SMC Assignments – Garbled Circuits

Lorenzo Deflorian

February 21, 2026

1 Assignment 1: Mini Garbled Circuit

This assignment builds a tiny garbled circuit for a single NAND gate using Yao's construction. The code creates random wire labels, garbles the gate into a small garbled table, and then evaluates the table to compute NAND without exposing the underlying bits.

1.1 Implementation Overview

There are three moving pieces:

1. **Wire Labels:** Each wire $w \in \{x, y, z\}$ gets two fresh 128-bit labels, one for 0 and one for 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 encrypts the correct output label for every input pair:

```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 tries to decrypt each entry until one yields a valid output label:

```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

At evaluation time, the evaluator holds one label for x (L_x^a) and one for y (L_y^b), with $a, b \in \{0, 1\}$. From those two labels the evaluator derives a single key $K = \text{KDF}(L_x^a || L_y^b)$ and tries it against the four ciphertexts in the table.

Only one entry opens successfully, for these reasons:

1. **Deterministic key derivation:** The same pair of input labels always maps to the same key $K[a, b]$. In the code this is:

```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 table entry uses its own key:** During garbling we derive four keys, one per input pair, and encrypt the matching output label $L_z^{\text{NAND}(a,b)}$ under each key to form $C[a, b]$.

3. **Only the matching key works:** The derived key matches exactly one of the four garbling keys. Decrypting the other three entries with the wrong key produces garbage that will not equal any valid output label.

4. **Valid labels are recognizable:** The evaluator knows both possible output labels L_z^0 and L_z^1, so they can spot the one correct decryption.

So the cryptographic setup guarantees that exactly the ciphertext corresponding to the true inputs (a, b) yields a valid output label; all other attempts fail.

1.3 (b) Inefficiency: Why Four Decryptions Are Required

The evaluator has to try all four ciphertexts because:

1. **The table is shuffled:** The four ciphertexts are stored in random order to hide which entry matches which input pair.

2. **The evaluator cannot pick the right entry upfront:** The labels do not reveal the underlying bits, so there is no way to select the correct ciphertext without trying it.

3. **Success is only visible after decryption:** The evaluator only discovers the valid label after decrypting, so all entries must be checked. The loop in the code makes this explicit:

```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 is a known limitation of the bare-bones construction. Optimizations such as point-and-permute or free-XOR can cut down the number of decryptions.

2 Assignment 2: Garbled Circuits in Practice

Here the goal is privacy-preserving array equality. Alice and Bob each have a private boolean array of length n and want to learn only whether all positions match: $\bigwedge_i (a_i = b_i)$. The implementation provides two versions: a fully oblivious one and a faster early-exit (leaky) version that leaks limited information.

2.1 Implementation Overview

Both versions implement the same circuit. For each index i, equality $a_i = b_i$ is built as $(a_i \wedge b_i) \vee (\neg a_i \wedge \neg b_i)$ using NAND gates, and then all equalities are combined with an AND chain. The difference is in control flow: the oblivious version always runs through all n indices, while the early-exit version stops at the first mismatch.

2.2 Threat Model

We assume semi-honest (honest-but-curious) adversaries. The protocol **protects**:

- **Individual inputs:** Alice's array $A = (a_0, \ldots, a_{n-1})$ and Bob's array $B = (b_0, \ldots, b_{n-1})$ stay secret; only the final match/no-match bit is revealed.
- **Intermediate values:** In the oblivious version, per-index equality results are never decoded, only the final AND output.

The protocol does *not* protect against malicious parties who deviate from the protocol, nor does it address side channels (timing, power) beyond what is discussed below.

2.3 Leakage Analysis

Oblivious version: The oblivious implementation always processes all n indices. Control flow is data-independent, so runtime is (to first order) independent of the input values. The only thing revealed is the final boolean output.

Early-exit (leaky) version: The early-exit implementation stops as soon as it finds a mismatch. This creates **timing leakage**: anyone who can measure runtime can infer the *index of the first mismatch*. Formally, the leakage function is:

$$\mathcal{L}(A, B) = \min\{i : a_i \neq b_i\}$$

with the convention that if $A = B$, the leakage is n. From the runtime, an adversary learns how many indices were checked before the mismatch.

2.4 Trade-off Discussion

The early-exit (leaky) version intentionally reveals the index of the first mismatch in exchange for performance. The oblivious version must always build and evaluate the full AND chain over all n equality results; the early-exit version skips that work by decoding each equality as it goes and returning as soon as one fails. This makes runtime depend on how far the arrays match before they diverge, which is exactly the timing leakage described above. The trade-off is explicit: sacrifice index privacy for faster execution when mismatches occur early, which can matter in settings where arrays often differ near the start (for example, version hashes, checksums, or incremental comparisons).

When the additional leakage may be acceptable:

- **Non-adversarial settings:** Both parties are trusted and efficiency matters (for example, internal checks or debugging).
- **Index is non-sensitive:** The location of the mismatch is not confidential (for example, version comparisons where the value matters more than the position).
- **Controlled environments:** Timing channels are mitigated with constant-time code or padded delays.

When the leakage is unacceptable:

- **Privacy-critical applications:** The index itself is sensitive (for example, which digit differs in a password, or the position of a matching record in a database).
- **Adversarial settings:** One party may be compromised and timing measurements are feasible.
- **Regulatory compliance:** Strict privacy guarantees are required (no auxiliary leakage).

2.5 Performance Evaluation

Both versions were benchmarked for $n \in \{4, 8, 16\}$ with all elements equal (worst case for the early-exit version, since it has to check everything). Results are from a single run in debug mode:

n	Oblivious (s)	Leaky (s)
4	0.0017	0.0014
8	0.0032	0.0027
16	0.0065	0.0064

The oblivious version is slower because it builds and evaluates the full AND chain over all equalities. The early-exit version avoids that chain by decoding each equality and returning early. For arrays with an early mismatch, the speedup would be larger (for example, a mismatch at index 0 means minimal work). Both versions scale roughly linearly with n, as expected from the circuit size.

Note: The `README.md` in the `smpc` directory contains instructions to build and run the code.