Secrets and Spies

Secure Multiparty Computation for *Two Spies*

Kosi Nwabueze, Felix Prasanna, and Frederick Tang

Massachusetts Institute of Technology
{kosinw,fpx,ftang}@mit.edu

Abstract. Secure multiparty computation (MPC) enables multiple parties to collaboratively compute functions over private inputs without revealing those inputs to other parties. In this paper, we explore the application of MPC to a generalized version of the turn-based strategy game *Two Spies*. Our protocol allows N distributed players to participate in a perfectly secure game of *Two Spies* without a trusted moderator. Each undercover player's location remains confidential to other players while ensuring gameplay is both fair and verifiable. Our protocol works under two adversarial models: *semi-honest* and *malicious*. Under the *semi-honest* adversarial model, $t < N/2$ adversaries follow the protocol while trying to learn as much information as possible through collusion. Under the *malicious* adversarial model, $t < N/3$ adversaries may actively deviate from the protocol and collude to learn additional information.

Keywords: Secure multiparty computation · Secret sharing · Commitments · Games.

1 Introduction

Two Spies [Sof25] is a turn-based strategy game developed by Steamclock Software that pits two players against each other. Assuming the roles of spies from rival countries, each player takes alternating turns navigating across a map of Cold War Europe in an attempt to gain intelligence on the other spy's location. The objective is to be the first player to strike and eliminate the other spy before they strike and eliminate you. The core gameplay of *Two Spies* revolves around intelligence gathering, economy management, and movement across a map represented as a graph (see Fig. 1).

To simplify the discussion, we assume a game between two players, Alice, whose turn we are considering, and Bob, her opponent. In the remainder of this section, we describe the game setup, a typical turn for Alice looks like, and the core mechanics of the game, including cover and intel.

1.1 Setup

Two Spies requires a setup procedure that relies on randomness and a trusted moderator. First, the moderator procedurally generates a graph $G = (V, E)$.

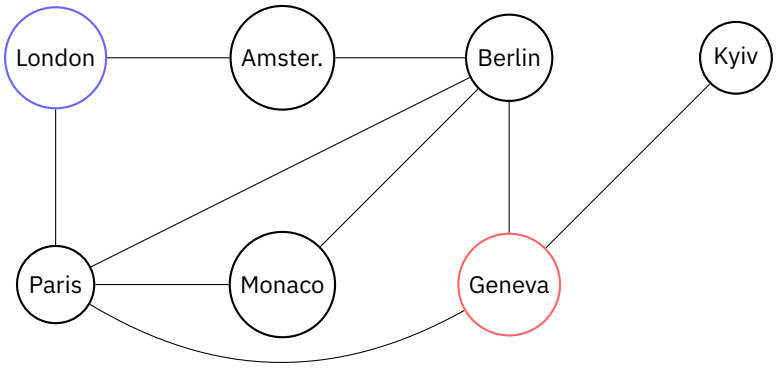

Fig. 1. A small, hypothetical *Two Spies* map. The spy Alice is currently in Geneva, while the spy Bob is in London.

Each graph G, also has a procedurally generated set of starting locations $S \subset V$, where each vertex in S must not be an articulation point. Let N be the number of players. The moderator randomly selects N locations from S and assigns one to each player. Each player then starts the game at her assigned location and automatically controls that city. The moderator then randomly chooses a permutation P of the set of players, which determines the order in which turns are taken. Every player starts the game with her cover blown.

1.2 Actions

By default, Alice performs two actions on her turn. She may gain bonus actions by preparing one or more times on her previous turn. She may also gain bonus actions for her next turn by moving onto a city with a bonus action. She may perform any of the following actions on her turn:

1. **Move**. Alice changes her location to an adjacent city on the map, corresponding to an adjacent vertex in the graph. Moving causes her to go under cover. If she moves to a controlled or occupied city, her cover is blown and her location is revealed.
2. **Control**. Alice takes control of the city she currently occupies. If Bob moves into a city controlled by Alice, his cover is blown and his location revealed. Alice may take control of cities already under another player's control.
3. **Strike**. Alice strikes a city. If Bob is occupying the struck city, he is eliminated. The last player remaining wins the game.
4. **Wait**. Alice remains in the same city. This action is functionally equivalent to moving, but without changing her location.
5. **Locate**. Alice spends intel to reveal Bob's location. If Bob is under deep cover, the action has no effect.

6. **Go Deep**. Alice goes into deep cover, which costs intel. While in deep cover, her cover cannot be blown during the rest of her turn.
7. **Prepare**. Alice gains an additional action for her next turn. This action also costs intel.
8. **Strike Reports**. For a one-time intel cost, Alice can unlock strike reports. Once unlocked, she learns Bob's location whenever he performs a strike.
9. **Rapid Recon**. For a one-time intel cost, Alice can unlock rapid recon. Bob will blow his cover and reveal his location if Alice moves into the city he occupies.

