

Continuity in Security: Leveraging LLM for Translating Security Properties Across Hardware Designs

Bulbul Ahmed, Sujan Kumar Saha, Jingbo Zhou, Sohrab Aftabjahani, Mark Tehranipoor, and Farimah Farahmandi



Why Security Verification?

To ensure the protection of security assets in the SoC

Asset

An entity in the SoC that must be protected to prevent the violation of Confidentiality, Integrity or Availability



SoC is Everywhere

Example of Assets

- On-chip device keys
- On-device sensitive data
- Device configuration
- Manufacturer Firmware
- Communication credentials
- Random number or entropy

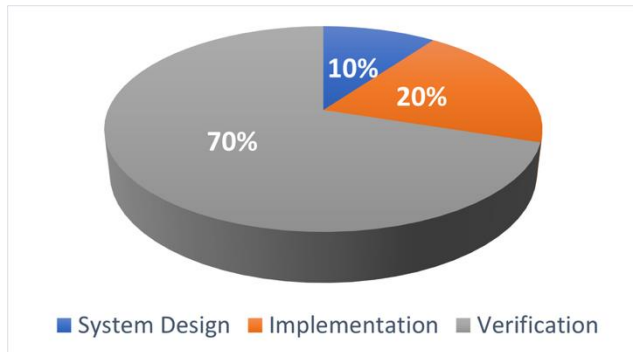
- Why is it called Formal?
 - Mathematic analysis of the space of behavior of the design
- Reasoning based on manipulation of formulas
- As opposed to simulation in which
 - Random set of stimulus only tests a tiny space of design behavior

Formal looks at the full space of possible design behavior

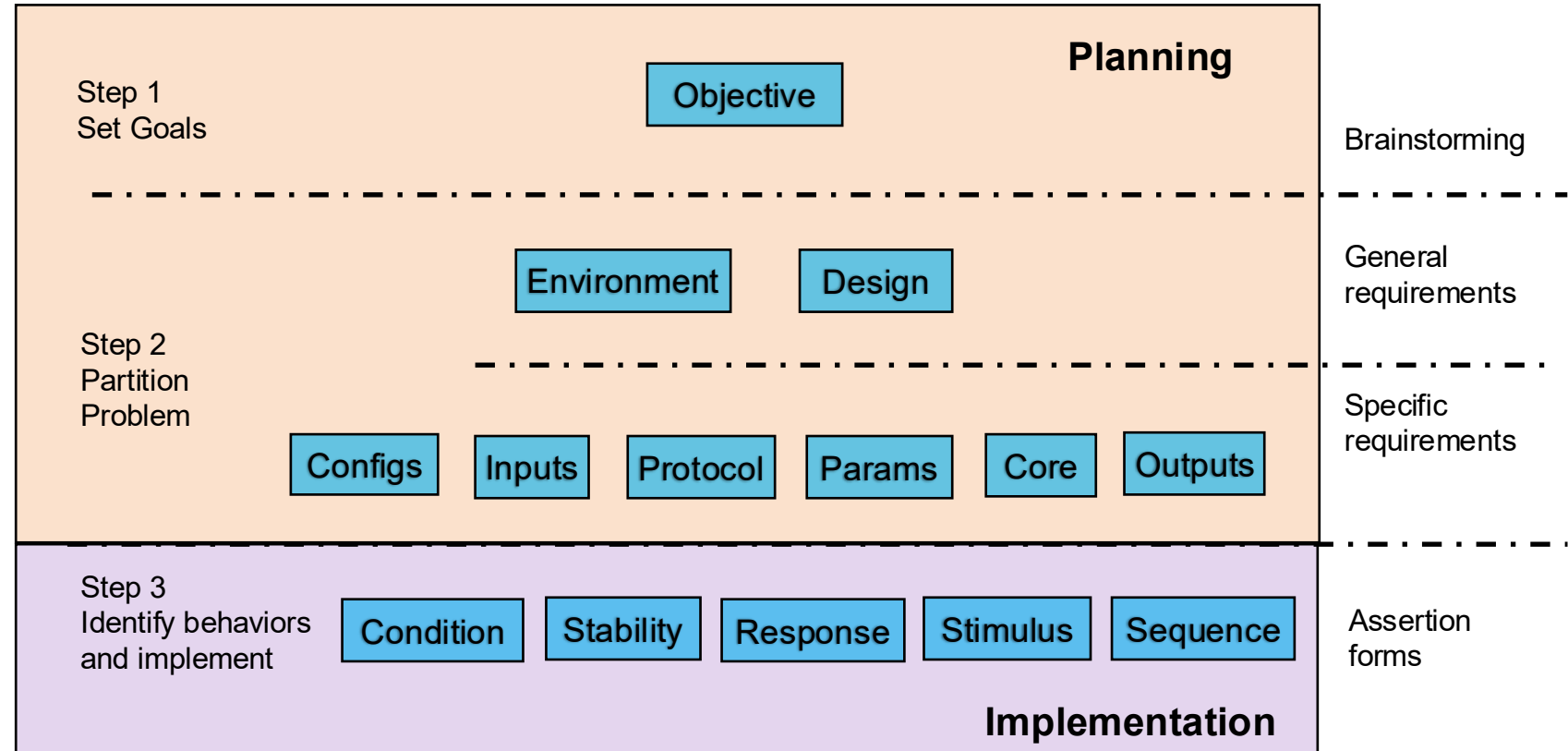


Property Development Process

- Define goals
- Categorize properties
- Specific requirement development
- Property implementation

