2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Uncovering Hardware Vulnerabilities: Formal Verification for Security-Focused Negative Testing

Vedprakash Mishra, Intel Corporation, Bengaluru, India (vedprakash.mishra@intel.com)

***Abstract*— Ensuring robust hardware security in increasingly complex and integrated System-on-Chip (SoC) designs require systematic, formal, and scalable methodologies. This paper presents a category-driven framework that unifies the Confidentiality-Integrity-Availability (CIA) triad with the rigor of formal verification, integrating both Security Path Verification (SPV) and Formal Property Verification (FPV). We detail a ten-category hardware security classification scheme aligned to Common Weakness Enumeration (CWE) and demonstrate how security properties are systematically derived from this taxonomy. By employing CWE-driven property generation, strategic abstraction, relaxed constraint handling, and staged convergence, our methodology delivers comprehensive, reusable, and early-stage security verification. This approach enables efficient identification and mitigation of vulnerabilities, not only for confidentiality, but equally for integrity and availability, shifting hardware security left in the design lifecycle. The presented framework is broadly reusable, extensible, and provides actionable pathways for rigorous hardware security analysis in modern SoC environments.**

Keywords*— Hardware security, Formal verification, CIA Triad, Security Path Verification (SPV), Formal Property Verification (FPV), Common Weakness Enumeration (CWE), System-on-Chip (SoC) security, Security property generation, Abstraction and convergence, Vulnerability classification.

I. Introduction

System-on-Chip (SoC) designs have evolved dramatically in scale and complexity, integrating diverse processing cores, accelerators, memory subsystems, and extensive peripheral interfaces on a single silicon die. This heterogeneous integration, while enabling unprecedented performance and functionality, has simultaneously introduced a broad and complex attack surface that adversaries can exploit to compromise hardware security. Unlike software vulnerabilities, hardware design flaws can be deeply embedded and propagate to undermine trust in critical system assets—such as cryptographic keys, control and status registers (CSRs), privileged memory regions, and communication channels—potentially without detection at higher layers.

The hardware security challenge is further exacerbated by the increasing presence of untrusted or semi-trusted entities within the SoC fabric, including diverse intellectual property (IP) blocks, third-party subsystems, and multi-tenant frameworks. These components often operate with varying privilege levels and access capabilities, making it essential to rigorously enforce security policies at the hardware boundary to prevent unauthorized data disclosure, tampering, or service disruption. Failure to do so can lead to severe consequences including leakage of sensitive information, unauthorized control or manipulation of system functions, and denial-of-service (DoS) conditions affecting system availability.

CONFIDENTIALTY
HARDWARE SECURITY
INTEGRITY
AVAILABILITY
Figure 1: Illustration of CIA Triad

Traditional security verification approaches relying on simulation or emulation face fundamental limitations for SoC hardware security. Their inherent dependence on a finite set of execution scenarios and test vectors means rare corner cases or subtle combinational attacks often remain undetected. Moreover, hardware security properties encompass multiple dimensions—specifically confidentiality, integrity, and availability (CIA, shown in Figure 1); each requiring different reasoning paradigms. For instance, ensuring confidentiality demands rigorous proof that sensitive data cannot leak through any possible execution path, while integrity mandates validation that unauthorized alterations or corruptions cannot occur, and availability requires guarantees that the system cannot be coerced into unresponsive or failed states.

Formal verification offers a mathematically rigorous path to proving the absence of hardware vulnerabilities; by exhaustively traversing all possible system states, it provides stronger guarantees than simulation-based approaches. This work aims to operationalize hardware security verification by:

- Structuring security analysis along the Confidentiality-Integrity-Availability (CIA) triad,
- Building verification properties directly from the industry-standard Common Weakness Enumeration (CWE) taxonomy,
- Organizing properties into ten hardware-specific security categories, each connected to real attack surfaces and CWE entries,
- Strategically applying SPV and FPV to maximize coverage and detection of both data-leak and service-manipulation issues, and
- Employing abstraction, constraint relaxation, and convergence handling to support practical, reusable, and scalable verification setups.

