Hardware Root of Trust - Verification Plan

Version 1.0

---

1. Executive Summary

This document outlines the comprehensive verification plan for the Hardware Root of Trust (HRoT) system. The plan addresses both functional correctness and security properties of the design, ensuring that the system provides robust cryptographic services while maintaining isolation of sensitive key material.

Verification Scope: - Root of Trust Top-Level Integration (`root_of_trust_top`) - PUF modules (DUS and Device ID) - Key Derivation Function (KDF) - Secure Key Distributor - Cryptographic modules (SHA-256, HMAC-SHA-256, AES-CTR) - Security isolation and fault injection testing

Verification Methodology: - UVM-based constrained-random verification - Directed tests for critical security scenarios - Functional coverage tracking - Security-specific assertions and checkers - Fault injection and negative testing

---

2. Verification Goals

2.1 Functional Goals

1. **Initialization Sequence**: Verify correct PUF enrollment, key derivation, and key distribution
2. **Cryptographic Operations**: Validate SHA-256, HMAC-SHA-256, and AES-CTR functionality
3. **State Machine Behavior**: Verify all state transitions and control flows
4. **Error Handling**: Ensure proper fault detection and recovery mechanisms
5. **Bus Interface**: Validate accessible vs. isolated signal paths

2.2 Security Goals

1. **Isolation Verification**: Prove DUS and cryptographic keys are never exposed to bus
2. **Zeroization**: Verify complete key erasure on zeroize command
3. **PUF Robustness**: Validate error correction and regeneration reliability
4. **Key Uniqueness**: Ensure derived keys are cryptographically different
5. **Fault Resistance**: Test system behavior under fault injection attacks
6. **Side-Channel Awareness**: Identify potential timing or power-based leakage points

---

3. Feature List and Priority

Feature	Priority	Type	Description
System Initialization	P0	Functional	Complete boot sequence from reset to operational
PUF DUS Enrollment	P0	Functional/Security	First-time PUF characterization and helper data generation
PUF DUS Regeneration	P0	Functional/Security	Secret regeneration using helper data
PUF Device ID	P1	Functional	Unique device identifier generation
KDF Key Derivation	P0	Security	Derive HMAC, AES, SHA keys from DUS
Key Distribution	P0	Security	Secure transfer of keys to crypto modules
Emergency Zeroization	P0	Security	Complete key erasure in all registers
SHA-256 Operation	P1	Functional	Hash computation
HMAC-SHA-256 Operation	P1	Functional	HMAC computation with derived key
AES-CTR Operation	P1	Functional	Encryption/decryption with derived key
DUS Isolation	P0	Security	DUS never visible on bus interface
Key Isolation	P0	Security	Crypto keys never visible on bus interface
Error Detection (PUF)	P1	Functional	Detect and report uncorrectable PUF errors
Fault State Handling	P1	Functional	Proper fault state entry and recovery
Re-derivation	P2	Functional	Derive new keys from existing DUS

Priority Levels: - **P0**: Critical - Must be verified before tape-out - **P1**: High - Required for production - **P2**: Medium - Should be verified if schedule permits

---

4. Test Scenarios

4.1 Initialization and Enrollment Tests

Test 4.1.1: Cold Boot First-Time Enrollment **Objective**: Verify complete system initialization from power-on with fresh PUFs

Stimulus: 1. Apply reset 2. Assert `system_init` with `puf_dus_enroll` = 1 3. Monitor state transitions: IDLE → INIT_PUF_DUS → INIT_PUF_DEVID → INIT_WAIT_DUS → INIT_DERIVE_KEYS → INIT_WAIT_KDF → INIT_DISTRIBUTE_KEYS → INIT_COMPLETE

Expected Results: - `puf_dus_helper_valid` asserted with valid helper data - `device_id_valid` asserted with non-zero device ID - `system_ready` asserted - `keys_active` asserted - All crypto modules receive valid keys

Coverage Points: - State machine coverage: All enrollment states visited - Helper data generation - Device ID generation - Key distribution completion

---

Test 4.1.2: Warm Boot with DUS Regeneration **Objective**: Verify system initialization using previously enrolled PUF