1.3 Cover

Cover is a crucial mechanic that introduces uncertainty about a spy's location. After moving or waiting in a city, Alice goes under cover, making her invisible to other players. However, her cover can be blown in the following situations:

1. Alice controls a city.
2. Alice ends her turn in a city occupied by Bob.
3. Alice moves into a city controlled by Bob.
4. Bob uses the *locate* action to find Alice.
5. Alice strikes a city after Bob has purchased strike reports.
6. Bob, after purchasing *rapid recon*, moves into a city occupied by Alice.
7. Alice moves to a city with bonus intel or a bonus action.

Normally, Alice's cover can be compromised through these actions. However, if she is under *deep cover*, her cover remains intact throughout her turn, even if one of the conditions above is met. Deep cover can only be activated by performing the *go deep* action.

1.4 Intel

Intel, short for intelligence, acts as the in-game currency for *Two Spies*. Intel allows players to purchase actions during their turns. By default, Alice earns +2 intel per turn. She can earn bonus intel by controlling normal cities, controlling bonus cities, or moving to cities with bonus intel.

Cities with bonus intel initially reward +10 bonus intel when a player moves onto them. The amount of bonus intel available is visible to all players and increases by a random amount each turn if left unclaimed. The formula governing the intel Alice earns each turn is:

$$\text{intel/turn} = 2 + 4 \times \#\text{bonus cities} + \#\text{normal cities}.$$

1.5 Shutdowns

To prevent games from lasting indefinitely, the map shrinks over time, reducing the search space and forcing players into closer encounters. Every few turns, cities farther from the center of the map–specifically non-articulation points–*shut down*. Once a city is shut down, no player may move into it for the rest of the game.

If a player is occupying a city that shuts down, they are forced to move to an active city. Importantly, players continue to earn intel for shut-down cities they control, even though they can no longer move to or occupy them.

2 Background

We explore the use of secure multiparty computation to implement a protocol for generalized *Two Spies* without the need for a trusted moderator. Prior work [SRA79; WUG16] have demonstrated the application of cryptography to playing games without a moderator. In this section, we review the necessary preliminaries to properly describe our protocol, including secret sharing, secure multiparty computation, and commitments.

2.1 Secret sharing

Secret sharing divides a secret into N private values, called shares, which are then distributed among N parties. Many variants of secret sharing have been proposed; the two most popular in multiparty computation being *additive secret sharing* and *polynomial secret sharing*. Both secret sharing schemes offer perfect or information-theoretic security, meaning no adversary, even with computationally unbounded power, can learn any information about the secret given an unauthorized subset of shares. Despite the simplicity of additive secret sharing, our protocols rely on polynomial secret sharing due to important properties discussed in the next section.

Shamir [Sha79] originally proposed a polynomial secret sharing scheme that enables (t, n)–threshold secret sharing. In a threshold secret sharing scheme, any subset of t or more shares from $\{1, \ldots, n\}$ can reconstruct the secret. We now formally define the security and correctness properties of the scheme:

Definition 1 (Threshold secret sharing). *A (t, n)–threshold secret sharing scheme over message space $\mathcal{M}$ consists of a pair of efficient algorithms* (Share, Reconstruct).

Share *is a randomized algorithm that takes as input a message $m \in \mathcal{M}$ and outputs a sequence of n shares $(\llbracket m_1 \rrbracket, \ldots, \llbracket m_n \rrbracket)$.*

Reconstruct *is a deterministic algorithm that given a set of 2-tuples $\{(i, \llbracket m_i \rrbracket)\}_{i \in I}$ for $|I| = t$, outputs a message $m \in \mathcal{M}$.*

1. ***Correctness.*** *For every $m \in \mathcal{M}$ and for every $I \subseteq \{1, \ldots, n\}$ of size t,*

$$\Pr\left[\mathsf{Reconstruct}(\{(i, \llbracket m_i \rrbracket)\}_{i \in I}) = m \;:\; (\llbracket m_1 \rrbracket, \ldots, \llbracket m_n \rrbracket) \leftarrow \mathsf{Share}(m)\right] = 1.$$

Algorithm 1: Share

Input: Secret $m \in \mathbb{F}$, threshold t, number of parties n
Output: Shares $(s_1, \ldots, s_n)$
Choose random $a_1, \ldots, a_{t-1} \in \mathbb{F}$;
Let $f(X) = s + a_1 X + a_2 X^2 + \cdots + a_{t-1} X^{t-1}$;
for $i \leftarrow 1$ **to** n **do**
  $s_i \leftarrow f(i)$;
return $(s_1, \ldots, s_n)$

Algorithm 2: Reconstruct

Input: t shares $(i_1, s_{i_1}), \ldots, (i_t, s_{i_t})$
Output: Secret s
$s \leftarrow 0$;
for $k \leftarrow 1$ **to** t **do**
  $\lambda_k \leftarrow \prod_{\substack{j=1 \\ j \neq k}}^{t} \frac{i_j}{i_j - i_k}$;
  $s \leftarrow s + \lambda_k \cdot s_{i_k}$;
return s

Fig. 2. Polynomial secret sharing as given by [Sha79]. Both algorithms are parameterized over message space $\mathbb{F}$, number of parties n, and threshold t.

