Cryptographic protocol for playing Risk in an untrusted setting

Jude Southworth

Bachelor of Science in Computer Science and Mathematics
The University of Bath
2023

1 Outline

Risk is a strategy game developed by Albert Lamorisse in 1957. It is a highly competitive game, in which players battle for control over regions of a world map by stationing units within their territories in order to launch attacks on neighbouring territories that are not in their control.

2 Existing solutions

For playing games over an internet connection, multiple solutions already exist. These can roughly be broken down into those that are centralised and those that are decentralised, although many decentralised systems rely on federated or centralised communications for peer discovery.

2.1 Centralised

In highly centralised networks, traffic is routed to a number of servers that are operated by the same organisation who maintains the game or service. This is the current standard for the majority of the internet: in fact, this is the methodology used by the official version of Risk, playable as an app.

Without patching the executables, there is no way for a user to run their own servers, or to connect to a third party's server. This has two main advantages:

- **Moderation.** The developers can enforce their own rules through some form of EULA, and this would be properly enforceable, as if a user is banned from the official servers, there is no alternative.
- **Security.** The server acts as a trusted party, and validates all communications from players. Hence, players cannot subvert a (properly implemented) service's protocol.

2.2 Peer-to-peer networks

In peer-to-peer (P2P) networks, traffic may be routed directly to other peers, or servers may be operated by third parties (sometimes called "federated networks"). This form of communication is still popular in certain games or services, for example BitTorrent is primarily a P2P service; and titles from the Counter-Strike series are federated, with a wide selection of third party hosts.

The main advantage of peer-to-peer networks over centralised networks is longevity. Games such as Unreal Tournament 99 (which is federated) still have playable servers, as the servers are community-run, and so as long as people still wish to play the game, they will remain online (despite the original developers no longer making any profit from the title) [3].

However, security can often be worse in fully peer-to-peer networks than that of fully centralised networks. Peers may send malicious communications, or behave in ways that violate the general rules of the service. As there is no trusted server, there is no easy way to validate communications to prevent peers from cheating.

Some peer-to-peer services try to address issues with security. In file-sharing protocols such as BitTorrent, a tracker supplies hashes of the file pieces to validate the file being downloaded [8]. However, the downside of this approach is that a trusted party (in this case the tracker) is still required. A malicious tracker could supply bad hashes, or an outdated tracker may expose the peer to security vulnerabilities.

2.3 Untrusted setups

Currently, there exists an online centralised version of the board game Risk.

We aim to apply bit-commitment schemes and zero-knowledge proof protocols to an online P2P variant of Risk, to allow peers to play the game whilst preventing cheating and needing no trusted parties. The variant of interest is the "fog of war" variant, where a player cannot see the unit counts of regions besides those that they own or are neighbouring.

3 Literature review

Centralised systems can securely perform the generation of random values, through using a cryptographically secure random number generator on the server-side, and distributing the values to the clients. This is how dice rolls are processed in centralised online games. However, in a P2P system, something else must be done to simulate the randomness.

For dice rolling, we want that

- No peer can change the probable outcome of the dice (random),
- No peer can deny having rolled the dice (non-repudiation).

We apply the concept of bit commitment schemes to form these guarantees.

3.1 Bit commitment schemes

Bit commitment schemes provide a mechanism for one party to commit to some hidden value and reveal it later. This can be achieved through the use of commutative cryptographic algorithms and with one-way functions.

Commutative cryptography

[23] provides a protocol using bit commitment to play poker. They offer a bit commitment scheme using commutative encryption algorithms based on modular arithmetic. This scheme works by each player encrypting cards, and decrypting in a different order as to obscure the value of the actual cards until all players have decrypted.

Many encryption schemes are not commutative however. One alternative is to use some well-known one-way function, such as SHA, with randomly generated salts.

Bit commitment with one-way functions

Bit commitment schemes can also be implemented using one-way functions:

1. The first party decides on the value m to be committed to.

2. The first party generates some random value r.
3. The first party generates and publishes some value $c = H(m, r)$, where H is an agreed-upon public one-way function.
4. The first party publishes m and r to the second party some time later.
5. The second party computes $c' = H(m, r)$ and validates that $c = c'$.

[5] provides a protocol for flipping fair coins across a telephone, which is isomorphic to selecting a random value from a set of two values. This cannot be simply repeated though to generate numbers in the range of 1-6, as 6 is not a power of 2.

However, a similar protocol can be used where each player commits to a single value $x \in \mathbb{Z}_6$. As the distribution of outcomes of addition in the group $\mathbb{Z}_n$ is fair, we can then sum the values of x committed to by both players to deduce a final value for the roll. To decrease the amount of communications required for rolling a number of dice, a vector of values can be used.

This protocol relies only on the ability for one party to produce random numbers. We can consider the $\mathbb{Z}_6$-set on $\mathbb{Z}_6$: upon one party selecting $x \in \mathbb{Z}_6$, the other party's selection is from the group $x \cdot \mathbb{Z}_6 = \{x + 0, \ldots, x + 5\} \cong \mathbb{Z}_6$. So, the potential outcomes only require one party to select randomly.

If both parties were to collude and generate non-randomly, this protocol falls through. A potential way around this is to involve other players in the protocol: the same rule applies of only a single player needs to be selecting randomly to produce random outputs. Therefore, so long as there are non-colluding players, this would protect against basic collusion.

3.2 Zero-knowledge proofs

Zero-knowledge proofs form a subset of minimum disclosure proofs, and beyond that, a subset of interactive proofs. Zero-knowledge proofs are typically defined by three properties:

- **Completeness.** If the conjecture is true, an honest verifier will be convinced of its truth by a prover.
- **Soundness.** If the conjecture is false, a cheating prover cannot convince an honest verifier (except with some small probability).
- **Zero-knowledge.** This is the condition for a minimum disclosure proof to be considered zero-knowledge. If the conjecture is true, the verifier cannot learn any other information besides the truthfulness.