Stimulus: 1. Use helper data from previous enrollment test 2. Apply reset 3. Assert `system_init` with `puf_dus_enroll` = 0, `puf_dus_regenerate` = 1 4. Provide `puf_dus_helper_in` from enrollment

Expected Results: - PUF successfully regenerates DUS (within error threshold) - Keys derived match original keys (if DUS matches) - `system_ready` asserted - No `security_fault` or `dus_error`

Coverage Points: - PUF regeneration path - Error correction mechanism - Key consistency across boots

---

Test 4.1.3: PUF Error Handling - Excessive Bit Errors **Objective**: Verify proper error handling when PUF regeneration fails

Stimulus: 1. Provide corrupted or mismatched `puf_dus_helper_in` 2. Attempt regeneration

Expected Results: - `dus_error` asserted - State machine transitions to FAULT state - `security_fault` output asserted - Keys not distributed (`keys_active` = 0) - System remains in fault state until reset

Coverage Points: - Error detection coverage - Fault state entry - Security fault reporting

---

4.2 Key Derivation and Distribution Tests

Test 4.2.1: Key Derivation Correctness **Objective**: Verify KDF generates valid, unique keys

Stimulus: 1. Complete system initialization 2. Monitor KDF outputs

Expected Results: - `kdf_hmac_key`, `kdf_aes_key`, `kdf_sha_key` all non-zero - All three keys are different from each other - Keys are different from DUS input - `kdf_keys_valid` asserted

Coverage Points: - KDF activation - Key uniqueness - Key validity checks

---

Test 4.2.2: Secure Key Distribution **Objective**: Verify keys transferred securely from KDF to crypto modules

Stimulus: 1. After KDF completion, monitor key distributor 2. Verify keys reach HMAC, AES, SHA modules

Expected Results: - `hmac_key_valid_internal`, `aes_key_valid_internal`, `sha_key_valid_internal` all asserted - Crypto module key inputs match KDF outputs - `keys_distributed` asserted - `keys_active` asserted

Coverage Points: - Key distributor state machine - Key validity propagation - Lock mechanism after distribution

---

Test 4.2.3: Key Re-derivation **Objective**: Verify ability to derive new keys from existing DUS

Stimulus: 1. Complete initial system initialization 2. Assert `derive_new_keys` 3. Monitor key re-derivation process

Expected Results: - State machine: INIT_COMPLETE → INIT_DERIVE_KEYS → INIT_WAIT_KDF → INIT_DISTRIBUTE_KEYS → INIT_COMPLETE - New keys generated (different from original if context changes) - System remains operational

Coverage Points: - Re-derivation path - Key update mechanism

4.3 Security Isolation Tests

Test 4.3.1: DUS Isolation - Bus Non-Observability **Objective**: Prove DUS is never exposed on bus-accessible signals

Verification Method: 1. Use formal verification or comprehensive simulation monitoring 2. Monitor all output ports during all test scenarios 3. Ensure `dus_secret` never appears on any external output

Expected Results: - DUS value never visible on: `device_id`, `sha_hash`, `hmac_mac`, `aes_data_out` - DUS only flows to KDF (internal signal)

Coverage Points: - Signal propagation analysis - Output correlation checks

---

Test 4.3.2: Cryptographic Key Isolation **Objective**: Verify derived keys are never visible on bus interface

Verification Method: 1. Monitor all bus-accessible outputs 2. Attempt to correlate outputs with known key values 3. Verify key registers are not directly readable

Expected Results: - Key values never appear on bus outputs - Only encrypted/hashed data visible externally - No correlation between bus outputs and key material

Coverage Points: - Key path isolation - Crypto module input/output separation

---

Test 4.3.3: KDF Isolation **Objective**: Verify KDF module is not accessible from bus

Verification Method: 1. Analyze interface connections 2. Verify no direct bus access to KDF inputs/outputs 3. Confirm KDF only controlled by internal state machine

Expected Results: - KDF control signals only from internal state machine - No external control of `kdf_derive` - KDF outputs only to secure key distributor