2. ***Security.*** *For every $m, m' \in \mathcal{M}$ and for every $I \subseteq \{1, \ldots, n\}$ such that $|I| < t$,*

$$(\llbracket m_i \rrbracket)_{i \in I} \equiv (\llbracket m'_i \rrbracket)_{i \in I}.$$

where $(\llbracket m_1 \rrbracket, \ldots, \llbracket m_n \rrbracket) \leftarrow \mathsf{Share}(m)$ *and* $(\llbracket m'_1 \rrbracket, \ldots, \llbracket m'_n \rrbracket) \leftarrow \mathsf{Share}(m')$.

The algorithms for polynomial secret sharing are shown in Fig. 2. Here, the secret $m \in \mathbb{F}$ is hidden inside a randomly chosen polynomial $f(X) \in \mathbb{F}[X]$ such that $f(0) = m$. Using Lagrange interpolation, a polynomial of degree $t - 1$ can be reconstructed from t unique points, allowing recovery of the original message.

2.2 Secure multiparty computation

Secure multiparty computation (MPC) focuses on designing methods that allow a set of n parties, each holding private input, to jointly evaluate a public function f over their inputs, without revealing anything beyond the output of the function. We now give a formal definition of the security model.

Definition 2 (Secure multiparty computation). *An n-party protocol securely computes a function f in the presence of at most t corruptions, if for every PPT adversary $\mathcal{A}$ controlling at most t parties, there exists a PPT simulator $\mathcal{S}$ controlling the same subset of parties in the ideal world, such that for any set of inputs $(x_1, \ldots, x_n)$,*

$$\mathsf{Real}_{\mathcal{A}}(x_1, \ldots, x_n) \equiv \mathsf{Ideal}_{\mathcal{S}}(x_1, \ldots, x_n).$$

where $\mathsf{Real}_{\mathcal{A}}(x_1, \ldots, x_n)$ is the output of all the parties after running the protocol; $\mathsf{Ideal}_{\mathcal{S}}(x_1, \ldots, x_n)$ is the output of all the parties after handing their inputs to the trusted party, who computes f on their input and returns the output y. The honest parties output y and the adversarial parties (controlled by the simulator) output whatever they want.

Algorithm 3: Π_{BGW}

Setting: Each party P_i has input $x_i \in \mathbb{F}$.

Input Phase: Each party P_i secret shares input x_i using polynomial secret sharing. The parties obtain shares $[\![x_i]\!]$.

Output Phase: Let $[\![z_i]\!]$ be the output of the computation. Each party P_i broadcasts its share $[\![z_i]\!]$ to all other parties. After each party P_i receives $t+1$ shares, it reconstructs the output z.

Addition Gates: For each addition gate with inputs $[\![x]\!]$ and $[\![y]\!]$, each party locally computes: $[\![x+y]\!] \leftarrow [\![x]\!] + [\![y]\!]$.

Multiplication Gates: For each multiplication gate with inputs $[\![x]\!]$ and $[\![y]\!]$, the parties proceed as follows:

1. Each party locally computes $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$.
2. Each party P_i secret shares 0. Each party adds up shares of 0 with $[\![z]\!]_{2t}$ to obtain $[\![z']\!]_{2t}$.
3. Each party truncates its share $[\![z']\!]_{2t}$ to $[\![z']\!]_t$. This requires running Π_{BGW} to convert a $2t$ polynomial to a t polynomial.

Fig. 3. Secure multiparty computation protocol Π_{BGW} as given by [BGW88].

Ben-Or, Goldwasser, and Widgerson [BGW88] introduced an MPC protocol based on polynomial secret sharing. Beaver, Micali and Rogaway [BMR90] developed a protocl based on additive secret sharing. Other notable MPC constructions are based on a technique known as *garbled circuits*. Yao [Yao86] introduces garbled circuits in the context of a two-party computation protocol (2PC).

We consider a modified [Esc22] version of the MPC protocol introduced by Ben-Or, Goldwasser, and Widgerson [BGW88]. The core idea of this protocol is to perform a secure function evaluation of $f(x_1, \ldots, x_k)$ on private inputs $(x_1, \ldots, x_k) \in \mathbb{F}^n$ by distributing polynomial secret shares $[\![x_j]\!]^i$ to party i for input j. Prior to evaluation, the function f would be compiled to a circuit of multiplication and addition gates. To compute an addition gate $[\![x+y]\!]$, each party locally performs the calculation $[\![x]\!] + [\![y]\!]$. To compute a multiplication gate $[\![x \cdot y]\!]$, each party would first locally compute $[\![x \cdot y]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$. Each party would engage in a $\Theta(n^2)$ round of communications to reduce their share $[\![x \cdot y]\!]\, 2t$ to $[\![x \cdot y]\!]\, t$. A formal description of this protocol can be found in Fig. 3.

