2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Formal Verification + CIA Triad: Winning Formula for Hardware Security

Vedprakash Mishra, Anshul Jain
Intel Corporation
vedprakash.mishra@intel.com, anshul.jain@intel.com

***Abstract*- The increasing complexity and interconnectedness of System-on-Chip (SOC) designs have introduced new challenges in ensuring the security of sensitive assets. SOC architectures involve multiple initiators/masters, such as cores, accelerators, and devices, which may not possess equal privileges. Untrustworthy sources can exploit vulnerabilities and bypass security mechanisms, posing a significant risk to the protection of assets distributed across the SOC. This paper explores the application of formal verification as an effective method to detect and mitigate security risks in hardware designs. By verifying that a design satisfies a set of security properties, including confidentiality, integrity, and availability, formal verification ensures that the hardware design is secure and free from vulnerabilities. Moreover, it facilitates a shift-left approach by preventing the introduction of potential vulnerabilities during subsequent design and development stages using three pillars of CIA Triad**

Keywords— Hardware security, CIA Triad, Formal verification, Shift-left approach, Sensitive asset, Security analysis

I. INTRODUCTION

System-on-Chips (SOCs) security has become an essential part of modern electronic world. In the past decade, the field of security has expanded to include the hardware domain. It has become evident that the security of SOCs can be compromised through various means, such as side-channel attacks, untrustworthy initiators/devices. These attacks target the underlying hardware components of a system and exploit vulnerabilities to gain unauthorized access or extract sensitive information.

Hardware-based attacks in SOCs designs aim to exploit vulnerabilities present in the system and steal assets such as configuration and status registers (CSRs), keys, and regions in memory. Such attacks have the potential to compromise the confidentiality, integrity, and availability of critical data and transactions. Traditional security mechanisms are often insufficient to address the evolving threat landscape, and new approaches are required to enhance the security of SOC designs.

This paper explores the application of formal verification (FV) as an effective method for detecting and mitigating security risks in hardware designs within SOC architectures. Formal verification can be used to verify that a design satisfies a set of security properties, such as confidentiality, integrity, and availability (as shown in Figure 1). FV is a powerful technique which can be thoroughly exploited in the field of hardware security to ensure the correctness and robustness of hardware designs with respect to security properties.

Through an examination of the role of formal verification in hardware security, this paper aims to highlight its significance in safeguarding of sensitive assets in SOC designs. Furthermore, it discusses the benefits of adopting a shift-left approach, where security verification is incorporated early in the design process to proactively prevent security vulnerabilities from causing functional and performance breakdown.

Figure 1: Illustration of CIA Triad

II. FORMAL PROPERTY VERIFICATION FOR HARDWARE SECURITY

Formal verification is a rigorous mathematical technique used to prove the correctness of a design with respect to its specifications. Unlike simulation-based methods that rely on test cases, formal verification exhaustively explores all possible input combinations and system states to ensure that the design functions as intended, making it idle candidate for security verification. By utilizing formal methods, designers can obtain mathematical certainty regarding the correctness and behavior of their hardware designs.

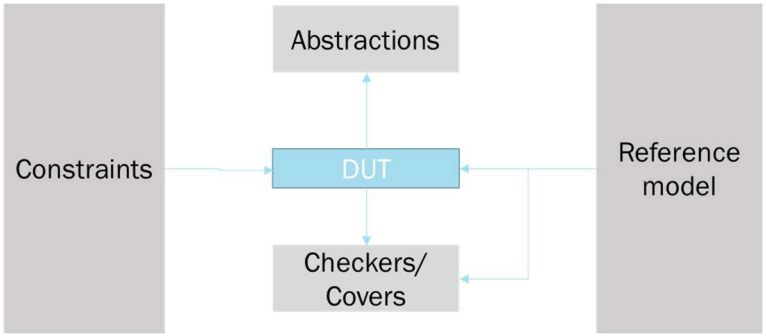

Figure 2: Components of FV setup and their interactions

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.

