Code-Snippet

```
# echo_keygen_deterministic.rs ```rust // echo_keygen_deterministic.rs
use hmac::{Hmac, Mac}; use sha3::Sha3_256; use crate::echo_struct::
{Symbol, State, SymbolPath, EchoGraph}; use
crate::echo_keygen::EchoKeyPair; type HmacSha3_256 = Hmac<Sha3_256>;
pub fn echo_keygen_seeded(graph: &EchoGraph, seed: [u8; 32]) ->
EchoKeyPair { let mut mac =
HmacSha3_256::new_from_slice(&seed).expect("HMAC init failed"); let mut
sk_symbols = Vec::with_capacity(28); for _ in 0..28
{ mac.update(&[sk_symbols.len() as u8]); let result =
mac.finalize_reset(); sk_symbols.push(result.into_bytes()[0]); } let sk
= SymbolPath { symbols: sk_symbols }; let v_priv = graph.resolve(0,
&sk); let mut pk_symbols = Vec::with_capacity(28); let mut
current_state = v_priv; let mut mac_pk =
HmacSha3_256::new_from_slice(&seed).expect("HMAC init failed");
mac_pk.update(&[28u8]); // Offset for PK generation for i in 0..28
{ mac_pk.update(&[i as u8]); let result = mac_pk.finalize_reset(); let
next_sym = result.into_bytes()[0]; let idx = (next_sym % 16) as usize;
current_state = graph.transitions[current_state as usize][idx];
pk_symbols.push(next_sym); } let pk = SymbolPath { symbols:
pk_symbols }; EchoKeyPair { sk, pk } }
```

echo_decaps.rs

Rust

```
// echo_decaps.rs use sha3::{Digest, Sha3_256}; use
crate::echo_struct::{Symbol, State, SymbolPath, EchoGraph}; pub fn
echo_decaps(graph: &EchoGraph, sk: &SymbolPath, r: &SymbolPath) ->
Result<[u8; 32], &'static str> { if r.symbols.len() != 28 { return
Err("Invalid input length for r"); } let v_priv = graph.resolve(0, sk);
let v_dec = graph.resolve(v_priv, r); let v_dec_bytes = [(v_dec & 0xFF)
as u8, (v_dec >> 8) as u8]; let mut hasher = Sha3_256::new();
hasher.update(&v_dec_bytes); hasher.update(&r.symbols); let k_prime:
[u8; 32] = hasher.finalize().into(); Ok(k_prime) }
```

echo_test_deterministic.rs

Rust

```
// echo_test_deterministic.rs #[cfg(test)] mod tests { use
crate::echo_struct::{SymbolPath, EchoGraph, Mutator}; use
crate::echo_keygen::EchoKeyPair; use
crate::echo_keygen_deterministic::echo_keygen_seeded; use
crate::echo_encaps::echo_encaps; use crate::echo_decaps::echo_decaps;
fn create_test_graph() -> EchoGraph { let mut transitions = [[0u16;
16]; 256]; // Simple deterministic graph for testing for i in 0..256 {
for j in 0..16 { transitions[i][j] = (i + j) as u16; } } EchoGraph
{ transitions } } #[test] fn test_deterministic_keygen() { let graph =
create_test_graph(); let seed: [u8; 32] = [0u8; 32]; let keypair1 =
echo_keygen_seeded(&graph, seed); let keypair2 =
echo_keygen_seeded(&graph, seed); assert_eq!(keypair1.sk.symbols,
keypair2.sk.symbols); assert_eq!(keypair1.pk.symbols,
keypair2.pk.symbols); } #[test] fn test_deterministic_full_cycle() {
let graph = create_test_graph(); let seed: [u8; 32] = [0x42u8; 32]; let
keypair = echo_keygen_seeded(&graph, seed); let pk = &keypair.pk; let
sk = &keypair.sk; let encapsulation = echo_encaps(&graph, pk); let r =
&encapsulation.r; let k = encapsulation.k; let k_prime_result =
echo_decaps(&graph, sk, r); assert!(k_prime_result.is_ok()); let
k_prime = k_prime_result.unwrap(); assert_eq!(k, k_prime); println!
("Deterministic Full Cycle Test: OK"); } #[test] fn
test_decaps_invalid_r_length() { let graph = create_test_graph(); let
seed: [u8; 32] = [0x1a; 32]; let keypair = echo_keygen_seeded(&graph,
seed); let sk = &keypair.sk; let r_symbols: Vec<u8> = (0..27).map(|_|
rand::random::<u8>()).collect(); let r = SymbolPath { symbols:
r_symbols }; let result = echo_decaps(&graph, sk, &r); assert!
(result.is_err()); assert_eq!(result.err().unwrap(), "Invalid input
```