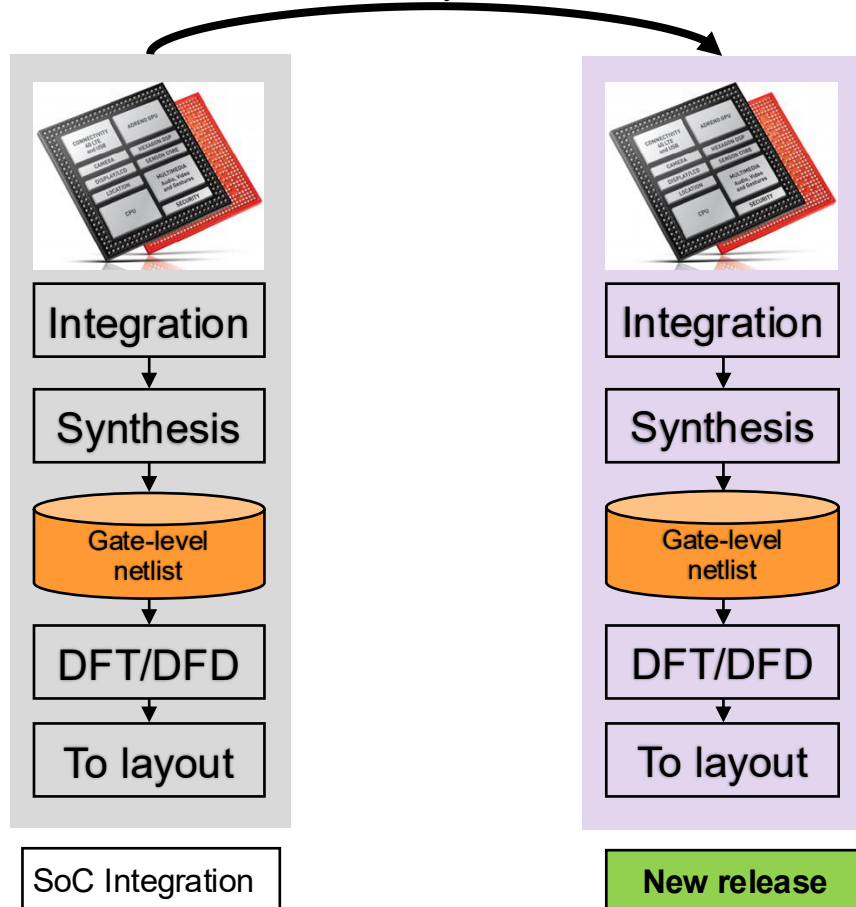


Relative time and effort at the different stages of the design-flow



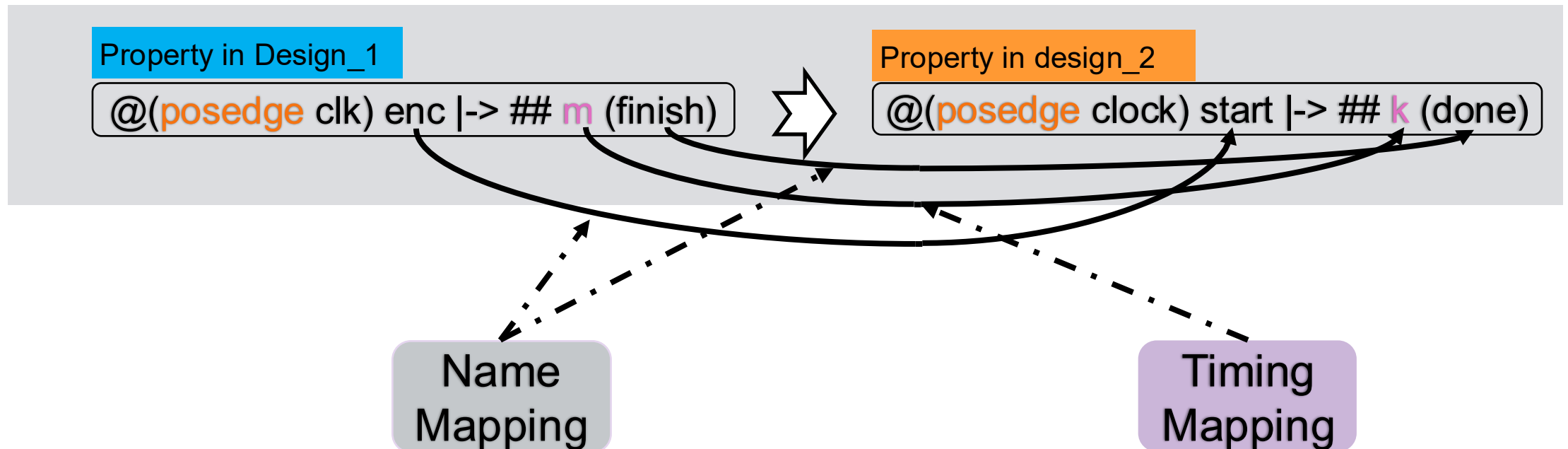
Transition Between Design Versions

- Security continuity across design version
- Verification reuse across design version
- New vulnerability identification



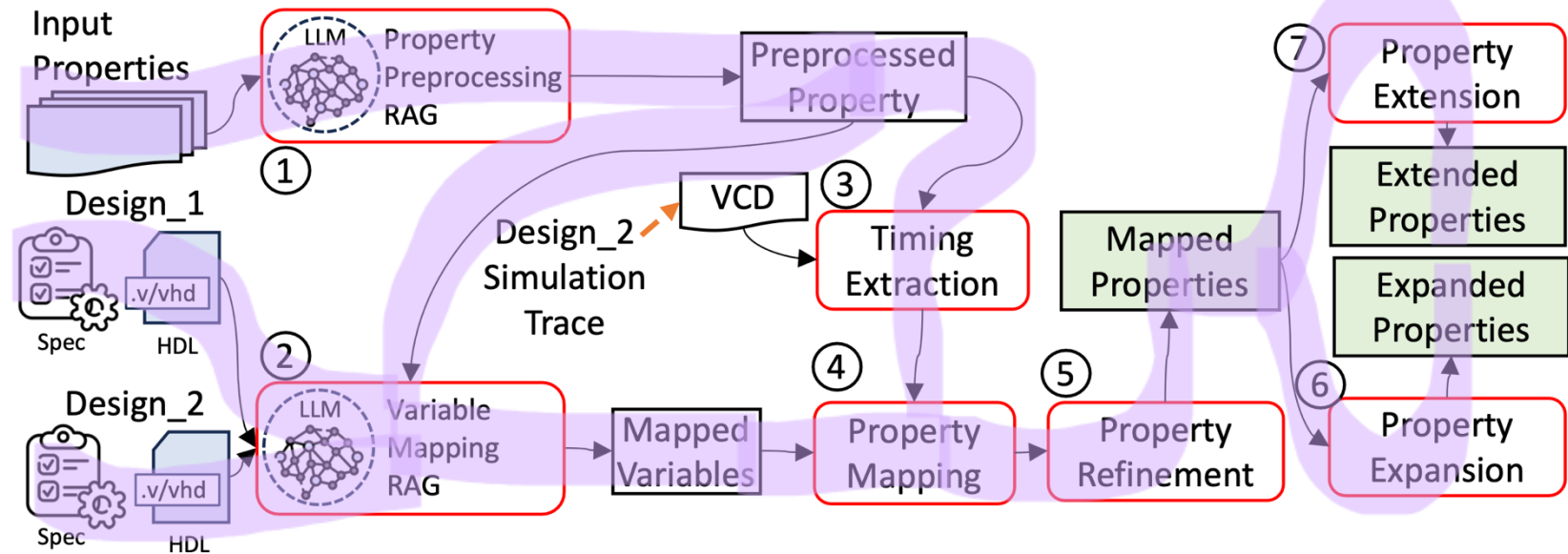
- **Shortened Time-to-Market:** Tight schedules for new releases can compromise thorough security verification.
- **Design Optimization:** Performance enhancements may ignore or weaken existing security measures.
- **Feature Additions:** New functionalities may introduce unexpected vulnerabilities.
- **IP Integration Changes:** Adding or updating third-party IP blocks can introduce new threats.
- **Algorithm Updates:** Changes in encryption or data handling algorithms may lead to implementation flaws.
- **Resource Reallocation:** Shifting hardware resources for efficiency can impact security-critical components.
- **Software-Hardware Interface Modifications:** Altered interfaces may expose new attack surfaces.

