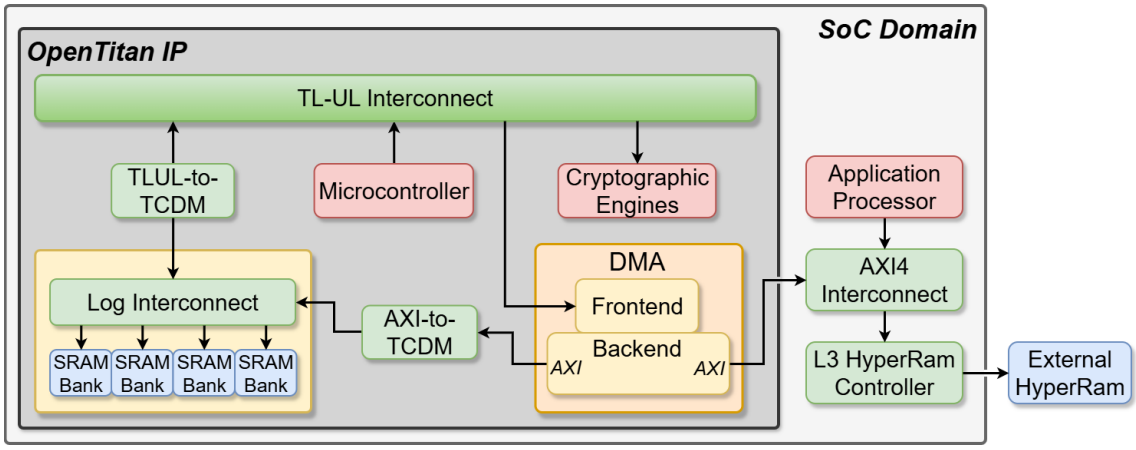

Figure 4.1: Block diagram illustrating the integration of a dedicated Direct Memory Access (DMA) engine and Tightly Coupled Data Memory (TCDM) within the OpenTitan SoC.

tographic workloads. Conversely, larger payloads allow the accelerator to amortize software overhead, progressively approaching a fraction of the theoretical limits. However, even for the largest payloads, the measured performance remains significantly below the hardware peak, highlighting that memory access, data movement, and CPU-mediated interactions with the accelerator FIFOs continue to be the primary bottlenecks.

These findings underscore that, while TitanSSL successfully offloads cryptographic operations to hardware, further optimization, such as reducing memory-copy overhead, improving DMA-like mechanisms, or streamlining the software stack, is required to better exploit OpenTitan's accelerators and approach their theoretical throughput. Overall, this performance gap shows that, despite the efficiency gains from the software-hardware co-design and offloading, communication and data movement overheads, exacerbated by the absence of a dedicated DMA engine, still prevent full utilization of the cryptographic hardware. These observations, critical findings from the software-hardware co-design process, underscore the urgent need for architectural improvements focused on streamlining data movement, directly motivating the enhancements discussed in this chapter.

4.2 Integration of Direct Memory Access and Tightly Coupled Data Memory

Following the findings on data movement and memory access bottlenecks, particularly the CPU-intensive transfers performed by the Ibex microcontroller and the resulting performance stalls, the SoC integrating OpenTitan, as described in Section 2.2, was updated with targeted architectural improvements. Building on insights from the software-hardware co-design process, which exposed the limitations of the original architecture, the Earl Grey configuration was extended to incorporate a dedicated DMA engine and TCDM. These enhancements are essential for optimizing data paths and substantially improving the overall cryptographic throughput and efficiency of the system.

The original OpenTitan design lacked a dedicated DMA engine, forcing the secure Ibex microcontroller to manually manage all data transfers between the system's memory hierarchy and the cryptographic accelerator FIFOs. This reliance on the Ibex core for data movement not only introduced substantial latency but also prevented the accelerators from achieving their theoretical maximum performance due to frequent stalls and underutilization. As detailed in Figure 4.1, the integrated DMA engine is specifically designed to offload these data transfer responsibilities. It is dedicated exclusively to handling data movement across the OpenTitan perimeter, from and towards the external memory hierarchy, thereby freeing the Ibex core to focus on control logic and other security-critical tasks. This capability allows for efficient burst transactions, a significant improvement over the TL-UL interconnect's inability to support such operations, which previously bound performance strictly to the Ibex core's data moving capabilities.

Complementing the DMA, a TCDM was integrated, specifically designed for the accelerators. This TCDM provides a high-bandwidth memory region thanks to its logarithmic interconnect. Its purpose is to offer fast, local memory access for cryptographic operations, minimizing memory access latencies that were a major cause of performance degradation. While the DMA can exploit the TCDM's full bandwidth to fetch data from or store data to external memory, the TCDM ensures that data accessed by the Ibex during cryptographic operations within the OpenTitan perimeter benefits from significantly reduced latency.

This integrated approach, combining a dedicated DMA for efficient bulk data transfer and a TCDM for low-latency local storage, represents a critical architectural enhancement for optimizing the cryptographic engines' data path. By effectively bypassing the Ibex core for large data transfers and providing accelerators with high-speed local memory, these modifications are intended to alleviate the previously identified bottlenecks, aiming to reduce stalls and improve the potential utilization of the cryptographic accelerators.

4.3 Implementation Details and Operational Flow of the Enhanced Architecture

The integration of dedicated hardware components, such as the DMA engine and TCDM, is abstracted from the higher-level software modules in the AP level. This means that TitanSSL Engine continues to process data through its standard APIs (`EVP_DigestUpdate()`, `EVP_DigestInit_ex()`, `EVP_DigestFinal_ex()`, etc.), and the TitanSSL Driver interfaces remain hardware-agnostic, preserving the same functional contract as in the baseline configuration. At the implementation level, all modifications required to exploit the new hardware capabilities are confined to the TitanSSL Firmware. The firmware assumes responsibility for orchestrating efficient data transfers, managing TCDM partitioning, programming DMA transactions, handling synchronization, and updating buffer states. By centralizing these operations within the firmware, the design maintains unaltered high-level software

behavior while delegating the complexity of data movement and accelerator coordination to the underlying platform. This clear separation of concerns is fundamental for achieving high cryptographic throughput while preserving software compatibility and modularity.

To enable fully pipelined, page-granular streaming between system DRAM and the cryptographic accelerators, the TCDM region was explicitly partitioned into four physically independent 4 KiB pages. The partitioning scheme is organized as follows:

- Two Input pages are reserved for input staging, implementing a double-buffering mechanism that allows the DMA to prefetch the next page from external memory while the accelerators process the current page from the TCDM. This arrangement ensures continuous data flow and minimizes idle cycles in the accelerator. These pages are referred to as Slot 0 and Slot 1.
- Two Output pages are allocated for storing accelerator output. These pages serve as staging areas for any accelerator-generated data before it is transferred back to system DRAM. They allow the accelerator to write results independently of the DMA transfer schedule, enabling pipelined operation. The firmware coordinates access to these pages to ensure that data is not overwritten before it is successfully flushed to external memory. These pages are referred to as Slot 2 and Slot 3.

The decision to partition the TCDM into four pages was driven by two key design considerations. First, aligning each TCDM page with the platform's 4 KiB virtual memory page size simplifies the handling of physical-to-virtual address translation and reduces the need for additional logic to manage boundary crossings within the firmware. Second, allocating two pages per direction implements a double-buffering scheme that allows DMA transfers and accelerator computations to overlap, while keeping the scheduling logic simple and deterministic—a critical requirement in a security-sensitive firmware environment.

The overall processing flow for a multi-page input payload can be described through a sequence of logical stages, highlighting the overlap of DMA transfers, accelerator consumption, and output flushing:

1. *Initial DMA prefetch*: The firmware starts by programming the DMA to fetch the first 4 KiB page from system DRAM into Input Slot 0 of the TCDM. The accelerator must wait until this page is fully available to ensure correct state initialization and avoid processing incomplete data.
2. *Pipeline startup*: Once Input Slot 0 contains valid data, the DMA is immediately programmed to transfer the next page into Input Slot 1. Only after initiating this transfer does the accelerator begin processing the data in Input Slot 0. At this point, the accelerator and DMA operate concurrently, with the accelerator consuming the first page while the DMA prefetches the second.
3. *Input steady-state*: After the accelerator finishes processing Input Slot 0, it switches to Input Slot 1, which has now been filled by the DMA. Simultaneously, if a subsequent logical page is available, the DMA reuses Input Slot 0

to fetch this next page (page N+2). Once the DMA transfer to Slot 0 is initiated, the accelerator continues processing the data in Slot 1. This alternating, double-buffered pattern repeats, ensuring that the accelerator always has a page ready for processing while the DMA maintains a continuous prefetch schedule.

4. *Termination*: When no further input pages remain, the pipeline naturally terminates. The last page is processed by the accelerator without additional DMA prefetches, and the firmware waits for any outstanding DMA transfers to complete before signaling overall operation completion to higher-level software.

This page-level, double-buffered pipeline presents several important considerations. First, when processing a single-page input, the entire DMA transaction cost is incurred upfront, as no overlap with accelerator computation is possible. Consequently, the initial latency is proportionally higher for very small payloads, but becomes negligible for multi-page streams, where the steady-state pipeline hides most of the DMA transfer latency.

A similar principle applies to the output path, however a distinction must be made between AES and SHA. AES produces larger output blocks and therefore benefits from DMA-assisted flushing of completed pages back to system DRAM. The firmware implements a double-buffered mechanism for output pages, allowing accelerator write-back to overlap with ongoing input consumption and DMA prefetch operations. In contrast, SHA-256 produces much smaller output, typically a fixed digest size, which does not justify the overhead of DMA transfers. In this case, the accelerator writes results directly to the DRAM, avoiding descriptor setup and DMA completion handling. This approach keeps latency low for small outputs and maintains efficient utilization of the accelerator without unnecessary DMA overhead. The following sequence describes the operational flow for output page handling:

1. *Initial TCDM fill*: The firmware reads data produced by the accelerator and stores it into Output Slot 2 of the TCDM until the slot is full.

2. *Pipeline startup*: Once Output Slot 2 is full, the firmware initiates a DMA transfer to system DRAM while simultaneously reading the next output from the accelerator into Output Slot 3.

3. *Output steady-state*: After Output Slot 3 is filled, the firmware initiates a DMA transfer to system DRAM. If additional output remains, the firmware continues reading accelerator results into Output Slot 2. The process then alternates between Output Slot 2 and Output Slot 3: whenever a slot becomes full, a DMA transfer to system DRAM is started, and the other slot is used to continue receiving output from the accelerator. This alternating sequence continues until all output pages have been processed.

4. *Termination*: Once all output has been processed, the firmware initiates the final DMA transfer of the last slot to system DRAM and waits for its completion.

Table 4.1: Measured throughput and effective accelerator utilization for varying payload sizes on the FPGA-based system. The table reports raw throughput (KiB/s), speedup relative to native OpenSSL and baseline TitanSSL without DMA, and effective use of the OpenTitan hardware accelerators (OTL, OpenTitan Limit) as a percentage of theoretical maximum.

Payload	SHA				AES			
[Byte]	KiB/s	OpenSSL	TitanSSL	OTL	KiB/s	OpenSSL	TitanSSL	OTL
16	1.34	1.55x	0.94x	0.01 %	1.30	0.49x	0.96x	0.01 %
32	3.04	1.37x	1.09x	0.02 %	2.57	0.38x	0.94x	0.01 %
48	4.40	1.32x	1.00x	0.02 %	4.01	0.47x	1.07x	0.02 %
64	5.69	1.31x	0.95x	0.03 %	5.18	0.44x	1.02x	0.02 %
112	9.73	1.41x	0.97x	0.05 %	8.93	0.47x	0.97x	0.04 %
128	11.80	1.43x	1.00x	0.06 %	11.02	0.49x	1.03x	0.05 %
240	17.62	1.19x	0.98x	0.09 %	19.70	0.55x	1.02x	0.08 %
256	19.70	1.36x	1.08x	0.10 %	20.65	0.63x	1.00x	0.08 %
496	39.83	1.57x	1.09x	0.20 %	39.11	0.76x	1.06x	0.16 %
512	37.10	1.40x	0.88x	0.19 %	39.80	0.79x	1.08x	0.16 %
1008	84.30	1.77x	1.08x	0.43 %	66.64	0.90x	0.98x	0.27 %
1024	84.75	1.87x	1.00x	0.43 %	74.54	1.05x	1.07x	0.31 %
2032	173.45	2.49x	1.03x	0.89 %	127.06	1.37x	1.04x	0.52 %
2048	181.34	2.60x	1.07x	0.93 %	128.55	1.33x	1.06x	0.53 %
4080	344.47	3.35x	1.15x	1.76 %	196.34	1.81x	1.15x	0.80 %
4096	310.60	3.20x	1.14x	1.59 %	197.39	1.80x	1.16x	0.81 %
8176	645.75	5.14x	1.23x	3.31 %	289.82	2.57x	1.17x	1.19 %
8192	640.57	5.10x	1.27x	3.28 %	268.60	2.35x	1.20x	1.10 %
16368	1093.62	7.44x	1.22x	5.60 %	329.45	2.82x	1.18x	1.35 %
16384	1081.56	7.34x	1.28x	5.54 %	327.07	2.81x	1.19x	1.34 %
32752	1816.13	11.33x	1.44x	9.30 %	376.52	3.17x	1.18x	1.54 %
32768	1903.97	11.69x	1.58x	9.75 %	386.58	3.26x	1.24x	1.58 %
65520	3020.56	17.81x	1.89x	15.47 %	318.57	2.78x	0.96x	1.30 %
65536	3002.46	17.63x	1.87x	15.37 %	352.44	3.08x	1.12x	1.44 %
131056	3775.14	21.61x	2.08x	19.33 %	404.62	3.52x	1.18x	1.66 %
131072	4014.97	23.08x	2.20x	20.56 %	402.67	3.50x	1.18x	1.65 %

This design ensures that accelerator output is never stalled by slow memory transfers: while one page is being flushed to system DRAM, the firmware can continue reading subsequent results from the accelerator into the other DRAM output page. The double-buffered arrangement maintains a continuous flow of data, maximizing throughput and keeping the accelerator fully utilized. Efficiency gains are most significant for algorithms producing large output blocks, such as AES. For small-output algorithms, including SHA and RSA operations executed on OTBN, DMA pipelining provides little benefit because the total data volume is low; in these cases, direct firmware-mediated writes to DRAM achieve lower latency and reduce implementation complexity.

