
EchoPulse Replay Verifier CLI: Deterministic Path Validation for Symbolic KEM Sessions

1. Purpose

The "EchoPulse Replay Verifier CLI" is a command-line interface (CLI) tool designed to assist in the debugging, testing, and security analysis of EchoPulse Key Encapsulation Mechanism (KEM) implementations. Its primary purpose is to deterministically validate the decapsulation process of an EchoPulse ciphertext (CT) using a given private key (SK) within a specific mutation context (graph_id, mutation_index).

Beyond basic decapsulation, the tool aims to:

- Verify the consistency of the symbolic graph's state transitions and mutation across different "sessions" (mutation contexts).
- Detect and warn about potential replay attacks by identifying if a decapsulated final state (v_dec) has been observed before within a known set of session outputs.
- Provide granular debugging information, such as per-symbol path divergence, in case of a mismatch.

2. Inputs and CLI Syntax

The tool will be implemented in Python (echopulse_verify.py) and executed via the command line.

2.1. Basic Syntax:

Bash

```
python echopulse_verify.py --sk <private_key_file> --ct <ciphertext_data> --graph <graph_identifier> --mutation-index <session_index> [options]
```

2.2. Input Parameters:

- **--sk <file_path> (Required)**
  - **Description:** Path to the file containing the EchoPulse private key. The file should contain the raw binary representation of the secret key (e.g., 32 bytes for a 256-bit SK).
  - **Format:** Raw binary file.
  - **Example:** --sk my_private_key.bin
- **--ct <data> (Required)**
  - **Description:** The EchoPulse ciphertext to be verified. This can be provided either as a hexadecimal string or a path to a binary file.
  - **Format:** Hexadecimal string (prefixed with 0x) or file path.
  - **Example:** --ct 0x00010203... or --ct ciphertext.bin
- **--graph <identifier> (Required)**
  - **Description:** The identifier for the base symbolic graph (graph_id) used in the EchoPulse session. This maps to the graph_id negotiated via the echopulse_parameters TLS extension.
  - **Format:** Integer (decimal or hexadecimal, e.g., 1 or 0x0001).
  - **Example:** --graph 1 or --graph 0x0001
- **--mutation-index <index> (Required)**
  - **Description:** The session's mutation index (t). This value, often derived from the TLS handshake transcript, dictates the specific mutated graph Gt used for this session. It also implicitly selects the mutation_schedule_id if the graph definition implies it.
  - **Format:** Integer (decimal).
  - **Example:** --mutation-index 123
- **--session-record <file_path> (Optional)**
  - **Description:** Path to a JSON file containing a history of past session outputs (e.g., v_dec values, session_id, mutation_index). This is used to check for replay warnings.
  - **Format:** JSON array of objects, e.g., [{"session_id": 1, "t": 0, "v_dec": "0x1234"}, {"session_id": 2, "t": 1, "v_dec": "0x5678"}]
  - **Example:** --session-record session_history.json
- **--verbose (Optional)**
  - **Description:** Enable verbose output, showing intermediate steps and debugging information.
  - **Example:** --verbose
- **--output-json (Optional)**

- **Description:** Output results in JSON format rather than human-readable text.
- **Example:** --output-json

3. Output Format

The tool provides output primarily to the standard output. The default format is human-readable text. If --output-json is specified, the output will be a JSON object.

3.1. Default Human-Readable Output:

```
EchoPulse Replay Verifier Results:

Input Parameters:
  Private Key File:       my_private_key.bin
  Ciphertext:             0x...
  Graph ID:               1
  Mutation Index (t):     123

Decapsulation Result:
  Validation:             [SUCCESS | FAILED]
  Derived KEM Shared Secret: [K_EP_HEX_VALUE]
  Final Decapsulated State (v_dec): 0x[V_DEC_HEX_VALUE]

Replay Analysis:
  Replay Warning:         [True | False]
  Matching Previous Session(s): [Session ID(s) if True]
  If no --session-record is provided, replay analysis is skipped.

[Optional: Verbose Output]
  Trace of Private Key Path (v_priv): v0 -> v1 -> ... -> v_priv (0xABCD)
  Trace of Ciphertext Path (r): v_priv -> v_enc_calc (0xWXYZ)
  Expected v_enc (from CT): 0x[V_ENC_FROM_CT_HEX_VALUE]
  Path Comparison:
    Per-symbol drift (if mismatch):
```

```
    Symbol 0: OK (Graph: 0x.., Expected: 0x..)
    Symbol 1: MISMATCH (Graph: 0x.., Expected: 0x..)
    ...
  Total divergence index: X
```

