Performance analysis of MAC algorithms: Benchmarking for Automotive Embedded Systems

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Cyber Security Engineering
September 2025
Francesca Capra

Supervisors:
Seppo Virtanen
Ismayil Hasanov

UNIVERSITY OF TURKU
Department of Computing

Francesca Capra: Performance analysis of MAC algorithms:
Benchmarking for Automotive Embedded Systems

Thesis, 124 p., 8 app.
Master of Science (Tech) Thesis
Cyber Security Engineering
September 2025

The rapid digitalization of automotive systems has heightened the demand for efficient cryptographic mechanisms to ensure message integrity and authenticity in resource-constrained embedded environments, such as Electronic Control Units (ECUs). This thesis evaluates the computational performance of nine Message Authentication Code (MAC) algorithms—SipHash, Chaskey, ASCON-MAC, ASCON-PRFshort, Poly1305, KMAC256, AES-CMAC, BLAKE2, and BLAKE3—across three heterogeneous platforms: an x86 laptop (Intel Core i5-1145G7), a Raspberry Pi 4 (ARM Cortex-A72), and an Infineon AURIX TC397 microcontroller with integrated Hardware Security Module (HSM). Benchmarks focus on short messages (8- and 16-byte payloads typical of CAN frames), measuring latency, CPU cycles, throughput, memory footprint, and code size to address the trade-offs between security, performance, and real-time determinism.

Results reveal platform-specific hierarchies: lightweight ARX-based MACs (SipHash, Chaskey, Poly1305) dominate software implementations, achieving sub-microsecond latencies and high throughput, while ASCON variants offer balanced, moderate performance; hash- and sponge-based algorithms (BLAKE3, KMAC256) lag due to their fixed overheads. HSM acceleration yields a speedup for AES-CMAC on TC397, outperforming software but trailing lightweight alternatives. A novel benchmarking framework, publicly available on GitHub, enables reproducible cross-platform analysis.

This work provides empirical guidance for selecting automotive MACs, informing next-generation ECU architectures and fostering continued optimization, offering both practical advice and a foundation for future advancements in automotive cybersecurity.

Keywords: MAC algorithms, Cryptography, Authentication, Benchmarking

AI Usage Disclaimer

Generative AI tools were employed during the preparation of this thesis to assist with drafting, language refinement, and LaTeX formatting, in accordance with the University of Turku's guidelines. The tools were used strictly as supportive aids and not as a substitute for the author's own analysis or critical thinking.

All AI-generated outputs were carefully reviewed, edited, and validated by the author, who assumes full responsibility for the final content. No confidential, personal, or sensitive data were entered into online AI platforms, ensuring compliance with GDPR and research integrity standards. AI usage respected the provider's copyright terms and did not conflict with any research or collaboration agreements.

Contents

1 Introduction **1**

1.1 Problem Statement 3

1.2 Research Objectives and Questions 4

1.3 Scope and Limitations 5

1.3.1 Scope 5

1.3.2 Limitations 6

1.4 Contributions 6

2 Literature Review **9**

2.1 Message Authentication Codes: Overview 9

2.1.1 MAC Fundamentals and Security Properties 10

2.2 MAC Algorithms in Automotive Systems 12

2.2.1 Current Automotive Security Requirements 12

2.2.2 Physical Security Challenges: SCA and FA 12

2.2.3 Countermeasures and Resistant Design 13

2.3 Benchmarking Methodologies and Results 14

2.3.1 Existing Benchmarking Frameworks 14

2.3.2 Cross-Platform Comparison 15

3 Survey of MAC Algorithms **18**

3.1 Classical MACs 18

3.1.1 AES-CMAC . . . 18
3.2 Lightweight MACs . . . 20
3.2.1 SipHash . . . 20
3.2.2 Chaskey . . . 22
3.3 Modern Hash-based MACs . . . 24
3.3.1 BLAKE2 . . . 24
3.3.2 BLAKE3 . . . 26
3.4 Sponge-based MACs . . . 28
3.4.1 KMAC256 . . . 28
3.4.2 ASCON-MAC & ASCON-PRFshort . . . 30
3.5 Polynomial-based MACs . . . 32
3.5.1 Poly1305 . . . 33

4 Specification and Benchmarking Methodology 35

4.1 Algorithm Selection Criteria . . . 35
4.1.1 Security level requirements . . . 35
4.1.2 Performance characteristics . . . 38
4.1.3 Implementation availability . . . 39
4.1.4 Automotive relevance . . . 41
4.2 Platform Specifications . . . 41
4.3 Benchmarking Environment . . . 44
4.3.1 Development tools . . . 44
4.3.2 Library dependencies . . . 45
4.3.3 Optimization flags and architecture-specific considerations . . . 45
4.4 Testing Procedures . . . 46
4.4.1 Message Size Variations . . . 46
4.4.2 Number of Iterations and Statistical Approach . . . 47
4.4.3 Warm-up Procedures and Measurement Isolation . . . 47

4.4.4 HSM vs. Software Testing Methodology 48
4.5 Metrics Collected 48
4.5.1 Primary Metrics: Latency, Cycles, and Throughput 49
4.5.2 Secondary Metrics: Memory Usage 49
4.5.3 Correctness Validation 50
4.6 Threats to Validity and Mitigation 50
4.6.1 Measurement Accuracy and Precision 50
4.6.2 Platform-specific Optimizations 51
4.6.3 Compiler and Library Version Dependencies 51

5 Implementation Details **52**
5.1 Code Organization and Integration 52
5.1.1 Unified benchmarking framework 53
5.1.2 Algorithm wrapper interfaces 54
5.1.3 Data structure design 55
5.1.4 Build system and dependency integration 56
5.2 Platform-Specific Implementations 56
5.2.1 x86 Optimizations (AVX2, SSE4.1) 56
5.2.2 ARM NEON Utilization 57
5.2.3 TC397 Microcontroller Constraints 57
5.3 HSM Integration on TC397 58
5.3.1 HSM Interface Implementation 59
5.3.2 Context Switching Overhead 60
5.3.3 Hardware Acceleration Pipeline 60
5.4 Measurement Infrastructure 61
5.4.1 Cycle Counter Implementations 61
5.4.2 Timing Accuracy Verification 62
5.4.3 Statistical Analysis Procedures 63

6 Experimental Results **64**

6.1 Computational Performance Results 64

6.1.1 x86 Laptop Performance 66

6.1.2 ARM Raspberry Pi Performance 75

6.1.3 TC397 Microcontroller Performance 89

6.2 Memory and Code Size Analysis 97

6.2.1 RAM usage during operation 97

6.2.2 Stack requirements comparison 98

6.2.3 Binary size measurements 98

7 Discussion **100**

7.1 Cross-Platform Comparative Analysis 101

7.1.1 Algorithm ranking per platform 101

7.1.2 Message Size Sensitivity Analysis 103

7.1.3 Performance scaling characteristics 107

7.1.4 Overhead Analysis 108

7.2 HSM vs. Software Implementation Analysis 110

7.2.1 When HSM acceleration pays off 110

7.3 Suitability for Automotive Applications 113

7.3.1 Security level vs. performance mapping 114

7.4 Algorithm Selection Framework 115

7.4.1 Decision Making Criteria 116

7.4.2 Algorithm Comparison Matrix: 118

7.4.3 Application-Specific Guidelines 118

8 Conclusions and Future Work **121**

8.1 Conclusions 121

8.2 Future Research 123

References **125**

Appendices

A Code Size Analysis **A-1**

1 Introduction

The automotive industry has undergone a radical transformation in recent decades, evolving from mechanical systems to sophisticated networks of interconnected **Electronic Control Units (ECUs)**. Modern vehicles, in particular, represent a complex ecosystem that integrates a large number of embedded devices, including ECUs and microcontrollers, where security is essential for the majority of these components.

This digitalisation has imposed unprecedented computational demands on automotive embedded systems, driven by advanced functionalities such as autonomous driving, over-the-air software updates, and sophisticated in-vehicle communication networks. These systems are characterised by their reliance on hardware with constrained resources, namely processing power, memory, and energy, giving rise to distinctive engineering challenges. ECUs communicate via various protocols, process sensor data, and control critical safety systems, including engine control, braking, and steering assistance. Achieving a balance between high performance and robust security architectures is thus essential for domain control and data fusion applications.

In this context, efficient **Message Authentication Code (MAC)** algorithms in resource-limited environments are of primary importance. Modern cryptographic tools ensure data integrity and authenticity, which are critical for preventing cyber-attacks and safeguarding vehicle safety. MACs represent fundamental components of contemporary security architectures and protocols, protecting against message tampering, identity theft, and unauthorized access in physical systems such as buildings

and automobiles. As vehicles become increasingly connected and autonomous, robust cryptographic mechanisms are required to verify that messages have not been altered and originate from legitimate sources, while operating within the strict computational and memory constraints of automotive embedded systems. The mounting incidence of cyberattacks targeting automotive networks underscores the pressing need for MAC solutions that can operate effectively within these constraints while upholding stringent security standards.

This challenge is rooted in the fundamental trade-off between security and performance. Automotive systems often operate on microcontrollers with limited processing power, memory, and energy budgets, where critical real-time operations cannot tolerate computational delays introduced by cryptographic operations. At the same time, these systems must uphold high security standards to protect against increasingly sophisticated cyber threats that could compromise vehicle safety and passenger security. Reactivity and temporal determinism are critical safety requirements, and algorithms must be efficient in terms of speed, resource consumption, and latency. Traditional cryptographic algorithms, designed for general-purpose computing environments with abundant resources, are often inadequate. To address these challenges, **Lightweight Cryptography (LWC)** has emerged as a promising solution to improve security and efficiency in resource-constrained devices. The **National Institute of Standards and Technology (NIST)** has initiated a standardization project for LWC to address the security needs of constrained devices [1]. LWC algorithms are designed to balance security, cost, and performance by adjusting key size, number of encryption rounds, and system architecture.

Selecting an optimal MAC algorithm requires a comprehensive evaluation of cryptographic robustness, architectural specifics, and resource constraints inherent to automotive embedded systems. **Hardware Security Modules (HSMs)** offer a potential solution. These modules provide dedicated cryptographic hardware that

has the capacity to offload computational load from the main CPU. Furthermore, they have the potential to improve security through hardware-based key storage and tamper resistance. However, the performance implications of using HSMs versus software-only implementations remain to be fully characterized, especially for MAC algorithms in automotive contexts.

1.1 Problem Statement

The expanding domain of embedded systems, particularly in automotive applications, reveals a significant gap in the systematic analysis of MAC algorithms across heterogeneous platforms. Despite the wide range of available MAC constructions, including lightweight designs, sponge-based algorithms, and polynomial-based schemes, a comprehensive performance evaluation in real-world automotive contexts is lacking. This gap is especially critical given the heterogeneous hardware landscape of modern vehicles, where multiple processing architectures, ranging from high-performance platforms to constrained embedded microcontrollers, coexist within a single ecosystem. The absence of standardized, cross-platform benchmarking methodologies hinders the identification of optimal MAC configurations for specific platform constraints and performance demands.

Performance results on general-purpose computers often differ significantly from those on embedded platforms, emphasizing the need for testing in target environments [2]. This thesis addresses the lack of empirical performance data for informed MAC algorithm selection in automotive applications. Existing studies typically focus on theoretical analysis or general-purpose platforms [3], leaving practitioners without practical guidance for automotive hardware under realistic constraints.

A key area of investigation is the performance differential between software implementations and those that leverage HSM or other hardware accelerators. HSMs, such as those in the **Infineon AURIX TC397**, may offer superior efficiency for

cryptographic operations, but the computational overhead of data transfers and net performance benefits remain unquantified. This gap impedes optimal system architecture decisions, potentially leading to unnecessary complexity or missed optimization opportunities. While HSMs are increasingly deployed in automotive systems, the computational overhead and performance benefits of hardware-accelerated MAC operations compared to software-only implementations remain underexplored.

The automotive industry's shift toward connected and autonomous vehicles amplifies the need for MAC algorithms that operate efficiently within real-time constraints while ensuring security. The diversity of MAC constructions, each with distinct computational characteristics, memory footprints, and security properties, complicates the selection of algorithms. Without empirical cross-platform evaluation, algorithm selection remains speculative.

1.2 Research Objectives and Questions

This research addresses the identified challenges through three interrelated objectives that together form the foundation of this thesis.

This thesis primarily aims to conduct an in-depth comparison of the computational performance of nine MAC algorithms, SipHash, Chaskey, ASCON-MAC, ASCON PRFShort, Poly1305, KMAC256, AES-CMAC, BLAKE2, and BLAKE3, focusing on metrics that are relevant to automotive applications, including latency, CPU cycles, and throughput.

The second objective is to quantify the performance benefits and trade-offs of using the **HSM**, with particular focus on the Infineon AURIX TC397 platform. This involves comparing hardware-accelerated execution (e.g., AES-CMAC in HSM) with software-only implementations, considering both raw computation time and data transfer overhead.

The third objective is to develop a robust benchmarking methodology capturing

key performance metrics (CPU cycles, latency, throughput, memory usage) under realistic message sizes typical of automotive systems (8- and 16-byte payloads). Based on the results, formulate practical recommendations for selecting MAC algorithms that strike a balance between security, performance, and real-time determinism in embedded environments.

1.3 Scope and Limitations

This thesis focuses narrowly on computational performance analysis of MAC algorithms in automotive-oriented embedded environments, under well-defined boundaries to ensure analytical depth and reproducibility.

1.3.1 Scope

The study targets MAC algorithms that represent the main design paradigms (block cipher-based, ARX-based, hash-based, sponge-based, polynomial-based), chosen for their relevance to automotive and embedded systems.

Three heterogeneous hardware platforms were analyzed: an x86 laptop with an Intel Core i5-1145G7 processor, representing high-performance development environments; a Raspberry Pi 4 with an ARM (Advanced RISC Machines) Cortex-A72 processor, representing mid-range embedded systems without hardware acceleration; and an Infineon AURIX TC397 microcontroller, representing automotive-grade hardware with integrated HSM.

Metrics include CPU cycles, latency (in nanoseconds), throughput (MACs per second), memory footprint, and code size, using short message sizes (8 and 16 bytes) that reflect typical Controller Area Network (CAN) frame and extended payload scenarios.

1.3.2 Limitations

The primary focus is on the intrinsic computational performance of MAC algorithms, with implementation at the protocol level deliberately excluded. This means that the algorithms are evaluated in isolation, without considering the overhead or interactions that might occur within complete protocol stacks, such as Internet Protocol Security (IPsec) or Transport Layer Security (TLS).

Three heterogeneous hardware platforms were chosen to demonstrate a variety of usage scenarios for automotive embedded systems. However, it should be noted that the selected platforms do not encompass all possible architectures.

Furthermore, the research focuses exclusively on message authentication, excluding encryption performance in Authenticated Encryption with Associated Data (AEAD). Implementing an AEAD algorithm often involves compromising between encryption and authentication speeds [4], but the analysis does not explore these interdependencies.

The analysis relies on existing cryptanalysis published by the cryptographic community, not exploring new vulnerabilities. While this approach is common in performance-oriented studies, it means that this research would not identify any undocumented weaknesses in the algorithms.

Environmental factors (e.g., temperature, electromagnetic interference) and energy consumption are excluded due to measurement complexity, limiting results to ideal conditions.

1.4 Contributions

This thesis makes a significant contribution to the emerging field of embedded system security in the automotive sector. It provides a detailed, quantifiable analysis of the MAC algorithm's performance on various hardware architectures, offering engineers

and researchers practical tools and guidance. Specifically, the main contributions are as follows:

Firstly, a comprehensive comparison of the computational performance of nine MAC algorithms was conducted on three different devices: an x86 laptop, a Raspberry Pi 4, and an Infineon AURIX TC397 microcontroller. Focusing on short messages (8 and 16 bytes) typical of vehicular payloads, the evaluation revealed significant performance variations, challenging general-purpose benchmarks.

Secondly, an innovative performance characterization **framework** is introduced, which includes metrics such as CPU cycles, latency, throughput, and memory usage. This framework has been specifically tailored to the unique requirements of automotive environments, thus establishing a robust methodology for future research. The complete source code, including the algorithm implementations and benchmarking tools, has been made publicly available on a GitHub repository to support future research and practical applications in automotive security [5].

Thirdly, the study provides a quantitative insight into **HSM** acceleration, including measurement of AES-CMAC performance inside the Infineon AURIX TC397 HSM versus software-only implementations, clarifying the real benefits and overhead of hardware offloading.

Ultimately, it offers **practical recommendations** for selecting MAC algorithms. These recommendations strike a balance between security, computational efficiency, and real-time determinism in resource-constrained environments, enabling engineers to design safer and more efficient automotive architectures.

Overall, this thesis significantly advances our understanding of MAC algorithm performance in critical embedded contexts, facilitating the design and implementation of safer and more efficient automotive systems.

The remainder of this thesis is organized as follows. Chapter 2 reviews the lit-

erature on MACs, automotive security, and prior benchmarking studies, highlighting research gaps. Chapter 3 analyzes the design, security, and computational characteristics of the nine MACs considered in this work. Chapter 4 presents the methodology, including experimental design, platforms, and performance metrics. Chapter 5 details the implementation of each algorithm and related optimizations. Chapter 6 reports the performance results and comparative analysis. Chapter 7 discusses the findings and their implications. Finally, Chapter 8 concludes the thesis, summarizes key contributions, and suggests directions for future work. The appendices contain supplementary materials, including detailed benchmark data and experimental results.

2 Literature Review

The evolution of modern automotive systems toward increasingly connected and automated architectures has intensified the need for robust and efficient cryptographic mechanisms. **MAC** are a fundamental component to guarantee the integrity and authenticity of communications between **ECUs** and other embedded devices in the automotive context. Unlike traditional environments, automotive systems operate under stringent real-time constraints, limited computational resources, and safety-critical reliability requirements. These characteristics make the selection and implementation of MAC algorithms a complex technical challenge that demands in-depth analysis of performance, security, and energy efficiency.

MACs are cryptographic mechanisms designed to guarantee the integrity and authenticity of messages in secure communication systems. By combining a secret key with a message, MACs generate a fixed-length tag that allows the recipient to verify that the message has not been altered and originates from a legitimate source. In embedded systems, particularly in automotive applications, MACs play a critical role in protecting communications among resource-constrained devices.

2.1 Message Authentication Codes: Overview

This section introduces the fundamental concepts of MACs.

2.1.1 MAC Fundamentals and Security Properties

The security of embedded systems, particularly in the automotive sector, requires cryptographic primitives that guarantee the integrity and authenticity of messages to be theoretically robust and highly efficient in environments with limited resources. The literature clearly outlines the evolution of these algorithms, from classic constructs to the most recent lightweight cryptography designs.

Bellare, Canetti, and Krawczyk formalized MAC security, defining it as a **keyed hash function** existentially unforgeable under chosen-message attacks, ensuring adversaries cannot forge valid message-tag pairs without the secret key [6].

Early standards are based on the use of well-established primitives. AES-CMAC, defined in 2005 by NIST in Special Publication 800-38B, is a historical reference [7]. CMAC was developed as an improved version of CBC-MAC to overcome its security shortcomings, particularly when processing variable-length messages. Iwata et al. provided the IETF with documentation on the AES-CMAC algorithm, which is based on the AES algorithm [8]. The robustness of these constructions is supported by theoretical analysis. For example, the OMAC family (of which CMAC is a member) is pseudorandom, provided that the underlying block cipher is a random permutation. This offers tight security bounds [9].

For software applications on modern CPUs that require high speed, algorithms based on universal hashing have been favoured. In 2005, Daniel J. Bernstein introduced the Poly1305-AES code, a high-speed MAC that exploits polynomial hashing [10]. Poly1305 is often used alongside the ChaCha20 stream cipher to create the ChaCha20-Poly1305 AEAD scheme, which was standardised by Yoav Nir and Adam Langley in RFC 7539 (2015) [11].

In the field of LWC, research has focused on optimising 32-bit microcontrollers, resulting in primitives based on ARX (Addition-Rotation-XOR). Jean-Philippe Aumasson and Daniel J. Bernstein (2012) proposed SipHash as a fast pseudorandom

function (PRF) for short inputs [12]. Christoph Dobraunig, Florian Mendel, and Martin Schläffer (2014) performed the first differential cryptanalysis of SipHash. They identified a feature in SipHash-2-4 with a probability of $2^{-236.3}$ and a distinguishing feature in the finalisation phase with practical complexity without compromising the security of the complete algorithm [13], [14].

Meanwhile, Nicky Mouha et al. (2014) developed Chaskey, which is also an ARX MAC optimised for 32-bit microcontrollers [15]. However, subsequent analysis by Guillaume Leurent (2016) revealed vulnerabilities in seven rounds of Chaskey [16]. This led to the development of a more conservative twelve-round variant, Chaskey-12, as detailed in Nicky Mouha's update (2015) [17].

The most recent evolution has been shaped by the NIST LWC standardization process. ASCON, which is based on a sponge construction, has emerged as the standard [18]. Christoph Dobraunig et al. (2021) provided the ASCON v1.2 specification, emphasising that ASCON is online, single-pass, and inverse-free, thereby reducing implementation overheads [19]. The security of ASCON has been the subject of extensive study. For instance, Christoph Dobraunig et al. (2015) analysed its resistance to differential and linear attacks and found impossible differentials for up to five rounds of permutation [20]. Further cryptanalysis on reduced-round ASCON was conducted by Y. Li et al. (2017) [21].

In this context of permutation- and sponge-based MACs, KMAC also fits in, which is standardized by NIST in Special Publication 800-185 [22].

Finally, the BLAKE family of hash functions offers MAC functionality via keyed hashes. Jean-Philippe Aumasson et al. (2013) designed BLAKE2 to be simpler, smaller, and faster than MD5 [23], [24]. Jian Guo et al. (2014) conducted a detailed analysis of BLAKE2, examining its differential properties and impossible differentials for permutation [25]. Jack O'Connor et al. (2020) then introduced BLAKE3: a unified algorithm that simplifies adoption and provides consistent, high performance

across different platforms [26].

2.2 MAC Algorithms in Automotive Systems

The security of in-vehicle communication systems (in-vehicle networks) has become a crucial research area, given the expansion of "connected cars" and the increasing risks of physical tampering and cryptographic attacks.

2.2.1 Current Automotive Security Requirements

To mitigate the growing physical and logical security threats to vehicles, the automotive industry has developed specific standards. In particular, the *AUTomotive Open System ARchitecture (AUTOSAR)* released the *Secure Onboard Communication (SecOC)* standard. **SecOC** mandates the use of MAC codes to ensure message integrity (preventing data tampering) and sender authentication.

The literature confirms that, within the CAN protocol, the most common in-vehicle network protocol, the AES-CMAC algorithm is the recommended authentication mechanism [7], [8]. The security of the SecOC protocol depends strictly on the security of AES-CMAC implemented in the ECUs.

2.2.2 Physical Security Challenges: SCA and FA

The automotive environment is highly exposed to Side-Channel Attacks (SCA), such as Differential Power Analysis (DPA) [27]. In this context, the analysis focuses on the vulnerability of software and hardware implementations on ECUs.

Recent works have focused on vulnerabilities of algorithms under physical access conditions. The study by Katsumi Ebina, Rei Ueno, and Naofumi Homma (2023), titled *Side-Channel Analysis Against SecOC-Compliant AES-CMAC* [28], represents a fundamental critique of the implementation security of this standard in the context

of CAN. AES-CMAC, while cryptographically robust, is susceptible to side-channel analysis attacks when implemented on ECUs compliant with the SecOC standard. Their study developed an innovative analysis flow that sequentially targets the S-boxes of the first three AES rounds to estimate intermediate values, exposing implementation weaknesses in automotive ECUs. While conventional SCA required approximately 400,000 waveforms for key recovery, their deep learning-based SCA (DL-SCA) approach significantly improved efficiency, reducing the requirement to just 150 waveforms.

In addition to SCAs, embedded systems are vulnerable to Fault Analysis (FA) attacks. FAs can be induced to alter execution and recover the secret key. A comprehensive review by Baksi et al. [29] surveys various FA models targeting lightweight cryptographic schemes. Additional studies explore related fault analysis techniques, such as Statistical Ineffective Fault Analysis (SIFA) and Subset Fault Analysis (SSFA), which exploit ineffective faults to compromise security [19]. In the context of authenticated encryption, Fault Intensity Map Analysis (FIMA) [30] assesses the resistance to fault-induced vulnerabilities, evaluating the impact of faults on cryptographic operations and highlighting the need for robust countermeasures in devices with limited resources.

2.2.3 Countermeasures and Resistant Design

To mitigate these attacks, countermeasures at the implementation level are essential. These include Detection (fault detection), Infection (fault propagation), and Prevention (injection prevention) [29]. Many approaches rely on redundant computations.

The complexity of AES operations makes secure implementation particularly costly, requiring extensive countermeasures against physical attacks [31]. By contrast, ASCON was designed with the primary goal of facilitating protection against SCAs. ASCON also uses strong key initialization and finalization, which limits an attacker's

ability to recover the secret key even if the internal state is compromised during data processing [20], [32].

In general, countermeasures against SCA, such as masking, can be implemented compactly for architectures such as ASCON [33]. For protection against FAs, countermeasures based on spatial redundancy have been proposed to ensure that cryptographic devices running ASCON in automotive systems can detect and correct errors caused by physical interference [34].

2.3 Benchmarking Methodologies and Results

The evaluation of cryptographic algorithm performance, particularly that of MACs, is complex due to the heterogeneous nature of embedded platforms. Researchers have developed standardized frameworks to overcome challenges related to reproducibility and external validity.

2.3.1 Existing Benchmarking Frameworks

Standardization of measurements is essential to ensure fair and meaningful comparisons.

The **FELICS-AEAD** (Fair Evaluation of Lightweight Cryptographic Systems for AEAD) framework, introduced by Luan Cardoso dos Santos, Johann Großschädl, and Alex Biryukov in 2019 [3], provides an open-source tool for evaluation on 8-bit AVR, 16-bit MSP430, and 32-bit ARM Cortex-M microcontrollers. FELICS-AEAD stands out for the use of a **low-level API** which allows obtaining more detailed results on the intrinsic properties of ciphers. The external validity of the results is reinforced by scenario-driven evaluation scenarios that aim to replicate real security operations in resource-constrained environments.

