Implementing MPC using SFDL

Secure Scalar Product Computation using Fairplay

Student: Mohamed Trigui
Course: Data Privacy
Assignment: HW3-3 — Multi-Party Computation with SFDL

October 16, 2025

1 Executive Summary

This project implements a secure two-party computation protocol using Fairplay and SFDL (Secure Function Definition Language) to compute the scalar product of two private Boolean vectors without revealing the inputs to either party. Alice holds a private 10-element Boolean vector A, and Bob holds a private 10-element Boolean vector B. The protocol computes $A \cdot B$ securely using garbled circuits.

2 Protocol Design

2.1 Problem Statement

Inputs:

- Alice's private vector: $A \in \{0, 1\}^{10}$
- Bob's private vector: $B \in \{0, 1\}^{10}$

Output:

- Both parties learn: $A \cdot B = \sum_{i=0}^{9}(A[i] \times B[i])$

Security Requirements:

- Alice learns only the scalar product result, not B
- Bob learns only the scalar product result, not A
- The computation is secure against semi-honest adversaries

2.2 Solution Approach

We use Yao's garbled circuit protocol implemented in Fairplay:

1. Circuit Generation: The SFDL program is compiled into a Boolean circuit (SHDL format).
2. Garbled Circuit Creation: Bob (circuit generator) creates garbled gates.
3. Oblivious Transfer: Alice obtains wire labels for her inputs without revealing them to Bob.
4. Circuit Evaluation: Alice evaluates the garbled circuit.
5. Output Revelation: Both parties learn the final result.

2.3 Mathematical Foundation

The scalar product (dot product) of two Boolean vectors is computed as:

$$A \cdot B = \sum_{i=0}^{9} \big(A[i] \wedge B[i]\big) = (A[0] \times B[0]) + \cdots + (A[9] \times B[9]).$$

For Boolean values, multiplication is equivalent to the AND operation. The result ranges from 0 to 10 (requiring 4 bits to represent).

3 Implementation Details

3.1 Task 1: Input Generation (3 Points)

Alice's private Boolean vector A:

$$A = [1, 1, 1, 1, 0, 1, 1, 1, 1, 1].$$

Bob's private Boolean vector B:

$$B = [0, 1, 0, 0, 1, 1, 0, 1, 1, 1].$$

Expected scalar product:

$$\begin{aligned} A \cdot B &= (1 \cdot 0) + (1 \cdot 1) + (1 \cdot 0) + (1 \cdot 0) + (0 \cdot 1) \\ &\quad + (1 \cdot 1) + (1 \cdot 0) + (1 \cdot 1) + (1 \cdot 1) + (1 \cdot 1) \\ &= 0 + 1 + 0 + 0 + 0 + 1 + 0 + 1 + 1 + 1 = 5. \end{aligned}$$

These vectors are stored in `hw3-3-vectors.txt` and `hw3-3-alice-input.txt` / `hw3-3-bob-input.txt`.

3.2 Task 2: SFDL Program Design (12 Points)

The SFDL program `hw3-3-ScalarProduct.txt` implements the secure scalar product computation. Key components:

1. *Type Definitions:* `BoolVector` (array of 10 Booleans), `Result` (4-bit integer).
2. *Input/Output:* Alice provides 10 Boolean inputs; Bob provides 10 Boolean inputs; both receive the same 4-bit result.
3. *Logic:* Element-wise AND followed by a population count (sum).

SFDL code structure:

Listing 1: SFDL outline for secure scalar product

```
program ScalarProduct {
    const VectorSize = 10;
    type BoolVector = Bool[VectorSize];
    type Result = Int<4>;

    function Output output(Input input) {
        var Result sum = 0;
```

```
        for (i = 0 to VectorSize-1) {
            product = input.alice[i] & input.bob[i];
            if (product) sum = sum + 1;
        }
        output.alice = sum;
        output.bob = sum;
    }
}
```

3.3 Task 3: Compilation (5 Points)

Compilation command:

```
.\run_bob.bat -c progs\hw3-3-ScalarProduct.txt
```

Sample output:

```
Program compiled.
Performing multi-to-single-bit transformation.
Transformation finished.
Unique vars transformations.
Unique vars transformations finished.
Program Optimization: Phase I.
Program Optimization: Phase II.
Optimization finished.
Writing to circuit file.
Completed.
Writing to format file.
Completed.
```

Generated files:

- `hw3-3-ScalarProduct.txt.Opt.circuit` (4,444 bytes): 84 gates in SHDL format.
- `hw3-3-ScalarProduct.txt.Opt.fmt` (932 bytes): input/output wire mappings.

