SNO: Securing Network Function Offloading on FPGA-based SmartNICs in Untrusted Clouds

Yunkun Liao[1,2,3], Jingya Wu[1], Wenyan Lu[1,4], Hang Lu[1,3(✉)], Xiaowei Li[1,3(✉)], Guihai Yan[1,4(✉)]
{liaoyunkun20s, wujingya, luwenyan, luhang, lxw, yan}@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences[1]
University of Chinese Academy of Sciences[2], *Zhongguancun Laboratory*[3], *YUSUR Technology Co., Ltd.*[4]
Beijing, China

***Abstract*—As network bandwidth outpaces host CPU compute capability, Smart Network Interface Cards (SmartNICs) are increasingly deployed to offload network functions from the host CPU. FPGA-based SmartNICs excel due to their programmability at hardware speed, enabling high-performance and customized offloading. Securing offloaded network functions on FPGA-based SmartNICs is a critical challenge in the cloud, as the sensitive user cannot fully trust the cloud service provider (CSP). CPU Trusted Execution Environments (TEEs) protect software code, not FPGA hardware circuits. Existing FPGA TEEs fail to provide packet I/O protection, System-on-Chip (SoC) CPU utilization, and user-friendly memory access interfaces. To address this gap, we introduce SNO, the first TEE for FPGA-based SmartNICs with the secure boot, the SNO Manager for attestation and network function lifecycle orchestration, and the SNO Guard for I/O encryption and authentication. SNO increases SoC CPU utilization (+6.6% for 8-CPU SoC) by co-locating the SNO Manager with the CSP software while isolating the security-critical components of SNO Manager inside SoC CPU TEE, reduces performance overhead by integrating a fully-pipelined AES-GCM engine and overlapped execution, and offers a user-friendly (86.9% user code reduction) streaming interface. The experimental results show that SNO introduces a relative latency overhead of 7.7–143.2% (corresponding to absolute overheads up to 96 nanoseconds) across five network functions, significantly offset by microsecond-level latency savings from offloading.**

I. Introduction

In modern cloud infrastructure, the stagnation of host CPU performance increasingly lags behind the rapid growth of datacenter network bandwidth. To bridge this gap, host applications involving packet processing (i.e., network functions) are increasingly offloaded to the Smart Network Interface Card (SmartNIC). SmartNICs augment traditional NICs with programmable compute units. Among them, FPGA-based SmartNICs [1], [2] are widely adopted for their hardware-level programmability, enabling high-performance, customized offloading. A growing trend [3] is to empower cloud users to deploy their network functions directly on FPGA-based SmartNICs.

Offloading network functions to FPGA-based SmartNICs introduces critical security challenges for cloud users. These stem from the inherent trust asymmetry between users and the cloud service provider (CSP) [4], who controls both the privileged software stack and the hardware (i.e., SmartNIC's FPGA shell). Private users must ensure their data is not exposed to the CSP. However, the shell has visibility into all I/O of user-deployed network functions, making sensitive data vulnerable to inspection or tampering by a malicious or curious CSP. This threat model is realistic, as insider-initiated data breaches remain a well-documented risk in public cloud environments [5].

In a threat model where the user distrusts the CSP, a Trusted Execution Environment (TEE) offers a hardware-isolated secure zone

Hang Lu, Xiaowei Li, and Guihai Yan are corresponding authors. This paper is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. (62090024, 92373206, 62090020, 62172387), and in part by the Chinese Academy of Sciences under grant No.(XDB0660100, XDB44030100), and in part by the internship program and KAIWU (Data Network Development Platform, DNDP) of YUSUR Technology Co., Ltd.

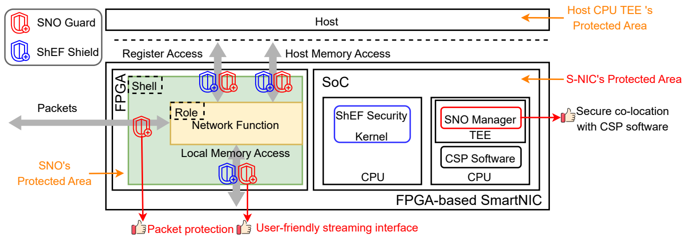


Fig. 1: SNO is the TEE for FPGA-based SmartNICs. S-NIC [6] and ShEF [8] are the state-of-the-art SmartNIC and FPGA TEEs.

that protects sensitive computations from privileged software and physical attacks while preserving performance. However, existing TEEs fall short for FPGA-based SmartNICs. First, the host CPU TEE cannot secure network functions executing on SmartNICs, as these functions execute outside the TEE's security boundary. Second, the currently available SmartNIC TEE named S-NIC [6] is designed for software-based network functions running on the SoC CPU, making it incompatible with hardware-based network functions running in the FPGA. Third, FPGA TEEs [7], [8]—such as the state-of-the-art ShEF [8]—target traditional CPU-FPGA acceleration and face key limitations in context of SmartNIC: (1) ShEF Shield lacks packet I/O protection, leaving a critical attack surface exposed; (2) ShEF Security Kernel requires a dedicated SoC CPU, reducing CPU resource for CSP software; and (3) ShEF Shield presents a complex low-level memory-mapped interface for memory access, impeding usability and adoption.

To bridge the research gap in this area, we introduce SNO, the first TEE for FPGA-based SmartNICs, as illustrated in Fig. 1. SNO targets emerging heterogeneous SmartNIC architectures combining FPGA (partitioned into shell and role regions) and SoC. To ensure security, SNO implements three key components: (1) the secure boot verifying the SmartNIC image, (2) the SNO Manager managing attestation and network function lifecycles, and (3) the SNO Guard encrypting and authenticating packet, memory, and register I/O between the user's network function and the shell.

In addition to securing network packets, SNO offers three key advantages: (1) It improves SoC CPU utilization by co-locating the SNO Manager with CSP software on an SoC CPU while isolating its security-critical components within the SoC CPU TEE to mitigate co-location threats. (2) SNO Guard offers a user-friendly streaming interface for memory access at the frontend, abstracting the complexity of controlling the memory-mapped interface at the backend. (3) SNO Guard significantly reduces the performance overhead of authenticated encryption by integrating a fully-pipelined AES-GCM engine that runs concurrently with the network function.

This paper makes the following key contributions:

- We identify research gaps in existing security frameworks for protecting network functions on FPGA-based SmartNICs.
- We propose SNO, the first TEE for FPGA-based SmartNICs,

comprising secure boot, SNO Manager, and SNO Guard. SNO protects the confidentiality and integrity of user network function.

- We quantitatively evaluate SNO using an FPGA prototype and RTL simulation. Leveraging the SoC CPU TEE for the SNO Manager incurs negligible overheads (0.58 seconds for boot and 40 milliseconds for bitstream loading). SoC CPU utilization is increased by 6.6% on an 8-CPU SoC. SNO Guard introduces a relative latency overhead of 7.7–143.2% across five network functions, with the absolute overhead under 96 nanoseconds—orders of magnitude smaller than the latency savings from offloading (several microseconds). SNO Guard streaming interface reduces the user memory access code by 86.9% compared to ShEF.

II. Background and Motivation

A. SmartNIC

Unlike traditional NICs, which solely transfer packets between the host network stack and the network, SmartNICs incorporate onboard compute and memory resources to enable custom packet processing. A key use case of SmartNIC is offloading packet-driven network functions from the host. This offloading offers two main benefits: (1) reduced latency by avoiding packet transfers to the host and enabling optimized in-place processing, and (2) freed host CPU cycles by shifting network stack and application processing to the SmartNIC.