Our results demonstrate that this methodology can systematically uncover significant security flaws, including subtle manipulation of service responses, unauthorized asset accesses, and pipeline hangs susceptible to denial-of-service—all of which traditional methods may overlook. Overall, this paper provides a detailed exposition of the method, tool application rationale, property engineering techniques, and illustrative case studies, advancing the state of hardware security verification such that SoC designs can achieve higher assurance and resilience against evolving attacker strategies.

II. BACKGROUND AND KEY CONCEPTS

A. *Formal Verification for Hardware Security*

Formal verification setup combines constraints, abstractions, checkers/covers, and reference models to analyze the design's behavior and verify its properties as shown in Figure 2. These components are interconnected and influence one another. Constraints restrict the exploration space and shape the analysis by defining valid inputs and system behaviors. Abstractions help manage complexity by simplifying the design representation while preserving key properties. Checkers or coverage properties define the security requirements that the design must meet, and their evaluation determines the verification outcome. The reference model serves as a benchmark for comparison and evaluation. Together, these components form an integrated approach to formal verification, enabling comprehensive analysis of the design's security properties

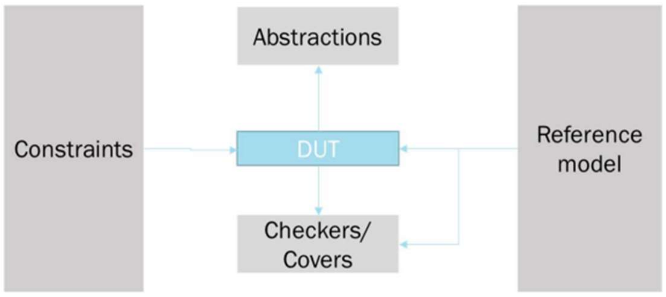


Figure 2: Components of FV setup

Constraints in formal verification (FV) critically shape the input and state space explored during analysis. While functional FV typically assumes protocol-compliant inputs, relying solely on these legal assumptions(constraints) risks missing security vulnerabilities that arise from malicious or unexpected behaviors. For comprehensive hardware security verification, constraints must be carefully relaxed to include both compliant and non-compliant scenarios. This broader input coverage enables early detection of corner-case vulnerabilities and enhances resilience against unknown attacks. Alongside updating abstractions and reference models, loosening constraints allows security-focused FV to explore a wider threat space, improving assurance beyond traditional functional correctness

B. *CIA Triad in Hardware Security*

In the context of hardware security, formal verification plays a crucial role in verifying a set of security properties. These properties encompass three aspects that are as follows:

- Confidentiality: Restricts access to sensitive hardware assets so only authorized entities can retrieve them.
- Integrity: Ensures data and transaction consistency and correctness; prevents unauthorized alteration or corruption.
- Availability: Defends against denial-of-service and ensures timely access and operation of secure system.

C. *CWE and security tool*

CWE is a community-curated catalog of known security weaknesses. Mapping hardware verification properties to CWE IDs (e.g., CWE-1262: Missing Authorization for Sensitive Resource) ensures that coverage is systematic, universal, and industrially relevant. To rigorously detect and mitigate these CWE-classified vulnerabilities, formal hardware security verification tools are employed:

- Security Path Verification (SPV): Formally proves (or disproves) that information cannot flow from source assets to unauthorized destinations, thus specializing in confidentiality checks.
- Formal Property Verification (FPV): Asserts and exhaustively proves behavioral properties (using System Verilog Assertions or similar), suitable for modeling and verifying complex requirements for integrity and availability.

III. THREE-PHASE SECURITY METHODOLOGY

Three-phased methodology is essential for comprehensive hardware security verification using formal methods. This structured approach involves distinct but interrelated phases, each addressing critical aspects of deploying formal techniques for rigorous security assurance. Below is a brief overview of the methodology, followed by a deeper dive into each phase:

Overview: Three-Pronged Methodology

- Planning: Threat Modeling and Property Generation
- Properties Coding: CIA-Driven Assertions and Checks
- Abstraction & Convergence: Optimizing Coverage and Scalability

A. *Planning: Threat Modeling and Property Generation*

The methodology begins with a thorough security planning phase centered on threat modeling. Here, engineers catalogue all critical hardware assets and interfaces, analyze potential attack surfaces, and systematically reference industry-standard taxonomies such as the Common Weakness Enumeration (CWE). This enables them to identify relevant vulnerability classes (e.g., unauthorized access, improper reset handling) and derive a concrete set of security properties that target these known threats. By leveraging the CWE database and using generic and targeted security-centric questions, the process ensures thorough coverage and that no area is overlooked during planning.