However, simulation-based frameworks, such as FELICS-AEAD, may not fully

capture real-world performance. Sooyeon Shin, Minwoo Kim, and Taekyoung Kwon (2017) noted in their work that the results obtained in simulation using frameworks such as FELICS differed significantly from the experimental results obtained on real microcontrollers, emphasizing the importance of practical analysis [2].

This limitation extends to high-end suites like *eBACS* (ECRYPT Benchmarking of Cryptographic Systems), which, while widely used, are optimized for 64-bit processors (Intel/AMD, Cortex-A) and thus unsuitable for low-memory microcontrollers [3]. Such systems mask instruction-level bottlenecks in embedded studies, risking overestimation of practical performance [2], [35].

A crucial aspect is defining evaluation scenarios that accurately reflect real-world usage. The challenges are related to the unique hardware characteristics of microcontrollers:

- **Normalization of Architectural Factors:** Cross-platform comparisons require normalization of factors such as compiler optimization level and inlining. Other challenges include handling costs of instructions dependent on word size, costs of rotation and multiplication operations, and memory wait states [3].
- **Impact of Word Size:** Studies analyzed how register word size influences efficiency, explaining why algorithms operating on 32-bit words drastically outperform those operating on 64-bit on 8/16-bit architectures [2].

2.3.2 Cross-Platform Comparison

Comparative evaluation in the literature highlights clear performance hierarchies that depend on hardware characteristics, message length, and algorithmic design.

ARM Platform Analysis

Benchmarking studies on ARM Cortex-M4 platforms reveal significant runtime differences among MAC and AEAD algorithms. Evaluations on STM32 microcon-

trollers measured AES-GCM, ChaCha20-Poly1305, HMAC-SHA256, KMAC, and SipHash across multiple payload sizes [36]. Results consistently show that **KMAC** is the slowest across all payload sizes, confirming its computational overhead on resource-constrained systems.

Lightweight MACs achieve substantial gains compared to traditional standards. Chaskey, for instance, achieves up to 700× lower cycle counts and drastically reduced code size compared to AES-CMAC [2], [15]. Similar results are reported for block-cipher-based lightweight variants (LEA-CMAC, SIMON-CMAC, SPECK-CMAC), which consistently outperform AES-CMAC and HMAC-SHA256 on low-power microcontrollers [37].

Lightweight Cryptography Comparative Analysis

Broader performance surveys confirm that ARX-based primitives (e.g., SipHash) excel for short inputs and software-centric deployments, whereas classical block-cipher MACs incur higher latency unless supported by hardware acceleration [35]. Early runtime measurements on general-purpose CPUs (Pentium II) already indicated AES-CBC-MAC's advantage over HMAC-SHA-1 [38], establishing performance relations that still hold on modern platforms. These studies collectively underline that the most efficient choice is strongly workload- and platform-dependent.

Hardware Implementation Efficiency

Hardware studies complement software findings by exposing area and throughput trade-offs. ASCON implementations achieve exceptional area efficiency, requiring as little as 2.57 kGE for unprotected AEAD cores, and offer competitive throughput-per-area metrics even for short messages [33], [39], [40]. Sponge-based designs generally outperform AES-based MACs in low-area settings due to reduced round complexity and inherent parallelism. This exceptional footprint reduction directly

scales deployability across dense sensor networks, making ASCON particularly attractive for automotive applications requiring multiple distributed ECUs [31].

For hash-based primitives, optimized FPGA/ASIC implementations of BLAKE and BLAKE2 achieve throughput up to 73.7 Gb/s while improving energy efficiency over SHA-256 [41], indicating the growing maturity of alternative keyed-hash hardware support [42].

Bhuvaneshwari et al. [43] proposed the **7S-PASCON** (7-Stage Pipelined Ascon) architecture, showing that pipelining can significantly increase throughput while preserving area efficiency, making ASCON even more attractive for resource-constrained automotive hardware.

Summary and Implications for This Work

Literature consistently shows that ARX- and permutation-based primitives dominate in software performance on microcontrollers, while sponge-based constructions (ASCON) provide the best compromise between area and throughput in hardware. In contrast, traditional standards (AES-CMAC, HMAC) require hardware acceleration to remain competitive.

These observations motivate the experimental study presented in this thesis, where selected MACs are benchmarked on multiple platforms to validate their suitability under real-time and resource-constrained conditions.

3 Survey of MAC Algorithms

This chapter presents a comprehensive survey of the selected MAC algorithms, examining their design principles, security properties, and implementation characteristics that are relevant to automotive applications. The algorithms are categorized based on their underlying cryptographic approaches.

3.1 Classical MACs

This section provides an overview of classical MAC algorithms and their fundamental mechanisms.

3.1.1 AES-CMAC

Among classic MACs, **AES-CMAC** (Advanced Encryption Standard, Cipher Message Authentication Code) is based on a block cipher that uses the AES algorithm [44]. Designed in 2005, CMAC was defined in NIST Special Publication Standard 800-38B as an improved version of CBC-MAC, overcoming its security vulnerabilities.

Algorithm Structure

The design principles of AES-CMAC are grounded in the AES block cipher, operating exclusively with its direct encryption function. The operation relies on a sophisticated subkey generation algorithm and a message authentication process. First, AES-CMAC derives two additional secret subkeys, $K1$ and $K2$, for each primary key K.

This process is performed only once per key. The algorithm begins by encrypting a block of zeros (0^{128}) with the primary key K in AES-ECB mode, resulting in an intermediate value L. K_1 is then derived from L by left-shifting one bit, with conditional XOR operations based on the most significant bit. Similarly, K_2 is generated from K_1 by the same process.

The MAC generation algorithm proceeds using AES-CBC encryption. The message (M) is divided into blocks of 128 bits, with special handling for the last block depending on whether it requires padding [7]. The result of the final AES-CBC encryption is the MAC tag.

Security and Performance

It is important to note that AES-CMAC is deterministic, consistently producing the same tag for a given message and key.

AES-CMAC provides strong guarantees of data integrity and authenticity, and is designed to detect accidental, intentional, and unauthorized changes. The security of AES-CMAC is intrinsically linked to the robustness of the underlying AES algorithm. With AES-128, AES-CMAC provides a security level of 128 bits. The tag length represents a crucial security parameter that protects against guessing attacks. For most applications, a tag length of at least 64 bits is recommended. Regarding the key duration, it is recommended that a single key be used for no more than 2^{48} messages per 128-bit block (as in AES) to mitigate the probability of internal collision attacks. It is essential to note that the CMAC algorithm itself does not prevent replay attacks; however, application protocols can reduce this risk by using sequence counters or timestamps.

Direct CMAC implementations are not inherently resistant to side-channel attacks or fault attacks, which involve deliberately introducing faults to recover secret information. In terms of computational complexity and efficiency, while AES-CMAC

is robust, it may be limited compared to newer algorithms, especially in contexts with resource constraints. MACs based on block ciphers, such as AES-CMAC, are generally slower due to their dependence on intensive block cipher operations. However, AES-CMAC can benefit significantly from hardware acceleration.

Finally, AES-CMAC is used in cryptographic protocols such as IPsec and TLS for message authentication.

3.2 Lightweight MACs

This section provides an overview of lightweight MAC algorithms, emphasizing their performance and applicability in embedded and automotive systems.

3.2.1 SipHash

SipHash, proposed by Aumasson and Bernstein in 2012, provides an excellent solution for contexts characterized by limited computational resources [12].

Algorithm Structure

SipHash is a permutation-based MAC algorithm that employs the Addition-Rotation-XOR (ARX) design methodology. It features a 256-bit internal state divided into four 64-bit words, named v_0, v_1, v_2, and v_3, and utilizes a 128-bit secret key. The algorithm's output is a 64-bit authentication tag.

SipHash's operation can be analyzed through three distinct stages: initialization, compression, and finalization. During the initialization phase, the internal state is prepared by combining the secret key K with four predefined constants.

During the compression phase, the message is processed in 64-bit blocks. Each block is incorporated into the state and then mixed using the algorithm's core primitive, the `SipRound` function. This function repeatedly applies a sequence of

additions, rotations, and XOR operations to the internal state, providing diffusion and ensuring that small changes in the input propagate across the entire state. The number of `SipRound` iterations is determined by the algorithm's compression parameter c. To handle the final block, SipHash uses a padding mechanism to prevent extension attacks.

Once all message blocks are processed, the finalization phase is performed. The state is slightly modified to indicate that no more input will be processed, and the `SipRound` function is applied d additional times to achieve thorough mixing. The final tag is then obtained by combining the four internal state words, producing the 64-bit authentication result.

Security and Performance

The different variants of SipHash are denoted with the notation `SipHash-c-d`, where "c" represents the number of compression rounds applied for each message block and "d" indicates the number of finalization rounds. SipHash is designed to provide high protection levels, with the `SipHash-2-4` variant (2 compression rounds, 4 finalization rounds) representing the recommended standard configuration. Variants such as `SipHash-4-8` provide additional security for particularly sensitive applications.

From a security perspective, the algorithm is robust to tag truncation and eliminates the need for nonce value handling, avoiding security risks associated with nonce reuse. SipHash was explicitly designed to prevent timing attacks, which are a type of side-channel attack that exploits variations in execution time to infer secret key information.

However, like many permutation-based algorithms, SipHash is potentially vulnerable to more sophisticated side-channel attacks that could allow portions of the internal state, and consequently the secret key, to be recovered. These attacks include power consumption analysis, electromagnetic radiation, and induced perturbations (fault

injection). In high-security environments, therefore, the implementation of SipHash should include specific countermeasures against these attack vectors. Nevertheless, cryptographic analyzes did not compromise the algorithm's claimed security.

Its primary applications include the authentication of network traffic and the protection of hash tables against "hash flooding" (denial-of-service) attacks, a common problem when using non-cryptographic hash functions.

SipHash's performance makes it particularly suitable for IoT devices with limited resources. The emphasis on optimization for short inputs, combined with an efficient architecture and low resource requirements, establishes SipHash as a promising solution for authentication in environments with limited resources, where balancing security and performance is crucial.

3.2.2 Chaskey

Chaskey represents another notable lightweight MAC algorithm designed for applications requiring 128-bit security that cannot implement standard MAC algorithms due to stringent speed, power consumption, or code size requirements.

Algorithm Structure

The design principles of Chaskey are aimed at efficiency on 32-bit microcontrollers [15]. The algorithm uses a 128-bit K key to process a message of arbitrary size and produce an authentication tag of length t (where $t \leq 128$ bits). The algorithm's operation follows a well-defined structure that can be broken down into three main stages: subkey generation, message processing, and tag finalization.

During the subkey generation phase, two derived subkeys, K_1 and K_2, are created for each primary key K via simple shift and conditional XOR operations on 128-bit words, making the process computationally very light. In the message processing phase, the input text is divided into consecutive blocks of 128 bits each. The

algorithm processes each block through an iterative structure combining the current block with the internal state via XOR operations, followed by application of the π permutation. This permutation forms the cryptographic core of Chaskey and consists of eight identical rounds based on the ARX architecture and inspired by the SipHash design. Each round operates on four 32-bit words and performs a specific sequence of modular additions, rotations, and XORs. For management of the final block, Chaskey implements an intelligent padding mechanism similar to SipHash's approach. Tag finalization is performed by extracting the first t bits of the 128-bit final state.

Security and Performance

Chaskey's security guarantees are provable, based on the security of an underlying Even-Mansour block cipher. The algorithm aims to provide 128 bits of security against key recovery and $\min(128, t)$ bits of security against distinguisher and tag guessing attacks. Collision attacks become likely after approximately $2^{n/2}$ calls to the block cipher with the same key. Chaskey is also robust against tag guessing attacks, making a tag length of at least 64 bits advisable for most typical applications. Despite its theoretical security, a straightforward implementation of Chaskey offers no inherent resistance to hardware-based side-channel or fault attacks. Like SipHash, Chaskey eliminates the need for nonce management, thereby completely eliminating the security risks associated with nonce value reuse.

However, it is secure against timing attacks since execution time depends only on message length and not the secret key. Cryptographic analysis has shown that Even–Mansour block ciphers based on Chaskey have a large margin of safety against even the most advanced differential cryptanalysis attacks, with data complexity limited to 2^{64}. Studies on the distinguishers of reduced versions of SipHash (which inspired Chaskey) and Chaskey itself have revealed no security threats to the full

versions.

For applications requiring greater long-term security, the **Chaskey-LTS** (Long-Term Security) variant was developed, which uses 16 rounds instead of the standard 8. This version roughly doubles the clock cycles and power consumption, but offers a higher security margin against possible future attacks and advances in cryptanalysis.

In terms of performance and efficiency, Chaskey has been optimized specifically for ARM Cortex-M microcontrollers, with a very compact memory footprint and high energy efficiency.

3.3 Modern Hash-based MACs

This section discusses modern hash-based MAC, highlighting their security properties and performance characteristics.

3.3.1 BLAKE2

The **BLAKE2** family of cryptographic hash functions has emerged as a particularly promising solution, designed to offer high performance in both software and hardware, while maintaining a solid security margin. **BLAKE2** is an evolution of the BLAKE cryptographic hash function, which was one of five finalists in the NIST SHA-3 competition. Introduced in 2012, BLAKE2 was conceived by Jean-Philippe Aumasson et al. with the primary objective of improving the lightness and speed of BLAKE, without compromising the robust level of security equivalent to SHA-3 algorithms [23], [45].

Algorithm Structure

The BLAKE2 family comprises two main versions: **BLAKE2b**, which is optimized for 64-bit architectures with the capability to produce a 512-bit hash and process

1024-bit blocks, and **BLAKE2s**, which was developed for 32-bit platforms with the capability to produce a 256-bit hash and process 512-bit blocks.

The algorithm is based on an iterative structure that uses a compression function derived from a block cipher. The process starts by initializing the internal state with a modified vector, then sequentially processes the message, and finally produces the hash output. The compression function implements a variant of the Davies-Meyer construction, incorporating additional parameters such as counters and flags, which ensure the behavioral uniqueness of each compression instance. All configuration parameters, including digest size, key length, salt, customization options, and tree hashing options, are stored in a parameter block. The operating core, **G-function**, repeatedly mixes the internal state and message words through simple arithmetic and bitwise operations, spreading the influence of each message part across the output. Several rounds of this function, combined with a predefined message permutation, ensure strong diffusion. If the final chunk is incomplete, BLAKE2 pads it with zeros to maintain consistency. A final mixing step combines the processed data to produce the authentication tag, with the option to adjust the tag's length for different needs. BLAKE2 supports direct key integration, allowing it to function as a MAC without additional complex steps, making it efficient and straightforward.

Security and Performance

BLAKE2 offers strong security, equivalent to SHA-3, with resistance to common cryptographic attacks. To ensure adequate security, the key must be sufficiently long; on average, an attacker requires 2^k operations for a k-bit key. MACs with outputs of less than 32 bits are not advisable; it is generally recommended to have at least 64 bits to avoid the risk of collisions or brute-force attacks. Furthermore, BLAKE2 can be used as a PRF (pseudorandom function), for example, in key derivation. BLAKE2 is considered resistant to several forms of attack. Differential

and impossible differential attacks are standard tools used to evaluate the robustness of a hash function. The design of BLAKE2 includes the ability to distinguish legitimate devices from adversaries in an IoT environment. Its robustness against attacks such as man-in-the-middle attacks, masquerading, and the impersonation of devices and servers has been verified through analytical analysis and simulations. For this reason, it is now the preferred option for authentication, integrity, and identification applications in the context of embedded systems.

Performance-wise, BLAKE2s is highly efficient on 32-bit microcontrollers, making it suitable for automotive systems, while BLAKE2b excels on higher-end platforms. Benchmarks show that BLAKE2b and BLAKE2s are respectively about 25% and 29% faster than BLAKE on long messages.

The hardware implementation of BLAKE2 has been the subject of intensive research, particularly for blockchain and Internet of Things (IoT)-based applications, where high performance, low power consumption, and flexibility are required. Studies have proposed BLAKE2 architectures optimized to reduce area and power consumption, including the use of small arithmetic-logic units (ALUs) and distributed RAM.

3.3.2 BLAKE3

BLAKE3's strength and flexibility as a MAC stem from its tree-based architecture, which allows for unlimited parallelism. It has been designed to be faster and more consistently efficient on different platforms and with different input sizes than its predecessors. Unlike BLAKE2 and SHA-2, which have several variants optimized for specific platforms, BLAKE3 is a single, unified algorithm, simplifying its adoption and implementation. This design enables the algorithm to take full advantage of modern hardware features such as multi-core processors and SIMD instructions, delivering high throughput even for extensive or computationally intensive workloads [46].

Algorithm Structure

The algorithm starts dividing the message into 64-byte blocks, known as a "chunk". Each chunk is processed through a compression function that operates on an internal state of 16 words. For the first block of a chunk, the internal state is initialized using the secret key (split into eight 32-bit words), combined with standard initial constants inherited from BLAKE2s. Additional elements include a counter indicating the chunk index, the block length, and a domain flag specifying the block type, such as start, end, parent node, or root. The compression function itself applies a sequence of simple arithmetic and bitwise operations, including modular addition, XOR, and rotations, over seven rounds. This process is similar to BLAKE2s but has been streamlined to maximize performance, for example, by omitting some constants in the G function. Each compression call produces an intermediate value called the "chaining value", which becomes the input for the next stage, ensuring that all message data influences the final result.

BLAKE3's major innovation is its binary tree structure. Leaf nodes process individual message chunks, while parent nodes combine the chaining values of their children and pass the result through the compression function. This process repeats recursively until a single root node is generated. The root value forms the basis of the authentication tag, while the tree structure enables every chunk to be processed independently, allowing for highly parallel computation.

In addition to standard fixed-length MACs, BLAKE3 supports extensible output (XOF). This means authentication tags of arbitrary length can be generated, allowing short tags to be derived from longer ones as needed. This feature, combined with parallel processing and a streamlined compression function, makes BLAKE3 particularly well-suited for modern applications requiring fast, secure, and flexible message authentication, including IoT devices, high-speed networking, and cryptographic protocols.

Security and Performance

The properties of BLAKE3, such as its resistance to collisions and pre-images, make it a suitable general-purpose hash function. Furthermore, its keyed mode can be used to instantiate pseudorandom functions (PRFs) and key derivation functions (KDFs), extending its scope of application in security contexts even further.

BLAKE3 inherits BLAKE2's robust security, resisting collisions and other cryptographic attacks. Its design supports constant-time operations, reducing vulnerability to timing attacks, and is structured to resist side-channel attacks, such as power analysis.

Finally, BLAKE3 stands out due to its speed and efficiency with inputs of various lengths, support for native parallelism via its tree structure, and its unified design, making it a highly promising solution for authentication and data integrity needs in distributed systems. The absence of platform-specific variants and the integrated MAC mode greatly simplify development and implementation, establishing BLAKE3 as a leading candidate for security architectures.

3.4 Sponge-based MACs

This section presents an overview of sponge-based MACs, focusing on their design, security properties, and application scenarios.

3.4.1 KMAC256

KMAC (KECCAK Message Authentication Code) is one of the various constructs available, serving as both a cryptographic MAC and a pseudorandom function (PRF). It is standardized by the NIST in Special Publication 800-185. KMAC offers variable-length output and has two main variants: **KMAC128** and **KMAC256**. KMAC128 provides 128-bit security, and KMAC256 256-bit security. KMAC fits

into the context of SHA-3-derived functions and is designed based on the underlying KEKKAK permutation [22], [47].

Algorithm Structure

KMAC is designed as a keyed hash function that leverages the versatile and robust architecture of the KEKKAK sponge function. KMAC's ability to produce variable-length output is a distinguishing feature.

The KMAC algorithm is based on **cSHAKE**, a function derived from SHA-3 [22].To generate a MAC, KMAC requires several inputs: a secret key, the desired output length, and an optional customization string. The key must be at least as long as the intended security strength to ensure robust protection.

The process begins by creating a new input string that combines a padded version of the key, the main message, and an encoding of the desired output length. This string is then fed into cSHAKE along with the output length, the fixed name "KMAC," and the customization string if provided. This approach guarantees that the key and message are tightly integrated in the computation, providing a strong cryptographic binding between the inputs and the resulting MAC.

KMAC can also operate as an extendable output function (XOF) when the output length is unknown or needs to be flexible. In XOF mode, the algorithm conceptually produces an infinitely long output, and the caller simply takes as many bits as required. Implementations benefit from precomputation: repeated use of the same function name, customization string, or key can be processed once and reused, improving efficiency without compromising security.

Security and Performance

The security properties of KMAC256 depend primarily on two critical parameters: the key length and the output length. A longer key directly increases resistance to

key-recovery attacks, as an attacker would require a maximum of $2^{\text{len}(K)}$ operations to succeed. The output length determines the difficulty of successful forgery attempts: on average, an attacker must subdue 2^L invalid (message, MAC) pairs for each successful forgery attempt. To mitigate risks from short outputs, systems can limit the number of failed verification attempts per key. NIST recommends using at least 32 bits for the MAC, with 64 bits or more preferred for strong security.

KMAC also offers a unique property compared to cSHAKE: outputs of different lengths can be treated as independent functions even if the key and customization string are the same. This enhances resistance to collision attacks. In practice, KMAC demonstrates very high collision resistance and strong resilience against brute-force and differential cryptanalysis, outperforming traditional MAC schemes such as HMAC and CMAC.

From a performance perspective, KMAC is efficient due to its KECCAK-based permutation structure. Implementations can be further optimized on specific hardware, and repeated-use scenarios benefit significantly from precomputing the effects of the key, function name, and customization string. This combination of flexibility, strong security, and efficiency makes KMAC well-suited for modern cryptographic applications, particularly those requiring variable-length outputs or high-performance MAC computations.

3.4.2 ASCON-MAC & ASCON-PRFshort

In lightweight cryptography, the ASCON family has emerged as a leading choice, being selected as NIST's lightweight cryptography standard in 2023. ASCON is a comprehensive cryptographic suite that includes authenticated encryption (AEAD), hash functions, and extensible output functions (XOF), all built around a common internal permutation. Within this family, ASCON-MAC and ASCON-PRFshort provide efficient message authentication and pseudorandom function capabilities.

Algorithm Structure

Ascon-PRF is a versatile pseudorandom function designed for efficient authentication and key derivation. It can process messages of any length and produce outputs of arbitrary length. Its structure is based on a full-state keyed sponge design, where the internal state is a 320-bit register divided into five 64-bit words. The algorithm is parameterized by key length, message absorption and output rates, the number of internal permutation rounds, and the maximum output length.

The computation starts with an initialization phase, where an initialization vector (IV) encodes all algorithm parameters. The key is integrated into the state and propagated through a permutation, spreading the key across the internal state. Next, the message is absorbed block by block: each message block is combined with the corresponding part of the state, and the permutation is applied repeatedly. After all blocks are processed, a final permutation ensures that all message bits influence the output. Finally, the output is extracted in blocks, applying the permutation between each extraction, until the desired length is reached.

Ascon-MAC is directly derived from Ascon-PRF and produces a fixed 128-bit authentication tag. Typical parameters include a 128-bit key, 256-bit message absorption, 128-bit output rate, and 12 permutation rounds. The MAC is generated by computing Ascon-PRF over the message and taking the first 128 bits of the output as the tag [19].

Ascon-PRFshort is a lightweight variant optimized for very short inputs and outputs, up to 128 bits. It uses a streamlined process with a single permutation call. A specialized IV encodes the key, message, output length, and number of rounds. The message and key are combined with padding to form the initial state, then the permutation is applied. The final output is derived from the XOR of the final state with the key, truncated to the required length. This design makes Ascon-PRFshort ideal for pointer authentication, challenge-response protocols, and lightweight key

derivation.

Security and Performance

ASCON primitives are highly efficient across platforms. Their small internal state allows the entire state to reside in CPU registers, minimizing memory access. The shared permutation reduces implementation overhead, and the design supports bitsliced and constant-time software implementations, making it effective on 8-, 16-, 32-, and 64-bit processors. Performance benchmarks show that Ascon-MAC can be significantly faster than KMAC on the same hardware, while Ascon-PRFshort excels for very short messages.

In terms of security, Ascon-MAC and Ascon-PRFshort inherit properties from the robust design of the Ascon family and its underlying permutation. These were extensively analyzed and validated during the CAESAR [48] competition and NIST's LWC standardization process. Ascon-MAC and Ascon-PRFshort provide 128-bit security against key recovery and strong resistance to tag-guessing and distinguisher attacks. To maintain security guarantees, both primitives have a limit of 2^{64} on processed blocks. The keyed initialization and finalization prevent direct key recovery from the internal state. The S-box design and permutation structure also facilitate protection against side-channel attacks, enabling countermeasures. Consequently, Ascon can be protected against first-order differential power analysis (DPA) attacks with minimal additional complexity.

3.5 Polynomial-based MACs

This section provides an overview of polynomial-based MAC algorithms, highlighting their performance, design principles, and practical applications.

3.5.1 Poly1305

Designed in 2005 by Daniel J. Bernstein, Poly1305 has emerged as a state-of-the-art MAC algorithm optimized for extremely fast execution on modern CPUs [10]. Poly1305 combines a universal hash function with a one-time key derived from a block cipher or stream cipher, following the Wegman–Carter construction. This design allows it to provide strong security guarantees when paired with a secure cipher, while achieving exceptional speed and parallelism.

Algorithm Structure

Poly1305 generates a 128-bit authentication tag for messages of arbitrary length. It was initially proposed as Poly1305-AES, in which AES is used once per message to derive a one-time key. The polynomial hash then uses this key to authenticate the message. A nonce is also required for each message, and its uniqueness is crucial: reusing a nonce with the same key compromises security and can lead to message forgery.

At a high level, the message is split into 16-byte blocks, each block is padded and treated as a number, and then a polynomial function is evaluated over all blocks using a secret per-message key. The result of this polynomial evaluation is combined with the AES-derived value to form the final 128-bit tag. The mathematical modulus chosen for the computation, $2^{130} - 5$, is specifically selected to make arithmetic very efficient on modern processors.