Correct operation of the pipeline requires careful coordination between DMA transfers, accelerator consumption, and firmware control flow. Slot states are updated atomically to prevent race conditions, DMA completions trigger subsequent transfers, and page-aligned scheduling ensures data ordering and coherence. By leveraging a modest but targeted hardware extension, one DMA engine and a TCDM region, the sequential CPU-bound copy pattern is transformed into a fully pipelined,

page-granular streaming architecture. This design overlaps DMA prefetch, accelerator computation, and output flushes, maximizing throughput while maintaining low implementation complexity and full software compatibility with existing OpenSSL APIs. The resulting firmware solution provides robust, high-performance memory-bound cryptographic operations on embedded platforms.

4.4 Impact on Performance and Efficiency

The architectural enhancements introduced with the integration of a dedicated DMA engine and a TCDM region were evaluated using the same FPGA-based SoC environment and benchmarking methodology described in Section 3.5. This approach ensures a consistent comparison with both the baseline TitanSSL implementation and native OpenSSL, using identical payload sizes, clock frequency, and test harness. The evaluation focuses on memory-bound cryptographic operations, particularly SHA-256 hashing and AES-256-CBC encryption, to quantify the effect of the new DMA/TCDM pipeline on throughput and accelerator utilization.

Table 4.1 reports the measured performance for varying input sizes. The columns indicate raw throughput in KiB/s, relative speedup compared to native OpenSSL and to TitanSSL without DMA, and effective utilization of the theoretical accelerator bandwidth (OTL).

For small payloads (16–128 B), the DMA-assisted pipeline provides limited benefits. With only a single TCDM page processed, the full cost of the DMA transaction is incurred upfront, without overlap with accelerator computation. As a result, throughput improvements over the baseline TitanSSL implementation are modest. For instance, SHA-256 throughput for a 128 B payload reaches 11.8 KiB/s, corresponding to a 1.43× speedup over native OpenSSL and approximately 1.0× relative to TitanSSL without DMA. Similarly, AES-256-CBC achieves 11.0 KiB/s, yielding 0.49× over OpenSSL and 1.03× over the previous TitanSSL. Effective hardware utilization remains extremely low at this scale, around 0.05–0.06% for SHA-256 and 0.04–0.05% for AES, reflecting minimal occupation of the accelerator.

As payload sizes increase beyond 1–2 KiB, the advantages of the page-granular, double-buffered pipeline become apparent. For SHA-256, throughput scales almost linearly in the steady-state regime, as DMA prefetches and TCDM buffering effectively hide memory transfer latency. At 16 KiB, measured throughput reaches approximately 1081 KiB/s, corresponding to a speedup of 7.3× over OpenSSL and 1.28× over baseline TitanSSL, with hardware utilization values around 5.5–5.6%. For 128 KiB payloads, throughput climbs to 4015 KiB/s, speedup over OpenSSL reaches 23×, over TitanSSL without DMA 2.2×, and hardware utilization rises to 20.6%, demonstrating that the pipeline keeps the accelerator fully utilized for memory-bound workloads.

AES-256-CBC shows a similar trend, though gains are more moderate due to the reduced impact of memory latency on overall computation. For payloads larger than 4 KiB, throughput ranges between 197 KiB/s and 404 KiB/s, speedups over baseline TitanSSL reach 1.15–1.24×, and over OpenSSL 1.8–3.2×. Effective accelerator utilization, as a percentage of theoretical bandwidth, remains lower than SHA-256, between 0.8–1.6% for mid-size payloads and 1.3–1.6% for the largest transfers, reflecting the fact that AES computation is less memory-bound and the pipeline is not fully saturated.

Across both algorithms, the DMA/TCDM enhancements consistently improve throughput relative to both native OpenSSL and the baseline TitanSSL. The steady-state, double-buffered design allows overlapping of DMA prefetch, accelerator computation, and output flushing, minimizing idle cycles and enabling continuous data flow. Small payloads see limited improvement due to fixed DMA overhead and software stack latency, while multi-KiB streams fully exploit the hardware pipeline.

A closer examination of effective accelerator utilization across payloads reveals potential bottlenecks beyond the DMA/TCDM pipeline, such as internal throughput limits within the accelerators, contention on input/output ports, and TCDM arbitration delays. Other hardware-level factors may also constrain the full exploitation of available resources. Additional constraints may arise at different levels of the TitanSSL stack, such as the Linux scheduler, which can influence the timing of tasks; the kernel-space driver, responsible for managing memory layout; and the user-space API interactions, which mediate requests to the accelerators. Each of these layers can affect the effective use of hardware resources. Examining these layers individually allows for a deeper understanding of the factors limiting utilization, whether inherent to the hardware or arising from software interactions, and can highlight opportunities to better overlap computation and data movement, providing a comprehensive view of overall system performance.

Overall, the results demonstrate that the DMA/TCDM enhancements deliver a robust, low-complexity mechanism for accelerating memory-bound cryptographic operations on embedded platforms, without requiring changes to OpenSSL APIs or external drivers, while highlighting areas for further hardware-level optimization to approach theoretical throughput limits.

4.4.1 TitanSSL Firmware Cycle-Level Performance Analysis

To gain a deeper understanding of the timing behavior of SHA-256 and AES-256-CBC operations on OpenTitan, we performed a cycle-level analysis restricted to the TitanSSL firmware execution. While cryptographic tests are launched from Linux as usual, this analysis isolates and measures only the cycles spent on internal firmware tasks, providing a detailed view of the pipeline behavior without including user-space, kernel, or OS overhead. Cycle-level data were collected for both the baseline and DMA/TCDM-optimized configurations, allowing a direct comparison of their internal execution characteristics and the identification of the main performance contributors within each phase.

Table 4.2: Cycle-level measurements of SHA-256 and AES-256-CBC on the TitanSSL Firmware for 16 KiB and 64 KiB payloads. The table details per-phase and per-task breakdowns (`INIT`, `UPDATE`/`DO CIPHER`, `FINALIZE`/`CLEANUP`), total cycles per phase, effective throughput, and approximate utilization of the theoretical accelerator bandwidth (OTL). Totals are provided for each phase and the full operation, comparing DMA/TCDM-enhanced and baseline (no DMA) scenarios with percentage speed-up.

Phase	**Task**	**DMA/TCDM Enhanced**			**Baseline (No DMA)**			**Speed up [%]**
		cycles [#]	[MiB/s]	OT Limit %	cycles [#]	[MiB/s]	OT Limit %	
SHA-256 - 16 KiB Payload								
INIT		929	-	-	929	-	-	-
UPDATE	process data	31.753	12,30	64,50	160.820	2,43	12,73	5,06
	dma trx	2.000	-	-	-	-	-	-
	logic	997	-	-	3.164	-	-	-
	TOT	**34.750**	**11,24**	**58,93**	**163.984**	**2,38**	**12,49**	**4,79**
FINALIZE		1.638	-	-	1.611	-	-	-
TOT		**37.317**	**10,47**	**54,88**	**166.524**	**2,35**	**12,30**	**4,46**
SHA-256 - 64 KiB Payload								
INIT		928	-	-	933	-	-	-
UPDATE	process data	168.292	9,28	48,68	642.687	2,43	12,75	3,82
	dma trx	5.819	-	-	-	-	-	-
	logic	997	-	-	5.089	-	-	-
	TOT	**175.108**	**8,93**	**46,78**	**647.776**	**2,41**	**12,65**	**3,70**
FINALIZE		1.634	-	-	1.624	-	-	-
TOT		**177.670**	**8,79**	**46,11**	**650.333**	**2,40**	**12,60**	**3,66**
AES-256-CBC - 16 KiB Payload								
INIT		3.556	-	-	3.646	-	-	-
DO CIPHER 1	process data	851.657	0,46	1,92	1.136.440	0,34	1,80	1,33
	dma trx	5.320	-	-	-	-	-	-
	logic	14.100	-	-	16.660	-	-	-
	TOT	**871.077**	**0,45**	**1,90**	**1.153.100**	**0,34**	**1,78**	**1,32**
DO CIPHER 2	process data	728	-	-	931	-	-	-
	dma trx	1.216	-	-	-	-	-	-
	logic	1.870	-	-	2.952	-	-	-
	TOT	**3.814**	**-**	**-**	**3.883**	**-**	**-**	**-**
CLEANUP		477	-	-	475	-	-	-
TOT		**878.924**	**0,44**	**1,87**	**1.161.104**	**0,34**	**1,76**	**1,32**
AES-256-CBC - 64 KiB Payload								
INIT		3.654	-	-	3.659	-	-	-
DO CIPHER 1	process data	3.676.853	0,42	1,78	4.547.124	0,34	1,44	1,24
	dma trx	12.508	-	-	-	-	-	-
	logic	14.104	-	-	21.208	-	-	-
	TOT	**3.703.465**	**0,42**	**1,77**	**4.568.332**	**0,34**	**1,44**	**1,23**
DO CIPHER 2	process data	728	-	-	931	-	-	-
	dma trx	1.216	-	-	-	-	-	-
	logic	1.891	-	-	2.959	-	-	-
	TOT	**3.853**	**-**	**-**	**3.890**	**-**	**-**	**-**
CLEANUP		482	-	-	479	-	-	-
TOT		**3.711.454**	**0,42**	**1,77**	**4.576.360**	**0,34**	**1,43**	**1,23**

Cycle counts were measured on the FPGA using the Ibex-provided function `ibex_mcycle_read()`, which reads the current CPU cycle counter atomically. Measurements were taken for two representative payload sizes (16 KiB and 64 KiB), covering all main phases of the operations: `INIT`, `UPDATE`, and `FINALIZE` for SHA-256; `INIT`, `DO CIPHER`, and `CLEANUP` for AES. Within each phase, tasks such as data processing, DMA transfers, and accelerator logic were timed separately to identify the primary contributors to execution time.

Table 4.2 reports the detailed cycle counts and derived performance metrics for both the DMA/TCDM-enhanced and baseline (no DMA) configurations.

SHA hardware transfer limits

For SHA-256, the `UPDATE` phase dominates the overall execution time, with data processing representing the largest share of cycles. The introduction of DMA and TCDM buffering leads to a speedup of approximately 4.5 times for the 16 KiB payload and 3.7 times for the 64 KiB payload, reducing the total cycle count from 166.5k to 37.3k and from 650k to 177.7k, respectively. Despite the shorter execution time, accelerator utilization increases significantly, from 12.3% to 54.9% for 16 KiB and from 12.6 to 46.1% for 64 KiB. This confirms that the DMA/TCDM pipeline effectively overlaps data movement and computation, keeping the accelerator active for a larger fraction of time and reducing idle periods caused by memory stalls. These results are consistent with the throughput measurements reported in Table 4.1, where large-payload SHA transfers reach up to 15–20% of the theoretical OpenTitan limit, compared to around 9–10% in the baseline configuration. When comparing these cycle-level measurements with the end-to-end throughput results obtained from Linux, it becomes evident that a significant fraction of performance is lost due to software and system overheads. Specifically, for a 16 KiB payload, the firmware-only configuration achieves up to 54% of the theoretical accelerator bandwidth for SHA-256, whereas the same operation reaches only about 15% when executed through the full Linux stack. This suggests that approximately 70% of the available hardware performance is hidden by the combined overhead of user-space mediation, kernel-level memory management, and scheduling latency on the application processor, with only about 30% of the firmware-level throughput observable at the system level.

