SMC Assignments – Garbled Circuits

Lorenzo Deflorian

February 10, 2026

1 Assignment 1: Mini Garbled Circuit

This assignment implements a minimal garbled NAND gate following Yao's Garbled Circuits construction. The implementation generates random wire labels, constructs a garbled gate table, and evaluates it to compute the NAND function without revealing input or output values.

1.1 Implementation Overview

The implementation consists of three main components:

1. **Wire Labels:** Each wire $w \in \{x, y, z\}$ has two random 128-bit labels representing 0 and 1:

```rust
pub struct WireLabels {
    pub zero: WireLabel,  // L_w^0
    pub one: WireLabel,   // L_w^1
}

impl WireLabels {
    pub fn random<R: RngCore + ?Sized>(rng: &mut R) -> Self {
        Self {
            zero: WireLabel::random(rng),
            one: WireLabel::random(rng),
        }
    }
}
```

2. **Garbling:** The garbler constructs the gate by encrypting output labels for all input combinations:

```rust
// Build the 4 ciphertexts C[a,b] = Enc(K[a,b], L^{a NAND b}_z)
for (a, b) in &[(0u8, 0u8), (0, 1), (1, 0), (1, 1)] {
    let in_x = if *a == 0 { &x_labels.zero } else { &x_labels.one };
    let in_y = if *b == 0 { &y_labels.zero } else { &y_labels.one };

    let nand_result = if *a == 1 && *b == 1 { 0 } else { 1 };
    let out_z = if nand_result == 0 { &z_labels.zero } else {
        &z_labels.one };

    let k = derive_key(in_x, in_y);  // K[a,b] = KDF(L_x^a || L_y^b)
    let c = encrypt_label(&k, out_z); // C[a,b] = Enc(K[a,b],
        L_z^{NAND(a,b)})
    table.push(c);
}
fastrand::shuffle(&mut table);  // Randomize order
```

3. **Evaluation:** The evaluator decrypts all ciphertexts until finding the valid output:

```rust
pub fn evaluate(&self, inputs: GarbledNandInputs) -> GarbledNandOutput {
    for c in &self.table {  // Try all 4 ciphertexts
        let k = derive_key(&inputs.x, &inputs.y);
        let decoded = decrypt_label(&k, c);

        if decoded == self.z_labels.zero || decoded == self.z_labels.one {
            return GarbledNandOutput { z: decoded };
        }
    }
    // Exactly one decryption succeeds
}
```

1.2 (a) Correctness: Why Exactly One Ciphertext Decrypts Correctly

During evaluation, the evaluator receives one label for wire x (denoted L_x^a where $a \in \{0,1\}$) and one label for wire y (denoted L_y^b where $b \in \{0,1\}$). The evaluator then attempts to decrypt all four ciphertexts in the garbled table using the derived key $K = \text{KDF}(L_x^a || L_y^b)$.

Exactly one ciphertext decrypts correctly because:

1. **The key derivation is deterministic:** For a given pair of input labels (L_x^a, L_y^b), the key derivation function $\text{KDF}(L_x^a || L_y^b)$ always produces the same key $K[a,b]$. This is implemented as:

```rust
fn derive_key(x: &WireLabel, y: &WireLabel) -> [u8; 32] {
    let mut hasher = Sha256::new();
    hasher.update(&x.0);  // L_x^a
    hasher.update(&y.0);  // L_y^b
    let result = hasher.finalize();
    // Returns K = SHA256(L_x^a || L_y^b)
}
```

2. **Each ciphertext is encrypted with a unique key:** During garbling, for each input combination $(a,b) \in \{0,1\}^2$, a distinct key $K[a,b] = \text{KDF}(L_x^a || L_y^b)$ is derived. This key is used to encrypt the corresponding output label $L_z^{\text{NAND}(a,b)}$, producing ciphertext $C[a,b]$.

3. **Only the matching key decrypts correctly:** When the evaluator derives $K = \text{KDF}(L_x^a || L_y^b)$ from their input labels, this key matches exactly one of the four keys used during garbling—specifically, $K[a,b]$ where (a,b) corresponds to the actual input values. Attempting to decrypt the other three ciphertexts with this key fails because they were encrypted with different keys ($K[0,0]$, $K[0,1]$, $K[1,0]$, or $K[1,1]$), resulting in random-looking decrypted values that do not match any valid output label.

4. **The output label is recognizable:** The garbler provides the two possible output labels L_z^0 and L_z^1. The evaluator can verify that exactly one decryption produces a label matching one of these two values, confirming correctness.

Therefore, the cryptographic properties of the key derivation function and encryption scheme ensure that only the ciphertext corresponding to the actual input combination (a,b) decrypts to a valid output label, while all other decryption attempts produce invalid values.

1.3 (b) Inefficiency: Why Four Decryptions Are Required

The evaluator must attempt to decrypt all four ciphertexts in the garbled table because:

1. **The table order is randomized:** After garbling, the four ciphertexts $\{C[0,0], C[0,1], C[1,0], C[1,1]\}$ are stored in random order. This randomization prevents the evaluator from learning which ciphertext corresponds to which input combination, preserving privacy.

2. **The evaluator cannot identify the correct ciphertext a priori:** Without knowing which input combination (a, b) their labels represent, the evaluator cannot determine which of the four ciphertexts is the correct one to decrypt. The only way to find the valid output is to try decrypting all entries.

3. **No early termination is possible:** Since the evaluator cannot distinguish valid from invalid decryptions until after attempting decryption, they must process all four entries. The first successful decryption (producing a label matching L_z^0 or L_z^1) indicates the correct output, but there is no way to know which entry will succeed without trying them all. This is evident in the evaluation code:

```rust
for c in &self.table {  // Must iterate through all 4 entries
    let k = derive_key(&inputs.x, &inputs.y);
    let decoded = decrypt_label(&k, c);

    if decoded == self.z_labels.zero || decoded == self.z_labels.one {
        // Found valid output, but had to try all entries
        return GarbledNandOutput { z: decoded };
    }
}
```

This inefficiency is inherent in this minimal construction. In optimized garbled circuit schemes, techniques such as point-and-permute or free-XOR can reduce the number of decryptions needed.