Categories were defined by mapping critical hardware asset flows to documented vulnerability classes in the CWE database, reflecting major domains in hardware security. The goal: ensure complete coverage of all asset classes, privilege levels, and attack surfaces common in SoCs. The 10 categories stem from the intersection of security-critical functionality and real-world attack surfaces, as reflected in CWE10

Below is a brief description of representative CWEs, vulnerabilities, and types of checks developed for each category:

#	Category	Criteria	Example Vulnerability & CWE	Title	Properties & Verification Approach
1	Manufacturing & Lifecycle	Fault tolerance, decommission protocols	CWE-1248 (Improper Physical Access Control)	Lack of protection against fault attacks, e.g.,	FPV: Properties ensuring secure decommission, error handling; abstraction of

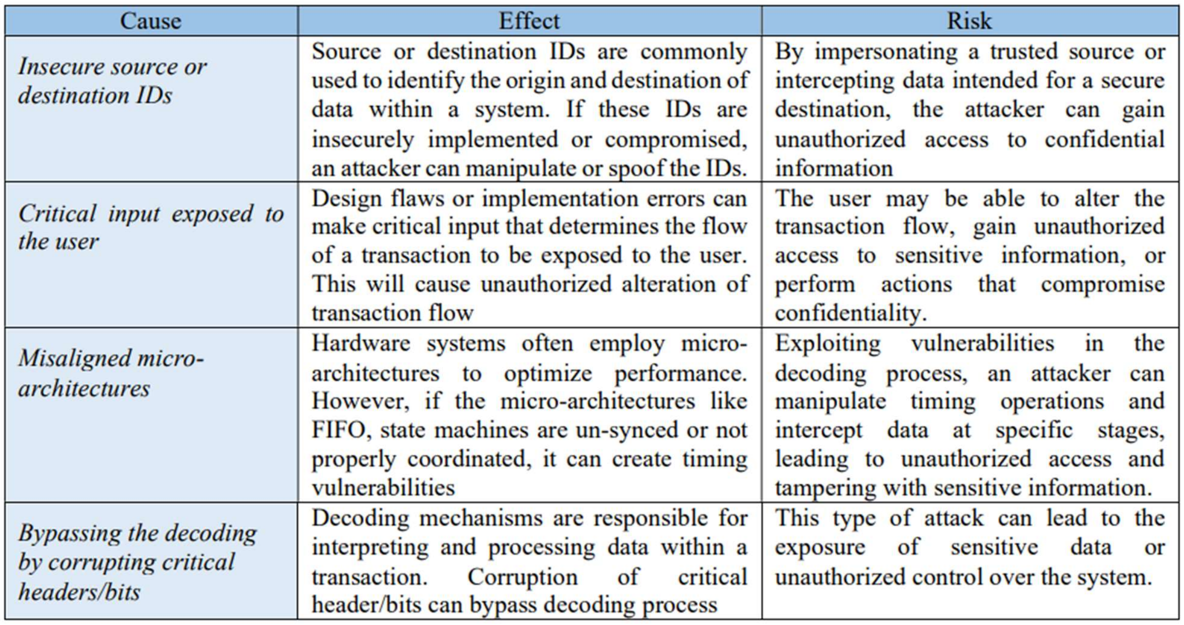
				missing scrubbing of sensitive data after use	manufacturing test flows
2	Security Flows	Design blocks accessing protected assets	CWE-1190 (Improper Access Control of Asset)	Unauthorized sub-IP asset access	FPV: IP block asset access, properties for privilege separation, path checks using SPV for all accessible assets
3	Integration Issues	Reset hierarchies, SoC-level operation	CWE-1276 (Improper Reset Handling)	Resets causing security bypasses or loss of protection	FPV: Reset abstraction, properties for reset domain security, convergence on all reset transitions
4	Privilege Separation & Access Control	Multiple asset owners/interfaces, policy	CWE-1262 (Missing Authorization for Sensitive Resource)	Missing enforcement on multi-interface/register access	SPV+FPV: Properties on access policies, coverpoints for interface crossing, check asset privilege on every access
5	General Circuit/Logic	Reserved/sticky/lock bits, FSMs	CWE-1209 (Missing Reserved Bit Protection)	Unauthorized writing of reserved/sticky bits or FSM bypass	FPV: Assertion/check properties for lock bit behavior, FSM state abstraction, deadlock/livelock FPV checks
6	Core & Compute	Security-relevant compute logic	CWE-1252 (Improper Handling of Sensitive Data in Compute Element)	Compute units mishandling security state	FPV: Properties on secure compute element operation, abstraction of microcode and privileged operations
7	Memory & Storage	Protected memory/register regions	CWE-226 (Sensitive Data in Memory Not Properly Cleared)	Sensitive keys or data lingering in accessible memory	SPV: Proves lack of leaks, FPV: Properties for zeroing mechanisms, coverpoints for memory retirement
8	Peripherals, On-Chip Networks	Bus/fabric/endpoint relating to data flow	CWE-1315 (Incorrect Bus Master Permissions)	Over-permissive endpoint registers, bus misconfigurations	FPV: Transaction correctness, bus permissions, abstraction of bus arbitration scenarios
9	Crypto	Side-channels, crypto asset residence	CWE-1319 (Insufficient Side Channel Protection)	Observable crypto operation or	FPV: Information flow checks, cover properties for side-channel resilience