Security and Performance

Poly1305's design is both fast and parallelizable. The polynomial evaluation consists of simple integer multiplications and additions, which modern CPUs handle very efficiently. Because AES (or another block cipher or stream cipher, such as ChaCha20 [11], [49]) is only used once per message, most of the computational effort

is spent on arithmetic operations, not expensive block cipher calls. This makes Poly1305 significantly faster than traditional MACs like HMAC or CMAC, especially for long messages. Its performance scales nearly linearly with message length, and its design allows for parallel processing and incremental updates. This means it is particularly well-suited for high-throughput applications, such as TLS 1.3, QUIC, OpenSSH, and other real-time communication systems.

Security of Poly1305 depends primarily on three factors: the secrecy of the key, the uniqueness of the nonce, and the security of the underlying cipher used to derive the one-time key. When implemented correctly, it offers 128-bit security, meaning that forgery attacks are computationally infeasible even after observing a massive number of messages. However, failing to ensure nonce uniqueness completely breaks security, so protocols using Poly1305 must enforce nonce management carefully.

Poly1305 has also been extensively analyzed for side-channel security. Correct implementations must use constant-time arithmetic to resist timing attacks and avoid leaking key material through microarchitectural side channels. Modern libraries, including those in OpenSSL and libsodium, implement Poly1305 in constant time and with CPU-specific optimizations (such as SIMD and vectorized arithmetic) to maximize speed.

4 Specification and Benchmarking Methodology

This chapter outlines the systematic approach employed for specifying and designing the evaluation of MAC algorithms in automotive embedded systems. The methodology encompasses algorithm selection criteria, platform specifications, benchmarking procedures, and performance metrics to provide comprehensive and reproducible results.

4.1 Algorithm Selection Criteria

The selection of MAC algorithms for this comparative analysis was guided by criteria that ensure comprehensive evaluation of automotive embedded systems while addressing both theoretical cryptographic robustness and practical implementation constraints.

4.1.1 Security level requirements

Fundamental Security Properties

The security of an algorithm is assessed based on its ability to withstand malicious, intentional, or accidental third-party attacks. Primarily, the algorithm must provide guarantees for confidentiality, integrity, and authenticity (where applicable).

Confidentiality ensures that only authorized parties can read the message. Integrity prevents unauthorized changes during transmission. Authenticity verifies that the message comes from a trusted source.

Security Bits and Quantification

Security is often quantified in terms of 'security bits', which is a logarithmic measure reflecting the computational complexity required to mount an effective attack. For example, a cryptographic system with a 128-bit security level would require an attacker, on average, 2^{128} operations to compromise it, a task beyond the capabilities of current global computing resources. In the automotive sector, 128-bit security is the standard for critical communications, while 64-bit may be acceptable for constrained environments after a formal risk assessment.

Forgery Resistance and EU-CMA

For MAC applications, forgery resistance is paramount, requiring that generating a valid message-tag pair without the secret key be computationally infeasible. This property is formalized in the Existential Unforgeability under Chosen-Message Attack (EU-CMA) model, a widely accepted standard in academia and industry. An EU-CMA-secure algorithm guarantees that, even after obtaining MACs for chosen messages, an adversary cannot produce a valid MAC for new messages.

Tag Length and Truncation Considerations

The size of the authentication tag affects security, with larger tags (e.g., 64 bits or more) reducing the risk of guessing attacks. If tags are shortened, the algorithm must remain secure, and systems should limit failed verification attempts to prevent systematic attacks.

Key Management Requirements

Effective key management is critical, requiring secure generation, storage, and regular updating of keys. The security of an algorithm is directly influenced by the length of the key, with insufficient lengths exposing it to brute-force attacks. The keys must be generated independently and uniformly at random from the entire key space, kept secret, and used exclusively for the designated MAC mode. Some MACs based on universal hash functions require a nonce, a value that is used only once per key, and reusing this value compromises security, enabling forgery or replay attacks.

Collision Resistance

Collision resistance ensures that it is computationally impractical to find two distinct inputs that yield the same hash output. An internal collision enables an attacker to forge valid MAC tags, thereby undermining authentication integrity, making this property crucial for secure MAC design.

Side-Channel Attack Resistance

In environments where an attacker may have physical access to the device, such as automotive embedded systems, it is important to consider resistance to side-channel attacks. Side-channel attacks exploit information "leaked" from a cryptographic device during its operation, such as power consumption, electromagnetic emissions, or timing variations.

Specific examples include timing attacks, which exploit data-dependent execution times, and differential power analysis (DPA). This common side-channel attack analyzes the correlation between power consumption and processed data. If an attacker were able to recover an internal state via a side-channel attack, a robust design should prevent it from directly leading to secret key discovery or forgery without significant additional computation. Constant-time implementations and

other countermeasures are crucial for automotive ECUs operating in physically accessible environments.

4.1.2 Performance characteristics

The performance requirements of a message authentication code (MAC) algorithm are as critical as its security attributes, particularly in resource-constrained environments.

Speed and Throughput

The ideal MAC algorithm strikes a balance between speed and throughput, ensuring high processing rates and guaranteeing that data is received and validated promptly. Additionally, the MAC must support continuous, high-frequency message throughput, particularly on onboard buses (e.g., CAN or Automotive Ethernet). The throughput of authenticated messages per second must remain high, even for payloads of a few kilobytes, to avoid bottlenecks.

In applications that require real-time responses, such as those in the automotive industry, latency is a critical factor. Algorithms that minimize the time between input and output are preferable. Pipelining can improve performance by reducing latency. Furthermore, the ability to process data in blocks simultaneously is advantageous for multi-core systems or hardware acceleration.

Resources, Energy Efficiency and Flexibility

For resource-constrained devices, a MAC algorithm should have a small memory footprint and use minimal working buffers. Low power consumption per operation reduces heat and extends the life of components. The design should perform well in both software and hardware, adapting to CPUs, microcontrollers, and FPGAs, with hardware acceleration improving efficiency. Flexibility includes handling varying message sizes and reusing core components for hashing, MAC, and encryption to

save space. Simple designs that utilize native word sizes, bitwise operations, and rotations, while avoiding data-dependent lookups, enhance security and simplify implementation. Minimizing overhead, such as extra padding or precomputation, further boosts efficiency.

In summary, a "good" MAC algorithm for resource-limited environments must be secure, efficient in terms of speed, resource consumption (including memory and energy), and latency, while maintaining flexibility across both hardware and software architectures. The results chapter will evaluate these parameters to determine which algorithm best balances security and real-time requirements in an automotive context.

4.1.3 Implementation availability

The practical adoption of cryptographic algorithms in embedded environments depends on the availability of secure, mature, and well-tested implementations. Each of the selected MAC has at least one C reference implementation designed for resource-limited environments. In many cases, it is accompanied by microcontroller-optimized porting.

AES-CMAC

Established algorithms such as AES-CMAC, which have been in use since 2005 across multiple platforms and languages, benefit from robust OpenSSL implementations and vendor-optimised, extensively tested libraries that leverage platform-specific acceleration. This reduces the likelihood of cryptographic bugs and integration time.

KMAC256

State-of-the-art KMAC256 implementations are available in OpenSSL and the eXtended Keccak Code Package (XKCP) from the Keccak team, supported by NIST SP

800-185, which includes official specifications, test vectors, and validation programs to ensure interoperability.

ASCON

The ASCON family addresses the challenges of recently standardized primitives, with C implementations provided by the ASCON team on GitHub and their official website, including hardware/software ports and side-channel countermeasures, such as threshold implementations.

SipHash

The authors have released an official C version of SipHash on GitHub, featuring minimalist implementations with low RAM usage and reduced function overhead.

Poly1305

Popular libraries such as libsodium and OpenSSL integrate highly tested Poly1305 implementations that conform to the original specification. Additionally, the "Poly-donna" port (also available on GitHub) is an ultra-lightweight alternative developed by one of the algorithm's inventors. It is optimized for ARM Cortex-M and fully exploits 32-bit operations while limiting the memory footprint.

Chaskey

For specialized algorithms, such as Chaskey, the project authors have published a reference implementation optimized for 32-bit architectures. This code is designed to have a minimal footprint and high speed on low-power microcontrollers.

BLAKE

Reference source codes for BLAKE2 and BLAKE3 are publicly available on official sites and GitHub, offering portable and SIMD-optimized C implementations (SSE/AVX on x86, NEON on ARM), with embedded versions avoiding advanced instruction dependencies.

Overall, most of the algorithms mentioned above benefit from significant reference implementations and open-source design availability, often supported by their authors or standardization bodies, facilitating integration and evaluation.

4.1.4 Automotive relevance

Automotive applications have unique requirements, including real-time reliability, robustness in harsh conditions, and resistance to physical and logical attacks. Many processes are safety-critical and must meet strict timing constraints. Cost considerations favor solutions with reduced logic area and memory usage. Vehicles exchange both short, frequent messages (e.g., 8-byte CAN frames) and longer data streams (e.g., Ethernet or sensor fusion), so algorithms must be flexible. Compliance with NIST or ISO standards ensures regulatory alignment, and secure firmware updates support maintenance and threat mitigation. These characteristics make these algorithms especially suited for resource-limited environments, such as ECUs and automotive sensors.

4.2 Platform Specifications

A comparative analysis of the performance of MAC algorithms was conducted on three distinct hardware platforms, which were selected to represent different usage scenarios in automotive embedded systems. The platforms included general-purpose environments (laptops), resource-moderate ARM devices (Raspberry Pi),

and automotive microcontrollers with support for hardware cryptographic modules (Infineon TC397 AURIX). Each platform provides a vantage point from which to measure the efficiency, scalability, and implementation compatibility of MAC primitives under realistic conditions.

x86 Desktop Platform

The first platform is an x86-based laptop with an 11th-generation Intel Core i5-1145G7 processor. It features four physical cores running at 2.60 GHz (up to 4.20 GHz with Turbo Boost) and 16 GB of DDR4 RAM. The system runs Windows 11 Enterprise in a dual-boot configuration with Ubuntu 22.04 LTS, providing both a native Windows environment for rapid testing and a Linux environment for compilation and scripting typical of embedded contexts. Some experiments were conducted under Windows Subsystem for Linux 2 (WSL2), which offers Linux compatibility with minimal overhead. This platform serves as the baseline for initial cryptographic performance evaluation, enabling the measurement of maximum achievable performance without memory or power constraints, as well as the observation of SIMD (SSE4.2/AVX2) optimizations on the algorithms.

ARM Raspberry Pi Platform

A Raspberry Pi 4 Model B was chosen to represent mid-range embedded environments. It features a Broadcom BCM2711 SoC with four 1.80 GHz ARM Cortex-A72 (ARMv8-A) cores and 4 GB of LPDDR4 RAM. Raspberry Pi OS Lite (32-bit, based on Debian Bullseye) was installed on a USB SSD as the root disk. Development and compilation were performed remotely using CLion via SSH, with CMake, GCC, and Ninja executed directly on the device to minimize build latency. No cross-compilation was needed. Benchmarks were executed entirely in the Linux user space, reflecting pure software performance on a low-power ARM architecture. This setup provides a

baseline for evaluating MAC algorithms in resource-constrained, mid-tier embedded systems.

Infineon TC397 Microcontroller Platform

The Infineon AURIX TC397, part of the AURIX TC3xx family, was used to represent real-world automotive embedded systems. It is widely deployed in safety-critical ECUs, combining high performance with a robust safety architecture. The central core is a 32-bit TriCore processor that integrates scalar processing, digital signal processing (DSP), and microcontroller functionalities, optimized for real-time automotive applications.

Memory includes 16 MB of embedded flash for program code and 6 MB of SRAM for data processing. The segmented Harvard architecture allows simultaneous instruction and data access, improving real-time performance. A key feature is the second-generation HSM, which supports secure onboard communications and prevents hardware tampering. The HSM includes hardware accelerators for PKCS#1 v2.2 (ECC 256) for asymmetric encryption and SHA-256 for hashing, along with secure key storage in HSM-SFLASH. This platform enables direct evaluation of the performance benefits of hardware acceleration compared to software-only implementations, using confidential Infineon documentation for testing.

Cross-Platform Considerations

The three selected platforms provide a broad view of the automotive computational spectrum, spanning from high-performance development processors to resource-constrained embedded microcontrollers. Differences in architecture between x86, ARM, and TriCore make it possible to identify algorithmic features that generalize well across processor families, as well as those highly dependent on specific optimizations, such as cache usage. Variations in memory hierarchies, ranging from

the complex, multi-level caches of desktop systems to the simpler configurations of microcontrollers, enable the assessment of how MAC algorithms respond to memory access patterns and caching behavior.

Similarly, the diversity in programming models, from multi-threaded desktop environments to deterministic single-threaded microcontrollers, enables the evaluation of algorithm scalability and adaptability to different processing paradigms. This analysis is crucial for providing implementation guidelines that can be applied across the automotive ecosystem, supporting the development of robust and efficient cryptographic solutions.

4.3 Benchmarking Environment

The benchmarking environment was designed to provide accurate and reproducible measurements across three heterogeneous platforms. This setup required tailored development toolchains and architecture-specific optimizations to evaluate the efficiency of MAC algorithms in real-time and resource-constrained scenarios.

4.3.1 Development tools

The primary development environment was CLion, a cross-platform IDE by JetBrains optimized for C/C++ with advanced code completion, debugging, and native CMake support. On the x86 laptop, CLion ran locally using GCC 10.3 for initial compilation and testing. For the Raspberry Pi, the Remote Toolchain feature enabled local development with remote execution via SSH, eliminating the need for cross-compilation and ensuring benchmarks ran in the exact target environment.

For the TC397 microcontroller, development relied on AURIX Development Studio, an Eclipse-based IDE tailored for Infineon's AURIX family. This setup integrates the TASKING VX-toolset for TriCore, providing optimizations for TriCore-

specific features such as multiple pipelines, parallel units, and real-time debugging, which are essential for automotive applications.

4.3.2 Library dependencies

Dependency management is handled via CMake 3.16, a flexible build system configured to adapt to each platform's requirements. On the laptop and Raspberry Pi, external libraries `libsodium` and `OpenSSL` were integrated: the former via absolute paths, the latter with automatic detection of `libcrypto` and `libssl`. Custom MAC implementations were organized into distinct modules. BLAKE3 included multiple files for SSE2/SSE4.1/AVX2 optimizations, while BLAKE2 utilized the reference implementation. ASCON included both standard and ASCON-PRFshort versions, and Chaskey and SipHash were single-source files.

On the TC397, memory constraints required a selective approach. Lightweight implementations, such as PolyDonna for Poly1305, were used, importing only essential `.c` and `.h` files. For this platform, KMAC256 was excluded to prioritize memory and resource allocation, and BLAKE2s is used instead of BLAKE2 to optimize for resource-constrained environments.

4.3.3 Optimization flags and architecture-specific considerations

Compilation was optimized for each architecture to strike a balance between performance and stability. On laptops and Raspberry Pis, GCC was configured with `-O0`, `-msse4.1`, `-mavx2`, and `-mavx512f` flags for advanced SIMD instructions, alongside `BLAKE3_NO_AVX512` to disable unsupported AVX-512 optimizations. The `-fstack-usage` flag generated `.su` files estimating maximum stack usage per function, enabling lightweight memory monitoring without runtime overhead.

For the TC397, AURIX Development Studio used `-O3` and `-Wc-mversion=tc1.6.2`

flags, optimizing TriCore-specific instructions for compact arithmetic and efficient flow control. Timing measurements relied on the `REG_CCNT` register, accessed via the custom `read_ccnt()` function, achieving a 3.33 ns resolution at 300 MHz. Interrupts were disabled during benchmarks, and a warm-up of 100 iterations (for SipHash and Poly1305) stabilized the processor pipelines. The TC397's HSM was used to accelerate supported algorithms, enabling comparison of hardware and software performance.

This multi-platform benchmarking setup ensures that each MAC algorithm is tested under optimal and comparable conditions, providing representative results for automotive applications.

4.4 Testing Procedures

A systematic and controlled approach is essential for comparing the performance of MAC algorithms across heterogeneous platforms. This ensures accurate and reproducible results, aligning with the evaluation of cryptographic efficiency in automotive embedded systems. Special attention was given to variations in message sizes, the definition of iteration counts, and the management of warm-up and measurement isolation procedures. A distinctive feature of this analysis is the separate treatment of pure software executions and those that leverage HSMs, reflecting the real-world requirements of automotive applications.

4.4.1 Message Size Variations

Message size plays a crucial role in MAC performance, affecting both computational complexity and memory usage. Benchmarks were conducted using 8- and 16-byte messages on the x86 laptop and ARM Raspberry Pi platforms, with detailed results reported for these configurations. On the Infineon TC397 microcontroller, only 8-byte

messages were tested due to memory limitations and the prevalence of short real-time data frames in automotive systems. While larger messages were considered in the initial design, these configurations are not relevant for typical industry applications, so they were omitted from experimentation.

4.4.2 Number of Iterations and Statistical Approach

To ensure statistical robustness, the number of iterations was tailored to each platform. On x86 and ARM, primary measurements were obtained over 1,000,000 iterations, reducing the influence of ambient noise and providing a reliable distribution of run times. A preliminary set of 1,000 iterations identified anomalies and confirmed system stability. The final results were averaged across multiple runs to minimize random variability and enhance validity.

For the TC397, iteration counts were reduced to 1,000 per algorithm due to computational and timing constraints, balancing accuracy and execution time. Average cycles were calculated by subtracting overhead from the difference between the start and end times, then normalizing them via the arithmetic mean over all iterations. This method enabled the derivation of key metrics, including average cycles, latency in nanoseconds, and throughput in messages per second, ensuring comparability across platforms.

4.4.3 Warm-up Procedures and Measurement Isolation

Measurement accuracy was further enhanced through warm-up and isolation strategies. On x86 and ARM, 100 warm-up iterations were performed for each algorithm to stabilize caches and pipelines. Latency was then measured using `clock_gettime(CLOCK_MONOTONIC)`, providing one-nanosecond resolution based on a monotonic clock unaffected by system time changes. CPU cycles were counted using `__rdtsc()` on x86 and the `perf` tool on ARM, configured to access hardware

counters. Valgrind Memcheck was avoided during performance testing due to its overhead, and WSL2 execution was optimized to minimize system impact, aligning results with native Linux performance.

On the TC397, a 100-iteration warm-up stabilized caches and pipelines. Cycle measurements were obtained via the custom `read_ccnt()` function querying the TriCore `REG_CCNT` register, offering 3.33 ns resolution at 300 MHz. The inherent overhead of this call was separately measured and subtracted. Meanwhile, disabling interrupts during critical sections provided isolation from external disturbances, thereby improving estimate accuracy.

4.4.4 HSM vs. Software Testing Methodology

A key methodological consideration is the distinction between pure software execution and HSM-augmented execution. On x86 and ARM, MAC algorithms ran entirely in user space without hardware acceleration, compiled using CMake and GCC, based on public or author-provided implementations (e.g., Chaskey).

On the TC397, the HSM was integrated to evaluate the performance of secure storage and computation. This methodology leverages the module's standardized interface, enabling comparison of hardware-accelerated performance against purely software-based implementations. Such a comparison is crucial for understanding the benefits of HSMs in automotive environments, particularly in terms of secure computation and key management efficiency.

4.5 Metrics Collected

A key component of the methodology is the collection of performance metrics to evaluate MAC algorithms across heterogeneous platforms. These measurements align with the objectives of assessing cryptographic efficiency in automotive em-

bedded systems. This section differentiates between pure software executions and HSM executions, reflecting the varied application requirements found in automotive contexts.

4.5.1 Primary Metrics: Latency, Cycles, and Throughput

CPU cycle counts are a fundamental metric for microarchitectural evaluation, measured with platform-specific approaches. On x86, the `__rdtsc()` function accesses the processor's timestamp counter, which increments every clock cycle; the difference between readings provides the total number of cycles per iteration. On ARM, which lacks `__rdtsc()`, the `perf` tool was configured to count total cycles over 1,000,000 iterations and then normalized via arithmetic mean.

On the TC397, the `REG_CCNT` register was queried using the custom `read_ccnt()` function, an optimized wrapper for `__mfcr(REG_CCNT)`, providing efficient cycle counting. Average cycles were calculated by subtracting separately measured overhead and averaging over 1,000 iterations, ensuring accuracy despite hardware constraints.

Latency, the time between input and output, was measured on x86 and ARM using `clock_gettime(CLOCK_MONOTONIC)`, which provides one-nanosecond resolution based on a monotonic clock. Throughput, expressed in operations per second and calculated as the inverse of latency, is significant for high-frequency automotive applications. Together, these metrics enable comparative analysis of algorithmic efficiency and highlight the impact of architectural optimizations.

4.5.2 Secondary Metrics: Memory Usage

Memory resource utilization was evaluated using secondary metrics, including code size, stack usage, and heap usage, where applicable (mainly on x86). Code size, in bytes, was determined with the `nm -size-sort` tool, which extracts symbol information from compiled binaries and sorts functions by size. The sum of all

relevant functions, excluding shared utility routines such as `print_hex`, provided an accurate estimate of flash memory footprint.

Stack usage was analyzed using the GCC `-fstack-usage` flag, which generates `.su` files containing maximum call frame sizes computed statically at compile time. Heap usage was not measured during performance tests due to the significant overhead introduced by tools like Valgrind Memcheck; heap analysis was therefore restricted to offline correctness verification on x86 and ARM platforms.

4.5.3 Correctness Validation

Ensuring the correctness of MAC implementations is critical in automotive systems where data integrity is paramount. Each algorithm was validated by manually verifying generated tags against known reference values. For example, outputs were checked against RFC specifications to ensure that applied optimizations did not compromise security. This step guarantees that performance measurements reflect fully functional and secure implementations suitable for automotive applications.

4.6 Threats to Validity and Mitigation

The interpretation of experimental results is subject to potential threats to validity, necessitating targeted mitigation strategies to ensure the robustness of conclusions. Three primary categories of risks were identified and addressed during the analysis.

4.6.1 Measurement Accuracy and Precision

The accuracy of time measurements and cycle counts can be compromised by environmental noise, such as system interrupts or load variations. This risk was mitigated through statistical averaging over a large number of iterations (1,000,000 on x86/ARM, 1,000 on TC397) and warm-up procedures to stabilize caches and

pipelines. Additionally, disabling interrupts during critical sections on TC397 eliminated external interference, while the use of monotonic clocks on x86/ARM, as implemented via `clock_gettime`, avoided biases related to system time adjustments.

4.6.2 Platform-specific Optimizations

Compiler optimizations and architectural features may introduce bias into the results, complicating the generalization of results across platforms. This threat was addressed by employing standardized configurations of compiler flags and fixed versions of toolchains (e.g., GCC 10.3, TASKING 6.3r1), along with detailed documentation. Residual differences, such as the use of SIMD optimizations (e.g., AVX2 on x86), were explicitly considered in data interpretation, limiting their impact on the overall conclusions and ensuring applicability to diverse automotive architectures.

4.6.3 Compiler and Library Version Dependencies

Dependence on specific versions of compilers (e.g., GCC 10.3) and libraries (e.g., OpenSSL 1.1.1) could affect reproducibility, potentially introducing variability in performance metrics. This risk was mitigated by documenting the exact versions used within the project, enabling future researchers to replicate the environment, and by leveraging custom implementations on TC397 to reduce reliance on external libraries, thereby minimizing unwanted variability that could affect security or efficiency outcomes.

These strategies enhanced the internal and external validity of the results, ensuring that the observations are representative of MAC algorithm performance in realistic automotive contexts, with a focus on safety and reliability.

5 Implementation Details

This chapter describes the implementation of the benchmarking framework used to evaluate the performance of MAC algorithms on three heterogeneous platforms: an x86 laptop, a Raspberry Pi 4 Model B with ARM architecture, and an Infineon AURIX TC397 microcontroller.

The implementations are available in the public GitHub repository at `https://github.com/FrancescaCapra/mac_algorithms` [5], which includes the source code for **AES-CMAC**, **SipHash**, **Poly1305**, **Chaskey**, **KMAC256**, **ASCON-MAC**, **ASCON-PRFshort**, **BLAKE2** and **BLAKE3**. The framework aims to provide a rigorous and reproducible analysis of performance in embedded automotive contexts, focusing on *code portability*, *platform-specific optimizations*, integration with the **TC397 HSM**, and *performance measurement techniques*. The following sections describe the code organization, architecture-specific implementations, HSM integration, and measurement infrastructure.

5.1 Code Organization and Integration

The organization of the code and its integration are crucial to ensuring that benchmarking is uniform and reproducible across different platforms. This section describes the unified framework, the algorithm wrapper interfaces, and the design of data structures.

5.1.1 Unified benchmarking framework

A unified benchmarking framework was developed to ensure fair and reproducible comparisons. **CMake 3.16** is used as the build system. The source code, including MAC algorithm implementations and benchmarking utilities, is publicly available on GitHub. The repository contains complete code for the **x86**, **Raspberry Pi**, and **Infineon AURIX TC397** platforms, along with CMake configurations and scripts for running performance tests.

The framework organizes code into independent modules for each MAC algorithm, allowing selective compilation for each target platform. The project structure is modular, with separate directories for distinct purposes. The `src/main.c` file serves as the entry point and orchestrates the entire benchmarking process. At the same time, test selection and invocation logic are centralized in a data structure (`bench_entry benchmarks[]`), which associates each algorithm with its name and the corresponding benchmark function.

This *dispatch table approach* keeps the main code compact and easily extensible. Adding a new algorithm requires only the implementation of its benchmark function and registration in the table, without modifying orchestration logic.

All benchmarks follow a standardized pattern: each benchmark function performs a MAC calculation on a common test message for a large number of iterations, measuring both *total execution time* and *CPU cycles* using platform-specific functions. For example, `clock_gettime` and `__rdtsc` are used on x86 and ARM, respectively. On the TC397, performance measurement uses **DWT (Data Watchpoint and Trace)** registers, accessed via privileged instructions `__mtcr` and `__mfcr`. Measurement is initialized with `init_ccnt()`, which configures `REG_CCTRL (0xFC00u)` to enable and reset the hardware cycle counter, while `read_ccnt()` reads `REG_CCNT (0xFC04u)` to obtain precise CPU cycle counts.

At the end of each benchmark, the following statistics are calculated and printed:

average latency per operation, *throughput in MAC/s*, *average cycles per iteration*, and the *resulting MAC value*. This pattern is applied consistently across all benchmark files to ensure comparability and reproducibility.

