EchoPulse Formal Verification Plan: Toolchain Mapping and Modular Proof Strategy

1. Abstract

This document outlines a comprehensive plan for the formal verification of the EchoPulse Key Encapsulation Mechanism (KEM). Given EchoPulse's novel symbolic, non-algebraic design, traditional proof methods may be insufficient. This plan proposes a multi-toolchain approach leveraging Tamarin, CryptoVerif, and EasyCrypt to rigorously verify critical security properties including trace properties, replay safety, and indistinguishability (IND-CCA2) under the Symbolic Graph Path Unpredictability (SGPU) assumption. The strategy decomposes EchoPulse into modular units, specifying individual verification goals and mapping them to the most suitable formal methods tools.

2. Tool Mapping Table (Formal Tool Architect)

| Tool / Category | Primary Verification Goal(s) | Suitability Symbol Path Coverage Testing for EchoPulse: Experimental Validation of Graph Mutation Dynamics

1. Introduction & Purpose

This document provides a formal specification for experimentally validating the robust dynamics and path coverage of the EchoPulse Key Encapsulation Mechanism (KEM). Given EchoPulse's reliance on a novel symbolic, non-algebraic design featuring deterministic graph transitions and mutating path-based key generation, empirical validation is paramount. The primary purpose of this testing is to demonstrate that repeated key encapsulations, across multiple sessions and evolving through the mutation schedule, generate a sufficiently diverse set of symbol-driven graph paths, resulting in high uniqueness of visited states and edges, unpredictable symbol reuse patterns, and effective mutation diffusion. This directly underpins the integrity and supports the assumptions of the Symbolic Graph Path Unpredictability (SGPU) assumption, crucial for EchoPulse's overall security.

2. Test Harness Description (Symbolic Graph Testing Architect)

The test harness will simulate a large number of EchoPulse encapsulation sessions, systematically recording data pertinent to graph traversal.

2.1. Session-Based Testing Setup:

For N total test sessions:

1. **System Initialization:**
   - Load the initial public parameters: base graph $G0(V,E)$, predefined initial state $vinitial \in V$, and the deterministic, time-dependent mutation schedule function $\mu: N \to Transform(G)$.
   - Establish a global, cryptographically secure pseudo-random number generator (CSPRNG) seeded with a **fixed, known seed** for the entire test run. This ensures deterministic repeatability of the experiment. All subsequent "random" values (e.g., payloads r) will be derived from this single, reproducible seed.
2. **Per-Session Execution Loop (for** $i=1$ **to** N**):**
   - **Logical Time Step Determination:** A logical time step ti will be assigned to each session. This can be:
     - **Sequential:** $ti=i-1$ (simple increment).
     - **Sampled:** $ti \leftarrow Uniform(0,Tmax)$ (random sampling within a defined maximum time horizon Tmax). The latter provides better coverage of different mutated graph instances.
   - **Graph Mutation Application:** Apply the mutation function μ to the base graph G0 to obtain the graph instance for the current session's time step: $Gti = \mu(G0,ti)$.
   - **Payload Generation:** Generate a cryptographically random payload $ri \in R$ using the established CSPRNG. The length of ri is fixed (e.g., 256 bits, parsed into 32 8-bit symbols).
   - **Path Traversal Simulation:** Simulate the core EchoPulse encapsulation logic:
     - Initialize the CurrentState to vinitial in the context of Gti.
     - Initialize empty lists for TraversedStates_Seq, TraversedEdges_Seq, and AppliedSymbols_Seq for the current session.
     - For each symbol sj in the parsed payload ri:
       - Compute NextState = $\delta(CurrentState,sj)$ using the transition function of Gti.
       - Record (CurrentState, NextState, s_j) as a tuple in TraversedEdges_Seq.
       - Add NextState to TraversedStates_Seq.
       - Add s_j to AppliedSymbols_Seq.

        - Update CurrentState = NextState.
    - The final state reached is venc=CurrentState.
  - **Session Data Collection:** Store all relevant data for the current session for post-processing.