				exposure of key registers	
10	Debug/Test	Debug interface/path control	CWE-1234 (Unrestricted Debug Access)	Asset leakage during debug/testing	SPV: Properties sealing off access in production, FPV: Assertion of debug privilege enforcement

B. Properties Coding: CIA-Driven Assertions and Checks

With complete planning, the next phase is the implementation of verification properties—both as Security Path Verification (SPV) and Formal Property Verification (FPV) assertions. These are explicitly aligned to the Confidentiality, Integrity, and Availability (CIA) triad.

1. Confidentiality: : Secure access of Assets

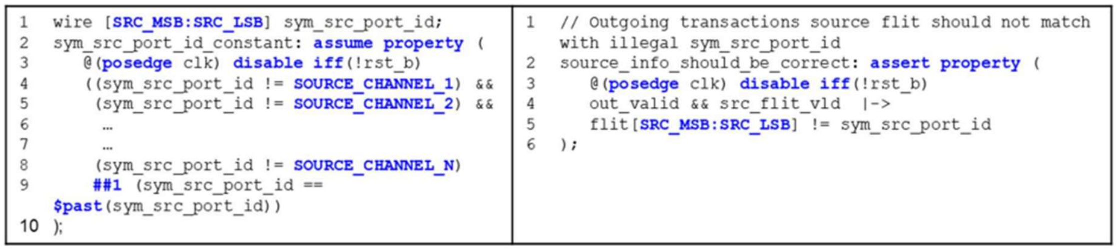


Cause	Effect	Risk
Insecure source or destination IDs	Source or destination IDs are commonly used to identify the origin and destination of data within a system. If these IDs are insecurely implemented or compromised, an attacker can manipulate or spoof the IDs.	By impersonating a trusted source or intercepting data intended for a secure destination, the attacker can gain unauthorized access to confidential information
Critical input exposed to the user	Design flaws or implementation errors can make critical input that determines the flow of a transaction to be exposed to the user. This will cause unauthorized alteration of transaction flow	The user may be able to alter the transaction flow, gain unauthorized access to sensitive information, or perform actions that compromise confidentiality.
Misaligned micro-architectures	Hardware systems often employ micro-architectures to optimize performance. However, if the micro-architectures like FIFO, state machines are un-synced or not properly coordinated, it can create timing vulnerabilities	Exploiting vulnerabilities in the decoding process, an attacker can manipulate timing operations and intercept data at specific stages, leading to unauthorized access and tampering with sensitive information.
Bypassing the decoding by corrupting critical headers/bits	Decoding mechanisms are responsible for interpreting and processing data within a transaction. Corruption of critical header/bits can bypass decoding process	This type of attack can lead to the exposure of sensitive data or unauthorized control over the system.