To better understand the remaining limitations of the SHA-256 accelerator, we performed a cycle-level analysis of the firmware execution. These measurements reveal that most of the execution time is spent in the `UPDATE` phase, specifically within the data-handling loop that feeds the HMAC accelerator. To investigate why SHA-256 does not reach its theoretical maximum throughput, a micro-benchmark was executed on the Ibex core using ModelSim, focusing on the basic loop of a `memcpy` transferring data from DRAM or TCDM to the accelerator FIFO. The assembly sequence, shown in Listing 4.1, reports per-instruction cycle counts.

Listing 4.1: Assembly sequence executed by the Ibex core as the basic loop of a `memcpy` to transfer data from DRAM or TCDM to the HMAC accelerator FIFO, showing per-instruction cycle counts (DRAM: 37 cycles, TCDM: 10 cycles).

```
1 c.lw    x14, 0(x14)          # DRAM: 30 cycles    TCDM: 3  cycles
2 add     x15, x10, x13        # DRAM: 1  cycle     TCDM: 1  cycle
3 c.addi  x13, 4               # DRAM: 1  cycle     TCDM: 1  cycle
4 c.sw    x14, 0(x15)          # DRAM: 3  cycles    TCDM: 3  cycles
5 bltn    x13, x16, <addr>     # DRAM: 1  cycle     TCDM: 1  cycle
6 add     x14, x11, x13        # DRAM: 1  cycle     TCDM: 1  cycle
7
8 # Totals:                     DRAM: 37 cycles    TCDM: 10 cycles
```

This experiment shows that transferring a single 32-bit word from DRAM to the accelerator FIFO requires 37 cycles, while the same operation from TCDM takes 10 cycles. Each transfer involves six instructions in the basic `memcpy` loop, and the per-word latency is dominated by the time required to move the data into the accelerator FIFO. The FIFO, implemented alongside the HMAC accelerator, imposes a physical limit on how quickly new data can be written and processed. Even with the DMA/TCDM pipeline fully active, this internal transfer latency constrains the maximum achievable throughput. In other words, the bottleneck is inherently hardware-bound: the FIFO depth and the accelerator's processing rate establish a hard ceiling that software optimizations cannot overcome. This explains why SHA-256, despite being memory-bound, does not reach the theoretical maximum throughput predicted by idealized DMA-only models.

AES hardware transfer limits

AES-256-CBC exhibits a more computation-bound profile. The `DO CIPHER 1` phase dominates the total execution time, while DMA transfers and control logic contribute only a minor fraction of the total cycles. In this case, the DMA/TCDM integration yields modest performance improvements, with a speedup of about 1.3 times for 16 KiB and 1.2 times for 64 KiB. Accelerator utilization increases only slightly, from 1.76% to 1.87% and from 1.43% to 1.77% respectively. The additional `DO CIPHER 2` call, triggered by the user-space API to handle padding and finalization, introduces additional overhead and slightly increases the total execution time of the AES operation. This indicates that AES performance on OpenTitan is primarily limited by internal computation latency rather than by memory bandwidth, which is consistent with the theoretical accelerator bandwidth values reported in Table 4.1, where utilization remains around 1–2%.

General Firmware Insights

A closer look at the firmware execution across both algorithms highlights general trends and overhead patterns that emerge regardless of the specific cryptographic operation. For both algorithms, the `INIT`, `FINALIZE`, and `CLEANUP` phases contribute negligibly to total runtime at the firmware level, indicating that setup and teardown

costs are minor for multi-KiB payloads. However, each phase corresponds to a separate user-space API call, introducing additional overhead when the operation is executed through the full software stack. This explains why small messages exhibit limited speedup during TitanSSL execution, as the fixed cost of multiple user-space interactions dominates total execution time.

At the firmware level, this fine-grained analysis reveals how the distribution of cycles across tasks maps to the TitanSSL software stack and hardware layers. Effective utilization depends not only on the DMA/TCDM pipeline itself but also on secondary factors such as accelerator throughput limits, I/O port contention, TCDM arbitration latency, and higher-level software effects such as Linux scheduling and memory management overhead. These measurements identify where computation or data movement bottlenecks occur and highlight opportunities for deeper overlap between processing and communication.

Overall, the cycle-level analysis confirms the throughput trends observed earlier. SHA-256, being a memory-bound algorithm, benefits significantly from the DMA/TCDM co-design, achieving near-optimal accelerator utilization for large payloads. AES-256-CBC, on the other hand, remains primarily compute-bound and therefore less affected by improvements in memory transfer efficiency. Together, these findings demonstrate that the proposed DMA/TCDM pipeline provides an effective and low-overhead mechanism for improving hardware accelerator efficiency in embedded cryptographic workloads.

4.4.2 System-Level Performance Analysis and Latency Breakdown

The system-level profiling provides a detailed view of how execution time is distributed across the TitanSSL software stack, highlighting the overheads introduced beyond the internal firmware execution. The breakdown includes contributions from the user-space TitanSSL Engine, the kernel driver, and the communication and coordination with the firmware. As discussed in the previous section, firmware-level cycle counts were previously measured using FPGA benchmarks; in this context, these numbers are used to quantify the internal execution cost without being double-counting on the Engine or Driver totals.

Benchmarking was conducted on the FPGA setup with the DMA/TCDM-enhanced configuration, focusing on a 16 KiB payload. SHA-256 was selected as a representative memory-bound algorithm, while AES-256-CBC exhibits similar performance characteristics, with minor differences due to padding management and handling of multiple output buffers and initialization metadata such as keys and IVs. Cycle counts for user-space and kernel-level operations were collected using the CVA6 processor's high-resolution hardware cycle counter, which increments on each CPU cycle and can be read atomically. This allows precise measurement of the cycles associated with each task in user-space and kernel-level operations.

Table 4.3: Breakdown of execution cycles for a 16 KiB SHA-256 operation across the TitanSSL software stack. The table distinguishes contributions from the user-space Engine, the kernel Driver, and the OpenTitan Firmware. Cycle counts are hierarchical: the Engine measurement inherently includes both driver and firmware-mediated tasks, while the Driver column isolates kernel-level operations but still involves firmware interaction. Firmware cycles are reported for reference and are not additive.

Phase			**Cycles**		
Engine	**Driver**	**Firmware**	**E**	**D**	**F**
		INIT			
`set_digests()`	-	-	8.404	-	-
`sha256_init()`	-	-	116.992	-	-
-	`ioctl()`	-	-	1.252	-
-	data layout	-	-	3.824	-
-	`fence.i`	-	-	15.564	-
-	-	firmware init	-	-	929
-	`wait_event_intr`	-	-	32.271	-
-	others	-	-	45.234	-
		Total	**125.396**	**98.145**	**929**
		UPDATE			
`sha256_update()`	-	-	299.843	-	-
-	`ioctl()`	-	-	1.004	-
-	data layout	-	-	71.005	-
-	`fence.i`	-	-	15.817	-
-	-	firmware update	-	-	34.750
-	`wait_event_intr`	-	-	180.438	-
-	others	-	-	15.214	-
		Total	**299.843**	**281.554**	**34.750**
		FINALIZE			
`sha256_final()`	-	-	127.420	-	-
-	`ioctl()`	-	-	1.013	-
-	data layout	-	-	14.136	-
-	`fence.i`	-	-	16.216	-
-	-	firmware finalize	-	-	1.638
-	`wait_event_intr`	-	-	28.575	-
-	others	-	-	48.443	-
`sha256_cleanup()`	-	-	2.066	-	-
		Total	**129.486**	**108.383**	**1.638**
		CUMULATIVE TOTAL	**554.725**	**488.082**	**37.317**

The collected data provide a hierarchical non-additive breakdown of cycles across the TitanSSL software stack. Table 4.3 summarizes these results. It is important to note that the TitanSSL Engine measurement includes the driver and firmware execution, whereas the driver column isolates kernel-level mediation but still reflects the time spent interacting with the firmware. This methodology allows identifying precisely where cycles are lost without overestimating the contributions of each layer.

In the `INIT` phase, the bulk of cycles in the TitanSSL Engine is associated with the preparation of internal data structures and the setup of input/output buffers. Within this execution, kernel-level operations are responsible for mapping buffers from virtual to physical memory, enforcing memory locks, and coordinating with the firmware, including necessary synchronization. The measured cycle contribution of the firmware itself is comparatively minor, but the combined overhead of buffer preparation, kernel mediation, and engine coordination accounts for the majority of total latency in this phase. This decomposition clarifies that performance is primarily lost in tasks related to buffer management and kernel-mediated coordination rather than in the intrinsic computation performed by the firmware.

During the `UPDATE` phase, which represents the main computational workload for SHA-256, the firmware is the core of the intrinsic computation, but its absolute cycle count is much smaller than the Engine total because Engine includes driver-mediated coordination and overheads. However, user-space and kernel interactions introduce additional cycles due to repeated API calls, IOCTL invocations, and memory transfers that coordinate the movement of data to and from the accelerator. These contributions are particularly significant in multi-KiB payloads, where the engine must manage multiple buffer segments while the kernel driver performs address translation and locking for each segment. The system-level overhead explains why end-to-end throughput measured on Linux remains substantially below the hardware limits observed at the firmware level, despite the DMA/TCDM optimizations.

In the `FINALIZE` phase, the firmware operations are short, yet measurable latency arises from user-space API calls and driver-level tasks. This includes finalization of buffers, completion of any remaining DMA transfers, and synchronization of the accelerator state with the system. While the absolute number of cycles is smaller than in `UPDATE`, the relative contribution of system-level operations becomes more prominent, highlighting how small, fixed overheads in user-space and kernel mediation can dominate short phases.

Taken together, this decomposition demonstrates that the apparent dominance of the TitanSSL Engine in system-level measurements is due to the hierarchical inclusion of driver and firmware interactions, not only the intrinsic computation of the firmware. By isolating the contributions of kernel-level tasks such as buffer mapping, memory locking, and synchronization, it is possible to identify the precise points at which cycles are lost. Comparing the Engine total with the isolated firmware cycles, we estimate that approximately 70% of potential accelerator throughput is obscured by user-space and kernel overheads, leaving only about 30% of the raw hardware-level performance visible in practice.

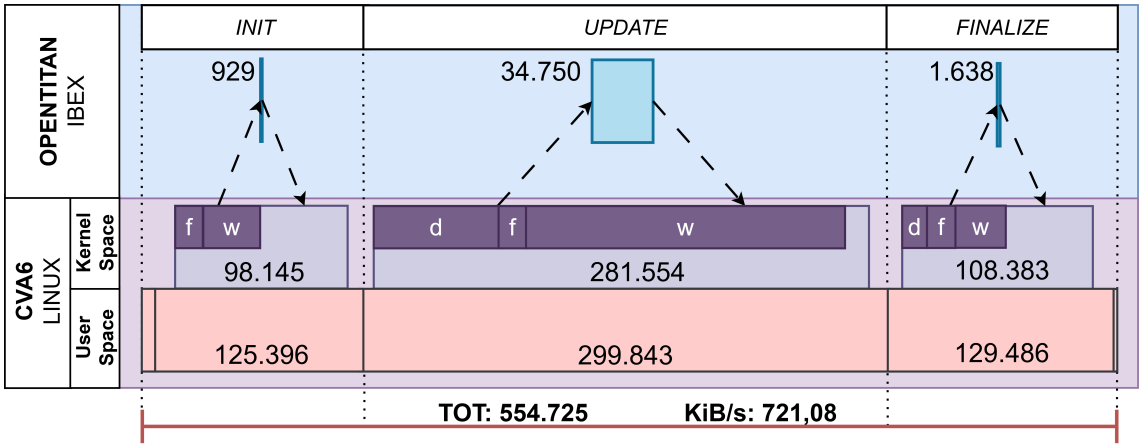

Figure 4.2: Visual representation of the hierarchical distribution of execution cycles for a 16 KiB SHA-256 operation across the TitanSSL software stack. The figure is drawn to scale, and the annotated numbers indicate the measured cycle counts for each task within the Engine, Driver, and Firmware layers. Within the Driver layer, the rectangular blocks correspond to the most cycle-consuming phases, also represented to scale. The labels `f`, `w`, and `d` denote, respectively, `fence.i` operations, `wait_event_interruptible` calls, and data-handling activities such as paging or memory management.

These results are also illustrated visually in Figure 4.2, which shows the hierarchical distribution of execution time across the Engine, Driver, and Firmware layers.

This analysis complements the cycle-level analysis of DMA/TCDM-enhanced firmware performance by linking the observed hardware efficiency to system-level interactions. While DMA/TCDM optimizations enable near-optimal accelerator utilization for memory-bound workloads such as SHA-256, the effective system throughput remains constrained by software stack overheads. This comprehensive view provides a clear mapping of latency across layers, identifies the dominant contributors at each stage, and highlights specific tasks, such as buffer management, memory translation, and synchronization, as targets for future optimization in order to better exploit the underlying hardware resources.

