Guardians of the Chip: Mastering Next-Gen Security for SoCs and IPs

Jagata Sridevi, sridevi@cadence.com , Cadence Design Systems, India
Vishnu Prasad K V, vishnuv@cadence.com , Cadence Design Systems, India
Deep Mehta, deep@cadence.com , Cadence Design Systems, India

***Abstract*- In the era of high-speed data protocols driving next-generation system-on-chips (SoCs) and intellectual properties (IPs), ensuring robust security has become critical. This paper focuses on PCI Express (PCIe) and Compute Express Link (CXL) protocols, both pivotal in high-performance environments such as data centers, AI-driven systems, and cloud computing. This paper explores the Layered IDE Packet Security Framework for PCIe/CXL. The main intent of this paper is to categorize key verification blind spots—ranging from error detection and state transitions to resource management and protocol-specific handling. By applying a suite of targeted verification techniques, including State Machines, Operation Codes, White Box Logging, Modular Debug Hooks, and Flexible Verification Methods, we achieve a comprehensive ~97% coverage in protocol verification. Our findings underscore the importance of rigorous security verification to mitigate risks, enhance data integrity, and ensure reliable performance in complex, high-speed systems. This research provides a practical framework for addressing emerging security challenges in SoCs and IPs, paving the way for secure and efficient system designs.**

Keywords—IDE(Integrity and Data Encryption), AAD(Association Address), AES(Advanced Encryption Stamderd)

I. INTRODUCTION

In the pursuit of secure, high-performance system-on-chips (SoCs) and intellectual properties (IPs), understanding the underlying protocols and their security mechanisms is essential. This section provides an overview of key protocols—PCI Express (PCIe), Compute Express Link (CXL), Ethernet (MACsec), MIPI, Thunderbolt, HDMI, and DisplayPort—highlighting their roles in enabling high-speed data transfer across applications from data centers to embedded systems. While each protocol integrates specific security frameworks, they also present unique vulnerabilities that, without rigorous verification, may go undetected and lead to security breaches. In these protocols, Advanced Encryption Standard (AES) algorithms, along with robust key management techniques, form the backbone of their security frameworks.

A. PCI Express (PCIe):

PCIe is a high-speed interface connecting GPUs and SSDs, crucial in data centers and AI. It uses the IDE framework with AES-GCM encryption and MACs to secure data against unauthorized access.

B. Compute Express Link (CXL):

CXL facilitates data sharing between processors and accelerators for cloud and AI applications. It relies on IDE protections to maintain data integrity in complex memory-sharing scenarios, requiring multi-layered security measures.

This overview highlights each protocol's functionality and security approach, emphasizing the need to address blind spots that could compromise system integrity in high-performance environments.

TABLE I
WIDELY USED PROTOCOLS AND ITS SECURITY FRAMEWORKS

Protocol	**Security framework**	**Authentication Mechanisms**
PCIe and CXL	Integrity and Data Encryption	AES-GCM and MAC
Ethernet	Media Access Control Security (MACsec)	AES-GCM -128, 192, 256 bit
MIPI	CMAC and GMAC	MAC Algorithms
HDMI	High Band width Digital Content Protection (HDCP)	Copy Protection Scheme

As high-stakes applications like autonomous vehicles, data centers, and AI-driven systems rely on rapid data exchanges, secure and resilient protocols are essential to prevent catastrophic failures and breaches. This is especially critical in high-performance environments such as data centers and AI computing, where protocols like PCIe and CXL enable high-speed communication between processors, memory, and accelerators.

II. Motivation: Application Example

C. GPS Data Integrity in Cloud-Based Autonomous Vehicle Fleets

In a cloud-based data center supporting autonomous vehicle operations, high-speed data transfer between processors and memory modules is vital for processing real-time GPS coordinates and sensor data. However, without robust security measures, these high-speed channels introduce vulnerabilities that attackers could exploit to compromise data integrity. For example, an attacker could intercept, manipulate, or inject GPS data, leading to potential risks in vehicle navigation and fleet management."

For instance, an attacker could use an interposer—a hardware device inserted between PCIe or CXL communication paths—to intercept and manipulate GPS data as it flows through the data center. With an interposer in place, the attacker could:

1) *Intercept and Monitor GPS Data:* By capturing real-time GPS coordinates, the attacker gains unauthorized access to vehicle locations, violating data privacy and exposing sensitive positional information.
2) *Spoof or Manipulate GPS Signals*: The interposer could modify GPS data in real time, misguiding vehicles to unintended or unsafe routes. This misrouting not only jeopardizes passenger safety but also risks traffic disruptions and costly mismanagement of the autonomous fleet.
3) *Inject False Data*: By injecting falsified GPS or sensor signals, an attacker could misdirect multiple vehicles simultaneously, resulting in cascading navigation errors across the fleet.

This scenario underscores the necessity of implementing secure, encrypted data transfer across PCIe and CXL protocols, emphasizing the role of end-to-end security measures and verification to prevent hardware-based threats like interposer attacks.

