RESEARCH-ARTICLE

Designing Size-aware Cryptographic Primitives for FPGA-based Accelerators

ADITYA PEER

ROHAN SUDHIR BASUGADE

SIDDHANT

NEERAJ KUMAR YADAV

AGAMDEEP SINGH

PRAVEEN TAMMANA

View all

Proceedings of the ACM on Networking, Volume 4, Issue CoNEXT1 (March 2026)
https://doi.org/10.1145/3786286
EISSN: 2834-5509

Designing Size-aware Cryptographic Primitives for FPGA-based Accelerators

ADITYA PEER, Indraprastha Institute of Information Technology Delhi, India
ROHAN SUDHIR BASUGADE, Indraprastha Institute of Information Technology Delhi, India
SIDDHANT, Indraprastha Institute of Information Technology Delhi, India
NEERAJ KUMAR YADAV, Indraprastha Institute of Information Technology Delhi, India
AGAMDEEP SINGH, Indraprastha Institute of Information Technology Delhi, India
PRAVEEN TAMMANA, Indian Institute of Technology Hyderabad, India
SUMIT DARAK, Indraprastha Institute of Information Technology Delhi, India
RINKU SHAH, Indraprastha Institute of Information Technology Delhi, India

Telecom operators and military/defense applications (e.g., UAVs) prefer to offload compute-intensive cryptographic processing to FPGA-based cryptographic accelerators over fixed-function ASICs for reconfigurability and short deployment cycles. The state-of-the-art FPGA-based cryptographic accelerators are designed either for high throughput or power efficiency. We propose *FlexMesh*, a programmable, dynamically reconfigurable, and size-aware framework for 5G/6G cryptographic primitives. We design variable-sized FPGA-based cryptographic cores by varying the degree of parallelism, and demonstrate that if the cryptographic core is chosen wisely, we can achieve performance without trading resource or power efficiency for certain use cases. *FlexMesh* dynamically orchestrates multiple asymmetric cryptographic cores (i.e., variable-sized cores) based on the observed request size distribution. To efficiently utilize the deployed cryptographic cores, we design a workload-aware, dynamically configurable load balancer and a scheduler. We test our system over realistic 5G datasets, and our results demonstrate that asymmetric cryptographic cores reduce the request processing latency and improve resource and power efficiencies, compared to the baseline symmetric cryptographic cores.

CCS Concepts: • **Networks** → **Programmable networks**; **In-network processing**; • **Security and privacy**;

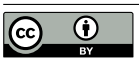
Additional Key Words and Phrases: in-network computing, cryptographic acceleration, in-network scheduler, FPGA-based accelerators, 5G cryptography

ACM Reference Format:
Aditya Peer, Rohan Sudhir Basugade, Siddhant, Neeraj Kumar Yadav, Agamdeep Singh, Praveen Tammana, Sumit Darak, and Rinku Shah. 2026. Designing Size-aware Cryptographic Primitives for FPGA-based Accelerators. *Proc. ACM Netw.* 4, CoNEXT1, Article 3 (March 2026), 25 pages. https://doi.org/10.1145/3786286

1 Introduction

Modern telecommunication networks have adopted the software-defined networking paradigm, and the 5G/6G network functions run on virtualized VMs or Docker containers in data centers [8]. Telecom network services secure communication between the mobile users and base stations

Authors' Contact Information: Aditya Peer, adityap@iiitd.ac.in, Indraprastha Institute of Information Technology Delhi, India; Rohan Sudhir Basugade, Indraprastha Institute of Information Technology Delhi, India, rohan22416@iiitd.ac.in; Siddhant, Indraprastha Institute of Information Technology Delhi, India, siddhant21204@iiitd.ac.in; Neeraj Kumar Yadav, Indraprastha Institute of Information Technology Delhi, India, neerajy@iiitd.ac.in; Agamdeep Singh, Indraprastha Institute of Information Technology Delhi, India, agamdeep21306@iiitd.ac.in; Praveen Tammana, Indian Institute of Technology Hyderabad, India, praveent@cse.iith.ac.in; Sumit Darak, Indraprastha Institute of Information Technology Delhi, India, sumit@iiitd.ac.in; Rinku Shah, Indraprastha Institute of Information Technology Delhi, India, rinku@iiitd.ac.in.



ACM 2834-5509/2026/3-ART3
https://doi.org/10.1145/3786286

Use case	User experienced data rate		User density
	Downlink	Uplink	
Urban WAN	50Mbps	25Mbps	10,000/km^2
Indoor hotspot	1Gbps	500Mbps	250,000/km^2
Broadband access in a crowd	25Mbps	50Mbps	500,000/km^2
Dense urban	300Mbps	50Mbps	250,000/km^2
Federated Learning for image/video processing	80.88 Mbps	80.88 Mbps	(To be decided)

Table 1. 5G/6G use cases with diverse requirements [9].

Algorithm	Year	Standard
AES	~2001	3G-UMTS
EEA/EIA (Snow-3G, ZUC)	~2009	4G-LTE
NEA/NIA (Snow-3G, ZUC-128)	~2018	5G-NR
ZUC-256, AES-256	~2028	6G
Post-Quantum Crypto	~2030	6G (for AI)

Table 2. Cryptographic standards evolve every decade.

(gNB/eNB) using cryptographic algorithms such as Snow-V [25], Rocca-S [45], and ZUC [72], and terminate/initiate IPsec tunnel (AES-GCM, AES-CTR) at the core network. A recent study [49] indicates that even when using CPU acceleration instructions like AES-NI [28], an application consumes 74% of the total CPU cycles for encryption and 60% for decryption. To improve performance and reduce CPU usage, operators often offload cryptographic functions to specialized hardware accelerators. The CPU is then responsible for handling the less-frequent handshake operations of the cryptographic algorithm, while the accelerators, designed with features for virtualization, multitenancy, and isolation, manage the more frequent, CPU-intensive data path operations.

State-of-the-art cryptographic accelerators. SmartNICs such as Nvidia's Bluefield [1]) and AMD Pensando's DSC [48] offer off-data-path, fixed-function accelerators (*i.e.*, ASICs) for popular security algorithms. However, due to high fabrication costs, ASICs are typically developed only for cryptographic algorithms that have demonstrated long-term stability. Few studies design reconfigurable, on-data-path cryptographic accelerators for smartNICs and programmable switches. For example, (a) Netronome smartNIC with ChaCha algorithm implementation [80], (b) Intel Tofino switches with implementations of HalfSipHash [55, 74], ChaCha [75], and partial AES [21], and (c) FPGA hardware implementations for 5G/6G ciphers such as Rocca-S [13], Snow-V [15], ZUC [35], and AES-GCM [66, 67].

FPGA-based accelerators enable faster adaptation to changing requirements. Table 2 demonstrates that cryptographic algorithms standardized by telecommunication frequently change, typically every decade. This rapid evolution makes it impractical to develop an ASIC for the ciphers. Additionally, Table 1 highlights the diverse data rate and scale requirements for various 5G applications. To meet these requirements, telecom operators can leverage FPGA-based accelerators, which offer programmability and reconfigurability in real-time, while maintaining efficiency. Similarly, military and defense applications often necessitate in-field reprogramming to adapt to evolving threat models. For example, an encrypted link for an unmanned aerial vehicle (UAV) may require a new cipher; an FPGA-based module can load a new bitstream in the field, while an ASIC cannot. Furthermore, if a UAV is captured, the device can be wiped clean remotely to complicate efforts to reverse engineer the cipher.

Energy efficiency is a crucial challenge in next-generation networks. As communication networks have advanced, energy consumption has risen, particularly in 5G, where technologies like massive MIMO and network densification contribute to a higher energy footprint. Further, emerging AI-driven NextG networks risk increasing power demands, especially due to expanded edge computing. Improving energy efficiency is crucial for reducing operational costs and expanding broadband access [46].

Real-world applications require both high performance and resource efficiency. URLLC-based applications such as Connected Autonomous Vehicles (CAVs) offload real-time sensor and command data to the network edge (*i.e.*, gNB or edge server) with limited compute capabilities, expecting latency in the order of milliseconds. Also, a 5G/6G backhaul comprising LEO satellites has

payloads that are constrained in power and compute, but expect high-bandwidth ($\sim 10 - 20Gbps$) encrypted communication between satellites and the terrestrial gNBs [9].

Previous research on designing FPGA hardware for 5G and 6G cryptographic algorithms has primarily focused on two approaches: (a) high-throughput solutions that replicate multiple highly optimized cryptographic cores to process multiple requests in parallel. While these solutions achieve high throughput, they consume significant FPGA area and power resources [10, 18, 38, 43]. Alternatively, (b) resource-efficient solutions have been developed that reuse components to optimize resource and power utilization. However, this approach often leads to increased latency due to the sequential nature of processing [37, 38, 50].

FlexMesh. In this paper, we propose *FlexMesh*, a request size-aware, dynamically reconfigurable, and scalable cryptographic framework, leveraging FPGAs. Our key idea is to utilize the request length distribution statistics and design cryptographic core variants that aim to overcome the tradeoff between performance and resource utilization.

Research gaps. To realise *FlexMesh*, we have two main research gaps. (1) The state-of-the-art FPGA-based cryptographic systems trade off between performance and resource utilization (*i.e.*, hardware gate area and power). To mitigate this gap, we design cryptographic cores of variable sizes, where the performance and resource utilization are proportional to the size of the cryptographic core (early ideas proposed in our poster [32]). Our system monitors the workload characteristics in real-time to determine the appropriate cryptographic core variant that optimizes both performance and resource efficiency (§4). To elaborate, the cryptographic core's performance depends on the request size to be (de)encrypted, and for certain request lengths, small-sized cryptographic cores yield performance comparable to large-sized cores. If the workload comprises request lengths within such a range, choosing a small-sized cryptographic core saves resources without compromising performance. (2) Replicating variable-sized cryptographic cores for scalability, results in inefficiencies such as load imbalance and additional memory overhead to store out-of-order request delivery. We design a workload-aware load balancer and a work-conserving scheduler to mitigate this gap. These components are dynamically configured to minimize the memory required to store out-of-order requests (§4.1).

The key contributions of this paper are:

- We motivate the need for cryptographic core variants by analyzing the latency, resource efficiency, and power efficiency across real-world 5G workloads (§2).
- We design variable-sized cryptographic cores for 5G cryptographic algorithms, Rocca-S, and AES-GCM that observe the request length distribution to facilitate opportunistic resource and power savings without compromising performance (§4.3).
- We design a custom load balancer and a scheduler that can be dynamically configured with workload-specific thresholds to achieve optimal efficiency, and reduce memory to store out-of-order requests (§4.2).
- We develop a prototype of Rocca-S and AES-GCM cryptographic cores (viz, S, M, and L) for the FPGA target, Xilinx ZCU106 [2]. Additionally, we develop a discrete-event simulator (in C++) for the custom load balancer and scheduler (§4.2) [1]. We evaluate our prototype for real-world 5G applications such as URLLC, MMTC, IoT, and video-streaming [5, 41, 44]. Our evaluation demonstrates that *FlexMesh* helps (a) reduce the request median latency by up to 31.58% respectively, and (b) improve resource and power efficiency by up to 17.94% and 45.24%, respectively, for MMTC workload [5], compared to the baseline symmetric cryptographic cores.(§6).

[1]To support reproducibility, our open-source code is available at: https://github.com/pnl-iiitd/flexmesh.

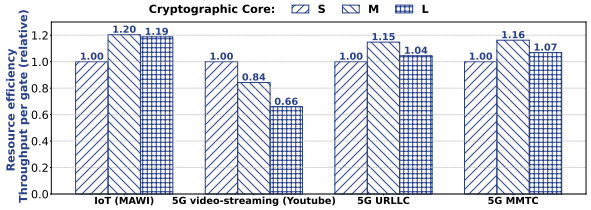


Fig. 1. Rocca-S's resource efficiency across cryptographic core variants for 5G applications relative to the cryptographic core, *S*. The *S* core outperforms for video-streaming traffic, and the *M* core performs better across the rest.