B. FPGA-based SmartNIC

FPGA-based SmartNICs leverage FPGAs as computation units for packet processing. FPGA offers configurable logic blocks, memory blocks, and interconnects, which can be used to implement custom packet processing logic. Packet processing on FPGA delivers orders-of-magnitude speedup over software-based solutions while maintaining programmability, making them widely adopted.

Figure 2 illustrates the typical FPGA-based SmartNIC architecture adopted in cloud environments. This architecture consists of four key components: hardened logic blocks, a system-on-chip (SoC), FPGA, and off-chip memory. Hardened logic blocks are pre-designed, non-programmable units optimized for high-speed I/O, such as PCIe and Ethernet. The SoC integrates general-purpose CPUs (e.g., ARM) and a full memory hierarchy. While earlier FPGA-based SmartNICs did not include an SoC, modern designs (e.g., Intel C5000X-PL [9], AMD SN1000 [10], Alibaba Cloud SmartNIC [11]) increasingly incorporate the SoC for offloading CSP's infrastructure software. Including the SoC in our FPGA-based SmartNIC model aligns with current technological trends. The FPGA follows the shell-role partition [12], where the shell is managed by the CSP and the role by the user. Both the SoC and FPGA interface with independent off-chip memory.

The shell of an FPGA-based SmartNIC serves as the infrastructure layer for the role and comprises the following components:

- The packet management subsystem handles low-level network protocols (e.g., physical coding, media access control) on both transmit (TX) and receive (RX) sides, converting packets between wire form and user-side streaming signals. On the RX side, this subsystem matches packet headers against user-defined rules, forwarding unmatched packets to the host software network stack while scheduling matched packets for network functions.
- The PCIe direct memory access (DMA) subsystem facilitates data transfer between the host and FPGA. The host software accesses FPGA memory regions via Memory-Mapped I/O (MMIO), while the FPGA initiates DMA to read/write host memory.
- The memory subsystem provides a user interface for accessing local off-chip memory.
- The interconnect exchanges data among all functional blocks.

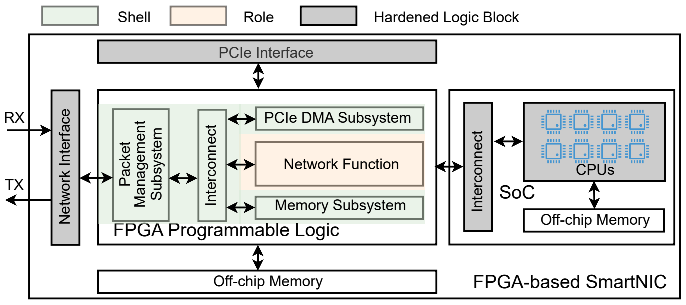


Fig. 2: Typical FPGA-based SmartNIC architecture used in the cloud.

Leveraging FPGA partial reconfiguration [13], the shell is deployed as static logic, remaining unchanged during operation, while the role is implemented as dynamic logic, allowing reconfiguration of its bitstream without disrupting the shell.

C. Research Gaps

When users perform remote computation on CSP-provided infrastructure, data privacy and security are critical concerns. Threats can originate from both malicious insiders such as system administrators and external attackers. In an untrusted cloud, users often trust the hardware vendor over the CSP, as vendors are economically motivated to protect their reputation. Hardware vendors provide TEEs, such as Intel SGX Enclaves [4], protecting sensitive code and data using hardware-isolated zones that prevent unauthorized access and tampering.

Research Gap in CPU TEEs: Host CPU TEEs protect applications from untrusted CSP-controlled privileged software such as the hypervisor. However, network functions offloaded to FPGA-based SmartNICs run outside the CPU TEE's security boundary. Moreover, CPU TEEs are designed to protect software, whereas FPGA-based network functions are implemented as hardware circuits.

Research Gap in Prior SmartNIC TEEs: To the best of our knowledge, S-NIC [6] is the only TEE designed for SmartNICs. However, it targets scenarios where network functions are offloaded as software to the SoC CPU. S-NIC relies on CPU-enforced memory access control to prevent CSP software from observing or tampering with the network function. In contrast, FPGA-based network functions are implemented as hardware circuits directly interfacing with the CSP-controlled shell, leaving all I/O exposed.

Research Gap in Native FPGA Security Enhancements: FPGA vendors have proposed several security enhancements in their products [14], including key storage (e.g., eFuse), cryptographic accelerators, image/bitstream encryption and authentication, and secure boot. However, these enhancements assume that the FPGA is exclusively owned by the user. In contrast, in the cloud, the CSP-controlled shell co-locates with the user's network function and monitors the network function's I/O. While architecting a TEE based on native FPGA security enhancements is desirable, additional design considerations are needed for the FPGA-based SmartNIC.

Research Gap in Prior FPGA TEEs: FPGA TEEs for traditional CPU-FPGA computing offload have recently emerged, with notable solutions including SGX-FPGA [7] and ShEF [8]. SGX-FPGA, the first FPGA TEE, aims to protect user data on the host CPU, FPGA, and the transmission path between them. On the FPGA side, SGX-FPGA introduces a Security Monitor, trusted by the user, to manage and encrypt memory I/O for sensitive computing kernels. However, SGX-FPGA's Security Monitor is incompatible with the shell-role architecture used in the cloud, as the Security Monitor overrides the shell's I/O monitoring capability. Additionally, SGX-FPGA fails to address how the Security Monitor's trustworthiness is ensured, given the Security Monitor is deployed by an unknown entity. ShEF,

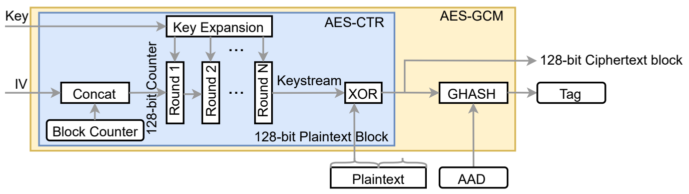


Fig. 3: AES-GCM workflow.

the state-of-the-art FPGA TEE, is compatible with the shell-role architecture. Users connect their accelerators to the ShEF Shield module, which authenticates and encrypts all I/O. However, ShEF has several limitations: (1) it lacks packet I/O protection, the most critical requirement for FPGA-based SmartNICs; (2) it places its Security Kernel on a dedicated SoC CPU, isolating it from CSP software but wasting the SoC CPU; and (3) its memory-mapped interface is complex for users to interact with.

D. Understanding AES-GCM

AES-GCM [15] is a widely adopted authenticated encryption algorithm that protects both confidentiality and integrity in a single processing step, as illustrated in Fig. 3. The design combines counter-mode AES (AES-CTR) for encryption and Galois HASH (GHASH) for authentication. ***Encryption***: A 128-bit counter (96-bit initialization vector (IV) + 32-bit block counter) is encrypted via AES substitution-permutation network (10/12/14 rounds for AES-128/192/256-bit key, respectively), using round keys derived from the secret key. The resulting keystream is XORed with a 128-bit plaintext block to produce a 128-bit ciphertext block. ***Authentication***: GHASH processes ciphertext blocks, additional authenticated data (AAD), and their lengths. Each block is multiplied by a hash subkey (from AES-encrypting a zero block) and accumulated iteratively. The final tag is produced by encrypting the GHASH output with the initial counter. Decryption regenerates the keystream to recover plaintext and independently recomputes GHASH for tag verification. A mismatch invalidates the payload, ensuring tamper detection.