The protocol Π_{BGW} provides perfect security in the semi-honest adversarial model with an honest majority ($t < n/2$). With the incorporation of verified secret sharing (VSS), it also achieves perfect security under the malicious model with a two-thirds honest majority ($t < n/3$), as proven by Asharov and Lindell [AL17].

Algorithm 4: Commit

Input: Secret $m \in \mathbb{G}$, public generators g, h of a group $\mathbb{G}$ of prime order q
Output: Commitment $C \in \mathbb{G}$ and nonce $r \in \mathbb{Z}_q$
$r \leftarrow\!\!\$\ \mathbb{Z}_q$;
$C \leftarrow g^m h^r$;
return C

Fig. 4. Discrete logarithm-based commitment scheme as given by [Ped92].

2.3 Commitments

Commitment schemes are cryptographic primitives that allow a party to commit to a value while keeping it hidden, with the ability to reveal it later. A commitment must satisfy two key properties: **binding** (the committer cannot change the committed value) and **hiding** (the commitment reveals no information about the value until it is opened).

Definition 3 (Commitment scheme). *A commitment scheme over message space $\mathcal{M}$ consists of a pair of efficient algorithms (Gen, Commit).*

Gen is an efficient, randomized algorithm that takes as input security parameter 1^λ and outputs public parameters pp.

Commit is an efficient, randomized algorithm that takes as input public parameters pp, a message $m \in \mathcal{M}$ and randomness $r \leftarrow\!\!\$\ \{0,1\}^\lambda$ and outputs a commitment.

1. ***Hiding**. For every $m_0, m_1 \in \mathcal{M}$, $pp \leftarrow \mathsf{Gen}(1^\lambda)$ and $r_0, r_1 \leftarrow\!\!\$\ \{0,1\}^\lambda$,*

$$(pp, \mathsf{Commit}(pp, m_0; r_0)) \approx (pp, \mathsf{Commit}(pp, m_1; r_1)).$$

2. ***Binding**. For every efficient adversary $\mathcal{A}$, $(m_0, r_0, m_1, r_1) \leftarrow \mathcal{A}(pp)$.*

$$\Pr[m_0 \neq m_1 \text{ and } \mathsf{Commit}(pp, m_0; r_0) = \mathsf{Commit}(pp, m_1; r_1)] = \mathsf{negl}(\lambda).$$

Pedersen [Ped92] proposed a commitment scheme based on the hardness of the discrete log problem in a prime-order group G. In this scheme, the commitment is **computationally binding** and **statistically hiding**. Additionally, the scheme is additively and multiplicatively homomorphic, though these properties are not utilized in our work. The generation algorithm Gen is trivial in Pedersen commitments, requiring only public generators g, h of the group G (such that $\log_g(h)$ is unknown). The commitment algorithm Commit is given in Fig. 4

3 Protocols

In this section, we detail the protocols required to play the *Two Spies* game without a moderator. From this point forward, we refer exclusively to a generalization of *Two Spies* with an arbitrary number of players, rather than two.

For clarity of exposition, we introduce the parties Alice, Bob, Carol, Dave, and Eve. Figures will illustrate two parties for simplicity, but the concepts generalize naturally to many players.

3.1 Threat Model

Following the assumptions in Π_{BGW}, we assume that each party communicates synchronously using *private and authenticated point-to-point channels*. We do not consider Byzantine faults arising from the reliable asynchronous model, which allows adversaries to delay messages arbitrarily.

Each party uses remote procedure calls (RPCs) for communication, with retries upon timeout and parallel issuing of RPCs to improve performance.

The threat model assumes that adversaries attempt to gain unfair advantages during the game. Certain information, such as players' locations or the number of players sharing a location, must remain private. A key security goal is to ensure that an adversary cannot learn any information about players' locations unless explicitly revealed according to the game rules. Additionally, adversaries are assumed to want to continue playing the game, so we assume timely communication without arbitrary message delays and well-formed messages.

3.2 Setup

One party, say Alice, initially acts as the host of the game. Players join the game by sending an RPC to Alice, at which point she adds them to a list of active players. We assume that Alice remains honest until all parties have joined. Each player supplies Alice with their network address addr, public key pk, and verification key vk upon joining.

Alice records this information for all players and then broadcasts it to everyone once the game setup is complete. After this, Alice may be corrupted by an adversary without affecting the security guarantees.