Zero-knowledge proofs are particularly applicable to the presented problem. They primarily solve two problems:

- The disclosure of some information without leaking other information,
- The proof presented can only be trusted by the verifier, and not by other parties.

We can further formalise the general description of a zero-knowledge proof. [16] provides a common formalisation of the concept of a zero-knowledge proof system for a language L

by stating that

- For every $x \in L$, the verifier will accept x following interaction with a prover.
- For some polynomial p and any $x \notin S$, the verifier will reject x with probability at least $\frac{1}{p(|x|)}$.
- A verifier can produce a simulator S such that for all $x \in L$, the outputs of $S(x)$ are indistinguishable from a transcript of the proving steps taken with the prover on x.

The final point describes a proof as being *computationally zero-knowledge*. Some stronger conditions exist, which describe the distributions of the outputs of the simulator versus the distributions of the outputs of interaction with the prover.

- **Perfect.** A simulator produced by a verifier produces outputs that are distributed identically to real transcripts.
- **Statistical.** A simulator produced by a verifier gives transcripts distributed identically, except for some constant number of exceptions.

Some proofs described are *honest-verifier* zero-knowledge proofs. In these circumstances, the verifier is required to act in accordance with the protocol for the simulator distribution to behave as expected. We consider verifiers as honest, as it appears they may only impede themselves by acting dishonestly.

Games as graphs

The board used to play Risk can be viewed as an undirected graph. Each region is a node, with edges connecting it to the adjacent regions. For convenience, we also consider the player's hand to be a node, which has all units not in play placed upon it.

Furthermore, the actions taken when playing the game can be seen as constructing new edges on a directed weighted graph. This makes us interested in the ability to prove that the new edges conform to certain rules.

The main game protocol can be considered as the following graph mutations for a player P:

- **Reinforcement.** A player updates the weight on some edges of the graph that lead from the hand node H_P to region nodes $R_1, \ldots, R_n$ in their control.
  - Any adjacent players will then need to undergo proving the number of units on neighbouring regions.
- **Attack.** Player P attacks R_B from R_A. In the event of losing units, the player updates the edge on the graph from R_A to the hand node H_P.

  In the event of winning the attack, the player updates the edge from R_A to R_B to ensure some non-zero amount of units is located in the region.
- **Unit movement.** The player updates an edge from one region R_1 to another neighbouring region R_2.

The goal is then to identify ways to secure this protocol by obscuring the edges and weights, whilst preventing the ability for the player to cheat.

Graphs & ZKPs

[12] identifies methods to construct zero-knowledge proofs for two graphs being isomorphic or non-isomorphic.

Identifying Risk as a graph therefore enables us to construct isomorphisms as part of the proof protocol. For example, when a player wishes to commit to a movement, it is important to prove that the initial node and the new node are adjacent. This can be proven by communicating isomorphic graphs, and constructing challenges based on the edges of the original graph.

Adjacency proofs

Proving adjacency of two nodes is akin to proving isomorphism of two graphs. A protocol using challenges could be constructed as follows:

1. The prover commits a new edge between two nodes.
2. The prover constructs an isomorphic graph to the game, and encrypts the edges.
3. The verified challenges either:
    - That the graphs are isomorphic.
    - That the new edge is valid.
4. The prover sends a total decryption key for the graph's nodes, to prove isomorphism to the game board; or a decryption key for the new edge to the isomorphism, to prove adjacency.

These challenges restrict the ability for the prover to cheat: if the two nodes they are committing to are not adjacent, either the prover will need to commit an invalid isomorphism (detected by challenge 1), or lie about the edge they have committed (detected by challenge 2).

Selection between two challenges is the ideal number of challenges to use, as the probability of cheating being detected is $\frac{1}{2}$. Using more challenge options (e.g, n) means the likelihood of the prover cheating a single challenge reduces to $\frac{1}{n}$. This would require much larger numbers of communications to then convince the verifier to the same level of certainty.

Adjacency proofs are necessary to ensure that players move units fairly.

Cheating with negative values

Zerocash is a ledger system that uses zero-knowledge proofs to ensure consistency and prevent cheating. Ledgers are the main existing use case of zero-knowledge proofs, and there are some limited similarities between ledgers and Risk in how they wish to obscure values of tokens within the system.

Publicly-verifiable preprocessing zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) are the building blocks of Zerocash [4], and its successor Zcash. A zk-SNARK consists of three algorithms: `KeyGen`, `Prove`, `Verify`.

These are utilised to construct and verify transactions called `POUR`s. A `POUR` takes, as input, a certain "coin", and splits this coin into multiple outputs whose values are non-negative

and sum to the same value as the input. The output coins may also be associated with different wallet addresses.

Zerocash then uses zk-SNARKs as a means to prove that the value of the inputs into a `POUR` is the same as the value of the outputs. This prevents users from generating "debt", or from generating value without going through a minting process (also defined in the Zerocash spec).

Ensuring consistency of weights

A similar issue appears in the proposed system: a cheating player could update the weights on their graph to cause a region to be "in debt". Therefore, we need the protocol to ensure players prove that the sum of all edges is equal to how many units the player has in play (a well-known value).

Additive homomorphic cryptosystems

Some cryptosystems admit an additive homomorphic property: that is, given the public key and two encrypted values $\sigma_1 = E(m_1), \sigma_2 = E(m_2)$, the value $\sigma_1 + \sigma_2 = E(m_1 + m_2)$ is the ciphertext of the underlying operation.

[19] defined a cryptosystem based on residuosity classes, which expresses this property. [9] demonstrates an honest-verifier zero-knowledge proof for proving a given value is 0. Hence, clearly, proving a summation $a + b = v$ can be performed by proving $v - a - b = 0$ in an additive homomorphic cryptosystem.