III. Threat Model

Since no standard business model exists for FPGA-based SmartNICs in the cloud, we define our own as follows. The CSP purchases host machines and SmartNICs from hardware vendors and offers them for rental. The CSP installs its software and FPGA shell on the SmartNIC's SoC and FPGA. The user rents a virtual machine instance with SmartNIC acceleration and deploys network functions on the FPGA. These functions are composed of the user's in-house logic and intellectual property (IP) cores from third-party vendors.

In our threat model, the CSP is considered an honest-but-curious adversary [16]. It follows pre-negotiated protocols, such as resource allocation, but may attempt to infer sensitive information from the user's network function. As such, all privileged software and hardware under CSP control are untrusted, including the host hypervisor and the FPGA shell. For example, the CSP can monitor all I/O through the shell, enabling inspection, recording, or tampering with unprotected data. Physical attacks on off-chip components—such as host memory and PCIe links—are also possible due to malicious CSP insiders [17].

We do not consider malicious co-located tenants in our threat model, as commercial cloud FPGA platforms currently adopt a single-tenant model [8]. However, SNO inherently mitigates future inter-tenant attacks: all I/O from a sensitive network function is protected by SNO Guard's authenticated encryption, and the secrets are known only to the owning user.

We assume that the SmartNIC vendor, user, and IP vendor are trusted. Therefore, the SmartNIC and IP vendor do not intentionally

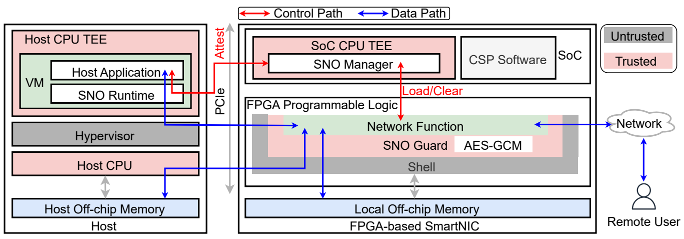


Fig. 4: High-level system architecture of SNO.

implant attack surfaces like the hardware Trojan in the SmartNICs or IP cores [18]. We also assume there is a host CPU TEE to protect the user's host application. The host application takes charge of offloading the network function to the SmartNIC, such as performing attestation, sending the FPGA bitstream to the SmartNIC, and configuring the network function. We assume the user synthesizes the network function bitstream in a secure environment after the user has obtained necessary design sources, including the shell interfaces, SNO Guard source code, IP cores, and FPGA layout constraints. The secure environment can be a private server owned exclusively by the user.

We do not address side-channel or covert-channel attacks, including those targeting SmartNIC microarchitecture [6] and FPGA power consumption [19]. Additionally, denial-of-service attacks are beyond the scope of this work. However, given the reconfigurable nature of FPGA, defenses against such attacks can be incorporated in the future.

IV. SNO Design

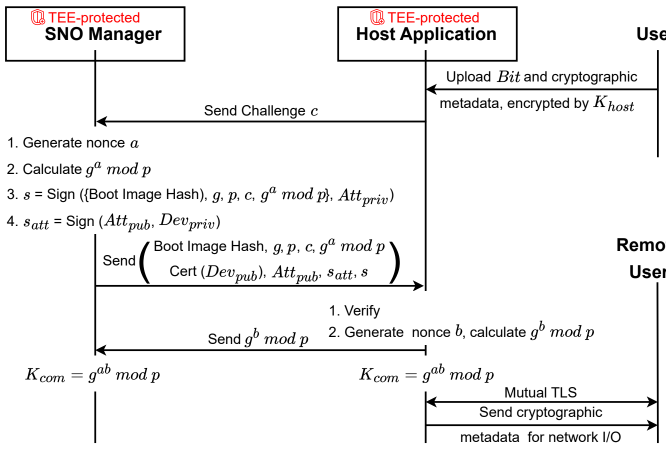
A. Overview

SNO introduces a TEE for FPGA-based SmartNICs to safeguard user-deployed network functions. Figure 4 illustrates the system architecture. SNO spans both the host and the SmartNIC. On the host, it assumes the user's application is protected by existing CPU TEEs at the virtual machine level, such as Intel TDX [20] and AMD SEV [21]. On the SmartNIC, the SNO Manager and SNO Guard protect the network function from CSP software and hardware in both the control and data paths. The SNO Manager runs inside the SoC CPU TEE and secures critical operations such as attestation, bitstream loading, and clearing. The SNO Guard applies AES GCM to encrypt and authenticate all I/O between the user's network function and the CSP-controlled shell.

B. Control Path: Secure Boot, Attestation and SNO Manager

1) Secure Boot: SNO proposes a secure boot to ensure the FPGA-based SmartNIC only accepts the verified boot image and loads the SNO Manager to the desired state. During manufacturing, the vendor installs an AES device key (K_{dev}) into the FPGA's eFuse as the Root of Trust. When the user instructs the CSP to launch the FPGA-based SmartNIC, the two-stage secure boot sequence begins.

In the first stage, dedicated secure-boot modules (e.g., Platform Management Unit (PMU) and Configuration and Security Unit (CSU) of AMD UltraScale+ MPSoC [14]) execute fixed-state machines or immutable ROM code, with cryptographic integrity verified by each module before transitioning to the next. The boot image, provisioned by the SmartNIC vendor, includes bootloaders, firmware for the TEE (e.g., ARM Trusted Firmware), and an operating system with the TEE and SNO Manager enabled. While the CSP can deploy its software code and shell bitstream on the SmartNIC, the CSP cannot alter the boot image. In the second stage, each part of the boot image is authenticated and decrypted before loading. SNO secure boot uses the rolling key features to minimize the risk of K_{dev} exposure: The part i of the boot image contains an pair of encrypted AES-GCM operational key ($K_{op,i}$) and IV (IV_i), which is encrypted and authenticated by the



TEE-protected
SNO Manager
TEE-protected
Host Application
User
Upload Bit and cryptographic metadata, encrypted by K_{host}
Send Challenge c
1. Generate nonce a
2. Calculate g^a mod p
3. s = Sign ({Boot Image Hash}, g, p, c, g^a mod p}, Att_{priv})
4. s_{att} = Sign (Att_{pub}, Dev_{priv})
Send (Boot Image Hash, g, p, c, g^a mod p Cert (Dev_{pub}), Att_{pub}, s_{att}, s)
Remote User
1. Verify
2. Generate nonce b, calculate g^b mod p
Send g^b mod p
$K_{com} = g^{ab}$ mod p
$K_{com} = g^{ab}$ mod p
Mutual TLS
Send cryptographic metadata for network I/O

Fig. 5: SNO attestation procedure.

$K_{op,i-1}$ and IV_{i-1}. The first part of the boot image is decrypted by the K_{dev}, accomplished by the last secure-boot modules.

2) TEE-protected SNO Manager: The SNO Manager is a daemon process running in the SoC CPU, responsible for attestation, loading the CSP's shell and the user's network function bitstream onto the FPGA, and clearing the network function bitstream upon the user's request. To enable full-lifecycle SmartNIC attestation, an asymmetric key pair (Dev_{pub}/Dev_{priv}) is embedded in the SNO Manager binary during boot image provisioning. To reduce the risk of key exposure during runtime attestation, the SNO Manager generates a fresh ephemeral key pair (Att_{pub}/Att_{priv}) after secure boot.