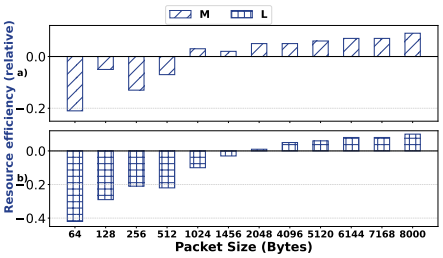
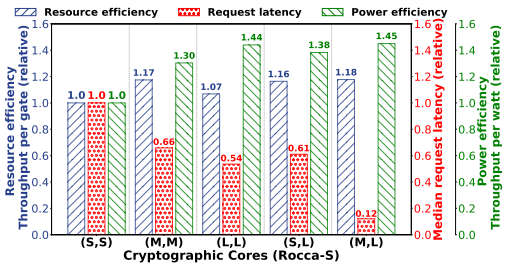
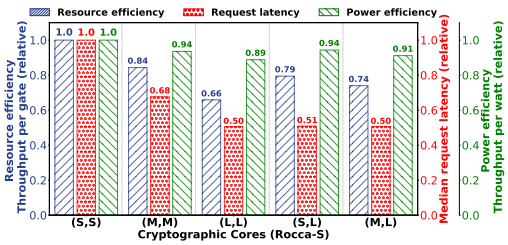


Fig. 2. AES-GCM's efficiency relative to the cryptographic core, *S*. The threshold choice for the size-aware load balancer is 512 bytes and 2048 bytes for *(S, M)* and *(S, L)* asymmetric cryptographic cores, respectively.



(a) MMTC Dataset.



(b) 5G video streaming (YouTube) Dataset.

Fig. 3. Symmetric vs. asymmetric cryptographic cores: For MMTC, *(M, L)* core is the most efficient and has comparable latency with the fastest core. For YouTube workload, choose *(S, S)* for resource/power efficiency or *(S, L)* for min latency.

2 Design requirements and solution approaches

This section states the design requirements (R) and describes the solution approaches (S) employed by *FlexMesh*. The details about the datasets used in the motivation plots are available in Table 8.

R1 *To avoid the rigidity and inefficiency of fixed-function cryptographic cores, resource provisioning must support dynamic and adaptable operation.*

S1 *Design variable-sized FPGA-based cryptographic cores.* Static resource provisioning techniques designed for peak performance or peak efficiency are sub-optimal and inflexible. We designed three cryptographic core variants for both Rocca-S and AES-GCM ciphers, viz., small (S), medium (M), and large (L), that vary in terms of performance and resource utilization. For example, an *L* core is designed with the maximum permissible parallelism, resulting in maximum FPGA resource utilization (see Table 3). Fig. 1 shows Rocca-S's resource efficiency for real-world workloads (viz., IoT [44], 5G video-streaming [41], 5G URRLC [5], and 5G MMTC [5]) across cryptographic core variants, relative to the *S* core. Here, resource efficiency measures the throughput per resource unit (*i.e.*, per logic gate in the case of FPGAs). We observe that the *S* core outperforms in the case of video-streaming by 34%, and the *M* core outperforms by up to 16% for other workloads, motivating the need for flexibility to configure hardware resources to come up with a core that suits best for a specific workload. We observe similar trends across AES-GCM cryptographic core variants as well.

R2 *To ensure effective scaling, systems must be designed to optimize resource efficiency, power efficiency, and latency concurrently.*

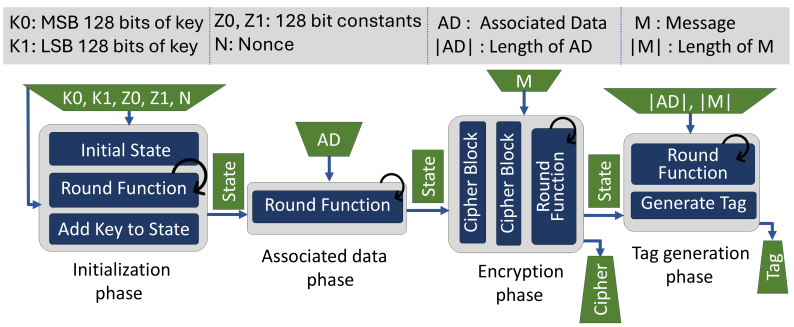


Fig. 4. Overview of Rocca-S algorithm.

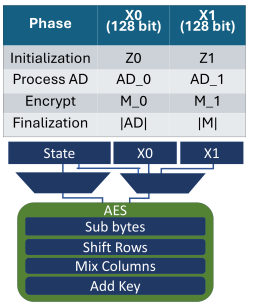


Fig. 5. AES round function.

S2 *Orchestrate asymmetric cryptographic cores (i.e., variable-sized cores) based on the workload distribution.* Figures 3a and 3b shows the resource efficiency, power efficiency, and median latency of Rocca-S cryptographic algorithm for 5G-MMTC [5] and 5G video streaming [41] workloads, relative to the baseline, *(S, S)*. We observe that for MMTC workload, *(M, L)* cryptographic core is up to 18% resource-efficient, 45% power efficient, and reduces the median latency by 88%, whereas, for the YouTube dataset, *(S, S)* is the most resource and power-efficient, but *(S, L)* reduces the median latency by 50% with the second highest power efficiency, and the third highest resource efficiency. We argue that the choice of cryptographic core variants depends on the use case requirements and workload distribution, and this distribution can change over time. We leverage FPGA-based accelerators that support dynamic orchestration and reconfiguration of the offload hardware.

R3 *Need for load balancer and scheduler design to efficiently distribute requests across asymmetric cryptographic cores.*

S3 *Design of a size-aware load balancer.* State-of-the-art cryptographic offload solutions employ simple scheduling policies such as round-robin (RR) to distribute the incoming requests across replicas of the same cryptographic core (*i.e.*, symmetric) for scale. *FlexMesh* replicates cryptographic cores having varying capabilities, raising the need for a heterogeneity-aware load balancer and a scheduler. Fig. 2 shows the resource efficiency of AES-GCM cryptographic cores, *M* and *L*, relative to the *S* core. We observe that the *M* core and the *L* core outperforms after request size of 512 bytes, and 2048 bytes, respectively.

R4 *The system must optimize additional hardware resources requirements to store requests processed out-of-order for cost efficiency.*

S4 *Custom scheduler with request-size prioritization.* To optimize the performance, SOTA schedulers prioritize requests based on an objective function, resulting in out-of-order processing.The out-of-order processed requests within a flow are held in buffers until the missing requests are processed, requiring additional memory hardware. We redesign and configure the existing in-network, scalable PIEO [58] scheduler to employ request prioritization and work-conservation (see §4.2). Results show that *FlexMesh*'s scheduler can significantly reduce the out-of-order memory (see Fig. 14).

3 Background

In this section, we provide an overview of an emerging 6G, viz., Rocca-S cryptographic algorithm. The 5G/6G AES-GCM cryptographic algorithm overview is presented in §A.

Overview of Rocca-S Algorithm

Rocca-S [13, 45] is a high-speed, AEAD (authenticated encryption with associated data) algorithm designed to meet the stringent performance and security demands of emerging 6G networks on the wireless link, *i.e.*, between the mobile user (UE) and the base station (gNB).

Rocca-S phases. It is composed of four phases: (1) initialization, (2) processing associated data, (3) encryption, and (4) finalization (*i.e.*, tag generation) (see Fig. 4).

Rocca-S inputs and output. The input consists of a 256-bit key $K = K0||K1$, a nonce `N` of between 12 and 16 octets in length, the associated data `AD`, and the message `M`. The output is the corresponding ciphertext `C` and a 256-bit tag `T`. The algorithm also comprises a state array `S` of *7 blocks* where each block is 128 bits, and two constants `Z0` and `Z1` of 128 bits each. This state is updated throughout the algorithm and is used in message encryption and tag generation.

The AES round function. Fig. 5 shows the AES round function, `R(S,X0,X1)`, which takes as input the state S and two blocks `(X0,X1)`, each 128 bits. The values taken by `(X0,X1)` depends on the Rocca-S phase of operation, for example, in initialization phase `(X0,X1)` are initialized with `(Z0,Z1)`. The output of the round function is the update to the state array, `Snew = R(S,X0,X1)`. The detailed algorithm of the round function is shown in Algorithm 1.

The encryption phase. In this phase, the message `M` is first padded to `PAD(M)` to generate blocks of 256 bits each, and then `PAD(M)` will be absorbed with the round function. The ciphertext `C` is generated at the end of this phase. If the last block of `M` is incomplete and its length is `b` bits, i.e., `0 < b < 256`, the last block of `C` will be truncated to the first `b` bits. The detailed algorithm for the encryption phase is shown in Algorithm 2.

Algorithm 1 Round Function of Rocca-S

Input: State $S[0..6]$, X_0, X_1
Output: Updated $S_{\text{new}}[0..6]$
$S_{\text{new}}[0] \leftarrow S[6] \oplus S[1]$
$S_{\text{new}}[1] \leftarrow \text{AES}(S[0], X_0)$
$S_{\text{new}}[2] \leftarrow \text{AES}(S[1], S[0])$
$S_{\text{new}}[3] \leftarrow \text{AES}(S[2], S[6])$
$S_{\text{new}}[4] \leftarrow \text{AES}(S[3], X_1)$
$S_{\text{new}}[5] \leftarrow \text{AES}(S[4], S[3])$
$S_{\text{new}}[6] \leftarrow \text{AES}(S[5], S[4])$

Algorithm 2 Rocca-S Encryption

Input: Padded Message $\text{PAD(M)}[0 \ldots m-1]$, State S; $m = |PAD(M)|/256$
Output: Ciphertext Blocks $C[0 \ldots m-1]$
for $i = 0$ to $m - 1$ **do**
    $C[i]_0 \leftarrow \text{AES}(S[3] \oplus S[5],\ S[0]) \oplus \text{PAD(M)}[i]_0$
    $C[i]_1 \leftarrow \text{AES}(S[4] \oplus S[6],\ S[2]) \oplus \text{PAD(M)}[i]_1$
    $S \leftarrow R(S,\ \text{PAD(M)}[i]_0,\ \text{PAD(M)}[i]_1)$
end for

Bottleneck. The *Round function* is the most compute-intensive component of Rocca-S. The initialization phase and the finalization phase call the round function 16 times, while the associated data processing and encryption phase invoke the function based on the number of AD and M blocks, respectively. Table 4 shows the cycle breakdown of our baseline Rocca-S implementation ($S_{Rocca-S}$) for the phases, initialization, process AD, encryption, and finalization. We observe that the four phases consume 98, 194, 2002, and 98 cycles, respectively, for 500 blocks. This implies that the encryption phase is the bottleneck. We discuss the optimizations to reduce the cycles required to encrypt, and design cryptographic core variants in §4.3.

4 Design

In this section, we describe the design overview and discuss the details of each *FlexMesh* component.

4.1 *FlexMesh* design overview

FlexMesh architecture (see Fig. 6) comprises (a) *data plane:* the cryptographic accelerator pipeline on the FPGA NIC, and (b) *control plane:* the orchestrator module at the host, which periodically monitors the workload statistics, and dynamically (re)configures the accelerator pipeline components. The data plane components include (a) the pre-enqueue function, (b) a *size-aware* load balancer, (c) a *workload-aware* scheduler, and (d) the cryptographic cores. The FPGA resources and the onboard memory are also consumed to store (a) the dataplane components, (b) the workload distribution

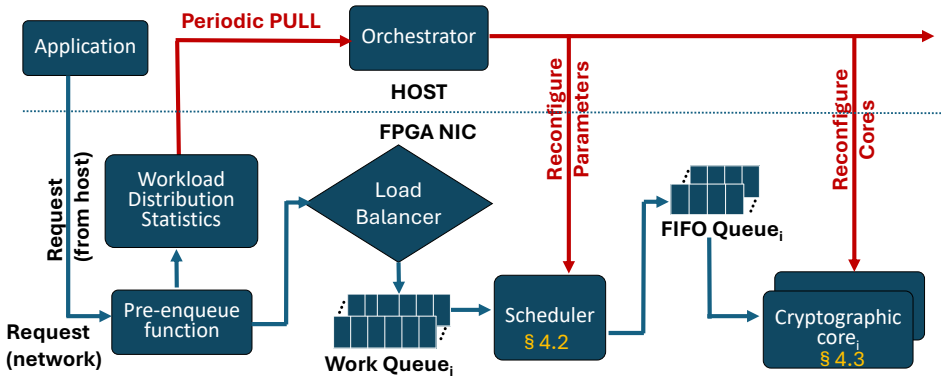


Fig. 6. FlexMesh Design

.

statistics, and (c) the out-of-order (OOO) processed requests. We discuss the choice of parameters and their values for the load balancer and scheduler in §5.4. We now discuss each design component.