4.5 Collaborative Contributions and Broader Implications

The detailed investigations and architectural advancements presented throughout this chapter highlight a profoundly collaborative effort aimed at transforming OpenTitan into a more efficient and capable secure embedded platform. This section synthesizes the collaborative nature of these hardware optimizations, specifically the integration of a dedicated DMA engine and TCDM, and elucidates their broader implications for the overall OpenTitan ecosystem and the advancement of secure RISC-V SoC design.

4.5.1 Synergistic Co-design and Performance Realization

The analysis began with a meticulous performance characterization of OpenTitan's cryptographic accelerators, which, despite offering substantial speedups over software-only implementations, revealed a critical limitation: performance was overwhelmingly constrained by data movement and memory access latencies. The results showed that a dominant proportion of execution cycles (up to 93.78%) were consumed by memory operations, particularly data retrieval from system DRAM (over 72%), rather than the cryptographic computations themselves (Table 3.2). This stark discrepancy underscored a significant underutilization of the accelerators, with figures as low as 16% of theoretical bandwidth for HMAC and SHA-256 when data resided in system DRAM.

This precise identification of performance bottlenecks, stemming from a comprehensive software-hardware co-design process, directly motivated the architectural enhancements detailed in this chapter. The integration of a dedicated DMA engine and TCDM was a collaborative response to these challenges. The DMA offloads the Ibex microcontroller from manual data transfers, enabling efficient burst transactions, while the TCDM provides high-bandwidth, low-latency local memory access for cryptographic operations. The resulting double-buffered, page-granular pipeline, designed to overlap DMA prefetching, accelerator computation, and output flushing, was implemented within the TitanSSL firmware. This innovative approach effectively transforms sequential, CPU-bound data movement into a parallel, streaming architecture.

The impact on performance has been substantial. For memory-bound workloads like SHA-256, the DMA/TCDM pipeline yielded speedups of up to 23 times over native OpenSSL and 2.2 times over the baseline TitanSSL for 128 KiB payloads. This also significantly boosted accelerator utilization, reaching up to 20.6% for SHA-256, a marked improvement from the baseline. While AES-256-CBC, being more compute-bound, showed more moderate gains (up to 3.5 times over OpenSSL), the enhancements still provided consistent improvements in throughput. These results underscore the effectiveness of targeted hardware augmentation, informed by empirical software performance analysis, in unlocking the true potential of cryptographic accelerators within embedded systems.

4.5.2 Broader Implications for the OpenTitan Ecosystem

The contributions arising from these architectural and firmware optimizations extend beyond mere performance metrics, carrying significant implications for the broader OpenTitan ecosystem and the future of secure hardware:

- *Realized Cryptographic Potential*: By effectively mitigating memory access bottlenecks, these efforts have brought OpenTitan's cryptographic accelerators closer to their theoretical performance limits for memory-bound workloads. This makes OpenTitan a more compelling and practical solution for offloading computationally intensive security tasks in real-world applications. The ability

to perform high-throughput cryptographic operations efficiently strengthens OpenTitan's role as a robust RoT.

- *Validated Co-design Methodology*: This work exemplifies the critical importance of a tight software-hardware co-design methodology. Starting from detailed software performance characterization to identify precise bottlenecks, and subsequently engineering targeted hardware solutions, provides a powerful blueprint for future platform optimizations. This iterative process ensures that architectural investments directly address empirically observed limitations.

- *Foundation for Future Enhancements*: The DMA/TCDM architecture provides a more efficient data path that can benefit a wide array of future cryptographic algorithms and security features integrated into OpenTitan. It establishes a more robust foundation for scaling cryptographic performance and integrating more complex security primitives without incurring prohibitive data transfer overheads.

- *Strengthening Open-Hardware Security*: By openly addressing and resolving fundamental performance challenges, this work contributes to the maturation and credibility of open-source hardware projects like OpenTitan. Demonstrating that an open platform can achieve high performance and efficiency through systematic engineering reinforces the value proposition of transparent, verifiable hardware designs for critical security functions. The detailed analysis of system-level overheads, indicating that approximately 70% of potential accelerator throughput is currently obscured by software stack latency, also highlights clear areas for future collaborative research and optimization within the Linux kernel and user-space layers to further maximize hardware utilization.

In summary, the collaborative endeavor to integrate DMA and TCDM, driven by a deep understanding of OpenTitan's performance characteristics, not only delivers significant practical benefits in terms of cryptographic throughput and efficiency but also provides valuable insights into the optimal co-design of secure embedded systems. These efforts solidify OpenTitan's position as a leading open-source platform for robust and high-performance hardware-based security.

Chapter 5

Enhancing System Integrity and Forward-Looking Cryptography

5.1 TitanCFI: Strengthening System Integrity with Control-Flow Integrity

The increasing prevalence of sophisticated software attacks, particularly code-reuse attacks like Return-Oriented Programming and Jump-Oriented Programming, necessitates robust control-flow integrity mechanisms in modern embedded systems. As established in Section 2.3, these attacks subvert a program's intended execution path, often in memory-unsafe languages like C, leading to arbitrary code execution or full system compromise. TitanCFI addresses these limitations by introducing a novel approach that leverages the security and isolation properties of OpenTitan as a RoT to enforce CFI policies. This enables software-defined control-flow policies without requiring custom hardware monitors or modifications to the host core pipeline.

5.1.1 Motivation for RoT-Based Control-Flow Integrity

Traditional CFI enforcement mechanisms in embedded systems often face limitations due to resource constraints and the inherent vulnerabilities of software-only solutions. Purely software-based CFI approaches can introduce significant runtime overhead, while hardware-based solutions may require custom ISA modifications or ad hoc binary toolchains, complicating deployment and maintenance. Furthermore, the security metadata crucial for CFI enforcement can itself become a vulnerability if stored in potentially compromised memory regions outside a trusted hardware boundary.

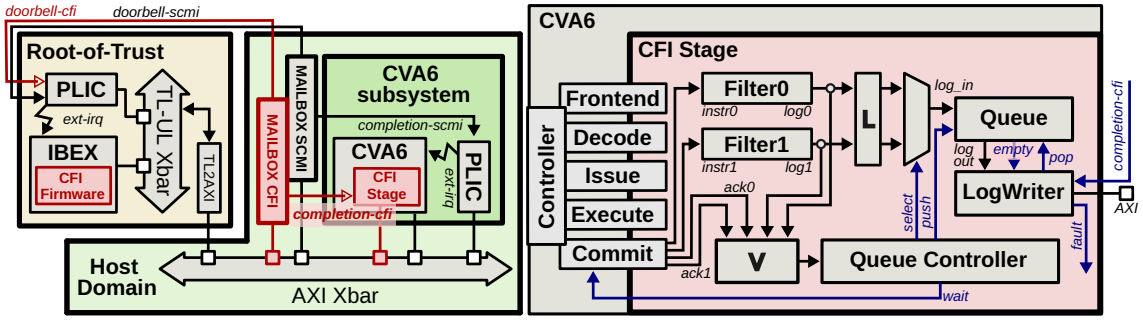

Figure 5.1: Architecture of TitanCFI. The diagram provides a detailed view of the proposed architectural modifications, highlighting the integration and interaction between the SoC components (shown on the left) and the CVA6 core (depicted on the right), emphasizing the design changes introduced to support secure execution and control-flow integrity mechanisms.

While existing state-of-the-art CFI solutions such as FIXER and DExIE offer different approaches, they often involve trade-offs in terms of required toolchain modifications, protection of legacy binaries, or hardware overhead [68, 69]. For instance, FIXER is an ISA-based CFI enforcement module that employs a shadow stack and a jump table. It signals control-flow events using custom opcodes, which, despite providing lightweight execution overhead, requires access to a custom toolchain and the rebuilding of all source code that needs protection. This requirement significantly limits its compatibility with legacy binaries and standard development workflows [96].

In contrast, DExIE implements CFI on the protected host core through a hardware Enforcement Finite State Machine coupled with a shadow stack, representing a hardware monitor approach [96]. While DExIE uses custom-designed hardware IP to enforce its CFI policy, minimizing area occupancy and runtime overhead by observing instruction streams, it inherently requires specific custom hardware components and a dedicated interface to the main core [96]. Additionally, the authors note that DExIE can lead to a reduction in the clock frequency of the cores when interfaced with its security monitor [96]. These limitations highlight a lack of flexibility in dynamically updating CFI enforcement policies, often requiring new hardware designs for any significant policy changes.

TitanCFI overcomes these limitations by integrating CFI enforcement directly within the OpenTitan RoT. This approach leverages OpenTitan's tamper-proof memory and cryptographic accelerators to secure CFI metadata and execute enforcement policies in an isolated environment. Unlike solutions that modify the host core pipeline or require custom toolchains, TitanCFI maximizes hardware resource reuse and is compatible with standard toolchains, allowing protection of legacy binaries without recompilation. By leveraging the RoT's trusted execution environment, TitanCFI enables software-defined CFI policies to be implemented, offering superior flexibility without the need for custom hardware monitors or invasive modifications to the host core pipeline. This makes TitanCFI a highly adaptable and secure solution for strengthening system integrity against control-flow hijacking attacks.

Table 5.1: Detailed breakdown of the instruction counts, cycle counts, and relative contributions for implementing the return address protection policy in OpenTitan firmware. The table compares three firmware strategies, IRQ-based, Polling, and Optimized, highlighting the cycles spent in logic operations, memory accesses within the RoT, and memory accesses in the SoC. Total cycles and their percentages are reported for both function call (`CALL`) and return (`RET`) operations, providing a clear view of the overheads and benefits of each approach in enforcing CFI.

	Op.		Instructions [#]			Cycles [#]			Cycles [%]		
			IRQ	CFI	TOT	IRQ	CFI	TOT	IRQ	CFI	TOT
IRQ	CALL	Logic	8	15	23	59	27	86	23	10	33
		Mem. RoT	14	5	19	74	28	102	29	9	38
		Mem. SoC	2	4	6	22	48	70	10	19	29
		TOT	24	24	48	155	103	258	62	38	100
	RET.	Logic	8	15	33	59	45	104	21	16	37
		Mem. RoT	14	5	19	74	28	102	27	10	37
		Mem. SoC	2	4	6	22	48	70	9	17	26
		TOT	24	34	58	155	121	276	57	43	100
Polling	CALL	Logic	–	15	15	–	27	27	–	26	26
		Mem. RoT	–	5	5	–	28	28	–	27	27
		Mem. SoC	–	4	4	–	48	48	–	47	47
		TOT	–	24	24	–	103	103	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	37	37
		Mem. RoT	–	5	5	–	28	28	–	23	23
		Mem. SoC	–	4	4	–	48	48	–	40	40
		TOT	–	34	34	–	121	121	–	100	100
Optimized	CALL	Logic	–	15	15	–	27	27	–	42	42
		Mem. RoT	–	5	5	–	5	5	–	08	08
		Mem. SoC	–	4	4	–	32	32	–	50	50
		TOT	–	24	24	–	64	64	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	55	55
		Mem. RoT	–	5	5	–	5	5	–	06	06
		Mem. SoC	–	4	4	–	32	32	–	39	39
		TOT	–	34	34	–	82	82	–	100	100

5.1.2 TitanCFI Architecture and Design

The architecture of TitanCFI, based on the RISC-V host core and the OpenTitan RoT described in Section 2.2, involves a collaborative mechanism between the two components. The core idea is to modify the commit stage of the protected host core to stream control-flow instructions to the RoT. The overall architecture, showing the interaction between the host core and the OpenTitan, is depicted in Figure 5.1.

The primary components and their interactions are:

- *Host Core Modifications*: The host core's commit stage is augmented to detect and stream control-flow instructions, such as function calls, returns, and indirect jumps, to a dedicated CFI Queue. This modification is minimal, focusing on observation rather than direct enforcement.
- *CFI Mailbox*: A specialized communication channel, the CFI Mailbox, acts as the interface between the CFI Queue on the host side and the OpenTitan RoT. It ensures secure and reliable transmission of control-flow events. The CFI Mailbox operates by transmitting commit logs from the CFI Queue to the RoT using AXI transactions, setting a `Doorbell` interrupt to signal the RoT, and waiting for a `Completion` signal.
- *OpenTitan RoT Firmware*: Within OpenTitan, custom firmware implements the desired CFI policy. This firmware resides within the interrupt service routine dedicated to the CFI Mailbox. It processes incoming control-flow events, enforces the policy, for example shadow-stack protection, and reports any security violations.
- *Policy Enforcement*: TitanCFI implements CFI enforcement policies in C code embedded within the RoT firmware. For example, a shadow-stack policy involves pushing expected return addresses onto a shadow stack for call instructions and verifying them against actual return addresses for return instructions. Any mismatch triggers a security violation.

This design ensures that the critical CFI enforcement logic resides within the trusted and isolated environment of OpenTitan, making it resilient to attacks that might compromise the host OS or application software.