The SNO Manager must protect security-critical components from untrusted CSP software at runtime. These include attestation metadata, the plaintext network function bitstream, and the operations that process this data. ShEF [8] addresses this by dedicating an FPGA SoC CPU to its Security Kernel and storing sensitive data in secure on-chip memory. However, this design has two key drawbacks: (1) dedicating a CPU incurs significant resource cost, and (2) secure on-chip memory is severely limited (e.g., 256KB on AMD UltraScale+ MPSoCs [22]). In contrast, SNO adopts a cost-effective and scalable solution by leveraging the SoC CPU TEE available in modern FPGA-based SmartNICs. For instance, the AMD SN1000 [10] includes sixteen TrustZone-enabled ARM CPUs, enabling the secure execution of sensitive code and data. Prior work [23], [24] has demonstrated that TrustZone effectively secures bitstream loading. Therefore, SNO places its security-critical functions inside a TrustZone-compatible TEE such as OP-TEE [25], allowing CSP software to co-locate on the same CPU in the non-secure world. OP-TEE uses TrustZone-enforced isolation to protect sensitive data and operations, supporting secure memory regions of several megabytes [26].

3) Attestation: To establish trust between the user's host application and the SNO Manager, SNO employs an attestation procedure based on standard cryptographic primitives (public key infrastructure and Diffie-Hellman Key Exchange [27]) executed within the TEE-protected SNO Manager. At the same time, sensitive data and secrets are synchronized among the user, host application, SNO Manager, and the remote user. Figure 5 shows the attestation procedures: ① ***User Uploads Encrypted Bitstream and Cryptographic Metadata***: The user uploads the encrypted network function bitstream (Bit) and SNO Guard cryptographic metadata (including AES-GCM keys and initial IVs of the Register, Packet, DMA, and Local Memory Guard) to the host application via a CSP-provided channel. The files are encrypted with K_{host}, as the CSP-provided channel is untrusted, where K_{host} is the secret key owned by the user, fused into the host application binary (e.g., via user configuration or secure injection), and protected by the host CPU TEE. The SNO Guard cryptographic metadata, fused into Bit during bitstream synthesis, enables secure data exchange between the host application, remote user, and the network function. ② ***Host Application Sends Challenge***: The host application invokes the SNO runtime to send a challenge c to the SNO Manager to initiate attestation. ③ ***SNO Manager Generates Attestation Response***: Upon receiving c, the SNO Manager generates a nonce a, computes g^a mod p, and signs the combination of the boot image hash, g, p, c, and g^a mod p with Att_{pub} to produce signature s. It then sends the boot image hash, g, p, c, g^a mod p, the certificate of Dev_{pub}, Att_{pub}, s_{att}, and s to the host application. ④ ***Host Application Verifies Attestation***: The host application verifies that the certificate of Dev_{pub} is signed by a trusted SmartNIC vendor. It then uses Dev_{pub} to verify s_{att} and Att_{pub} to verify s. ⑤ ***Symmetric Communication Key Establishes***: The host application generates a nonce b, computes g^b mod p, and sends it to the SNO Manager. Both parties derive a symmetric communication key $K_{com} = g^{ab}$ mod p, which is used to exchange data (e.g., Bit) securely. ⑥ ***Remote User Setup and Metadata Exchange***: The remote user establishes a TLS session with the host application and acquires cryptographic metadata of the Packet Guard to exchange packets with the network function.

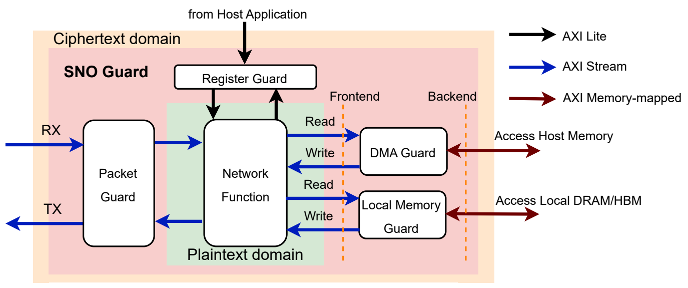


Fig. 6: SNO Guard provides streaming interfaces to the user's network function kernel, enabling secure decryption of data while ensuring integrity protection.

C. Data Path: SNO Guard

SNO Guard ensures the confidentiality and integrity of I/O exchanged between a protected network function and an untrusted shell that monitors this I/O. Implemented as a hardware module interposing on the path between network function and shell, SNO Guard authenticates and encrypts all egress I/O leaving the network function. Conversely, it decrypts ingress ciphertext I/O before forwarding plaintext to the NF. This ensures the NF operates solely on plaintext data. The attestation protocol (§IV-B3) protects the cryptographic keys used by SNO Guard from disclosure to untrusted entities.

SNO Guard secures all I/O channels associated with the network function, including host-side register access, host and local memory accesses, and network packet I/O. Unlike prior work such as ShEF Shield [8], SNO Guard extends protection to network packet I/O. This enhancement is essential because offloaded network functions fundamentally operate on packets.

SNO Guard enables efficient and user-friendly bulk data handling through streaming interfaces and overlapped execution. Streaming interfaces for memory access and packet I/O simplify integration by abstracting the bus interconnect complexity found in memory-mapped designs like ShEF Shield. SNO Guard overlaps cryptographic processing with network function processing to preserve high performance, effectively exploiting packet-level parallelism.

1) User-friendly Streaming Interface: SNO Guard provides the network function with streaming interfaces for three types of data-intensive I/O, as shown in Fig. 6. ① ***Network packet reception and transmission***: In FPGA-based SmartNICs [28], [29], network packets are continuous byte streams carried via streaming interfaces (e.g., AXI Stream). Packet Guard retains this feature, enabling users to handle network I/O efficiently. ② ***Host memory access***: Network functions access shared host memory to exchange data with host applications. DMA Guard provides a frontend streaming interface with six channels (three for write/read, including address handshake, data, and response), simplifying integration for the user by requiring only valid-ready handshake and data transfer handling. Unlike complex AXI memory-mapped interfaces [30], [31] adopted by ShEF [8], which involve burst control, outstanding control, address alignment, and signal dependency management, DMA Guard removes these complexities in the frontend interface. On the backend, DMA Guard manages PCIe communication, supporting AXI memory-mapped interfaces for PCIe bridges or vendor-specific IPs like AMD QDMA [32]. ③ ***Local memory access***: Similar to DMA Guard, the Local Memory Guard provides a frontend streaming interface with a backend that communicates with the shell's local memory interface (e.g., AXI memory-mapped).

2) Secure Register Access: Register Guard maintains an industry-standard register access interface (e.g., AXI Lite) for both the network function and the untrusted shell. To ensure integrity, the Register Guard employs a Message Authentication Code (MAC) to protect ciphertext register values. To preserve the register space layout, Register Guard reuses the register address for storing the MAC, requiring the host application to follow a specific write sequence: first writing the encrypted register value, then writing the MAC (partitioned to adapt with the register width if necessary). Once the complete MAC is received, Register Guard verifies the integrity of the encrypted value and, upon successful authentication, writes the decrypted value to the network function's register space. For reads, the host application issues multiple read operations: the first retrieves the encrypted value, while subsequent reads fetch the MAC. During read handling, Register Guard reads the plaintext register value from the network function, encrypts the register value, and computes the corresponding MAC.

3) AES-GCM Engines: SNO Guard employs the 128-bit-key AES-GCM engine to ensure the confidentiality and integrity of the network function's I/O. AES-GCM was selected due to its proven security and high performance in memory [33] and packet encryption [34]. The AES-GCM engine computes an authentication tag during plaintext I/O encryption and verifies the tag during ciphertext I/O decryption. To minimize performance overhead of data-intensive I/O, SNO integrates a fully pipelined AES-GCM implementation for Packet Guard, DMA Guard, and Local Memory Guard, sustaining a throughput of one 128-bit input per clock cycle in steady-state operation. At a typical FPGA frequency of 250MHz, the AES-GCM engine can sustain the bandwidth of 25Gbps Ethernet. AES-GCM engine design supporting higher network bandwidth like 100Gbps [35] can be seamlessly integrated into SNO Guard. For Register Guard, which secures 32-bit register accesses where performance is less critical, SNO employs a lightweight AES-GCM implementation that processes one 128-bit input every ten clock cycles, significantly reducing FPGA resource utilization. Leveraging FPGA reconfigurability, users can also customize the AES-GCM design to balance performance and resource constraints based on their requirements.