So, using some such scheme to obscure edge weights should enable verification of the edge values without revealing their actual values.

Reducing communication

In the presented algorithms, interaction is performed fairly constantly, leading to a large number of communications. This will slow the system considerably, and make proofs longer to perform due to network latency.

An alternative general protocol is the Σ-protocol [13]. In the Σ-protocol, three communications occur:

- The prover sends the conjecture.
- The verifier sends a random string.
- The prover sends some proofs generated using the random string.

This reduces the number of communications to a constant, even for varying numbers of challenges.

The Fiat-Shamir heuristic [10] provides another method to reduce communication by constructing non-interactive zero-knowledge proofs using a random oracle. For ledgers, non-interactive zero-knowledge proofs are necessary, as the ledger must be resilient to a user going offline. However, in our case, users should be expected to stay online for an entire session of Risk, and each session is self-contained. So this full transformation is not necessary.

Set membership proofs

Another approach to the problem is to use set membership, which is a widely considered problem in zero-proof literature. In this case, each region would be associated with a set of units from a public "pool" of units. Then, a player needs to prove the cardinality of a set, and the uniqueness/distinctness of its members. A number of constructs exist for analysing and proving in obscured sets.

4 Implementation

The implementation provided uses WebSockets as the communication primitive. This is therefore a centralised implementation. However, no verification occurs in the server code, which instead simply "echoes" messages received to all connected clients.

Despite this approach being centralised, it does emulate a fully peer-to-peer environment, and has notable benefits:

- It is faster to develop, use, and test than using a physical system such as mail;
- There is no need for hole-punching or port-forwarding;
- WebSockets are highly flexible in how data is structured and interpreted.

In particular, the final point allows for the use of purely JSON messages, which are readily parsed and processed by the client-side JavaScript.

4.1 Message structure

Messages are given a fixed structure to make processing simpler. Each JSON message holds an `author` field, being the sender's ID; a message ID to prevent replay attacks and associate related messages; and an `action`, which at a high level dictates how each client should process the message.

The action more specifically is one of `ANNOUNCE`, `DISCONNECT`, `KEEPALIVE`, `RANDOM`, `PROOF`, and `ACT`. The first three of these are used for managing the network by ensuring peers are aware of each other and know the state of the network. `RANDOM` and `PROOF` are designated to be used by sub-protocols defined later on. `ACT` is used by players to submit actions for their turn during gameplay.

Each message is also signed to verify the author. This is a standard application of RSA. A hash of the message is taken, then encrypted with the private key. This can be verified with the public key.

Players trust RSA keys on a trust-on-first-use (TOFU) basis. TOFU is the same protocol as used by Gemini [1]. The main issue with TOFU is that if a malicious party intercepts the first communication, they may substitute the RSA credentials transmitted by the intended party, resulting in a man-in-the-middle attack.

4.2 Paillier cryptosystem

Paillier requires the calculation of two large primes for the generation of public and private key pairs. ECMAScript typically stores integers as floating point numbers, giving precision

up to 2^{53}. This is clearly inappropriate for the generation of sufficiently large primes.

In 2020, ECMAScript introduced `BigInt` [25], which are, as described in the spec, "arbitrary precision integers". Whilst this does not hold true in common ECMAScript implementations (such as Chrome's V8), these "big integers" still provide sufficient precision for the Paillier cryptosystem, given some optimisations and specialisations are made with regards to the Paillier algorithm and in particular the modular exponentiation operation.

It must be noted that `BigInt` is inappropriate for cryptography in practice, due to the possibility of timing attacks as operations are not necessarily constant time [25]. In particular, modular exponentiation is non-constant time, and operates frequently on secret data. A savvy attacker may be able to use this to leak information about an adversary's private key; however, as decryption is not performed, this risk is considerably reduced as there is less need to perform optimisations based on Chinese remainder theorem which would require treating the modulus n as its two components p and q.

4.3 Modular exponentiation

As `BigInt`'s V8 implementation does not optimise modular exponentiation, we employ the use of addition chaining, as described in [21]. Addition chaining breaks a modular exponentiation into repeated square-and-modulo operations, which are computationally inexpensive to perform.

The number of operations is dependent primarily on the size of the exponent. For an exponent of bit length L, somewhere between L and $2L$ multiply-and-modulo operations are performed, which gives overall a logarithmic time complexity supposing bit-shifts and multiply-and-modulo are constant time operations.

4.4 Generating large primes

I chose to use primes of length 2048 bits. This is a typical prime size for public-key cryptography, as this generates a modulus $n = pq$ of length 4096 bits.

Generating these primes is a basic application of the Rabin-Miller primality test [20]. This produces probabilistic primes, however upon completing sufficiently many rounds of verification, the likelihood of these numbers actually not being prime is dwarfed by the likelihood of hardware failure.

4.5 Public key

In the Paillier cryptosystem, the public key is a pair (n, g) where $n = pq$ for primes p, q satisfying $\gcd(pq, (p-1)(q-1)) = 1$ and $g \in \mathbb{Z}^*_{n^2}$. We restrict the range of plaintexts m to $m < n$.

The Paillier cryptosystem is otherwise generic over the choice of primes p, q. However, by choosing p, q of equal length, the required property on $pq, (p-1)(q-1)$ coprime is guaranteed.

Proposition 4.1. *For p, q prime of equal length,* $\gcd(pq, (p-1)(q-1)) = 1$.

Proof. Without loss of generality, assume $p > q$. Suppose $\gcd(pq, (p-1)(q-1)) \neq 1$. Then, $q \mid p-1$. However, the bit-lengths of p, q are identical. So $\frac{1}{2}(p-1) < q$. This is a contradiction to $q \mid p-1$ (as 2 is the smallest possible divisor), and so we must have $\gcd(pq, (p-1)(q-1)) = 1$ as required. □

As the prime generation routine generates primes of equal length, this property is therefore guaranteed. The next optimisation is to select $g = 1 + n$.