Pre-enqueue function. This component is responsible for three primary tasks. (1) Aggregation of incoming requests: The requests are aggregated into a message for protocols such as TLS, which operate on per-message (de)encryption. In contrast, the 5G encryption standards, such as Rocca-S, Snow-V, ZUC, and IPSec, implement per-packet (de)encryption. (2) Append and initialize request metadata: The request descriptor for a request "i" is initialized with the metadata, including: (a) rank, $r_i.rank$, this parameter is used by the scheduler to prioritize requests to reduce the memory overhead to store out-of-order requests. The pre-enqueue function assigns a unique rank to each request size range, with the largest range assigned the rank '0', indicating the highest priority. For example, if the maximum permissible request length is 8000B, then the request length range of 7500B to 8000B is assigned a rank "0", 7000B to 7500B is assigned a rank "1", and so on; (b) request length, $r_i.len$, is used by the load balancer for request classification; and (c) $t^i_{eligible}$ indicates the time after which the request is eligible for scheduling. (3) Maintain workload distribution statistics: The pre-enqueue function keeps track of the distribution of request counts for each rank, *i.e.*, request range. The orchestrator periodically retrieves these statistics to determine if the workload conditions have changed. If changes are detected, the orchestrator uses the monitored statistics to identify a new configuration for the load balancer and the scheduler (see §C).

Size-aware load balancer. To improve the balance between performance and resource utilization, *FlexMesh* proposes the use of variable-sized cryptographic cores. The orchestrator deploys multiple instances of these cryptographic cores to address the scaling demands of incoming requests. As shown in Fig. 13, the performance of Rocca-S cryptographic cores varies with request lengths, *i.e.*, the M core performs best from request sizes of 128 bytes, while the L core outperform at request sizes over 1456 bytes (see Fig. 2 for AES-GCM details). Consequently, standard load distribution techniques like round robin (RR) may not be optimal. *FlexMesh* proposes a size-aware load balancer that classifies incoming requests and allocates them to the respective core's work queues based on size thresholds set by the orchestrator.

Worload-aware scheduler. In *FlexMesh*, based on the scaling demand and resource availability, the orchestrator can deploy "n" variable-sized cryptographic cores (C_i), with "k_i" instances ($i = 1..n$). Each core C_i is assigned a logical work queue (WQ_i) and a short FIFO queue ($FIFO_i$). The load balancer enqueues the requests in the work queue, WQ_i, based on the classifier's predicate. The scheduler picks the appropriate request from the work queue and enqueues it in the FIFO queue, based on the scheduling policy. The scheduler design must adhere to the following primary

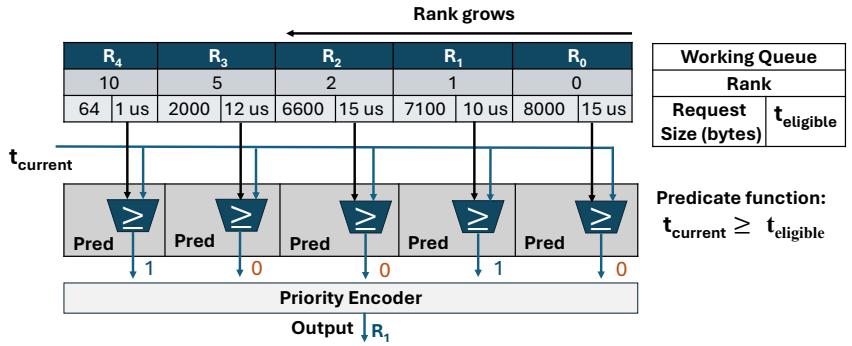


Fig. 7. $FlexMesh_p$ scheduler. To reduce on-device memory to store OOO, the scheduler prioritizes the packets of a flow based on the request size. The predicate function avoids starvation by filtering packets that have waited longer than t_{thresh}. The current time, $t_{current}$, in this example is $10\mu s$.

properties: (a) flexibility: the scheduler parameters must be configurable, (b) work-conserving: the cryptographic cores must never remain idle if the system has pending requests, (c) scalability: the scheduler must manage enough concurrent flows (say, $\sim 100K$), and (c) performance: can take scheduling decision at line rate. We leverage the PIEO scheduling abstraction [58] that supports these scheduler properties. To utilize PIEO, we decide the scheduling policy based on our use case and identify the right configuration values for the scheduler parameters. A detailed discussion of the *FlexMesh* scheduler with priority ($FlexMesh_p$) design is discussed in §4.2.

Given that the scheduler is programmable, we compare scheduling policies in §6, viz., round-robin, non-work-conserving scheduler with load balancer ($FlexMesh_{nWC}$), work-conserving with load balancer (*FlexMesh*), and work-conserving with load balancer and request-size prioritization ($FlexMesh_p$), and the deployer can make a choice based on the workload and the objective function. We explain all the scheduling policies with the help of a toy example in §B.

Variable-sized cryptographic cores. As discussed earlier, the objective of designing variable-sized cryptographic cores is to minimize the performance and efficiency tradeoff. We discuss the design of variable-sized cryptographic cores for Rocca-S and AES-GCM algorithms in §4.3.

Orchestrator. The orchestrator serves as *FlexMesh*'s control plane that periodically monitors dynamic workload characteristics, determines if the data plane configuration needs to change, deploys (or removes) cryptographic core instances, and configures the load balancer and scheduler parameters.Given the monitored load, we design a model to predict the optimal set of cryptographic cores (details in §C). A deployer configures the model's parameters to guide the orchestrator's decision. For example, the deployer can assign tolerance limits beyond optimal configuration for latency, resource efficiency, and throughput based on the use-case demand and the QoS requirements.

4.2 *FlexMesh* scheduler

The scheduler needs to address two primary questions: (1) When does an element become eligible for scheduling? and (2) What should be the scheduling order among the eligible elements? The answer depends on the objective function. In the case of an in-network cryptographic system, we state three objectives: (1) minimize the memory required to store out-of-order (OOO) requests [2], (2) avoid starvation, *i.e.*, bounded waiting time, and (3) ensure work-conservation.

To achieve the scheduling objectives, we configure the components and parameters of the PIEO [58] primitive, (a) request's rank, $r_i.rank$; (b) the maximum wait time after which the request

[2]OOO requests do not impact the correctness of the cryptographic operation. For protocols that require message ordering prior to (en)decryption, the requests are aggregated during the pre-enqueue phase.

Resource	$M_{Rocca\text{-}S}$	$L_{Rocca\text{-}S}$	$(S, L)_{Rocca\text{-}S}$	$(M, L)_{Rocca\text{-}S}$	$M_{AES\text{-}GCM}$	$L_{AES\text{-}GCM}$	$(S, M)_{AES\text{-}GCM}$	$(S, L)_{AES\text{-}GCM}$
LUT	1.02×	1.16×	1.08×	1.09×	1.58×	2.44×	1.29×	1.72×
BRAM	2×	3.33×	2.16×	2.66×	1×	0.93×	1×	0.96×
Flip flops	0.94×	0.94×	0.97×	0.94×	1.36×	1.91×	1.18×	1.45×
Dynamic Power (W)	1.03×	1.06×	1.03×	1.04×	1.30×	1.71×	1.15×	1.35×
Gate Equivalence	1.18×	1.51×	1.25×	1.35×	1.33×	1.78×	1.16×	1.39×

Table 3. Comparing Rocca-S and AES-GCM's resource utilization for individual and scaled cryptographic cores, relative to the corresponding S and (S, S) variants, respectively.

is eligible to be scheduled, t_{thresh}; and (c) *the predicate function* that selects the next request to be scheduled. Fig. 7 shows the design of the $FlexMesh_p$ scheduler.

Scheduler workflow. To schedule a request for a cryptographic core, C_i, the scheduler works in two stages.

Stage 1: Filter all eligible requests. To maintain a bound on the request wait time (t_{thresh}), we define a parameter, $t^i_{eligible}$ for request "i", as

$$t^i_{eligible} = t_{thresh} + t^i_{arrival\_time}; \qquad if(t_{current} \geq t^i_{eligible})\ \{r_i.eligible = true;\}$$

If $t_{current} \geq t^i_{eligible}$, *i.e.*, request r_i has waited longer than the maximum admissible wait time, then mark this request as eligible for scheduling, *i.e.*, $r_i.eligible = true$. The scheduler checks the request eligibility for all the requests in the work queue, WQ_i, and the filtered (*i.e.*, eligible) requests proceed to the next stage. *FlexMesh* configures the scheduler parameter, t_{thresh}, based on the application Service Level Objectives (SLO). A higher t_{thresh} value implies a higher waiting time in the queue.

Stage 2: Choose amongst the eligible requests. The scheduler evaluates the predicate function (user-defined) to pick one element out of the eligible ones. (see Fig. 7) To minimize the memory requirement to store OOO requests, we assign the request's rank, $r_i.rank$, based on the request size range. The scheduler maintains the list data structure with request descriptors sorted by the request rank. The scheduler hardware computes the $r^i_{eligible}$ for all the requests in the list using parallel comparators. The predicate function selects the eligible request ($r^i_{eligible} = true$) with the smallest rank (*i.e.*. the one with the largest request length range) from the head of the list [3]. If the filtered request list is *empty*, *i.e.*, no eligible requests, the output of the predicate evaluation is $NULL$. In such a case, the first packet in the queue is chosen (FIFO).

Work-conserving. While scheduling the request for C_i, if WQ_i is empty, the scheduler picks the first packet from other work queues, ensuring a work-conserving solution.

4.3 Designing variable-sized cryptographic cores

To optimize the hardware-accelerated implementation of Rocca-S and AES-GCM, we apply optimizations such as loop unrolling and array partitioning across the entire algorithm, wherever feasible. Loop unrolling is used to increase parallelism by generating multiple instances of loop iterations, reducing control overhead, and enabling concurrent execution of independent operations. Array partitioning transforms memory structures by splitting them into smaller, independently accessible segments, reducing access conflicts and improving memory bandwidth. Table 3 shows the resource utilization of individual and scaled cryptographic cores relative to the corresponding S and (S, S) variants, respectively.

[3]Alternatively, *rank* can be assigned based only on the flow ID, and the predicate function can prioritize requests from the oldest flow. Nevertheless, these predicate functions do not guarantee a bound on the out-of-order memory, as the scheduler deliberately operates independently on each work queue to eliminate shared state and synchronization overheads.

Function	$S_{Rocca-S}$	$M_{Rocca-S}$	$L_{Rocca-S}$
Initialization	98	34	18
Process AD	194	66	34
Encrypt	2002	502	252
Tag generation	98	34	18

(a) Cycles per 500 blocks (i.e., 64Kb)

Function	$S_{Rocca-S}$	$M_{Rocca-S}$	$L_{Rocca-S}$
Encrypt	8	2	1

(b) Cycles per two blocks (i.e., 256b)

Table 4. Performance of Rocca-S components under different parallelism configurations.

Function	$S_{AES-GCM}$	$M_{AES-GCM}$	$L_{AES-GCM}$
Key expansion	127	127	127
GHASH	5027	2782	1435
GCTR	1531	1531	1531
Others	1019	1027	1035

(a) Cycles per 500 blocks (i.e., 64Kb)

Function	$S_{AES-GCM}$	$M_{AES-GCM}$	$L_{AES-GCM}$
GHASH	10	6	3
GCTR	3	3	3

(b) Cycles per block (i.e., 128b)

Table 5. Performance of AES-GCM components under different parallelism configurations.

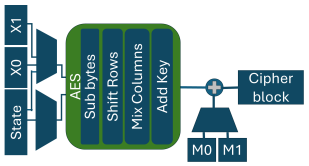


Fig. 8. $S_{Rocca-S}$ cryptographic core design.

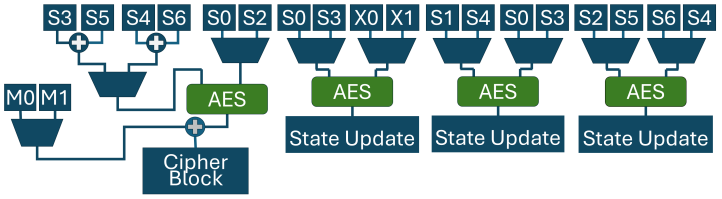


Fig. 9. $M_{Rocca-S}$ cryptographic core design.