Figure 3: Causes of confidentiality breach

Confidentiality breaches can occur in various ways within a hardware system as shown in Figure 3. To mitigate these risks as shown in Figure 3 and protect the confidentiality of sensitive information, identification and role attributes for initiators and responders are crucial in a hardware system. These attributes uniquely identify entities and specify their roles, enabling secure and controlled access. They help authenticate initiators, enforce authorization, and access control, establish secure communication channels, and facilitate accountability and auditing. By incorporating these attributes, hardware systems ensure that only authorized entities interact with resources, prevent unauthorized access, and maintain system confidentiality

Formal Property Verification (FPV) is highly beneficial for ensuring confidentiality in a hardware system. FPV enables the verification of properties related to attribute constraints, which play a crucial role in preserving confidentiality. The provided example demonstrates how FPV can be used to enforce confidentiality requirements by preventing transactions with incorrect attributes (e.g., source/destination IDs, etc) from being propagated.



```
wire [SRC_MSB:SRC_LSB] sym_src_port_id;
sym_src_port_id_constant: assume property (
   @(posedge clk) disable iff(!rst_b)
   ((sym_src_port_id != SOURCE_CHANNEL_1) &&
    (sym_src_port_id != SOURCE_CHANNEL_2) &&
    …
    …
    (sym_src_port_id != SOURCE_CHANNEL_N)
   ##1 (sym_src_port_id ==
$past(sym_src_port_id))
);
```

```
// Outgoing transactions source flit should not match
with illegal sym_src_port_id
source_info_should_be_correct: assert property (
   @(posedge clk) disable iff(!rst_b)
   out_valid && src_flit_vld |->
   flit[SRC_MSB:SRC_LSB] != sym_src_port_id
);
```

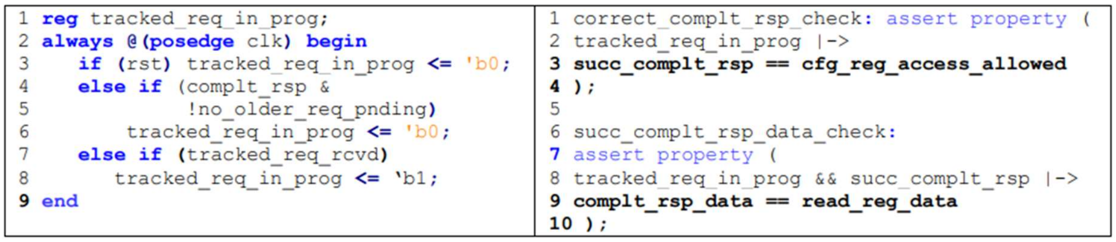
Figure 4: Pseudo code to assess Confidentiality

The property (on the left) ensures that the symbolic source port ID (sym_src_port_id, a free variable) does not match any of the allowed values of initiators (e.g., SOURCE_CHANNEL_1, SOURCE_CHANNEL_2, etc.). This way symbolic source port ID has anything but legal values. The second property verifies that if an outgoing transaction is valid (out_valid) and contains a source flit (src_flit_vld), the source flit's source port ID (flit[SRC_MSB:SRC_LSB]) should not match the sym_src_port_id. This property ensures that transactions with the incorrect source port ID are not allowed to pass through the output interface, thus maintaining the confidentiality of the system In addition to the provided example, other properties can be formulated to validate that filtered transactions, based on their attributes, remain unseen on the output interface. These properties help enforce data filtering and confidentiality requirements, ensuring that sensitive information is not leaked through the output interface

2. Integrity: Accurate and Consistent Service

Integrity ensures that data and system responses remain accurate and trustworthy. A breach occurs when an unauthorized entity alters the completion response of a legal non-posted transaction—changing a successful response to unsuccessful or vice versa. Such manipulation can disrupt service by causing false transaction failures or masking errors, leading to incorrect system actions, data corruption, unauthorized access, or operational failures. The impact ranges from financial loss and data integrity compromise to broader security breaches, emphasizing the critical need to protect transaction integrity.