3.3 Randomness

Two Spies relies heavily on centralized randomness–for example, in procedurally generating the graph $G = (V, E)$, determining starting locations $S \subset V$, and selecting turn orders and city shutdowns. Without a centralized moderator, all parties must independently derive the same sequence of random decisions. This is achieved through a *shared seed* that is securely negotiated. Each party contributes a private nonce $r_i \leftarrow\!\!\$\ \mathbb{F}$. The public nonce r is computed as

$$r = \sum_{i=1}^{n} r_i.$$

To prevent a malicious party, say Eve, from choosing her nonce after seeing others' contributions, we use a commitment-based protocol Π_{Seed}:

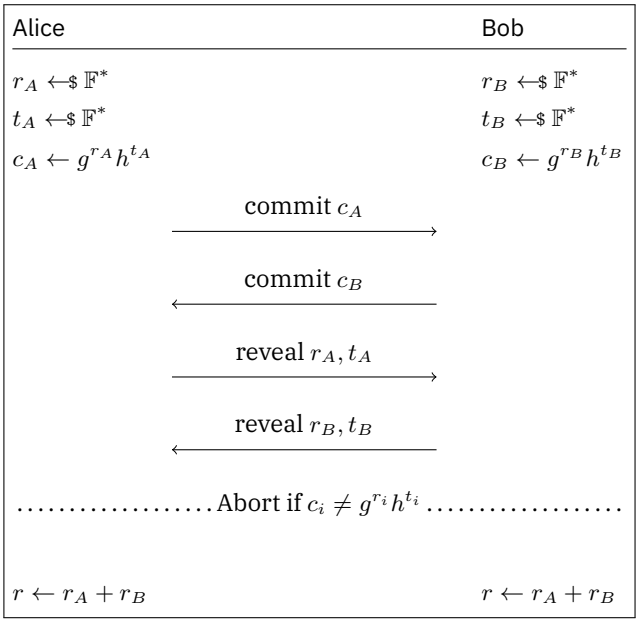

Fig. 5. Seed generation protocol Π_{Seed} using Pedersen commitments. g and h are generators of the multiplicative group $\mathbb{F}^*$ such that $\log_g(h)$ is not known.

1. Each party commits to their private nonce r_i using a Pedersen commitment $C(r_i; t_i) = g^{r_i} h^{t_i}$.
2. After all commitments are broadcast, each party reveals (r_i, t_i).
3. Each party checks that the revealed values match the commitments.
4. If the verifications succeed, each party computes the public nonce r.

The random oracle $H(r)$ is then used as the seed for a common cryptographically secure pseudorandom number generator (CSPRNG). Fig. 5 illustrates protocol Π_{Seed}.

3.4 Privacy

The primary challenge in playing *Two Spies* without a moderator is preserving the privacy of each player's location while ensuring that moves are valid. We address this using secret sharing and secure multiparty computation.

Each player's location is encoded as a one-hot vector $x_u \in 0, 1^{|V|}$ where:

$$(x_u)_i = \begin{cases} 1 & \text{if } i = u \\ 0 & \text{otherwise} \end{cases}$$

Each party holds private shares of each player's location. For example, Eve stores:

$$\left(\llbracket x_{\mathsf{Alice}} \rrbracket^{\mathsf{Eve}}, \llbracket x_{\mathsf{Bob}} \rrbracket^{\mathsf{Eve}}, \llbracket x_{\mathsf{Carol}} \rrbracket^{\mathsf{Eve}}, \llbracket x_{\mathsf{Dave}} \rrbracket^{\mathsf{Eve}}, \llbracket x_{\mathsf{Eve}} \rrbracket^{\mathsf{Eve}}\right) : \mathsf{list}\ \mathbb{F}^{|V|}.$$

Suppose Eve wishes to move from Monaco to Berlin. She must prove to all players that the move is valid, that is, $(\text{Monaco}, \text{Berlin}) \in E$, without revealing her current or new location. We model the adjacency matrix of G as $A(G) \in \{0,1\}^{|V|\times|V|}$ where:

$$A(G)_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

To verify the move, players jointly compute:

$$f(x_u, x_v) = \langle x_v, A(G)x_u \rangle.$$

where x_u and x_v represent Eve's starting and ending locations. If $f(x_u, x_v) = 1$, the move is valid. Otherwise, the protocol aborts. Fig. 6 describes the move verification protocol Π_{Verify}.