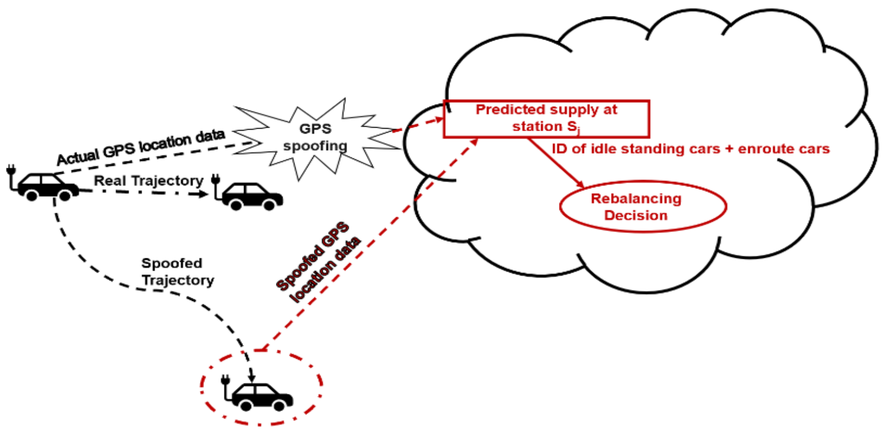
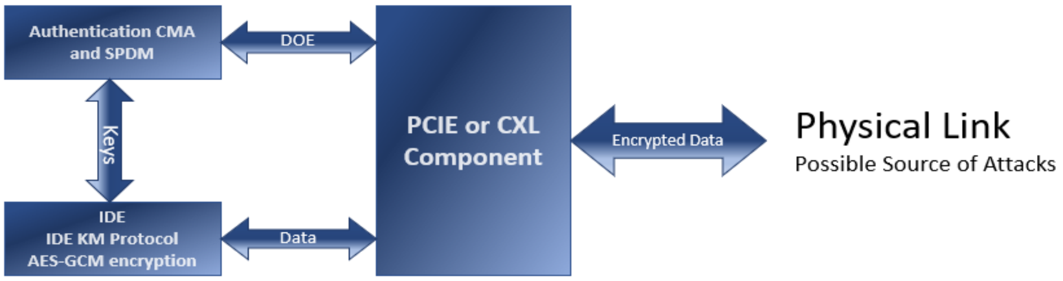

Actual GPS location data
GPS spoofing
Real Trajectory
Spoofed Trajectory
Spoofed GPS location data
Predicted supply at station S_j
ID of idle standing cars + enroute cars
Rebalancing Decision

Figure 1. GPS spoofing attack in vehicle.

III. Background and Literature Review

Integrity & Data Encryption (IDE) is a security framework for PCIe and CXL, providing confidentiality, integrity, and replay protection for Transaction Layer Packets (TLPs). IDE supports multiple use models and ensures interoperability, utilizing cryptographic mechanisms aligned with industry best practices to adapt to evolving security requirements.

IDE mitigates threats from physical attacks on PCIe/CXL links, such as attempts to access, modify, reorder, or delete confidential data using specialized equipment or malicious devices. It secures data as they traverse switches, safeguarding against risks from compromised or reprogrammed switches. The framework also supports secure traffic management within trusted execution environments.



.

Figure 2. Functionality of IDE.

The above diagram illustrates the security framework for PCIe/CXL, highlighting authentication, IDE, and encryption mechanisms to secure data across vulnerable physical links. Certificate Management and Authentication (CMA) and Security Protocol and Data Model (SPDM) are used to authenticate connected devices, ensuring only trusted devices exchange data. IDE includes Key Management (KM) and AES-GCM encryption. AES-GCM provides data confidentiality and integrity by securing data packets, while the KM protocol handles encryption key generation, distribution, and refreshment.

D. Overview of PCIe IDE:

PCIe IDE establishes two types of streams between ports: **Selective IDE Streams**, which apply to specific TLPs based on defined association rules, and **Link IDE Streams**, which encompass all TLPs transmitted using a particular traffic class, excluding those under a Selective IDE Streams. This dual-stream configuration enables tailored security policies for different types of traffic.

The PCIe IDE framework includes interoperable mechanisms for stream establishment and key management, leveraging industry specifications. It emphasizes a structured approach to trust establishment and key exchange, ensuring that security measures are robust and flexible enough to adapt to evolving threats while maintaining compatibility across various system components.

E. Overview of CXL IDE

CXL IDE safeguards data traffic within the CXL architecture, ensuring that sensitive information remains confidential and tamper-proof while in transit. This is particularly important for high-performance environments such as cloud computing, memories and autonomous vehicles.

The specification details distinct security protocols for different types of traffic. CXL.io IDE adapts principles from PCIe IDE while accounting for protocol-specific constraints, ensuring seamless performance. Meanwhile, CXL.cachemem IDE secures CXL.cache and CXL.mem traffic, covering encryption, integrity checks, and unique handling of control flits to maintain data security without compromising protocol efficiency.

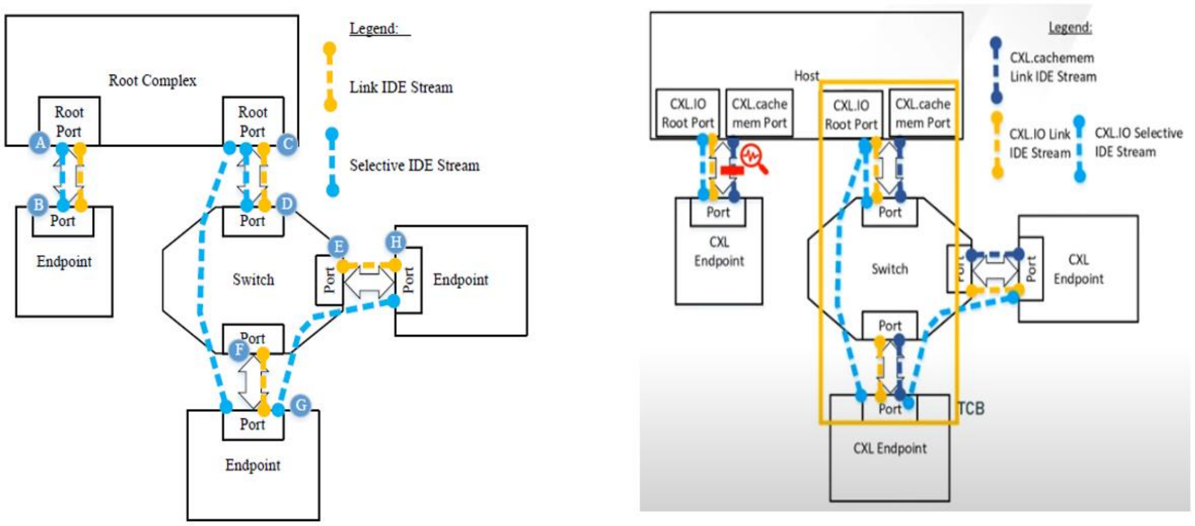


Figure 3. IDE Secures TLPs Between Ports for PCIe and CXL Protocols

IV. IMPLEMENTATION AND RELATED WORK

F. Layered IDE Packet Security Framework for PCIe/CXL: Techniques and Strategies:

Our multilayered verification strategy for IDE, as depicted in the flowcharts, is designed to ensure robust security from initialization through to transmission and reception, protecting against unauthorized access, data alteration, and malicious replays.