2.2. Test Harness Structure (Pseudocode / High-Level Logic):

Code-Snippet

```
FUNCTION RunEchoPulsePathCoverageTest(N_sessions, T_max,
InitialCSPRNG_Seed):
 // Global Initialization
 CSPRNG.SetSeed(InitialCSPRNG_Seed)
 GlobalGraph_G0 = EchoPulse.InitializeBaseGraph() // G_0(V,E)
 MutationSchedule_Mu = EchoPulse.LoadMutationSchedule() // The function μ

 SET RecordedSessionData = [] // List to store results for all sessions

 FOR session_id FROM 1 TO N_sessions:
  // Determine the logical time step for this session
  CurrentTime = CSPRNG.SampleUniformInteger(0, T_max) // Random time within
range

  // Apply mutation for the current time step
  CurrentGraph_Gt = MutationSchedule_Mu(GlobalGraph_G0, CurrentTime)

  // Generate random payload for this session
  RandomPayload_R = CSPRNG.GenerateBytes(PAYLOAD_LENGTH_BYTES) // e.g., 32
bytes for 256 bits

  // Simulate EchoPulse path traversal
  CurrentState = GlobalGraph_G0.InitialState // Always start from v_initial
  SET SessionTraversedStates = [CurrentState]
  SET SessionTraversedEdges = [] // Stores (from_v, to_v, symbol)
  SET SessionAppliedSymbols = []
```

```
  FOR EACH symbol_byte IN RandomPayload_R: // Iterate byte by byte if symbols are
bytes
    SET NextState = CurrentGraph_Gt.TransitionFunction(CurrentState, symbol_byte)
    SessionTraversedStates.Add(NextState)
    SessionTraversedEdges.Add((CurrentState, NextState, symbol_byte))
    SessionAppliedSymbols.Add(symbol_byte)
    CurrentState = NextState

  // Record data for the current session
  RecordedSessionData.Add({
    "SessionID": session_id,
    "TimeStep": CurrentTime,
    "PayloadHash": Hash(RandomPayload_R), // Store hash to save space
    "TraversedStates": SessionTraversedStates,
    "TraversedEdges": SessionTraversedEdges,
    "AppliedSymbols": SessionAppliedSymbols
  })

 RETURN RecordedSessionData // For subsequent metric calculation and analysis
END FUNCTION
```

3. Metric Definitions (Metric Designer for Symbolic Path Evaluation)

These metrics empirically assess the path coverage and dynamism of EchoPulse, supporting the SGPU assumption.

- **Unique State Visit Ratio (**UVRS**):**
  - **Definition:** The proportion of unique states in the entire state space V that have been visited at least once across all N simulated sessions.
  - **Computation:** $UVRS = |V|\,|\bigcup_{i=1}^{N}\{\text{states in SessionTraversedStates}_i\}|$.
  - **Rationale for SGPU:** A high UVRS (approaching 1.0) signifies that the combination of random payloads and graph mutations effectively explores a large portion of the state space. This reduces the ability of an adversary to predict or narrow down potential final states (venc) by constraining reachable states, directly bolstering the unpredictability aspect of SGPU.
- **Unique Edge Traversal Ratio (**UVRE**):**
  - **Definition:** The proportion of unique state-symbol-next_state transitions (edges) that have been traversed at least once across all N sessions, relative