4) Overlapped Execution for Packet-Level Parallelism: To preserve the acceleration benefits of offloading network functions from the host to the FPGA-based SmartNIC, SNO Guard must minimize the performance overhead of AES-GCM within the packet processing pipeline. To exploit parallelism between SNO Guard and the network function, we design an overlapped execution flow for consecutive

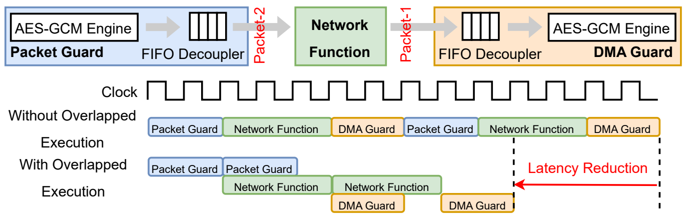


Fig. 7: Overlapped execution in SNO Guard using FIFO decouplers, reducing latency.

packets, as shown in Fig. 7. The key idea is to decouple SNO Guard from the network function, effectively hiding AES-GCM engine latency. A First-In-First-Out (FIFO) buffer with a streaming interface is implemented as the decoupling mechanism. As illustrated in Fig. 7, processing of the second packet begins as soon as the first packet completes Packet Guard processing, rather than waiting for DMA Guard processing of the first packet. This overlap significantly reduces overall latency compared to a non-overlapped design.

5) Integrity Protection Considerations: The MAC and version number (VN) ensure data integrity and protect against replay attacks. In AES-GCM, the MAC is the GHASH tag of the ciphertext and additional authenticated data (e.g., packet headers). The VN, concatenated with a random value, forms the IV for an AES-GCM invocation. Two key design considerations follow.

- ***Function-specific Integrity Granularity and Version Number Protection*** The data integrity granularity enforced by a MAC should align with the network function's data access pattern. The CPU TEE's 64-byte cacheline granularity is unsuitable due to the high overhead of authentication tag computation, MAC storage, and VN management. For network I/O, Packet Guard adopts packet-level granularity, assigning one MAC and one VN per packet. Packet Guard maintains two internal counters: `tx_vn` (transmit-side VN) and `rx_vn` (expected receive-side VN). The initial `tx_vn` is negotiated between the host application and the remote user during the SNO attestation, and `rx_vn` is initialized to the remote user's `tx_vn`. On packet transmission, `tx_vn` is incremented; on reception, Packet Guard verifies the packet's VN against `rx_vn` before decryption and authentication, ensuring replay attack detection. Notably, Packet Guard's MAC and VN management operate independently of network transport protocols, which remain under the control of an untrusted shell [36]. For host or local memory I/O, the protection granularity is function-specific. For instance, in multi-slot DMA ring buffers used for communication between host and the network function, slot-level granularity can be enforced, with a global VN managed by DMA Guard and the host application.
- ***Pipeline-efficient MAC Placement***: To maximize pipeline efficiency, the MAC should follow the ciphertext. Figure 8 compares two network packet structures and their ciphertext output timing in Packet Guard. Since the AES-GCM standard finalizes the authentication tag after processing all ciphertext and AAD, placing the MAC after the ciphertext naturally aligns with the algorithm's data flow, avoiding pipeline stalls or buffering that would occur if the MAC needed to precede the ciphertext. Furthermore, users must allocate sufficient ciphertext buffer space to enable DMA Guard and Local Memory Guard to append the MAC after the ciphertext in a single memory transfer, avoiding initiating separate memory transfers for the ciphertext and the MAC.

6) Cache for Memory Access: BRAM- or URAM-based internal caches can be optionally embedded in DMA Guard and Local Memory Guard to reduce performance overhead for repeated data access. The

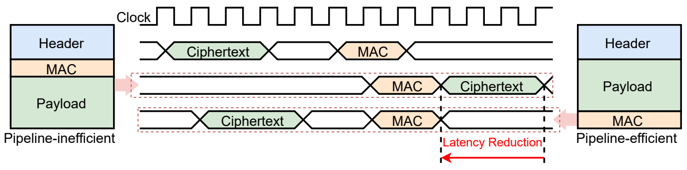


Fig. 8: The pipeline-efficient approach reduces ciphertext output latency compared to alternative MAC placements.

internal cacheline can be designed to match the memory integrity protection granularity, storing plaintext data. Encryption and authentication occur only when a cacheline is evicted, while cache line fetches from off-chip memory involve decryption and integrity verification using the corresponding MAC. One key design consideration follows. ***Cache should be used carefully***: If FPGA resource constraints allow, caching can be beneficial for memory exclusively owned by the network function. As demonstrated by ShEF [8], caching significantly improves performance for workloads with high locality. However, for memory shared between the network function and the host application, caching should be bypassed to ensure memory coherence. For example, if the writes to host memory to notify the host of an event, writes cached in DMA Guard remain invisible to the host CPU's memory controller, preventing timely event processing.

V. Evaluation

We first quantitatively evaluate SNO across the following dimensions: data path performance, control path performance, SoC CPU utilization, user friendliness, and FPGA resource consumption. Finally, we provide an informal security analysis of SNO.

A. Experimental Setup

1) RTL Simulation: We describe SNO Guard at the register-transfer level (RTL) using SystemVerilog to evaluate its cycle-accurate performance overhead and detailed FPGA resource consumption. The AES-GCM engine in SNO Guard is implemented based on an open-source design [37]. To determine SNO Guard's upper-bound performance, we employ a fully pipelined AES-GCM engine by default. However, a more resource-efficient AES-GCM configuration can be used when actual performance requirements are lower. Host memory access is simulated using the DDR4 model from the Alibaba Cloud FPGA Simulation Platform [38]. Currently, PCIe-related latency for host memory access is not modeled due to the lack of an available simulation model. Including PCIe latency in the performance evaluation would reduce SNO Guard's relative performance overhead. The typical PCIe latency (over 800 ns [39]) is an order of magnitude larger than the maximum absolute overhead introduced by SNO Guard (96 ns), reinforcing that SNO's impact is negligible within the end-to-end data path. For simulation and resource analysis, we use AMD Vivado 2023.2, targeting the xcu200-fsgd2104-2-e FPGA, which is similar to the FPGA used in AWS F1 instances.

2) FPGA Prototype: We prototype the TEE-protected bitstream loader of SNO Manager on an AMD Zynq UltraScale+ MPSoC ZU3EG, focusing on the secure boot and TEE-protected bitstream loading mechanism. OP-TEE v3.18.0 runs on the ARM Cortex-A53 CPU, providing a secure execution environment alongside the non-secure Linux kernel. The SNO bitstream loader operates as a Pseudo Trusted Application at EL1 in the Secure World, ensuring hardware-enforced isolation. For comparison, we implement the ShEF [8] bitstream loader as a bare-metal application on the same platform. Both implementations follow identical bitstream loading sequences to ensure a fair evaluation. The tools used include AMD Vivado 2023.2, AMD Vitis 2023.2, and Petalinux 2023.2.