1) IDE Encryption:

The process starts with **IDE Settings** configuration, where foundational security parameters are set, laying the groundwork for secure communication. Following this, the **Stream Configuration** step prepares the data pathways, determining the conditions under which secure data exchange will occur. **Key Management** then initiates, where keys for encryption and decryption are established, periodically refreshed, and securely distributed, ensuring that even long-running systems maintain cryptographic robustness. Encryption is achieved through **AES-GCM** (Advanced Encryption Standard in Galois Counter Mode), providing both confidentiality and integrity for each packet.

Once these initial configurations are complete, **Secure Stream Enablement** is activated, transforming the Trusted Execution Environment (TEE) into a secure state. Only after this point does **Traffic Transfer** begin, with each data packet traveling through a controlled and secure stream, ensuring its integrity and confidentiality are upheld.

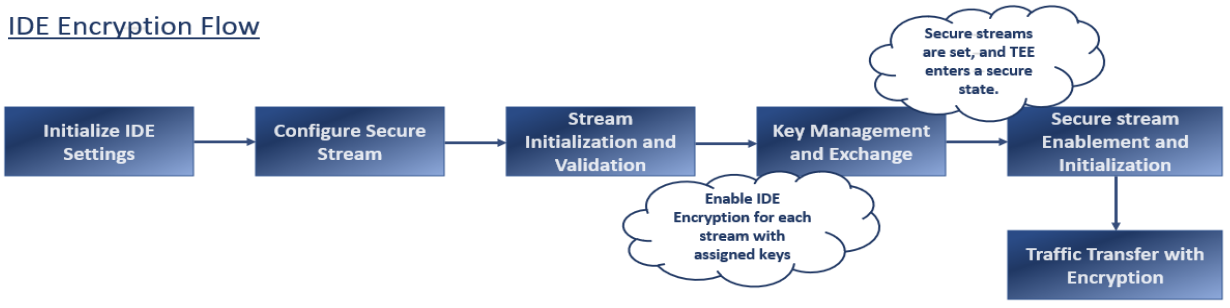

IDE Encryption Flow
Initialize IDE Settings
Configure Secure Stream
Stream Initialization and Validation
Key Management and Exchange
Secure stream Enablement and Initialization
Traffic Transfer with Encryption
Secure streams are set, and TEE enters a secure state.
Enable IDE Encryption for each stream with assigned keys

Figure 4. IDE Encryption Flow

2) IDE Decryption:

On the receiving end, **Traffic Reception** initiates the multi-layer verification of the incoming data. The data is first **decrypted** if it is an IDE packet, transforming it back into readable information while verifying its confidentiality. Next, the **Check for Stream** ensures that the data packet belongs to a recognized and authenticated stream, preventing any unauthorized streams from contaminating the secure environment.

Once stream validation passes, the **Check for MAC (Message Authentication Code)** verifies the data's integrity, confirming that it has not been tampered with during transmission. The subsequent **Check for PCRC (Packet Cyclic Redundancy Check)** layer further validates packet integrity, detecting any errors or corruption that may have occurred.

If any verification stage fails (e.g., incorrect stream, MAC mismatch, PCRC error), the protocol triggers error-handling mechanisms. Any untrusted or misrouted packets, failed IDE checks, or detected errors cause an **Insecure State** transition, halting the transfer and isolating the compromised data. The secure link can only be restored after a reset, ensuring that no insecure data persists in the system.

This multilayered IDE verification strategy essentially combines *proactive* and *reactive* security measures to create a flexible, high-assurance environment. By structuring security checks at each level—from initialization through to error handling—it builds redundancy into the system's defenses, fortifying the protocol against a spectrum of potential threats. The layered approach is robust enough to catch protocol-specific vulnerabilities yet versatile, balancing security and performance with mode-specific adaptations.

This approach also underscores a "zero-trust" principle, where every packet undergoes rigorous validation at multiple checkpoints, ensuring that each layer adds a unique safeguard. This structure, shown in the flow charts, reflects a multi-dimensional defense strategy that extends beyond traditional IDE verification and positions the protocol to withstand emerging threats in high-speed, data-intensive environments.