to the total set of unique edges that *could* be present across all mutated graphs G_t encountered.
  - **Computation:** $UVRE = \frac{|\bigcup_{t \in \{t_1, \dots, t_N\}} E_{G_t}|}{|\bigcup_{i=1}^{N} \{\text{edges in } SessionTraversedEdges_i\}|}$. The denominator represents the union of all unique edges present in all graphs G_t explored within the T_{max} range.
  - **Rationale for SGPU:** A high UVRE implies that the specific transitions taken are highly diverse. This means an adversary cannot easily infer the sequence of states by analyzing common or predictable edge traversals, further strengthening the path unpredictability aspect of SGPU.
- **Symbol Reuse Frequency (SRF) & Entropy:**
  - **Definition:** Measures the distribution and unpredictability of symbol usage across paths. It comprises two sub-metrics:
    1. **Global Symbol Distribution Entropy ($H(\Sigma_{used})$):** Shannon entropy of the empirical probability distribution of all symbols observed in AppliedSymbols_Seq across all sessions.
    2. **Contextual Symbol Uniqueness:** The average number of unique (v, s, v') triplets observed for each distinct symbol s.
  - **Computation:**
    1. Aggregate all symbols from all SessionAppliedSymbols lists. Compute the frequency $f(s)$ for each $s \in \Sigma$. Calculate $P(s) = f(s) / \sum f(s)$. $H(\Sigma_{used}) = -\sum_{s \in \Sigma} P(s) \log_2 P(s)$.
    2. For each unique triplet (v, s, v') observed, count its occurrences. Compute the average unique (v, s, v') for a given s.
  - **Rationale for SGPU:** A high $H(\Sigma_{used})$ indicates that all symbols contribute fairly to path generation, preventing an adversary from focusing on a limited, high-frequency subset. High contextual uniqueness (many different v' for a given (v, s) across different mutations) means the path is hard to guess even with some partial knowledge.
- **Mutation Diffusion Factor (MDF):**
  - **Definition:** Quantifies the extent and impact of the graph mutation function μ on the graph structure over time, ensuring it genuinely creates new traversal dynamics rather than superficial changes.
  - **Computation:** For a pair of graphs G_t and $G_{t'}$ where $t_o = t'$ (e.g., consecutive time steps $t' = t+1$, or randomly sampled distinct time steps):
    1. **Edge Divergence ($MDF_E(t, t')$):** $\frac{|E_{G_t} \Delta E_{G_{t'}}|}{|E_{G_t} \cup E_{G_{t'}}|} = \frac{|(E_{G_t} \setminus E_{G_{t'}}) \cup (E_{G_{t'}} \setminus E_{G_t})|}{|E_{G_t}| + |E_{G_{t'}}| - |E_{G_t} \cap E_{G_{t'}}|}$.

Compute average MDFE over many random pairs (t,t').

2. **Path Divergence (**$MDFP(t,t')$**):** For a large sample K of random (v_{start} ,symbol) pairs, calculate $v_{next,t}=\delta_{G_t}(v_{start},symbol)$ and $v_{next,t'}=\delta_{G_{t'}}(v_{start},symbol)$. $MDFP(t,t')$ is the proportion of these pairs where $v_{next,t} \neq v_{next,t'}$. Compute average MDFP.

- **Rationale for SGPU:** High average MDFE and MDFP values (e.g., above 0.5 for a significant portion of the graph) confirm that μ substantially alters the graph's connectivity. This is vital for SGPU, as it prevents an adversary from pre-computing paths or relying on long-term static graph properties, forcing them to always consider the current mutated graph state, which is hard to predict without the exact time step and mutation logic. This directly ensures independence of graph structure over time concerning path derivation.

4. Mutation Pattern Verifier

This role focuses on ensuring that the mutation function μ leads to meaningfully new and unpredictable traversal opportunities, rather than trivial or repetitive changes.

- **Goal:** Verify that $\mu(G,t)$ genuinely creates new traversal paths and sufficient divergence from G_0 and G_{t-1}.
- **Heuristics for Independence of Graph Structure over Time:**
  - **Non-trivial Edge Changes:** Ensure that the MDFE is consistently high across consecutive time steps. A low MDFE would indicate a "stale" mutation.
  - **Path Divergence on Fixed Inputs:** Take a set of *fixed* (initial state, symbol sequence) pairs. Simulate their traversal on $G_0, G_1, G_2, \ldots, G_T$. Observe how rapidly the resulting TraversedStates_Seq (and especially v_{enc}) diverge. A rapid divergence indicates effective mutation.
  - **Cycle Analysis (Qualitative):** Ensure that μ does not introduce short, easily detectable cycles or strong attractors that could lead to predictable loops in the graph. While complex to quantify, statistical analysis of observed path lengths and cycle occurrences can provide insight.
  - **"Randomness" of Changes:** While μ is deterministic, the *effect* of μ on the graph should appear pseudo-random to an adversary without knowledge of μ's internal workings. This can be evaluated by observing the distribution of edge changes and comparing it to a truly random edge rewiring.
  - **Impact on** v_{enc} **Distribution:** Verify that the distribution of final states v_{enc} is close to uniform across the state space V over many sessions, even if the initial state v_0 is fixed. This confirms that the mutation combined with random payloads disperses outcomes effectively.