From hardware security verification perspective, constraints play a critical role. Functional FV setup can be re-used for security verification by carefully tweaking these constraints. Legal constraints that assume protocol-compliant inputs can introduce a risk of missing potential security issues. When designing security verification processes, it's crucial to consider a broad range of input scenarios, including both compliant and non-compliant inputs, to ensure comprehensive analysis. This broader analysis helps detect potential security weaknesses and vulnerabilities that may not be evident when focusing solely on compliant inputs. Loosening constraints enables the evaluation of the design's response to unexpected or malicious inputs, aiding in early vulnerability detection and proactive mitigation. By considering a wider range of input possibilities, formal verification helps uncover security risks that might be missed under stricter constraints. Example, header payload always accompanied with data payload may be protocol compliant, but it is a strict constraint from security point of view. It also enhances the design's resilience against unknown vulnerability attacks by uncovering and addressing them before they are exploited. Overall, loosening constraints in formal verification provides a more comprehensive and robust security verification process by exploring corner cases beyond legal assumptions. While updating abstractions and reference models for security verification, consider security-specific abstractions, incorporate threat modeling, define security properties and assertions.

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 mentioned in table below.

Confidentiality	Ensures that sensitive data is only accessible to authorized parties, preventing unauthorized access or leakage
Integrity	Guarantees that critical data or transactions remain unaltered and untampered during transportation or processing
Availability	Ensures that the system remains operational and accessible when needed, preventing denial-of-service or disruption attacks

The current scope of this paper is to talk about these three aspects of verification for security analysis.

A. *Confidentiality: Secure access of CSRs*

Formal verification techniques can be employed to ensure confidentiality in hardware designs. By modelling and analysing the flow of information within the system, formal verification can identify potential leakage points or unauthorized data accesses. Techniques such as information flow analysis and secure information flow tracking can be applied to verify that sensitive data remains protected and inaccessible to unauthorized entities.

Confidentiality breaches can occur in various ways within a hardware system. Let's explore different ways in which the confidentiality aspect of security can be breached based on table mentioned below:

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.

These are just a few examples of how the confidentiality aspect of security can be breached in a hardware system. It is important to note that these vulnerabilities should be addressed through robust security measures, such as secure hardware design practices, thorough testing, and techniques like encryption and access control. By identifying and mitigating these vulnerabilities (as shown in Figure 3), hardware systems can better protect the confidentiality of sensitive information and prevent unauthorized access or exposure.

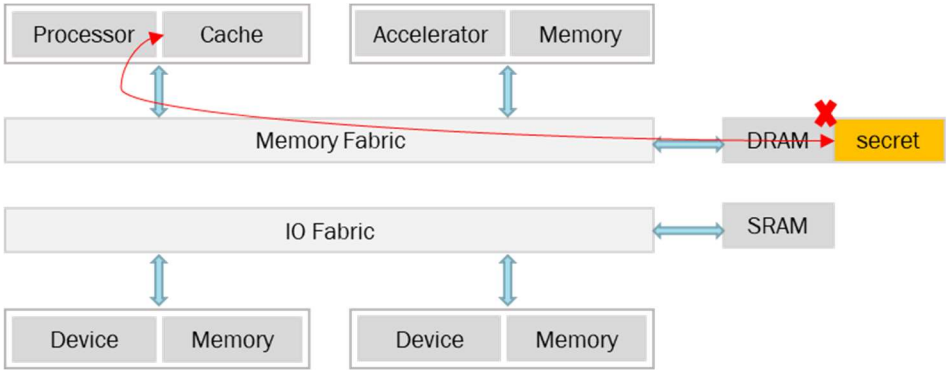

Figure 3: Illustration of a hardware attack

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.

```
1  wire [SRC_MSB:SRC_LSB] sym_src_port_id;
2  sym_src_port_id_constant: assume property (
3     @(posedge clk) disable iff(!rst_b)
4     ((sym_src_port_id != SOURCE_CHANNEL_1) &&
5      (sym_src_port_id != SOURCE_CHANNEL_2) &&
6       …
7       …
8      (sym_src_port_id != SOURCE_CHANNEL_N)
9      ##1 (sym_src_port_id ==
   $past(sym_src_port_id))
10 );
```

```
1  // Outgoing transactions source flit should not match
   with illegal sym_src_port_id
2  source_info_should_be_correct: assert property (
3     @(posedge clk) disable iff(!rst_b)
4     out_valid && src_flit_vld  |->
5     flit[SRC_MSB:SRC_LSB] != sym_src_port_id
6  );
```

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.

B. Integrity: Accurate and consistent Service

Integrity is a critical aspect of security that guarantees the accuracy, consistency, and trustworthiness of data and systems. Breaching the integrity of a system's security involves unauthorized changes to the completion response of a legal transaction, either from successful to unsuccessful or vice versa. Such breaches can have severe consequences for the system's reliability and the integrity of its data. If an intruder's illegal transaction manages to change the completion response of a legal non-posted transaction from successful to unsuccessful or vice versa, it can lead to the manipulation of the service being provided. This can have serious implications for the system's operation and the integrity of the data being processed.