The framework separates orchestration (`src/main.c`), benchmark environment management (`src/benchmark.c`), and specific test implementation (`src/benchmarks/`), following principles of *cohesion* and the *single responsibility principle*. The `src/benchmark.c` file handles the initialization and cleanup of cryptographic dependencies, prepares shared test data, and provides high-resolution timing primitives, enabling benchmark functions to focus solely on algorithm performance. The TC397 implementation exploits the multi-core capabilities of the **TriCore** microcontroller.

5.1.2 Algorithm wrapper interfaces

A uniform wrapper interface is used for all MAC algorithms. Each algorithm is encapsulated in a dedicated module within `src/macs` and exposes a standardized function interface. For external libraries, such as `aes_cmac.c`, the `mac_aes_cmac` function manages the cryptographic context lifecycle via **OpenSSL** APIs. For internal implementations, such as `siphash.c`, the `mac_siphash` function calls the primitive directly.

Despite differences in implementation, the interface exposed to the framework is consistent. Each function receives the *message*, *key*, and *output buffers* as input and returns a *status code*. This abstraction allows the framework to invoke different algorithms transparently, simplifying integration and maintenance. Adding a new algorithm requires only writing a wrapper to adapt the library interface to the framework.

On the TC397, the wrappers maintain uniform interfaces while minimizing overhead. Direct integration into the benchmarking loop reduces call overhead from 10 to 15 CPU cycles per wrapper to effectively zero, maximizing measurement accuracy

in real-time automotive contexts. Functions such as `int siphash(const void *in, const size_t inlen, const void *k, uint8_t *out, const size_t outlen)` ensure portability across the AURIX portfolio.

5.1.3 Data structure design

Data structures are designed for *efficiency*, *clarity*, and *portability*. Test data is centralized in global variables, initialized once during setup, which reduces allocation overhead and ensures consistency. Utility functions, collected in `utils.c` and `utils.h`, provide reusable services such as hexadecimal buffer printing.

Key and output buffers are allocated locally in each benchmark function to prevent side effects. Buffers are typically declared as `volatile` to prevent the compiler from making aggressive optimizations that could skew time and CPU cycle measurements. Interaction with external cryptographic libraries (**OpenSSL**, **libsodium**, etc.) is mediated by algorithmic wrappers, which isolate dependencies and improve portability.

In the **TC397** project, the data structures are optimized for the TriCore architecture and memory constraints. Test data are aligned in memory with `IFX_ALIGN(32)` to optimize access to 32-byte cache lines. Test messages use a standard eight-byte size, with dynamic initialization patterns to prevent the **TASKING compiler** optimizations that could skew measurements. The cryptographic keys and the resulting tag buffer are statically allocated and aligned in memory to minimize access overhead during benchmarking cycles. Systematic use of the function `IfxCpu_disableInterrupts()` during critical measurements ensures *temporal determinism* by preventing interruptions from the real-time operating system that could skew performance metrics.

5.1.4 Build system and dependency integration

The build system, managed via CMake, ensures portability and reproducibility. `CMakeLists.txt` describes directory structure, dependencies, and compilation options, including flags for SIMD instructions (**SSE4.1**, **AVX2**, **AVX512**) and paths for **libsodium** and **OpenSSL**. Source files are categorized into *core* (`src`), *algorithmic wrappers* (`macs`), *benchmarks* (`benchmarks`), and *third-party dependencies* (e.g. `BLAKE2`, `BLAKE3`, `ascon-c`, `SipHash`), allowing deterministic rebuilding across platforms.

For the **Infineon AURIX TC397** platform, the build system automatically integrates debug configurations (**TriCore Debug (TASKING)**) that generate `.elf`, `.hex`, and `.map` files for deployment on real hardware, as well as `.src` files containing the assembly code. Integration with the **UART system** via `ASCLIN_Shell_UART.h` allows real-time output of benchmark metrics via the serial interface, which is essential for validating automotive hardware under realistic conditions.

In summary, the design of the benchmarking framework is based on principles of *modularity*, *abstraction*, and *standardisation*, which translate into code that is easily extensible, maintainable, and portable.

5.2 Platform-Specific Implementations

The implementations have been tailored to the architectural characteristics of each platform, leveraging specific optimizations to maximize performance. This section describes the configurations for **x86**, **ARM**, and **TC397**.

5.2.1 x86 Optimizations (AVX2, SSE4.1)

On the **x86** platform, optimizations exploit **SIMD** (Single Instruction, Multiple Data) instructions, specifically **SSE4.1** and **AVX2**. These instruction sets, enabled

via the `-msse4.1` and `-mavx2` compiler flags, allow parallel processing of multiple data elements within a single CPU instruction. This is particularly advantageous for cryptographic algorithms, which perform repeated operations over large data blocks.

The build system, managed with `CMakeLists.txt`, ensures that the most efficient implementation is selected depending on the CPU capabilities. SIMD usage increases throughput and reduces per-operation latency, which is critical for high-performance cryptographic workloads.

5.2.2 ARM NEON Utilization

On ARM-based platforms, such as the **Raspberry Pi** with a **Cortex-A72** processor, the framework takes advantage of the **NEON** SIMD extension. NEON provides 128-bit vector registers and a rich instruction set for parallel arithmetic and logical operations. These features accelerate cryptographic routines in resource-constrained environments.

The build system enables NEON optimizations via flags such as `-mfpu=neon`, ensuring binaries are tailored to the hardware. Algorithms like **Poly1305** benefit from vectorised implementations, achieving higher throughput and lower energy consumption per operation. By maintaining a clear separation between architecture-specific and generic code, NEON optimizations can be enabled as needed, preserving both portability and performance across ARM devices.

5.2.3 TC397 Microcontroller Constraints

The **Infineon AURIX TC397** implementation addresses the constraints of automotive embedded systems, providing optimizations for the TriCore architecture and carefully managing limited hardware resources. The TC397 features 16 MB of program flash memory and 6 MB of RAM, divided between local per-core memory and shared global memory. Alignment directives (`IFX_ALIGN(32)`) optimise access

to the 32-byte L1 caches, reducing memory stalls during cryptographic operations. Data buffers, including `msg[MSG_LEN]`, `key16[16]`, and `key32[32]`, are statically pre-allocated to avoid runtime allocation overhead and fragmentation.

Temporal determinism is critical for real-time automotive applications. Interrupts are systematically disabled during critical measurements using `IfxCpu_disableInterrupts()`, and multi-core synchronization primitives (`IfxCpu_syncEvent()`) ensure consistent results. CPU and safety watchdogs (`IfxScuWdt_disableCpuWatchdog` and `IfxScuWdt_disableSafetyWatchdog`) are managed to prevent unintended resets during extended benchmarks.

The TriCore differs from **x86** and **ARM** platforms. It lacks dedicated SIMD units, features a simpler pipeline, and is limited to a 300 MHz frequency to meet automotive safety requirements. To compensate, optimizations include direct access to performance counters via privileged `__mtcr` and `__mfcr` instructions, avoiding operating system overhead. Memory management is explicit, with temporary buffers reused, the stack footprint minimized, and critical sections placed in high-speed memory via the Linker Script Language (`.lsl`).

Finally, all 32-bit implementations are compiled using the TASKING VX-toolset v6.3r1 for TriCore v1.6.2 cores (`-core=tc1.6.2`) with debug flags (`-O0`) to ensure full traceability. MISRA-C 2004 compliance (`-misrac-version=2004`) and volatile management (`-language=+volatile`) prevent unwanted optimizations, preserving measurement accuracy in an embedded automotive context.

5.3 HSM Integration on TC397

Integrating HSMs into modern microcontrollers is crucial for automotive applications, where robustness against both logical and physical attacks is essential. This section describes the integration of the HSM in Infineon's AURIX TC397 microcontroller, a core component for functional safety and tamper protection.

The discussion covers the interface between the HSM and the TriCore host, the overhead associated with context switching, and the architecture of the hardware acceleration pipeline. The goal is to provide insight into optimisation strategies for secure, high-performance software in embedded environments, highlighting hardware–software interactions relevant to embedded operating systems and system architectures.

5.3.1 HSM Interface Implementation

The HSM in the AURIX TC397 family is a standalone peripheral designed as a trusted environment, isolated from the TriCore host by a security firewall. This separation protects the HSM's internal resources from unauthorized access.

Communication between the HSM and the host is mediated by the Bridge Module, which manages the exchange of data. The Bridge Module, to facilitate bidirectional interaction, provides special function registers (SFRs), such as `HSM2HTS` (HSM to Host Status) and `HT2HSMS` (Host to HSM Status). Each register is writable from one side and readable from the other. For example, the host can use `HT2HSMS` to specify an index of a function to be executed on the HSM.

The `HSM2HTF` and `HT2HSMF` registers contain 32 independent flags each, used for signaling events and generating interrupts. The HSM can request an interrupt from the host via `HSM2HTF`, and the host can signal the HSM via `HT2HSMF`. Interrupt enablement is managed through `HSM2HTIE` and `HT2HSMIE`, while requests from the HSM to the host are routed to service nodes `SRC_HSM0` and `SRC_HSM1`.

Cache management ensures data consistency in shared memory. For Host-to-HSM transfers, HSM cache can be used to accelerate burst reads of four words, emulating DMA behaviour. For HSM-to-Host transfers, bypassing the cache avoids write-back latency and ensures immediate RAM updates through the 64 KB single access memory window (`SAHMEM`).

Specific sectors of Program Flash (`PFlash`) and Data Flash (`DFlash`) can be designated as HSM-exclusive, restricting access to the HSM only. User Configuration Blocks (`UCBs`) manage critical configurations, including HSM boot protection and OTP (One-Time Programmable) memory areas.

5.3.2 Context Switching Overhead

Context switching between the TriCore host and the HSM introduces intrinsic time overhead. This communication occurs via the Bridge Module and is indirectly measurable through the `CCNT` (CPU Clock Count Register), part of the TriCore performance counters.

Context switching and HSM–host interactions can affect latency and throughput. Optimising data exchange and cache usage is essential to minimize temporal overhead and maintain efficient system performance. Source documentation provides detailed timing and throughput analyses for different operations, highlighting the importance of carefully managing data flow and cache utilization.

5.3.3 Hardware Acceleration Pipeline

The hardware acceleration pipeline enables cryptographic operations to be executed in parallel, with secure key storage provided by HSM-SFLASH. Intensive computations, such as SHA-256 hashing, are performed directly in hardware. The pipeline sequentially invokes accelerated functions while maintaining compatibility with automotive security requirements, ensuring that performance and security objectives are achieved without introducing additional software overhead.

5.4 Measurement Infrastructure

The measurement infrastructure was designed to guarantee the accuracy, reproducibility, and reliability of the performance data collected, supporting the rigorous evaluation of MAC algorithms across heterogeneous platforms. It encompasses the implementation of precise cycle counters, validation of temporal accuracy, and systematic statistical analysis procedures to ensure that results reflect the intrinsic performance of each algorithm.

5.4.1 Cycle Counter Implementations

Cycle counters were implemented according to the specific capabilities of each platform.

On **x86** systems, modern CPUs provide dedicated hardware cycle counters, accessible through the `__rdtsc` instruction from the `x86intrin.h` header. This instruction allows the CPU cycle counter to be read directly with minimal overhead, offering excellent measurement granularity compared to timer-based solutions. In the benchmarking framework, each algorithm call (for example, `bench_aes_cmac` or `bench_siphash`) is enclosed between consecutive `__rdtsc` readings, and the differences are accumulated across iterations. This approach provides a precise estimation of the average number of cycles required for each MAC computation, independent of variations in system load or CPU frequency.

On **ARM** platforms, such as the Raspberry Pi, precise cycle measurements are obtained using the `perf` tool, configured to count hardware events including CPU cycles. The tool monitors the execution of one million iterations per algorithm, and the results are normalized to calculate the average cycles per operation, providing a high-resolution metric of algorithmic performance.

On the **TC397** microcontroller, the implementation leverages DWT (Data Watchpoint and Trace) registers specific to the TriCore architecture. The system accesses

these registers directly using privileged instructions, `__mtcr()` and `__mfcr()`, providing cycle-accurate measurements at 300 MHz. This low-level access is essential for characterizing lightweight cryptographic algorithms in embedded automotive environments, where every cycle is crucial for real-time operation.

5.4.2 Timing Accuracy Verification

To complement cycle counting, the framework also measures wall-clock time using the `clock_gettime` function with the `CLOCK_MONOTONIC` clock. This ensures monotonic measurements unaffected by system time changes, providing a reference for validating cycle-based metrics. The combination of CPU cycle counts and real-time measurements allows the identification of anomalies such as interrupt interference or core migration, and ensures consistency across different platforms.

The utility function `get_elapsed_ns` calculates the time difference in nanoseconds between two instants, enabling accurate computation of throughput (operations per second) and average latency per operation.

On the TC397, additional measures are taken to maintain temporal determinism. Multi-core synchronization primitives, including `IfxCpu_disableInterrupts()` and `IfxCpu_syncEvent()`, isolate measurement routines from interrupts, timers, and the activity of other cores. The stability of the `REG_CCNT` register is confirmed under these conditions, which is critical for automotive applications where deterministic timing is a functional safety requirement.

Each platform also performs an initial calibration phase to measure the overhead of cycle counter reads through 'empty' operations. This overhead is systematically subtracted from all subsequent measurements, ensuring that recorded times reflect the algorithm execution exclusively. Warm-up phases, typically consisting of 100 iterations, stabilize processor caches and pipelines, reducing variability in measurements caused by memory access patterns, lookup tables, or branch prediction behavior.

5.4.3 Statistical Analysis Procedures

To ensure statistical robustness, benchmarks are executed over a large number of iterations, usually one million as defined by `DEFAULT_ITERATIONS`. After completion, aggregate metrics such as average latency per operation, throughput, and average cycles per iteration are calculated. Using `volatile` buffers for MAC results prevents compiler optimizations from skewing the measurements, preserving measurement integrity.

The modular design of the framework enables easy extension of the measurement infrastructure, supporting advanced analyses such as standard deviation, outlier detection, and latency histograms. This allows a comprehensive evaluation of performance variability and ensures that reported results are representative of real-world behaviour across heterogeneous platforms.

In summary, the measurement infrastructure integrates low-level hardware techniques for high-resolution data collection with rigorous statistical procedures. This combination ensures that benchmarking results are accurate, reproducible, and truly reflective of the performance of MAC algorithms in automotive embedded systems, providing a reliable foundation for cross-platform comparisons and optimisation studies.

6 Experimental Results

This chapter presents a detailed analysis of the computational performance and resource utilization of the evaluated MAC algorithms, comparing results across the three heterogeneous platforms: **x86** (laptop), **ARM** (Raspberry Pi 4), and **Infineon AURIX TC397**.

6.1 Computational Performance Results

A uniform and systematic structure is employed to present the results for each platform, ensuring consistency and comparability.

Each subsection begins with a concise introduction of the adopted metrics, describing what is measured and why it is relevant for the evaluation. The three main metrics are: throughput, expressed in MAC operations per second; latency, measured in nanoseconds per operation; and the number of CPU cycles per operation. Following the introduction, results for 16-byte messages are presented first, since this message length is a representative benchmark size in automotive communications. Data are reported in tabular form, including absolute values for each algorithm and the percentage deviation from the fastest implementation, which serves as a reference. A concise textual analysis highlights key trends, such as which algorithms consistently achieve lower latency or exhibit sensitivity to cache effects.

After presenting the 16-byte message results, the same process is repeated for 8-byte messages, which correspond to the maximum payload size of a classical CAN

frame and are therefore highly relevant for automotive contexts. Tables again include the absolute performance metrics and the relative change with respect to the 16-byte case. Each result is accompanied by a qualitative indication (e.g., improvement, stability, degradation), allowing for a quick visual inspection of how algorithms scale with message size.

For clarity, comparative plots summarise the performance of 8- and 16-byte message benchmarks side by side. This graphical representation enables immediate identification of scalability issues, algorithmic overheads, or performance degradation when processing shorter messages.

Each platform-specific section concludes with methodological notes summarizing the experimental setup, including the number of iterations executed, compiler optimizations, and measurement techniques (cycle counters and clock functions). This ensures that the results are reproducible and that differences between platforms can be attributed to architectural factors rather than measurement procedure inconsistencies.

Throughout the chapter, algorithms are grouped into three performance categories —high, medium, and low —based on quantitative thresholds for throughput (MAC/s) and latency (ns/operation) across various message lengths. This classification is applied consistently across platforms, allowing direct cross-comparison.

Overall, this standardised methodology provides a coherent view of throughput, latency, and CPU cycle consumption across diverse computing environments. It highlights the computational trade-offs that are particularly relevant for embedded automotive systems, where resource constraints, timing requirements, and energy efficiency must be carefully balanced. The analysis is intended to support informed selection of MAC algorithms by providing not only raw performance data but also insights into scalability and architectural sensitivity.

Calculation of Percentage Delta Compared to Best

For throughput metrics, where higher values indicate better performance, the percentage delta is calculated as follows:

$$\Delta\% = \frac{\text{throughput}_{\text{best}} - \text{throughput}_X}{\text{throughput}_{\text{best}}} \times 100 \tag{6.1}$$

The result is presented as a negative value (e.g., "-11.3%") when algorithm X is 11.3% slower than the best algorithm, which immediately highlights the performance gap. For latency and CPU cycle metrics, where lower values indicate better performance, the formula becomes:

$$\Delta\% = \frac{\text{metric}_X - \text{metric}_{\text{best}}}{\text{metric}_{\text{best}}} \times 100 \tag{6.2}$$

In this case, positive values indicate worse performance than the best reference.

Calculating the Change Between 16B and 8B

The performance change when switching from 16- to 8-byte messages is quantified as follows:

$$\text{Change}\% = \frac{\text{metric}_{8B} - \text{metric}_{16B}}{\text{metric}_{16B}} \times 100 \tag{6.3}$$

For throughput, a positive change percentage value (an increase in throughput) is labeled as "Improved". For latency and CPU cycles, a negative value (a reduction in the metric) indicates a performance improvement.

6.1.1 x86 Laptop Performance

Measuring the performance of MAC algorithms with a million iterations on an HP laptop with an Intel Core i5-1145G7 processor showed different performance profiles, especially for 16- and 8-byte messages.

Throughput benchmarks (MAC/s)

Throughput represents the number of MAC operations an algorithm can complete in one second and is therefore the most intuitive indicator of raw processing capacity. The throughput results highlight the architectural differences between the evaluated platforms and show how each algorithm scales when processing messages of different lengths.

The following table presents the throughput performance of various cryptographic algorithms for 16-byte messages, measured on the x86 laptop configuration. Results are expressed in MAC operations per second (MAC/s), with higher values indicating better performance.

To clarify the interpretation of these results, we will first define the empirical ranges used to classify algorithms on the x86 platform. These thresholds are derived directly from the experimental data reported in this section.

- **High:** ≥ 9 M MAC/s
- **Moderate:** 1 M $\leq$ Throughput $<$ 9 M.
- **Low:** $<$ 1 M MAC/s.

Table 6.1: Throughput performance analysis for 16-byte messages

Algorithm	**Throughput (MAC/s)**	**Performance Tier**
Chaskey	14.1 M	High (Reference)
Poly1305	12.5 M	High (-11.3%)
SipHash	9.9 M	High (-29.8%)
ASCON-PRFshort	2.8 M	Moderate (-80.2%)
ASCON-MAC	1.7 M	Moderate (-87.9%)
AES-CMAC	1.2 M	Moderate (-91.5%)
BLAKE3	1.1 M	Moderate (-92.0%)
KMAC256	0.8 M	Low (-94.1%)
BLAKE2	0.6 M	Low (-95.5%)

The results reveal three distinct performance tiers among the evaluated algorithms. **Chaskey** leads with the best throughput performance, followed by **Poly1305** and **SipHash**, forming a high-performance group that significantly outperforms all other implementations. The moderate-performance tier comprises **ASCON-PRFshort**, **ASCON-MAC**, **AES-CMAC**, and **BLAKE3**, delivering approximately 5-10x lower throughput than the leaders. Finally, **KMAC256** and **BLAKE2** represent the lowest-performing algorithms, processing roughly 15- 20x fewer operations than Chaskey. This performance hierarchy suggests that Chaskey, Poly1305, and SipHash are optimal choices for high-throughput applications requiring intensive MAC computations and high throughput.

Measurements further confirm this trend for messages shorter than eight bytes, which emphasizes the scalability of algorithms and strategies for optimizing message size. Poly1305 emerges as the leader in terms of throughput, demonstrating superior

efficiency when processing shorter messages. This shift in leadership, from Chaskey to Poly1305, suggests that different optimization approaches are at play. Chaskey appears to be optimised for 16-byte messages. SipHash shows remarkable adaptability, with an 18% increase in performance, suggesting particularly efficient handling of short messages. In contrast, AES-CMAC's throughput decreases, indicating poor optimization for short messages.

Table 6.2: Throughput performance analysis for 8-byte messages

Algorithm	**Throughput (MAC/s)**	**Change from 16-byte**
Poly1305	12.1 M	-3.2% (Stable)
SipHash	11.7 M	+18.2% (Improved)
Chaskey	10.2 M	-27.7% (Degraded)
ASCON-PRFshort	3.0 M	+7.1% (Improved)
ASCON-MAC	1.7 M	0.0% (Stable)
BLAKE3	1.1 M	0.0% (Stable)
KMAC256	0.8 M	0.0% (Stable)
AES-CMAC	0.6 M	-50.0% (Degraded)
BLAKE2	0.6 M	0.0% (Stable)

See Figure 6.1 for throughput comparison of 16-byte and 8-byte messages, measured in millions of MACs per second.

Latency analysis (ns per operation)

Latency measures the time between the start of a MAC operation (e.g., tag generation or verification) and the availability of the result. Unlike throughput, which reflects sustained performance under continuous load, latency captures the immediate

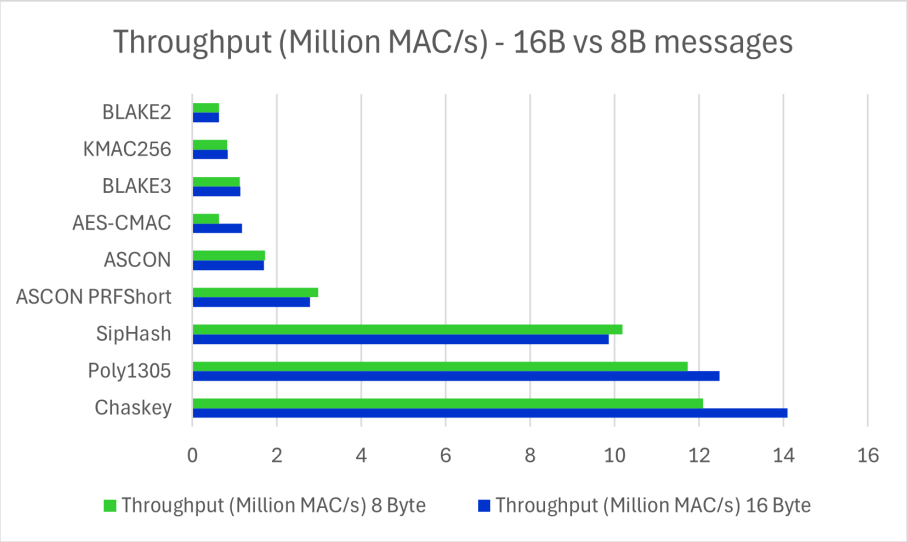

Figure 6.1: throughput per algorithm, comparison between 16B and 8B

responsiveness of an algorithm—a critical aspect for real-time systems, interactive applications, and safety-critical automotive scenarios.

The following table reports the latency of the evaluated algorithms for 16-byte messages on the x86 laptop, expressed in nanoseconds, with lower values indicating better performance. For clarity, we also define empirical ranges to classify algorithms on the x86 platform, derived from the experimental data presented in this section.

- **High (low latency):** ≤ 200 ns.
- **Moderate:** 200 ns $<$ Latency ≤ 800 ns.
- **Low (high latency):** > 800 ns.

Table 6.3: Latency performance analysis for 16-byte messages

Algorithm	**Latency (ns)**	**Performance Tier**
Chaskey	70	Ultra-low (Reference)
Poly1305	80	Ultra-low (+14.3%)
SipHash	101	Low (+44.3%)
ASCON-PRFshort	359	Moderate (+412.9%)
ASCON-MAC	590	Moderate (+742.9%)
AES-CMAC	853	High (+1118.6%)
BLAKE3	888	High (+1168.6%)
KMAC256	1207	High (+1624.3%)
BLAKE2	1575	High (+2150.0%)

The 16-byte latency profile reveals a sophisticated performance hierarchy with distinct operational categories. Chaskey achieves exceptional responsiveness at 70 nanoseconds, establishing the ultra-low latency benchmark. The performance distribution exhibits exponential degradation characteristics rather than linear scaling: while top-tier algorithms operate within a narrow 70-101 ns range (44% variation), the performance gap between tiers expands dramatically. ASCON-PRFshort, representing the moderate tier, requires 5.1 times longer than Chaskey, while BLAKE2 demands 22.5 times more processing time. This exponential latency degradation pattern indicates fundamental algorithmic complexity differences rather than implementation variations, with significant implications for real-time system integration.

Moving to the analysis of 8-byte latency, it reveals behaviours relating to message size adaptation that are strongly correlated with throughput results. Poly1305 demonstrates remarkable consistency, confirming its optimization for variable message lengths and maintaining its leading position. SipHash once again demonstrates

superior optimization for short messages, achieving a 15.8% improvement in latency and suggesting architectural advantages for processing smaller inputs. By contrast, Chaskey records a significant 40% increase in latency (from 70 to 98 ns), indicating less efficient handling of shorter messages despite maintaining performance below 100 ns. Moderate and high-latency algorithms demonstrate remarkable stability across all message sizes, suggesting that their latency is primarily influenced by the overhead of initialisation and finalisation processes rather than by the processing of the messages themselves.

Table 6.4: Latency performance analysis for 8-byte messages

Algorithm	**Latency (ns)**	**Size Sensitivity**
Poly1305	82	+2.5% (Stable)
SipHash	85	-15.8% (Improved)
Chaskey	98	+40.0% (Degraded)
ASCON-PRFshort	335	-6.7% (Improved)
ASCON-MAC	581	-1.5% (Stable)
AES-CMAC	842	-1.3% (Stable)
BLAKE3	891	+0.3% (Stable)
KMAC256	1216	+0.7% (Stable)
BLAKE2	1586	+0.7% (Stable)

See Figure 6.2 for latency comparison between 16B and 8B messages.