Proof sketch. *Since each location is represented as a one-hot vector, the product $A(G)x_u$ yields a vector whose entries indicate the set of cities adjacent to Eve's current location u. Taking the inner product $\langle x_v, A(G)x_u \rangle$ isolates the entry corresponding to the proposed destination v. Thus, $f(x_u, x_v) = 1$ if and only if $(u, v) \in E$, meaning the move is valid. This computation preserves privacy while verifying move correctness.* □

At times, a player must reveal their location to other players. For example, if Alice ends her turn on Carol's city, then Alice must reveal her location to Carol. Another example, if Dave uses the locate action to reveal Bob's location, then Bob must reveal his location to Dave. Depending on the game mechanics, there are three types of location sharing:

1. **Private sharing**. A player reveals their location to a single party (e.g., when two spies end their turns on the same city, or when *rapid recon* reveals a location).
2. **Semi-public sharing**. A player reveals their location to a subset of parties–for example, to those who have unlocked *strike reports* when a strike occurs.
3. **Public sharing**. A player reveals their location to all parties–for example, when controlling a city, being located, or moving onto a bonus intel/action city.

Semi-public sharing and public sharing. Suppose Carol must reveal her location to a subset $I \subseteq P$ of parties (which could include all players). Each party $p \in I$ runs the following protocol $\Pi_{\mathsf{PublicShare}}$ with Carol. In public or semi-public sharing, all players know that Carol's location is being revealed, though they may not all learn the location themselves (in semi-public sharing). Fig. 7 describes the public sharing protocol $\Pi_{\mathsf{PublicShare}}$.

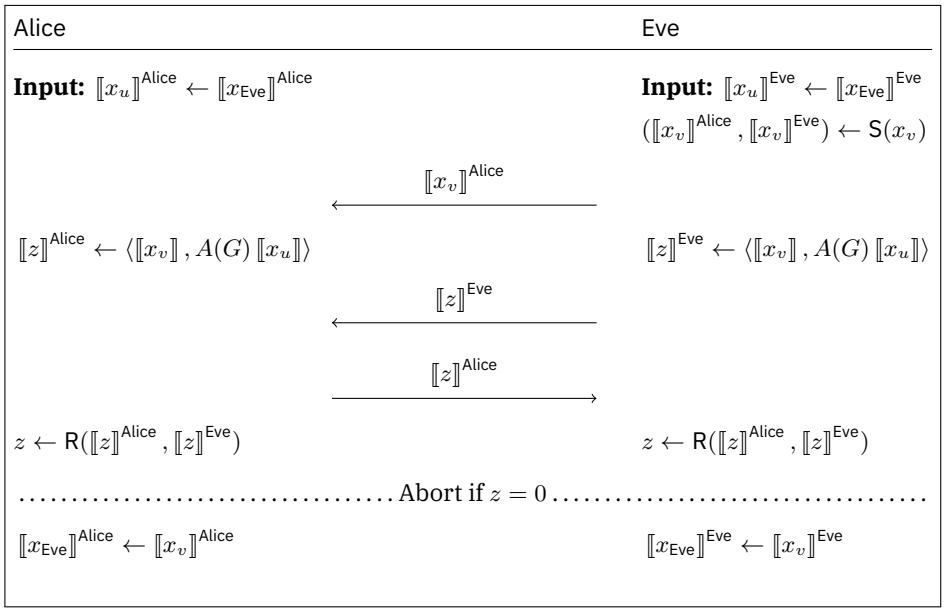

Fig. 6. Move verification protocol Π_{Verify}. x_u is the starting position for Eve and x_v is the ending position.

Private sharing. In private sharing, we want only the designated recipient, Carol, to learn whether another player, Bob, occupies the same city without others learning anything. To detect a location match, we compute:

$$b = \langle x_{\text{Bob}}, x_{\text{Carol}} \rangle.$$

where $b = 1$ indicates that Bob and Carol share the same city. Since neither player can trusted to compute b correctly alone, all parties jointly compute the inner product using secure multiparty computation. All other parties own shares of their locations, so all parties will perform a multiparty computation to compute the function $g : \{0,1\}^{|V|} \times \{0,1\}^{|V|} \to \{0,1\}$:

$$g(x_a, x_b) = \langle x_a, x_b \rangle.$$

Since we only want Carol to learn the result of g, each party sends Carol shares of $g(x_{\text{Bob}}, x_{\text{Carol}})$. If $g(x_{\text{Bob}}, x_{\text{Carol}}) = 1$, then Carol knows Bob shares her location without Bob having to send his location using $\Pi_{\text{PublicShare}}$. The computation of g is implemented by protocol $\Pi_{\text{PrivateShare}}$ in Fig. 8.