To ensure the integrity of a system, it is essential to deploy both functional correctness properties and data correctness measures as shown in Figure 5. These two aspects work hand in hand to maintain the reliability and trustworthiness of the system's operations and data.



```
reg tracked_req_in_prog;
always @(posedge clk) begin
   if (rst) tracked_req_in_prog <= 'b0;
   else if (complt_rsp &
            !no_older_req_pnding)
       tracked_req_in_prog <= 'b0;
   else if (tracked_req_rcvd)
      tracked_req_in_prog <= 'b1;
end
```

```
correct_complt_rsp_check: assert property (
tracked_req_in_prog |->
succ_complt_rsp == cfg_reg_access_allowed
);

succ_complt_rsp_data_check:
assert property (
tracked_req_in_prog && succ_complt_rsp |->
complt_rsp_data == read_reg_data
);
```

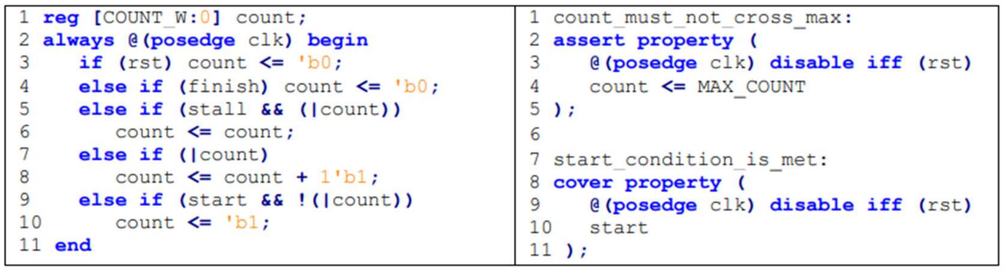
Figure 5: Pseudo code to assess Integrity

Functionality correctness checkers ensure that transactions comply with required rules—such as verifying the legality of read/write operations based on decoded attributes—so only valid requests are completed successfully. Data correctness checkers, meanwhile, confirm consistency by comparing actual read data with expected results. Together, these checks uphold system integrity by enforcing proper operation execution and validating that no incorrect or inconsistent data propagates through the system.

3. Availability: Hang free system (No DoS)

A pipeline hang can undermine system availability and expose hardware to denial-of-service (DoS) attacks by stalling data processing. When corrupted transactions occur, the IP should appropriately handle each case: discard or flag errors for invalid transactions; ensure non-posted transactions receive an unsuccessful completion response to prevent architectural hangs; and strictly enforce the one request–one response rule to maintain system integrity. Formal verification helps detect pipeline hang vulnerabilities and ensures that, even for illegal transactions, an appropriate completion response is sent within a bounded time, thereby safeguarding against DoS risks and maintaining reliable operation.

Formal verification can detect pipeline hangs by ensuring that, upon detecting illegal transactions, an unsuccessful response is sent within a finite number of cycles to prevent denial-of-service (DoS). This is implemented using a counter with start, stall, and finish conditions: it starts counting when illegal transactions are detected, stalls if dependencies block progress, and resets once an unsuccessful completion is sent. An assertion enforces that the counter never exceeds a predefined maximum (MAX_COUNT), which is based on the design's microarchitecture. A complementary cover property verifies that the start condition occurs, preventing vacuous passes. Together, these checks ensure timely handling of illegal transactions and maintain pipeline forward progress as shown in Figure 6.