Example of Security property mapping between design versions



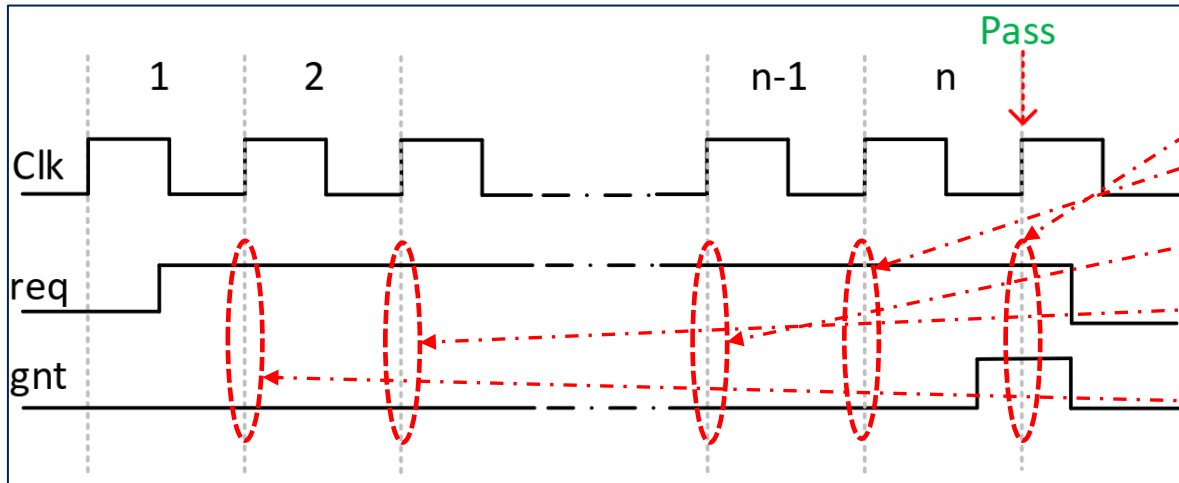
Property Mapping Overview

1. Property preprocessing
2. Variable mapping
3. Timing extraction
4. Property mapping
5. Property refinement
6. Property extension
7. Property expansion



Property Preprocessing

Property P1: If *req* is asserted, *gnt* will eventually be asserted



Consequent becomes **true**

- P1 is resolved

Antecedent: True, Consequent: False
(n-1)th copy of P1 is stored

Antecedent: True, Consequent: False
(n-2)th copy of P1 is stored

Antecedent: True, Consequent: False
2nd copy of P1 is stored

Antecedent: True, Consequent: False
1 copy of P1 is stored

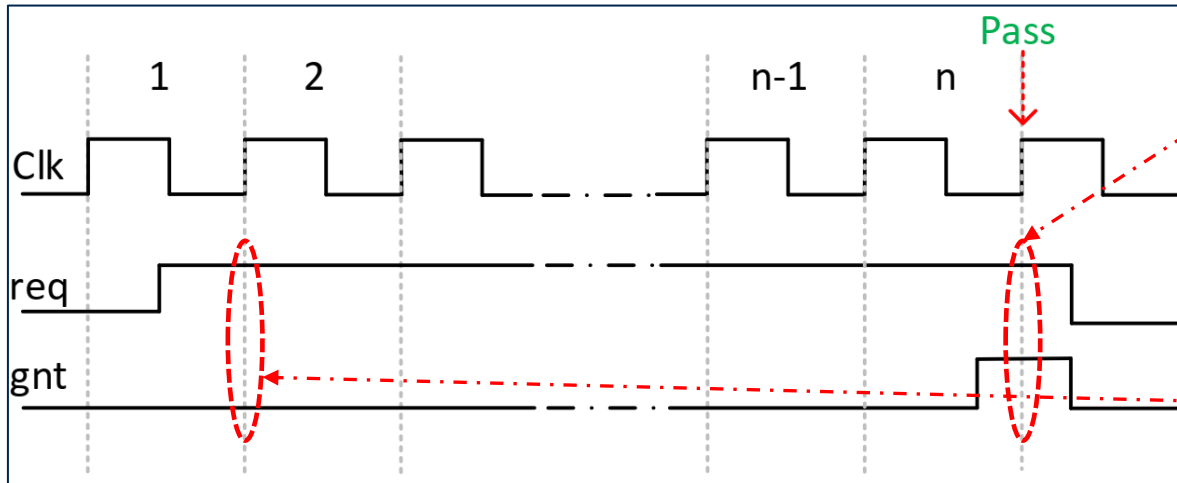
```
property p1;  
@(posedge clk) req |-> s_eventually(gnt)  
endproperty
```