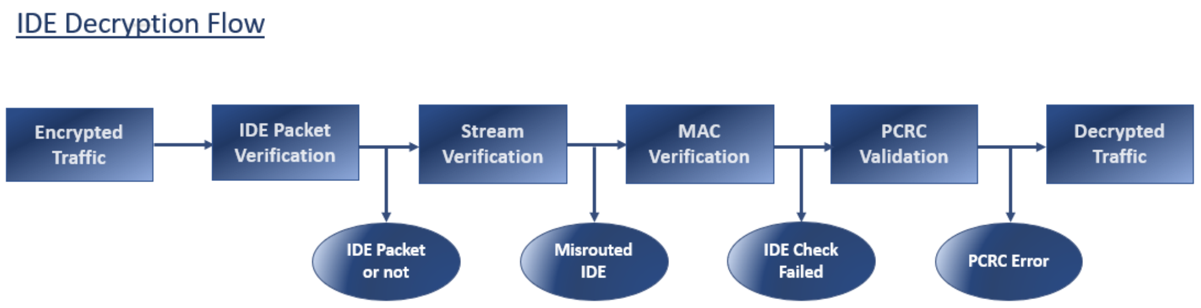


Figure 5. IDE Decryption Flow

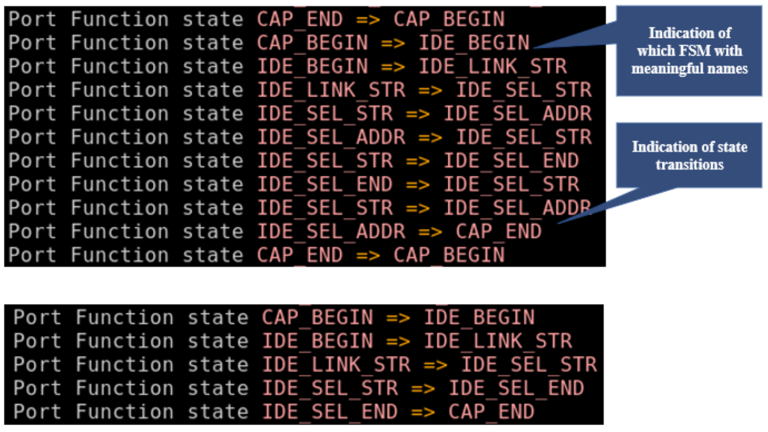
Building upon the implemented framework, several specialized techniques were developed to address the unique challenges encountered in the verification process. The following sections describe each technique in detail, illustrating how they address specific verification needs and contribute to the overall effectiveness of the system.

Technique 1: State Machine for IDE Capability Discovery

The State Machine for IDE Capability Discovery is a streamlined technique that simplifies managing IDE capability discovery, particularly in complex configurations like link and selective streams. By organizing each step of the discovery process into distinct states, such as IDE_BEGIN, IDE_LINK_STR, and IDE_SEL_ADDR, this approach provides clear transitions and enables precise tracking of each phase. This structure enhances alignment within verification teams, offering a predictable framework that aligns with the IDE specification and improves overall verification accuracy.

The implementation involves defining specific states and transitions that map directly to IDE discovery requirements. The FSM is integrated into the testbench, automating the process by handling IDE checks and transitioning through states as defined by the specification, ensuring the discovery process follows the correct flow.

In practice, this FSM approach serves as a roadmap for verification engineers, offering a structured path to monitor IDE discovery. It improves team communication by providing a common framework and enables consistent and repeatable verification of IDE capabilities, especially in complex scenarios, ensuring compliance with IDE standards.



```
Port Function state CAP_END => CAP_BEGIN
Port Function state CAP_BEGIN => IDE_BEGIN
Port Function state IDE_BEGIN => IDE_LINK_STR
Port Function state IDE_LINK_STR => IDE_SEL_STR
Port Function state IDE_SEL_STR => IDE_SEL_ADDR
Port Function state IDE_SEL_ADDR => IDE_SEL_STR
Port Function state IDE_SEL_STR => IDE_SEL_END
Port Function state IDE_SEL_END => IDE_SEL_STR
Port Function state IDE_SEL_STR => IDE_SEL_ADDR
Port Function state IDE_SEL_ADDR => CAP_END
Port Function state CAP_END => CAP_BEGIN
```

Indication of which FSM with meaningful names
Indication of state transitions

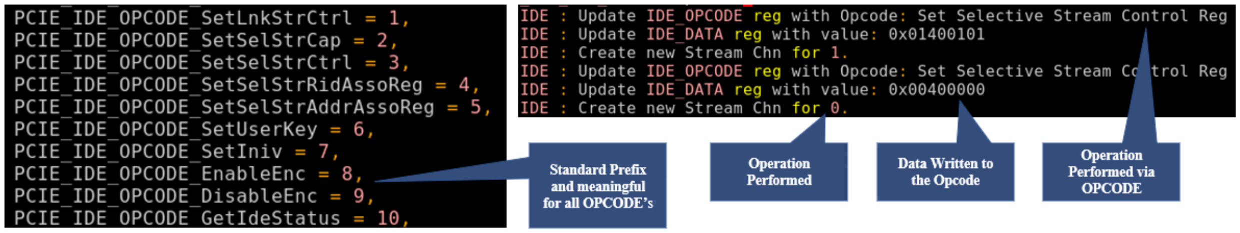
```
Port Function state CAP_BEGIN => IDE_BEGIN
Port Function state IDE_BEGIN => IDE_LINK_STR
Port Function state IDE_LINK_STR => IDE_SEL_STR
Port Function state IDE_SEL_STR => IDE_SEL_END
Port Function state IDE_SEL_END => CAP_END
```

Figure 6. FSM for IDE Capability Discovery

Technique 2: Operation Codes in IDE Context:

In the complex environment of IDE protocol verification, managing numerous unique operations can lead to inefficiencies, especially when each operation requires a dedicated API. To streamline this process, an opcode-based system offers a unified solution. Each operation is assigned a unique operation code (opcode), prefixed for easy recognition (e.g., PCIE_IDE_OPCODE_), allowing a single API to handle multiple functions efficiently. This approach enhances code readability, promotes consistency, and eliminates the need for fragmented APIs.

The unified API framework is built to recognize various IDE operations by their assigned opcodes. Instead of separate APIs, each function—such as SetLnksStrCtrl, EnableCmEnc, or GotoInsecure—is mapped to a unique opcode. This single API structure manages all operations, with the prefixed opcodes ensuring easy identification and invocation of specific functions. This setup simplifies the API's structure and supports future expansion by allowing additional operations to be added without restructuring. Each opcode corresponds to a distinct operation, enabling streamlined testing and debugging.



```
PCIE_IDE_OPCODE_SetLnkStrCtrl = 1,
PCIE_IDE_OPCODE_SetSelStrCap = 2,
PCIE_IDE_OPCODE_SetSelStrCtrl = 3,
PCIE_IDE_OPCODE_SetSelStrRidAssoReg = 4,
PCIE_IDE_OPCODE_SetSelStrAddrAssoReg = 5,
PCIE_IDE_OPCODE_SetUserKey = 6,
PCIE_IDE_OPCODE_SetIniv = 7,
PCIE_IDE_OPCODE_EnableEnc = 8,
PCIE_IDE_OPCODE_DisableEnc = 9,
PCIE_IDE_OPCODE_GetIdeStatus = 10,
```

```
IDE : Update IDE_OPCODE reg with Opcode: Set Selective Stream Control Reg
IDE : Update IDE_DATA reg with value: 0x01400101
IDE : Create new Stream Chn for 1.
IDE : Update IDE_OPCODE reg with Opcode: Set Selective Stream Control Reg
IDE : Update IDE_DATA reg with value: 0x00400000
IDE : Create new Stream Chn for 0.
```