5.1.3 Implementation Details and Firmware Analysis

The implementation of TitanCFI involves detailed firmware development within OpenTitan. The CFI policy, written in C, follows a common structure consisting of IRQ entry, policy enforcement, and IRQ exit. To analyze the overhead of implementing CFI enforcement policies in the RoT, different firmware implementations were evaluated: IRQ firmware, Polling firmware, and an Optimized RoT firmware. The latency required for CFI enforcement, specifically the average number of cycles to process a function call and return, was benchmarked for these implementations. These crucial details are summarized in Table 5.1, which details the cycle counts for different operations and firmware versions.

This table highlights the performance overheads and benefits of the proposed optimizations.

- *IRQ Firmware*: Each event triggers an interrupt to the Ibex core. Wake-up takes 45 cycles, entry/exit adds 105 cycles, and `CALL`/`RET` average 267 cycles, mainly due to memory accesses.
- *Polling Firmware*: Reduces IRQ overhead via busy-wait polling without hardware changes. Average latency drops to 112 cycles (-58%), with `CALL` at 77 and `RET` at 95 cycles.
- *Optimized RoT Firmware*: Architectural improvements reduce memory access latency (1 cycle internal, 8 cycles SoC). Return address protection latency drops to 73 cycles (-70%), with `CALL` at 43 and `RET` at 61 cycles.

The analysis clearly demonstrates that integrating CFI enforcement into the RoT introduces some latency. Both firmware-level and architectural optimizations can drastically reduce this overhead. The number of instructions required for the CFI policy logic ranges between 48 and 58 opcodes. This confirms the practicality of implementing return address protection in software within the RoT. This software-based approach offers a flexible and cost-effective alternative to fully customized hardware modules. It is effective provided the RoT architecture is adequately optimized for instructions per cycle.

The dedicated mechanism, built upon the host core modifications, actively manages the CFI Queue and its communication with the CFI Mailbox. This mechanism retrieves commit logs from the CFI Queue, divides them into data chunks, and initiates AXI transactions to transmit these chunks to the CFI Mailbox. It sets a `Doorbell` interrupt to signal the RoT that new data is available for processing. Upon receiving a `Completion` signal from the RoT firmware, the host-side mechanism reads the result of the CFI enforcement check from the Mailbox. It triggers an exception if a control flow violation is detected. This mediated process ensures that hardware access remains controlled, validated, and safe. It effectively prevents unauthorized manipulation or accidental corruption of critical system resources.

5.1.4 Performance Evaluation

The performance of TitanCFI was rigorously assessed by synthesizing the proposed architectural modifications on a Xilinx VCU118 FPGA. Various benchmarks were simulated to estimate their impact on hardware utilization and runtime overhead. This comprehensive evaluation provides a clear understanding of TitanCFI's efficiency in both performance and resource consumption.

Table 5.2: Comparison of runtime slowdowns introduced by TitanCFI against state-of-the-art CFI architectures, DExIE[68] and FIXER[69], across a selection of EmBench-IoT v1.0 and RISC-V-Tests benchmarks. The table reports the slowdown values for the three firmware implementations of TitanCFI (Optimized, Polling, and IRQ-based) and highlights how TitanCFI maintains competitive performance while providing flexible, RoT-based CFI enforcement. This detailed comparison illustrates the trade-offs between runtime overhead and enforcement strategy, showing the practical applicability of the proposed approach in embedded systems.

	Benchmark	DExIE	FIXER	Opt.	Poll.	IRQ
EmBench	`aha-mont64`	48	n.a.	–	–	–
	`edn`	47	n.a.	1	1	2
	`matmult-int`	48	n.a.	–	–	1
	`ud`	48	n.a.	12	18	43
RISC-V Tests	`rsort`	n.a.		–	–	1
	`median`	n.a.		3	5	12
	`qsort`	n.a.	2	–	–	1
	`multiply`	n.a.		2	3	6
	`dhrystone`	n.a.		360	553	1318

Runtime Overhead

To evaluate the runtime overhead, a diverse set of benchmarks from EmBench-IoT v1.0 and RISC-V-Tests was utilized, built with a standard RISC-V toolchain featuring GCC 12.2.0 and the `-O3` optimization level. The slowdown was computed by simulating the Register-Transfer Level of the reference SoC and extracting cycle-accurate execution traces, which report the number of cycles required to commit each instruction. These traces were then fed into a trace-driven model that emulated the latency required for CFI enforcement based on the three firmware analysis results, averaging the cycles for function calls and returns.

As shown in Table 5.2, TitanCFI demonstrates competitive runtime performance when compared against state-of-the-art CFI architectures like DExIE and FIXER [68, 69]. For a fair comparison, the CFI Queue depth was constrained to 1, simulating core stalling upon a single control flow instruction retirement. In this setup, TitanCFI often imposes lower runtime overhead than DExIE in several benchmarks, a notable achievement given that DExIE's implementation can reduce the clock frequency of the tested cores. While FIXER generally reports a 1.5% runtime overhead, TitanCFI's mean overhead, even in the worst-case scenario with the IRQ firmware, is only marginally higher for most benchmarks, excluding the `dhrystone` test. This validates TitanCFI's efficiency, especially considering the flexibility offered by its RoT-based approach.

A more comprehensive evaluation, presented in Table 5.3, details the execution cycles, control flow instructions retired, and slowdowns for a larger pool of benchmarks, including the entire EmBench-IoT suite and most RISC-V-Tests. This assessment was performed with a CFI queue size of 8, enabling a more realistic simulation of system behavior. The results indicate that for the majority of kernel benchmarks,

Table 5.3: Detailed statistics and runtime slowdowns for the EmBench-IoT v1.0 and RISC-V-Tests benchmark suites when executing under TitanCFI's control-flow integrity enforcement. The table reports, for each benchmark, the total committed cycles, the number of control-flow instructions (CF), and the measured slowdowns [%] for the three firmware implementations of TitanCFI: Optimized, Polling, and IRQ-based. Empty entries indicate benchmarks not applicable or not instrumented in the corresponding firmware setup. .

	Benchmark	Cycles	CF	Slowdown [%]		
				Opt.	Poll.	IRQ
EmBench	aha-mont64	2.51E+6	1.50E+1	–	–	–
	crc32	3.49E+6	1.50E+1	–	–	–
	cubic	1.10E+6	2.01E+4	46	107	390
	edn	4.23E+6	3.67E+2	–	–	–
	huffbench	3.49E+6	2.28E+3	1	3	11
	matmult-int	4.69E+6	2.05E+2	–	–	–
	minver	4.75E+5	4.50E+3	–	7	153
	nbody	1.21E+5	4.29E+3	163	301	849
	nettle-aes	5.20E+6	7.95E+2	–	–	–
	nettle-sha256	4.73E+6	8.57E+3	1	2	11
	nsichneu	5.24E+6	1.70E+1	–	–	–
	picojpeg	4.97E+6	2.14E+4	5	15	58
	qrduino	4.61E+6	4.35E+3	–	–	–
	sglib-combined	3.67E+6	2.62E+4	9	32	142
	slre	3.57E+6	6.69E+4	38	110	401
	st	1.47E+5	2.31E+2	–	–	2
	statemate	3.22E+6	2.75E+4	–	–	129
	ud	1.87E+6	2.98E+3	–	–	–
	wikisort	4.38E+5	7.69E+3	94	158	418
RISC-V Tests	dhrystone	4.57E+5	2.25E+4	260	452	1215
	median	2.53E+4	1.10E+1	–	–	–
	memcpy	1.20E+5	1.10E+1	–	–	–
	mm	1.41E+6	2.33E+5	1108	1752	4311
	mt-matmul	5.76E+4	2.38E+2	11	22	65
	mt-memcpy	4.08E+5	1.80E+1	–	–	–
	mt-vvadd	1.48E+5	3.30E+1	–	–	–
	multiply	3.72E+4	9.00E+0	–	–	–
	pmp	9.01E+5	5.90E+1	–	–	–
	qsort	2.68E+5	1.10E+1	–	–	–
	rsort	3.32E+5	1.10E+1	–	–	–
	spmv	1.67E+5	1.10E+1	–	–	–
	towers	2.01E+4	9.00E+0	–	–	–

TitanCFI introduces less than 10% overhead across its three firmware versions, underscoring its practical applicability in real-world embedded scenarios where kernel functions are paramount. The varying slowdowns observed across firmware versions further reinforce the benefits of the optimized firmware, aligning with the detailed latency analysis from the previous section. Higher overheads are primarily confined to benchmarks not typically associated with kernel functions, such as linear algebra, which are generally not targeted for CFI enforcement in practical systems.

Table 5.4: Comparison of hardware resource utilization for TitanCFI and DExIE[68], showing the impact of CFI enforcement in terms of LUTs, registers, and BRAM, and highlighting the minimal overhead introduced by TitanCFI compared to DExIE. The column *w.o CFI* reports the resources used without enabling CFI, *CFI* shows the resources with CFI enabled, and Δ indicates the absolute increase in each resource type. The *Overhead* column reports the relative percentage increase. A comparison with DExIE is also included for reference.

		w.o CFI	CFI	Δ	Overhead
Host	LUT	5.02E+4	5.14E+4	1.16E+3	+2.3%
	Registers	3.04E+4	3.22E+4	1.77E+3	+5.8%
	BRAM	6.60E+1	6.60E+1	–	–
SoC	LUT	4.41E+5	4.41E+5	1.33E+3	+0.3%
	Registers	2.57E+5	2.58E+5	2.19E+3	+0.9%
	BRAM	2.68E+2	2.68E+2	–	–
DIxIE	LUT	4.66E+3	8.02E+3	3.36E+3	+72.1%
	Registers	3.09E+3	5.33E+3	2.24E+3	+72.5%
	BRAM	1.36E+2	1.42E+2	6.00E+0	+4.4%

Hardware Utilization Overhead

The hardware utilization overhead of TitanCFI was meticulously evaluated through FPGA synthesis. Key metrics such as Look-Up Tables, registers, and Block RAMs required for implementation were reported.

As indicated in Table 5.4, the architectural modifications introduced by TitanCFI result in a negligible impact on hardware resource utilization. The overhead is less than 1% for the entire SoC and less than 6% when considering only the host core. This minimal footprint is particularly advantageous for resource-constrained embedded systems. Compared to DExIE, TitanCFI demonstrates superior efficiency, utilizing 60% fewer LUTs and requiring no BRAM slices, whereas DExIE necessitates BRAM. This highlights TitanCFI as a significantly lighter solution in terms of hardware resources. These modifications do not affect the maximum operating frequency of the SoC, ensuring that the integration of CFI does not introduce performance bottlenecks at the hardware level.

In summary, TitanCFI achieves a robust balance, providing strong security guarantees against control-flow hijacking attacks. It maintains minimal overheads in both runtime performance and hardware resource utilization. This detailed evaluation confirms its practicality and efficiency for RISC-V-based embedded systems.

5.1.5 Security Guarantees and Implications

TitanCFI provides strong security guarantees by isolating the CFI enforcement logic within the OpenTitan RoT. This isolation ensures that even if the software running on the host core is compromised, the integrity of the control flow remains protected by the tamper-proof and cryptographically secured environment of OpenTitan. The RoT's ability to authenticate CFI metadata and store it in protected regions further enhances its resilience against manipulation.

The work focuses on protecting software written in memory-unsafe languages like C, which are prone to exploitable vulnerabilities. A key security assumption is that the CFI Mailbox, the communication channel between the host and OpenTitan, cannot be tampered with by other entities in the SoC. This ensures the integrity of the control flow events transmitted for validation.

In conclusion, TitanCFI demonstrates a practical and secure approach to enforcing CFI in RISC-V systems by leveraging OpenTitan as a RoT. It effectively balances robust security with minimal performance and hardware overhead. TitanCFI provides a crucial defense mechanism against advanced code-reuse attacks in embedded environments. This study paves the way for future CFI policies and integrations in more complex multi-core platforms.

5.2 Exploring Post-Quantum Cryptography on Accelerated Platforms

This section examines the critical area of PQC in embedded systems, focusing on the potential of open-hardware platforms and the RISC-V architecture to address the emerging threat of quantum computers. Recent research on OpenTitan has demonstrated initial PQC implementations, including OTBN extensions and instruction set enhancements for schemes such as ML-KEM and ML-DSA [21, 52, 77]. However, dedicated hardware support on this platform is still in the early stages of development. Meanwhile, the present work investigates PQC integration on RISC-V platforms, exploring how instruction set extensions and the modern OpenSSL Provider system can be leveraged to accelerate PQC algorithms. This strategy provides a proactive preparation for the post-quantum era, complementing ongoing efforts such as OpenTitan's dedicated hardware development.

5.2.1 Motivation for Post-Quantum Cryptography in Embedded Systems

The urgent need for PQC in embedded systems stems directly from the imminent threat posed by quantum computers to current cryptographic standards, as comprehensively outlined in Section 2.3. This threat, particularly the risk of "harvest now, decrypt later" attacks, is critical for embedded devices because of their typically long operational lifespans [53].