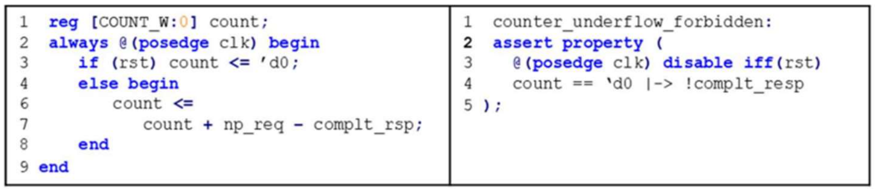
```
reg [COUNT_W:0] count;
always @(posedge clk) begin
   if (rst) count <= 'b0;
   else if (finish) count <= 'b0;
   else if (stall && (|count))
      count <= count;
   else if (|count)
      count <= count + 1'b1;
   else if (start && !(|count))
      count <= 'b1;
end
```

```
count_must_not_cross_max:
assert property (
   @(posedge clk) disable iff (rst)
   count <= MAX_COUNT
);

start_condition_is_met:
cover property (
   @(posedge clk) disable iff (rst)
   start
);
```

Figure 6: Pseudo code to assess Availability

To ensure "one request-one response" principle, it is important to ensure that the number of completion responses sent is never more than the number of non-posted requests received. This ensures that every completion response is associated with a valid transaction and prevents spurious completion responses from being generated as shown in Figure 7.



```
reg [COUNT_W:0] count;
always @(posedge clk) begin
   if (rst) count <= 'd0;
   else begin
      count <=
         count + np_req - complt_rsp;
   end
end
```

```
counter_underflow_forbidden:
assert property (
  @(posedge clk) disable iff(rst)
  count == 'd0 |-> !complt_resp
);
```

Figure 7: Pseudo code to assess Spuriousness

C. *Abstraction & Convergence: Optimizing Coverage and Scalability*

Abstraction is fundamental to achieving tractable and scalable formal verification of complex hardware systems. By distilling designs to essential behaviors relevant to security, abstraction allows the verification setup to efficiently explore vast state spaces and focus computational effort on meaningful threat scenarios. Several specialized abstraction techniques are deployed to optimize coverage and manage complexity:

- **Property Dissection**: Large or complex security assertions are systematically broken down into simpler, more focused properties. This modular approach allows for targeted analysis of specific conditions, makes debugging easier when a property fails, and helps pinpoint security weaknesses to precise design components.
- **Cycle and State Swamping**: Security verification often requires consideration of numerous system states and temporal behaviors. Cycle and state swamping abstracts (reduces) the number of cycles or states under consideration without losing coverage of critical transitions. For example, long initialization or idle sequences are collapsed, allowing the tool to prioritize security-sensitive transitions, like privilege escalation or data leakage events.

- **Reset Abstraction**: Since resets are frequent sources of integration and security flaws, reset abstraction simplifies and unifies the treatment of different reset types and hierarchies. It abstracts away the details of protocol-specific or multi-level resets while ensuring that the transitions relevant to security—such as unintended loss of asset protection upon reset—are thoroughly verified.

- **Parameter Reduction**: Designs often use multiple configuration parameters, leading to state space explosion in formal verification. Parameter reduction identifies non-security-critical parameters and fixes their values, while keeping parameters with security implications variable. This targeting reduces proof complexity, focuses verification on the aspects that impact security, and ensures that properties remain valid across relevant configuration options.

IV. Case Study: Manipulation of Service Vulnerability Detected via FPV

The targeted IP integrates multiple subsystem IPs within a System-on-Chip (SoC), managing interactions related to power and reset flows as well as configuration and static registers. It directs incoming transactions based on their source attributes to appropriate destinations such as power managers or CSR modules. A significant bug was detected through FPV involving manipulation of service: a legal non-posted transaction awaited a successful completion response but lacked credit. During this time, a corrupt posted transaction was received and correctly flagged with an error. When credits became available, the system asserted a completion response; however, the success indicator was incorrectly de-asserted due to the posted error presence. This flaw shows how corrupted transactions can erroneously affect the completion status of legitimate transactions, potentially disrupting system operation. FPV's exhaustive property checking was critical in uncovering this subtle security vulnerability by verifying expected behaviors and exposing violations.