Proposition 4.2. $1 + n \in \mathbb{Z}^*_{n^2}$.

Proof. We see that $(1+n)^n \equiv 1 \mod n^2$ from binomial expansion. So $1+n$ is invertible as required. □

The selection of such g is ideal, as the binomial expansion property helps to optimise exponentiation. Clearly, from the same result, $g^m = 1 + mn$. This operation is far easier to perform, as it can be performed without having to take the modulus to keep the computed value within range.

4.6 Encryption

The ciphertext is, in general, computed as $c = g^m r^n \mod n^2$ for $r < n$ some random secret value. To make this easier to compute, we compute the equivalent value $c = (r^n \mod n^2) \cdot (g^m \mod n^2) \mod n^2$.

4.7 Private key

The private key is the value of the Carmichael function $\lambda = \lambda(n)$, defined as the exponent of the group $\mathbb{Z}^*_n$. From the Chinese remainder theorem, $\lambda(n) = \lambda(pq)$ can be computed as $\text{lcm}(\lambda(p), \lambda(q))$. From Carmichael's theorem, this is equivalent to $\text{lcm}(\phi(p), \phi(q))$, where ϕ is Euler's totient function. Hence, from the definition of Euler's totient function, and as p, q are equal length, $\lambda = (p-1)(q-1) = \phi(n)$.

We are also interested in the ability to compute $\mu = \lambda^{-1} \mod n$ as part of decryption. Fortunately, this is easy, as from Euler's theorem, $\lambda^{\phi(n)} \equiv 1 \mod n$, and so we propose $\mu = \lambda^{\phi(n)-1} \mod n$. As $\phi(n)$ is well-known to us, we get $\mu = \lambda^{(p-1)(q-1)} \mod n$, a relatively straight-forward computation.

4.8 Decryption

Let c be the ciphertext. The corresponding plaintext is computed as $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$, where $L(x) = \frac{x-1}{n}$. This is relatively simple to compute in JavaScript.

4.9 Implementation details

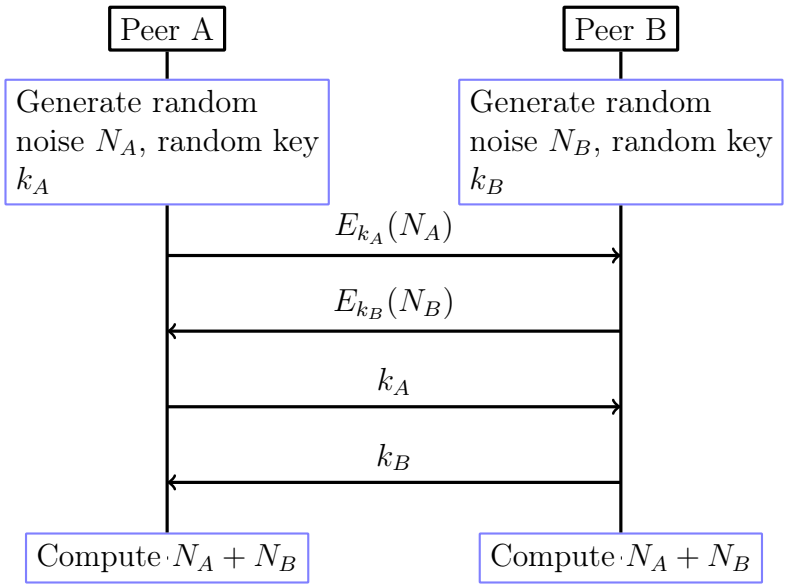
Paillier is implemented by four classes: `PubKey`, `PrivKey`, `Ciphertext`, and `ReadOnlyCiphertext`. `PubKey.encrypt` converts a `BigInt` into either a `Ciphertext` or a `ReadOnlyCiphertext` by the encryption function above. The distinction between these is that a `ReadOnlyCiphertext` does not know the random r that was used to form it, and so is created by decrypting a ciphertext that originated with another peer. A regular `Ciphertext` maintains knowledge

of r and the plaintext it enciphers. This makes it capable of proving by the scheme presented below.

4.10 Shared random values

A large part of Risk involves random behaviour dictated by rolling some number of dice. To achieve this, some fair protocol must be used to generate random values consistently across each peer without any peer being able to manipulate the outcomes.

This is achieved through bit-commitment and properties of $\mathbb{Z}_n$. The protocol for two peers is as follows, and generalises to n peers trivially.



Depending on how $N_A + N_B$ is then turned into a random value within a range, this system may be manipulated by an attacker who has some knowledge of how participants are generating their noise. As a basic example, suppose a random value within range is generated by taking $N_A + N_B \mod 3$, and participants are producing 2-bit noises. An attacker could submit a 3-bit noise with the most-significant bit set, in which case the probability of the final result being a 1 are significantly higher than the probability of a 0 or a 2. This is a typical example of modular bias. To avoid this problem, peers should agree beforehand on the number of bits to transmit. Addition of noise will then operate modulo 2^ℓ, where ℓ is the agreed-upon number of bits.

The encryption function used must also guarantee the integrity of decrypted ciphertexts to prevent a malicious party creating a ciphertext which decrypts to multiple valid values through using different keys.

Proposition 4.3. *With the above considerations, the scheme shown is not manipulable by a single cheater.*

Proof. Suppose $P_1, \ldots, P_{n-1}$ are honest participants, and P_n is a cheater with a desired outcome.

In step 1, each participant P_i commits $E_{k_i}(N_i)$. The cheater P_n commits a constructed noise $E_{k_n}(N_n)$.