Coverage Points: - Interface topology verification - Control path isolation

---

4.4 Cryptographic Functional Tests

Test 4.4.1: SHA-256 Computation **Objective**: Verify SHA-256 produces correct hashes

Stimulus: 1. After system initialization 2. Provide known test vectors to `sha_message` 3. Assert `sha_init` and `sha_start`

Expected Results: - `sha_ready` $\rightarrow$ `sha_valid` handshake - `sha_hash` matches expected NIST test vectors - Ready for next operation after completion

Test Vectors: Use NIST FIPS 180-4 test vectors - Empty message - 448-bit message - 512-bit message - Multi-block messages

Coverage Points: - SHA core state machine - Message block processing - Hash output validation

---

Test 4.4.2: HMAC-SHA-256 Computation **Objective**: Verify HMAC-SHA-256 with derived keys

Stimulus: 1. After key distribution 2. Provide known messages to `hmac_message` 3. Use `hmac_init`, `hmac_start`, `hmac_final` protocol

Expected Results: - `hmac_ready` → `hmac_valid` handshake - `hmac_mac` output matches expected HMAC with known key - Multi-block HMAC processing works correctly

Test Vectors: Use NIST test vectors - Short messages (< 512 bits) - Block-aligned messages - Multi-block messages

Coverage Points: - HMAC state machine - Key loading - Message processing - Final block handling

---

Test 4.4.3: AES-CTR Encryption/Decryption **Objective**: Verify AES-CTR mode operation

Stimulus: 1. After key distribution 2. Provide `aes_nonce` and `aes_counter_init` 3. Encrypt known plaintext with `aes_start` 4. Decrypt ciphertext to verify round-trip

Expected Results: - Encryption produces known ciphertext (with known key) - Decryption recovers original plaintext - Counter increments properly for stream generation - `aes_ready` → `aes_valid` handshake

Test Vectors: Use NIST AES-CTR test vectors - Various nonce values - Counter overflow scenarios - Multiple blocks

Coverage Points: - AES core state machine - CTR mode counter management - Key loading - Data path

---

4.5 Zeroization and Fault Tests

Test 4.5.1: Emergency Zeroization **Objective**: Verify complete key erasure on zeroize command

Stimulus: 1. After system fully operational with active keys 2. Assert `zeroize_all` 3. Monitor all key registers

Expected Results: - State machine transitions to FAULT state - All key registers in `secure_key_distributor` set to 0 - `keys_active` deasserted - `security_fault` asserted - Crypto modules receive zeroed keys - `hmac_key_valid_internal`, `aes_key_valid_internal`, `sha_key_valid_internal` all deasserted

Verification Method: - Use force/release to observe internal key registers - Verify complete zeroization within N clock cycles

Coverage Points: - Zeroization path coverage - Key register clearing - Fault state entry from all states

---

Test 4.5.2: Zeroization from Multiple States **Objective**: Verify zeroize works from any operational state

Stimulus: 1. Trigger zeroize from: INIT_IDLE, INIT_PUF_DUS, INIT_DERIVE_KEYS, INIT_COMPLETE 2. Monitor state transitions and key clearing

Expected Results: - Zeroize always succeeds regardless of current state - All sensitive data cleared - System enters FAULT state

Coverage Points: - Zeroize from all states - State machine override mechanism

---

Test 4.5.3: Fault Injection - Key Corruption Detection **Objective**: Verify integrity checking detects key corruption

Stimulus: 1. System in ACTIVE state with valid keys 2. Force corrupt key register values (inject faults) 3. Wait for integrity check cycle

Expected Results: - `integrity_fail` signal asserted in `secure_key_distributor` - Key outputs forced to zero - `keys_active` deasserted

Coverage Points: - Integrity counter rollover - Corruption detection - Automatic key invalidation

---

4.6 Device ID Tests

Test 4.6.1: Device ID Enrollment and Read **Objective**: Verify Device ID PUF operation

Stimulus: 1. Enroll Device ID PUF during initialization 2. Read Device ID multiple times

Expected Results: - `device_id` stable across multiple reads - `device_id_valid` asserted - Device ID is non-zero and unique

Coverage Points: - Device ID PUF enrollment - Device ID read operation - ID consistency