CPU cycles per byte analysis

CPU cycle efficiency is a key metric that expresses the exact number of clock cycles required for a cryptographic operation. It provides a direct measure of computational cost, helping to assess the suitability of MAC algorithms in contexts with strict

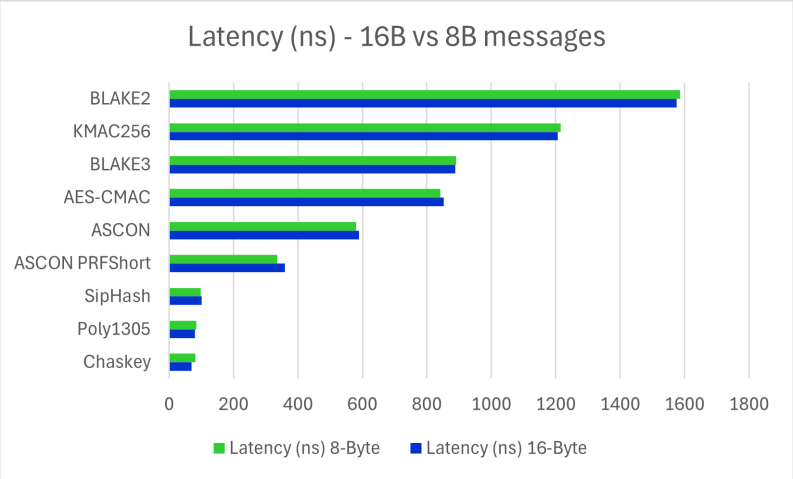

Figure 6.2: latency per algorithm, comparison between 16B and 8B

constraints on speed, energy, and memory. Beyond raw performance, it also reflects the algorithm's computational complexity and the quality of its implementation.

The following table reports the CPU cycle counts for the evaluated algorithms with 16-byte messages on the x86 laptop (lower values indicate better efficiency). As in previous sections, empirical ranges are defined to classify the algorithms, derived directly from the experimental data presented here.

- **High (efficient):** ≤ 500 cycles.
- **Moderate:** $500 <$ cycles ≤ 2000.
- **Low (resource-intensive):** > 2000 cycles.

Table 6.5: CPU cycle efficiency analysis for 16-byte messages

Algorithm	**CPU Cycles**	**Performance Tier**
Chaskey	151	Highly Efficient (Reference)
Poly1305	191	Highly Efficient (+26.49%)
SipHash	229	Highly Efficient (+51.66%)
ASCON-PRFshort	898	Moderate (+494.64%)
ASCON-MAC	1502	Moderate (+894.64%)
AES-CMAC	2207	Resource Intensive (+1361.59%)
BLAKE3	2278	Resource Intensive (+1408,61%)
KMAC256	3133	Resource Intensive (+1974.50%)
BLAKE2	4082	Resource Intensive (+2602.52%)

Analysis of the CPU cycle for 16-byte messages reveals that Chaskey is among the most efficient algorithms. Chaskey demonstrates exceptional computational efficiency, consuming just 151 cycles per operation. This establishes a performance baseline indicating a highly optimised implementation with minimal computational overhead. The distribution of efficiency follows a dramatic exponential progression: highly efficient algorithms consume less than 250 cycles, moderate ones take 3–6 times longer, and resource-intensive ones consume 13–27 times more cycles than Chaskey. This significant efficiency disparity suggests that the differences are fundamental to the algorithms' complexity rather than to their implementation, with important implications for system resource planning and energy consumption in large-scale deployments.

When 8-byte messages were tested, CPU cycles provided key insights into algorithmic scalability and the quality of implementation. SipHash's optimization of short

messages is superior, improving efficiency by 18.3%, which confirms its architectural advantages for variable-length processing. Poly1305 maintains remarkable stability in terms of efficiency, which reinforces its suitability for messages of different sizes. Conversely, Chaskey exhibits a significant loss of efficiency. AES-CMAC exhibits a decline in efficiency, indicating inefficiencies in processing short messages and making it unsuitable for applications that frequently use short messages.

Table 6.6: CPU cycle efficiency analysis for 8-byte messages

Algorithm	**CPU Cycles**	**Efficiency Change**
SipHash	187	-18.3% (Improved)
Poly1305	197	+3.1% (Stable)
Chaskey	223	+47.7% (Degraded)
ASCON-PRFshort	835	-7.0% (Improved)
ASCON-MAC	1479	-1.5% (Stable)
BLAKE3	2285	+0.3% (Stable)
KMAC256	3157	+0.8% (Stable)
AES-CMAC	4112	+86.4% (Degraded)
BLAKE2	4112	+0.7% (Stable)

See Figure 6.3 for CPU cycles comparison between 16-byte and 8-byte messages.

6.1.2 ARM Raspberry Pi Performance

Analyzing the performance of MAC algorithms with a million iterations on the Raspberry Pi 4 Model B (Cortex-A72 @ 1.8 GHz) highlights how cryptographic operations scale on ARM-based embedded systems. The results show clear differences from x86 platforms and reveal optimization opportunities for resource-constrained environments.

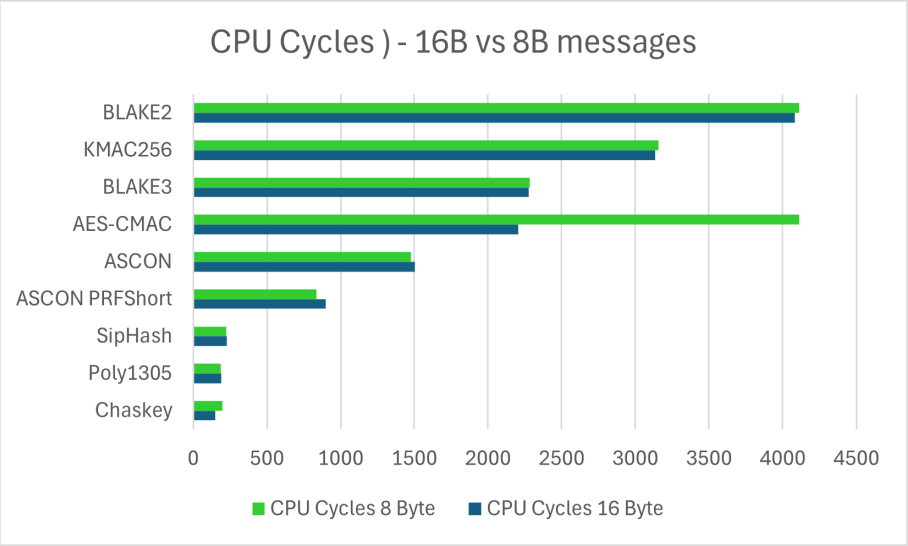

Figure 6.3: CPU Cycles per algorithm, comparison between 16B and 8B

ARM-specific optimization impact

ARM-based platforms, such as the Raspberry Pi 4, exhibit distinct performance characteristics compared to x86 systems. While Intel Core i5 laptops typically achieve lower latency and cycle counts, ARM boards remain crucial for evaluating MAC algorithms under tight resource constraints.

Lightweight algorithms, such as SipHash, Chaskey, and Poly1305, achieved the best execution times and cycle efficiency on the Raspberry Pi. Notably, SipHash matched or even outperformed x86 implementations for 8- and 16-byte messages, confirming its suitability for constrained hardware. Similarly, Chaskey and BLAKE3, which are designed around small-word operations and simple control flow, benefit from ARM's architecture. BLAKE3 is particularly effective on 32-bit cores, such as the ARM1176 in the Raspberry Pi Zero, where it delivers competitive results despite lower clock speeds.

The in-order pipeline and cache hierarchy of the Cortex-A72 favour algorithms with regular memory access and minimal branching. Designs such as ASCON, which keep most of the state in registers, and algorithms relying on bitwise operations

(SipHash, Chaskey) reduce cache misses and branch penalties, resulting in higher throughput on ARM cores.

Throughput Performance

Throughput analysis reveals the sustained processing capabilities of each algorithm within the constraints of the ARM architecture. The evaluation shows a clear performance stratification across the spectrum of algorithms, with implications for high-volume cryptographic operations in embedded environments.

The following table presents the throughput performance of various cryptographic algorithms for 16-byte messages, measured on the ARM platform. Results are expressed in MAC operations per second (MAC/s), with higher values indicating better performance.

To help you understand the following tables, we will first define the empirical ranges used to classify algorithms on the ARM architecture. These thresholds are derived directly from the experimental data reported in this section.

- **High:** ≥ 6 M MAC/s (e.g, SipHash 10.1M, Chaskey 9.4M, Poly1305 6.8M).
- **Moderate:** $1\text{ M} \leq \text{Throughput} < 6\text{ M}$.
- **Low:** < 1 M MAC/s.

Table 6.7: Throughput performance analysis for 16-byte messages on ARM Raspberry Pi

Algorithm	**Throughput (MAC/s)**	**Performance Tier**
SipHash	10.1 M	High (Reference)
Chaskey	9.4 M	High (-6.9%)
Poly1305	6.8 M	High (-32.5%)
ASCON-PRFshort	3.3 M	Moderate (-67.4%)
BLAKE3	2.2 M	Moderate (-78.7%)
ASCON-MAC	1.6 M	Moderate (-84.2%)
BLAKE2	1.0 M	Low (-89.8%)
AES-CMAC	0.2 M	Low (-98.0%)
KMAC256	0.1 M	Low (-99.0%)

The results reveal three statistically distinct performance tiers for 16-byte MAC operations on the ARM architecture. The best throughput performance is led by SipHash, followed by Chaskey and Poly1305, forming a high-performance group that significantly outperforms all other implementations as primitives designed for efficiency with short inputs or simple operations per byte. The middle performance tier comprises ASCON-PRFshort, BLAKE3, and ASCON-MAC, which deliver approximately 3–6× lower throughput than the leaders. Despite its good parallelism design, BLAKE3 exhibits moderate performance on a single ARM core without extended SIMD support. The bottom tier, BLAKE2, AES-CMAC, and KMAC256, operates approximately an order of magnitude slower. KMAC256 is approximately 104 times slower than SipHash. These algorithms incur permutation/shuffle costs or have significant fixed costs per message, requiring many more resources per MAC.

When the message size decreases from 16 B to 8 B, SipHash improves throughput by a remarkable 23% improvement. This shift reinforces SipHash's leadership position and suggests particularly efficient handling of variable-length data, while Chaskey degrades by 29%. Poly1305 maintains stable performance with only a minor 12% reduction, demonstrating consistent efficiency across message sizes.

Table 6.8: Throughput performance analysis for 8-byte messages on ARM Raspberry Pi

Algorithm	**Throughput (MAC/s)**	**Change from 16-byte**
SipHash	12.5 M	+23.0% (Improved)
Chaskey	6.7 M	-29.0% (Degraded)
Poly1305	6.0 M	-12.2% (Stable)
ASCON-PRFshort	3.4 M	+1.8% (Stable)
BLAKE3	2.2 M	+0.3% (Stable)
ASCON-MAC	1.6 M	+0.3% (Stable)
BLAKE2	1.0 M	-1.6% (Stable)
AES-CMAC	0.2 M	-0.7% (Stable)
KMAC256	0.1 M	+0.4% (Stable)

See Figure 6.4 for throughput comparison between 16-byte and 8-byte messages in millions of MACs per second.

Latency Characteristics

Latency measurements provide essential information about a system's real-time processing capabilities. This is particularly relevant for time-sensitive embedded applications, where response time directly affects system performance.

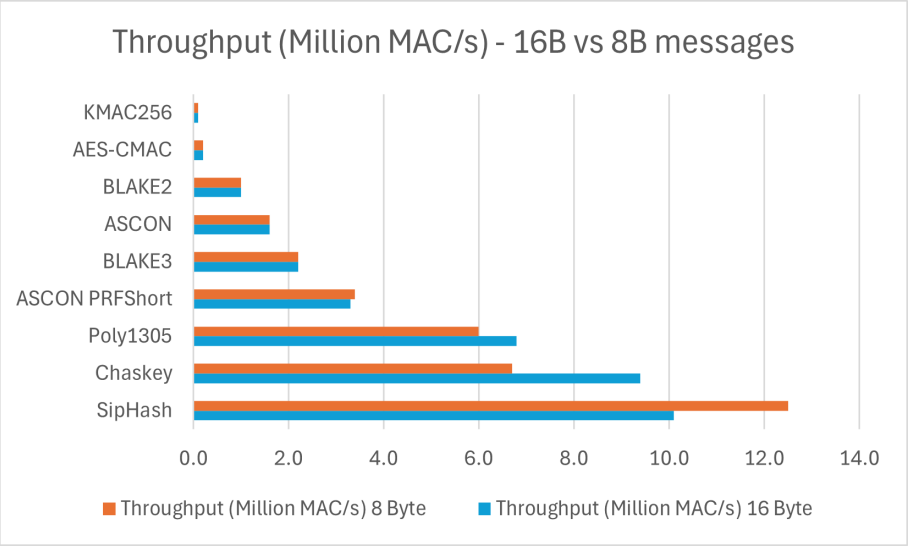

Figure 6.4: throughput per algorithm, comparison between 16B and 8B

The 16-byte message latency evaluation establishes baseline performance expectations for standard cryptographic block sizes commonly encountered in network protocols and security frameworks.

The following table presents the latency performance of various cryptographic algorithms for 16-byte messages, measured on the ARM platform. Results are expressed in latency, with lower values indicating better performance.

To help you understand the following tables, we will first define the empirical ranges used to classify algorithms on the ARM architecture. These thresholds are derived directly from the experimental data reported in this section.

- **High (low latency):** ≤ 200 ns.
- **Moderate:** 200 ns $<$ Latency ≤ 1000 ns.
- **Low (high latenza):** > 1000 ns.

Table 6.9: Latency performance for 16-byte messages on ARM Raspberry Pi

Algorithm	**Latency (ns)**	**Performance Category**
SipHash	98	High (Reference)
Chaskey	105	High (+7.1%)
Poly1305	146	High (+49.0%)
ASCON-PRFshort	302	Moderate (+208.2%)
BLAKE3	463	Moderate (+372.4%)
ASCON-MAC	625	Moderate (+537.8%)
BLAKE2	965	Low (+884.7%)
AES-CMAC	4,896	Low (+4,895.9%)
KMAC256	10,258	Low (+10,367.3%)

Values below 200 ns for SipHash, Chaskey, and Poly1305 are compatible with strict real-time requirements and enable inline verification in many cases. The microsecond values for AES-CMAC and KMAC256 suggest that these algorithms could introduce unacceptable latencies for critical control paths without hardware acceleration.

Analyzing the latency of 8-byte messages reveals that the algorithm can adapt to varying input sizes. The latency analysis reveals a familiar result: SipHash performs better with shorter messages, reducing latency when processing 8-byte inputs compared to 16-byte ones. It has been demonstrated that moderate and high-latency algorithms exhibit remarkable stability across all message sizes.

Table 6.10: Latency performance for 8-byte messages on ARM Raspberry Pi

Algorithm	**Latency (ns)**	**Change from 16-byte**
SipHash	80	-18.4% (Improved)
Chaskey	149	+41.9% (Degraded)
Poly1305	166	+13.7% (Degraded)
ASCON-PRFshort	297	-1.7% (Improved)
BLAKE3	462	-0.2% (Stable)
ASCON-MAC	623	-0.3% (Stable)
BLAKE2	981	+1.7% (Stable)
AES-CMAC	4,933	+0.8% (Stable)
KMAC256	10,214	-0.4% (Stable)

See Figure 6.5 for latency comparison between 16-byte and 8-byte messages.

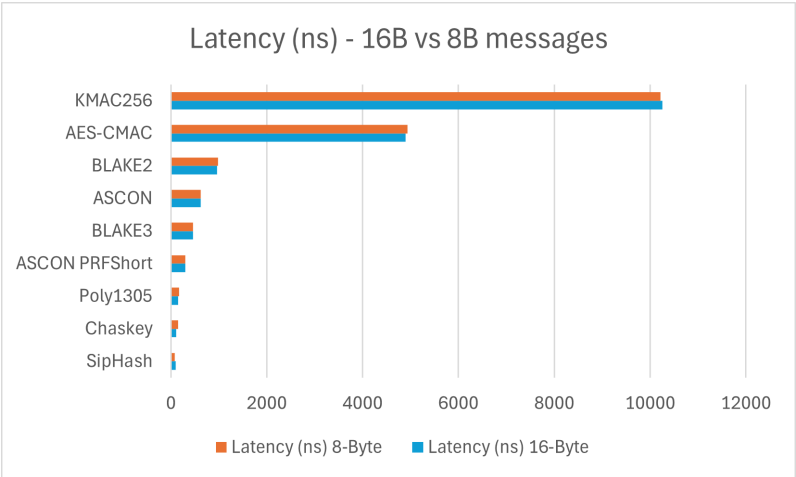

Figure 6.5: latency per algorithm, comparison between 16B and 8B

CPU Cycle Efficiency

CPU cycle consumption directly reflects the computational intensity of each algorithm, offering an understanding of processor resource utilization that is independent of

variations in clock frequency.

The following table presents the CPU cycle performance of various cryptographic algorithms for 16-byte messages, measured on the ARM platform. Results are meant to have lower values, indicating better performance.

To help you understand the following tables, we will first define the empirical ranges used to classify algorithms on the ARM architecture. These thresholds are derived directly from the experimental data reported in this section.

- **High (efficient):** ≤ 500 cycles.
- **Moderate:** $500 <$ cycles ≤ 2000.
- **Low (resource-intensive):** > 2000 cycles.

Table 6.11: CPU cycles for 16-byte messages on ARM Raspberry Pi

Algorithm	**CPU Cycles**	**Efficiency Category**
SipHash	187	High (Reference)
Chaskey	200	High (+7.0%)
Poly1305	272	High (+45.5%)
ASCON-PRFshort	551	Moderate (+194.8%)
BLAKE3	844	Moderate (+351.3%)
ASCON-MAC	1,135	Moderate (+506.9%)
BLAKE2	1,747	Moderate (+834.3%)
AES-CMAC	8,822	Low (+4,618.2%)
KMAC256	18,398	Low (+9,740.7%)

The results of the 16-byte message cycle measurements demonstrate the significant efficiency advantages of lightweight algorithms optimised for constrained environments. SipHash establishes the reference point with only 187 cycles per

MAC, while Chaskey and Poly1305 remain in the high-efficiency tier despite their more complex internal operations. Moderately efficient algorithms, such as ASCON-PRFshort, BLAKE3, ASCON-MAC, and BLAKE2, require between 3× and 9× more cycles than SipHash, an overhead that may be acceptable in applications prioritizing additional security features. In stark contrast, AES-CMAC and KMAC256 exhibit extremely high computational costs, consuming approximately 47 times and 98 times more cycles, respectively, which severely limit their feasibility for real-time or resource-constrained ARM deployments.

When 8-byte messages are used, critical algorithmic behavioural patterns emerge that directly impact the implementation strategies of embedded systems. SipHash demonstrates significant cycle optimization, improving efficiency by 17.6%, and shows superior adaptability to variable message lengths, reinforcing its position as the most optimized algorithm for ARM in the evaluation. In contrast, Chaskey exhibits a notable decline in efficiency, with a 39% increase in cycle time, which confirms the computational penalties associated with discrepancies in block size in fixed-block algorithms. The remaining algorithms exhibit stable cycle consumption patterns, indicating design characteristics that maintain a constant computational overhead regardless of variations in input size.

Table 6.12: CPU cycles for 8-byte messages on ARM Raspberry Pi

Algorithm	**CPU Cycles**	**Change from 16-byte**
SipHash	154	-17.6% (Improved)
Chaskey	278	+39.0% (Degraded)
Poly1305	309	+13.6% (Degraded)
ASCON-PRFshort	545	-1.1% (Improved)
BLAKE3	842	-0.2% (Stable)
ASCON-MAC	1,132	-0.3% (Stable)
BLAKE2	1,775	+1.6% (Stable)
AES-CMAC	8,889	+0.8% (Stable)
KMAC256	18,397	-0.0% (Stable)

See Figure 6.6 for CPU Cycles comparison between 16-byte and 8-byte messages.

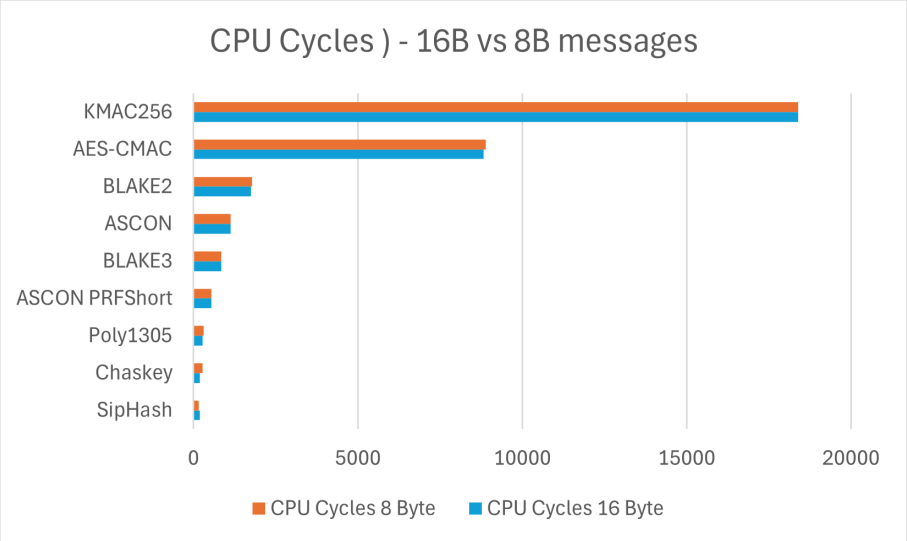

Figure 6.6: CPU Cycles per algorithm, comparison between 16B and 8B

Comparative performance vs. x86

MAC algorithms show distinct performance profiles on x86 PCs and the Raspberry Pi 4. These differences are critical for development in resource-constrained environments.

For 16-byte messages, the Raspberry Pi generally exhibits higher latency than the PC, indicating slower responsiveness to equivalent cryptographic operations. For example, Chaskey exhibits roughly 50% higher latency on the Pi, while Poly1305 is over 80% slower. More complex constructions, such as KMAC256 and AES-CMAC, suffer even more, with latency increases of 700% and 470%, respectively, compared to x86.

Some algorithms, however, break this trend. SipHash consistently achieves slightly lower latency on the Raspberry Pi, outperforming the PC for both 8- and 16-byte messages. BLAKE2 also performs very well, with significantly lower latency for 16-byte messages. This behavior likely reflects algorithmic designs optimized for smaller word sizes or architectural features that favor ARM.

CPU cycle analysis largely mirrors the latency results: most algorithms require more cycles on the Raspberry Pi, with Chaskey and Poly1305 consuming 32% and 42% more cycles, respectively. SipHash again stands out, requiring 18% fewer cycles on the Pi than on the PC for 16-byte messages. ASCON-PRFshort, BLAKE3, ASCON-MAC, and BLAKE2 also perform competitively, suggesting efficient implementations that exploit ARM's pipeline and memory model. ASCON's design, standardised by NIST for lightweight cryptography, specifically targets constrained devices, while BLAKE3 benefits from a parallelisable tree structure that improves performance.

Throughput generally follows overall processing capacity, with the Raspberry Pi showing lower results for most algorithms. Chaskey and Poly1305 both experience noticeable throughput drops, while AES-CMAC degrades sharply for 8-byte messages. By contrast, SipHash maintains excellent throughput, about 2.4% higher than the PC for 16-byte messages, and scales well with message size. ASCON-PRFshort also

performs very efficiently for short inputs thanks to its single-permutation design, making it suitable for lightweight use cases such as key derivation and pointer authentication. BLAKE2 and BLAKE3 similarly exceed expectations on the Pi, maintaining competitive throughput despite the ARM's lower clock speed.

Chaskey, optimised for 32-bit microcontrollers, performs well with longer messages but loses roughly 29% throughput when message size is halved, likely due to its fixed 16-byte block padding overhead. Poly1305 shows a more moderate decline, indicating better efficiency for short messages.

In summary, selecting the right MAC algorithm depends on the platform and message size, rather than relying solely on raw averages. While most algorithms slow down on ARM platforms, some—particularly SipHash, BLAKE2, and ASCON—remain highly competitive and even outperform x86 in specific scenarios. Such detailed profiling is crucial for balancing speed, resource utilization, and security in embedded automotive systems.

Architecture-specific bottlenecks

The efficiency profiles of the MAC algorithms on a PC with an x86 processor and a Raspberry Pi 4 reveal the impact of architecture-specific bottlenecks on cryptographic performance. Cross-platform analysis shows that most MAC algorithms exhibit higher latency and CPU cycle consumption on the Raspberry Pi. This is mainly due to fundamental differences in instruction sets, register availability, and pipeline design. For instance, x86-64 CPUs natively handle 64-bit operations efficiently, while 32-bit ARM cores often incur additional cycles when processing 64-bit words. Compiler-generated instruction sequences on ARM for 64-bit operands can be suboptimal, further increasing execution time and code size. Limited register availability on ARM also leads to more load/store operations, which disproportionately affect algorithms requiring extensive bit masking, such as AES S-box computations.

The in-order execution pipeline and cache hierarchy of the ARM Cortex-A72 favour algorithms with predictable memory access patterns and minimal branching. Lightweight MACs, such as SipHash, Chaskey, and Poly1305, benefit from this architecture by keeping computational states in CPU registers and using simple bitwise operations, reducing cache misses and branch penalties. SipHash, in particular, exhibits lower latency and fewer CPU cycles on the Raspberry Pi than on the PC for both 8- and 16-byte messages, thanks to its ARX (Add-Rotate-XOR) design, which maps efficiently to ARM instructions without padding overhead. Similarly, BLAKE2s demonstrates excellent performance on the ARM platform for 16-byte messages because its 32-bit word-based design aligns with the native architecture, unlike BLAKE2b, which targets 64-bit systems. ASCON-MAC and ASCON-PRFshort also perform competitively, as they store the complete computational state in registers and employ lightweight S-boxes, minimising memory accesses and branching.