The encryption function E_k holds the confidentiality property: that is, without k, P_i cannot retrieve m given $E_k(m)$. So P_n's choice of N_n cannot be directed by other commitments.

The final value is dictated by the sum of all decrypted values. P_n is therefore left in a position of choosing N_n to control the outcome of $a + N_n$, where a is selected uniformly at random from the abelian group $\mathbb{Z}_{2^\ell}$ for ℓ the agreed upon bit length.

As every element of this group is of order 2^ℓ, the distribution of $a + N_n$ is identical no matter the choice of N_n. So P_n maintains no control over the outcome of $a + N_n$. □

This extends inductively to support $n - 1$ cheating participants, even if colluding. Finally, we must consider how to reduce random noise to useful values.

4.11 Avoiding modular bias

The typical way to avoid modular bias is by resampling. To avoid excessive communication, resampling can be performed within the bit sequence by partitioning into blocks of n bits and taking blocks until one falls within range. This is appropriate in the presented use case as random values need only be up to 6, so the probability of consuming over 63 bits of noise when resampling for a value in the range 0 to 5 is $\left(\frac{1}{4}\right)^{21} \approx 2.3 \times 10^{-13}$.

4.12 Application to domain

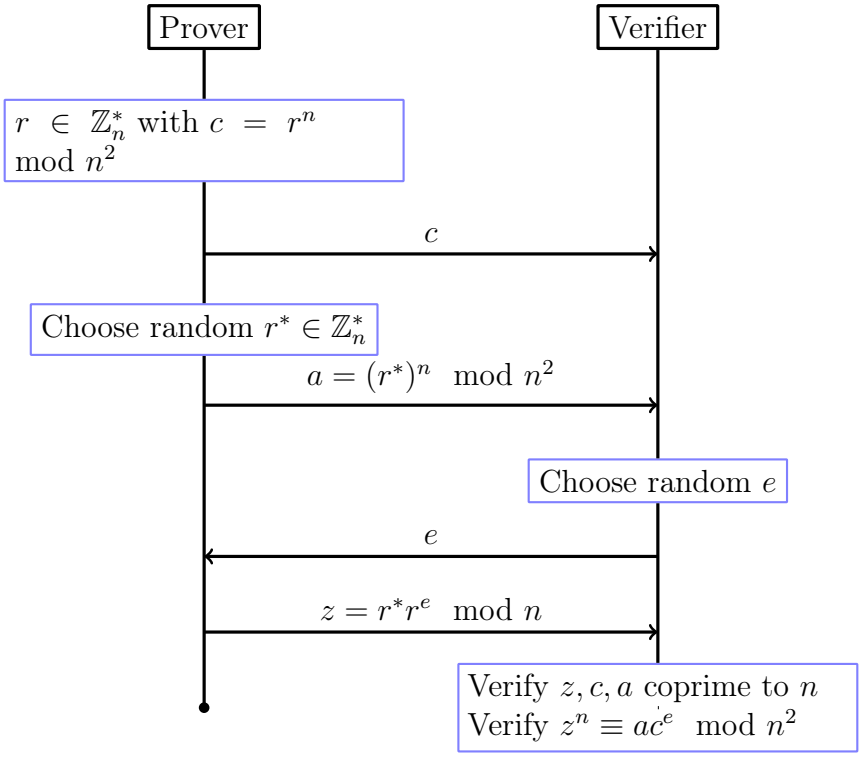
Random values are used in two places.

- Selecting the first player.
- Rolling dice.

4.13 Proof system

The first proof to discuss is that of [9]. The authors give a method to prove knowledge of an encrypted value. The importance of using a zero-knowledge method for this is that it verifies knowledge to a single party. This party should be an honest verifier: this is an assumption we have made of the context, but in general this is not true, and so this provides an attack surface for colluding parties.

The proof system presented is an interactive proof for a given ciphertext c being an encryption of zero.



A proof for the following homologous problem can be trivially constructed: given some ciphertext $c = g^m r^n \mod n^2$, prove that the text $cg^{-m} \mod n^2$ is an encryption of 0. The text cg^{-m} is constructed by the verifier. The prover then proceeds with the proof as normal, since cg^{-m} is an encryption of 0 under the same noise as the encryption of m given.

4.14 Implementation details

Proofs of zero use messages labelled as `"PROOF"` to resolve, and resolve between two parties. The proof is initiated by the verifier as part of the game protocol, who sends a request containing the region to prove. Initiating proofs on the verifier side has benefits to synchronisation, and helps to reduce race conditions, as the proof is only requested after the verifier has updated their state.

The prover responds with the fields `conjecture: int` and `a: str` (where `a` is the serialisation of a `BigInt` representing `a` and `conjecture` is the proposed plaintext).

The prover then waits on an event listener to respond to the verifier's challenge in a non-blocking way when received.

The verifier receives the message above, and responds with a random challenge selected by generating a cryptographically secure pseudorandom number of 2048 bits, and then dropping the LSB. Using 2047 bits guarantees that the challenge is smaller than p or q, as is suggested in the original paper. The verifier then waits on an event listener to receive the prover's proof.

Verifying the proof is a simple application of extended Euclidean algorithm to check coprimality, and a modular exponentiation and reduction to check the final equivalence. The ciphertext on the verifier's instance is then tagged with the proven plaintext (should the proof succeed). This tag is removed in the case that the ciphertext is updated.

4.15 Application to domain

Players should prove a number of properties of their game state to each other to ensure fair play. These are as follows.

1. The number of reinforcements placed during the first stage of a turn.
2. The number of units on a region neighbouring another player.
3. The number of units available for an attack/defence.
4. The number of units lost during an attack/defence (including total depletion of units and loss of the region).
5. The number of units moved when fortifying.

(2) and (4) are both covered by the proof above. (3) is okay between two players, as it is a subcase of (2). But in the case of more players, the availability of units should be proven. One way to achieve this is with a range proof.