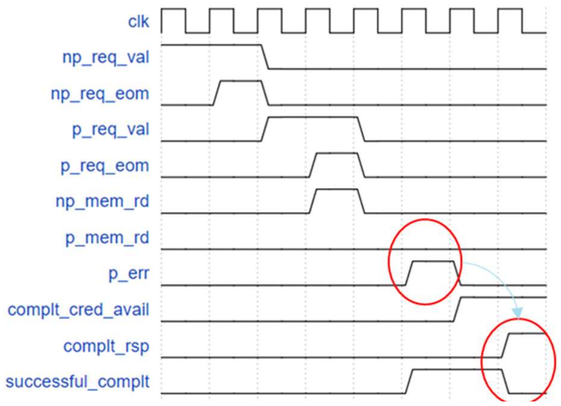


Figure 8: Illustration of manipulation of service

V. Result and Discussion

The application of our category-driven formal verification methodology, leveraging both Security Path Verification (SPV) and Formal Property Verification (FPV), produced significant findings in a complex, security-critical IP within a System-on-Chip (SoC) design. Over a six-month period, more than 40 security issues and the following key outcomes were achieved:

A. *Summary of Discovered Vulnerabilities*

The detected security bugs encompass all three pillars of the CIA triad, with notable findings in integrity and availability, areas traditionally difficult to cover thoroughly with path-based methods:

- Confidentiality: Unauthorized access to Control/Status Registers (CSRs) via security attribute bypass; Acceptance of malformed transactions due to missing attribute checks.

- Integrity: Manipulation of service where malformed transactions caused corruption of successful completion responses; Failure to raise errors on incorrect transaction flits, potentially allowing incorrect processing; Dropping of valid transactions due to interference from malformed transactions.

- Integrity: Manipulation of service where malformed transactions caused corruption of successful completion responses; Failure to raise errors on

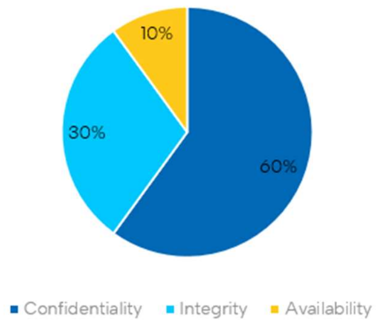


Figure 9: Bug Ratio across three pillars of CIA

incorrect transaction flits, potentially allowing incorrect processing; Dropping of valid transactions due to interference from malformed transactions.

B. Security Issues by Domain

While SPV excels at detecting confidentiality leaks by exhaustively proving absence of data flows, it lacks selectivity and expressiveness to fully detect integrity and availability issues. FPV demonstrated specific advantages by:

- Enabling detection of deadlock and livelock conditions within transaction pipelines.
- Verifying response correctness and error handling under adversarial inputs.
- Allowing reuse of functional correctness properties with relaxed constraints to catch security leaks without recreating the entire verification environment.

VI. CONCLUSION

This paper presents a systematic, category-driven formal verification methodology that integrates the Confidentiality-Integrity-Availability (CIA) triad with CWE-based property generation to comprehensively address hardware security challenges in complex SoC designs. By strategically leveraging both Security Path Verification (SPV) for confidentiality and Formal Property Verification (FPV) for integrity and availability, the approach enables scalable, reusable, and early-stage detection of a broad spectrum of vulnerabilities. The methodology's thoughtful use of abstraction, relaxed constraints, and convergence management facilitates thorough exploration of both compliant and malicious scenarios, uncovering subtle bugs that traditional methods often miss. Our results demonstrate significant gains in coverage and bug detection, highlighting the practical value of formally verifying all aspects of hardware security, not just confidentiality. Ultimately, this framework advances shift-left security by embedding robust, comprehensive protection into SoC design flows, enhancing trust and resilience against evolving threats.

REFERENCES

[1] Achutha KiranKumar, Erik Seligman, Tom Schubert. "Formal Verification – An Essential Toolkit for VLSI Design." 2023

[2] MITRE Corporation. "Common Weakness Enumeration (CWE) Database." Available: cwe.mitre.org

[3] Vedprakash Mishra, Carlston Lim, Zhi Feng Lee, Jian Zhong Wang, Anshul Jain, Achutha KiranKumar. "OIL check of PCIe with Formal Verification." DVCON, 2022

[4] Swaroop Bhunia, Mark Tehranipoor. "Hardware Security: A Hands-on Learning Approach." 20181