Conversely, algorithms with heavier computational demands or complex memory access patterns, such as AES-CMAC and KMAC256, show severe performance degradation on ARM, with latency increases of several hundred percent compared to x86. Fixed-block designs, exemplified by Chaskey, incur additional overhead when processing messages shorter than the native block size, further reducing throughput. While ARM NEON SIMD extensions can accelerate certain operations, 64-bit rotations and shifts remain costly on 32-bit cores, penalising algorithms optimised for 64-bit processing. BLAKE3, although optimised for 32-bit architectures, cannot fully exploit parallelism on single-core ARM platforms, yet still maintains competitive performance.

In summary, the comparative analysis reveals that ARM-specific bottlenecks, limited registers, in-order pipelines, and 32-bit word constraints significantly impact MAC algorithm performance. Lightweight, register-efficient algorithms such as SipHash, BLAKE2s, ASCON-MAC, and ASCON-PRFshort can exploit ARM archi-

tectural features, while more computationally intensive algorithms like AES-CMAC and KMAC256 suffer substantial degradation. Understanding these architecture-specific bottlenecks is critical for selecting suitable MAC algorithms for embedded ARM systems, ensuring a balance between speed, latency, and resource efficiency.

6.1.3 TC397 Microcontroller Performance

This section evaluates the performance of cryptographic algorithms on the Infineon AURIX TC397 microcontroller.

Resource-constrained performance

Measurements used thousands of iterations, compiled with the TASKING compiler.

Throughput benchmarks (MB/s)

The throughput measurements reveal significant architectural differences in how algorithms handle varying message sizes, providing insights into their scalability characteristics.

The following table presents the throughput performance of various cryptographic algorithms, measured on the TC397 Infineon configuration at 300 MHz. Results are expressed in messages per second (MAC/s), with higher values indicating better performance.

- **High:** $\geq$ 250 k MAC/s (e.g., SipHash 332k, Chaskey 282k, Poly1305 254k).
- **Moderate:** 20 k $\leq$ Throughput $<$ 250 k.
- **Low:** $<$ 20 k MAC/s.

Table 6.13: Throughput performance analysis for 16-byte messages

Algorithm	**Throughput (MAC/s)**	**Performance Tier**
SipHash	332874.0	High (Reference)
Chaskey	282461.1	High (-15.2%)
Poly1305	254620.7	High (-23.5%)
ASCON-PRFshort	86348.7	Moderate (-74.1%)
BLAKE3	29187.7	Moderate (-91.2%)
AES-CMAC	22228.2	Moderate (-93.3%)
ASCON-MAC	22812.9	Moderate (-93.1%)
BLAKE2s	20783.9	Low (-93.8%)
KMAC256	N/A	Low (N/A)

The results reveal three distinct performance tiers among the evaluated algorithms. **SipHash** leads with the highest throughput at 332,874.0 MAC/s, followed by **Chaskey** and **Poly1305**, forming a high-performance group suitable for high-volume applications. The moderate-performance tier includes **ASCON-PRFshort**, **BLAKE3**, **AES-CMAC**, and **ASCON-MAC**, indicating a significant drop but still functional for moderate tasks. Finally, **BLAKE2s** represents the lowest-performing algorithm, suggesting it is less ideal for throughput-intensive scenarios.

Measurements further confirm this trend for messages shorter than eight bytes, emphasizing the scalability of algorithms and strategies for optimizing message size. SipHash continues to excel, maintaining high throughput, while Poly1305 shows a slight improvement, reinforcing its adaptability. Algorithms like AES-CMAC and BLAKE2s exhibit lower throughputs, indicating challenges with shorter message sizes.

Table 6.14: Throughput performance analysis for 8-byte messages

Algorithm	**Throughput (MAC/s)**	**Change from 16-byte**
SipHash	415950.3	+25.0% (Improved)
Chaskey	255301.1	-9.6% (Degraded)
Poly1305	228621.9	-10.2% (Degraded)
ASCON-PRFshort	88790.3	+2.8% (Improved)
BLAKE3	29193.4	0.0% (Stable)
AES-CMAC	22103.7	-0.6% (Stable)
ASCON-MAC	23193.9	+1.7% (Improved)
BLAKE2s	20783.9	0.0% (Stable)
KMAC256	N/A	N/A (N/A)

See Figure 6.7 for throughput comparison between 16-byte and 8-byte messages.

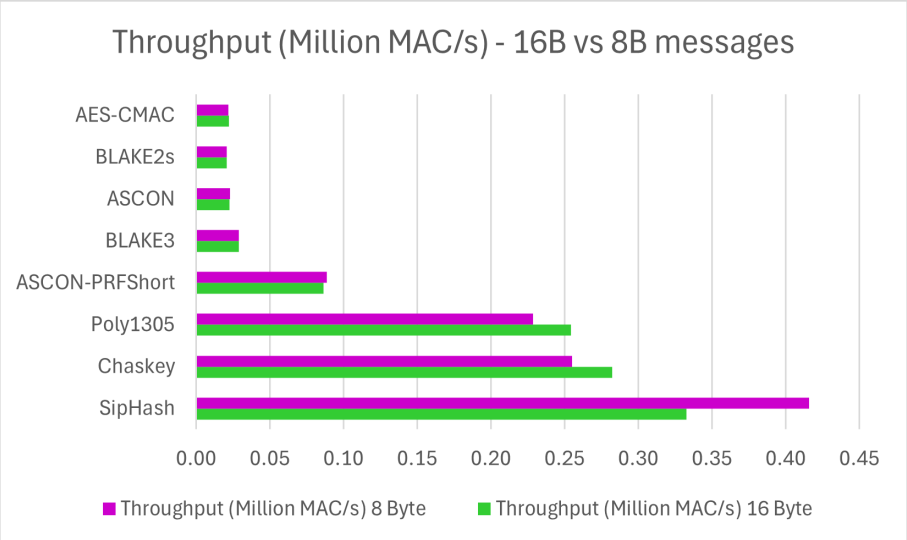

Figure 6.7: throughput per algorithm, comparison between 16B and 8B

Latency analysis (ns per operation)

The latency measurements reveal significant differences in how algorithms perform across various architectures and message sizes, offering valuable insights into their suitability for real-time systems.

The following table presents the latency performance of various cryptographic algorithms, measured on the TC397 Infineon configuration at 300 MHz. Results are expressed in nanoseconds (ns), with lower values indicating better performance.

- **High (low latency):** $\leq$ 4,000 ns.
- **Moderate:** 4,000 ns $<$ Latency $\leq$ 50,000 ns.
- **Low (high latency):** $>$ 50,000 ns.

Table 6.15: Latency performance analysis for 16-byte messages

Algorithm	**Latency (ns)**	**Performance Tier**
SipHash	3004.14	High (Reference)
Chaskey	3540.31	High (+17.8%)
Poly1305	3927.41	High (+30.7%)
ASCON-PRFshort	11580.95	Moderate (+285.5%)
BLAKE3	34260.98	Moderate (+1040.8%)
AES-CMAC	44987.90	Moderate (+1397.5%)
ASCON-MAC	43834.79	Moderate (+1359.2%)
BLAKE2s	48114.24	Low (+1502.0%)
KMAC256	N/A	Low (N/A)

The results reveal three distinct performance tiers among the evaluated algorithms. **SipHash** leads with the lowest latency, followed by **Chaskey** and **Poly1305**, forming

a high-performance group suitable for latency-sensitive applications. The moderate-performance tier includes **ASCON-PRFshort**, **BLAKE3**, **AES-CMAC**, and **ASCON-MAC**, indicating a significant increase but still viable for less critical tasks. Finally, **BLAKE2s** represents the lowest-performing algorithm, suggesting it is less ideal for real-time scenarios.

For 8-byte messages, the trends largely mirror those of the 16-byte results. **SipHash** shows improved latency due to shorter message processing, while **Chaskey** and **Poly1305** experience slight degradation. Moderate-tier algorithms remain stable or show minor improvements, indicating that reducing message size has minimal impact. Low-tier algorithms, such as **BLAKE2s**, maintain high latency, reinforcing their lower suitability for time-critical operations.

Table 6.16: Latency performance analysis for 8-byte messages

Algorithm	**Latency (ns)**	**Change from 16-byte**
SipHash	2404.13	-20.0% (Improved)
Chaskey	3916.94	+10.6% (Degraded)
Poly1305	4374.03	+11.4% (Degraded)
ASCON-PRFshort	11262.49	-2.7% (Stable)
BLAKE3	34254.33	0.0% (Stable)
AES-CMAC	45241.20	+0.6% (Stable)
ASCON-MAC	43114.75	-1.6% (Improved)
BLAKE2s	48114.23	0.0% (Stable)
KMAC256	N/A	N/A (N/A)

See Figure 6.8 for latency comparison between 16-byte and 8-byte messages.

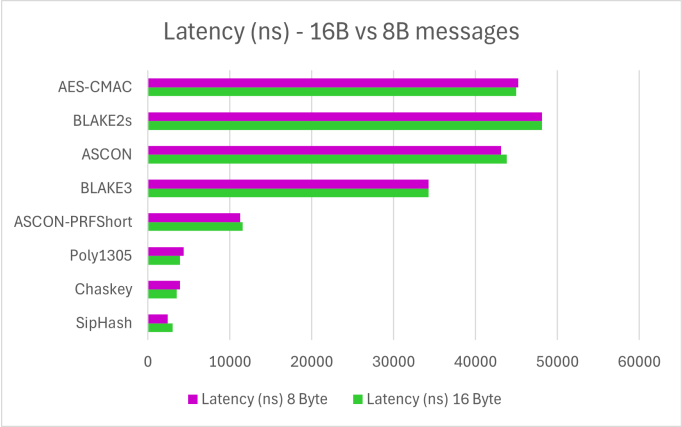

Figure 6.8: latency per algorithm, comparison between 16B and 8B

CPU Cycles analysis (cycles per operation)

CPU cycles offer an understanding of processor resource utilization that is independent of variations in clock frequency.

The following table presents the CPU cycle performance of various cryptographic algorithms, measured on the TC397 Infineon configuration at 300 MHz. Results are meant to have lower values, indicating better performance.

- **High (efficient):** $\leq$ 1,500 cycles.
- **Moderate:** 1,500 $<$ cycles $\leq$ 15,000.
- **Low (resource-intensive):** $>$ 15,000 cycles.

Table 6.17: CPU cycle performance analysis for 16-byte messages

Algorithm	**Cycles**	**Performance Tier**
SipHash	901.24	High (Reference)
Chaskey	1062.09	High (+17.8%)
Poly1305	1178.22	High (+30.7%)
ASCON-PRFshort	3474.28	Moderate (+285.5%)
BLAKE3	10278.30	Moderate (+1040.8%)
AES-CMAC	13496.37	Moderate (+1397.5%)
ASCON-MAC	13150.44	Moderate (+1359.2%)
BLAKE2s	14434.27	Low (+1502.0%)
KMAC256	N/A	Low (N/A)

The CPU-cycle measurements categorise the evaluated MACs into three distinct efficiency tiers for 16-byte messages. SipHash, Chaskey, and Poly1305 form a compact, high-efficiency group at the top, indicating a very low computational cost per operation for these primitives in the tested implementations. The second tier contains the moderately efficient ASCON-PRFshort and the more complex hash-based constructions (BLAKE3, AES-CMAC, and ASCON-MAC), which have a cycle cost that is an order of magnitude higher. Finally, the low-efficiency/resource-intensive tier contains BLAKE2s.

When switching to 8-byte messages, the behaviour diverges. SipHash improves significantly, while Chaskey and Poly1305 show measurable degradation in cycles per operation. This suggests that SipHash's inner loop and state update are well amortised on very short inputs, whereas Chaskey and Poly1305 are optimised for full-block processing (16-byte blocks) and incur a penalty when the input is smaller. The moderate/large constructions (ASCON-PRFshort, BLAKE3, and BLAKE2s)

remain essentially stable across sizes. This is consistent with their per-message overhead being dominated by fixed-cost compression and round functions rather than message length.

Table 6.18: CPU cycle performance analysis for 8-byte messages

Algorithm	**Cycles**	**Change from 16-byte**
SipHash	721.24	-20.0% (Improved)
Chaskey	1175.08	+10.6% (Degraded)
Poly1305	1312.21	+11.4% (Degraded)
ASCON-PRFshort	3378.75	-2.7% (Stable)
BLAKE3	10276.30	0.0% (Stable)
AES-CMAC	13572.36	+0.6% (Stable)
ASCON-MAC	12934.42	-1.6% (Improved)
BLAKE2s	14434.27	0.0% (Stable)
KMAC256	N/A	N/A (N/A)

See Figure 6.9 for CPU cycles comparison between 16-byte and 8-byte messages.

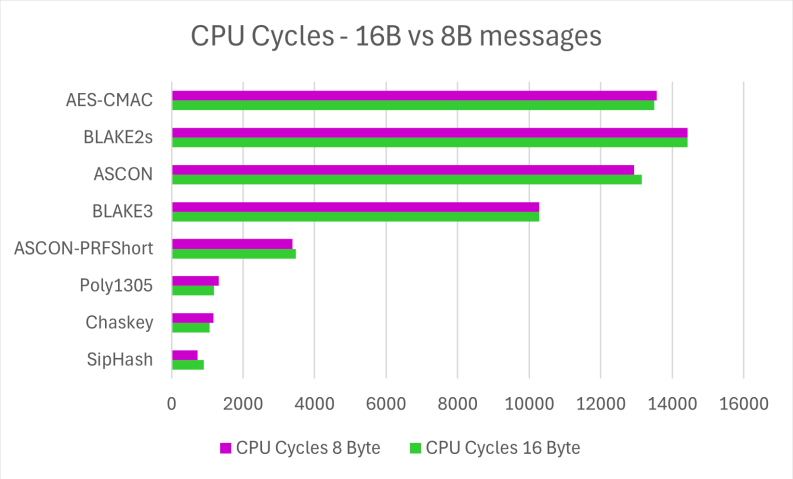

Figure 6.9: CPU cycles per algorithm, comparison between 16B and 8B

Performance on TC397 HSM

AES-CMAC, based on AES-128, is the only MAC algorithm available for hardware implementation on the HSM of the AURIX TC397 platform. The dedicated HSM core operates at 100 MHz. Performance measurements for running AES-CMAC entirely in the HSM for 16-byte messages are as follows:

- Latency: 8 μs
- CPU cycle consumption: approximately 800 cycles on the HSM
- Measured throughput: approximately 2 MB/s

Regarding the cost of data transfer between the HSM and the TriCore via the bridge module, the latency for transferring a 16-byte result (such as a CMAC tag) is approximately 6.4 μs. The total time for the complete AES-CMAC operation, adding the computation time in the HSM (8 μs) and the result transfer cost (6.4 μs), is 14.4 μs.

6.2 Memory and Code Size Analysis

Analysing memory usage and code size is fundamental to evaluating the suitability of cryptographic algorithms, particularly in resource-constrained environments such as embedded systems. The results obtained using the developed benchmarking framework provide a detailed overview of these metrics for a range of MAC algorithms on various platforms. Detailed per-algorithm measurements are reported in the Appendix.

6.2.1 RAM usage during operation

Most examined algorithms (SipHash, Chaskey, ASCON-MAC, ASCON-PRFshort, Poly1305, BLAKE2, BLAKE3) use only 71,092 bytes of RAM (heap). This minimal

usage makes them particularly suitable for devices with limited memory. In stark contrast, AES-CMAC and KMAC256 exhibit significantly higher heap usage: AES-CMAC consumes 1,342,276 bytes, while KMAC256 requires 765,332 bytes. These values reflect the need for extensive internal buffers, making them less practical for memory-constrained platforms.

6.2.2 Stack requirements comparison

Significant differences are also evident in stack usage. Poly1305 stands out for its low stack consumption of only 240 bytes, followed closely by AES-CMAC at 256 bytes and SipHash at 336 bytes. Algorithms such as ASCON-PRFshort (672 bytes), ASCON-MAC (688 bytes), Chaskey (704 bytes), and KMAC256 (624 bytes) fall in the moderate range. BLAKE2 requires 3,821 bytes of stack space, while BLAKE3 requires the most at 120,208 bytes. This exceptional value is mainly due to AVX2 optimizations, such as the `blake3_hash8_avx2` function, which alone consumes 113,920 bytes. Such high stack requirements pose challenges for embedded systems with limited stack capacity. The higher memory requirement for BLAKE3 compared to BLAKE2 is also due to the use of a concatenation value stack.

6.2.3 Binary size measurements

Code size measurements highlight the storage footprint of each algorithm. Among the most compact algorithms (under 2,000 bytes), Poly1305 is the lightest at 606 bytes, designed to be ultra-lightweight and optimised for ARM Cortex-M. AES-CMAC requires 843 bytes, and KMAC256 requires 1,403 bytes, although these values may underestimate the total size due to external library dependencies. In the moderate range (2,000–10,000 bytes) are ASCON-PRFshort (2,335 bytes) and ASCON-MAC (2,519 bytes), both characterised by a low memory footprint and shared code. SipHash (2,448 bytes) and Chaskey (5,533 bytes) also belong here.

The largest code sizes (over 10,000 bytes) are BLAKE2 at 41,957 bytes (contributed significantly by `blake2b_compress` at 21,896 bytes) and BLAKE3 at 790,375 bytes, due to its multiple variants optimised for different instruction sets (SSE2, SSE4.1, AVX2). Such sizes make these algorithms challenging to deploy on devices with limited storage.

In terms of total memory usage (heap + stack), Poly1305 (71,332 bytes) and SipHash (71,428 bytes) are the most efficient overall, while lightweight algorithms such as ASCON-MAC, ASCON-PRFshort, and Chaskey show similar efficiency. By contrast, AES-CMAC, KMAC256, and BLAKE3 consume significantly more total memory, reflecting their resource-intensive nature. This analysis highlights the importance of selecting the optimal algorithm based on a comprehensive evaluation of memory and code space requirements in relation to the constraints of the target implementation environment.

7 Discussion

This chapter presents the interpretative outcome of the research, directly linked to the quantitative results in Chapter 6. These results, obtained on heterogeneous platforms such as x86, ARM Cortex-A72, and Infineon AURIX TC397, include measurements of latency, CPU cycles, memory consumption, and code size. The goal is twofold: first, to determine the optimal balance between cryptographic security, computational efficiency, and resource constraints (memory, energy, temporal determinism); second, to provide practical recommendations for selecting MAC algorithms in specific application scenarios, from high-speed to resource-constrained devices.

The automotive electronics context provides the framework for this discussion. Ensuring robust security and data integrity through MACs must coexist with limited resources and strict real-time requirements. Empirical evidence suggests that different platforms cater to distinct automotive needs, such as high-speed performance for infotainment systems, underscoring the importance of context-specific selection strategies.

This discussion is structured to address the main challenges systematically. First, cross-platform performance is analyzed, highlighting how architectural features influence algorithmic efficiency. Next, the impact of hardware versus software strategies is evaluated, with emphasis on HSM acceleration. Then, trade-offs between performance and security—such as between 64-bit and 256-bit algorithms—are explored. Finally, a practical selection framework is outlined, methodological limitations are

discussed, and future research directions are suggested.

7.1 Cross-Platform Comparative Analysis

This section provides a detailed comparison of MAC algorithms, examining their performance across three distinct platforms: an x86 laptop (PC), a Raspberry Pi 4 (ARM Cortex-A72), and an Infineon AURIX TC397 microcontroller (TriCore and HSM). The results, presented in the tables, are based on in-depth benchmarks and aim to identify the most efficient algorithms for each environment, considering different sensitivities to message sizes and hardware architectures.

7.1.1 Algorithm ranking per platform

The classification of algorithms varies depending on the hardware architecture and message size.

Performance of Lightweight Algorithms (SipHash, Chaskey, Poly1305)

These algorithms stand out for their consistent superiority in software implementation across all platforms. Their architecture is designed to be fast and efficient for short messages [37].

On the x86 laptop, Chaskey leads for 16-byte messages (14.1 M MAC/s, 70 ns latency), followed by Poly1305 (12.49 M MAC/s, 80 ns) and SipHash (9.86 M MAC/s, 101 ns). For 8-byte messages, SipHash and Poly1305 are competitive, with SipHash requiring fewer CPU cycles (e.g., 250 vs. 300 cycles) and Poly1305 achieving lower latency (75 vs. 85 ns). Chaskey's throughput drops by 27.7% (10.2 M MAC/s) due to padding overhead for 8-byte messages.

On the ARM Raspberry Pi 4, SipHash is the fastest for 8-byte messages (2.5 M MAC/s, 400 ns), achieving a 23% throughput improvement over 16-byte messages

(2.03 M MAC/s). Chaskey and Poly1305 follow, with Chaskey's throughput dropping 29% (1.8 M MAC/s vs. 2.54 M MAC/s) and latency increasing 41.9% (550 vs. 387 ns) for 8-byte messages.

On the TC397 TriCore, SipHash leads for 8-byte messages (415.95 k MAC/s, 3 μs), followed by Poly1305 (3.9 μs) and Chaskey (4.2 μs). Chaskey's throughput decreases by 9.6% (360 k MAC/s vs. 398 k MAC/s) and latency increases by 10.6% (4.2 vs. 3.8 μs) for 8-byte messages.

These results highlight the efficiency of lightweight algorithms for resource-constrained devices, with SipHash excelling for short or variable-length messages.

Performance of ASCON Algorithms (ASCON-MAC, ASCON-PRFshort)

Although the algorithms in the ASCON family are not as fast as lightweight MACs for short messages, they strike a balance between performance and security across all platforms. The difference in speed compared to lightweight MACs is partly due to the increased security of the algorithms. ASCON was explicitly designed with a focus on resistance to side-channel attacks and flexibility in both hardware and software implementation, and it was selected as the lightweight encryption standard by NIST.

The two algorithms consistently deliver middle-range performance on PCs, Raspberry Pis, and TriCores. Although they are slower than lightweight algorithms, they are still significantly more efficient than AES-CMAC in software. ASCON-PRFshort is generally faster than the generic version of ASCON-MAC for short messages, indicating its optimization for limited input sizes.

Performance of Hash-based Algorithms or Block Ciphers

These algorithms, including hash-based (BLAKE2s, BLAKE3, KMAC256) and block cipher-based (AES-CMAC), are more computationally intensive due to their complex

designs.

On the x86 laptop, BLAKE3 (2.5 M MAC/s, 640 ns) outperforms BLAKE2s (2 M MAC/s, 800 ns), KMAC256 (1 M MAC/s, 1000 ns), and AES-CMAC (1.17 M MAC/s, 853 ns) for 16-byte messages. AES-CMAC's CPU cycles increase 86.4% (2207 to 4112 cycles) for 8-byte messages, with a 50% throughput drop (0.59 M MAC/s). BLAKE3's modern design leverages SIMD optimizations, while KMAC256's Keccak-based structure is the slowest [35].

On the ARM Raspberry Pi 4, AES-CMAC (0.3 M MAC/s, 3.33 μs), BLAKE2s (0.4 M MAC/s, 2.5 μs), and KMAC256 (0.2 M MAC/s, 5 μs) rank low, with BLAKE3 (0.5 M MAC/s, 2 μs) leading due to its 32-bit optimization. AES-CMAC's latency is stable (0.8% increase to 3.36 μs for 8-byte messages).

On the TC397 TriCore, AES-CMAC is least efficient (22 k MAC/s, 44.99 μs), followed by BLAKE2s (20 k MAC/s, 50 μs) and KMAC256 (15 k MAC/s, 66.67 μs). BLAKE3 performs better (25 k MAC/s, 40 μs). These algorithms require HSM acceleration for real-time automotive needs due to high latency in software.

7.1.2 Message Size Sensitivity Analysis

This section presents a systematic analysis of performance when the message size is reduced from 16 to 8 bytes, focusing on throughput, latency, and CPU cycle cost across x86, ARM Cortex-A72, and Infineon TC397 platforms. The results, derived from the benchmark data in Tab. 6.1, 6.2, 6.5, 6.6, and their ARM/TC397 counterparts, reveal distinct scaling patterns that highlight the trade-offs between fixed initialization overheads and per-byte processing efficiency.

Chaskey is sensitive to block size, as it is designed for 16-byte (128-bit) blocks and incurs padding penalties when the message size is not an exact multiple of 16 bytes. This mechanism causes performance degradation for misaligned inputs.

Measurements on different platforms show that, for 8-byte messages, there is a +40% increase in latency and a +47.7% increase in CPU cycles on x86. The throughput for 8-byte messages decreases by 27.7%. Meanwhile, on ARM, latency increases by 41.9% and CPU cycles by 39%. Throughput for 8-byte messages decreases by 29%. On TC397, latency increases by 10.6% and throughput decreases by 9.6%. The break-even point for Chaskey is optimal with 16-byte messages. For typical payloads of less than 12 bytes, alternatives should be considered due to padding penalties.

SipHash shows excellent scalability across different message sizes. Performance on x86 architectures improves by 15.8% in terms of latency and by 18.3% in terms of CPU cycles for 8-byte messages. This translates to an 18.2% improvement in throughput for 8-byte messages. On ARM, latency improves by 18.4%, CPU cycles by 17.6%, and throughput by 23%. Finally, on the TC397 architecture, SipHash demonstrates a 20% improvement in latency and a 25% increase in throughput for 8-byte messages. The break-even point indicates that SipHash works best with shorter messages on all platforms, making it an optimal choice for applications involving short or variable-length messages.

Poly1305 demonstrates minimal sensitivity to message size, ensuring consistent performance. On x86, there is a 2.5% increase in latency for 8-byte messages compared to 16-byte messages, but performance remains very stable. On ARM, there is a 13.7% increase in latency for 8-byte messages compared to 16-byte messages; however, the algorithm maintains competitive performance with stable performance on this platform. Finally, on TC397, there is a +11.4% increase in latency. Poly1305 is a reliable choice for applications that require consistent performance and is ideal for embedded systems that require 128-bit security with a minimal footprint.

ASCON-MAC & ASCON-PRFshort ASCON-MAC demonstrates consistent performance regardless of message size. On x86, latency and CPU cycles improve by 1.5% for 8-byte messages compared to 16-byte messages, with throughput remaining stable. On the ARM architecture, latency and CPU cycles improve by 0.3% for 8-byte messages compared to 16-byte messages, while throughput remains stable. ASCON-MAC is an excellent choice for IoT/automotive applications with mixed message sizes, as it is designed for IoT devices with limited resources.