[6] demonstrates a proof that some given ciphertext lies within an interval $[-\ell, 2\ell]$, where ℓ is some public value. This proof can easily be manipulated into a proof that a value lies within the interval $[n, 3\ell + n]$ from the additive homomorphic property. By selecting a sufficiently high ℓ and appropriate n, this proof is appropriate for proving to other players that the number of units being used in an attack is valid.

4.16 Range proof

[6]'s proof is a multi-round proof more similar in structure to the graph isomorphism proof presented in [12]. We select public parameter ℓ to be some sufficiently high value that a player's unit count should not exceed during play: an appropriate choice may be 1000. Select n as the number of units that the player is defending with, or in the case of attacking, let n be the number of units that the player is attacking with plus 1 (as is required by the rules of Risk).

4.17 Cheating with negative values

Using just the additive homomorphic property to guarantee (1) opens up the ability for a player to cheat by using negative values. This is a severe issue, as potentially the cheat could be completely unnoticed even in the conclusion of the game. To overcome this, we need a new protocol that is still in zero-knowledge, but proves a different property of a player's move.

One consideration is to use a range proof as above. The full proof would then be the combination of a proof that the sum of all ciphertexts is 1, and the range of each ciphertext is as tight as possible, which is within the range $[0, 3]$. This is acceptable in the specific application, however we can achieve a better proof that is similar in operation to [6].

Instead of proving a value is within a range, the prover will demonstrate that a bijection exists between the elements in the reinforcement set and a challenge set.

Protocol 4.4. The prover transmits the set

$$S = \{(R_1, E(n_1, r_1)), \ldots, (R_N, E(n_N, r_N))\}$$

as their reinforcement step. Verifier wants that the second projection of this set maps to 1 exactly once.

Run t times in parallel:

1. Prover transmits $\{(\psi(R_i), E(n_i, r_i^*)) \mid 0 < i \leq N\}$ where ψ is a random bijection on the regions.
2. Verifier chooses a random $c \in \{0, 1\}$.
   (a) If $c = 0$, the verifier requests the definition of ψ. They then compute the product of the $E(x, r_i) \cdot E(x, r_i^*)$ and verify proofs that each of these is zero.
   (b) If $c = 1$, the verifier requests a proof that each $E(n_i, r_i^*)$ is as claimed.

This protocol has the following properties, given that the proof of zero from before also holds the same properties [9].

- **Complete.** The verifier will clearly always accept S given that S is valid.
- **Sound.** A cheating prover will trick a verifier with probability 2^{-t}. So select a sufficiently high t.
- **Zero-knowledge.** Supposing each ψ, r_i, and r_i^* are generated in a truly random manner, the verifier gains no additional knowledge of the prover's private state.

Additionally, we can consider this protocol perfect zero-knowledge.

Proposition 4.5. *In the random oracle model, Protocol 4.4 is perfect zero-knowledge.*

Proof. To prove perfect zero-knowledge, we require a polynomial-time algorithm T^* such that for all verifiers and for all valid sets S, the set of transcripts $T(P, V, S) = T^*(S)$, and the distributions are identical.

Such a T^* can be defined for any S.

1. Choose a random ψ' from the random oracle.
2. Choose random $(r_i^*)'$ from the random oracle.
3. Encrypt under P's public-key.
4. Verifier picks c as before.
5. Perform proofs of zero, which are also perfect zero-knowledge from [9].

This gives T^* such that $T^*(S) = T(P, V, S)$, and the output distributions are identical. Hence, this proof is perfect zero-knowledge under random oracle model. □

4.18 Optimising

It is preferred that these proofs can be performed with only a few communications: this issue is particularly prevalent here as this protocol requires multiple rounds to complete. The independence of each round on the next is a beneficial property, as it means the proof can be performed in parallel, so the prover transmits *all* of their ψ's, then the verifier transmits all of their challenges. However, still is the issue of performing proofs of zero.

We can apply the Fiat-Shamir heuristic [10] to make proofs of zero non-interactive. In place of a random oracle, we use a cryptographic hash function. We take the hash of some public parameters to prevent cheating by searching for some values that hash in a preferable manner. In this case, selecting $e = H(g, m, a)$ is a valid choice. To get a hash of desired length, an extendable output function such as SHAKE256 [18] could be used. The library jsSHA [7] provides an implementation of SHAKE256 that works within a browser.

5 Review

5.1 Random oracles

Various parts of the implementation use the random oracle model: in particular, the zero-knowledge proof sections.

The random oracle model is used for two guarantees. The first is in the construction of truly random values that will not reveal information about the prover's state. In practice, a cryptographically secure pseudo random number generator will suffice for this application, as CSPRNGs typically incorporate environmental data to ensure outputs are unpredictable [2].

The second is to associate a non-random value with a random value. In practice, a cryptographic hash function such as SHA-3 is used. This gives appropriately pseudo-random outputs that appear truly random, and additionally are assumed to be preimage resistant: a necessary property when constructing non-interactive proofs in order to prevent a prover manipulating the signature used to derive the proof.

5.2 Efficiency

Storage complexity

Paillier ciphertexts are constant size, each ∼1.0kB in size (as they are taken modulo n^2, where n is the product of two 2048 bit primes). This is small enough for the memory and network limitations of today.

The proof of zero uses two Paillier ciphertexts, a challenge of size 2048 bits, and a proof statement of size 4096 bits. In total, this is a constant size of ∼2.8kB.

On the other hand, Protocol 4.4 requires multiple rounds. Assume that we use 42 rounds: this provides an acceptable level of soundness, with a cheat probability of $\left(\frac{1}{2}\right)^{-42} \approx 2.3 \times 10^{-13}$. Additionally, assume that there are 10 regions to verify. Each round then requires ten Paillier ciphertexts alongside ten proofs of zero. This results in a proof size of ∼1.7MB. Whilst this is still within current memory limitations, the network cost is extreme; and this value may exceed what can be reasonably operated on within a processor's cache.