Standard Prefix and meaningful for all OPCODE's
Operation Performed
Data Written to the Opcode
Operation Performed via OPCODE

Figure 7. Operation Codes and Its Implementation Results

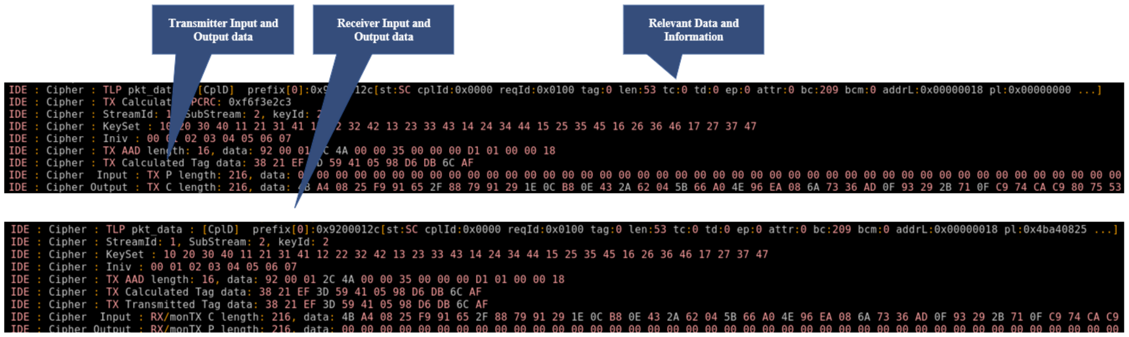
Technique 3: White Box Logging for Transparent Debugging:

White Box Logging provides a detailed view of each step within complex operations, particularly in high-security or performance-sensitive systems. Unlike black box logging, which only captures inputs and outputs, white box logging reveals internal parameters, transformations, and results, making it easier to trace issues and validate functionality.

White box logging is integrated into operations, such as cipher processes, to capture and display intermediate steps in a structured format. Logged Data contains Key parameters like StreamID, KeyID, and specific data points such as KeySet, Initialization Vector (IV), and AAD Length are recorded. This detail allows engineers to validate each component and transformation in real-time. Final Output Logging concludes with the processed output data, providing a full view of the input-output relationship and confirming the integrity of the operation.

White box logging enables quick issue tracing in security-critical operations, thorough validation of data integrity, and reliable tracking in high-stakes applications, ensuring immediate detection and correction of deviations.

Additionally, white box logging can be combined with **protocol checks** to further enhance error detection. Protocol checks ensure that all operations comply with defined standards and specifications.



Transmitter Input and Output data
Receiver Input and Output data
Relevant Data and Information

```
IDE : Cipher : TLP pkt_dat[illegible][CplD]  prefix[0]:0x9[illegible]12c[st:SC cplId:0x0000 reqId:0x0100 tag:0 len:53 tc:0 td:0 ep:0 attr:0 bc:209 bcm:0 addrL:0x00000018 pl:0x00000000 ...]
IDE : Cipher : TX Calculat[illegible]PCRC: 0xf6f3e2c3
IDE : Cipher : StreamId: 1[illegible]SubStream: 2, keyId: 2
IDE : Cipher : KeySet : 1[illegible]20 30 40 11 21 31 41 1[illegible]2 32 42 13 23 33 43 14 24 34 44 15 25 35 45 16 26 36 46 17 27 37 47
IDE : Cipher : Iniv : 00 [illegible] 02 03 04 05 06 07
IDE : Cipher : TX AAD len[illegible]th: 16, data: 92 00 01[illegible]C 4A 00 00 35 00 00 00 D1 01 00 00 18
IDE : Cipher : TX Calculated Tag data: 38 21 EF[illegible]D 59 41 05 98 D6 DB 6C AF
IDE : Cipher  Input : TX P length: 216, data: 0[illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IDE : Cipher Output : TX C length: 216, data: [illegible]B A4 08 25 F9 91 65 2F 88 79 91 29 1E 0C B8 0E 43 2A 62 04 5B 66 A0 4E 96 EA 08 6A 73 36 AD 0F 93 29 2B 71 0F C9 74 CA C9 80 75 53
```

```
IDE : Cipher : TLP pkt_data : [CplD]  prefix[0]:0x9200012c[st:SC cplId:0x0000 reqId:0x0100 tag:0 len:53 tc:0 td:0 ep:0 attr:0 bc:209 bcm:0 addrL:0x00000018 pl:0x4ba40825 ...]
IDE : Cipher : StreamId: 1, SubStream: 2, keyId: 2
IDE : Cipher : KeySet : 10 20 30 40 11 21 31 41 12 22 32 42 13 23 33 43 14 24 34 44 15 25 35 45 16 26 36 46 17 27 37 47
IDE : Cipher : Iniv : 00 01 02 03 04 05 06 07
IDE : Cipher : TX AAD length: 16, data: 92 00 01 2C 4A 00 00 35 00 00 00 D1 01 00 00 18
IDE : Cipher : TX Calculated Tag data: 38 21 EF 3D 59 41 05 98 D6 DB 6C AF
IDE : Cipher : TX Transmitted Tag data: 38 21 EF 3D 59 41 05 98 D6 DB 6C AF
IDE : Cipher  Input : RX/monTX C length: 216, data: 4B A4 08 25 F9 91 65 2F 88 79 91 29 1E 0C B8 0E 43 2A 62 04 5B 66 A0 4E 96 EA 08 6A 73 36 AD 0F 93 29 2B 71 0F C9 74 CA C9
IDE : Cipher Output : RX/monTX P length: 216, data: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 8. White box Logging and its Significance

Technique 4: Modular Debug Hooks for Layered Architectures:

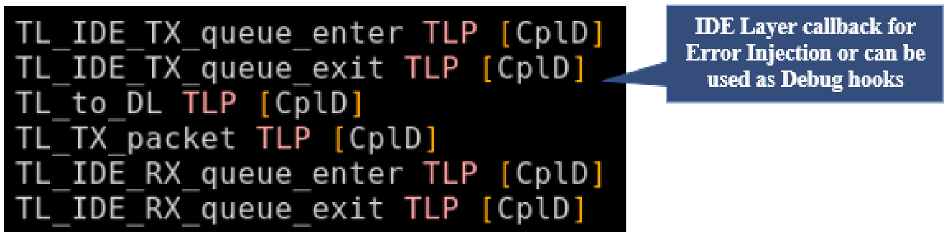
This technique introduces *modular* debug hooks within the PCIe/CXL architecture to enhance transparency and traceability in complex, layered environments. It uses structured callbacks for both the IDE and DOE layers, providing verification engineers with precise control over specific points in the data flow, which is crucial for efficient debugging.

*IDE Layer Callbacks***:** Callbacks are implemented to track transaction flow in transmit (TX) and receive (RX) directions, enabling monitoring of entry and exit points in transaction queues.

DOE Callbacks: Additional callbacks trace the start and end points of request and response transactions in the DOE layer, helping to pinpoint specific stages within the data flow.

This targeted approach enables precise monitoring of transaction flow, faster issue identification, and efficient protocol validation with minimal overhead. For Example

Precise Monitoring includes that the TX queue entry callback captures when the problematic transaction is queued, logging all relevant details (e.g., encryption key ID, TLP type, and length). If the transaction exits the TX queue without issues, the callback verifies that encryption and processing were successful. Efficient Debug involves correlating logs from the IDE and DOE callbacks, engineers identify that the issue occurs when TLPs with a specific Stream ID are incorrectly handled at the DOE layer. This pinpointing avoids manual inspection of unrelated layers, reducing debug time. Finally by Manually introducing error scenarios, such as injecting corrupted TLPs, mismatched encryption keys, or invalid routing IDs, provides additional insights into the system's fault tolerance. By combining modular debug hooks with targeted error injections, engineers can simulate and validate the system's responses to various fault conditions, ensuring robustness and compliance with the protocol specifications.