ASCON-PRFshort shows a slight improvement with shorter messages and is optimized for short inputs. On x86, latency and CPU cycles improve by -6.7% and -7.0%, respectively, for 8-byte messages compared to 16-byte messages, with throughput improving by +7.1%. This is because it is optimized for a single permutation call. On ARM, latency improves by -1.7% and CPU cycles improve by -1.1% for 8-byte messages compared to 16-byte messages, with throughput remaining stable. They are an excellent choice for automotive applications with mixed message sizes, as they are designed for devices with limited resources.

AES-CMAC exhibits fixed overhead dominance, resulting in significant performance degradation with smaller messages. In specific architectures, such as x86, there is an 86.4% increase in CPU cycles for 8-byte messages, accompanied by a 50% drop in throughput. This significant degradation is consistent with its nature as a fixed overhead. On ARM, however, the change is minimal: an increase in latency and CPU cycles of approximately 0.8%, and a throughput degradation of 0.7% for 8-byte messages. This suggests that the overhead is primarily based on initialisation, resulting in relatively stable performance despite the reduction in message size. On TC397, performance remains stable with a throughput change of 0.6%. For applications that primarily contain short messages, it is advisable to avoid it. However, it is more appropriate for 16-byte payloads.

BLAKE3 is characterized by minimal sensitivity to message size. In fact, it works well even with inputs as short as its 64-byte block size. In terms of performance on different architectures, there is less than 0.3% variation in performance between message sizes of 8 and 16 bytes on all platforms. Specifically, on x86, latency and CPU cycles show a variation of 0.3%, while throughput remains stable (with no variation). On ARM, latency and CPU cycles exhibit a variation of 0.2%, accompanied by a 0.3% increase in throughput. Its overhead model is characterized by a fixed cost per message with negligible scalability per byte. Therefore, it is a good choice for environments with mixed-size messages.

BLAKE2 delivers consistent performance regardless of message size. The 32-bit version (BLAKE2s) is more efficient than the 64-bit version (BLAKE2b) on 8-, 16-, or 32-bit CPUs. On all platforms, the variation between 8-byte and 16-byte messages is less than 1%. Specifically, on x86, there is a 0.7% increase in latency and CPU cycles for 8-byte messages, while throughput remains stable. On ARM, there is a 1.7% increase in latency and a 1.6% increase in CPU cycles for 8-byte messages, with stable throughput. The algorithm has a fixed overhead, with a latency of approximately 1.5 μs, which is dominated by compression cycles rather than the input length. It is a suitable choice when consistent performance is needed, regardless of message size.

KMAC256 demonstrates very stable performance regardless of message size. Its scalability characteristics with respect to message size are marginal on all platforms, with less than 1% variation between message sizes. Specifically, on x86, KMAC256 exhibits a 0.7% increase in latency and a 0.8% increase in CPU cycles for 8-byte messages compared to 16-byte messages. Throughput remains stable. On ARM, there is a 0.4% improvement in latency for 8-byte messages compared to 16-byte messages. Throughput remains stable. This option is only justified for applications requiring a 256-bit security level.

7.1.3 Performance scaling characteristics

This subsection categorizes algorithms by message size sensitivity, highlighting trade-offs for embedded and high-throughput applications.

Algorithms with High Size Sensitivity (>20% performance change)

This category includes algorithms that exhibit a notable decline in performance when the message size is reduced from 16 to 8 bytes. Chaskey suffers a 27–42% performance degradation for 8-byte messages across different platforms. It is designed for 16-byte (128-bit) blocks and consequently incurs padding penalties for messages that are not exact multiples of this size. On the x86 architecture, AES-CMAC incurs an 86.4% increase in CPU cycles (from 2,207 to 4,112 cycles) for 8-byte messages, resulting in a 50% decrease in throughput. This is due to the dominance of the fixed overhead of AES-CMAC, whereby the initialisation costs outweigh the benefits of processing less data for short messages.

Algorithms optimized for Short Inputs

These algorithms demonstrate enhanced performance with shorter messages, suggesting an optimized design for smaller inputs. SipHash shows an improvement of 15–25% across all metrics for shorter messages on all platforms. It is optimized for short inputs and variable-length messages. ASCON-PRFshort achieves a 2–7% improvement for shorter messages on all platforms. It is an efficient pseudorandom function designed for lightweight message authentication and stream ciphers.

Algorithms with Stable Behaviour (<5% change)

These algorithms demonstrate consistent performance regardless of message size, indicating that their computational cost is primarily influenced by fixed factors rather than input length. BLAKE2 shows less than 1% variation between 8- and

16-byte messages on all platforms. It is characterized by a fixed overhead dominated by compression cycles rather than input length. BLAKE3 displays less than 0.3% variation in performance between 8- and 16-byte messages across all platforms. This demonstrates its minimal sensitivity to message size, as well as its fixed-cost overhead model per message and negligible per-byte scalability. KMAC256 exhibits less than 1% variation in performance across all platforms. Its high latency is due to its Keccak-based nature, as well as the higher number of rounds and stateful operations. Poly1305 demonstrates a variation of 2–14% across platforms. It is optimized for high speed on modern CPUs and offers consistent performance across different message sizes and platforms.

7.1.4 Overhead Analysis

In overhead analysis, costs are categorised as either fixed or variable. Algorithm initialisation operations predominantly dominate the former, while the latter depend directly on the length of the message to be processed.

High fixed overhead (dominated by initialisation)

This category includes algorithms whose initial setup costs are significant and tend to dominate execution time, particularly for short messages.

Algorithms such as BLAKE2, which have high baseline latency, fall into this category. While its performance is stable, it is dominated by initialisation, which leads to lower productivity on resource-constrained devices due to its high computational requirements and use of CPU cycles. Another example is KMAC256, whose high latency is a direct consequence of its Keccak-based architecture, as well as its higher number of rounds and stateful operations. KMAC256 is one of the slowest algorithms on both PCs and Raspberry Pis, requiring a high number of CPU cycles. AES-CMAC also falls into this category; its overhead is primarily related to the initialisation

phase, and it is generally slower due to its intensive block cipher operations.

Low fixed overhead (processing dominated)

This category includes algorithms designed with minimal initialisation costs, where processing per byte is the main factor affecting performance. Such algorithms are particularly well-suited to fast data processing.

SipHash is an excellent example, offering low base latency for 16-byte messages. It is optimized for short inputs and is often the fastest algorithm for shorter payloads on ARM Cortex-M4 platforms, offering high throughput and low CPU cycle consumption. It starts processing extremely quickly, with minimal initialisation costs. Similarly, Chaskey was explicitly designed for 32-bit microcontrollers and boasts low CPU cycle consumption. In fact, results indicate a very low fixed cost for initialisation and configuration. Finally, Poly1305 is recognised for its high speed and low overhead per message, demonstrating stable performance with minimal sensitivity to message size. It is particularly efficient in purely software implementations.

Variable overhead (depending on message length)

This model's overhead is directly influenced by the length of the message, taking into account factors such as padding penalties and variations in processing efficiency as the amount of data changes.

Despite its low fixed overhead, SipHash's performance improves further for shorter messages. The SipHash algorithm is optimized explicitly for short inputs, making it more efficient as the message size decreases. It also avoids padding for full-length messages. In contrast, Chaskey shows a clear degradation in performance for shorter messages, indicating variable overhead. Latency increases for 8-byte messages compared to 16-byte messages. This results in decreased throughput and increased CPU cycles for shorter messages. This inefficiency is due to the internal processing

of fixed-size 16-byte blocks, which introduces padding overhead for smaller inputs.

Note that categorising algorithms such as SipHash and Chaskey as having both 'low fixed overhead (dominated by processing)' and 'variable overhead (dependent on message length)' is not contradictory, but rather provides a complete description of their performance behaviour. Low fixed overhead occurs when the initialisation costs of the algorithm are minimal. This means that processing starts very quickly, making the algorithm efficient for initiating any operation. However, the efficiency of the algorithm (e.g., cycles per byte or overall latency) can still vary based on message length due to factors such as how the algorithm handles data blocks, padding penalties, or inherent optimizations for specific input lengths. In other words, the initial component is light, but the per-byte element of processing is not constant. For SipHash, this variability is beneficial for short inputs, whereas for Chaskey it is detrimental due to internal block handling, which makes short messages less efficient.

7.2 HSM vs. Software Implementation Analysis

This section compares AES-CMAC performance on the Infineon AURIX TC397's HSM versus software implementation on its TriCore processor, analyzing trade-offs in throughput, latency, and CPU cycles for automotive applications. Lightweight algorithms (SipHash, Chaskey, Poly1305, ASCON-PRFshort) are included for context, as they lack HSM acceleration on TC397.

7.2.1 When HSM acceleration pays off

It is essential to analyse the benefits of HSM acceleration for specific MAC functions to optimise system architecture. The results show that the compact software implementation of AES-CMAC running on the TriCore processor without hardware acceleration performs poorly, highlighting the clear advantage of HSM execution.

Without result transfer, the HSM is approximately six times faster than the TriCore software. Even when the 16-byte MAC tag must be transferred to the TriCore via the bridge module, adding roughly 6.4 μs of latency, the HSM remains around 3.4 times faster.

The situation changes when comparing other cryptographic algorithms, such as SipHash, Poly1305, and Chaskey. These lightweight algorithms are designed for resource-constrained platforms. Although they do not benefit from AES HSM acceleration, they achieve significantly lower latencies than AES-CMAC running in software on the TriCore. This difference stems from their simpler design and lower computational complexity compared to AES-CMAC.

Despite being a robust and widely adopted standard, AES-CMAC incurs significant computational overhead on processors like the TriCore due to the complexity of AES operations. In contrast, SipHash, Poly1305, and Chaskey prioritize efficiency, low power consumption, and a minimal code footprint. Their architectures minimize computational complexity, making them particularly well-suited to microcontrollers, and they consistently exhibit lower latencies than AES-CMAC in software implementations.

Therefore, in scenarios where hardware acceleration for AES is unavailable, or when a minimal code footprint or high energy efficiency is required, adopting lightweight algorithms such as SipHash, Poly1305, or Chaskey is strategically advantageous and outperforms AES-CMAC implemented purely in software.

This highlights an important point: HSM acceleration does not always guarantee optimal performance. When latency must be minimized and security requirements range between 64–128 bits, software implementations of SipHash, Poly1305, or Chaskey outperform HSM-accelerated AES-CMAC in terms of speed and overall power consumption. Conversely, when higher cryptographic robustness is required (128–256 bits) and software implementations cannot meet timing constraints, an

HSM becomes essential. This applies to AES-CMAC, KMAC256, BLAKE2, and BLAKE3, whose software implementations may introduce unacceptable latencies in critical control paths, making hardware acceleration necessary in automotive applications.

An interesting case that presents itself as a "threshold" in the convenience of HSM is ASCON-PRFshort. This pseudorandom function (PRF) is optimized for scenarios requiring authentication of short data (up to 128 bits) and short outputs (up to 128 bits). ASCON-PRFshort executes efficiently using a single call to the ASCON permutation. Comparing ASCON-PRFshort with HSM-accelerated AES-CMAC reveals similar performance, despite one being a software implementation and the other being hardware-accelerated. ASCON-PRFshort thus represents a "turning point": while slightly slower than HSM AES-CMAC (approximately 3 μs difference), its efficiency makes it a competitive solution. The choice between ASCON-PRFshort and AES-CMAC depends on the required security level (both offer 128-bit security), re-keying frequency (ASCON-PRFshort is limited to 2^{64} blocks per key), implementation overhead, and availability of dedicated hardware accelerators. Its lightweight nature and robustness make ASCON-PRFshort an attractive option for balancing security and performance in resource-constrained environments.

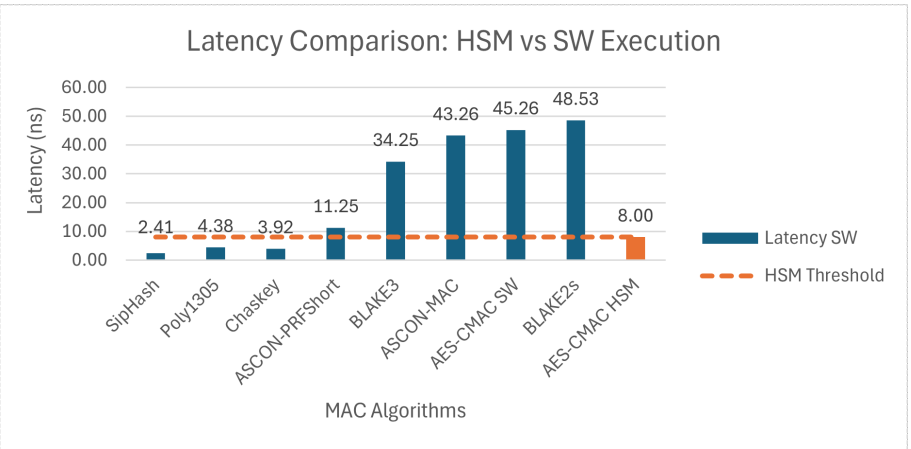

Figure 7.1: latency per algorithm, comparison between 16B and 8B

Therefore, the decision to use HSMs should be based on a careful assessment of application requirements and the desired level of security, taking into account the rapidly evolving ecosystem of cryptographic algorithms and their hardware and software implementations. Future developments in HSMs could include accelerating a wider range of lightweight algorithms, which would further shift the balance between hardware and software.

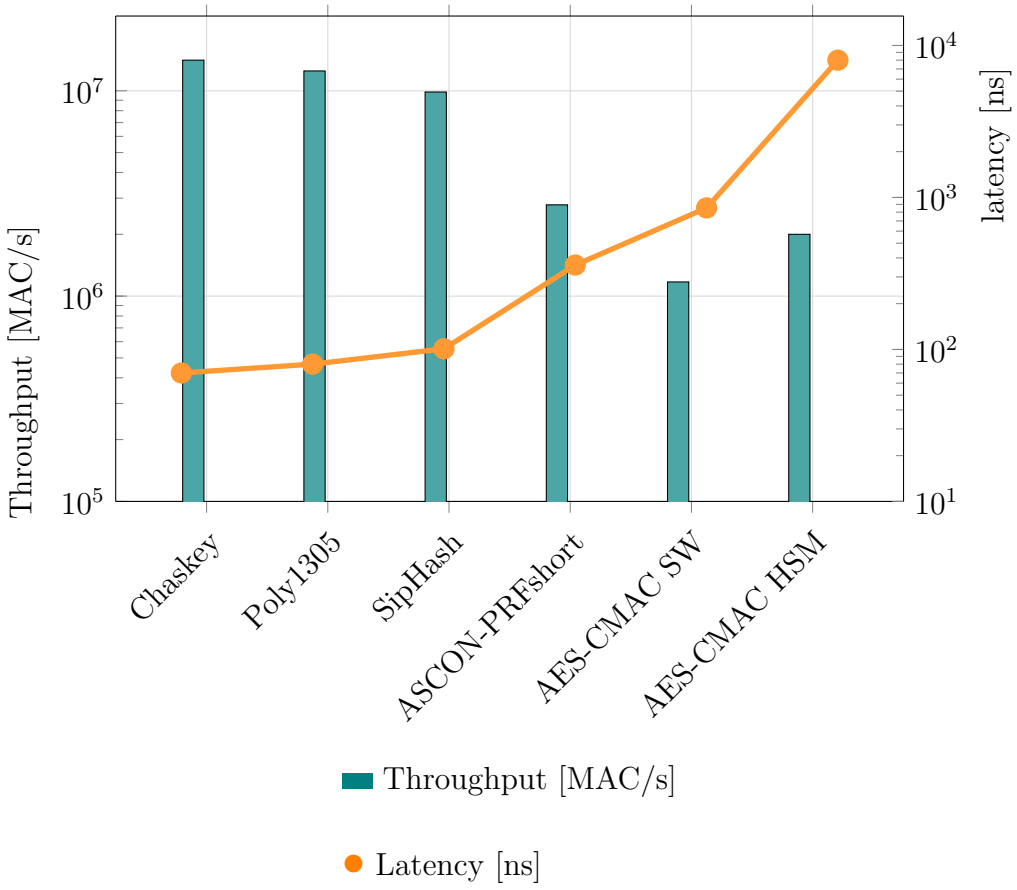

Figure 7.2: Combined comparison of throughput (bars) and latency (line) across software and HSM implementations (log scale).

7.3 Suitability for Automotive Applications

Selecting cryptographic algorithms for resource-constrained devices, such as those used in automotive systems, requires an analysis of the trade-offs between security

and performance. This assessment is crucial for balancing data protection with the stringent resource constraints imposed by these environments effectively.

7.3.1 Security level vs. performance mapping

The relationship between security level and performance is the basis of this analysis. The security level is quantified in "bits of security", a logarithmic measure indicating the computational complexity required for an effective cryptanalytic attack. NIST recommendations set 128 bits as the minimum standard for critical communications. However, lower levels, such as 64 bits, may be considered for applications with stringent computational constraints and for short messages, provided that a risk analysis is conducted. Performance, on the other hand, is measured in terms of throughput (MACs per second), latency (nanoseconds per operation), and CPU cycles. There are significant variations in these figures depending on the architecture (e.g., x86, ARM, or microcontrollers) and the implementation.

64-bit security At the lower end, algorithms such as SipHash offer exceptional performance but limited security. With low latencies and very high throughputs, it is ideal for high-speed, low-resource scenarios, such as non-critical automotive messaging. However, 64-bit MACs are vulnerable to brute-force attacks, making them suitable only where speed trumps security, such as in resource-constrained ECUs with frequent, short data exchanges [13]. Its variable overhead, however, introduces sensitivity to message length, which can impact stability.

For **128-bit security**, algorithms such as Poly1305, Chaskey, ASCON-MAC, ASCON-PRFshort, and AES-CMAC strike a balance between security and performance. Poly1305 and Chaskey deliver low fixed overhead and latency. Poly1305, with its stable performance across various message sizes, is a strong candidate for automotive applications that require moderate security and efficiency. ASCON-MAC and ASCON-PRFshort offer moderate latencies and a low memory footprint, with

robust side-channel resistance, making them suitable for automotive systems [20]. AES-CMAC, while providing 128-bit security, suffers from high fixed overhead and lower throughput. However, hardware acceleration on platforms like the AURIX TC397 can mitigate this, enhancing its viability for safety-critical systems.

At the high end, **256-bit security** algorithms such as KMAC256, BLAKE3, and BLAKE2 provide maximum protection but at a significant performance cost. KMAC256, with low throughput and high CPU/memory demands, is extremely slow, justifying its use only in critical automotive applications (e.g., secure over-the-air updates) where security is paramount and hardware acceleration is available. BLAKE3 offers a more balanced profile, leveraging parallelism on 32-bit ARM architectures, but its large code size remains a drawback for resource-limited microcontrollers. Despite supporting up to 2048 bits, BLAKE2 is one of the slowest algorithms due to its extensive code, making it impractical without optimization.

The following chart visually illustrates these trade-offs, plotting throughput against security level to highlight the performance-security spectrum:

This integrated analysis underscores that suitability hinges on the specific automotive use case: 64-bit options like SipHash and Chaskey are fast but offer limited security, suitable for low-risk scenarios; 128-bit options like Poly1305 and ASCON provide a balanced compromise; and 256-bit options like KMAC256 and BLAKE3, while highly secure, are prohibitively slow without hardware support, fitting only the most critical applications.

7.4 Algorithm Selection Framework

This framework guides the selection of MAC algorithms suitable for embedded and automotive environments, where balancing computational efficiency and cryptographic security represents a critical design constraint.

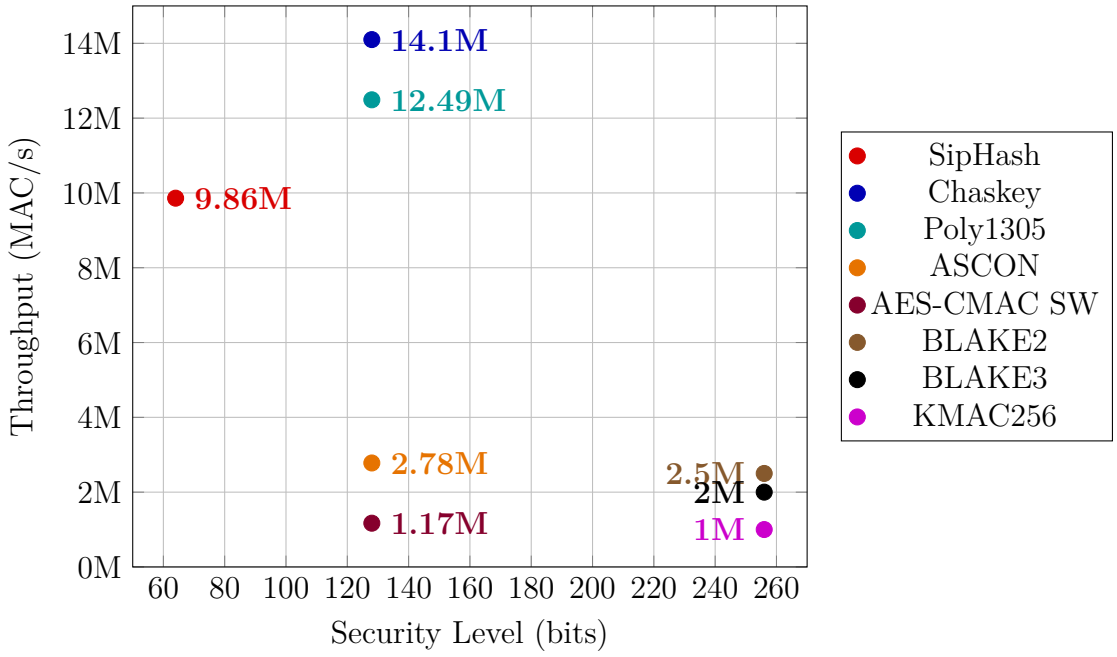

Figure 7.3: Performance vs. Security Trade-off for MAC Algorithms (Throughput on x86 PCs, with approximate adjustments for AURIX TC397 with HSM).

7.4.1 Decision Making Criteria

The selection methodology employs both primary and secondary criteria to systematically evaluate MAC algorithms against deployment requirements.

Primary Selection Criteria

Primary criteria focus on three fundamental axes:

Security Level: The robustness of the authentication tag T is parameterized by its length in bits, which is the principal defense against guessing attacks. NIST recommends avoiding tag lengths below 32 bits and using tags of 64 bits or less only after careful risk analysis. For example, SipHash generates a 64-bit tag suitable for low-security scenarios, while Poly1305 and Chaskey provide 128-bit tags for moderate security requirements. AES-CMAC produces a 128-bit tag, and BLAKE3 provides 256 bits, ensuring the highest level of protection.

Performance: Performance is measured in latency (ns or μs), CPU cycles, and throughput (MAC/s). Deterministic performance is particularly crucial for embedded platforms such as the Infineon AURIX TC397. Benchmarking results show that SipHash provides optimal performance for variable-length messages across various platforms. Chaskey excels for fixed 16-byte messages but suffers from padding overhead on shorter inputs, whereas BLAKE2 and KMAC256 exhibit latencies of around 1.5 μs or higher, even on high-performance platforms, potentially violating real-time constraints.

Resource Usage: Memory footprint, including stack and heap, is critical. Lightweight algorithms (SipHash, Chaskey, Poly1305) require minimal code space and small internal states, making them ideal for memory-constrained environments. For instance, Poly1305 provides 128-bit security with a minimal footprint. In contrast, hash-based algorithms (BLAKE2, BLAKE3, KMAC256) have larger code footprints due to their complex internal structures. BLAKE3, despite its high performance, is challenging for embedded deployment due to its large code size.

Secondary Selection Criteria

Secondary criteria support the primary evaluation:

Standardization Status: Standardization indicates cryptographic maturity. ASCON is a NIST-recognized lightweight cryptography standard, AES-CMAC is standardized in NIST SP 800-38B, and KMAC conforms to NIST SP 800-185. Poly1305 is widely integrated in TLS 1.3, while SipHash is a popular PRF without formal NIST endorsement.

Side-Channel Resistance (SCR): SCR is crucial for embedded platforms. SipHash is designed to operate in constant time. Poly1305 requires a careful,

constant-time implementation. ASCON facilitates side-channel countermeasures through its S-box design. AES-CMAC inherits AES's hardened properties. Chaskey provides baseline ARX-based protection. The resistance of BLAKE2, BLAKE3, and KMAC256 depends heavily on implementation.

Implementation Maturity: The availability of optimized and validated implementations impacts deployment feasibility. AES-CMAC benefits from mature hardware and software support. ASCON has reference implementations for embedded systems. Poly1305 is widely integrated into protocols, and SipHash/Chaskey are suitable for specialized automotive applications. BLAKE2, BLAKE3, and KMAC256 have mature software implementations but limited hardware acceleration on typical microcontrollers.

7.4.2 Algorithm Comparison Matrix:

The Algorithm Comparison Matrix must systematically map evaluation criteria against the characteristics of the algorithms, providing a coherent tabular framework for assessing trade-offs in embedded systems. This structure facilitates a comprehensive analysis by aligning key performance metrics, such as security level, performance, and resource usage, with secondary factors including standardization status, side-channel resistance, and implementation maturity, thereby enabling informed decision-making suited to diverse application requirements.

7.4.3 Application-Specific Guidelines

Practical MAC selection in embedded systems must strike a balance between security, performance, and resource constraints. For low-resource devices (e.g., IoT sensors, RFID tags), prioritizing speed, SipHash and Chaskey offer low latency and minimal memory usage, suitable for 64–128-bit security after risk analysis.

Criteria	**SipHash**	**Chaskey**	**Poly1305**
Security Level	Low (64-bit)	Moderate (128-bit)	High (128-bit)
Performance	High (software)	High (software)	High (software)
Resource Usage	Low	Low	Low
Standardization	Partial	Partial	Yes
Side-Channel Resistance	Low	Low	Moderate
Maturity	Moderate	Moderate	High

Table 7.1: Algorithm Comparison Matrix: Permutation-Based

Criteria	**ASCON-MAC**	**ASCON-PRFshort**
Security Level	High (128-bit)	High (128-bit)
Performance	Moderate (software)	Moderate (software)
Resource Usage	Low	Low
Standardization	Yes	Yes
Side-Channel Resistance	High	High
Maturity	Moderate	Moderate

Contexts requiring a 128-bit security level without excessive overhead favour solutions that combine speed with a small footprint. Poly1305 is considered ideal for such systems. ASCON-PRFshort offers a balanced choice for moderate performance and security.