One-hotness verification Note that the above protocols breaks if a player's position vector is not one-hot. A malicious party could share a vector with multiple 1's, and this could lead to gain of illegal information: in the above example, if Carol increased the number of 1's in her vector, the g function

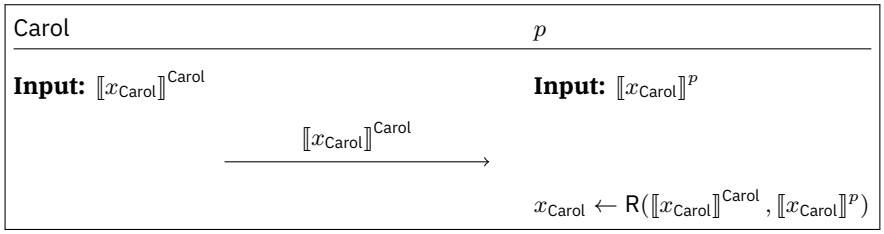

Fig. 7. Public location sharing protocol $\Pi_{\text{PublicShare}}$.

has an increased chance of returning a 1, and Carol can then narrow down the possibilities of Bob's location.
Our Security model only considers honest-but-curious parties, but we can extend our protocol slightly to prevent malicious parties from cheating while sharing location.
we propose the following protocol for checking that a shared location $x \in \mathbb{F}^{|V|}$ is indeed one hot:
We first modify how we generate the shares when sharing the location x. We use a n out of n Secret sharing scheme, which generates P random shares (with $P-1$ degrees of freedom), $x_1, x_2, \ldots, x_P \in \mathbb{F}^{|V|}$ such that $x_1 + x_2 + \cdots + x_P = x$ Using the Pseudorandom number generator and agreed upon seed in Section 3.3, all party will jointly generate a vector $r = \{r_1, r_2, \ldots, r_{|V|}\} \in \mathbb{F}^{|V|}$.
Also define another vector $R = \{r_1^2, r_2^2, \ldots, r_{|V|}^2\} \in \mathbb{F}^{|V|}$.
Each party P_i with share x_i independently computes $t_i = \langle x_i, r \rangle$ and $T_i = \langle x_i, R \rangle$. Using MPC, all parties will jointly compute $\sum_i^P t_i$ and $\sum_i^P T_i$ and then check if

$$(\sum_i^P t_i)^2 = \sum_i^P T_i$$

Clearly if x were a one hot vector, and had a 1 at entry k, $(\sum_i^P t_i)^2 = r_k^2$, and $\sum_i^P T_i = R_k = r_k^2$, so we have correctness.
It can be shown that if x were not one hot, the probability that the check would pass is bounded by $\frac{2}{||\mathbb{F}||}$, which is negligible for large $||\mathbb{F}||$.

4 Implementation

To demonstrate the practicality of our protocols, we built a networked prototype of the *Two Spies* game using the Go programming language [Aut25]. We chose Go over the more traditional choice of Python for cryptographic work for three main reasons:

1. **Static typing**. Go is statically typed, making it easier to catch errors at compile time rather than at runtime, improving code reliability.

2. **Standard library**. Go offers a robust standard library. Common cryptographic primitives SHA256 and RSA are available via `crypto/sha256` and `crypto/rsa`, respectively. Arithmetic with large integers is supported via `math/big`, and networking is facilitated by `net/rpc`.
3. **Concurrency**. Go's goroutines provide lightweight, easy-to-use concurrency, enabling better performance in networked environments. Unlike Python, which uses the Global Interpreter Lock (GIL) and primarily supports async I/O, Go can execute goroutines across multiple OS threads.

4.1 Security Considerations

While the prototype successfully implements the basic functionality, it is closer to a proof-of-concept than a production-ready build, primarily due to security limitations.

First, Π_{BGW} assumes *private and authenticated point-to-point channels*. In a production setting, we would enforce this using TLS (`crypto/tls`) to prevent eavesdropping or tampering. However, for the prototype, we opted for plaintext RPCs to reduce complexity, relying on a lightweight *sign-then-encrypt* scheme to protect sensitive information. When Alice wishes to send a message m to Bob, she sends:

$$c \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathsf{Bob}}, \mathsf{addr}_{\mathsf{Bob}} \| m \| \mathsf{Sign}(\mathsf{sk}_{\mathsf{Alice}}, H(\mathsf{addr}_{\mathsf{Bob}} \| m))).$$

where Enc is public-key encryption (RSA-OAEP), Sign is a digital signature scheme (RSA-PSS), and H is a hash function (SHA256). Upon receiving c, Bob decrypts it, verifies the address, and authenticates Alice's signature using her verification key.

Additionally, the prototype does not currently use constant-time cryptography. For instance, the `math/big` package is known to have timing side channels, which could be exploited by a powerful adversary.

4.2 Cryptographic Instantiations

The cryptographic primitives used in our implementation are as follows:

1. **Public-Key Encryption (**$\mathsf{Enc}, \mathsf{Dec}$**)**: RSA-OAEP, using `crypto/rsa`.
2. **Signatures (**$\mathsf{Sign}, \mathsf{Vf}$**)**: RSA-PSS, using `crypto/rsa`.
3. **Hash Function (**H**)**: SHA256, using `crypto/sha256`.
4. **CSPRNG**: ChaCha8, from `internal/chacha8rand`.

For polynomial secret sharing and Π_{BGW}, we implemented field operations over $\mathbb{Z}_p$ [1] using arbitrary-precision integers.