While robust open-hardware RoT like OpenTitan provide an excellent foundation for conventional cryptographic algorithms and system integrity, dedicated hardware support for computationally intensive lattice-based PQC schemes remains under active development [77]. OpenTitan is actively extending its OTBN accelerator with dedicated hardware for polynomial arithmetic, sampling, and Keccak instruction set extensions, aimed at accelerating PQC schemes such as Dilithium, Kyber, and Falcon [21, 52, 77]. These enhancements are essential to enable hybrid signature verification, combining classical and PQC schemes, as recommended by security agencies [52, 77].

This evolving landscape highlights a current gap: while OpenTitan's dedicated PQC hardware support matures, there is an immediate need to explore alternative RISC-V platforms where PQC algorithms can be efficiently implemented and accelerated. This proactive approach ensures readiness for the post-quantum era. Integrating PQC into resource-constrained embedded and IoT devices inherently presents significant challenges, including high computational complexity, larger key sizes, and increased memory requirements. These demands necessitate optimized implementations and, often, dedicated hardware acceleration to meet stringent performance, energy, and latency constraints, as further detailed in Section 2.3

5.2.2 OpenSSL Provider for Hardware-Accelerated ML-KEM on Sargantana

To effectively address the challenges of PQC implementation in embedded systems, particularly on RISC-V platforms, this work focuses on enabling and accelerating ML-KEM through the development of an OpenSSL Provider. This approach leverages the modular and extensible nature of the RISC-V architecture for PQC acceleration, particularly through the standard ISA extensions discussed in Section 2.3. Additionally, it also benefits from the modern OpenSSL Providers architecture, detailed in Section 2.4, which enables flexible integration of PQC algorithms into cryptographic software stacks. By integrating hardware-accelerated ML-KEM operations within a standardized cryptographic interface, we provide a flexible and efficient pathway toward quantum-resistant embedded systems.

Design and Implementation of the ML-KEM Provider

The ML-KEM provider, developed on OpenSSL 3.5, targets the Sargantana RV64GBV core and is designed to integrate seamlessly into OpenSSL's framework while exploiting hardware acceleration from the standard RISC-V bit manipulation (B) and vector (V) extensions. Its architecture emphasizes modularity, memory safety, and flexibility, supporting multiple security levels (`MLKEM512`, `MLKEM768`, `MLKEM1024`) without requiring modifications to existing cryptographic applications. At the highest level, the provider follows the OpenSSL "Provider" model, which relies on a dispatch table to map cryptographic operations to their corresponding implementations. The main entry point, `OSSL_provider_init`, is invoked when

OpenSSL loads the provider. It initializes the provider context, sets up the dispatch table, and registers supported operations. In this case, the provider exposes key management (`KEYMGMT`) and key encapsulation (`KEM`) operations. Queries to the provider are routed via the dispatch mechanism to the appropriate module, such as `pqcuark_keymgmt_query` or `pqcuark_kem_query`.

Key management is handled by the `keymgmt_mlkem` module, which provides functions for key creation, cleanup, parameter handling, and key generation. Each ML-KEM key is represented by the `mlkem_key_t` structure, which stores pointers to the public key, private key, and seed, along with their respective sizes. The function `keymgmt_mlkem_new` allocates and initializes a new key structure, while `keymgmt_mlkem_free` safely releases all associated memory. Key generation occurs in `keymgmt_mlkem_gen`, which computes the public and private keys using hardware-accelerated polynomial routines. The generation process also supports multiple modes via the `keymgmt_mlkem512_gen_init`, `keymgmt_mlkem768_gen_init`, and `keymgmt_mlkem1024_gen_init` functions, each initializing a generation context with the appropriate security parameters.

The provider also implements functions for importing and exporting keys through OpenSSL parameter structures (`OSSL_PARAM`), which allows seamless integration with the EVP API. For instance, `keymgmt_mlkem_export` builds parameter sets from key components, while `keymgmt_mlkem_get_params` and `keymgmt_mlkem_set_params` handle retrieval and modification of key attributes.

The `KEM` module handles encapsulation and decapsulation of shared secrets. Each operation runs in the context of a `kem_mlkem_ctx_t` structure, tracking the provider context, selected ML-KEM mode, and associated key. Context creation/destruction uses `kem_mlkem_newctx` and `kem_mlkem_freectx`, while `kem_mlkem_dupctx` provides duplication if needed.

Encapsulation is implemented in `kem_mlkem_encapsulate`. It validates contexts and buffers, computes ciphertext using hardware-accelerated routines, and returns the ciphertext and shared secret. Decapsulation (`kem_mlkem_decapsulate`) follows a similar structure. The hardware path exploits vectorized Number Theoretic Transform routines and bit-manipulation optimizations, while the fallback relies on pseudo-random generation.

The provider aims to exploit the performance advantages of RISC-V's B and V extensions. Bit manipulation accelerates Keccak-based routines, while vector instructions enable parallel processing of polynomial coefficients. Empirical evaluations on the Sargantana core have shown that these extensions can yield speedups of up to 5.2× for ML-KEM and 10.1x for ML-DSA, with specific operations such as ML-KEM [78]. Early evaluations indicate that hand-optimized routines can achieve substantial improvements compared to scalar implementations. Building upon these insights, the present implementation extends this optimization model into a software-integrated context, providing a seamless interface within the OpenSSL provider framework All key management and KEM operations are exposed via variant-specific dispatch tables (`kem_mlkem512_functions`, `kem_mlkem768_functions`, `kem_mlkem1024_functions`), mapping internal routines

to standard OpenSSL KEM identifiers. Parameter handling, memory allocation, and cleanup are consistently managed, ensuring sensitive material is securely cleared and memory leaks are prevented.

5.2.3 Experimental Setup and Performance Evaluation

The experiments were conducted on an Alveo U55c FPGA running at 25 MHz. The benchmark harness is implemented in C and uses the standard OpenSSL `EVP KEM` and `EVP KEYMGMT` API. The tests are identical across runs except for the selection of the cryptographic provider, either the native OpenSSL provider or the custom ML-KEM provider developed in this work. Each test exercises the three main KEM operations (`keygen`, `encap`, and `decap`) across all ML-KEM security levels (`MLKEM512`, `MLKEM768`, and `MLKEM1024`). Each operation is repeated 10 times and the results are averaged to reduce measurement noise, which may arise from the Linux scheduling and other background activity on the FPGA platform.

Performance data were collected using the Linux `perf` tool, sampling hardware performance counters such as CPU cycles and instructions retired. For each operation, three independent runs were performed, and the arithmetic mean of the measurements was considered. Execution time was estimated by dividing the number of cycles by the known FPGA clock frequency, which is 25 MHz.

The CPI (Cycles per Instruction) was computed as the ratio of cycles to instructions. Speedup, when reported, is defined as the ratio of the execution time of OpenSSL to that of the custom hardware-accelerated provider, providing a direct measure of the performance improvement obtained through the hardware-accelerated implementation. This setup allows for a consistent and repeatable evaluation of the benefits of the custom provider while isolating the effect of hardware acceleration provided by the RISC-V extensions.

5.2.4 Results and Analysis

Table 5.5 summarizes the measured cycles, instructions, derived execution times, CPI, and speedups for each operation and ML-KEM variant. The data reveal clear trends in the performance of the custom ML-KEM provider compared to the baseline OpenSSL implementation.

Key generation exhibits only marginal differences between the two providers, with speedup values slightly below 1 across all three security levels (`MLKEM512`, `MLKEM768`, `MLKEM1024`). This indicates that, for key generation, the hardware-accelerated implementation provides performance comparable to the baseline. The CPI values remain similar, suggesting that the instruction mix is not significantly altered by the optimized routines. These observations are consistent with the fact that key generation is relatively memory-bound and involves less intensive arithmetic operations.

Table 5.5: Comparison of performance metrics between the baseline OpenSSL implementation and the custom hardware-accelerated ML-KEM provider across different security levels. Measured metrics include cycles, execution time, cycles per instruction (CPI), and resulting speedup for key generation, encapsulation, and decapsulation operations.

Operation	OpenSSL			ML-KEM Provider			*Speedup*
	Cycles[#]	*Time*[s]	*CPI*	*Cycles*[#]	*Time*[s]	*CPI*	
MLKEM-512							
keygen	56,310,501.5	2.252	2.89	59,087,646.2	2.364	2.87	0.95x
encap	45,389,305.0	1.815	2.75	27,760,765.2	1.110	3.14	1.63x
decap	45,694,162.1	1.828	2.73	27,239,119.0	1.090	3.18	1.68x
MLKEM-768							
keygen	56,649,674.0	2.266	2.87	59,400,214.1	2.376	2.86	0.95x
encap	46,504,781.2	1.860	2.67	28,304,130.8	1.132	3.11	1.64x
decap	47,034,189.7	1.881	2.64	27,434,696.6	1.097	3.18	1.71x
MLKEM-1024							
keygen	57,284,400.6	2.291	2.84	60,066,048.2	2.403	2.84	0.95x
encap	47,778,442.5	1.911	2.58	28,745,081.9	1.150	3.06	1.66x
decap	49,133,156.7	1.965	2.55	27,675,540.1	1.107	3.16	1.78x

In contrast, encapsulation benefits substantially from the hardware acceleration. Across all security levels, the custom provider achieves speedups ranging from approximately 1.63 to 1.66. The measured cycles are significantly lower than those of the OpenSSL implementation, and the CPI values indicate efficient utilization of the RISC-V vector and bit-manipulation extensions. Encapsulation operations, which rely heavily on polynomial arithmetic and Keccak-based sampling, are therefore well-suited to acceleration, and the results demonstrate a clear reduction in execution time.

Decapsulation exhibits similar behavior, with speedups ranging from 1.68 (`MLKEM512`) up to 1.78 (`MLKEM1024`). The CPI values indicate higher instruction-level parallelism for arithmetic-intensive tasks, reflecting the vectorized processing of polynomial coefficients. The consistent improvement across security levels shows that the custom implementation scales effectively with increasing key sizes.

Overall, these results demonstrate that while key generation is relatively insensitive to hardware acceleration, both encapsulation and decapsulation can benefit considerably from the RISC-V vector and bit-manipulation extensions. The measured speedups confirm that careful exploitation of these extensions can significantly reduce computation time for critical post-quantum KEM operations. These findings highlight the practicality of deploying hardware-accelerated PQC in embedded environments and provide quantitative evidence of the advantages of the custom ML-KEM provider, as reported in Table 5.5.

Chapter 6

Conclusions

This thesis has explored critical advancements in the realm of secure embedded systems, focusing on the OpenTitan platform as a robust Root of Trust (RoT) for RISC-V System-on-Chip (SoC). Through a comprehensive approach encompassing architectural enhancements, innovative software frameworks, and novel security mechanisms, this research has significantly contributed to enhancing the performance, integrity, and future-readiness of open-source hardware security.

6.1 Summary of Key Contributions

Building on these results, the work presents several pivotal contributions towards enhancing security and performance in RISC-V SoC, particularly through the development and evaluation of the TitanSSL framework, architectural optimizations, TitanCFI, and preliminary explorations into Post-Quantum Cryptography (PQC).

TitanSSL: Bridging Hardware Acceleration and Software Ease-of-Use

A primary achievement is TitanSSL, a secure software stack designed to fully unlock the secure and performance-enhancing capabilities of OpenTitan within RISC-V SoCs. It establishes a secure, performant, and user-friendly cryptographic backend that significantly outperforms software-only solutions and addresses the rigidities and overheads of alternative security frameworks. Its design principles emphasize efficient exposure of OpenTitan's cryptographic hardware accelerators, a standardized OpenSSL application-facing interface, robust security guarantees through OpenTitan's inherent features like memory isolation and secure boot, a modular and extensible architecture, and optimized data flow. End-to-end measurements from Linux demonstrate substantial speedups for symmetric algorithms (e.g., SHA-256 up to 10× and AES-256-CBC up to 3×) and asymmetric cryptography (e.g., RSA-1024 signing 1.4×) over pure software implementations. Initial performance characterization of OpenTitan highlighted that memory access latency and data movement overheads constituted a significant bottleneck, consuming up to 93.78% of total ex-

ecution cycles for cryptographic operations. To overcome this, the integration of a dedicated Direct Memory Access (DMA) engine and Tightly Coupled Data Memory (TCDM) was crucial. The DMA offloads data transfer responsibilities from the Ibex core, enabling efficient burst transactions, while the TCDM provides high-bandwidth, low-latency local storage for accelerators. This dual enhancement, particularly through a page-granular, double-buffered pipeline, dramatically improved throughput and accelerator utilization for memory-bound workloads like SHA-256, achieving up to 2.2× speedup over the baseline TitanSSL without DMA and raising hardware utilization to 20.6% for large payloads. While significant performance gains were realized, a detailed cycle-level analysis revealed that up to 70% of potential accelerator throughput is still obscured by user-space and kernel overheads within the Linux stack.

