```
#[cfg(test)]
mod tests {
    use crate::echo_struct::{SymbolPath, EchoGraph, Mutator};
    use crate::echo_keygen::echo_keygen;
    use crate::echo_encaps::echo_encaps;
    use crate::echo_decaps::echo_decaps;

    #[test]
    fn test_echo_cycle() {
        let mutator = Mutator {
            seed: [0u8; 32],
            session_index: 0,
        };
        let graph = mutator.generate_graph();
        let keypair = echo_keygen(&graph);
        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 = echo_decaps(&graph, sk, r);

        if k == k_prime {
            println!("OK");
            // Optional:
```

```
            // let v_enc = (encapsulation.k[0] as u16) | ((encapsulation.k[1] as u16) << 8);
            // let v_dec = (k_prime[0] as u16) | ((k_prime[1] as u16) << 8);
            // println!("v_enc: {:04x}", v_enc);
            // println!("v_dec: {:04x}", v_dec);
        } else {
            println!("FAIL");
            // println!("K: {:?}", k);
            // println!("K': {:?}", k_prime);
        }
    }
}

mod echo_struct {
    pub type Symbol = u8;
    pub type State = u16;
    #[derive(Debug, Clone)]
    pub struct SymbolPath {
        pub symbols: Vec<Symbol>,
    }
    pub struct EchoGraph {
        pub transitions: [[State; 16]; 256],
    }
    impl EchoGraph {
        pub fn resolve(&self, start: State, path: &SymbolPath) -> State {
            let mut state = start;
            for sym in path.symbols.iter() {
                let idx = (sym % 16) as usize;
```

```
            state = self.transitions[state as usize][idx];
        }
        state
    }
  }
  pub struct Mutator {
    pub seed: [u8; 32],
    pub session_index: u32,
  }
  impl Mutator {
    pub fn generate_graph(&self) -> EchoGraph {
      EchoGraph {
        transitions: [[0; 16]; 256],
      }
    }
  }
}

mod echo_keygen {
  use rand::RngCore;
  use sha3::{Digest, Sha3_256};
  use crate::echo_struct::{Symbol, State, SymbolPath, EchoGraph};

  pub struct EchoKeyPair {
    pub sk: SymbolPath,
    pub pk: SymbolPath,
  }
```

```
    pub fn fingerprint(path: &SymbolPath) -> [u8; 32] {
        let mut hasher = Sha3_256::new();
        hasher.update(&path.symbols);
        hasher.finalize().into()
    }

    pub fn echo_keygen(graph: &EchoGraph) -> EchoKeyPair {
        let mut rng = rand::thread_rng();
        let sk_symbols: Vec<Symbol> = (0..28).map(|_| rng.next_u8()).collect();
        let sk = SymbolPath { symbols: sk_symbols };

        let v_priv = graph.resolve(0, &sk);

        let mut pk_symbols: Vec<Symbol> = Vec::with_capacity(28);
        let mut current_state = v_priv;
        for _ in 0..28 {
            let next_sym = rng.next_u8();
            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 }
    }
}
```

```
mod echo_encaps {
    use rand::RngCore;
    use sha3::{Digest, Sha3_256};
    use crate::echo_struct::{Symbol, State, SymbolPath, EchoGraph};

    pub struct EchoEncapsulation {
        pub r: SymbolPath,
        pub k: [u8; 32],
    }

    pub fn echo_encaps(graph: &EchoGraph, pk: &SymbolPath) -> EchoEncapsulation {
        let v_pub = graph.resolve(0, pk);
        let mut rng = rand::thread_rng();
        let r_symbols: Vec<Symbol> = (0..28).map(|_| rng.next_u8()).collect();
        let r = SymbolPath { symbols: r_symbols };

        let v_enc = graph.resolve(v_pub, &r);

        let v_enc_bytes = [(v_enc & 0xFF) as u8, (v_enc >> 8) as u8];

        let mut hasher = Sha3_256::new();
        hasher.update(&v_enc_bytes);
        hasher.update(&r.symbols);
        let k: [u8; 32] = hasher.finalize().into();

        EchoEncapsulation { r, k }
```

```
    }
}

mod echo_decaps {
    use sha3::{Digest, Sha3_256};
    use crate::echo_struct::{Symbol, State, SymbolPath, EchoGraph};

    pub fn echo_decaps(graph: &EchoGraph, sk: &SymbolPath, r: &SymbolPath) -> [u8; 32] {
        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);
        hasher.finalize().into()
    }
}
```