Antecedent

Consequent

Property Preprocessing

Property P1: If *req* is asserted, *gnt* will eventually be asserted



Consequent becomes true

- P1 is resolved

Antecedent: True, Consequent: False
1 copy of P1 is stored

```
property p1;  
@(posedge clk) req |-> s_eventually(gnt)  
endproperty
```

Antecedent

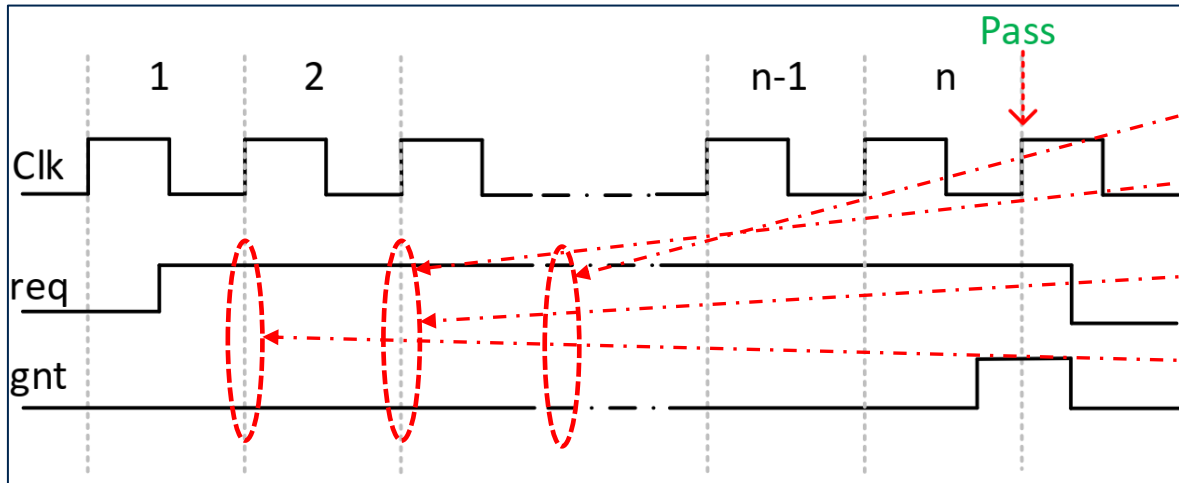
Consequent

```
property p1_1;  
@(posedge clk) $rose(req) |->  
s_eventually(gnt)  
endproperty
```

Consequent Antecedent

Property Preprocessing

Property P1: If *req* is asserted, *gnt* will eventually be asserted



Consequent becomes **true**

- P1 is resolved

Antecedent: True

1 copy of P1 is stored

Consequent becomes **true**

- P1 is resolved

Antecedent: True

1 copy of P1 is stored

```
property p1;
@(posedge clk) req |-> s_eventually(gnt)
endproperty
```

Antecedent

Consequent

```
property p1_1;
@(posedge clk) $rose(req) |->
s_eventually(gnt)
endproperty
```

Consequent Antecedent

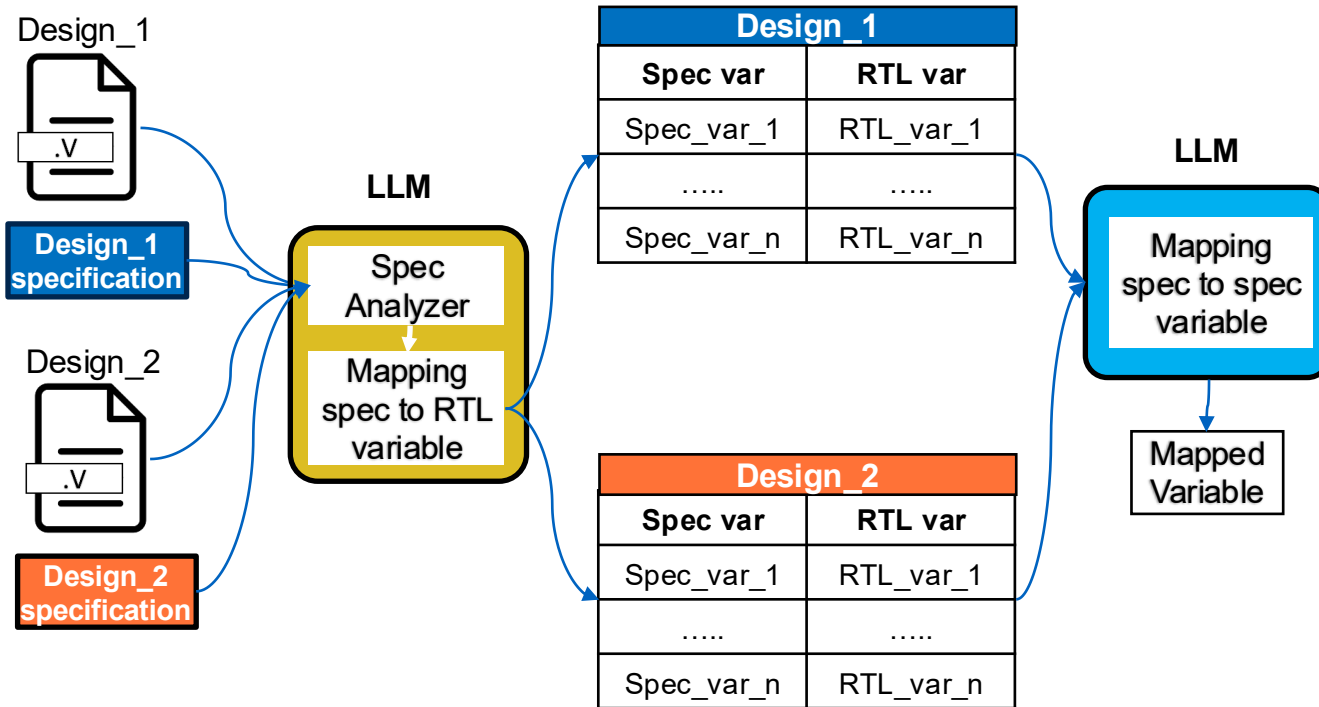
```
property p1_2;
@(posedge clk) req && !gnt |=> req
endproperty
```