This could be overcome by reducing the number of rounds, which comes at the cost of increasing the probability of cheating. In a protocol designed to only facilitate a single game session, this may be acceptable to the parties involved. For example, reducing the number of rounds to 19 will increase the chance of cheating to $\left(\frac{1}{2}\right)^{-19} \approx 1.9 \times 10^{-6}$, but the size would reduce considerably to ∼770kB.

This is all in an ideal situation without compression or signatures: in the implementation presented, the serialisation of a ciphertext is larger than this, since it serialises to a string of the hexadecimal representation and includes a digital signature for authenticity. Compression shouldn't be expected to make a considerable difference, as the ciphertexts should appear approximately random.

The size of the proof of zero communication is, in total, $3290 + 1744 + 2243$ characters, i.e ~7.3kB. This is about 2-3 times larger than the ideal size. A solution to this is to use a more compact format, for example msgpack [17] (which also has native support for binary literals).

This only considers the network footprint. The other consideration is the memory footprint. The proof of zero requires auxiliary memory beyond the new values communicated. In particular, it must clone the ciphertext being proven, in order to prevent mutating the original ciphertext when multiplying by g^{-m}.

Time complexity

It is remarked that Paillier encryption performs considerably slower than RSA on all key sizes. [19] provides a table of theoretic results, suggesting that Paillier encryption can be over 1,000 times slower than RSA for the same key size.

Timing results versus RSA are backed experimentally by my implementation. The following benchmarking code was executed.

```
console.log("Warming up")

for (let i = 0n; i < 100n; i++) {
    keyPair.pubKey.encrypt(i);
}

console.log("Benching")

performance.mark("start")
for (let i = 0n; i < 250n; i++) {
    keyPair.pubKey.encrypt(i);
}
performance.mark("end")

console.log(performance.measure("duration", "start", "end").duration)
```

Performing 250 Paillier encrypts required 48,800ms. On the other hand, performing 250 RSA encrypts required just 60ms.

The speed of decryption is considerably less important in this circumstance, as Paillier ciphertexts are not decrypted during the execution of the program.

There is little room for optimisation of the mathematics in Paillier encryption. Some possibilities are discussed below.

Public parameter. The choice of the public parameter g can improve the time complexity by removing the need for some large modular exponentiation. Selection of $g = n + 1$ is

good in this regard, as binomial theorem allows the modular exponentiation $g^m \mod n^2$ to be reduced to the computation $1 + nm \mod n^2$.

Smaller key size. The complexity of Paillier encryption increases with key size. Using a smaller key could considerably reduce the time taken [19].

Caching. As the main values being encrypted are 0 or 1, a peer could maintain a cache of encryptions of these values and transmit these instantly. Caching may be executed in a background "web worker". A consideration is whether a peer may be able to execute a timing-related attack by first exhausting a peer's cache of a known plaintext value, and then requesting an unknown value and using the time taken to determine if the value was sent from the exhausted cache or not.

Taking this idea further, one may simply cache r^n for a number of randomly generated r (as this is the slowest part of encryption). This eliminates the timing attack concern, and grants full flexibility with the values being encrypted.

Alternative Paillier scheme. [14] presents an optimised encryption scheme based on the subgroup of elements with Jacobi symbol +1. This forms a group as the Jacobi symbol is multiplicative, being a generalisation of the Legendre symbol.

Vectorised plaintexts. The maximum size of a plaintext is $|n|$: in our case, this is 4096 bits. By considering this as a vector of 128 32-bit values, peers could use a single ciphertext to represent their entire state. [22] uses this process to allow embedded devices to make use of the homomorphic properties of Paillier.

Protocol 4.4 can be modified by instead testing that the given ciphertext is contained in a set of valid ciphertexts. There would still be a large number of Paillier encryptions required during this proof.

The other proofs do not translate so trivially to this structure however. In fact, in some contexts the proofs required may be considerably more complicated, becoming round-based proofs which may be slower and use more Paillier encryptions to achieve the same effect.

Optimising language. An optimising language may be able to reduce the time taken to encrypt. On the browser, this could involve using WASM as a way to execute compiled code within the browser, although WASM does not always outperform JavaScript.

5.3 Quantum resistance

The security of Paillier relies upon the difficulty of factoring large numbers [19]. Therefore, it is vulnerable to the same quantum threat as RSA is, which is described by [24]. Alternative homomorphic encryption schemes are available, which are widely believed to be quantum-resistant, as they are based on lattice methods (e.g, [11]).

5.4 Side-channels

The specific implementation is likely vulnerable to side-channel attacks. The specification for BigInt does not specify that operations should be constant-time, and variation between browser engines could lead to timing attacks.

6 Wider application

Peer-to-peer software is an area of software that has fallen somewhat out of interest in more recent years, as companies can afford to run their own centralised servers (although no doubt interest still exists: many users are preferring federated services over centralised services, such as Mastodon, Matrix, XMPP). However, peer-to-peer solutions still have many benefits to end users: mainly being greater user freedom. I believe that the content presented here shows clear ways to expand peer-to-peer systems, and reduce dependence on centralised services.

I propose some ideas which could build off the content here.

6.1 Larger scale P2P games

Presented here was a basic implementation of a reduced rule-set version of the board game Risk. However, many other games exist that the same transformation could be applied to. Games of larger scale with a similar structure, such as Unciv, could benefit from peer-to-peer networking implemented in a similar manner.

This is not without its downsides: I found that the complexity of P2P networking is far greater than a standard centralised model. This would be a considerable burden on the developers, and could hurt the performance of such a game. The time taken to process and verify proofs also makes this inapplicable to games that are real-time.