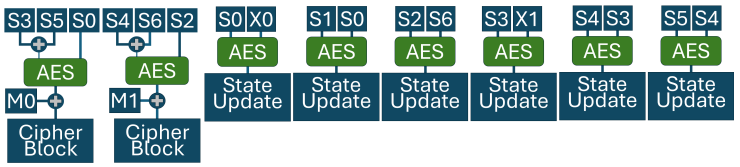


Fig. 10. $L_{Rocca-S}$ cryptographic core design.

4.3.1 Rocca-S cryptographic core design. Table 4 presents the performance in terms of the number of cycles consumed for each Rocca-S phase. We designed the baseline Rocca-S core ($S_{Rocca-S}$) and estimated the cycle information using High-level Synthesis (HLS) [7]. $S_{Rocca-S}$ did not consider round function parallelization, and acts as the baseline. Algorithms 1 and 2 show that the initialization and encryption functions use the AES function, 2 and 6 times, respectively. In total, 8 AES invocations are made to generate one block of ciphertext, including 2 AES invocations for generating the ciphertext and 6 AES invocations in the Round Function. The Round function operates on two 128-bit blocks. For $64Kb$ of data, *i.e.*, 500 blocks of 128 bits each, the *Encrypt* function consumes 2002 cycles, out of which 2000, *i.e.*, $8 * 250$, (250 AES iterations for 500 blocks) are consumed by the Round function. This implies that it takes 1 cycle to process one AES iteration.

Cryptographic core variants for Rocca-S. The *Round function* invokes *six* independent AES computations per iteration, and the *Encrypt* function has two more AES computations. Since these computations are independent, they can be executed in parallel by leveraging an optimized hardware implementation. For our baseline, *i.e.*, the $S_{Rocca-S}$ core (see Fig. 8), the entire implementation shares a single AES unit. This forces sequential execution but saves on resources, resulting in 8 cycles (2 + 6) in the encrypt function for a 256-bit block (see Table 4).

$M_{Rocca-S}$. We implement four AES units in parallel to compute four AES computations within one clock cycle (see Fig. 9). The two AES calls of the Encrypt Function and two out of the six AES calls of the Round Function can be completed in one cycle. The remaining four AES calls of the

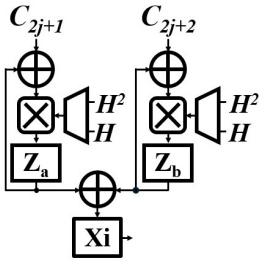


Fig. 11. $M_{\text{AES-GCM}}$ cryptographic core design.

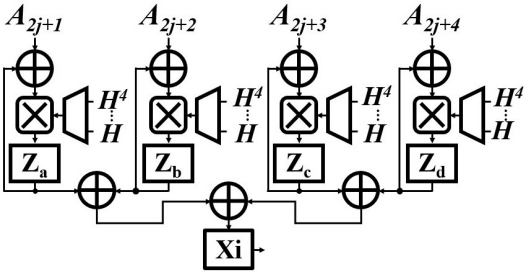


Fig. 12. $L_{\text{AES-GCM}}$ cryptographic core design.

Round Function can be completed in the next cycle. Therefore, each 256-bit block of cipher can be generated in two clock cycles.

$L_{Rocca-S}$. Here, we implement eight AES units in parallel so that all eight AES computations can be completed in one clock cycle to generate a 256-bit cipher block (see Fig. 10).

4.3.2 AES-GCM cryptographic core design. In §A, we have identified that the AES GHASH function, which is used for tag generation, is the primary bottleneck and it is inherently sequential. In AES-GCM, for each input block, GHASH requires one polynomial multiplication with the hash subkey, H. If the length of AD and cipher, C, is "m" and "n" blocks, respectively, the GHASH operation would require a total of $(m + n + 1)$ multiplications, making GHASH function cycle-intensive. Finite field multiplication over GF(2^{128}) has a complexity of $O(n^2)$, where n is the bit-width of the operands (128 bits).To reduce this computational burden, a Karatsuba-based multiplier [79] is employed, which lowers the multiplication complexity from $O(n^2)$ to $O(n^{\log_2 3}) \approx O(n^{1.585})$. Karatsuba's method recursively splits the operands and replaces some of the multiplications with XORs, which are less expensive in hardware. This structure supports pipelining and improves throughput, reducing the cycle count in the tag generation phase without fundamentally altering the algorithm's correctness.

Cryptographic core variants for AES-GCM. By reformulating GHASH as a polynomial over GF(2^{128}), parallelism can be exploited [53, 54]. The chained computation:

$$((((((A_1 \cdot H \oplus A_2) \cdot H \oplus A_3) \cdot H \oplus A_4) \cdot H \oplus A_5) \cdot H \oplus A_6) \cdots \quad (1)$$

can be rewritten by grouping terms and factoring out powers of H:

$$((A_1H^2 \oplus A_3)H^2 \oplus A_5 \cdots)H^2 \oplus ((A_2H^2 \oplus A_4)H^2 \oplus A_6 \cdots)H \quad (2)$$

This rearrangement enables parallel computation of the odd and even indexed input blocks, each forming a separate partial result that can be independently evaluated and combined later. This technique enables 2-way parallelism and can be extended to 4-way or higher parallelism by expressing the GHASH polynomial using higher-order groupings.

Fig. 11 and Fig. 12 show the design of our GHASH cryptographic core implementations that utilize 2-way ($M_{AES-GCM}$) and 4-way parallel ($L_{AES-GCM}$) versions of the GHASH function to reduce cycle consumption, trading off additional hardware resources. Table 5 shows the impact of the optimizations in reducing the cycles for the GHASH function, thereby improving overall throughput.

5 Implementation

This section will describe the implementation of cryptographic core variants and the discrete-event simulator for the load balancer and the scheduler. We implemented three Rocca-S cores ($S_{Rocca-S}$, $M_{Rocca-S}$, $L_{Rocca-S}$) and three AES-GCM cores ($S_{AES-GCM}$, $M_{AES-GCM}$, $L_{AES-GCM}$) on the FPGA ZCU-106 [2] board.

Num Request desc.	Tpt (Mpps)	Enqueue & Dequeue (#Clocks)	Load balancer (#Clocks)	Frequency (MHz)	Latency (ns)
2^8	100	8	1	160	56
2^{10}	65	8	1	140	64

Table 6. Scheduler's performance and overheads

Num	ZCU-106 [2]			Alveo U55C [3]		
Req. desc.	LUT(%)	FF(%)	BRAM(%)	LUT(%)	FF(%)	BRAM(%)
2^8	14	13	0	5.4	0.5	0
2^{10}	49.4	12.4	0	19.1	2.1	0

Table 7. Scheduler's resource utilization on different FPGA platforms.

5.1 Cryptographic Core Implementation

We wrote the AES-GCM and Rocca-S algorithms using the High Level Synthesis (HLS) [7] software tool in C/C++. The AES algorithm and the Rocca-S algorithms spanned around $1.3K$, and 460 lines of code, respectively. In HLS, we tested the software code with simulation, synthesis, co-simulation, and implementation. We verified the correctness of the algorithm by matching results with the test vectors from the AES-GCM RFC [52] and referred to the research paper for Rocca-S [13].

In the implementation of AES-GCM, we defined macros for modular programming. The macros help the automatic generation of variable-sized AES-GCM cores based on the configured number of instances of GCTR and GHASH. The resultant core ran at $240Mhz$. Using Vivado [71], we made a block design in which we connected the AES-GCM core to the board's CPU, a DMA engine, and BlockRAM (BRAM) to store the inputs. After validating the block design, we generate the required bitstream (*i.e.*, executable) for the FPGA fabric. The workload generator at the FPGA CPU was written using the Vitis tool [4], and the FPGA was connected to the host via JTAG [6]. We generated the workload with varying packet sizes, and using real-world workloads (see Table 8). We were restricted to a maximum request size of $8000B$, since our board did not have enough memory. For larger requests, we worked on fragmented requests.

5.2 *FlexMesh* scheduler hardware implementation

We implemented the pre-enqueue function and the load balancer on the ZCU-106 FPGA board to generate the request descriptors for the scheduler. We rely on the PIEO implementation [58] for the $FlexMesh_p$ scheduler's enqueue and dequeue functions, as it is a well-optimized Verilog implementation. We implemented the Round-Robin scheduler on the ZCU-106 FPGA's CPU with two AES-GCM cryptographic cores on the FPGA to validate *FlexMesh* simulator for scheduling.

Scheduler hardware performance and overheads. Table 6 shows that the *FlexMesh* scheduler's performance depends on the number of request descriptors in the scheduler's list data structure. The number of parallel hardware comparators increases with the list length, resulting in a reduced board frequency that impacts resource efficiency and throughput. The scheduling latency (with load balancer) for 2^8 list length implementation is 9 clock cycles, *i.e.*, 56 *ns* running at 160 Mhz. The scheduler throughput is 100 $Mpps$, *i.e.*, 282 $Gbps$, assuming an average packet size of IMIX traffic [69]. This throughput is more than sufficient to meet the requirements of our current cryptographic core implementations.

Scheduler resource overhead. Table 7 maps PIEO scheduler's resource utilization to the FPGA boards ZCU-106 and Alveo U55C. The scheduler utilizes significant resources on smaller FPGA boards. However, these FPGA boards can leverage *FlexMesh*'s workload-aware, variable-sized cryptographic cores with a round-robin scheduler design, without significant scheduling and resource overheads, *i.e.*, 1 clock cycle (4 *ns*) for the load balancer, and 0.03% and 0.01% of LUT and FF utilization.

Dynamic reconfiguration overhead. The time required to load a partial bitstream into a reconfigurable region of an FPGA, *i.e.*, Dynamic Partial Reconfiguration (DPR) depends on [12]: (a) bitstream size, and (b) reconfiguration bandwidth (*e.g.*, ICAP/MCAP/JTAG). For example, the Rocca-S bitstream DPR time is $\sim 2.33sec$ on the ZCU-106 FPGA board via JTAG, whereas the estimated

DPR time for the Alveo U55C FPGA board using PCIe MCAP is 24*msec* [12]. That is, *FlexMesh*'s dynamic reconfiguration framework is useful when the workload characteristics are stable for at least a few seconds. We recommend using high-bandwidth interfaces such as ICAP and MCAP to limit the reconfiguration time to 10*s* of milliseconds.

5.3 Discrete-event simulator for load balancing and scheduling

We implemented a discrete-event simulator for the size-aware classification-based load balancer, and all the scheduling policies, including *FlexMesh* versions and round-robin. This code spanned around 1.5*K* lines of code. We obtain the request processing latencies from our FPGA-based implementation and feed the service time information to our simulator. Our simulator implementation incorporates the real-world service time and the estimated hardware overheads of the scheduler. Since FPGAs are cycle-accurate by design, the simulator results represent the real-world performance. To validate the performance metrics generated by our simulator, we compared our FPGA-based Round Robin scheduler results with the simulation results. Our simulated request completion time is overestimated by up to 12% (*i.e.*, 0.46*ns*) for small request sizes, *i.e.*, 80*B*, and up to 3% for large request size, *i.e.*, 1.4*KB*, compared to the FPGA results.

5.4 *FlexMesh*'s parameter configuration

Threshold selection for the load balancer, $Thr_{X,X}$ to distribute requests across multiple cryptographic core instances.

To distribute the load efficiently across variable-sized cryptographic core instances, executing the same cipher algorithm, our classifier-based size-aware load balancer must be configured with threshold value(s). The threshold depends on the performance of the deployed cryptographic cores. We profile the cryptographic core variants for multiple request sizes to decide the threshold. Fig. 13 shows that Rocca-S's *M* core outperforms the *S* core from 128*B*, and *L* outperforms *M* core after 1456*B*, hence setting these values as thresholds for the asymmetric Rocca-S cryptographic cores, *(S,M)* and *(M,L)*, respectively. Fig. 2 shows the threshold configuration for AES-GCM cores.

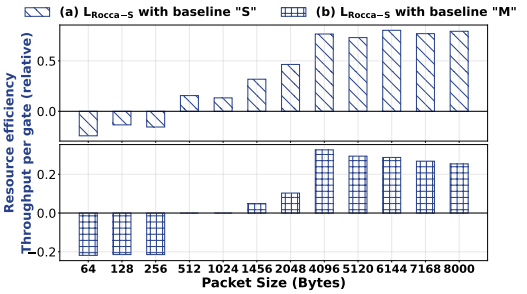


Fig. 13. Rocca-S's efficiency, i.e., throughput per gate, relative to the cryptographic cores, S and M.