Antecedent

Consequent

Ref: https://www.cadence.com/en_US/home/training/all-courses/86361.html

Variable Mapping



[Custom Instruction for Signal Mapper]

- Please act as a signal name mapping tool to link the specification file and the Verilog code.
- Firstly, I'll upload the the **design specification file** (in PDF format), and a Verilog file containing all the **signal definitions** (*.v format).
- Please analyze the specification file and use the code interpreter to analyze the Verilog file (both the signal declarations and comments). Then map every signal (including IO ports, wires, and registers) defined in Verilog with the description in the specification. Finally, please output each signal in the following format:

[Signal name in Verilog]: Signal definition in Specification file

[Prompt for Signal Mapper]

- [Upload the original SPEC file and the Verilog signal definition file] Here are the files for the design specification and the signal definitions, please analyze the two files carefully.
- Please output each signal with the mapping relationship

[Response from Signal Mapper]

Based on the analysis of both the design specification and the Verilog file, here is the mapping of signals, wires, and regs in the Verilog file to their respective descriptions in the specification:

(IO port)
 wb_clk_i: Master clock
 wb_we_i: Write enable input

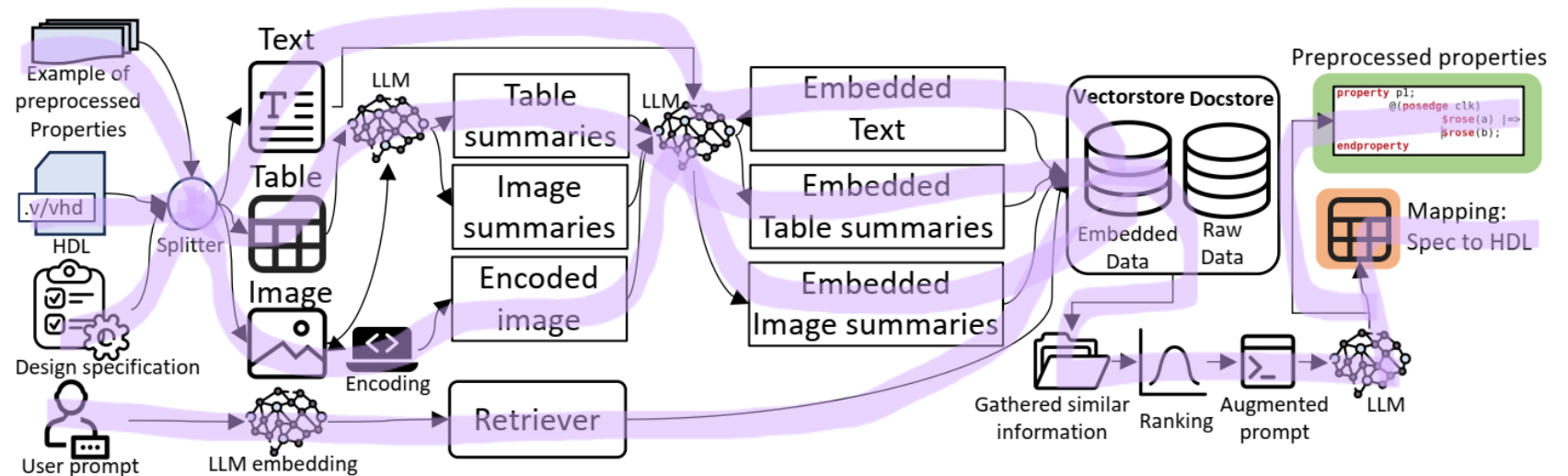
(Register)
 ctr: Control register
 sr: Status register

(Wire)
 en: Enable state of the I2C core.
 ien: Interrupt enable state of the I2C core

(Unmapped)
 The additional elements such as wires and regs like 'wb_wacc', 'sta' have been identified in the Verilog file. However, their specific descriptions are not directly found in the provided section of the specification.

RAG-based Variable Mapping & Property Preprocessing

- Context generation
 - HDL code
 - Specification
 - Example preprocessed properties
- User prompt augmentation
 - Retrieving relevant contents
 - Ranking contents based on similarity
 - Augmenting original prompt
- Response generation
 - Extracting response to the augmented prompt



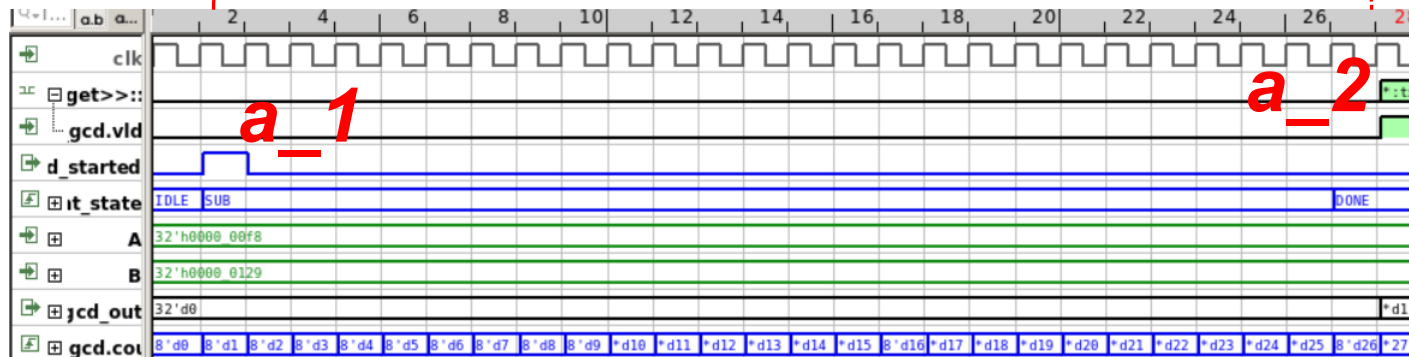
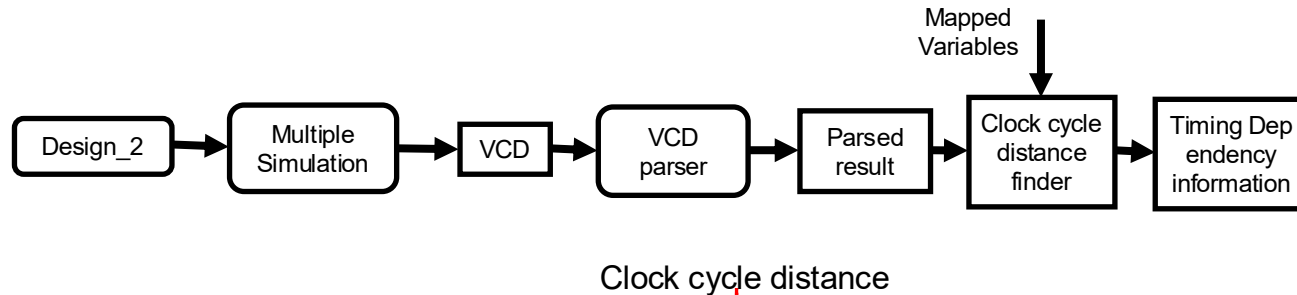
Timing Mapping Between Design Versions