```
TL_IDE_TX_queue_enter TLP [CplD]
TL_IDE_TX_queue_exit TLP [CplD]
TL_to_DL TLP [CplD]
TL_TX_packet TLP [CplD]
TL_IDE_RX_queue_enter TLP [CplD]
TL_IDE_RX_queue_exit TLP [CplD]
```

IDE Layer callback for Error Injection or can be used as Debug hooks

Figure 9. Callback Hooks for IDE Functionality

Technique 5: Flexible Verification Methods for IDE Functionality

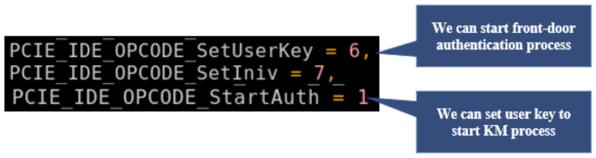
This technique provides flexible approaches to verify IDE functionality, especially when the Device Under Test (DUT) lacks native support for authentication and key management. By bypassing certain prerequisites, these methods enable comprehensive verification under constrained conditions.

Approach 1: Authentication Bypass

The first approach bypasses the authentication process, allowing verification to proceed even when the DUT is not fully equipped for authentication. Here, the VIP automatically initiates a "front-door" authentication process using an opcode, enabling the VIP to manage the entire authentication flow independently. This approach is particularly beneficial when authentication is either not ready in the DUT or managed by firmware or the operating system, providing a seamless workaround for testing purposes.

Approach 2: Backdoor Key Management Solution

The second approach introduces a backdoor method to bypass the standard Key Management protocol. Instead of following the conventional KM process, this method directly sets the IDE key, allowing verification to proceed without requiring full KM support from the DUT. This backdoor approach is useful when KM processes are unavailable or handled externally.