5. Export Format Plan (Strategic Integrator)

To facilitate reproducibility, collaboration, and integration into larger research efforts, all outputs will adhere to well-defined, open formats.

- **Test Harness Source Code:**
  - **Python:** Preferred for rapid prototyping, data handling, and integration with data science tools (e.g., pandas, numpy, networkx). Output: .py files.
  - **Rust:** For performance-critical simulations and production-grade code. Output: .rs files.
- **Raw Session Data:**
  - **JSON Lines (.jsonl):** Each line is a JSON object representing one session. Facilitates incremental processing.
  - **CSV (.csv):** Simpler for direct spreadsheet analysis.
- **Aggregated Metrics:**
  - **JSON (.json):** For structured, machine-readable summaries.
  - **CSV (.csv):** For quick tabular overviews.
- **Visualization Data:**
  - **JSON (.json) or CSV (.csv):** Specifically structured for plotting (e.g., time-series data for MDF).
- **Formal Verification Tool Inputs:**
  - **Tamarin:** .spthy files for specifying protocols, traces, and properties (e.g., for replay safety, indistinguishability of traces).
  - **CryptoVerif:** .cv or .ocv files. Modular .ocv files can be generated for init, encaps, decaps blocks. These will represent the KEM's operations in a process calculus-like notation.
  - **EasyCrypt:** .ec files for specifying cryptographic games, assumptions, and proof steps. Requires a bridge to formalize the SGPU assumption within its logic. Potentially, the graph structure might be abstracted.

6. Suggested Work Schedule (Strategic Integrator)

This schedule outlines a phased approach to implementing the formal verification plan.

Phase 1: Foundation & Test Harness (Weeks 1-4)

- **Week 1:** Finalize exact mathematical definitions of δ, μ, and graph representation.
- **Week 2:** Develop core test harness in Python (or Rust if performance is primary concern from start). Implement RunEchoPulsePathCoverageTest and basic data

logging.

- **Week 3:** Implement initial versions of UVRS, UVRE, and SRF metrics computation.
- **Week 4:** Conduct small-scale pilot runs. Refine test harness and data collection based on initial observations. Prepare output formats.

Phase 2: Mutation Dynamics & Tool Preparation (Weeks 5-8)

- **Week 5:** Implement MDF metrics (MDFE, MDFP).
- **Week 6:** Conduct large-scale path coverage tests (N≈106 sessions). Analyze results for UVRS, UVRE, SRF, MDF.
- **Week 7:** Begin formalization for Tamarin. Model KeyGen, Encaps, Decaps as Tamarin rules. Focus on trace properties (e.g., replay attacks).
- **Week 8:** Begin formalization for CryptoVerif. Define init, encaps, decaps processes. Map secret components for indistinguishability.

Phase 3: Formal Proof Integration & Refinement (Weeks 9-12)

- **Week 9:** Attempt first Tamarin proofs (e.g., replay safety). Analyze failure traces, refine model.
- **Week 10:** Attempt first CryptoVerif IND-CPA/CCA1 proofs. Identify where the SGPU assumption must be explicitly introduced.
- **Week 11:** Explore EasyCrypt for modeling SGPU. Potentially, this might be a manual proof with EasyCrypt assistance for logical steps.
- **Week 12:** Integrate proof components. Identify remaining gaps for IND-CCA2. Document limitations and open questions. Draft the initial formal proof document (EchoPulse v2 Security Proof).

This plan provides a structured roadmap for the rigorous formal verification of EchoPulse, combining empirical validation with multi-tool formal proof techniques to ensure its cryptographic soundness.