In a typical scenario, a legal non-posted transaction expects a completion response indicating whether the transaction was successfully executed or not. This completion response serves as an assurance of the transaction's outcome and is crucial for maintaining the integrity of the system. However, if an intruder can maliciously alter the completion response of a legal transaction, it can introduce manipulation and compromise the service being provided.

If the completion response of a successful transaction is changed to unsuccessful, it can falsely indicate a failure or error in the transaction. This can lead to incorrect actions being taken based on the erroneous completion response, such as reprocessing the transaction or triggering unnecessary alarms. On the other hand, if the completion response of an unsuccessful transaction is altered to successful, it can mask the actual failure and provide a false sense of success. This can lead to incorrect data being processed, erroneous decisions being made, or unauthorized access being granted. The consequences of such manipulation of service can vary depending on the context and the specific system

involved. It can range from financial losses, operational disruptions, compromised data integrity, to potential security breaches.

To ensure the integrity of a system, it is essential to deploy both functional correctness properties and data correctness measures. 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
);
```

Functionality correctness checkers are responsible for verifying the legality and adherence to specific requirements of transactions before allowing them to be completed or generating a completion response. For example, in the case of checking read/write operations of a configuration register, a functionality correctness checker examines the attributes decoded from the transaction. It verifies whether the transaction is legal according to these attributes and determines if the read/write operation is permitted or required (cfg_reg_access_allowed). If the transaction meets these criteria, a successful completion response is sent; otherwise, a completion response indicating failure should be generated.

A data correctness checker plays a crucial role in ensuring the integrity and reliability of data within a system. This type of checker verifies the correctness and consistency of data across different components or modules, aiming to identify any inconsistencies or errors that may arise. When checking the correctness of data read from a configuration register, a data correctness checker compares the completion data sent with the read data. If the configuration register was read, the checker compares the two sets of data to ensure they match. A mismatch between the completion data and the read data indicates a potential error or inconsistency within the system.

By combining functional correctness properties and data correctness measures, the system can achieve a comprehensive approach to integrity. Functional correctness properties address the proper execution of transactions and operations, ensuring that they meet specified criteria and adhere to the system's intended behavior. Data correctness measures, on the other hand, validate the accuracy and consistency of the data, preventing incorrect or inconsistent information from compromising the system's integrity.

C. Availability: Hang free system (No DoS)

Hang in a pipeline can have an impact on the availability aspect of security. If a stage in the pipeline hangs or becomes stuck, it can cause a bottleneck in the pipeline and prevent the system from processing any further data. This can lead to a denial of service (DoS) attack, where an attacker intentionally causes the pipeline to hang or becomes stuck, causing the system to become unavailable.

When the IP detects that a transaction is corrupt, there are different possibilities for how it handles the situation. Let's explore each case:

- *IP Discards the Transaction or Raises an Error*:
  If the IP determines that the transaction is corrupt, it may choose to discard the transaction and free the pipeline without raising an error. This approach assumes that the transaction is invalid or not compliant with the expected protocol, and thus, it is safe to ignore it. However, it's important to note that discarding the transaction without raising an error may not be sufficient in all cases, especially when considering the overall correctness and integrity of the system.
- *Non-Posted Transaction Requires Completion Response*:
  In the case of a non-posted transaction, which typically requires a completion response, it becomes crucial to handle the situation appropriately. If the corrupted transaction is non-posted, it is necessary to send an

unsuccessful completion response. Failing to do so could result in the system architecture hanging, as completion responses are required for non-posted transactions to maintain the flow and integrity of the system.

- *Sending Incorrect Number of Completion Responses*:
  If the IP accidentally sends two completion responses instead of one for a transaction, it violates the "one request-one response" principle. This can lead to confusion and potential issues in the system. Ideally, the IP should only send a single completion response for each transaction to maintain the expected behavior and integrity of the system.

In summary, when dealing with corrupt transactions, it is essential for the IP to handle them appropriately based on the transaction type and compliance with the protocol. This includes discarding or raising an error for corrupt transactions, ensuring that non-posted transactions receive the required completion response, and adhering to the principle of one request-one response to maintain the expected behavior and integrity of the system

Formal verification can also be used to detect potential pipeline hangs and ensure the correctness of the pipeline design. One approach to avoid DoS by illegal transactions is to ensure that an unsuccessful completion is sent out within a finite number of cycles. This can help prevent potential security issues that may arise from allowing illegal transactions to continue processing in the system.

To ensure this safety correctness approach, the system can include a mechanism that detects illegal transactions as they are received and triggers a counter in presence of non-zero illegal transactions in system, stalls in case of external dependency and finishes when unsuccessful completion is sent. The maximum number of cycles “MAX_COUNT” for any transaction to go out is calculated based on the microarchitecture and specification of design, not based on the implementation.

To summarize, this approach consists of counter which has start, stall and finish conditions. Here’s the implementation of counter on left side of below table:

```
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
);
```

The associated assertion to ensure forward progress is in the above code snippet. The assertion ensures that counter never crosses maximum value. The property misses an antecedent so if because of any over-constraint the design does not receive start condition then for this case the assertion will not fail. To check this condition of vacuous pass, a cover property is added. The above cover property checks for start condition, whether the required input scenario is getting generated or not. The assertion and cover together make sure that required inputs are getting properly generated and getting propagated outside within finite cycles.

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 below code snippet.

```
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
);
```

III. CASE STUDY:

A. Design Details

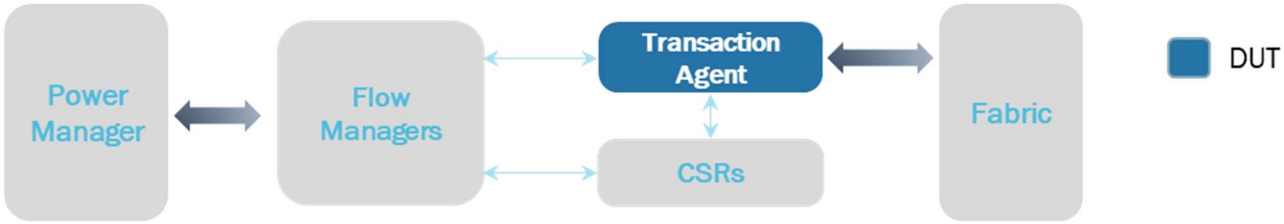

Figure 4: Illustration of working of targeted IP

The IP mentioned (as shown in Figure 4) is responsible for integrating different subsystem IPs within a System-on-Chip (SOC) and providing a unified modular interface. It manages interactions with these subsystem IPs for power and reset management purposes. When a transaction is received by the IP, it examines the source of the transaction and its attributes to determine the appropriate destination within the SOC. The IP distinguishes between different flows, such as power management flows, reset management flows, config registers, and static registers.

For power and reset management, the IP directs the transaction to the corresponding flow or power manager responsible for controlling the power states or handling reset operations of the subsystem IPs. This ensures that power and reset management within the SOC is properly coordinated and controlled. On the other hand, for transactions related to config registers and static registers, the IP routes the transaction to the appropriate modules (CSRs) responsible for handling these registers. This allows for the configuration and control of various settings within the subsystem IPs.

B. Bug Example

Figure 5 illustrates a specific bug related to the manipulation of service that was detected through Formal Property Verification (FPV). In this scenario, the legal non-posted transaction was supposed to get a successful completion response, but credit was unavailable. In a meantime, an illegal/corrupt posted transaction was received that was correctly decoded by the system and an error flag was raised due to corruption. Once the required credits became available, in the subsequent cycle, the completion (complt_rsp) was asserted. However, in this case, the successful field of the completion response got de-asserted due to the presence of the posted error (p_err).

This bug highlights a potential vulnerability in the system where a corrupted transaction can manipulate the service by causing errors in the completion response. The FPV technique was instrumental in detecting this bug by verifying the expected behaviour of the system and identifying any violations of specified properties.

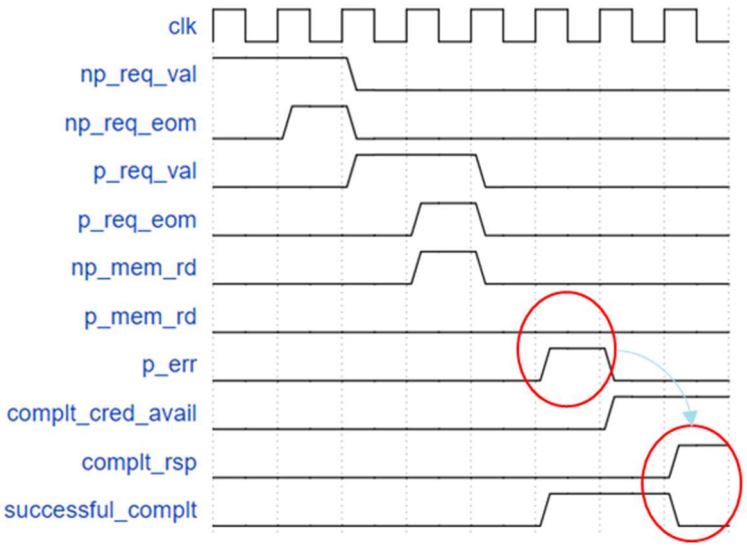

Figure 5: Illustration of manipulation of service

C. Timelines