Circuit analysis:

- Total gates: 84
- Input wires: 20 (10 for Alice, 10 for Bob)
- Output wires: 8 (4 for Alice's result, 4 for Bob's result)

3.4 Task 4: Protocol Execution (10 Points)

Bob (server) execution

```
.\run_bob.bat -r progs\hw3-3-ScalarProduct.txt randomseed123 4
```

Alice (client) execution

```
.\run_alice.bat -r progs\hw3-3-ScalarProduct.txt randomseed456 localhost
```

Input process (examples):
Bob's inputs (vector B):

```
input.bob[9]=1, [8]=1, [7]=1, [6]=0, [5]=1,
[4]=1, [3]=0, [2]=0, [1]=1, [0]=0
```

Alice's inputs (vector A):

```
input.alice[9]=1, [8]=1, [7]=1, [6]=1, [5]=1,
[4]=0, [3]=1, [2]=1, [1]=1, [0]=1
```

Actual execution output (October 16, 2025):
Bob:

```
Running Bob...
output.bob 5
```

Alice:

```
Running Alice...
output.alice 5
```

Verification:

$$A \cdot B = 5 \quad (\checkmark)$$

4 SHDL Circuit Analysis

4.1 Circuit Structure

The compiled circuit contains:

- 20 input gates
- 64 computation gates
- 2 (sets of) output wires producing the 4-bit results for both parties

4.2 Gate-Level Operations

Sample gates and roles:

- Element-wise AND: 10 AND gates compute $A[i] \wedge B[i]$.
- Addition: ripple-carry adder sums the 10 products into a 4-bit result.
- Output formatting: maps sum bits to both parties' outputs.

4.3 Security Properties

- Input privacy: wire labels are random; garbled values leak no input information.
- Circuit privacy: only outputs are revealed; intermediate values remain hidden.
- Correctness: the circuit implements the scalar product for all inputs.

5 Results and Analysis

5.1 Computation Results

Party	Input Vector	Output (Scalar Product)
Alice	[1,1,1,1,0,1,1,1,1,1]	5
Bob	[0,1,0,0,1,1,0,1,1,1]	5

5.2 Performance Metrics

- Circuit size: 84 gates
- Communication complexity: $O(n)$ in number of gates
- Computation time: milliseconds (network-dependent)
- Security level: semi-honest secure (computational)

5.3 Comparison to Plaintext Computation

Metric	Plaintext	Garbled Circuit
Privacy	None	Full input privacy
Computation	10 AND + 9 ADD	84 gate evaluations
Communication	Direct share	$\sim$1KB (circuit + OT)
Rounds	1 round	Constant rounds

6 Conclusions

6.1 Key Achievements

1. Implemented secure two-party computation for scalar product.
2. Demonstrated protection of private inputs during joint computation.
3. Verified correctness with known test vectors.
4. Analyzed circuit structure and security properties.

6.2 Lessons Learned

- SFDL offers a high-level way to express secure computations.
- Garbled circuits enable constant-round secure computation.
- Circuit size grows with computation complexity (84 gates here).
- Fairplay automates core cryptographic operations for MPC.

6.3 Potential Applications

Privacy-preserving ML (dot products), private set intersection, secure auctions, biometric matching.

6.4 Limitations and Future Work

Current limitations: semi-honest security, two-party only, limited scalability for large vectors. *Future enhancements:* malicious security (cut-and-choose, authenticated garbling), n-party MPC, adder optimizations, benchmarking with larger vectors.

7 File Inventory

All files are prefixed with `hw3-3-`:

1. `hw3-3-report.md` — report
2. `hw3-3-ScalarProduct.txt` — SFDL source
3. `hw3-3-ScalarProduct.txt.Opt.circuit` — SHDL circuit
4. `hw3-3-ScalarProduct.txt.Opt.fmt` — format file
5. `hw3-3-vectors.txt` — input vectors
6. `hw3-3-alice-input.txt` — Alice's input
7. `hw3-3-bob-input.txt` — Bob's input
8. `hw3-3-run-test.md` — execution instructions

8 References

1. Malkhi, D., Nisan, N., Pinkas, B., & Sella, Y. (2004). *Fairplay — A Secure Two-Party Computation System.* USENIX Security.
2. Yao, A. C. (1986). *How to Generate and Exchange Secrets.* IEEE FOCS.
3. Goldreich, O., Micali, S., & Wigderson, A. (1987). *How to Play ANY Mental Game.* ACM STOC.
4. Lindell, Y., & Pinkas, B. (2009). *A Proof of Security of Yao's Protocol for Two-Party Computation.* Journal of Cryptology.

Appendix A: Complete SFDL Source Code

Listing 2: Complete SFDL source

```
/*
 * Secure Scalar Product Computation
 *
 * Computes the scalar product (dot product) of two Boolean vectors.
 * Alice has a private Boolean vector A with 10 entries.
 * Bob has a private Boolean vector B with 10 entries.
 */

program ScalarProduct {
```

```
    // Constants
    const VectorSize = 10;

    // Type definitions
    type Bool = Boolean;
    type BoolVector = Bool[VectorSize];
    type Result = Int<4>;  // 4 bits can represent 0-15

    // Input types
    type AliceInput = BoolVector;
    type BobInput = BoolVector;

    // Output types
    type AliceOutput = Result;
    type BobOutput = Result;

    // Combined I/O structures
    type Input = struct {AliceInput alice, BobInput bob};
    type Output = struct {AliceOutput alice, BobOutput bob};

    // Main computation function
    function Output output(Input input) {
        var Result sum;
        var Int<4> i;
        var Bool product;

        // Initialize sum
        sum = 0;

        // Compute scalar product: sum of element-wise products
        for (i = 0 to VectorSize-1) {
            // Multiply corresponding elements (AND for Booleans)
            product = input.alice[i] & input.bob[i];

            // Add to running sum
            if (product) {
                sum = sum + 1;
            }
        }

        // Both parties receive the same result
        output.alice = sum;
        output.bob = sum;
    }
}
```