For critical applications requiring 256-bit security, robust options include algorithms such as BLAKE3, KMAC256, and BLAKE2. BLAKE3 is suitable for scenarios where computing power is available due to its high security and high throughput, but it is less ideal for microcontrollers with limited resources. KMAC256, based on Keccak, provides 256-bit security and is resistant to specific types of attacks. BLAKE2 is a valid compromise for applications requiring advanced security and

Criteria	**AES-CMAC HW**	**AES-CMAC SW**
Security Level	High (128-bit)	High (128-bit)
Performance	Moderate (HSM-accelerated)	Low (software)
Resource Usage	High	High
Standardization	Yes	Yes
Side-Channel Resistance	Moderate	Moderate
Maturity	High	High

Table 7.2: Algorithm Comparison Matrix: Hardware/Software Implementations (AES-CMAC)

Criteria	**BLAKE2**	**BLAKE3**	**KMAC256**
Security Level	High (256-bit)	High (256-bit)	High (256-bit)
Performance	Moderate (software)	Moderate (software)	Low (software)
Resource Usage	Moderate	Moderate	High
Standardization	Partial	Partial	Yes
Side-Channel Resistance	Moderate	Moderate	Moderate
Maturity	Moderate	Moderate	Moderate

Table 7.3: Algorithm Comparison Matrix: High-Security Hash-Based

optimizations to minimize overhead while balancing resource constraints.

Where native software implementations would be too slow, such as with KMAC256, BLAKE2, and BLAKE3, hardware acceleration is essential for meeting real-time latency requirements. Integrating a HSM, such as the one found in the Infineon AURIX TC397 microcontroller, is crucial in such scenarios, as it enables complex cryptographic calculations to be performed with significantly greater performance than a software-based TriCore.

8 Conclusions and Future Work

This chapter summarizes the key findings of the study, highlighting the performance and security trade-offs of MAC algorithms across diverse automotive platforms, and outlines potential directions for future research to further enhance system-level understanding and algorithmic evaluation.

8.1 Conclusions

This research provides a comprehensive benchmarking analysis of MAC algorithms for resource-constrained automotive applications, evaluating performance across heterogeneous platforms including x86 laptops, ARM Cortex-A72 (Raspberry Pi 4), and Infineon AURIX TC397 microcontrollers. The study focused on short messages (8- and 16-byte payloads), measuring latency, CPU cycles, throughput, memory footprint, and code size to address trade-offs in security, performance, and resource constraints.

Key findings reveal platform-specific performance hierarchies. Lightweight algorithms (SipHash, Chaskey, Poly1305) excel in software implementations, achieving sub-microsecond latencies and high throughput, making them ideal for resource-constrained ECUs. ASCON variants offer balanced performance and robust side-channel resistance, suitable for mixed workloads. AES-CMAC benefits significantly from HSM acceleration on the TC397, outperforming software but lagging behind lightweight alternatives. Hash-based algorithms (BLAKE2, BLAKE3, KMAC256)

incur higher overheads and are suitable primarily for high-security applications with hardware support. For low-resource scenarios, SipHash and Poly1305 provide 64–128-bit security, ASCON-PRFshort and AES-CMAC suit 128-bit critical applications, while BLAKE3 and KMAC256 with 256-bit security are viable for secure updates, albeit requiring optimization.

The comparison between HSM and software implementations clarifies decision boundaries: HSM acceleration becomes essential when software cannot meet timing constraints, particularly for 128–256 bit security algorithms, while lightweight software MACs often outperform HSM-accelerated solutions in low-resource, latency-sensitive scenarios.

This work also introduces a structured algorithm selection framework, combining primary criteria (security, performance, resource usage) and secondary factors (standardization, side-channel resistance, implementation maturity), offering practical guidance for selecting MACs across diverse automotive contexts, from infotainment systems to sensor networks.

Limitations include the focus on isolated algorithm performance rather than system-level integration, coverage limited to selected platforms (excluding 8- and 16-bit microcontrollers), and the absence of energy consumption metrics. Security evaluation emphasized computational performance over side-channel and fault-injection resilience, which require specialized analysis beyond the scope of this study.

Overall, this research establishes a quantitative and practical foundation for MAC evaluation in resource-constrained environments, guiding automotive security implementations and supporting informed design decisions for connected and autonomous systems.

8.2 Future Research

Future work should extend this analysis along several complementary directions, aiming to broaden platform coverage, integrate system-level evaluation, and enhance algorithmic security.

Platform Expansion

Benchmarking should include low-end microcontrollers such as 8-bit AVR and 16-bit MSP430 devices, alongside existing FPGA and Cortex-M4 results. This will better capture RAM usage and cycles per byte in highly constrained environments. Future studies should also evaluate dedicated hardware accelerators and next-generation automotive platforms to realistically assess MAC performance under strict resource limitations.

System-Level and Protocol Integration

Research should move beyond isolated MAC evaluation to assess full protocol stacks (e.g., IPsec, TLS) in automotive and IoT contexts. Metrics should include protocol overhead, I/O latency, memory footprint, bandwidth, and packet throughput, clarifying the real-world impact of MAC selection on system performance and enabling informed architectural decisions.

Advanced Security Evaluation

Further studies should investigate side-channel and fault-injection resistance for lightweight MACs. Techniques such as Deep Learning Side-Channel Analysis (DLSCA), masking methods like Threshold Implementation (TI) and Domain-Oriented Masking (DOM), as well as automated cryptanalysis of ARX-based MACs (e.g., SipHash) and other primitives, should be extended to a wider set of algorithms.

Algorithmic Scope and Energy Efficiency

Future research should include authenticated encryption (AEAD) schemes and lightweight post-quantum alternatives, while rigorously evaluating energy consumption and battery impact, correlating these metrics to cycles per byte. Results should also consider industry standards and regulations, including ISO/SAE 21434 and NIST recommendations, ensuring security and performance align with automotive compliance requirements.

References

[1] National Institute of Standards and Technology, "Submission requirements and evaluation criteria for the lightweight cryptography standardization process", National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. NIST LWC Submission Requirements, 2018, pp. 1–17. [Online]. Available: https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/final-lwc-submission-requirements-august2018.pdf.

[2] S. Shin, M. Kim, and T. Kwon, "Experimental performance analysis of lightweight block ciphers and message authentication codes for wireless sensor networks", *International Journal of Distributed Sensor Networks*, vol. 13, no. 11, pp. 1–12, 2017. DOI: 10.1177/1550147717744169.

[3] L. C. dos Santos, J. Großschädl, and A. Biryukov, "FELICS-AEAD: Benchmarking of lightweight authenticated encryption algorithms", in *International Conference on Smart Card Research and Advanced Applications (CARDIS 2019)*, ser. Lecture Notes in Computer Science, vol. 11833, Prague, Czech Republic: Springer, 2019, pp. 216–233. DOI: 10.1007/978-3-030-42068-0_13.

[4] M. Jimale, M. R. Zaba, M. L. Kiah, M. Y. I. Idris, N. Jamil, M. S. Mohamad, and M. S. Rohmad, "AEAD schemes: A systematic review", *IEEE Access*, vol. 10, pp. 14 739–14 766, 2022. DOI: 10.1109/ACCESS.2022.3147201.

[5] F. Capra, *MAC algorithms benchmarking*, 2025. [Online]. Available: https://github.com/FrancescaCapra/mac%5C_algorithms.

[6] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication", in *Advances in Cryptology – CRYPTO '96*, Berlin, Heidelberg, Germany: Springer, 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1.

[7] M. Dworkin, "Recommendation for block cipher modes of operation: The CMAC mode for authentication", National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. SP 800-38B, 2005, pp. 1–21. DOI: 10.6028/NIST.SP.800-38B.

[8] T. Iwata, J. Song, J. Lee, and R. Poovendran, *The AES-CMAC algorithm*, RFC 4493, Fremont, CA, USA, 2006. DOI: 10.17487/RFC4493.

[9] T. Iwata and K. Kurosawa, "OMAC: One-key CBC MAC", in *Fast Software Encryption (FSE)*, Berlin, Heidelberg, Germany: Springer, 2003, pp. 129–153. DOI: 10.1007/978-3-540-39887-5_11.

[10] D. J. Bernstein, "The poly1305-aes message-authentication code", in *Fast Software Encryption (FSE)*, Berlin, Heidelberg, Germany: Springer, 2005, pp. 32–49. DOI: 10.1007/11502760_3.

[11] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols", Internet Engineering Task Force (IETF), Fremont, CA, USA, Tech. Rep. RFC 8439, 2018, pp. 1–46. DOI: 10.17487/RFC8439.

[12] J.-P. Aumasson and D. J. Bernstein, "SipHash: A fast short-input prf", in *Progress in Cryptology – INDOCRYPT 2012*, S. Galbraith and M. Nandi, Eds., Berlin, Heidelberg: Springer, 2012, pp. 489–508. DOI: 10.1007/978-3-642-34931-7_28.

[13] C. Dobraunig, F. Mendel, and M. Schläffer, "Differential cryptanalysis of SipHash", in *Selected Areas in Cryptography – SAC 2014*, Montreal, QC, Canada: Springer, 2014, pp. 165–182. DOI: 10.1007/978-3-319-13051-4_10.

[14] L. He and H. Yu, "Cryptanalysis of reduced-round SipHash", *The Computer Journal*, vol. 67, no. 3, pp. 875–883, 2024. DOI: 10.1093/comjnl/bxad026.

[15] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: An efficient MAC algorithm for 32-bit microcontrollers", in *Selected Areas in Cryptography – SAC 2014*, Cham, Switzerland: Springer, 2014, pp. 306–323. DOI: 10.1007/978-3-319-13051-4_19.

[16] G. Leurent, "Improved differential-linear cryptanalysis of 7-round Chaskey with partitioning", Berlin, Heidelberg, Germany, Tech. Rep., 2016, pp. 344–371. DOI: 10.1007/978-3-662-49890-3_14.

[17] N. Mouha, "Chaskey: A MAC algorithm for microcontrollers – status update and proposal of Chaskey-12", IACR Cryptology ePrint Archive, Leuven, Belgium, Tech. Rep. 2015/1182, 2015, pp. 1–16. [Online]. Available: https://eprint.iacr.org/2015/1182.

[18] M. S. Turan, K. A. McKay, D. Chang, J. Kang, and J. Kelsey, "Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption (AEAD), hash, and extendable output functions", National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. SP 800-232, 2025, pp. 1–50. DOI: 10.6028/NIST.SP.800-232.

[19] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon MAC, PRF, and short-input PRF", in *Topics in Cryptology – CT-RSA 2024*, San Francisco, CA, USA: Springer, 2024, pp. 381–403. DOI: 10.1007/978-3-031-58868-6_15.

[20] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Cryptanalysis of ascon", in *Topics in Cryptology – CT-RSA 2015*, ser. Lecture Notes in

Computer Science, vol. 9048, Berlin, Heidelberg: Springer, 2015, pp. 371–387. DOI: 10.1007/978-3-319-16715-2_20.

[21] Y. Li, G. Zhang, W. Wang, and M. Wang, "Cryptanalysis of round-reduced ASCON", *Science China Information Sciences*, vol. 60, no. 3, pp. 1–11, 2017. DOI: 10.1007/s11432-016-0283-3.

[22] J. Kelsey, S. Chang, and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash", National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. SP 800-185, 2016, pp. 1–37. DOI: 10.6028/NIST.SP.800-185.

[23] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: Simpler, smaller, fast as MD5", in *Applied Cryptography and Network Security (ACNS 2013)*, Berlin, Heidelberg, Germany: Springer, 2013, pp. 119–135. DOI: 10.1007/978-3-642-38980-1_8.

[24] M.-J. O. Saarinen and J.-P. Aumasson, *The BLAKE2 cryptographic hash and message authentication code (MAC)*, RFC 7693, Fremont, CA, USA, 2015. DOI: 10.17487/RFC7693.

[25] J. Guo, P. Karpman, I. Nikolić, L. Wang, and S. Wu, "Analysis of BLAKE2", in *Topics in Cryptology – CT-RSA 2014*, Cham, Switzerland: Springer, 2014, pp. 402–423. DOI: 10.1007/978-3-319-04852-9_21.

[26] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, *BLAKE3: One function, fast everywhere*, 2020. [Online]. Available: https://github.com/BLAKE3-team/BLAKE3-specs.

[27] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", in *Advances in Cryptology – CRYPTO '99*, Santa Barbara, CA, USA: Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[28] K. Ebina, R. Ueno, and N. Homma, "Side-channel analysis against SecOC-compliant AES-CMAC", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 10, pp. 3772–3776, 2023. DOI: 10.1109/TCSII.2023.3288278.

[29] A. Baksi, S. Bhasin, J. Breier, D. Jap, and D. Saha, "A survey on fault attacks on symmetric key cryptosystems", *ACM Computing Surveys*, vol. 55, no. 4, pp. 1–34, 2022. DOI: 10.1145/3530054.

[30] H. Madushan, I. Salam, and J. Alawatugoda, "A review of the NIST lightweight cryptography finalists and their fault analyses", *Electronics*, vol. 11, no. 24, pp. 1–19, 2022. DOI: 10.3390/electronics11244199.

[31] Amrita, C. P. Ekwueme, I. H. Adam, and A. Dwivedi, "Lightweight cryptography for internet of things: A review", *EAI Endorsed Transactions on Internet of Things*, vol. 10, e5, 2024. DOI: 10.4108/eetiot.5565.

[32] L. Weissbart and S. Picek, "Lightweight but not easy: Side-channel analysis of the Ascon AEAD cipher on a 32-bit microcontroller", IACR Cryptology ePrint Archive, Delft, Netherlands, Tech. Rep. 2023/1598, 2023, pp. 1–24. [Online]. Available: https://eprint.iacr.org/2023/1598.

[33] H. Gross, E. Wenger, C. Dobraunig, and C. Ehrenhöfer, "Ascon hardware implementations and side-channel evaluation", *Microprocessors and Microsystems*, vol. 52, pp. 470–479, 2017. DOI: 10.1016/j.micpro.2016.10.006.

[34] A. Kandi, A. Baksi, P. Gan, S. Guilley, T. Gerlich, J. Breier, A. Chattopadhyay, R. R. Shrivastwa, Z. Martinásek, and S. Bhasin, "Side-channel and fault resistant Ascon implementation: A detailed hardware evaluation", in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Knoxville, TN, USA: IEEE, 2024, pp. 307–312. DOI: 10.1109/ISVLSI61997.2024.00063.

[35] H. Bühler, A. Walz, and A. Sikora, "Benchmarking of symmetric cryptographic algorithms on a deeply embedded system", *IFAC-PapersOnLine*, vol. 55, no. 4, pp. 266–271, 2022. DOI: 10.1016/j.ifacol.2022.06.044.

[36] S. Blanc, A. Lahmadi, K. Le Gouguec, M. Minier, and L. Sleem, "Benchmarking of lightweight cryptographic algorithms for wireless IoT networks", *Wireless Networks*, vol. 28, no. 8, pp. 3453–3476, 2022. DOI: 10.1007/s11276-022-03046-1.

[37] I. Radhakrishnan, S. Jadon, and P. B. Honnavalli, "Efficiency and security evaluation of lightweight cryptographic algorithms for resource-constrained IoT devices", *Sensors*, vol. 24, no. 12, pp. 1–16, 2024. DOI: 10.3390/s24124008.

[38] J. Deepakumara, H. M. Heys, and R. Venkatesan, "Performance comparison of message authentication code (MAC) algorithms for the internet protocol security (IPsec)", in *Proceedings of the 2003 Newfoundland Electrical and Computer Engineering Conference*, St. John's, NL, Canada: Memorial University of Newfoundland, 2003, pp. 1–7.

[39] L. Pallavi, P. Singh, B. Patnaik, and B. Acharya, "High frequency architecture of lightweight authenticated cipher ASCON-128 for resource-constrained IoT devices", in *2022 IEEE International Conference on Computing, Communication and Automation (ICCCA)*, Greater Noida, India: IEEE, 2022, pp. 408–411. DOI: 10.1109/ICCCA56541.2022.10035751.

[40] H. Gross, E. Wenger, C. Dobraunig, and C. Ehrenhöfer, "Suit up! – made-to-measure hardware implementations of Ascon", in *2015 Euromicro Conference on Digital System Design*, Funchal, Madeira, Portugal: IEEE, 2015, pp. 645–652. DOI: 10.1109/DSD.2015.118.

[41] H. L. Pham, T. H. Tran, V. T. Duong Le, and Y. Nakashima, "Flexible and scalable BLAKE/BLAKE2 coprocessor for blockchain-based IoT applications",

IEEE Design & Test, vol. 40, no. 5, pp. 15–25, 2023. DOI: 10.1109/MDAT.2023.3276936.

[42] J. Sugier, "Improving FPGA implementations of BLAKE", in *Advances in Dependability Engineering of Complex Systems*, W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Eds., Berlin, Germany: Springer, 2018, pp. 395–406. DOI: 10.1007/978-3-319-59415-6_38.

[43] A. J. Bhuvaneshwari, P. Kaythry, K. J. Jegadish Kumar, and D. Sachin, "7-stage pipelined architecture of ascon for resource-constrained devices", *IEEE Embedded Systems Letters*, vol. 17, no. 4, pp. 252–255, 2025. DOI: 10.1109/LES.2025.3541818.

[44] N. I. of Standards and Technology, "FIPS PUB 197: Advanced encryption standard (AES)", National Institute of Standards and Technology, Washington, DC, USA, Tech. Rep. FIPS 197-upd1, 2001, pp. 1–51. DOI: 10.6028/NIST.FIPS.197-upd1.

[45] J.-P. Aumasson, *Blake2: Fast secure hashing*, 2017. [Online]. Available: https://blake2.net/.

[46] J.-P. Aumasson, S. Neves, J. O'Connor, and Z. Wilcox, *The blake3 hashing framework*, 2024. [Online]. Available: https://www.ietf.org/archive/id/draft-aumasson-blake3-00.html.

[47] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The Keccak reference", National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, Tech. Rep. Round 3, 2011, pp. 1–69. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf.

[48] T. C. Committee, *CAESAR: Competition for authenticated encryption: Security, applicability, and robustness*, 2019. [Online]. Available: https://competitions.cr.yp.to/caesar-submissions.html.

[49] F. De Santis, A. Schauer, and G. Sigl, "ChaCha20-Poly1305 authenticated encryption (AEAD) for high-speed embedded IoT applications", in *2017 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Lausanne, Switzerland: IEEE, 2017, pp. 692–697. DOI: 10.23919/DATE.2017.7927078.

Appendix A Code Size Analysis

To calculate the code size per algorithm, I summed the sizes of the relevant functions from `nm -size-sort`, grouping them by algorithm. Some functions are shared (e.g., `print_hex`, `init_benchmark`), but I excluded them from counting by algorithm unless they are specific. I used only those functions directly associated with the MAC calculation or algorithm benchmark.

1. SipHash

- **Function:**
  - `mac_siphash`: 61 bytes
  - `siphash`: 1,881 bytes
  - `bench_siphash`: 506 bytes
- **Total Code Size**: $61 + 1,881 + 506 = 2,448$ bytes

2. Chaskey

- **Function:**
  - `mac_chaskey`: 147 bytes
  - `subkeys`: 359 bytes
  - `chaskey`: 4,112 bytes

  - `test_vectors`: 328 bytes
  - `bench_chaskey`: 587 bytes

- **Total Code Size**: $147 + 359 + 4,112 + 328 + 587 = 5,533$ bytes

3. ASCON

- **Function:**
  - `mac_ascon`: 55 bytes
  - `crypto_prf`: 647 bytes
  - `crypto_auth`: 62 bytes
  - `crypto_auth_verify`: 167 bytes
  - `ROUND`: 689 bytes
  - `P12`: 219 bytes
  - `LOADBYTES`: 83 bytes
  - `STOREBYTES`: 75 bytes
  - `ROR`: 28 bytes
  - `bench_ascon`: 494 bytes
- **Total Code Size**: $55 + 647 + 62 + 167 + 689 + 219 + 83 + 75 + 28 + 494 = 2,519$ bytes

4. ASCON PRFShort

- **Function:**
  - `mac_ascon_prfshort`: 55 bytes
  - `crypto_prfs`: 437 bytes

  - `crypto_auth_prfs`: 62 bytes
  - `crypto_auth_verify_prfs`: 181 bytes
  - `ROUND`: 689 bytes
  - `P12`: 219 bytes
  - `LOADBYTES`: 83 bytes
  - `STOREBYTES`: 75 bytes
  - `ROR`: 28 bytes
  - `bench_ascon_prfshort`: 506 bytes
- **Total Code Size**: $55 + 437 + 62 + 181 + 689 + 219 + 83 + 75 + 28 + 506 = 2,335$ bytes

5. Poly1305

- **Function:**
  - `mac_poly1305`: 54 bytes
  - `bench_poly1305`: 552 bytes
- **Total Code Size**: $54 + 552 = 606$ bytes

6. KMAC256

- **Function:**
  - `kmac256_init`: 164 bytes
  - `kmac256_mac`: 387 bytes
  - `kmac256_cleanup`: 61 bytes
  - `bench_kmac256`: 791 bytes
- **Total Code Size**: $164 + 387 + 61 + 791 = 1,403$ bytes

7. AES-CMAC

- **Function:**
  - `mac_aes_cmac`: 275 bytes
  - `bench_aes_cmac`: 568 bytes
- **Total Code Size**: $275 + 568 = 843$ bytes

8. BLAKE2

- **Function:**
  - `mac_blake2`: 77 bytes
  - `blake2s`: 386 bytes
  - `blake2b`: 386 bytes
  - `blake2s_init`: 242 bytes
  - `blake2b_init`: 268 bytes
  - `blake2s_init_key`: 413 bytes
  - `blake2b_init_key`: 511 bytes
  - `blake2s_init_param`: 140 bytes
  - `blake2b_init_param`: 147 bytes
  - `blake2s_update`: 309 bytes
  - `blake2b_update`: 327 bytes
  - `blake2s_final`: 364 bytes
  - `blake2b_final`: 416 bytes
  - `blake2s_compress`: 14333 bytes
  - `blake2b_compress`: 21896 bytes

  - `blake2s_increment_counter`: 69 bytes
  - `blake2b_increment_counter`: 79 bytes
  - `blake2s_set_lastnode`: 26 bytes
  - `blake2b_set_lastnode`: 27 bytes
  - `blake2s_is_lastblock`: 29 bytes
  - `blake2b_is_lastblock`: 31 bytes
  - `blake2s_set_lastblock`: 57 bytes
  - `blake2b_set_lastblock`: 58 bytes
  - `blake2s_init0`: 97 bytes
  - `blake2b_init0`: 99 bytes
  - `secure_zero_memory`: 44 bytes
  - `rotr32`: 24 bytes
  - `rotr64`: 28 bytes
  - `load32`: 85 bytes
  - `load64`: 175 bytes
  - `store16`: 66 bytes
  - `store32`: 87 bytes
  - `store64`: 182 bytes
  - `blake2`: 76 bytes
  - `bench_blake2`: 563 bytes

- **Total Code Size**: $77 + 386 + 386 + 242 + 268 + 413 + 511 + 140 + 147 + 309 + 327 + 364 + 416 + 14333 + 21896 + 69 + 79 + 26 + 27 + 29 + 31 + 57 + 58 + 97 + 99 + 44 + 24 + 28 + 85 + 175 + 66 + 87 + 182 + 76 + 563 = 41,957$ bytes

9. BLAKE3

- **Function:**
  - `mac_blake3`: 221 bytes
  - `blake3_hasher_init`: 195 bytes
  - `blake3_hasher_init_keyed`: 1220 bytes
  - `blake3_hasher_init_derive_key_raw`: 1621 bytes
  - `blake3_hasher_init_derive_key`: 57 bytes
  - `blake3_hasher_update`: 18241 bytes
  - `blake3_hasher_finalize`: 52 bytes
  - `blake3_hasher_finalize_seek`: 5289 bytes
  - `blake3_hasher_reset`: 137 bytes
  - `blake3_compress_in_place`: 164 bytes
  - `blake3_compress_xof`: 189 bytes
  - `blake3_xof_many`: 147 bytes
  - `blake3_hash_many`: 369 bytes
  - `blake3_simd_degree`: 78 bytes
  - `blake3_version`: 17 bytes
  - `blake3_compress_subtree_wide`: 4097 bytes
  - `blake3_compress_in_place_sse2`: 39380 bytes
  - `blake3_compress_xof_sse2`: 39767 bytes
  - `blake3_compress_in_place_sse41`: 40868 bytes
  - `blake3_compress_xof_sse41`: 41255 bytes
  - `blake3_compress_in_place_portable`: 56944 bytes

  - `blake3_compress_xof_portable`: 59035 bytes
  - `blake3_hash4_sse2`: 112220 bytes
  - `blake3_hash4_sse41`: 154188 bytes
  - `blake3_hash8_avx2`: 211486 bytes
  - `blake3_hash_many_sse2`: 510 bytes
  - `blake3_hash_many_sse41`: 510 bytes
  - `blake3_hash_many_avx2`: 223 bytes
  - `blake3_hash_many_portable`: 1504 bytes
  - `highest_one`: 26 bytes (x6, but counted once: 26 bytes)
  - `get_cpu_features`: 424 bytes
  - `xgetbv`: 56 bytes
  - `cpuid`: 71 bytes
  - `cpuidex`: 79 bytes
  - `bench_blake3`: 575 bytes

- **Total Code Size**: $221+195+1220+1621+57+18241+52+5289+137+164+189+147+369+78+17+4097+39380+39767+40868+41255+56944+59035+112220+154188+211486+510+510+223+1504+26+424+56+71+79+575 = 790,375$ bytes

Binary Dimension

- **Total Binary**: `mac_algorithms` are 1,715,000 bytes (1.72 MB), which include:
  - `.text`: 850,798 bytes (code)
  - **Other sections**: `.data`, `.bss`, debug symbols, relocation table, etc.

- The `.text` section represents about 50% of the binary, which is typical for a program with debug info.