```
PCIE_IDE_OPCODE_SetUserKey = 6,
PCIE_IDE_OPCODE_SetIniv = 7,
PCIE_IDE_OPCODE_StartAuth = 1
```

We can start front-door authentication process
We can set user key to start KM process

Figure 10: Opcode to Bypass Authentication and Key Management Mechanisms

By adapting these five IDE techniques, verification engineers can address diverse security needs across IPs and SoCs. These techniques enhance flexibility, modularity, and traceability, providing a robust toolkit for verifying security protocols in various high-stakes applications, ultimately strengthening security resilience in SoC designs.

V. Challenges and Vulnerabilities

G. Verification Blind Spots in IDE Protocols:

In the process of verifying high-speed protocols like PCIe and CXL, several challenges—referred to here as blind spots—can arise, posing risks to data integrity, security, and performance. These blind spots can complicate verification efforts, especially when dealing with complex configurations and stringent security requirements.

For clarity and focus, we have categorized these blind spots into five key areas: Error Detection and Classification Challenges, State and Configuration Transitions, Field-Specific Error Injection, Resource Management, and Protocol-Specific Handling and Precedence Rules.

The following sections outline each category of blind spots, provide practical examples, and link them to specific techniques that enhance the robustness and efficiency of the verification process.

1) Error Detection and Classification Challenges

a) Behavior on Invalid Routing ID Corruption

Description:

The specification may not clearly define how the RX side handles invalid routing IDs, leading to potential ambiguity in expected behavior. Undefined RX handling can result in unhandled exceptions or security risks. For Example, If an invalid routing ID is introduced during data transfer, the RX side may encounter unexpected behavior like should it be discarded or consumed. Testing should confirm that RX handling is secure and consistent.

Mitigation Strategy:

Introduce test cases that inject invalid routing IDs to verify secure operation.

b) Plaintext CRC Testing

Description:

Unlike CXL.io/PCIe IDE, PCRC (Plaintext CRC) is not transmitted over the link for CXL.cachemem. Verifying this optional feature requires ensuring that encryption and decryption handle PCRC inclusion and exclusion accurately.

MitigationStrategy:

Include random settings for PCRC in test vectors. Verify the encryption and decryption processes under different scenarios to ensure the correctness of the Plaintext CRC feature, even when it's optional.

2) State and Configuration Transitions

c) Handling Pending Packets During Secure-to-Insecure Transition

Description:

Transitioning from secure to insecure states poses risks, particularly with pending TLPs and non-IDE TLPs. Pending packets during such transitions can lead to misrouted errors if not properly managed, while non-IDE TLPs processed in secure mode could compromise security. For instance, a secure-to-insecure state change might inadvertently allow non-compliant TLPs or misroute pending packets.

Mitigation Strategy:

Verification should ensure that pending TLPs are cleared from queues before state transitions. Additionally, tests must confirm that only IDE-compliant TLPs are processed in secure states, with non-IDE TLPs being securely discarded.

d) Credit Management in Error Scenarios

Description:

Incorrect handling of credits during error scenarios can result in mismatched credits, causing protocol deadlocks. Credit mismatches can affect communication, leading to performance issues. For example, An IDE error may discard packets without consuming credits, leading to communication stalls. Verification should confirm credits are correctly managed in error conditions.

Mitigation Strategy:

Track credit usage to ensure they are consumed correctly before discarding packets.

3) Field-Specific Error Injection

e) Precision and Accuracy in Error Injection and Classification

Description:

Injecting specific errors like PCRC or MAC corruption must be precise to prevent unintended effects on adjacent fields, which could complicate error analysis. Moreover, overlapping errors such as Misrouted TLP and IDE Check failures require careful classification to avoid misdiagnosis. For example, improper injection may corrupt multiple fields, triggering incorrect error classifications and complicating fault isolation.

Mitigation Strategy:

Use fine-grained error injection techniques while ensuring that error detection and classification mechanisms are robust enough to distinguish between closely related errors.

4) Resource Management

f) RID Management with Limited Functionality

Description:

Limiting Routing ID (RID) management may lead to conflicts in selective stream IDE setups. RID conflicts can affect system performance and protocol integrity. For Example, With only one RID range available for selective stream IDE, address mapping conflicts could arise, affecting data flow. Verification should test for RID conflicts.

Mitigation Strategy:

Simulate limited functionality scenarios to test for address mapping conflicts.

g) Efficient Management of Association Address Registers

Description:

The Selective IDE Stream Register Block can repeat up to 255 times, with each block containing IDE Address Association Registers that can also repeat independently. Without a structured method for addressing these blocks, maintaining consistent access and ensuring correct register configuration becomes difficult, especially when handling multiple streams

Mitigation Strategy:

By assigning offsets to these tasks, each task can directly target a particular register block, allowing precise control over the IDE Address Association Register within each Selective IDE Stream Register Block.

5) ***Protocol-Specific Handling and Precedence Rules***

h) Handling Precedence in Selective and Link IDE Streams

Description:

Verifying precedence rules between Selective and Link IDE Streams can be challenging due to ambiguities. Misinterpretation of precedence rules may lead to misrouted packets. For example, When both stream types are active, TLPs may be misrouted if precedence is unclear. Testing should confirm priority is followed.

Mitigation Strategy:

Develop scenarios to validate priority handling between Selective and Link IDE Streams.

i) Overflow and Underflow Handling in Counters

Description:

According to the specification, implementing overflow at the receiver's DUT end is impractical, as sending more than 2^{1024} TLPs to test overflow is unfeasible. However, underflow conditions are possible as per spec guidelines. To address this limitation, both the Device and VIP should maintain independent Rx and Sent counters on each side.

Mitigation Strategy:

The specification does not explicitly outline requirements for these counters but having them separately is essential to monitor and manage potential overflow and underflow scenarios accurately.

H. Addressing Blind Spots with Targeted Techniques

Each of these categories represents a distinct type of verification challenge. However, with the implementation of our proposed techniques—such as State Machines, Operation Codes, White Box Logging, Modular Debug Hooks, and Flexible Verification Methods—verification engineers can effectively address and resolve these critical scenarios.

TABLE II
WIDELY USED PROTOCOLS AND ITS SECURITY FRAMEWORKS

Technique	Blind Spots Addressed	Description and Mitigation Role
State Machine for IDE Capability Discovery	State and Configuration Transitions (Blind Spots c, d)	The State Machine approach manages complex state transitions, ensuring consistency across devices. It helps prevent miscommunication and protocol errors during state changes.
Operation Codes for Unified Security Operations	Field-Specific Error Injection (Blind Spot e)	Operation Codes streamline API management, allowing precise control over security functions. This modular approach ensures targeted error injections without unintended side effects.
White Box Logging for Transparent Debugging	Error Detection and Classification Challenges (Blind Spots a, b)	White Box Logging provides detailed visibility into internal processing steps, aiding in accurate error classification and minimizing ambiguity in error detection and debugging.
Modular Debug Hooks for Layered Architectures	Resource Management (Blind Spots f, g)	Modular Debug Hooks allow selective monitoring of specific stages, enhancing control over resources like RIDs and association registers, which optimizes performance and efficiency.
Flexible Verification Methods for IDE Functionality	Protocol-Specific Handling and Precedence Rules (Blind Spots h, i)	Flexible methods allow for testing in scenarios where full security support may be lacking. This approach helps verify consistent handling of non-IDE TLPs and counter behavior.

VI. RESULTS AND CONCLUSION

In this study, we systematically addressed critical verification blind spots within the IDE protocols by applying targeted techniques, each selected to tackle specific challenges identified during the verification process. Our methods encompassed managing complex state transitions, injecting precise field-specific errors, ensuring transparent debugging, optimizing resource management, and handling protocol-specific precedence rules. By implementing these techniques, we were able to achieve significant improvements in both the robustness and accuracy of IDE protocol verification. Few Bugs that were found:

Bug 1: Protocol Flit Transmission Between MAC Epochs

An issue was identified where protocol flits from the next MAC Epoch were transmitted before the TMAC flit, contrary to the specification. This misalignment between MAC Epochs can lead to encryption inconsistencies and compromise data integrity, posing significant security risks.

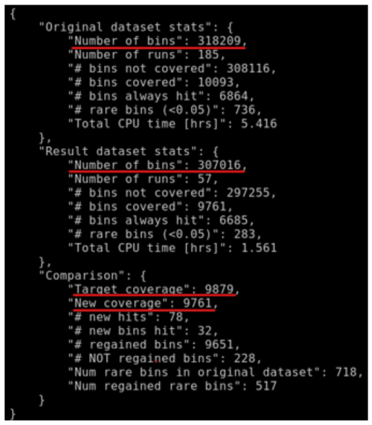
Bug 2: Mixing IDE and Non-IDE TLPs in the Same Flit

The handling of Selective Streams revealed challenges in detecting and managing mixed IDE and Non-IDE TLPs within the same flit. Although permitted by the specification, improper differentiation can compromise encryption integrity, cause key mismatches, and introduce security vulnerabilities, highlighting the need for enhanced validation mechanisms.

Bug 3: Re-Configuration of Selective Stream's Address Association Block Register

The Address Association Block Register was found to be non-reconfigurable after initial programming, limiting its adaptability for dynamic traffic allocation or error recovery. This restriction poses challenges in runtime flexibility and efficient resource utilization.

Our approach yielded a notable ~97% coverage in IDE verification, as shown in the ML-driven coverage regression snippet. This coverage level signifies a high degree of confidence in the protocol's security and integrity, confirming that critical blind spots have been effectively mitigated.