---

Test 4.6.2: Device ID Uniqueness (Multi-DUT) **Objective**: Verify different devices produce different IDs

Stimulus: 1. Simulate multiple instantiations with different DEVICE_SEED values 2. Compare generated device IDs

Expected Results: - Each device produces unique ID - No collisions across reasonable number of devices

Coverage Points: - PUF seed variation - ID uniqueness

4.7 Integration and System-Level Tests

Test 4.7.1: End-to-End Cryptographic Flow **Objective**: Verify complete data processing flow

Stimulus: 1. Initialize system (enrollment/regeneration) 2. Perform SHA-256 hash 3. Perform HMAC-SHA-256 with derived key 4. Perform AES-CTR encryption/decryption

Expected Results: - All operations complete successfully - Outputs are cryptographically correct - No interference between operations

Coverage Points: - Multi-operation sequence - Resource sharing - State machine interleaving

---

Test 4.7.2: Concurrent Operations **Objective**: Test multiple crypto operations in quick succession

Stimulus: 1. Queue SHA, HMAC, AES operations back-to-back 2. Monitor ready/valid handshakes

Expected Results: - Operations serialize correctly - No data corruption - All operations complete with correct results

Coverage Points: - Back-to-back operation transitions - Ready/valid protocol compliance

---

Test 4.7.3: Reset During Operations **Objective**: Verify safe reset behavior

Stimulus: 1. Start crypto operation 2. Assert reset mid-operation 3. Re-initialize and retry

Expected Results: - Reset safely aborts operation - No residual state corruption - System re-initializes cleanly

Coverage Points: - Reset from all states - State machine reset behavior

---

5. Coverage Metrics

5.1 Code Coverage

- **Line Coverage**: Target 100% of non-unreachable code
- **Branch Coverage**: Target 100% of all conditional branches
- **FSM Coverage**: 100% state coverage, 100% transition coverage
- **Toggle Coverage**: 95% on all data paths and control signals

5.2 Functional Coverage

5.2.1 Initialization Sequence Coverage

```
covergroup init_sequence_cg @(posedge clock);
    state_cp: coverpoint init_state {
        bins all_states[] = {INIT_IDLE, INIT_PUF_DUS, INIT_PUF_DEVID,
                             INIT_WAIT_DUS, INIT_DERIVE_KEYS, INIT_WAIT_KDF,
                             INIT_DISTRIBUTE_KEYS, INIT_COMPLETE, FAULT};
    }

    transition_cp: coverpoint init_state {
        bins enroll_flow = (INIT_IDLE => INIT_PUF_DUS => INIT_PUF_DEVID =>
                            INIT_WAIT_DUS => INIT_DERIVE_KEYS => INIT_WAIT_KDF =>
                            INIT_DISTRIBUTE_KEYS => INIT_COMPLETE);
        bins rederive_flow = (INIT_COMPLETE => INIT_DERIVE_KEYS => INIT_WAIT_KDF =>
                              INIT_DISTRIBUTE_KEYS => INIT_COMPLETE);
        bins fault_from_idle = (INIT_IDLE => FAULT);
        bins fault_from_puf = (INIT_PUF_DUS => FAULT);
        bins fault_from_complete = (INIT_COMPLETE => FAULT);
    }

    enroll_type: coverpoint puf_dus_enroll {
        bins first_enroll = {1};
        bins regenerate = {0};
    }
endgroup
```

5.2.2 PUF Error Rate Coverage

```
covergroup puf_error_cg @(posedge clock);
    error_count_cp: coverpoint error_count {
        bins no_errors = {0};
        bins low_errors = {[1:8]};
        bins medium_errors = {[9:16]};
```

```
        bins high_errors = {[17:31]};
        bins correctable_max = {32};
        bins uncorrectable = {[33:255]};
    }

    error_result: coverpoint dus_error {
        bins success = {0};
        bins failure = {1};
    }

    cross error_count_cp, error_result;
endgroup
```