3.2. JSON Output (with --output-json):

JSON


```
{
  "status": "success",
  "validation_result": true,
  "derived_shared_secret_kem":
"4B901C95E15469F5D211A41818B0091C5C7A28751480D6D1B482208151D87661",
  "final_decapsulated_state_v_dec": "0x1A2B",
  "replay_analysis": {
    "replay_warning": false,
    "matching_sessions": []
  },
  "debug_info": {
    "private_key_path_trace": "v0->v1->...->v_priv(0xABCD)",
    "ciphertext_path_trace": "v_priv->v_enc_calc(0xWXYZ)",
    "expected_v_enc_from_ct": "0xEF01",
    "path_comparison_details": [
      {"symbol_idx": 0, "status": "ok", "graph_output": "0x..", "expected_output": "0x.."},
      {"symbol_idx": 1, "status": "mismatch", "graph_output": "0x..", "expected_output": "0x.."}
    ],
    "total_divergence_index": 5
  },
  "error_message": null
}
```


(Note: error_message would be populated if status is "error".)

4. API Hook Option (as Python Module)

The core logic of the CLI tool will be encapsulated within a Python module, allowing it to be imported and used programmatically in other applications or test suites.

Python

```python
# echopulse_verify/verifier.py

class EchoPulseVerifier:
    def __init__(self, graph_manager, kem_core):
        """
        Initializes the verifier with dependencies for graph management
        and KEM core operations.
        :param graph_manager: An object/module handling graph definition and mutation.
        :param kem_core: An object/module implementing EchoPulse KEM operations.
        """
        self.graph_manager = graph_manager
        self.kem_core = kem_core

    def verify_ciphertext(self, sk_bytes: bytes, ct_bytes: bytes,
                          graph_id: int, mutation_index: int,
                          session_history: list = None, verbose: bool = False) -> dict:
        """
        Performs the decapsulation and verification of an EchoPulse ciphertext.

        :param sk_bytes: Raw bytes of the EchoPulse private key.
        :param ct_bytes: Raw bytes of the EchoPulse ciphertext (v_enc_id || symbolic_path_sequence).
        :param graph_id: The ID of the base graph (e.g., from TLS echopulse_parameters).
        :param mutation_index: The time index 't' for graph mutation.
        :param session_history: List of dicts with {"t": int, "v_dec": str} for replay analysis.
        :param verbose: If True, include detailed debugging information.
        :return: A dictionary containing verification results and debug info.
        """
        results = {
            "validation_result": False,
            "derived_shared_secret_kem": None,
            "final_decapsulated_state_v_dec": None,
            "replay_analysis": {"replay_warning": False, "matching_sessions": []},
            "debug_info": {},
```

```
            "error_message": None
        }

        try:
            # 1. Reconstruct G_t based on graph_id and mutation_index
            current_graph_state = self.graph_manager.get_mutated_graph(graph_id, mutation_index)

            # 2. Decapsulate CT to derive K_EP and v_dec_calc
            #    This includes:
            #    a. Deriving v_priv from SK
            #    b. Applying symbolic_path_sequence (r) to G_t starting from v_priv
            #       to get v_enc_calc (the calculated final state).
            #    c. Deriving K_EP from v_enc_id (from CT) and symbolic_path_sequence (r).
            #       (Note: K_EP derivation often uses H(v_enc_id || r) for EchoPulse)

            v_enc_from_ct, symbolic_path_r = self.kem_core.parse_ct(ct_bytes)

            # Step a: Derive v_priv from SK using initial graph G_0 or a known SK->v_priv mapping
            # This requires access to the EchoPulse SK generation logic, which is out of scope for this verifier's core.
            # Assume kem_core can map SK to v_priv (initial state for client's path)
            v_priv = self.kem_core.derive_private_state(sk_bytes, graph_id) # v_priv depends on SK and G_0

            # Step b: Traverse the mutated graph G_t with symbolic_path_r from v_priv
            v_dec_calc, path_trace_detailed = self.graph_manager.traverse_path(
                current_graph_state, v_priv, symbolic_path_r, verbose=verbose
            )

            # Step c: Derive the KEM shared secret (K_EP)
            derived_kem_secret = self.kem_core.derive_kem_secret(v_enc_from_ct, symbolic_path_r)

            results["derived_shared_secret_kem"] = derived_kem_secret.hex()
            results["final_decapsulated_state_v_dec"] = hex(v_dec_calc)

            # 3. Path Comparison (Path Comparison Debugger)
```

```
            if v_dec_calc == v_enc_from_ct:
                results["validation_result"] = True
            else:
                results["validation_result"] = False
                results["debug_info"]["expected_v_enc_from_ct"] = hex(v_enc_from_ct)
                results["debug_info"]["path_comparison_details"] = self._compare_paths(
                    path_trace_detailed, current_graph_state, v_enc_from_ct,
symbolic_path_r
                )
                results["debug_info"]["total_divergence_index"] = self._calculate_divergence(
                    results["debug_info"]["path_comparison_details"]
                )

            # 4. Replay Analysis (Determinism and Replay Validator)
            if session_history:
                for record in session_history:
                    if record.get("v_dec") and hex(v_dec_calc) == record["v_dec"] and \
                       record.get("t") and record["t"] != mutation_index:
                        results["replay_analysis"]["replay_warning"] = True
                        results["replay_analysis"]["matching_sessions"].append(record["session_id"])
                        # Optional: check if mutation_index itself was reused, which implies non-unique KEM
output
                        if record["t"] == mutation_index:
                            results["replay_analysis"]["warning_mutation_reused"] = True

        except Exception as e:
            results["status"] = "error"
            results["error_message"] = str(e)

        return results

    def _compare_paths(self, path_trace_detailed, graph_state, expected_v_enc, symbolic_path_r):
        """Compares the calculated path with the expected final state for debugging."""
        comparison_details = []
        # Logic to iterate through path_trace_detailed and compare transitions/outputs
        # against a simulated path using symbolic_path_r on graph_state
        # and checking if it deviates from expected_v_enc.
        # This is highly dependent on how graph_state and path_trace_detailed are structured.
```

```
        # Placeholder for actual comparison logic.
        return comparison_details

    def _calculate_divergence(self, comparison_details):
        """Calculates a numerical divergence index based on path comparison."""
        # Sum of mismatch counts, first mismatch index, etc.
        return sum(1 for d in comparison_details if d["status"] == "mismatch")

# Example usage (simplified, assumes mock graph_manager and kem_core)
if __name__ == "__main__":
    import argparse
    # Mock implementations for demonstration
    class MockGraphManager:
        def get_mutated_graph(self, graph_id, mutation_index):
            print(f"Mock: Getting mutated graph for ID {graph_id}, index {mutation_index}")
            # Return a simplified graph state for testing
            return {"nodes": [], "edges": []}
        def traverse_path(self, graph, start_node, symbolic_path, verbose):
            print(f"Mock: Traversing path from {start_node} with {symbolic_path.hex()}")
            # Simulate a path traversal. Return a dummy v_dec and path trace
            return 0x1A2B, [{"node": 0, "symbol": 0x01}, {"node": 10, "symbol": 0x02}]

    class MockKemCore:
        def parse_ct(self, ct_bytes):
            v_enc_id = int.from_bytes(ct_bytes[:2], 'big')
            symbolic_path_sequence = ct_bytes[2:]
            print(f"Mock: Parsing CT. v_enc_id: {v_enc_id}, path: {symbolic_path_sequence.hex()}")
            return v_enc_id, symbolic_path_sequence
        def derive_private_state(self, sk_bytes, graph_id):
            print(f"Mock: Deriving private state from SK {sk_bytes.hex()}")
            return 0 # Dummy starting node

        def derive_kem_secret(self, v_enc_id, symbolic_path_r):
            print(f"Mock: Deriving KEM secret from v_enc_id {v_enc_id} and path
{symbolic_path_r.hex()}")
            # Return a dummy KEM secret
            return
bytes.fromhex("4B901C95E15469F5D211A41818B0091C5C7A2875148OD6D1B482208151D87661")
```

```
    parser = argparse.ArgumentParser(description="EchoPulse Replay Verifier CLI.")
    parser.add_argument("--sk", type=str, required=True, help="Path to the private key file.")
    parser.add_argument("--ct", type=str, required=True, help="Ciphertext (hex string or file path).")
    parser.add_argument("--graph", type=lambda x: int(x, 0), required=True, help="Graph ID (decimal or hex).")
    parser.add_argument("--mutation-index", type=int, required=True, help="Mutation index (t).")
    parser.add_argument("--session-record", type=str, help="Path to JSON file with session history for replay analysis.")
    parser.add_argument("--verbose", action="store_true", help="Enable verbose output.")
    parser.add_argument("--output-json", action="store_true", help="Output results in JSON format.")
    args = parser.parse_args()

    # Load SK
    with open(args.sk, 'rb') as f:
        sk_bytes_arg = f.read()

    # Parse CT
    ct_bytes_arg = None
    if args.ct.startswith('0x'):
        ct_bytes_arg = bytes.fromhex(args.ct[2:])
    else:
        with open(args.ct, 'rb') as f:
            ct_bytes_arg = f.read()

    # Load session history if provided
    session_history_arg = None
    if args.session_record:
        import json
        with open(args.session_record, 'r') as f:
            session_history_arg = json.load(f)

    verifier = EchoPulseVerifier(MockGraphManager(), MockKemCore())
    results = verifier.verify_ciphertext(
```

```
        sk_bytes_arg, ct_bytes_arg, args.graph, args.mutation_index,
        session_history_arg, args.verbose
    )

    if args.output_json:
        import json
        print(json.dumps(results, indent=2))
    else:
        print("EchoPulse Replay Verifier Results:")
        print(f"  Validation: {results['validation_result']}")
        print(f"  Derived KEM Secret: {results['derived_shared_secret_kem']}")
        print(f"  Decapsulated State (v_dec): {results['final_decapsulated_state_v_dec']}")
        if args.session_record:
            print(f"Replay Warning: {results['replay_analysis']['replay_warning']}")
            if results['replay_analysis']['matching_sessions']:
                print(f"  Matching sessions: {results['replay_analysis']['matching_sessions']}")
        if args.verbose and results['debug_info']:
            print("\nDebug Info:")
            for key, value in results['debug_info'].items():
                print(f"  {key}: {value}")
```