TitanCFI: Strengthening System Integrity

TitanCFI introduces a novel mechanism to reinforce system integrity against sophisticated software attacks, such as code-reuse attacks. By leveraging OpenTitan's security and isolation properties as a RoT, TitanCFI enforces software-defined Control-Flow Integrity (CFI) policies without necessitating custom hardware monitors or intrusive modifications to the host core pipeline. The critical CFI enforcement logic is securely contained within OpenTitan's trusted and isolated environment, making it resilient to potential compromises of the host Operating System (OS) or application software. This solution achieves competitive runtime performance and introduces minimal hardware resource overhead, demonstrating superior efficiency compared to other state-of-the-art CFI architectures like DExIE.

Initial Steps in Post-Quantum Cryptography

In preparation for the post-quantum era, this research initiated the exploration of PQC integration on RISC-V platforms. It addressed the imminent threat posed by quantum computers to current cryptographic standards by developing an OpenSSL Provider for hardware-accelerated Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM). This approach capitalized on the modular nature of the RISC-V architecture and standard Instruction Set Architecture (ISA) extensions (B and V extensions). Performance evaluations showed substantial speedups for ML-KEM encapsulation (1.63× to 1.66×) and decapsulation (1.68× to 1.78×) compared to the baseline OpenSSL implementation, underscoring the practicality of deploying hardware-accelerated PQC in embedded environments.

6.2 Broader Impact and Significance of the Research

Building upon the results presented throughout this research, the work significantly advances the state of the art in secure RISC-V SoC design and open-hardware security. By demonstrating how a hardware RoT like OpenTitan can be effectively integrated to offload cryptographic operations and enforce security policies, it provides a robust foundation for building trustworthy embedded systems. The insights gained from architectural performance bottlenecks and their mitigation through DMA/TCDM integration highlight critical considerations for future hardware–software co-design efforts. Furthermore, the development of TitanCFI offers a vital defense mechanism against advanced code-reuse attacks, enhancing the overall resilience of embedded systems. The initial exploration of PQC introduces a flexible and efficient pathway toward quantum-resistant embedded systems, ensuring long-term security in an evolving threat landscape. Adherence to open standards and reliance on OpenTitan further contribute to the broader ecosystem, fostering collaborative development and wider adoption of secure hardware.

6.3 Future Research Directions

Looking ahead, several promising avenues for future research emerge:

- *Advancing PQC Integration with OpenTitan*: Further research can focus on seamlessly integrating PQC schemes directly within OpenTitan's secure environment or leveraging its existing and future accelerators, such as OpenTitan Big Number (OTBN) extensions for polynomial arithmetic and Keccak instruction set extensions, to achieve even greater performance and security.
- *Extended TitanCFI Capabilities*: Exploring more complex CFI policies beyond shadow-stack protection and extending support to multi-core host environments would enhance TitanCFI's applicability and robustness in diverse embedded systems.
- *Further Software-Hardware Optimizations*: Investigating new techniques to reduce the identified software and system-level overheads, particularly within the Linux stack, is crucial to fully unlock the theoretical maximum throughput of OpenTitan's hardware accelerators. This could include more streamlined memory management, optimized user-space interactions, and advanced scheduling mechanisms.

These endeavors collectively aim to further solidify the robust foundation laid by this work, ensuring the continued advancement of trustworthy and high-performance embedded systems in an ever-evolving security landscape.

Bibliography

[1] G. Sharma, G. Bousdras, S. Ellinidou, O. Markowitch, J.-M. Dricot, and D. Milojevic, "Exploring the security landscape: NoC-based MPSoC to Cloud-of-Chips," *Microprocessors and Microsystems*, vol. 84, p. 103 963, 2021, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2021.103963. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933121001411.

[2] Z. Rehman, I. Gondal, M. Ge, H. Dong, M. Gregory, and Z. Tari, "Proactive defense mechanism: Enhancing IoT security through diversity-based moving target defense and cyber deception," *Computers & Security*, vol. 139, p. 103 685, 2024, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103685. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823005953.

[3] S. Narashiman et al., *Chiplets on Wheels: Review Paper on Holistic Chiplet Solutions for Autonomous Vehicles*, 2024. arXiv: 2406.00182 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2406.00182.

[4] C. Labrado, H. Thapliyal, and S. P. Mohanty, "Fortifying Vehicular Security Through Low Overhead Physically Unclonable Functions," *CoRR*, 2021, Volume: abs/2106.02976. arXiv: 2106.02976. [Online]. Available: https://arxiv.org/abs/2106.02976.

[5] G. Kornaros, "Hardware-Assisted Machine Learning in Resource-Constrained IoT Environments for Security: Review and Future Prospective," *IEEE Access*, vol. 10, pp. 58 603–58 622, 2022. DOI: 10.1109/ACCESS.2022.3179047.

[6] E. Baccelli, *Internet of Things (IoT): Societal Challenges & Scientific Research Fields for IoT*. Dec. 2021. [Online]. Available: https://inria.hal.science/hal-03474888.

[7] N.-F. Polychronou, P.-H. Thevenon, M. Puys, and V. Beroulle, "A Comprehensive Survey of Attacks without Physical Access Targeting Hardware Vulnerabilities in IoT/IIoT Devices, and Their Detection Mechanisms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 1, Sep. 2021, ISSN: 1084-4309. DOI: 10.1145/3471936. [Online]. Available: https://doi.org/10.1145/3471936.

[8] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security Assurance for System-on-Chip Designs With Untrusted IPs," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017. DOI: 10.1109/TIFS.2017.2658544.

[9] S. Bhasin et al., "Secure Your SoC: Building System-an-Chip Designs for Security," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, 2020, pp. 248–253. DOI: 10.1109/SOCC49529.2020.9524760.

[10] S. Ul Haq, Y. Singh, A. Sharma, R. Gupta, and D. Gupta, "A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks," *Discover Internet of Things*, vol. 3, Oct. 2023. DOI: 10.1007/s43926-023-00045-2.

[11] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-Based Control-Flow Protection for IoT Devices," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 214–226. DOI: 10.1109/EuroSP.2018.00023.

[12] R. Yu et al., *Building Embedded Systems Like It's 1996*, 2022. arXiv: 2203.06834 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2203.06834.

[13] Q. Yu, Z. Zhang, and J. Dofe, "Proactive Defense Against Security Threats on IoT Hardware," in *Modeling and Design of Secure Internet of Things*. Publisher information not available, 2020, pp. 407–433. DOI: 10.1002/9781119593386.ch18.

[14] A. Shamsoshoara, A. Korenda, F. Afghah, and S. Zeadally, *A Survey on Physical Unclonable Function (PUF)-based Security Solutions for Internet of Things*, 2020. arXiv: 1907.12525 [cs.CR]. [Online]. Available: https://arxiv.org/abs/1907.12525.

[15] F. Fischer, M. Niedermaier, T. Hanka, P. Knauer, and D. Merli, "Analysis of Industrial Device Architectures for Real-Time Operations Under Denial of Service Attacks," in *Information and Communications Security*, W. Meng, D. Gollmann, C. D. Jensen, and J. Zhou, Eds., Cham: Springer International Publishing, 2020, pp. 462–478, ISBN: 978-3-030-61078-4.

[16] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, "Static analysis for discovering IoT vulnerabilities," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 1, pp. 71–88, Feb. 2021, ISSN: 1433-2779. DOI: 10.1007/s10009-020-00592-x. [Online]. Available: https://doi.org/10.1007/s10009-020-00592-x.

[17] S. R. Tanksalkar et al., *LEMIX: Enabling Testing of Embedded Applications as Linux Applications (Extended Report)*, 2025. arXiv: 2503.17588 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2503.17588.

[18] L. Leonardi, G. Lettieri, P. Perazzo, and S. Saponara, "On the Hardware—Software Integration in Cryptographic Accelerators for Industrial IoT," *Applied Sciences*, vol. 12, no. 19, 2022, DOI: 10.3390/app12199948, ISSN: 2076-3417. [Online]. Available: https://www.mdpi.com/2076-3417/12/19/9948.

[19] A. S. Siddiqui, Y. Gui, and F. Saqib, "Secure Boot for Reconfigurable Architectures," *Cryptography*, vol. 4, no. 4, 2020, ISSN: 2410-387X. DOI: 10.3390/cryptography4040026. [Online]. Available: https://www.mdpi.com/2410-387X/4/4/26.

[20] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Collective Remote Attestation at the Internet of Things Scale: State-of-the-Art and Future Challenges," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2447–2461, 2020. DOI: 10.1109/COMST.2020.3008879.

[21] E. Urquhart and F. Stajano, "Acceleration of Core Post-quantum Cryptography Primitive on Open-Source Silicon Platform Through Hardware/Software Co-design," in *Cryptology and Network Security: 23rd International Conference, CANS 2024, Cambridge, UK, September 24–27, 2024, Proceedings, Part I*, Cambridge, United Kingdom: Springer-Verlag, 2024, pp. 144–161, ISBN: 978-981-97-8012-9. DOI: 10.1007/978-981-97-8013-6_7. [Online]. Available: https://doi.org/10.1007/978-981-97-8013-6_7.

[22] Y. Yamak, S. Tosun, and M. Aydos, "DICEguard: enhancing DICE security for IoT devices with periodic memory forensics," *The Journal of Supercomputing*, vol. 80, pp. 1–21, May 2024. DOI: 10.1007/s11227-024-06194-7.

[23] B. Sá, J. Martins, and S. Pinto, *A First Look at RISC-V Virtualization from an Embedded Systems Perspective*, 2021. arXiv: 2103.14951 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2103.14951.

[24] T.-T. Hoang et al., "Trusted Execution Environment Hardware by Isolated Heterogeneous Architecture for Key Scheduling," *IEEE Access*, vol. 10, pp. 46014–46 027, 2022. DOI: 10.1109/ACCESS.2022.3169767.

[25] M. Bognar, J. Van Bulck, and F. Piessens, "Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1638–1655. DOI: 10.1109/SP46214.2022.9833735.

[26] T. M. Fernández-Caramés, "From Pre-Quantum to Post-Quantum IoT Security: A Survey on Quantum-Resistant Cryptosystems for the Internet of Things," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6457–6480, 2020. DOI: 10.1109/JIOT.2019.2958788.

[27] T. Liu, G. Ramachandran, and R. Jurdak, *Post-Quantum Cryptography for Internet of Things: A Survey on Performance and Optimization*, 2024. arXiv: 2401.17538 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2401.17538.

[28] S. Chawande, "Quantum computing threats to cybersecurity protocols," *World Journal of Advanced Engineering Technology and Sciences*, vol. 15, May 2025. DOI: 10.30574/wjaets.2025.15.2.0546.

[29] A. P. Bhatt and A. Sharma, "Quantum Cryptography for Internet of Things Security," *Journal of Electronic Science and Technology*, vol. 17, no. 3, pp. 213–220, 2019, ISSN: 1674-862X. DOI: https://doi.org/10.11989/JEST.1674-862X.90523016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1674862X19300345.

[30] S. Balogh, O. Gallo, R. Ploszek, P. Špaček, and P. Zajac, "IoT Security Challenges: Cloud and Blockchain, Postquantum Cryptography, and Evolutionary Techniques," *Electronics*, vol. 10, no. 21, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10212647. [Online]. Available: https://www.mdpi.com/2079-9292/10/21/2647.

[31] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 4, pp. 17–61, Aug. 2019. DOI: 10.13154/tches.v2019.i4.17-61. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8344.

[32] A. Aikata, A. C. Mert, M. Imran, S. Pagliarini, and S. S. Roy, "KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 2, pp. 747–758, 2023. DOI: 10.1109/TCSI.2022.3219555.

[33] J. Xiong, L. Shen, Y. Liu, and X. Fang, "Enhancing IoT security in smart grids with quantum-resistant hybrid encryption," *Scientific Reports*, vol. 15, Jan. 2025. DOI: 10.1038/s41598-024-84427-8.

[34] B. Tan, *The Quest to Build Trust Earlier in Digital Design*, 2024. arXiv: 2409.05832 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2409.05832.

[35] J. Mishra and S. K. Sahay, *Modern Hardware Security: A Review of Attacks and Countermeasures*, 2025. arXiv: 2501.04394 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2501.04394.

[36] C. Kocaoğullar et al., *A Confidential Computing Transparency Framework for a Comprehensive Trust Chain*, 2024. arXiv: 2409.03720 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2409.03720.

[37] V. Kumar et al., "Towards Designing a Secure RISC-V System-on-Chip: ITUS," *Journal of Hardware and Systems Security*, vol. 4, pp. 329–342, Dec. 2020. DOI: 10.1007/s41635-020-00108-8.

[38] T. Lu, *A Survey on RISC-V Security: Hardware and Architecture*, 2021. arXiv: 2107.04175 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2107.04175.

[39] *ERTS 2022 proceedings*, Jun. 2022. [Online]. Available: https://hal.science/hal-03704287.

[40] lowRISC CIC, *OpenTitan Official Documentation*, https://opentitan.org/book/doc/introduction.html, 2019.

[41] B. Crispo et al., *CROSSCON: Cross-platform Open Security Stack for Connected Devices*, 2024. arXiv: 2406.03401 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2406.03401.