[1] p is the 1536-bit prime declared in RFC 3526 [KK03]

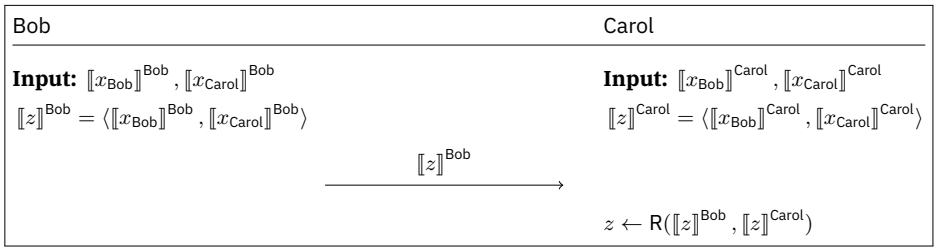

Fig. 8. Private location sharing protocol $\Pi_{\mathsf{PrivateShare}}$.

5 Conclusion

In this paper, we explored the application of secure multiparty computation (MPC) to the turn-based strategy game *Two Spies*, generalizing the game to an arbitrary number of players. Our work addresses the challenge of enabling secure gameplay without relying on a trusted moderator, ensuring that each player's information remains private while maintaining fairness and verifiability.

We detailed several protocols, including Π_{Seed} for secure random seed generation using Pedersen commitments, Π_{Verify} for validating moves using the adjacency matrix of the game map, and $\Pi_{\mathsf{PublicShare}}$ and $\Pi_{\mathsf{PrivateShare}}$ for various modes of location sharing.

To demonstrate the practicality of our approach, we developed a networked prototype of the game using the Go programming language. While the prototype successfully implements the core functionality, it is primarily a proof-of-concept and leaves room for improvement in security practices, such as enforcing TLS-secured channels and employing constant-time cryptography.

5.1 Contributions

Nwabueze drafted the initial rough version of the paper, while Prasanna and Tang refined the final version. All authors contributed to the paper's presentation. Tang developed the core theory behind Π_{Seed}, Π_{Verify}, $\Pi_{\mathsf{PublicShare}}$, and $\Pi_{\mathsf{PrivateShare}}$, with Nwabueze and Prasanna refining these ideas. All authors contributed to the development of the proof-of-concept game implementation.

5.2 Acknolwedgements

We thank our TA advisor, Lila Chen, for her guidance throughout the project. We also thank Professors Corrigan-Gibbs and Kalai for their passionate and instructive lectures during the semester. Their expertise and enthusiasm for cryptography inspired us to pursue this project.

References

[AL17] Gilad Asharov and Yehuda Lindell. “A Full Proof of the BGW Protocol for Perfectly Secure Multiparty Computation”. In: *Journal of Cryptology* 30.1 (2017), pp. 58–151. DOI: 10.1007/s00145-015-9214-4. URL: https://doi.org/10.1007/s00145-015-9214-4.

[Aut25] The Go Authors. *The Go Programming Language*. Accessed: 2025-04-24. 2025. URL: https://go.dev/.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation”. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*. Chicago, Illinois, USA, 1988, pp. 1–10. DOI: 10.1145/62212.62213. URL: https://dl.acm.org/doi/10.1145/62212.62213.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. “The Round Complexity of Secure Protocols”. In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 503–513. ISBN: 0897913612. DOI: 10.1145/100216.100287. URL: https://doi.org/10.1145/100216.100287.

[Esc22] Daniel Escudero. *An Introduction to Secret-Sharing-Based Secure Multiparty Computation*. Cryptology ePrint Archive, Paper 2022/062. 2022. URL: https://eprint.iacr.org/2022/062.

[KK03] Mika Kojo and Tero Kivinen. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. RFC 3526. May 2003. DOI: 10.17487/RFC3526. URL: https://www.rfc-editor.org/info/rfc3526.

[Ped92] Torben Pryds Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology — CRYPTO ’91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140.

[Sha79] Adi Shamir. “How to Share a Secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[Sof25] Steamclock Software. *Two Spies Homepage*. 2025. URL: https://www.playspies.com.

[SRA79] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. *Mental Poker*. Tech. rep. MIT-LCS-TM-125. Massachusetts Institute of Technology, 1979. URL: https://people.csail.mit.edu/rivest/pubs/SRA81.pdf.

[WUG16] Ashley Wang, Cristhian Ulloa, and Ronald Gil. *Distributed Cryptographic Mafia*. Accessed: 2025-04-12. 2016. URL: https://courses.csail.mit.edu/6.857/2016/files/7.pdf.

[Yao86] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets”. In: *27th Annual Symposium on Foundations of Computer Science (FOCS)*.

1986, pp. 162–167. URL: https://dl.acm.org/doi/abs/10.1109/SFCS.1986.25.