Configuration of number of request descriptors for $FlexMesh_p$. $FlexMesh_p$ scheduler can make better choice of the largest size request by comparing more requests in parallel, *i.e.*, with more comparators. However, the FPGA resource utilization increases with the increase in comparators, *i.e.*, number of request descriptors (see Table 7). We choose 256 request descriptors to limit the scheduler's resource utilization.

6 Evaluation

Experiment setup. We used an AMD Ryzen 9 5950X 16-Core Processor host with 32 cores for our scheduler simulation, and tested our cryptographic core implementations on the FPGA (ZCU-106+ [2]) board.

Configurable parameters. (a) Threshold values for the load balancer when asymmetric cryptographic hardware is deployed, viz., $Thr_{rocca-SL}$, $Thr_{rocca-ML}$, $Thr_{aes-gcm-SM}$, $Thr_{aes-gcm-SL}$ (see §5.4), (b) Threshold for request waiting time, and (c) Scheduling policy, viz, Round-robin

Workload	Workload characteristics					Application type	Metric to be prioritized
	Min (bytes)	Avg (bytes)	Max (bytes)	Data Rate (Gbps)	Inter-packet-time (sec)		
IOT (MAWI)	50	1010.74	1504	3.90	3.2	DNS,HTTP	Resource
5G Video-streaming	76	1080.34	1444	0.087	7.8	Youtube	Latency
5G URLLC	96	615.07	4764	8.83	0.2	Meet	Resource/Power
5G MMTC	88	854.37	10048	12.07	0.3	Gmail,Google Fi,Teams	Resource/Power

Table 8. 5G use cases depicting variability in data rate and traffic density requirements [5].

Performance & Resource util. (FPGA-based AES-GCM)	AES-GCM state-of-the-art FPGA solutions				*FlexMesh*		
	Cibik [23]	Henzen [30]	Zhou [79]	Stavrou [61]	$S_{AES-GCM}$	$M_{AES-GCM}$	$L_{AES-GCM}$
Number of clock cycles	21	11	19	31	13	9	6
Operating frequency (in MHz)	229	233	287	227	240	240	240
LUT utilization	85K	760K	211K	81K	36K	81K	142K
BRAM utilization	200	450	59	1	89	89	89

Table 9. Comparing *FlexMesh* with state-of-the-art AES-GCM FPGA solutions.

ASIC vs. FPGA Performance (AES-GCM)	In-path (de)encryption (64B)				Bulk (de)encryption (2KB)			
	ASIC [47] (4 cores)	$S_{AES-GCM}$ (1 core)	$M_{AES-GCM}$ (1 core)	$L_{AES-GCM}$ (1 core)	ASIC [47] (2 cores)	$S_{AES-GCM}$ (1 core)	$M_{AES-GCM}$ (1 core)	$L_{AES-GCM}$ (1 core)
Latency (ns)	2145	250	150	134	30749	6656	4608	3072
Throughput	884 Mbps	2 Gbps	3.4 Gbps	3.8 Gbps	125 Gbps	2.46 Gbps	3.55 Gbps	5.33 Gbps

Table 10. Comparing *FlexMesh* with commercial Nvidia's Bluefield AES-GCM ASIC solution.

(RR), *FlexMesh* $_{nWC}$ (non-work-conserving), *FlexMesh* (work-conserving), and $FlexMesh_p$ (work-conserving with request size prioritization)

Metrics. (a) *Throughput* refers to the number of bits encrypted per unit time; (b) *Resource-efficiency* refers to the maximum achievable throughput per unit FPGA resource (*i.e.*, gate); (c) *Power-efficiency* refers to the maximum achievable throughput per unit Watt; (d) *Latency* includes the request's processing time and the waiting time in the queue; and (e) *Memory overhead due to out-of-order (OOO) processing* refers to the amount of additional memory required to store OOO requests.

Workload. We test *FlexMesh* over real-world 5G datasets [5, 41, 44] with diverse use cases such as IoT, video streaming, 5G-URLLC, and 5G-MMTC (see Table 8).

Research Questions.

[RQ1] How do *FlexMesh*'s AES-GCM and Rocca-S cryptographic cores perform compared to the state-of-the-art FPGA, ASIC, and commercial solutions? Table 9 shows that the best performing *FlexMesh* core, $L_{AES-GCM}$, needs 5.1× fewer cycles for encryption and authentication compared to the slowest state-of-the-art FPGA solution [61] but uses 99% more BRAM resource. However, *FlexMesh*'s $L_{AES-GCM}$ uses 1.8× fewer cycles, 89% and 80% fewer LUT and BRAM resources, respectively, compared to the fastest state-of-the-art FPGA solution [30].

Table 10 compares *FlexMesh* AES-GCM (de)encryption performance with the commercial AES-GCM ASIC designed for Nvidia's Bluefield DPU [1]. The ASIC performance is obtained from Nvidia's DOCA benchmarks [47], and *FlexMesh* performance is obtained using the ZCU-106 FPGA board. In-path latency measures the duration between the submission of a single request and its completion across 4 parallel cores. In contrast, the bulk mode submits a group of requests from the same flow (a single decryption key is used) in parallel, utilizing 2 cores and 4 threads, to improve throughput. Nvidia's AES-GCM ASIC shows 93.8% higher average latency, and 0.26× throughput compared to *FlexMesh*'s $L_{AES-GCM}$ cryptographic core for in-path decryption of 64-byte requests, and up to 90% higher average latency compared to *FlexMesh*'s L_{AES_GCM} cryptographic core for large (*i.e.*, 2KB) requests in bulk mode. However, Nvidia's AES-GCM ASIC outperforms *FlexMesh* in terms of throughput by up to 24× in bulk mode, as the ASIC batches the requests, processes them in parallel across the cores, and amortizes the data fetch time. In contrast, a single *FlexMesh* core processes the requests sequentially.

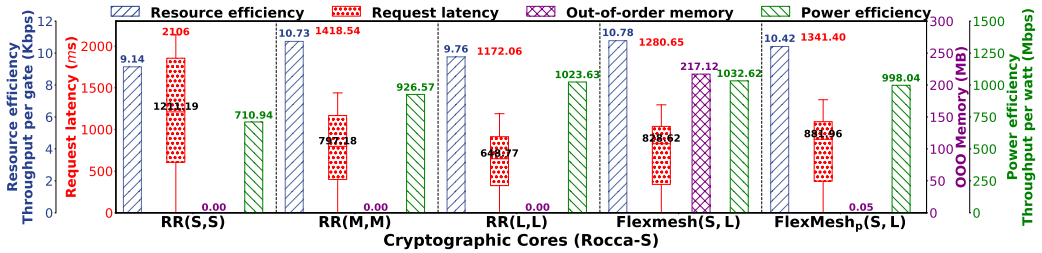


Fig. 14. Comparing scheduling policies for 5G-MMTC across Rocca-S's symmetric and asymmetric cryptographic cores. *FlexMesh* improves the efficiency with OOO memory trade-off, while $FlexMesh_p$ focuses on minimizing the tradeoff.

ASIC vs. FPGA (Rocca-S)	ASIC: serial ver. Anand et. al [13]	ASIC: unrolled ver. Anand et. al [13]	$S_{Rocca-S}$ (1 core)	$M_{Rocca-S}$ (1 core)	$L_{Rocca-S}$ (1 core)
Number of clock cycles (AD: 1024b, PT: 2048b)	2368	22	292	92	48
Operating frequency (in Mhz)	10	10	240	240	240
Throughput	1.53 Mbps	284 Gbps	8 Gbps	32 Gbps	64 Gbps
Resource utilzation (Gate Equivalence)	12.7K	1028K	203K	241K	309K
Power consumption	0.12 microW	3.5 mW	4.3W	4.5W	4.7 W

Table 11. Comparing *FlexMesh* with state-of-the-art Rocca-S ASIC solution.

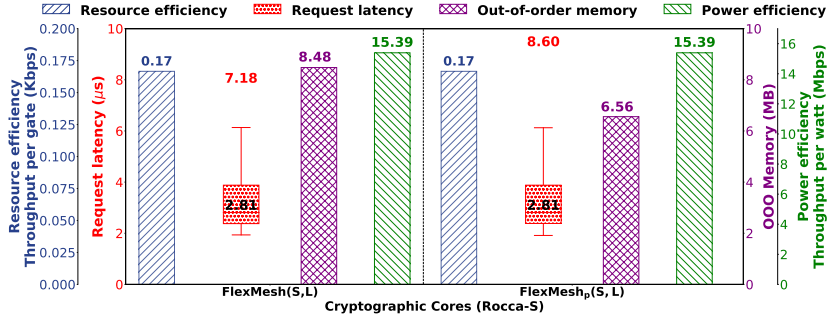


Fig. 15. Comparing OOO memory overhead of *FlexMesh* and $FlexMesh_p$ for Rocca-S asymmetric cores for 5G-gNodeB workload. The OOO improvement is marginal due to low load, but the performance does not degrade with prioritization.

Table 11 compares *FlexMesh*'s FPGA-based Rocca-S performance and resource/power utilization with the state-of-the-art [13] Rocca-S serial and unrolled ASIC variants. We observe that the unrolled state-of-the-art version 4.4× higher throughput, 54% lower latency, and 99.9% lower power consumption[4] compared to *FlexMesh*'s $L_{Rocca-S}$ cryptographic core, at the cost of 3.3× higher resource utilization. *FlexMesh*'s $L_{Rocca-S}$ shows 41.8× higher throughput, 97.9% lower latency compared to the serial version of the state-of-the-art Rocca-S ASIC, at the cost of 95.8% and 99.9% higher resource utilization and power consumption, respectively.

[RQ2] What are the tradeoffs across scheduling policies in terms of performance and efficiency? Fig. 14 compares the scheduling policies, round-robin (RR), non-work-conserving $FlexMesh_{nWC}$, *FlexMesh* (work-conserving), and $FlexMesh_p$ (work-conserving with request-size prioritization) across symmetric and asymmetric cryptographic cores. The work-conserving schedulers, *FlexMesh* and $FlexMesh_p$, outperform in terms of resource efficiency, power efficiency, and packet latency. We observe

[4]FPGAs are more power-hungry compared to ASICs.

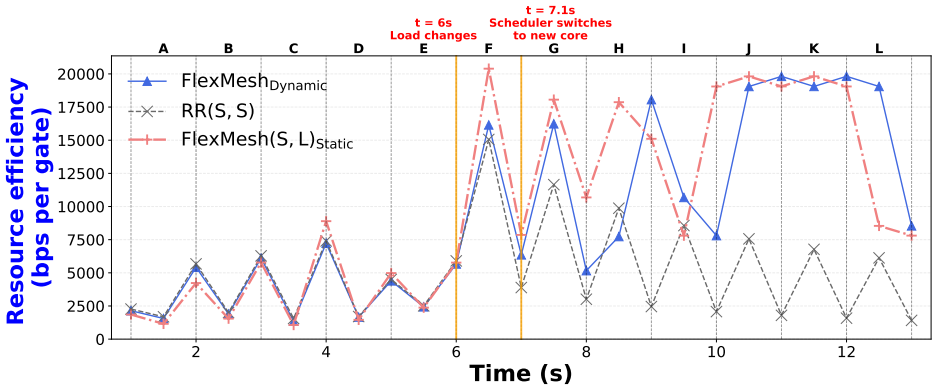


Fig. 16. Comparing performance of static (Round-Robin (RR) and $FlexMesh_{static}$) and dynamic scheduling policies for 5G-Mixed workload. RR uses (S,S) core, $FlexMesh_{static}$ uses (S,L) cores, and $FlexMesh_{dynamic}$ switches between the two.

that $FlexMesh_p$ (M,L) outperforms the baseline, $RR(S, S)$, in terms of resource-efficiency, power-efficiency, and latency by 17.94%, 45.24%, and 31.58%, respectively. Additionally, (L, L) core performs almost at par with $FlexMesh_p$ (M,L), but it is inefficient due to high FPGA hardware usage.

[RQ3] What is the performance tradeoff that one has to pay in terms of additional memory requirement to store OOO requests? Fig. 14 and Fig. 15 show the reduction in the memory required to store OOO requests for 5G-MMTC and 5G-gNodeB workloads, respectively. From Fig. 14, it is intuitive that *single queue* RR and non-work-conserving scheduler *FlexMesh* $_{nWC}$ will have minimal out-of-order requests since the flow order is not disturbed much. $FlexMesh_p$ reduces the impact of *FlexMesh*'s OOO scheduling by 99% and 22.64% for 5G-MMTC and 5G-gNodeB workloads, respectively.