```
{
    "Original dataset stats": {
        "Number of bins": 318209,
        "Number of runs": 185,
        "# bins not covered": 308116,
        "# bins covered": 10093,
        "# bins always hit": 6864,
        "# rare bins (<0.05)": 736,
        "Total CPU time [hrs]": 5.416
    },
    "Result dataset stats": {
        "Number of bins": 307016,
        "Number of runs": 57,
        "# bins not covered": 297255,
        "# bins covered": 9761,
        "# bins always hit": 6685,
        "# rare bins (<0.05)": 283,
        "Total CPU time [hrs]": 1.561
    },
    "Comparison": {
        "Target coverage": 9879,
        "New coverage": 9761,
        "# new hits": 78,
        "# new bins hit": 32,
        "# regained bins": 9651,
        "# NOT regained bins": 228,
        "Num rare bins in original dataset": 718,
        "Num regained rare bins": 517
    }
}
```

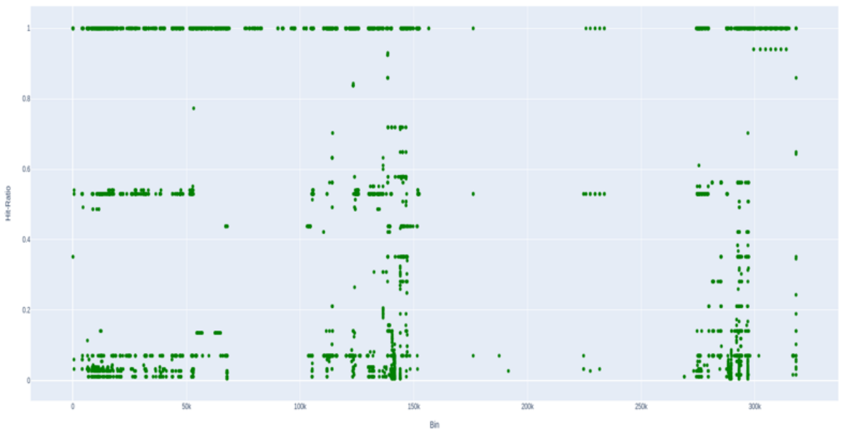


Figure 11. Coverage Results of PCIe IDE Regression

Ultimately, our methodology provides a practical and high-confidence verification framework that reduces manual intervention, improves debugging efficiency, and ensures that IDE protocol implementations meet stringent security and reliability standards. This outcome is vital for high-performance applications, such as data centers and AI-driven systems, where robust data integrity and security are paramount. These findings and methodologies, while demonstrated with Cadence Verification IP, extend beyond the realm of IP verification.

Additionally, while this work primarily focuses on PCIe and CXL, the proposed IDE techniques have the potential to be extended to other high-speed protocols like HDMI and MIPI, though each may present unique challenges in terms of encryption and data integrity specific to their operational environments.

Future IDE projects could benefit from implementing protocol-agnostic verification frameworks, designed to validate IDE mechanisms across multiple high-speed protocols like PCIe, CXL, and emerging standards. This approach would ensure that security verification processes are adaptable and reusable, reducing the time and effort needed for each new protocol. Additionally, introducing adaptive error recovery mechanisms within IDE protocols could enhance system resilience. For instance, dynamically rerouting secure data streams during IDE failures or encrypting fallback channels in real-time could mitigate the risks of partial protocol failures. Another innovative approach is real-time IDE stress testing, which integrates workload simulation with live traffic scenarios to assess the robustness of encryption, key management, and error detection mechanisms under extreme conditions. These ideas focus on process improvements and system-level robustness, aligning with the practical needs of designers and verification engineers.

I. References

1. PCI Express® Base Specification Revision 6.1

2. Compute Express LinkTM (CXLTM) Revision 3.1

3.CXL IDE https://computeexpresslink.org/webinars/compute-express-link-cxl-link-level-integrity-and-data-encryption-cxl-ide-333/

4. Application Example https://arxiv.org/html/2410.13031v1