Property: Design_1

$a_1 \rightarrow \#\#\ m\ b_1$

$P1: a_2 \rightarrow \#\#\ k\ b_2$

$P2: a_2 \rightarrow \#\#\ [k:p]\ b_2$

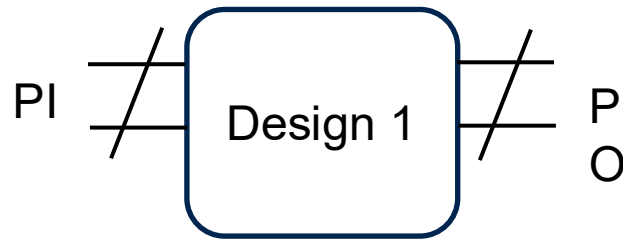


- If the clock cycle distance is constant, P1 is applicable
- If clock cycle distance varies, P2 is applicable

Cons: Design bugs, unintentional or the well-crafted security vulnerabilities can be picked up during property mapping.

Potential Solution: If any one between k and p is rare, it might be due to potential bug or security vulnerabilities. Accept P1, otherwise accept P2.

Design Transition



Property Extension

Property at Design 1
*Asset should not leak to **PI** and **PO***

Property
Extension

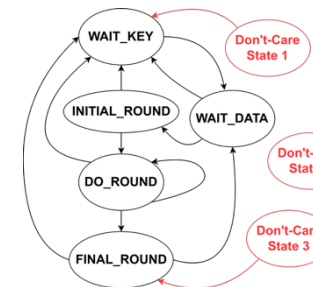
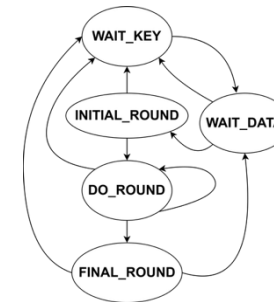
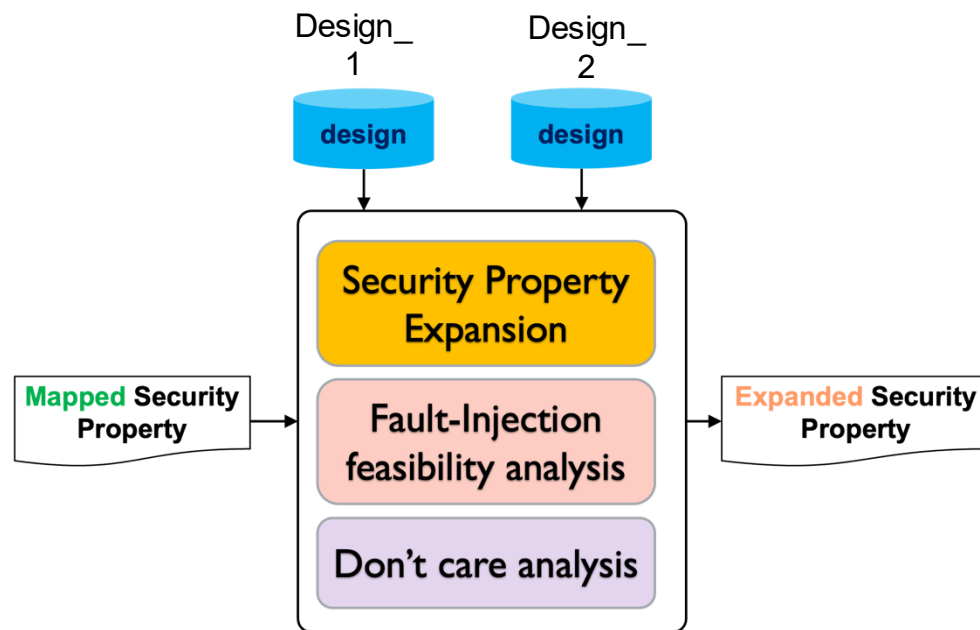
Extended Property at Design 2
*Asset should not leak to **PI**, **PO**, and **Additional PIs/POs***

Action Items

- Identification of additional attack surface
- Formulating the mapped properties based on
 - Asset
 - Threat model
- Does not create new properties

Security Property Expansion

- **Additional attack surface identification** → Finding additional attack surfaces, which can be exploited to violate the security goal that has already been proved in the higher abstraction level.
- **Expansion** → Creating new security properties, aligned with the original security goal, to verify it is not violated by the additional attack surfaces.