[RQ4] How does dynamic scheduling, $FlexMesh_{dynamic}$, perform compared to static scheduling for real-world workload? Fig. 16 compares the resource efficiency of $FlexMesh_{dynamic}$ scheduling with *FlexMesh* using Rocca-S(S,L) and Round-Robin scheduling that uses Rocca-S(S,S) cores. In the case of $FlexMesh_{dynamic}$, the orchestrator monitors the workload characteristics every 1 *sec*. $FlexMesh_{dynamic}$ was using RR(S,S) core from $t = 0s$, and the orchestrator observes a significant change in the load during the interval, F, and switches to Rocca-S(S,L) core with *FlexMesh* scheduling at interval, G, at $t = 7.1s$. The switching for dynamic scheduling requires $100ms$, including analytical model computation and DPR overhead. We observe that $FlexMesh_{dynamic}$ catches up with the best static design, achieving 52% and 7% improvements in resource efficiency over RR(S,S) and *FlexMesh* scheduling algorithms, respectively.

7 Related Work

Cryptographic accelerators. Modern CPUs support specialized CPU instructions for accelerated cryptographic processing (AES-NI [28] and SHA [29]), but they take up significant CPU cycles. SmartNICs [1, 19, 48]) come with off-data-path, fixed-function accelerators (or ASICs) for popular security algorithms. However, due to high fabrication costs, these fixed-function accelerators are typically only developed for cryptographic algorithms that have demonstrated long-term security. Prior works have proposed reconfigurable, on-data-path accelerators using programmable network cards or switches. For example, a) ChaCha cryptographic algorithm is implemented on the Netronome smartNIC [80] and Intel Tofino [75] switch, (b) Intel Tofino switches with implementations of HalfSipHash [55, 74] and partial AES [21], and (c) FPGA hardware implementations for 5G/6G

Cryptographic algorithm	Sub-function unrolled to generate cryptographic core variants	Key idea
AEGIS-256 [70]	AES round function	AEGIS uses multiple AES blocks to update the state elements after each encryption round. AEGIS hardware can be designed to increase the number of encrypted blocks per clock cycle by leveraging parallel instances of the AES round function.
ChaCha20 [56]	Quarter-round functions	ChaCha20's state matrix is updated using the quarter-round function after every use. ChaCha20 hardware can unroll up to four quarter-round functions to accelerate the encryption process.
SNOW-V-GCM [26]	GHASH function	In the case of SNOW-V-GCM, the bottleneck GHASH function could be used to generate cryptographic core variants based on FlexMesh's AES-GCM design.
SNOW-V [38]	Key generation function	Snow-V's key generation function, which uses the LFSR and FSM, can be unrolled to generate multiple keys in a single clock cycle.
ASCON [27]	Round transformation function	ASCON hardware can unroll the round transformation function to compute multiple rounds in a single clock cycle, viz., 1x, 2x, 3x and 6x, where the multiplier represents the unroll degree.
ISAP [24, 36]	Round transformation function	ISAP variants, ISAP-A-128A and ISAP-A-128, use the round transformation function similar to ASCON, and this function can be unrolled to achieve speedup.

Table 12. Example next-generation cryptographic algorithms that can leverage *FlexMesh*

cryptographic algorithms such as Snow-V [15], ZUC [72], Rocca-S [13], and AES-GCM [66, 67] cipher algorithms. However, the reconfigurable cryptographic accelerators are designed for either a high-performance [10, 18, 38, 43] or a resource-efficient solution [38, 50], but not both. We design workload-aware cryptographic cores that help balance the performance and efficiency trade-off.

Load balancing. To scale the system, multiple cryptographic cores are programmed on the reconfigurable FPGA NIC. Prior works propose load balancing algorithms such as Join shortest queue (JSQ), Join Lightest queue (JLQ) [57], and Join Bounded-Shortest-Ranked-Queue (JBSRQ) [42], that assume homogeneous target servers. On the contrary, *FlexMesh* uses workload-aware thresholds for request classification across heterogeneous cores.

Scheduling. Prior works, PIFO [59] and PIEO [58], design in-network scheduling frameworks that can be used to express multiple scheduling algorithms at line rate. To implement aging, PIEO (15Mpps, 10Gbps) strictly selects the smallest-ranked eligible packet, whereas PIPO [77] (70Mpps, 40Gbps) scales the solution by approximating the requirement to a small-ranked eligible packet. BMW-tree [73], BBQ [14], and ClubHeap [22] design optimized hardware data structures to scale up to $100K+$ flows with above $100Gbps$ data rates, and support for multiple tenants. *FlexMesh* adds a new use-case, scheduler for heterogeneous compute units, by reusing the SOTA components, and thereby complements the existing works.

8 Discussion

8.1 Extending *FlexMesh* to popular next-generation (5G/6G) cryptographic algorithms.

To generate cryptographic core variants, the key idea of *FlexMesh* is to identify compute-intensive and independent sub-functions that can be unrolled to multiple degrees based on performance requirements and resource efficiency. Leveraging existing literature and cryptographic standard specifications, we have identified the sub-functions of popular next-generation cryptographic algorithms that can be parallelized (a.k.a., unrolled). Table 12 shows how *FlexMesh* can be extended to a broader class of cryptographic algorithms such as: (a) AEGIS-256 [70]: a high-performance AEAD candidate for 6G networks; (b) ChaCha20 [56]: TLS1.3 and QUIC standard, and also a quantum-resistant candidate for 6G security; (c) SNOW-V-GCM and SNOW-V [26, 38]: 5G standard and a 6G candidate to secure communication in virtualized network environments; (d) ASCON [27]: a NIST standard, and a candidate for 6G IoT and embedded security; and (e) ISAP [24, 36]: a NIST standard, and a candidate for low power IoT devices that require strong side-channel resistance.

8.2 Security and reliability concerns with hardware-based cryptographic primitives

Ensuring security and reliability for FPGA/SoC-based hardware accelerator solutions is extremely critical. We discuss the state-of-the-art that complements *FlexMesh* towards security and reliability.

Variable-sized cores and load balancing expose side channels. Variable-size cores and queueing overheads due to load balancing introduce variability in task execution latency, power consumption, electromagnetic (EM) signatures, and resource contention, thermal, and spatial fingerprints. Prior work has demonstrated that attackers can construct side channels from the observability of timing [63], power [11, 60], contention [63], and thermal patterns [16, 78].

AI-based side-channel attacks. Deep learning AI models can efficiently identify complex patterns in side-channel data like power consumption, EM emissions, and timing variations, allowing for faster and efficient extraction of cryptographic keys [17]. These models significantly reduce the amount of side-channel data (traces) an attacker needs to collect to successfully recover a secret key, making attacks more feasible.

Mitigation for side-channel attacks. Resource partitioning [39], secure reconfiguration channels [64], randomized or constant-time scheduling [65], and power balancing [34] can help mitigate the side channel attacks. Solutions such as Coyotev2 [51] and [33], implement a shell that partition resources across tenants (or cores), and supports dynamic partial reconfiguration of services and user logic. AI-based systems can monitor side-channel leakage in real-time and detect unusual patterns indicative of an ongoing attack [17].

Dynamic reconfiguration impacts reliability. Dynamic partial reconfiguration (DPR) results in reliability concerns such as bitstream corruption [64, 76], attacks during reconfiguration [20], and stress on FPGA's solder joints and routing interconnects due to repeated thermal cycling [40], which makes fault identification and recovery complex. Prior works implement wear-leveling across FPGA fabric [62], thermal-aware scheduling [31], error-detecting bitstreams [76], centralized fault logging, and checkpointing [68] mechanisms for reliability.

9 Conclusion

We present *FlexMesh*, an in-network cryptographic primitive that leverages the workload's request size characteristics to design cryptographic accelerators with high resource and power efficiency, without compromising the performance. We design and implement variable-sized cryptographic primitives for next-generation mobile networks (*i.e.*, Rocca-S), and a programmable scheduler that prioritizes requests to minimize the on-board storage for OOO requests, and ensures work-conservation.

10 Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. We thank Mayank Rawat, Lasani Hussain, Sumit Kumar, and Arjun Temura for their feedback and help on the earlier drafts. This work is supported by the IHUB NTIHAC Foundation, IIT Kanpur (Grant Id: IHUB-NTIHAC/2023/01/6).

References

[1] 2021. NVIDIA BLUEFIELD-3 DPU PROGRAMMABLE DATA CENTER INFRASTRUCTURE ON-A-CHIP. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.
[2] 2021. *Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit.*
[3] 2023. *Alveo SN1000 SmartNIC Accelerator Card.*
[4] 2025. AMD Vitis™ Unified Software Platform. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html
[5] 2025. genesys-neu/TRACTOR. https://github.com/genesys-neu/TRACTOR original-date: 2023-05-08T14:42:03Z.

[6] 2025. jtag targets • Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400) • Reader • AMD Technical Information Portal.

[7] 2025. Vitis HLS. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

[8] 3GPP. 2017. 5G 3GPP specifications. https://www.3gpp.org/ftp/Specs/archive/23_series/23.502/.

[9] 3gpp. 2025. Service requirements for next generation new services and markets. https://www.3gpp.org/ftp/Specs/archive/22_series/22.261/.

[10] Karim M. Abdellatif, R. Chotin-Avot, and H. Mehrez. 2013. Improved method for parallel AES-GCM cores using FPGAs. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 1–4. doi:10.1109/ReConFig.2013.6732299

[11] Mahya Morid Ahmadi, Lilas Alrahis, Ozgur Sinanoglu, and Muhammad Shafique. 2023. FPGA-Patch: Mitigating Remote Side-Channel Attacks on FPGAs using Dynamic Patch Generation. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. doi:10.1109/ISLPED58423.2023.10244526

[12] Xilinx AMD. 2025. Vivado Design Suite User Guide: Dynamic Function eXchange (UG909). https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Configuration-Time. [Online; accessed 10-10-2025].

[13] Ravi Anand, Subhadeep Banik, Andrea Caforio, Kazuhide Fukushima, Takanori Isobe, Shisaku Kiyomoto, Fukang Liu, Yuto Nakano, Kosei Sakamoto, and Nobuyuki Takeuchi. 2023. An ultra-high throughput AES-based authenticated encryption scheme for 6G: Design and implementation. In *European Symposium on Research in Computer Security*. Springer, 229–248.

[14] Nirav Atre, Hugo Sadok, and Justine Sherry. 2024. BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 455–475. https://www.usenix.org/conference/nsdi24/presentation/atre

[15] Milad Bahadori, Kimmo Järvinen, and Valtteri Niemi. 2021. FPGA Implementations of 256-Bit SNOW Stream Ciphers for Postquantum Mobile Security. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29, 11 (2021), 1943–1954. doi:10.1109/TVLSI.2021.3108430

[16] Amrou Zyad Benelhaouare, Idir Mellal, Maroua Oumlaz, and Ahmed Lakhssassi. 2024. Mitigating Thermal Side-Channel Vulnerabilities in FPGA-Based SiP Systems Through Advanced Thermal Management and Security Integration Using Thermal Digital Twin (TDT) Technology. *Electronics* 13, 21 (2024), 4176.

[17] Sesibhushana Rao Bommana, Sreehari Veeramachaneni, Syed Ershad, and MB Srinivas. 2025. Mitigating side channel attacks on FPGA through deep learning and dynamic partial reconfiguration. *Scientific Reports* 15, 1 (2025), 13745.

[18] Benjamin Buhrow, Karl Fritz, Barry Gilbert, and Erik Daniel. 2015. A highly parallel AES-GCM core for authenticated encryption of 400 Gb/s network protocols. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–7. doi:10.1109/ReConFig.2015.7393321

[19] Cavium. 2013. Cavium LiquidIO® Server Adapter Family. https://datasheet.octopart.com/CN6130-110SV-G-Cavium-Networks-datasheet-26366670.pdf. [Online; accessed 16-March-2022].