5.2.3 Cryptographic Operation Coverage

```
covergroup crypto_ops_cg @(posedge clock);
    sha_ops: coverpoint sha_start {
        bins sha_active = {1};
    }

    hmac_ops: coverpoint hmac_start {
        bins hmac_active = {1};
    }

    aes_ops: coverpoint aes_start {
        bins aes_active = {1};
    }

    // Cover concurrent requests
    cross sha_ops, hmac_ops, aes_ops;
endgroup
```

5.2.4 Security Event Coverage

```
covergroup security_events_cg @(posedge clock);
    zeroize_events: coverpoint zeroize_all {
        bins zeroize_cmd = {1};
    }

    fault_events: coverpoint security_fault {
        bins fault_detected = {1};
    }

    key_state: coverpoint keys_active {
        bins keys_on = {1};
        bins keys_off = {0};
```

```
    }

    // Cover zeroization from different key states
    cross zeroize_events, key_state;
endgroup
```

5.3 Assertion Coverage

- **Concurrent Assertions**: 100% assertion activation
- **Security Assertions**: 100% of security-critical properties checked
- **Protocol Assertions**: 100% of interface protocols verified

---

6. Security-Specific Verification

6.1 Isolation Verification Plan

6.1.1 Information Flow Analysis **Objective**: Formally prove no information flow from DUS/keys to bus

Method: 1. **Taint Tracking**: Mark DUS and key signals as "tainted" 2. **Propagation Analysis**: Track tainted signals through design 3. **Output Verification**: Verify no tainted signals reach bus outputs

Tools: - Custom monitors in testbench - Formal verification (if available) - Signal correlation analysis

Success Criteria: - Zero correlation between DUS/keys and bus-accessible outputs - Formal proof of isolation (if formal tools used)

6.1.2 Side-Channel Awareness Testing **Objective**: Identify potential side-channel leakage points

Tests: 1. **Timing Analysis**: Measure operation latency variance based on key/data values 2. **Power Estimation**: Identify operations with key-dependent power (simulation) 3. **Information Leakage**: Test for correlation between intermediate values and secrets

Expected Results: - Document potential leakage points - Recommendations for physical implementation (e.g., balanced routing, power filtering)

Note: Full side-channel resistance requires silicon implementation and testing

---

6.2 Fault Injection Testing

6.2.1 Single-Bit Fault Injection **Stimulus**: - Inject single-bit flips in key registers, state registers, control signals - Target: key_dist_inst registers, KDF outputs, PUF outputs

Expected Results: - Integrity checking detects key corruption - State machine recovers or enters safe fault state - No silent corruption of cryptographic operations

---

6.2.2 Multi-Bit Fault Injection **Stimulus**: - Inject multiple simultaneous bit flips - Test Byzantine fault scenarios

Expected Results: - System enters fault state - Keys zeroized or marked invalid - No undefined behavior

---

6.2.3 Clock Glitch Injection **Stimulus**: - Inject clock glitches during critical operations: - During key distribution - During KDF operation - During PUF regeneration

Expected Results: - System detects timing violations (if implemented) - Safe failure mode (fault state) - No partial key exposure

---

6.3 Cryptographic Correctness

6.3.1 Test Vector Compliance **Requirement**: All crypto modules pass standard test vectors

Test Vectors: - **SHA-256**: NIST FIPS 180-4 test vectors (all message lengths) - **HMAC-SHA-256**: NIST test vectors (various key and message lengths) - **AES-CTR**: NIST SP 800-38A test vectors (all modes)

Success Criteria: 100% test vector pass rate

6.3.2 Key Derivation Validation **Objective**: Verify KDF produces cryptographically strong keys

Tests: 1. **Uniqueness**: Different DUS inputs produce different key sets 2. **Independence**: HMAC, AES, SHA keys are uncorrelated 3. **Non-determinism**: Keys appear random (statistical tests) 4. **Consistency**: Same DUS always produces same keys

Statistical Tests: - NIST SP 800-22 randomness tests on derived keys - Correlation tests between key pairs

---

6.4 PUF Reliability Testing

6.4.1 Error Correction Boundary Testing **Objective**: Find maximum correctable error rate

Stimulus: - Inject controlled bit errors in PUF regeneration - Sweep error count from 0 to 64 bits