[42] M. Hassan, "Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities," *IEEE Design & Test*, vol. 35, no. 4, pp. 47–55, 2018. DOI: 10.1109/MDAT.2017.2771447.

[43] Y. Sovyn, V. Khoma, and M. Podpora, "Comparison of Three CPU-Core Families for IoT Applications in Terms of Security and Performance of AES-GCM," *IEEE Internet of Things Journal*, vol. 7, no. 1, pp. 339–348, 2020. DOI: 10.1109/JIOT.2019.2953230.

[44] T. L. Loo, M. K. Ishak, and K. Ammar, "Design and implementation of secure boot architecture on RISC-V using FPGA," *Microprocess. Microsyst.*, vol. 101, no. C, Sep. 2023, ISSN: 0141-9331. DOI: 10.1016/j.micpro.2023.104889. [Online]. Available: https://doi.org/10.1016/j.micpro.2023.104889.

[45] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019, ISSN: 1557-9999. DOI: 10.1109/tvlsi.2019.2926114. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2019.2926114.

[46] A. Meza, F. Restuccia, J. Oberg, D. Rizzo, and R. Kastner, "Security Verification of the OpenTitan Hardware Root of Trust," *IEEE Security & Privacy*, vol. 21, no. 3, pp. 27–36, 2023. DOI: 10.1109/MSEC.2023.3251954.

[47] S. Suhail, R. Hussain, A. Khan, and C. S. Hong, "On the Role of Hash-Based Signatures in Quantum-Safe Internet of Things: Current Solutions and Future Directions," *IEEE Internet of Things Journal*, vol. 8, no. 1, pp. 1–17, 2021. DOI: 10.1109/JIOT.2020.3013019.

[48] M. Telesklav and S. Tauner, *Comparative Analysis and Enhancement of CFG-based Hardware-Assisted CFI Schemes*, 2021. arXiv: 2103.04456 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2103.04456.

[49] G. Yeo, Y. Kim, S. Song, and D. Kwon, "Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M," *IEEE Access*, vol. 10, pp. 132 675–132 684, 2022. DOI: 10.1109/ACCESS.2022.3230791.

[50] L. Buckwell, O. Gilles, D. G. Pérez, and N. Kosmatov, *Execution at RISC: Stealth JOP Attacks on RISC-V Applications*, 2023. arXiv: 2307.12648 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2307.12648.

[51] T. Mishra, T. Chantem, and R. Gerdes, "Survey of Control-flow Integrity Techniques for Real-time Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 4, pp. 1–32, Jul. 2022, ISSN: 1558-3465. DOI: 10.1145/3538275. [Online]. Available: http://dx.doi.org/10.1145/3538275.

[52] A. Abdulrahman et al., "Towards ML-KEM & ML-DSA on OpenTitan," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 1–19. DOI: 10.1109/SP61157.2025.00220.

[53] X. Carril et al., "Hardware Acceleration for High-Volume Operations of CRYSTALS-Kyber and CRYSTALS-Dilithium," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 3, Sep. 2024, ISSN: 1936-7406. DOI: 10.1145/3675172. [Online]. Available: https://doi.org/10.1145/3675172.

[54] J. Soto-Cruz, E. Ruiz-Ibarra, J. Vázquez-Castillo, A. Espinoza-Ruiz, A. Castillo-Atoche, and J. Mass-Sanchez, "A Survey of Efficient Lightweight Cryptography for Power-Constrained Microcontrollers," *Technologies*, vol. 13, no. 1, 2025, ISSN: 2227-7080. DOI: 10.3390/technologies13010003. [Online]. Available: https://www.mdpi.com/2227-7080/13/1/3.

[55] W. Wang, J. Han, X. Cheng, and X. Zeng, "An energy-efficient crypto-extension design for RISC-V," *Microelectronics Journal*, vol. 115, p. 105 165, Jul. 2021. DOI: 10.1016/j.mejo.2021.105165.

[56] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono, "Hardware Acceleration of OpenSSL Cryptographic Functions for High-Performance Internet Security," in *2010 International Conference on Intelligent Systems, Modelling and Simulation*, 2010, pp. 374–379. DOI: 10.1109/ISMS.2010.89.

[57] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-Quantum Signatures on RISC-V with Hardware Acceleration," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 2, Mar. 2024, ISSN: 1539-9087. DOI: 10.1145/3579092. [Online]. Available: https://doi.org/10.1145/3579092.

[58] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 219–242, Jun. 2020. DOI: 10.46586/tches.v2020.i3.219-242.

[59] V. B. Dang, K. Mohajerani, and K. Gaj, "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber," *IEEE Transactions on Computers*, vol. 72, no. 2, pp. 306–320, 2023. DOI: 10.1109/TC.2022.3222954.

[60] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, "On-Board Decision Making in Space with Deep Neural Networks and RISC-V Vector Processors," *Journal of Aerospace Information Systems*, vol. 18, pp. 1–17, Jun. 2021. DOI: 10.2514/1.I010916.

[61] K. Cheang, C. Rasmussen, D. Lee, D. W. Kohlbrenner, K. Asanović, and S. A. Seshia, *Verifying RISC-V Physical Memory Protection*, 2022. arXiv: 2211.02179 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2211.02179.

[62] S. Johnson, D. Rizzo, P. Ranganathan, J. McCune, and R. Ho, "Titan: Enabling a transparent silicon root of trust for cloud," in *Hot Chips: A Symposium on High Performance Chips*, vol. 194, 2018.

[63] M. Ciani et al., "Unleashing OpenTitan's Potential: a Silicon-Ready Embedded Secure Element for Root of Trust and Cryptographic Offloading," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 5, Sep. 2025, ISSN: 1539-9087. DOI: 10.1145/3690823. [Online]. Available: https://doi.org/10.1145/3690823.

[64] F. Antognazza, A. Barenghi, and G. Pelosi, "Metis: An Integrated Morphing Engine CPU to Protect Against Side Channel Attacks," *IEEE Access*, vol. 9, pp. 69 210–69 225, 2021. DOI: 10.1109/ACCESS.2021.3077977.

[65] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009, ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. [Online]. Available: https://doi.org/10.1145/1609956.1609960.

[66] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 985–999. DOI: 10.1109/SP.2019.00076.

[67] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012, ISSN: 1094-9224. DOI: 10.1145/2133375.2133377. [Online]. Available: https://doi.org/10.1145/2133375.2133377.

[68] C. Spang, Y. Lavan, M. Hartmann, F. Meisel, and A. Koch, "DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity," in *Workshop on Design and Architectures for Signal and Image Processing (14th Edition)*, ser. DASIP '21, Budapest, Hungary: Association for Computing Machinery, 2021, pp. 26–34, ISBN: 9781450389013. DOI: 10.1145/3441110.3441146. [Online]. Available: https://doi.org/10.1145/3441110.3441146.

[69] A. De, A. Basu, S. Ghosh, and T. Jaeger, "FIXER: Flow Integrity Extensions for Embedded RISC-V," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 348–353. DOI: 10.23919/DATE.2019.8714980.

[70] P. Nasahl and S. Mangard, "SCRAMBLE-CFI: Mitigating Fault-Induced Control-Flow Attacks on OpenTitan," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, ser. GLSVLSI '23, Knoxville, TN, USA: Association for Computing Machinery, 2023, pp. 45–50, ISBN: 9798400701252. DOI: 10.1145/3583781.3590221. [Online]. Available: https://doi.org/10.1145/3583781.3590221.

[71] E. Diemert, A. Betlei, C. Renaudin, M.-R. Amini, T. Gregoir, and T. Rahier, *A Large Scale Benchmark for Individual Treatment Effect Prediction and Uplift Modeling*, 2021. arXiv: 2111.10106 [stat.ML]. [Online]. Available: https://arxiv.org/abs/2111.10106.

[72] J. Li et al., *Zipper Stack: Shadow Stacks Without Shadow*, 2020. arXiv: 1902.00888 [cs.CR]. [Online]. Available: https://arxiv.org/abs/1902.00888.

[73] A. Caulfield, N. Rattanavipanon, and I. D. O. Nunes, *ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation*, 2023. arXiv: 2303.16282 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2303.16282.

[74] A. Ehret, E. Del Rosario, K. Gettings, and M. A. Kinsy, "A Hardware Root-of-Trust Design for Low-Power SoC Edge Devices," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6. DOI: 10.1109/HPEC43674.2020.9286164.

[75] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "Instruction-Set Accelerated Implementation of CRYSTALS-Kyber," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648–4659, 2021. DOI: 10.1109/TCSI.2021.3106639.

[76] E. Karabulut and A. Aysu, "RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 26–32. DOI: 10.1109/FPL50879.2020.00016.

[77] T. Stelzer, F. Oberhansl, J. Schupp, P. Karl, and H. Turcuman, "Extended version: enabling lattice-based post-quantum cryptography on the opentitan platform," *Journal of Cryptographic Engineering*, vol. 15, Apr. 2025. DOI: 10.1007/s13389-025-00369-5.

[78] X. Carril, A. M. Pasoot, E. Parisi, C. A. Lara-Nino, O. Farràs, and M. Moretó, "Pqcuark: A scalar risc-v isa extension for ml-kem and ml-dsa," *IACR Cryptol. ePrint Arch.*, vol. 2025, p. 2178, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:283673355.

[79] L. Cuomo et al., "Towards a RISC-V Open Platform for Next-generation Automotive ECUs," in *2023 12th Mediterranean Conference on Embedded Computing (MECO)*, IEEE, Jun. 2023, pp. 1–8. DOI: 10.1109/meco58584.2023.10154913. [Online]. Available: http://dx.doi.org/10.1109/MECO58584.2023.10154913.

[80] O. S. Foundation, *Source code for the OpenSSL software*, https://github.com/openssl/openssl, 1998.

[81] D. Schrammel et al., "Donky: domain keys – efficient in-process isolation for RISC-V and x86," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20, USA: USENIX Association, 2020, ISBN: 978-1-939133-17-5.

[82] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, "Nile: A Programmable Monitoring Coprocessor," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 92–95, 2018. DOI: 10.1109/LCA.2017.2784416.

[83] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," *CoRR*, vol. abs/2009.05262, 2020. arXiv: 2009.05262. [Online]. Available: https://arxiv.org/abs/2009.05262.

[84] C. Koenig, B. Forsberg, and L. Benini, "HeroSDK: Streamlining Heterogeneous RISC-V Accelerated Computing from Embedded to High-Performance Systems," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 280–287. DOI: 10.1109/ICCD63220.2024.00050.

[85] A. Musa, E. Parisi, L. Barbierato, E. Patti, A. Acquaviva, and F. Barchi, "Integrating OpenTitan as a Security Controller for Cryptographic Tasks in RISC-V SoCs," in *ITASEC/SERICS*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:279077465.

[86] A. Musa, E. Parisi, L. Barbierato, A. Acquaviva, and F. Barchi, "End-to-end Integration of OpenTitan Security Features in a Pure RISC-V SoC," in *2024 20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2024, pp. 1–4. DOI: 10.1109/SMACD61181.2024.10745397.

[87] A. Musa et al., "TitanSSL: Towards Accelerating OpenSSL in a Full RISC-V Architecture Using OpenTitan Root-of-Trust," in *Computer Safety, Reliability, and Security: 43rd International Conference, SAFECOMP 2024, Florence, Italy, September 18–20, 2024, Proceedings*, Florence, Italy: Springer-Verlag, 2024, pp. 169–183, ISBN: 978-3-031-68605-4. DOI: 10.1007/978-3-031-68606-1_11.

[88] F. Conti et al., "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2481–2494, 2017. DOI: 10.1109/TCSI.2017.2698019.

[89] G. P. Group, *Global Platform Official Website*, https://globalplatform.org/, 2023.

[90] NXP, *EdgeLock Secure Enclave*, https://www.nxp.com/products/nxp-product-information/nxp-product-programs/edgelock-secure-enclave:EDGELOCK-SECURE-ENCLAVE, 2019.

[91] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3387532.

[92] E. Parisi et al., "Assessing the Performance of OpenTitan as Cryptographic Accelerator in Secure Open-Hardware System-on-Chips," in *Proceedings of the 21st ACM International Conference on Computing Frontiers*, ser. CF '24, Ischia, Italy: Association for Computing Machinery, 2024, pp. 172–179, ISBN: 9798400705977. DOI: 10.1145/3649153.3649213.

[93] B. Potter, "Microsoft SDL Threat Modelling Tool," *Network Security*, vol. 2009, no. 1, pp. 15–18, 2009, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(09)70008-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S135348580970008X.

[94] M. Bach-Nutman, *Understanding The Top 10 OWASP Vulnerabilities*, 2020. arXiv: 2012.09960 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2012.09960.

[95] C. W. Enumeration, *2022 CWE Top 25 Most Dangerous Software Weaknesses*, https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[96] E. Parisi et al., *TitanCFI: Toward Enforcing Control-Flow Integrity in the Root-of-Trust*, 2024. arXiv: 2401.02567 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2401.02567.