Three months to catch first security failure might sound surprising but that time was devoted in creation of FV setup which helped us to catch functional issues in DUT as shown in Figure 6. Thereafter the same setup was utilized to catch security issues seeing the significance of the IP. We were able to find 3 DoS related bugs by 4th month and 2 CR access related issues after 2 quarters of effort and 1 service manipulation by the end of 3rd quarter.

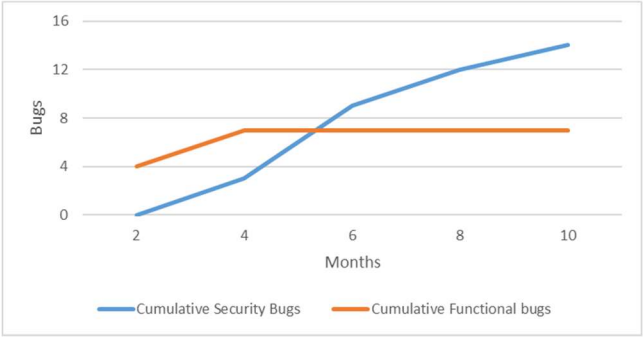

Figure 6: Timeline of bugs caught

D. Results

The Transaction Agent (TA) in the system was susceptible to several vulnerabilities that can compromise security and lead to various issues:

1. **Unsecure Access to CRs**: Transaction Agent (TA) claims transactions without security attributes, allows the attackers to modify/access Control/Status Registers (CSRs) (**2 vulnerabilities**)
2. **Denial of Service/ Hang**: TA fails to check header containing security attributes for malformed transaction, exposes TA to attackers for create hangs (**3 vulnerabilities**)
3. **Manipulation of Service**: TA unable to ignore malformed transactions appropriately, allows attackers to corrupt successful responses by injecting malformed transactions (**1 vulnerability**)

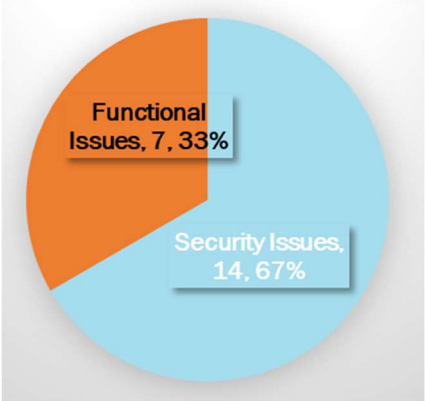

Figure 7: Illustration of total bugs caught

To summarize our result, we were able to identify and unearth issues related to both integrity and availability aspect of security along with ensuring confidentiality by deploying FPV on a security critical block. FPV provides distinct advantage over security path verification (SPV) when it comes to these two aspects (integrity and availability). In this security critical block, we were able to identify 21 issues out of which 7 were functional and 14 were security issues as shown in Figure 7.

IV. Conclusion

Usually when security of an IP is checked, the entire approach focuses on confidentiality; integrity and availability of system gets ignored. FPV provides this advantage to check them all at once. We have proven that instead of writing separate security properties, same functional properties could be exploited to verify security leaks by loosening the constraints. This way the formal setup created to functionally verify hardware components like processors, memory and input/output devices could be re-used to verify security leaks with minimal effort.

If an IP is security critical, then formal verification should be used to check not only functional and performance aspect but also security. Formal verification is an effective method for detecting and mitigating security risks in hardware designs. By verifying that a design satisfies a set of security/functional properties, formal verification can help ensure that a hardware design is secure and free from vulnerabilities and thus achieve shift-left.

References

[1] OIL check of PCIe with Formal Verification; DVCON 2022; Vedprakash Mishra, Carlston Lim, Zhi Feng Lee, Jian Zhong Wang, Anshul Jain and Achutha KiranKumar V M
[2] M, Achutha KiranKumar, Erik Seligman & Tom Schubert, Book on " Formal Verification – An Essential Toolkit for VLSI Design", 2023
[3] Swaroop Bhunia, Mark Tehranipoor , Book on " Hardware Security", 2018
[4] Jason Andress, Book on "The Basics of Information Security", 2018