[20] Jayeeta Chaudhuri, Hassan Nassar, Dennis R.E. Gnad, Jörg Henkel, Mehdi B. Tahoori, and Krishnendu Chakrabarty. 2024. Hacking the Fabric: Targeting Partial Reconfiguration for Fault Injection in FPGA Fabrics. In *2024 IEEE 33rd Asian Test Symposium (ATS)*. 1–6. doi:10.1109/ATS64447.2024.10915413

[21] Xiaoqi Chen. 2020. Implementing PaoloEncryption on Programmable Switches via Scrambled Lookup Tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure* (Virtual Event, USA) *(SPIN '20)*. Association for Computing Machinery, New York, NY, USA, 8–14. doi:10.1145/3405669.3405819

[22] Zhikang Chen, Haoyu Song, Zhiyu Zhang, Yang Xu, and Bin Liu. 2025. ClubHeap: A High-Speed and Scalable Priority Queue for Programmable Packet Scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1421–1436. https://www.usenix.org/conference/nsdi25/presentation/chen-zhikang

[23] Peter Cibik, Patrik Dobias, Sara Ricci, Jan Hajny, Lukas Malina, Petr Jedlicka, and David Smekal. 2024. Pushing AES-256-GCM to limits: Design, implementation and real FPGA tests. In *International Conference on Applied Cryptography and Network Security*. Springer, 303–318.

[24] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. 2016. ISAP v2.0. Submission to NIST LWC Project. https://isap.isec.tugraz.at.

[25] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. 2018. A new SNOW stream cipher called SNOW-V. *Cryptology ePrint Archive* (2018).

[26] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. 2018. A new SNOW stream cipher called SNOW-V. *Cryptology ePrint Archive* (2018).

[27] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. 2015. Suit up! – Made-to-Measure Hardware Implementations of ASCON. In *2015 Euromicro Conference on Digital System Design*. 645–652. doi:10.1109/DSD.2015.14

[28] Shay Gueron. 2010. Intel advanced encryption standard (AES) instructions set. *Intel White Paper, Rev* 3 (2010), 1–94.

[29] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghalia, J Guilford, and Gil Wolrich. 2023. *Intel SHA extensions*. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper-402097.pdf

[30] Luca Henzen and Wolfgang Fichtner. 2010. FPGA parallel-pipelined AES-GCM core for 100G Ethernet applications. In *2010 Proceedings of ESSCIRC*. 202–205. doi:10.1109/ESSCIRC.2010.5619894

[31] Fatemeh Hossein-Khani, Omid Akbari, and Muhammad Shafique. 2024. A Two-Level Thermal Cycling-Aware Task Mapping Technique for Reliability Management in Manycore Systems. *IEEE Access* 12 (2024), 113406–113421. doi:10.1109/ACCESS.2024.3443539

[32] Lasani Hussain, Mayank Rawat, Neeraj Kumar Yadav, Sumit Darak, Praveen Tammana, and Rinku Shah. 2023. Microservice-based in-network security framework for FPGA NICs. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. 328–330. doi:10.1109/CCGridW59191.2023.00074

[33] Muhammed Kawser Ahmed, Maximillian Panoff Kealoha, Joel Mandebi Mbongue, Sujan Kumar Saha, Erman Nghonda Tchinda, Peter Esenju Mbua, and Christophe Bobda. 2025. Multi-Tenant Cloud FPGA: A Survey on Security, Trust, and Privacy. *ACM Trans. Reconfigurable Technol. Syst.* 18, 2, Article 23 (April 2025), 44 pages. doi:10.1145/3713078

[34] HanBit Kim, HeeSeok Kim, and Seokhie Hong. 2020. Power-Balancing Software Implementation to Mitigate Side-Channel Attacks without Using Look-Up Tables. *Applied Sciences* 10, 7 (2020), 2454.

[35] Paris Kitsos, Nicolas Sklavos, and Athanassios N Skodras. 2011. An FPGA implementation of the ZUC stream cipher. In *2011 14th Euromicro Conference on Digital System Design*. IEEE, 814–817.

[36] Evangelia Konstantopoulou, George Athanasiou, and Nicolas Sklavos. 2025. Review and Analysis of FPGA and ASIC Implementations of NIST Lightweight Cryptography Finalists. *ACM Comput. Surv.* 57, 10, Article 246 (May 2025), 35 pages. doi:10.1145/3721122

[37] Evangelia Konstantopoulou, George S. Athanasiou, and Nicolas Sklavos. 2023. Securing 5G/6G Communications in Smart Cities: Novel SNOW-V/ZUC-256 Multimode Architectures. In *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. 2363–2369. doi:10.1109/CSCE60160.2023.00383

[38] Evangelia Konstantopoulou, George S Athanasiou, and Nicolas Sklavos. 2023. Towards Secure and Efficient Multi-generation Cellular Communications: Multi-mode SNOW-3G/V ASIC and FPGA Implementations. In *International Symposium on Applied Reconfigurable Computing*. Springer, 159–172.

[39] Jonas Krautter, Dennis R.E. Gnad, Falk Schellenberg, Amir Moradi, and Mehdi B. Tahoori. 2019. Active Fences against Voltage-based Side Channels in Multi-Tenant FPGAs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1109/ICCAD45719.2019.8942094

[40] Lei Li, Xinyu Du, Jibing Chen, and Yiping Wu. 2024. Thermal fatigue failure of micro-solder joints in electronic packaging devices: a review. *Materials* 17, 10 (2024), 2365.

[41] Athanasios Liatifis, Dimitrios Pliatsios, Sotirios Tegos, Ioannis Makris, Nikolaos Mitsiou, Konstantinos Kyranou, Thomas Lagkas, and Panagiotis Sarigiannidis. 2024. NANCY SNS JU Project - VR Video Streaming and iPerf3 Dataset. doi:10.21227/J56T-WW52 Accessed: 2025-06-06.

[42] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. 2023. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1293–1308. https://www.usenix.org/conference/nsdi23/presentation/lin

[43] Ming-Bo Lin and Jen-Hua Chuang. 2024. The Design of a High-Throughput Hardware Architecture for the AES-GCM Algorithm. *IEEE Transactions on Consumer Electronics* 70, 1 (2024), 425–432. doi:10.1109/TCE.2023.3332872

[44] MAWI Working Group. 2024. MAWI Working Group Traffic Archive. https://mawi.wide.ad.jp/mawi/. Accessed: 2025-06-05.

[45] Yuto Nakano, Kazuhide Fukushima, and Takanori Isobe. 2024. *Encryption algorithm Rocca-S*. Internet-Draft draft-nakano-rocca-s-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-nakano-rocca-s/05/ Work in Progress.

[46] NSF. 2025. NSF 25-539: Verticals-enabling Intelligent Network Systems (VINES). https://www.nsf.gov/funding/opportunities/vines-verticals-enabling-intelligent-network-systems/nsf25-539/solicitation#toc.

[47] Nvidia. 2025. Nvidia DOCA Bench. https://docs.nvidia.com/doca/sdk/nvidia-doca-bench.pdf. [Online; accessed 10-09-2025].

[48] Pensando. 2020. Pensando DSC-100 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/DSC-100-ProductBrief-v06.pdf. [Online; accessed 16-March-2022].

[49] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC Offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 18–35. doi:10.1145/3445814.3446732

[50] Lampros Pyrgas and Paris Kitsos. 2021. 5G Security: FPGA Implementation of SNOW-V Stream Cipher. In *2021 24th Euromicro Conference on Digital System Design (DSD)*. 381–384. doi:10.1109/DSD53832.2021.00064

[51] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. 2025. Coyote v2: Raising the Level of Abstraction for Data Center FPGAs. arXiv:2504.21538 [cs.AR] https://arxiv.org/abs/2504.21538

[52] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. 2008. *AES Galois Counter Mode (GCM) Cipher Suites for TLS.* Request for Comments RFC 5288. Internet Engineering Task Force. doi:10.17487/RFC5288 Num Pages: 8.

[53] Akashi Satoh, Takeshi Sugawara, and Takafumi Aoki. 2007. High-speed pipelined hardware architecture for Galois counter mode. In *Information Security: 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007. Proceedings 10.* Springer, 118–129.

[54] Akashi Satoh, Takeshi Sugawara, and Takafumi Aoki. 2007. High-Speed Pipelined Hardware Architecture for Galois Counter Mode. In *Information Security.* Springer Berlin Heidelberg, Berlin, Heidelberg, 118–129.

[55] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. 2019. Cryptographic Hashing in P4 Data Planes. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* 1–6. doi:10.1109/ANCS.2019.8901886

[56] Ronaldo Serrano, Ckristian Duran, Marco Sarmiento, Cong-Kha Pham, and Trong-Thuc Hoang. 2022. ChaCha20–Poly1305 authenticated encryption with additional data for transport layer security 1.3. *Cryptography* 6, 2 (2022), 30.

[57] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. 2023. Turbo: Smartnic-enabled dynamic load balancing of μs-scale rpcs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 1045–1058.

[58] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19).* Association for Computing Machinery, New York, NY, USA, 367–379. doi:10.1145/3341302.3342090

[59] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19).* Association for Computing Machinery, New York, NY, USA, 367–379. doi:10.1145/3341302.3342090

[60] Petr Socha, Jan Brejník, Josep Balasch, Martin Novotnỳ, and Nele Mentens. 2020. Side-channel countermeasures utilizing dynamic logic reconfiguration: Protecting AES/Rijndael and Serpent encryption in hardware. *Microprocessors and Microsystems* 78 (2020), 103208.

[61] Ioannis-T Stavrou. 2016. FPGA Implementation using VHDL of the AES-GCM 256-bit Authenticated Encryption Algorithm. (2016).

[62] Edward Stott and Peter Y. K. Cheung. 2011. Improving FPGA Reliability with Wear-Levelling. In *2011 21st International Conference on Field Programmable Logic and Applications.* 323–328. doi:10.1109/FPL.2011.65

[63] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. 2024. Covert-channels in FPGA-enabled SmartSSDs. *ACM Trans. Reconfigurable Technol. Syst.* 17, 2, Article 27 (April 2024), 23 pages. doi:10.1145/3635312

[64] Florian Unterstein, Tolga Sel, Thomas Zeschg, Nisha Jacob, Michael Tempelmeier, Michael Pehl, and Fabrizio De Santis. 2020. Secure update of fpga-based secure elements using partial reconfiguration. *Cryptology ePrint Archive* (2020).

[65] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19).* USENIX Association, Santa Clara, CA, 1411–1428. https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall

[66] Paolo Visconti, Ramiro Velazquez, Stefano Capoccia, and Roberto de Fazio. 2021. High-performance AES-128 algorithm implementation by FPGA-based SoC for 5G communications. *International Journal of Electrical & Computer Engineering (2088-8708)* 11, 5 (2021).

[67] P Visconti, R Velazquez, Carolina Del-Valle Soto, and R de Fazio. 2021. FPGA based technical solutions for high throughput data processing and encryption for 5G communication: A review. *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 19, 4 (2021), 1291–1306.

[68] Hoang-Gia Vu, Shinya Takamaeda-Yamazaki, Takashi Nakada, and Yasuhiko Nakashima. 2018. A Tree-Based Checkpointing Architecture for the Dependability of FPGA Computing. *IEICE Transactions on Information and systems* 101, 2 (2018), 288–302.

[69] Wikipedia. 2023. Internet Mix. https://en.wikipedia.org/wiki/Internet_Mix.

[70] Hongjun Wu and Bart Preneel. 2013. AEGIS: A fast authenticated encryption algorithm. In *International Conference on Selected Areas in Cryptography.* Springer, 185–201.

[71] Xilinx. 2025. Vivado Design Tools Archive. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html. Accessed: 2025-06-05.

[72] Yatao Yang, Wenchen Zhao, Liangqing Xiong, Ning Wang, and Yingjie Ma. 2021. Optimized implementations for ZUC-256 on FPGA. *Wireless Personal Communications* 116, 3 (2021), 2615–2632.

[73] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. 2023. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23).* Association for

Computing Machinery, New York, NY, USA, 208–219. doi:10.1145/3603269.3604862

[74] Sophia Yoo and Xiaoqi Chen. 2021. Secure Keyed Hashing on Programmable Switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure* (Virtual Event, USA) *(SPIN '21)*. Association for Computing Machinery, New York, NY, USA, 16–22. doi:10.1145/3472873.3472881

[75] Yutaro Yoshinaka, Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. 2022. On Implementing ChaCha on a Programmable Switch. In *Proceedings of the 5th International Workshop on P4 in Europe* (<conf-loc>, <city>Rome</city>, <country>Italy</country>, </conf-loc>) *(EuroP4 '22)*. Association for Computing Machinery, New York, NY, USA, 15–18. doi:10.1145/3565475.3569073

[76] Amir Zeineddini and Jim Wesselkamper. 2011. PRC/EPRC: Data integrity and security controller for partial reconfiguration. *XAPP887, January* 12 (2011), 17.