Target Threat: **Access control violation**

Version	Attack surface
Design_1	FSM Encoding
Design_2	Don't Care State

Security Property Expansion

Target Threat: Access control violation

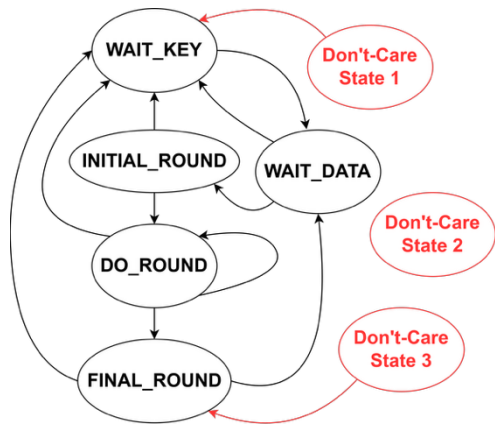
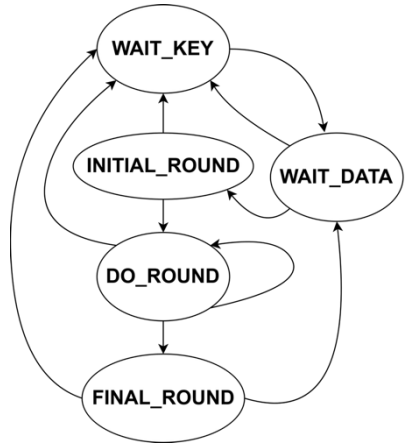
Abstraction | Attack surface

Design_1

FSM Encoding

Design_2

Don't Care States



Design_1 Property:

FINAL_ROUND should be accessed only from DO_ROUND

Property Expansion

Expanded Property in Design_2:

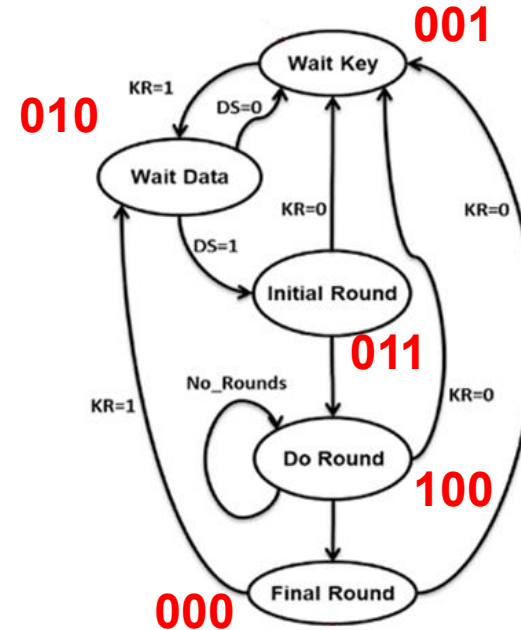
1. FINAL_ROUND should not be accessed from any don't care states
2. For any valid transition except the strongly connected state the FI feasibility metric must be 0

Algorithm to Calculate Fault-injection Feasibility

Algorithm 1: DETERMINING THE FEASIBILITY OF FAULT-INJECTION ATTACK

Input : gate-level Netlist
Output: feasibility of fault-injection attack

- 1 $T(x, y) \leftarrow$ *Extracted Finite StateMachine* // List of all state transitions
- 2 $T_p(x, y) \leftarrow$ *Strongly connected transitions with protected states*
- 3 **for** each $T(x, y) \notin T_p(x, y)$ **do**
- 4 $State_x = [b_{x(n-1)} \cdots b_{x1} b_{x0}]$
- 5 $State_y = [b_{y(n-1)} \cdots b_{y1} b_{y0}]$
- 6 $State_p = [b_{p(n-1)} \cdots b_{p1} b_{p0}]$
- 7 **FI feasibility** = $(\prod_{i=0}^{n-1} ((S_{yi} \oplus S_{pi}) \parallel (S_{xi} \oplus S_{pi})))$
- 8 **end for**



Expanded Property to verify Fault-Injection vulnerability in the FSM state encoding

```

property FI_vulnerability;
    @(posedge clk) (!(aes_binary.FSM==FINAL_ROUND) &&
    !($past(aes_binary.FSM)==FINAL_ROUND))
    |->
    (!(!(Raes_binary.FSM[0] ^ FINAL_ROUND[0])
    !($past(aes_binary.FSM[0]) ^ FINAL_ROUND[0]) )
    || &&
    ( !(aes_binary.FSM[1] ^ FINAL_ROUND[1])
    !($past(aes_binary.FSM[1]) ^ FINAL_ROUND[1]) )
    || &&
    ( !(aes_binary.FSM[2] ^ FINAL_ROUND[2])
    !($past(aes_binary.FSM[2]) ^ FINAL_ROUND[2]) )
    || );
endproperty
    
```

Algorithm to extract possible don't care states

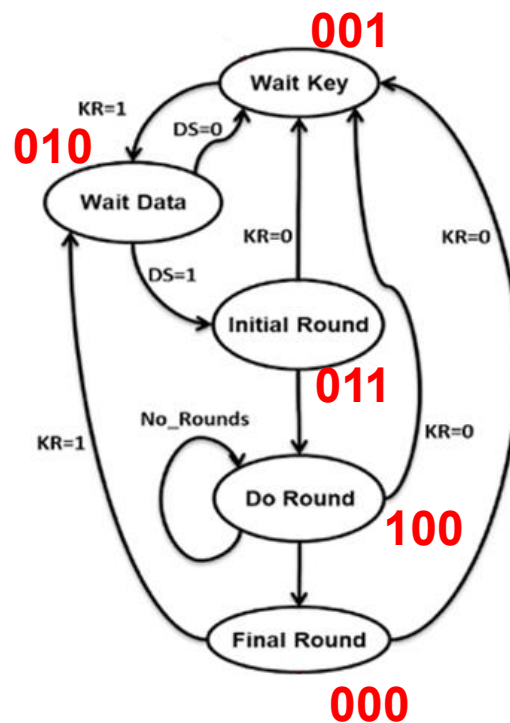
Algorithm 3: EXTRACTION OF THE POSSIBLE DON'T CARE STATES

```

Input : List of States, List of Transitions, List of transitions strongly
connected to the protected States
Output: List of Possible Don't Care States
1  $S \leftarrow$  List of all States
2  $T(x, y) \leftarrow$  List of all transitions
3  $T_p(x, y) \leftarrow$ 
List of strongly connected transitions with protected states

4  $D = []$  // List of don't care states
5
6 for each  $T(x, y) \notin T_p(x, y)$  do
7    $State_x = [b_{x(n-1)} \dots b_{x1} b_{x0}]$ 
8    $State_y = [b_{y(n-1)} \dots b_{y1} b_{y0}]$ 
9   if encoding==one-hot then
10    if  $(0_{(n-1)}0_{(n-2)}0_10_0) \notin$  both  $S$  &  $D$  then
11      $D.append((0_{(n-1)}0_{(n-2)}0_10_0))$ 
12     $PD_{OH} = (S_x \oplus S_y)$  // possible don't care
state
13    if  $(PD_{OH} \notin S)$  then
14      $D.append(PD_{OH})$ 
15   else if encoding==binary then
16     $PD_{bin(n-1)} = [b_{x(n-1)}b_{y(n-2)} \dots b_{y1}b_{y0}]$ 
17     $PD_{bin(n-2)} = [b_{y(n-1)}b_{x(n-2)} \dots b_{y1}b_{y0}]$ 
18    :
19     $PD_{bin(1)} = [b_{y(n-1)}b_{y(n-2)} \dots b_{x1}b_{y0}]$ 
20     $PD_{bin(0)} = [b_{y(n-1)}b_{y(n-2)} \dots b_{y1}b_{x0}]$ 
21   for each  $PD_{bin} \notin S$  do
22     $D.append(PD_{bin})$ 
23   end for
24 end for