Expected Results: - Errors 32 bits: Successful regeneration - Errors > 32 bits: Error detected, fault state entered - Boundary at 32-33 bits tested extensively

---

6.4.2 PUF Consistency Testing **Objective**: Verify PUF generates stable outputs

Stimulus: - Enroll PUF once - Regenerate 1000+ times with same helper data

Expected Results: - 100% regeneration success (in simulation) - Consistent DUS output - Error count within expected range

---

7. Testbench Architecture

7.1 UVM Environment Components

```
root_of_trust_tb_top
  root_of_trust_env
      puf_agent (passive monitor)
      sha_agent (active)
          sha_driver
          sha_monitor
          sha_sequencer
      hmac_agent (active)
          hmac_driver
          hmac_monitor
          hmac_sequencer
      aes_agent (active)
          aes_driver
          aes_monitor
          aes_sequencer
      control_agent (active)
          control_driver (init, zeroize, etc.)
          control_monitor
          control_sequencer
```

```
security_monitor
    Isolation checker
    Zeroization checker
    Fault injection controller
crypto_scoreboard
    SHA reference model
    HMAC reference model
    AES reference model
coverage_collector
     Functional coverage
     Security coverage
```

7.2 Reference Models

- **SHA-256**: OpenSSL or Python hashlib for golden reference
- **HMAC-SHA-256**: OpenSSL HMAC implementation
- **AES-CTR**: OpenSSL AES-CTR implementation
- **KDF**: NIST SP 800-108 KDF reference implementation

7.3 Key Components

7.3.1 Security Monitor Responsibilities: - Monitor all output ports for key/DUS leakage - Track key lifecycle (derivation → distribution → zeroization) - Verify isolation properties continuously - Inject faults on command

7.3.2 Crypto Scoreboard Responsibilities: - Collect crypto operation requests - Compute expected results using reference models - Compare DUT outputs with expected values - Report mismatches

7.3.3 Coverage Collector Responsibilities: - Functional coverage collection - Security event coverage - Protocol coverage - Generate coverage reports

---

8. Test Sequences

8.1 Base Sequences

1. **reset_sequence**: Apply reset, wait for idle
2. **enroll_sequence**: First-time PUF enrollment
3. **regenerate_sequence**: DUS regeneration with helper data
4. **zeroize_sequence**: Emergency zeroization
5. **sha_hash_sequence**: SHA-256 operation
6. **hmac_mac_sequence**: HMAC-SHA-256 operation
7. **aes_encrypt_sequence**: AES-CTR encryption
8. **aes_decrypt_sequence**: AES-CTR decryption

8.2 Scenario Sequences

1. **full_init_test**: Reset → Enroll → Derive → Distribute → Verify
2. **crypto_suite_test**: SHA → HMAC → AES operations
3. **fault_recovery_test**: Inject error → Verify fault state → Reset → Re-init
4. **zeroize_test**: Initialize → Operate → Zeroize → Verify clearing
5. **rederive_test**: Initialize → Operate → Re-derive keys → Verify new keys
6. **stress_test**: Random interleaving of all operations

8.3 Constrained Random Tests

```
class root_of_trust_random_test extends base_test;
    rand bit enroll_vs_regenerate;
    rand int num_crypto_ops;
    rand bit inject_zeroize;
    rand int zeroize_after_ops;

    constraint reasonable_c {
        num_crypto_ops inside {[10:100]};
        zeroize_after_ops < num_crypto_ops;
    }

    task run_phase(uvm_phase phase);
        // Random initialization
        if (enroll_vs_regenerate)
            run_enroll_sequence();
        else
            run_regenerate_sequence();

        // Random crypto operations
        repeat (num_crypto_ops) begin
            randcase
                40: run_sha_sequence();
                30: run_hmac_sequence();
                30: run_aes_sequence();
            endcase
        end

        // Random zeroization
        if (inject_zeroize)
            run_zeroize_sequence();
    endtask
endclass
```