3) Benchmarks: To evaluate the performance overhead of SNO Guard, we selected a diverse set of popular network functions typically offloaded to SmartNICs, including Aggregate, Histogram, All_Reduce, Filtering, and Strided_DDT. These five workloads are derived from the PsPIN SmartNIC benchmark [40] and recent FPGA-based SmartNIC research [41]. Since these workloads were neither designed for FPGA-based SmartNICs nor available as open-source, we reimplement them in SystemVerilog, ensuring adherence to their specified functionalities.

- Aggregate: Accumulates packet payloads in 4-byte granularity into an internal 8-byte accumulator and writes the accumulator to host memory at the end.
- Histogram: Increments frequency counts in a 4096-bin histogram based on packet payloads and notifies the host with an 8-byte DMA write at the end.
- All_Reduce: Reduces gradients in the incoming packet payload into a local gradient block and sends the neighboring gradient block to the network.
- Filtering: Computes a hash from the source IP using Jenkins Lookup3, modifies the UDP destination port based on the hash value, and writes the modified packet to host memory.
- Strided_DDT: Performs direct data transfer (DDT) that copies packet payloads to host memory using a specified starting address and stride.

B. Data Path Performance Evaluation

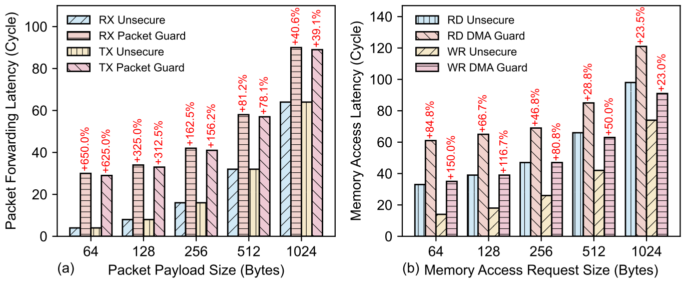


Fig. 9: (a) Packet forwarding latency comparison between Packet Guard and Unsecure, (b) Memory access latency comparison between DMA Guard and Unsecure, with relative latency overhead to the Unsecure baselines annotated.

1) Micro-Benchmark: We use micro-benchmarks to analyze the raw performance overhead of Packet and DMA Guard, independent of upstream and downstream network functions. ① For Packet Guard, we measure packet forwarding latency for ciphertext and plaintext packet in both RX and TX directions. The baseline (Unsecure) configuration forwards packets without additional processing. Figure 9(a) compares RX/TX latency between Unsecure and Packet Guard across increasing packet payload sizes. ② For DMA Guard, we evaluate memory access latency for one write (WR) and read (RD) request. The baseline (Unsecure) configuration performs direct memory access without additional processing. Direct quantitative comparison with ShEF Shield's DMA [8] is challenging due to fundamental differences in cryptographic algorithms and enforced transfer granularity. First, DMA Guard leverages AES-GCM, which offers greater parallelism than ShEF Shield's AES+HMAC [42]. Second, ShEF Shield enforces fixed 4096-byte transfers (64-byte bus width, cache-bypass mode) regardless of application demand, leading to mismatched workload assumptions for fine-grained I/O. Figure 9(b) compares WR/RD latency between Unsecure and DMA Guard with increasing memory access request size. The relative latency overhead of Packet and DMA Guard is computed against their Unsecure baselines.

We derive two key observations from Fig. 9. First, the relative latency overhead decreases as the size of protected data increases. The

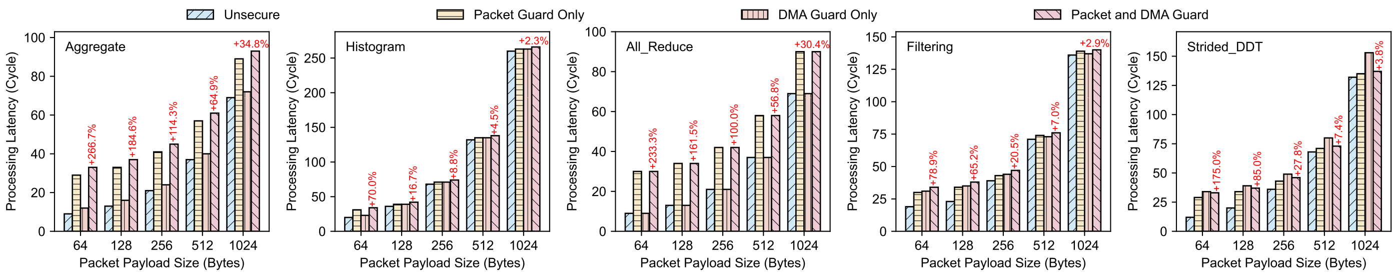


Fig. 10: Average packet processing latency comparison of Unsecure, Packet Guard, DMA Guard, and both Packet and DMA Guard across the five network functions, with the relative processing latency overhead to the Unsecure baseline annotated.

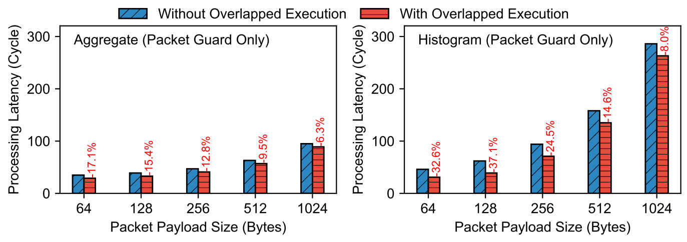


Fig. 11: Impact of overlapped execution on Packet Guard, with the relative processing latency reduction annotated.

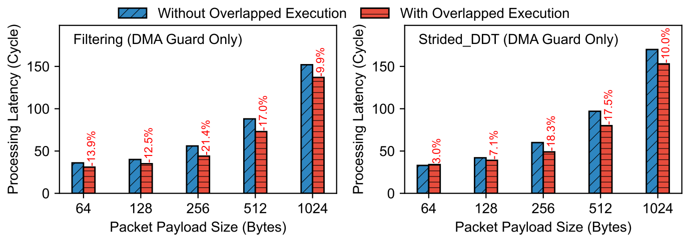


Fig. 12: Impact of overlapped execution on DMA Guard, with the relative processing latency reduction annotated.

AES-GCM engine is fully pipelined, meaning that after initializing the first keystream block upon receiving a new IV, the AES-GCM engine seamlessly encrypts or decrypts the entire data stream with only a single additional clock cycle of overhead—regardless of the total number of processed blocks. Since the AES-GCM initialization latency remains constant, its amortized impact diminishes as data size grows. Second, Packet Guard incurs a higher relative latency overhead than DMA Guard. This discrepancy arises because off-chip memory accesses exhibit significantly higher latency compared to packet forwarding. Consequently, the AES-GCM initialization latency is more effectively amortized in DMA operations.

2) Performance Evaluation on Network Functions: Figure 10 compares network function performance across varying security levels. The baseline (Unsecure) does not protect network or host memory I/O, while the highest security level integrates both Packet and DMA Guard. Intermediate configurations deploy either Packet or DMA Guard independently. Performance is evaluated as the average packet processing latency over eight consecutive packets, with each packet injected as soon as the network function becomes ready.

We derive three key observations from Fig. 10. First, the geometric mean of relative processing latency overhead across the five network functions ranges from 7.7% to 143.2% (corresponding to absolute overheads between 56 ns and 96 ns at the 250 MHz FPGA frequency). We conclude that SNO Guard-protected network functions maintain the performance benefit of offloading, as the incurred overhead remains well below an order of magnitude of the software network stack latency (a few microseconds [43]). Second, SNO Guard's relative overhead decreases with larger packet payloads due to (1) greater amortization of the fixed AES-GCM initialization latency and (2) improved efficiency from overlapping execution across a wider processing window. Third, functions with minimal I/O and higher computational complexity exhibit lower overhead: reduced I/O lessens data protection overhead, while increased computational intensity amortizes SNO Guard's latency and further expands the overlapping execution window.