```



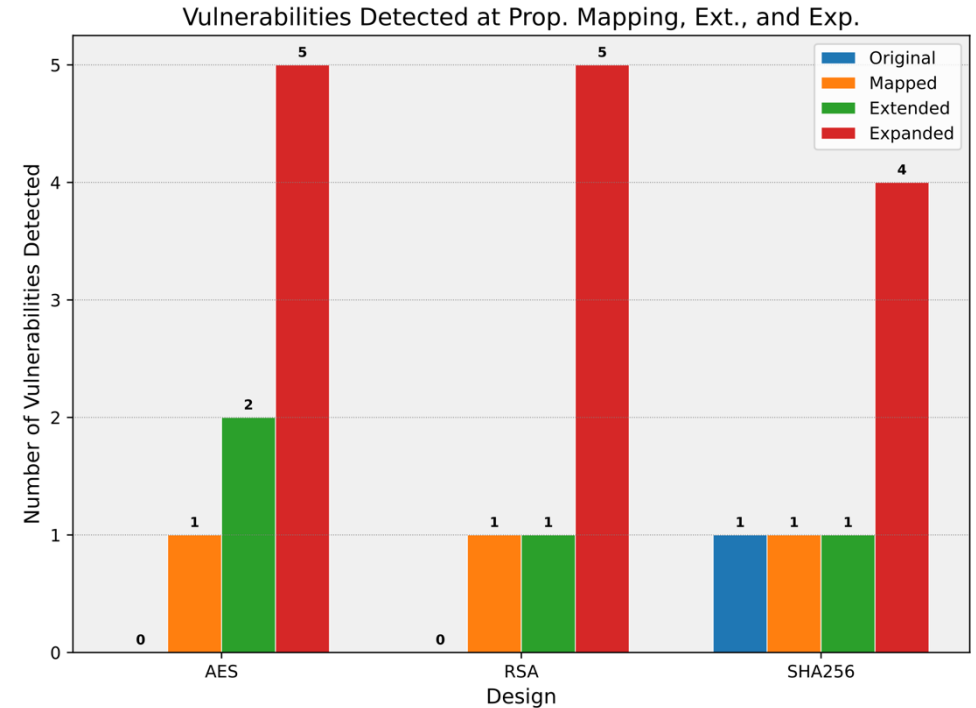
Binary Encoding

- Legal transition → 001 to 010
- Reachable states by FI → 000, 011
- Transition → 010 to 011
- Reachable states by FI → 010
- Transition → 011 to 100
- Reachable states by FI → 000, 110, 101

Expanded Property Expanded Property_1: Protected States should not be accessed from 110 state
Expanded Property_2: Protected States should not be accessed from 101 state

Property Mapping and Verification Result

Design	Prop. No.	# var	# correctly mapped	Temporal	Target Vulnerability	Verification Result (Detected Vulnerability)			
						Original	Mapped	Prop. Ext.	Prop. Exp.
AES (5 version)	P1	3	3	-	Information Leakage	-	-	-	FSE, FI
	P2	3	3	Yes	Timing SC	-	MI	-	FSE, FI
	P3	5	5	-	InformationLeakage	-	-	IL(HS)	-
	P4	5	4	-	Access Control	-	-	IL	-
	P5	3	3	-	Access Control	-	-	-	FSE, FI
	P6	3	3	-	Access Control	-	-	-	FSE, FI
	P7	2	2	-	Denial of Service	-	-	-	DOS
RSA (3 versions)	P1	3	3	-	Information Leakage	-	-	-	FSE, FI
	P2	3	3	-	Access Control	-	-	-	FSE, FI
	P3	4	4	-	Information Leakage	-	-	-	FSE, FI
	P4	3	3	Yes	Timing SC	-	MI	-	FSE, FI
	P5	3	3	-	Access Control InformationLeakage	-	-	IL	-
	P6	2	2	-	Denial of Service	-	-	-	DOS
SHA256 (5 versions)	P1	3	3	-	Information Leakage	-	-	-	FSE, FI
	P2	3	3	-	Information Leakage	-	-	-	FSE, FI
	P3	3	3	-	Access Control	-	-	-	FSE, FI
	P4	15	14	-	Access Control Information Leakage	IL(PIO)	IL(PIO)	IL(HS)	-
	P5	2	2/2	-	Denial of Service	-	-	-	DOS



- **Benchmarks**

- 5 versions of AES
- 3 versions of RSA
- 5 versions of SHA256

- **Inserted vulnerabilities:**

- Vulnerabilities were inserted in all version of AES and RSA except the original one.
- For a particular input pattern, intermediate states are bypassed

- **Successfully mapped all variables except the following**

- 1 variable in AES
- 1 variable in SHA256

- Objective: Introduced a framework for translating security properties across design versions.
- Problem: Security verification is time-consuming and often compromised due to rapid design iterations.
- Solution: Leverage LLMs and RAG-based methods to automate property translation, extension, and expansion.
- Results: Achieved significant reductions in verification time while maintaining security integrity identifying new vulnerabilities.
- Impact: Enhanced security continuity across design versions

Thank You