---

9. Assertion Plan

9.1 Interface Assertions

9.1.1 Ready-Valid Protocol

```
// SHA-256 interface
property sha_valid_after_start;
    @(posedge clock) disable iff (reset)
    (sha_start && sha_ready) |-> ##[1:$] sha_valid;
endproperty
assert property (sha_valid_after_start);

// HMAC interface
property hmac_ready_when_idle;
    @(posedge clock) disable iff (reset)
    (state == IDLE) |-> hmac_ready;
endproperty
assert property (hmac_ready_when_idle);
```

9.1.2 Mutual Exclusivity

```
// Enroll and regenerate are mutually exclusive
property puf_mutual_exclusive;
    @(posedge clock) disable iff (reset)
    not (puf_dus_enroll && puf_dus_regenerate);
endproperty
assert property (puf_mutual_exclusive);
```

9.2 Security Assertions

9.2.1 DUS Isolation

```
// DUS never appears on bus outputs
property dus_not_on_bus;
    @(posedge clock) disable iff (reset)
    (dus_secret != 256'h0) |->
        (device_id != dus_secret[127:0]) &&
        (sha_hash != dus_secret) &&
        (hmac_mac != dus_secret) &&
        (aes_data_out != dus_secret[127:0]);
endproperty
assert property (dus_not_on_bus);
```

9.2.2 Key Zeroization

```
// Keys must be zero after zeroization
property keys_zero_after_zeroize;
```

```
    @(posedge clock) disable iff (reset)
    $rose(zeroize_all) |-> ##[1:10] (
        (key_dist_inst.hmac_key_reg == 256'h0) &&
        (key_dist_inst.aes_key_reg == 256'h0) &&
        (key_dist_inst.sha_key_reg == 256'h0)
    );
endproperty
assert property (keys_zero_after_zeroize);
```

9.2.3 Key Stability

```
// Keys don't change while active
property keys_stable_when_active;
    @(posedge clock) disable iff (reset || zeroize_all)
    (keys_active && $past(keys_active)) |->
        $stable(key_dist_inst.hmac_key_reg) &&
        $stable(key_dist_inst.aes_key_reg) &&
        $stable(key_dist_inst.sha_key_reg);
endproperty
assert property (keys_stable_when_active);
```

9.3 State Machine Assertions

9.3.1 State Reachability

```
// All states must be reachable
cover property (@(posedge clock) init_state == INIT_IDLE);
cover property (@(posedge clock) init_state == INIT_PUF_DUS);
cover property (@(posedge clock) init_state == INIT_DERIVE_KEYS);
cover property (@(posedge clock) init_state == FAULT);
// ... etc for all states
```

9.3.2 Fault State Permanence

```
// Once in fault, stay in fault until reset
property fault_permanent;
    @(posedge clock) disable iff (reset)
    (init_state == FAULT) |=> (init_state == FAULT);
endproperty
assert property (fault_permanent);
```

---

10. Regression Plan

10.1 Smoke Test Suite (Fast - Daily)

Runtime Target: < 5 minutes **Tests**: 1. Basic initialization (enroll + regenerate) 2. Single SHA, HMAC, AES operation each 3. Basic zeroization test 4. Reset test

Purpose: Quick sanity check after code changes

10.2 Functional Regression (Medium - Nightly)

Runtime Target: < 2 hours **Tests**: - All directed tests from Section 4 - 100 iterations of constrained random tests - Full coverage collection

Purpose: Comprehensive functional verification

10.3 Security Regression (Long - Weekly)

Runtime Target: < 8 hours **Tests**: - All security tests from Section 6 - 1000+ iterations of fault injection - Extended PUF reliability tests - Full assertion checking - Side-channel analysis

Purpose: Thorough security validation

10.4 Gate-Level Regression (Pre-Tapeout)

Runtime Target: < 24 hours **Tests**: - All functional and security tests - Gate-level netlist simulation - Timing-aware simulation - SDF annotation (if available)

Purpose: Validate synthesis didn't break functionality

---

11. Success Criteria

11.1 Functional Success Criteria

- All directed tests pass (100%)
- All crypto test vectors pass (100%)
- Code coverage: Line 99%, Branch 99%, FSM 100%
- Functional coverage: 95% of defined coverpoints hit
- Zero critical or high-severity bugs open
- All assertions pass (zero failures)