3) Ablation Study: To assess the effectiveness of overlapped execution in SNO Guard, we selectively disable the overlapped execution in Packet Guard and DMA Guard and measure the processing latency across different network functions. Figure 11 illustrates the performance impact of overlapped execution in Packet Guard for Aggregate and Histogram, with DMA Guard disabled. Similarly, Figure 12 presents the performance impact of overlapped execution in DMA Guard for Filtering and Strided_DDT, with Packet Guard disabled. We compute the relative latency reduction of overlapped execution by comparing it against a non-overlapped baseline.

We derive three key insights from Fig. 11 and Fig. 12. First, overlapped execution significantly reduces processing latency by 6.3% to 37.1% for Packet Guard (Aggregate, Histogram) and 7.1% to 18.3% for DMA Guard (Filtering, Strided_DDT). Second, its benefits scale with network function complexity. For example, Histogram's bin dependencies introduce higher execution latency compared to Aggregate, widening the overlapping window for Packet Guard's AES-GCM engine. Similarly, Filtering's hash computation makes it compute-intensive compared to Strided_DDT, enhancing the effectiveness of overlapped execution. Third, DMA Guard incurs performance degradation for Strided_DDT at 64-byte payloads due to a 3-cycle FIFO overhead per packet, outweighing the gains from overlapped execution. However, enabling both Packet and DMA Guards together outperforms DMA Guard alone at 64-byte payloads, as shown in Fig. 10, where Packet Guard increases upstream execution latency, amplifying the benefits of overlap.

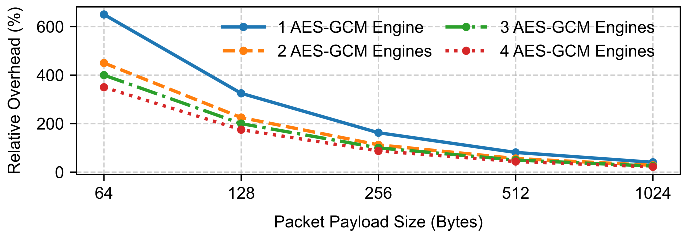


Fig. 13: Relative packet forwarding latency overhead of RX Packet Guard with varying numbers of AES-GCM engines.

4) Discussion: Reducing Performance Overhead. We propose an approach to further reduce performance overhead without compromising security. Deploying multiple AES-GCM engines in round-robin amortizes the fixed initialization latency, which dominates overhead

Bitstream Size	ShEF	SNO	SNO Overhead
1.44MiB	33.2 ms	76.1 ms	42.9 ms
2.95MiB	60.9 ms	104.9 ms	44.0 ms

TABLE I: Comparison of FPGA bitstream loading latency between ShEF and SNO.

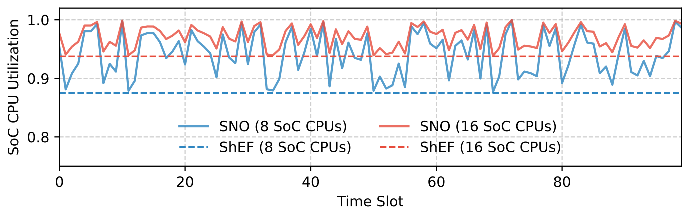


Fig. 14: SoC CPU utilization comparison between SNO and ShEF.

for small packets. The initialization process is security-critical, as each encryption requires a unique IV. Our proposal trades FPGA resources for performance: with m engines, the effective initialization interval drops from N to $\frac{N}{m}$ cycles. Figure 13 shows the relative packet forwarding latency overhead of RX Packet Guard using 1 to 4 AES-GCM engines. While multiple engines substantially reduce overhead for small packets, the benefit diminishes with larger packets.

Performance Overhead at Higher Network Bandwidth (e.g., 100Gbps). SNO Guard maintains constant performance overhead as network bandwidth scales, assuming sufficient FPGA resources. At 100Gbps and 250MHz, four 128-bit data blocks arrive per cycle, but a single AES-GCM pipeline processes only one block per cycle. We use the parallel-pipelined AES-GCM engine [35], which incorporates four AES-CTR pipelines and four GHASH units to enable four-way block processing in the steady state. Final tag generation incurs only **one additional cycle** to aggregate partial GHASH results.

C. Control Path Performance Evaluation

First, we evaluate the boot time overhead of OP-TEE initialization for SNO on our FPGA prototype. We measure the boot time of a baseline image (without OP-TEE) and an OP-TEE-enabled image across three iterations. The results show that OP-TEE adds 0.58 seconds on average to the boot process—a negligible overhead compared to the 39+ seconds required for VM setup in public clouds [44].

Second, we evaluate the FPGA bitstream loading latency of SNO Manager's TEE-protected loader compared to the ShEF loader. Table I shows that SNO introduces a consistent 40-millisecond overhead compared to ShEF. This overhead is negligible for network functions, which typically run for minutes to hours [6].

D. SoC CPU Utilization Improvement

SNO Manager can be co-located with CSP software on the SoC CPU, increasing the available SoC CPU resource for the CSP software. We model SoC CPU utilization to compare SNO and ShEF quantitatively. Assume the SoC has M SoC CPUs. In each time slot, the SNO Manager or the ShEF Security Kernel occupies x SoC CPUs, where x follows a uniform distribution between 0 and 1 for the SNO Manager but is fixed at 1 for the ShEF Security Kernel. SoC CPU utilization is defined as $\frac{M-x}{M}$, representing the available SoC CPUs for the CSP. Figure 14 shows that the SNO Manager achieves higher SoC CPU utilization, which improves the SoC CPU utilization by 6.6% and 3.3% for 8- and 16-CPU SoC.

E. User Friendliness Improvement

We evaluate user friendliness by comparing line-of-code (LoC) requirements for SNO Guard's streaming interface and ShEF Shield's memory-mapped interface. For the Strided_DDT workload, SNO Guard requires only 89 LoC, an 86.9% reduction from the 675 LoC needed for the memory-mapped baseline (aligned via SNO DMA Guard's backend for a fair comparison). This reduction highlights SNO Guard's user friendliness, enabling simpler integration, lower complexity, and faster user adoption.

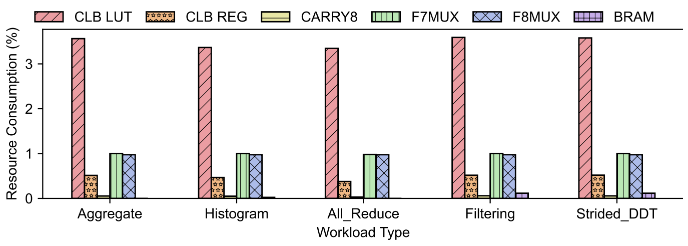


Fig. 15: Resource consumption of SNO Guard across workloads.

F. Resource Consumption

We evaluate the FPGA resource consumption of SNO Guard, including CLB LUT, CLB REG, CARRY8, F7MUX, F8MUX, and BRAM. Since FIFO decoupler usage varies across network functions, we report SNO Guard's consumption separately in Fig. 15, adjusting FIFO depths based on network function requirements. We have the following key result: SNO Guard incurs acceptable resource overhead, with CLB LUT usage below 4% and other resources below 1% in cloud FPGA settings.