5. Use Case Examples

5.1. Basic Decapsulation Verification:

Bash

```
# Assuming private_key.bin, ciphertext.bin exist
python echopulse_verify.py \
    --sk private_key.bin \
    --ct ciphertext.bin \
    --graph 1 \
    --mutation-index 123
```

Expected Output: Validation: SUCCESS (or FAILED if mismatch), derived KEM secret,

and v_dec.

5.2. Debugging a Decapsulation Failure (using verbose output):

Bash

```
python echopulse_verify.py \
  --sk private_key.bin \
  --ct 0xDEADBEEF... # Malformed or incorrect CT
  --graph 1 \
  --mutation-index 123 \
  --verbose
```

Expected Output: Validation: FAILED, followed by Debug Info showing path_comparison_details (e.g., where the symbolic path diverged from the expected v_enc_id from the CT).

5.3. Checking for Replay Attacks:

Bash

```
# Assuming valid inputs and session_history.json contains {"t": 100, "v_dec": "0x1A2B"}
python echopulse_verify.py \
  --sk private_key.bin \
  --ct valid_ciphertext_for_t_123.bin \
  --graph 1 \
  --mutation-index 123 \
  --session-record session_history.json
```

Expected Output:
If v_dec for t=123 is 0x1A2B, then:
Replay Warning: True
Matching Previous Session(s): [100]
This indicates that the same final state was derived in a previous session, which could signal a

replay if the CT itself is replayed.

5.4. Exporting Results in JSON Format:

Bash

```
python echopulse_verify.py \
  --sk private_key.bin \
  --ct ciphertext.bin \
  --graph 1 \
  --mutation-index 123 \
  --output-json > verification_results.json
```

Expected Output: A verification_results.json file with the structured JSON output.

6. Determinism and Replay Validation Logic (Determinism and Replay Validator)

The core validation logic is implemented within the verify_ciphertext method and its helpers.

- **δ-transition Determinism:** The graph_manager.traverse_path method implicitly validates this. Any non-deterministic behavior in the symbolic graph transition function ($\delta(v,s) \rightarrow v'$) would cause a mismatch between the calculated v_dec_calc and the v_enc_id extracted from the ciphertext. The _compare_paths method would highlight the exact symbol step where the calculated path deviated.
- **Mutation Consistency:** The tool assumes graph_manager.get_mutated_graph(graph_id, mutation_index) produces a *deterministic* graph Gt based on these inputs. This is crucial. If the same graph_id and mutation_index produce different graphs in different execution environments, this tool will flag validation_result: false. This implicitly verifies the $\mu(G,t)$ function.
- **Replay Warning:** The replay_analysis section specifically checks for two conditions:
  1. **Identical v_dec with different mutation_index (t):** If a new session generates the *exact same* final decapsulated state v_dec as a previous, *different* session (i.e., different t value), it triggers a replay_warning. This is a strong indicator of a potential replay attack, as the mutation function aims to make such identical outcomes highly improbable.

2. **Reused mutation_index:** If the mutation_index (i.e., the t value) itself is reused for a new KEM operation, this suggests a flaw in the session management (e.g., reusing an old TLS handshake transcript). While the EchoPulse KEM *might* still work (if the graph Gt is correctly generated), it breaks the liveness property of the t value and can open up other vulnerabilities if not handled by higher layers. This can be an optional warning (warning_mutation_reused).

This CLI tool provides an invaluable asset for ensuring the integrity and security of EchoPulse deployments by rigorously validating its core deterministic properties and offering insights into potential replay scenarios.