11.2 Security Success Criteria

- DUS isolation formally verified or exhaustively tested
- Key isolation formally verified or exhaustively tested
- Zeroization verified in 10 clock cycles
- Fault injection: 100% of faults handled safely
- No correlation between keys and bus outputs (statistical test passes)
- PUF regeneration success rate: 100% within error threshold
- All security assertions pass

11.3 Documentation Success Criteria

- Coverage report generated and reviewed
- Test results report published
- Known limitations documented
- Security analysis report completed
- Verification closure sign-off obtained

---

12. Risk Assessment

12.1 High-Risk Areas

Risk	Impact	Mitigation
Key leakage through bus	CRITICAL	Formal verification, extensive monitoring, isolation assertions
PUF unreliability	HIGH	Extended reliability testing, error injection, boundary testing
Zeroization incomplete	CRITICAL	Force/probe key registers, verify all storage elements cleared
Fault injection bypass	HIGH	Comprehensive fault injection campaign, multi-bit faults
State machine deadlock	MEDIUM	FSM coverage, timeout detection, formal liveness checks
Crypto algorithm bugs	HIGH	NIST test vectors, reference model comparison

12.2 Areas Requiring Extra Scrutiny

1. **PUF_DUS module**: Error correction robustness
2. **Secure_Key_Distributor**: Integrity checking and zeroization
3. **KDF_module**: Key derivation correctness
4. **Initialization state machine**: All transition paths
5. **Interface isolation**: DUT internal signals vs. bus-accessible signals

13. Tools and Resources

13.1 Required Tools

- **Simulator**: Any SystemVerilog/UVM-capable simulator (Questa, VCS, Xcelium)
- **Coverage Tools**: Built-in coverage or standalone coverage analyzer
- **Waveform Viewer**: For debug (VaporView recommended by user)
- **Formal Verification** (optional): For isolation proofs
- **Crypto Libraries**: OpenSSL or Python cryptography for reference models

13.2 Resource Requirements

- **Verification Engineers**: 2-3 engineers
- **Timeline**: 8-12 weeks for full verification
  - Weeks 1-2: Environment setup, base tests
  - Weeks 3-6: Directed tests, crypto validation
  - Weeks 7-9: Security tests, fault injection
  - Weeks 10-12: Coverage closure, regression, documentation
- **Compute Resources**: Regression farm for parallel test execution

---

14. Schedule and Milestones

Milestone	Week	Deliverable
M1: Environment Ready	Week 2	UVM testbench compiled, base sequences working
M2: Crypto Tests Pass	Week 4	All NIST test vectors pass
M3: Functional Complete	Week 6	All directed functional tests pass
M4: Security Tests Pass	Week 9	Isolation verified, fault injection complete
M5: Coverage Closure	Week 11	95% functional coverage, 99% code coverage
M6: Verification Signoff	Week 12	All success criteria met, documentation complete

---

15. Appendices

Appendix A: Test Vector Sources

- **SHA-256**: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values
- **HMAC**: https://tools.ietf.org/html/rfc4231
- **AES**: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

Appendix B: Acronyms

- **AES**: Advanced Encryption Standard
- **CTR**: Counter Mode
- **DUS**: Device Unique Secret
- **FSM**: Finite State Machine
- **HMAC**: Hash-based Message Authentication Code
- **HRoT**: Hardware Root of Trust
- **KDF**: Key Derivation Function
- **PUF**: Physical Unclonable Function
- **SHA**: Secure Hash Algorithm
- **UVM**: Universal Verification Methodology

Appendix C: References

1. NIST FIPS 180-4: Secure Hash Standard (SHS)
2. NIST FIPS 197: Advanced Encryption Standard (AES)
3. NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation
4. NIST SP 800-108: Recommendation for Key Derivation Functions
5. IEEE 1800-2017: SystemVerilog Language Reference Manual
6. Accellera UVM 1.2 User Guide

Document Control

Version	Date	Author	Changes
1.0	2024	Cognichip Verification Team	Initial verification plan

---

END OF VERIFICATION PLAN