[77] Chuwen Zhang, Zhikang Chen, Haoyu Song, Ruyi Yao, Yang Xu, Yi Wang, Ji Miao, and Bin Liu. 2021. PIPO: Efficient Programmable Scheduling for Time Sensitive Networking. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. 1–11. doi:10.1109/ICNP52444.2021.9651944

[78] Xin Zhang, Zhi Zhang, Qingni Shen, Wenhao Wang, Yansong Gao, Zhuoxi Yang, and Zhonghai Wu. 2024. ThermalScope: A Practical Interrupt Side Channel Attack Based on Thermal Event Interrupts. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 28, 6 pages. doi:10.1145/3649329.3656525

[79] Gang Zhou, Harald Michalik, and László Hinsenkamp. 2009. Improving throughput of AES-GCM with pipelined Karatsuba multipliers on FPGAs. In *Reconfigurable Computing: Architectures, Tools and Applications: 5th International Workshop, ARC 2009, Karlsruhe, Germany, March 16-18, 2009. Proceedings 5*. Springer, 193–203.

[80] Shaguftha Zuveria Kottur, Krishna Kadiyala, Praveen Tammana, and Rinku Shah. 2022. Implementing ChaCha Based Crypto Primitives on Programmable SmartNICs. In *Workshop on Formal Foundations and Security of Programmable network INfrastructure (FFSPIN'22)* (Amsterdam, Netherlands) *(FFSPIN'22)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3528082.3544833

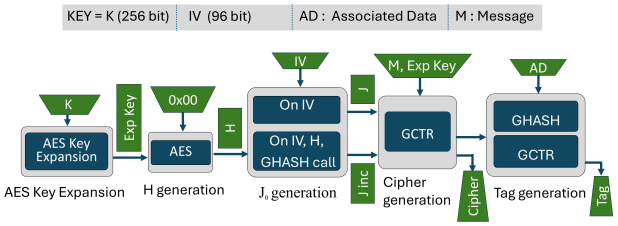


Fig. 17. Overview of AES-GCM algorithm.

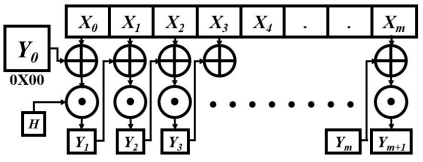


Fig. 18. GHASH function of AES-GCM.

A Overview of AES-GCM algorithm

AES-GCM (Galios/Counter Mode) [52] algorithm provides both confidentiality and integrity of control and user-plane traffic in 5G networks. It uses the AES in Counter mode (GCTR) for encryption and GHASH, a Galois field-based hash function for generating the authentication tag. Fig. 17 shows the main components of AES-GCM. The AES key expansion function is used to generate the expanded AES round keys once per encryption key, and this expanded key can be reused for all packets within a given flow.

AES-GCM phases. AES-GCM involves *four* main phases. (1) *Hash subkey generation.* A subkey, H, is initialized with the AES cipher generated by encrypting a 128-bit block of *zeros* with the secret key, K. (2) *Generation of block counter,* J_0. This function generates the initial block counter using the Initialization Vector (IV). (3) *Encryption.* The initial counter, J_0, is incremented and encrypted using AES with the secret key, K. The encrypted counters are then XORed with the plaintext block of 128-bits to generate the cipher. (4) *Tag generation.* The cipher text (C) and the associated data (AD), along with their lengths, and J_0 are sent to the GHASH function to generate the 128-bit tag.

The GHASH function. Fig. 18 shows the working of the GHASH function. Given a hash subkey $H = \text{AES}_K(0^{128})$ and a sequence of 128-bit input blocks $X_1, X_2, \ldots, X_m$, GHASH initializes an accumulator $Y_0 = 0^{128}$ and iteratively computes $Y_i = (Y_{i-1} \oplus X_i) \cdot H$ for each block. The final value Y_m is the GHASH output.

Bottleneck. The top-2 cycle-consuming functions are GCTR and GHASH. The encryption function, GCTR, is highly parallelizable since each plaintext block is XORed with independently encrypted counter blocks. In contrast, GHASH evaluates a polynomial over GF(2^{128}) using a chained recurrence relation $Y_i = (Y_{i-1} \oplus X_i) \cdot H$, meaning each computation depends on the previous one. This makes GHASH inherently sequential. Table 5 shows the cycle breakdown of our baseline AES-GCM implementation ($S_{AES-GCM}$) for the two main functions, GHASH and GCTR, and indicates GHASH as the bottleneck. We observe that GHASH and GCTR consume 5027 and 1531 cycles, respectively, for 500 blocks. We discuss the optimizations to reduce the cycles for the GHASH function in §4.3.

B *FlexMesh* scheduler: A toy example.

Fig. 19 shows the working of *FlexMesh* $_{nWC}$, *FlexMesh*, and $FlexMesh_p$ scheduling policies with the help of a toy example. The example comprises requests from two flows, viz., $flow_1$ and $flow_2$. $r_{i,j}$ represents j^{th} request from $flow_i$, and service time represents the time required to process (*i.e.*, encrypt) the request. Assume two cryptographic cores C_i with one work queue WQ_i each. The C_i list shows the request executed on the individual cores, and the *wire output* list shows the order in which the requests are sent out of the NIC after processing.

We show the output of three scheduling policies, *FlexMesh* $_{nWC}$, *FlexMesh*, and $FlexMesh_p$. All the policies implement the threshold-based classifier for load balancing. *FlexMesh* $_{nWC}$ implements FIFO scheduling policy (Fig. 19 c). We observe that C_2 is idle even when there are pending requests, and the completion time is 7 time units. *FlexMesh* implements FIFO scheduling with work-conservation,

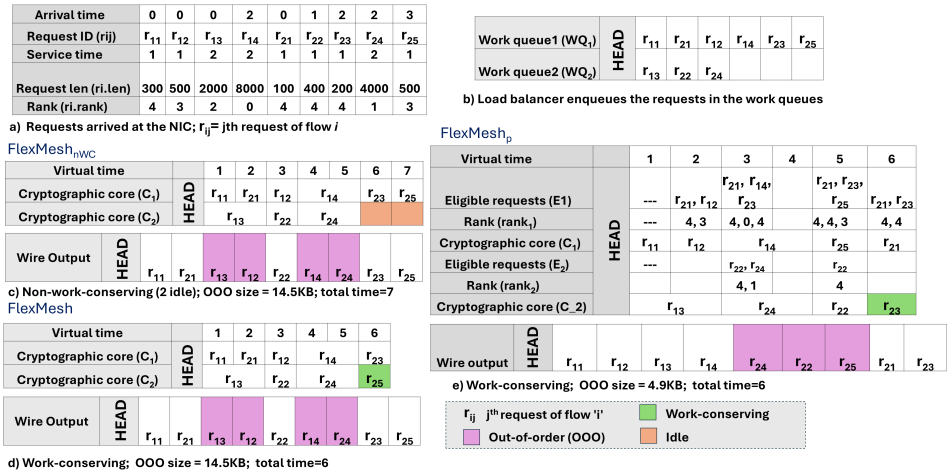


Fig. 19. Scheduling across asymmetric cryptographic cores (*FlexMesh* $_{nWC}$ vs. *FlexMesh* vs. *FlexMesh*$_p$).

ie, if WQ_i is empty, the scheduler steals work from other work queues Fig. 19 d). We observe that the completion time decreases (*i.e.*, 6 time units). Both policies require 14.5KB buffer for OOO request storage. *FlexMesh*$_p$ implements the PIEO primitive with the goal to minimize OOO request memory, work-conservation, and starvation avoidance (Fig. 19 e). We observe that the OOO memory reduces to 4.9KB and the completion time is 6 time units. However, request-size-based prioritization comes with the risk of increased request latency and starvation.

C Analytical model for workload-aware cryptographic core deployment decision

The objective of the analytical model running at the orchestrator is to monitor the current load and request size distribution, and identify the set of feasible cryptographic core designs that satisfy the throughput, latency, and resource efficiency requirements with the configured deviation tolerance.

Inputs and Parameters

$D = \{S, M, L, SS, MM, LL, SL, ML\}$	Set of available design (cryptographic core) variants,
$\mathcal{B} = \{b_1, b_2, \ldots, b_K\}$	Bins for request size ranges,
t	Time interval for request size distribution measurement,
$N_{b,t}$	Number of requests of bin b observed during interval t,
S_b	Request size (bits) for bin b,
$T_{d,b}$	Throughput of design d for bin b (capacity in bits/sec),
$L_{d,b}$	Processing latency of design d for bin b,
$E_{d,b}$	Resource efficiency of design d for bin b,
P_d	Power consumption (in Watt) of design d,
ϵ_L, ϵ_E, ϵ_T	Tolerance for latency, efficiency, and throughput ($0 \le \epsilon \le 1$),
γ	Allowed load overflow ratio per core instance ($\gamma > 0$),
λ	Total observed incoming load across cryptographic cores (bps).

Offered Load Computation (Time-Aggregated)

The time-averaged observed load per bin (bits/sec): $\bar{\lambda}_b = \frac{N_{b,t}\, S_b}{t}$

Total observed load across all packet sizes (bits/sec) during the interval t: $\lambda_{tot} = \frac{1}{t} \sum_{b \in \mathcal{B}} N_{b,t}\, S_b$

Expected Design Metrics (Weighted by Offered Load)

Table 13. Comparison of in-network scheduling solutions.

Scheduling technique	Aging support?	Flow count scalability	Single instance performance	Logical partitioning support?
PIFO [59]	No	$4K$ flows	$15Mpps$ ($10Gbps$)	No
PIEO [58]	Yes	$64K$ flows	$15Mpps$ ($10Gbps$)	No
PIPO [77]	Yes	NA flows	$70Mpps$ ($40Gbps$)	No
BMW-tree [73]	No	$100K$ flows	$150Mpps$ ($100Gbps$)	16 to 128 tenants
BBQ [14]	No	$100K+$ flows	$150Mpps$ ($100Gbps$)	16 to 18 tenants
ClubHeap [22]	No	$100K+$ flows	$200Mpps$	256 tenants

$$\text{Expected latency, } \bar{L}_d = \frac{1}{\sum_{b\in\mathcal{B}} N_{b,t}} \sum_{b\in\mathcal{B}} N_{b,t}\, L_{d,b} \tag{3}$$

$$\text{Expected efficiency, } \bar{E}_d = \frac{1}{\sum_{b\in\mathcal{B}} N_{b,t}} \sum_{b\in\mathcal{B}} N_{b,t}\, E_{d,b} \tag{4}$$

Feasibility Filtering. Let the best achievable latency and efficiency be:

$$L_{\min} = \min_{d\in D} \bar{L}_d, \qquad E_{\max} = \max_{d\in D} \bar{E}_d$$

Then, the feasible set of designs:

$$D^{\text{feas}} = \left\{ d \in D : \bar{L}_d \le (1+\epsilon_L) L_{\min},\ \bar{E}_d \ge (1-\epsilon_E) E_{\max} \right\}$$

Optimization Objective. If multiple feasible designs are close in latency and efficiency, the algorithm explores hybrid cryptographic core combinations, $\mathbf{n} = \{n_S, n_M, n_L, \dots\}$. Among the feasible hybrid cryptographic core combinations, choose the combination with the optimization objective:

Minimize the total power (static and dynamic):

$$\min_{\{n_d\}} \quad P_{\text{total}} = \sum_{d\in D^{\text{feas}}} n_d\, P_d$$

subject to:

$$\sum_{d\in D^{\text{feas}}} n_d\, T_{d,b} \ge (1-\epsilon_T)\bar{\lambda}_b, \qquad \forall b \in \mathcal{B},$$

$$n_d \in \mathbb{Z}_{\ge 0}, \qquad \forall d \in D^{\text{feas}}.$$

Resulting Optimal Configuration

$$\mathbf{n}^* = \{n_d^* : d \in D^{\text{feas}}\}$$

represents the number of active instances per design. That is, first, the designs with the minimum latency values (and those within the latency tolerance limits) are identified. Among these, designs with minimum resource efficiency values (and close designs within efficiency tolerance limits) are filtered. Finally, the design with the minimum power requirement is chosen.)

D Comparing state-of-the-art scheduler hardware

Table 13 compares existing hardware schedulers. *FlexMesh* scheduler utilizes the design of PIEO [58] and PIPO [77] to support the programmable predicate for request filtering and aging.

Received June 2025; revised November 2025; accepted December 2025