6.2 Decentralised social media

The schemes presented here and in [9] could be applies to the concept of a decentralised social media platform. Such a platform may use ZKPs as a way to allow for "private" profiles: the content of a profile may stay encrypted, but ZKPs could be used as a way to allow certain users to view private content in a manner that allows for repudiation, and disallows one user from sharing private content to unauthorised users.

The obvious issue is P2P data storage. Users could host their own platforms, but this tends to lead to low adoption due to complexity for normal people. IPFS is a P2P data storage protocol that could be considered. This poses an advantage that users can store their own data, if they have a large amount, but other users can mirror data effectively to protect against outages. The amount of storage can grow effectively as more users join the network.

6.3 Handling of confidential data

The ability to prove the contents of a dataset to a second party without guaranteeing authenticity to a third party is another potential application of the protocol presented. Handling of confidential data is a critical concern for pharmaceutical companies, where a data leak imposes serious legal and competitive consequences for the company. A second party does however need some guarantee that the data received is correct. Proofs are one way of achieving this, although other techniques such as keyed hashing may be more effective.

Another consideration in this domain is the use of homomorphic encryption schemes to allow a third party to process data without actually viewing the data. This protects the

data from viewing by the third party, and the processing methods from viewing by the first party. [15] states for example that common statistical functions such as regression can be performed on data that is encrypted under the Paillier scheme.

7 Limitations

Finally, I present a summary of other limitations that I encountered.

7.1 JavaScript

To summarise, JavaScript was the incorrect choice of language for this project. Whilst the event-based methodology was useful, I believe overall that JavaScript hampered development.

JavaScript is a slow language. Prime generation takes a considerable amount of time, and this extends to encryption and decryption being slower than in an implementation in an optimising compiled language.

JavaScript's type system makes debugging difficult. It is somewhat obvious that this problem is far worse in systems with more interacting parts, which this project certainly was. TypeScript may have been a suitable alternative, but most likely the easiest solution was to avoid both and go with a language that was designed with stronger typing in mind from the outset (even Python would likely have been easier, as there is at least no issue of `undefined`ness, and the language was designed with objects in mind from the start).

7.2 General programming

Peer-to-peer programming requires a lot more care than client-server programming. This makes development far slower and far more bug-prone. As a simple example, consider the action of taking a turn in Risk. In the peer-to-peer implementation presented, each separate peer must keep track of how far into a turn a player is, check if a certain action would end their turn (or if its invalid), contribute in verifying proofs, and contribute in generating randomness for dice rolls. In a client-server implementation, the server would be able to handle a turn by itself, and could then propagate the results to the other clients in a single predictable request.

The use of big integers leads to peculiar issues relating to signedness. This is in some ways a JavaScript issue, but would also be true in other languages. Taking modulo n of a negative number tends to return a negative number, rather than a number squashed into the range $[0, n]$. This leads to inconsistencies when calculating the GCD or finding Bezout coefficients. In particular, this became an issue when trying to validate proofs of zero, as the GCD returned -1 rather than 1 in some cases. Resolving this simply required changing the update and encrypt functions to add the modulus until the representation of the ciphertext was signed correctly. Whilst the fix for this was simple, having to fix this in the first place is annoying, and using a non-numerical type (such as a byte stream) may resolve this in general.

7.3 Resources

The peer-to-peer implementation requires more processing power and more bandwidth on each peer than a client-server implementation would. This is the main limitation of the peer-to-peer implementation. The program ran in a reasonable time, using a reasonable amount of resources on the computers I had access to, but these are not representative of the majority of people. Using greater processing power increases power consumption, which is definitely undesirable. In a client-server implementation, even with an extra computer, I predict that the power consumption should be lower than the peer-to-peer implementation presented.

Bibliography

[1]

[2] *random, urandom - kernel random number source devices*, September 2017.

[3] Eatsleeput.com, Feb 2022. Archive: https://archive.ph/Gp0Ou.

[4] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[5] M. Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[6] F. Boudot. Efficient proofs that a committed number lies in an interval. In *International Conference on the Theory and Application of Cryptographic Techniques*, 2000.

[7] Caligatio. jssha: A javascript/typescript implementation of the complete secure hash standard (sha) family. `https://github.com/Caligatio/jsSHA`, 2022.

[8] B. Cohen. Bittorrent.org, Feb 2017.

[9] I. Damgård, M. Jurik, and J. Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9:371–385, 04 2003.

[10] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[11] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, jul 1991.

[13] J. Groth. *Honest verifier zero-knowledge arguments applied.* PhD thesis, BRICS, 2004.

[14] M. Jurik. Extensions to the paillier cryptosystem with applications to cryptological protocols. 2003.

[15] H. Ma, S. Han, and H. Lei. Optimized paillier's cryptosystem with fast encryption and decryption. In *Annual Computer Security Applications Conference*, ACSAC '21, page 106–118, New York, NY, USA, 2021. Association for Computing Machinery.

[16] A. Mohr. A survey of zero-knowledge proofs with applications to cryptography. *Southern Illinois University, Carbondale*, pages 1–12, 2007.

[17] msgpack. Messagepack: Spec. `https://github.com/msgpack/msgpack`, 2021.

[18] N. I. of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, U.S. Department of Commerce, Washington, D.C., 2015.

[19] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[20] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[21] B. Schneier. *Applied cryptography*. John Wiley, 1996.

[22] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, and W. Hu. Talos: Encrypted query processing for the internet of things. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, page 197–210, New York, NY, USA, 2015. Association for Computing Machinery.

[23] A. Shamir, R. L. Rivest, and L. M. Adleman. *Mental Poker*, pages 37–43. Springer US, Boston, MA, 1981.

[24] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[25] TC39. Bigint: Arbitrary precision integers in javascript. `https://github.com/tc39/proposal-bigint`, 2020.