SNO Guard's resource consumption scales with network bandwidth. For example, the AES-GCM engine dominates the resource, consuming 85.5–96.8% of total CLB LUTs across all workloads. The parallel-pipelined [35] architecture linearly increases both AES-CTR pipelines and GHASH modules, resulting in near-linear CLB LUT scaling with network bandwidth.

G. Security Analysis

Attacks from the CSP. All I/O of a network function exposed to the CSP-controlled shell is secured via AES-GCM authenticated encryption. The shell, while monitoring all I/O (network packets, host DMA, memory accesses), cannot compromise data confidentiality or integrity without the correct keys and IVs. The CSP has no access to the correct combination of AES key and IV due to: (1) The host application safeguards these secrets in host CPU TEE; (2) The secrets within the network function are physically inaccessible from the CSP.

Physical Attacks on the Off-chip Components All I/O of a network function residing in local and host memory and traversing PCIe links is secured using AES-GCM authenticated encryption. AES-GCM keys and IVs are exclusively provisioned by users within secure environments. For instance, DMA symmetric keys are fused into encrypted bitstreams during network function synthesis. In our threat model, attackers cannot extract keys through package tampering, restricting their capabilities to (1) passively observing ciphertext (with confidentiality preserved) or (2) actively attempting data modifications (which are immediately detected through integrity verification).

VI. Conclusion

Ensuring the security of network functions on FPGA-based SmartNICs is crucial for sensitive users. We present SNO, the first TEE designed to enable secure, high-performance, and user-friendly offloading on FPGA-based SmartNICs. SNO consists of three key components: a secure boot mechanism for a verified execution environment, the TEE-hosted SNO Manager for secure network function deployment, and the SNO Guard with a streaming interface for I/O protection. Our evaluations demonstrate that SNO introduces negligible overhead, maintaining the acceleration benefits of offloading.

References

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking:smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[2] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang *et al.*, "From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 753–766.

[3] W. Lin, Y. Shan, R. Kosta, A. Krishnamurthy, and Y. Zhang, "Supernic: An fpga-based, cloud-oriented smartnic," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024, pp. 130–141.

[4] V. Costan, "Intel sgx explained," *IACR Cryptol, EPrint Arch*, 2016.

[5] W. Jansen, T. Grance *et al.*, "Guidelines on security and privacy in public cloud computing," 2011.

[6] Y. Zhou, M. Wilkening, J. Mickens, and M. Yu, "Smartnic security isolation in the cloud with s-nic," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 851–869.

[7] K. Xia, Y. Luo, X. Xu, and S. Wei, "Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 301–306.

[8] M. Zhao, M. Gao, and C. Kozyrakis, "Shef: Shielded enclaves for cloud fpgas," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1070–1085.

[9] Intel, "Fpga ipu platform c5000x-pl," https://www.intel.com/content/www/us/en/products/details/fpga/platforms/ipu/c5000x-pl-platform.html, 2025, accessed: 2025-03-24.

[10] AMD, "Amd alveo sn1000 smartnic accelerator card," https://www.amd.com/en/products/accelerators/alveo/sn1000/a-sn1022-p4.html, 2025, accessed: 2025-03-24.

[11] X. Li, X. Jiang, Y. Yang, L. Chen, Y. Wang, C. Wang, C. Xu, Y. Lv, B. Yang, T. Wu *et al.*, "Triton: A flexible hardware offloading architecture for accelerating apsara vswitch in alibaba cloud," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 750–763.

[12] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[13] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–39, 2018.

[14] E. Peterson, "Developing tamper-resistant designs with zynq ultrascale+ devices," *Xilinx Application Note*, 2018.

[15] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," 2007.

[16] A. Paverd, A. Martin, and I. Brown, "Modelling and automatically analysing privacy properties for honest-but-curious adversaries," *Tech. Rep*, 2014.

[17] W. R. Claycomb and A. Nicoll, "Insider threats to cloud computing: Directions for new research challenges," in *2012 IEEE 36th annual computer software and applications conference*. IEEE, 2012, pp. 387–394.

[18] M. Ye, X. Feng, and S. Wei, "Hisa: Hardware isolation-based secure architecture for cpu-fpga embedded systems," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[19] M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 229–244.

[20] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel tdx demystified: A top-down approach," *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–33, 2024.

[21] A. Sev-Snp, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, vol. 53, no. 2020, pp. 1450–1465, 2020.

[22] AMD, "On-chip memory," https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/On-chip-Memory, 2025, accessed: 2025-03-20.

[23] M. Gross, K. Hohentanner, S. Wiehler, and G. Sigl, "Enhancing the security of fpga-socs via the usage of arm trustzone and a hybrid-tpm," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–26, 2021.

[24] Y. Wang, X. Chang, H. Zhu, J. Wang, Y. Gong, and L. Li, "Towards secure runtime customizable trusted execution environment on fpga-soc," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 1138–1151, 2024.

[25] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee–an open virtual trusted execution environment," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 400–407.

[26] B. Hu, Y. Wang, J. Cheng, T. Zhao, Y. Xie, X. Guo, and Y. Chen, "Secure and efficient mobile dnn using trusted execution environments," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 274–285.

[27] Wikipedia contributors, "Diffie–hellman key exchange — Wikipedia, the free encyclopedia," 2025, [Online; accessed 30-March-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Diffie%E2%80%93Hellman_key_exchange&oldid=1279458222

[28] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An open-source 100-gbps nic," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 38–46.

[29] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "Panic: A high-performance programmable nic for multi-tenant networks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 243–259.

[30] R. Bhaktavatchalu, B. S. Rekha, G. A. Divya, and V. U. S. Jyothi, "Design of axi bus interface modules on fpga," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. IEEE, 2016, pp. 141–146.

[31] M. Zonta-Roudes, A. Meza, N. Hinderling, L. Deutschmann, F. Restuccia, R. Kastner, and S. Shinde, "expect: On the security implications of violations in axi implementations," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[32] AMD, "Dma for pci express (pcie) subsystem," https://www.xilinx.com/products/intellectual-property/pcie-dma.html, 2025, accessed: 2025-03-18.

[33] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 179–190, 2006.

[34] J. Salowey, A. Choudhury, and D. McGrew, "Aes galois counter mode (gcm) cipher suites for tls," Tech. Rep., 2008.

[35] L. Henzen and W. Fichtner, "Fpga parallel-pipelined aes-gcm core for 100g ethernet applications," in *2010 Proceedings of ESSCIRC*. IEEE, 2010, pp. 202–205.

[36] D. Korolija, T. Roscoe, and G. Alonso, "Do os abstractions make sense on fpgas?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 991–1010.

[37] BLu85, "Aes-gcm-128-192-256-bits," https://github.com/BLu85/AES-GCM-128-192-256-bits, 2023, gitHub repository.

[38] A. Cloud, "Alibaba cloud fpga: Open-source fpga development resources," https://github.com/aliyun/alibabacloud-fpga, 2019, gitHub repository.

[39] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 327–341.

[40] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler, "A risc-v in-network accelerator for flexible high-performance low-power packet processing," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 958–971.

[41] R. Ma, E. Georganas, A. Heinecke, S. Gribok, A. Boutros, and E. Nurvitadhi, "Fpga-based ai smart nics for scalable distributed ai training systems," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 49–52, 2022.

[42] S. Gueron, "Aes-gcm for efficient authenticated encryption–ending the reign of hmac-sha-1," *Real-World Cryptography*, 2013.

[43] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, "A study of networking software induced latency," in *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015, pp. 1–8.

[44] K. Hoffman, "Comparing the speed of vm creation and ssh access of cloud providers," https://blog.cloud66.com/, 2017.