Cryptographic protocol for playing Risk in an untrusted setting

Jude Southworth

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

0.1 Disambiguation

Symbol	Meaning
$\lvert a \rvert$	Bit length of value a
$\left(\frac{a}{b}\right)$	Jacobi symbol for a, b or division (context dependent)
$\frac{a}{b}$	Division
$\mathbb{Z}_k$	Additive group of integers modulo k
$\mathbb{Z}_k^*$	Multiplicative group of units modulo k
$\gcd(a, b)$	Greatest common divisor of a, b
$\operatorname{lcm}(a, b)$	Least common multiple of a, b
ϕ	Euler's totient function
λ	Carmichael's totient function
$H(\dots)$	Ideal cryptographic hash function

Chapter 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.

1.1 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.

1.1.1 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.

1.1.2 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) [7].

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 [5]. 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 peers to security vulnerabilities.

1.1.3 Untrusted setups

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

I aim to apply bit-commitment schemes, zero-knowledge proofs, and homomorphic encryption 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 the game that is 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.

1.2 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.

1.2.1 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

Protocols exist that utilise bit commitment to play poker [22]. 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.

However, almost all well-documented encryption schemes are not commutative. 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 $c = c'$.

Protocols exist for flipping fair coins "across a telephone", which is isomorphic to selecting a random value from a set of two values [2]. 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 should protect against basic collusion.

1.2.2 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 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. The common formalisation of the concept of a zero-knowledge proof system for a language L is

- 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. This imposes a significant issue: a malicious verifier may intentionally produce challenges that reveal more information.

One solution to this is to transform a proof into a non-interactive zero-knowledge proof. The Fiat-Shamir transformation [9] converts an interactive zero-knowledge proof into a non-interactive zero-knowledge proof. In this process, the ability for a verifier to behave maliciously is lost, as the verifier no longer produces challenges themselves. This relies strongly upon the random-oracle model however [18]. As the random-oracle model is not realistically attainable, it must be approximated, typically by a cryptographic hash function. This introduces greater ability for the prover to cheat if they know a preimage in the hash function used.

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 & zero-knowledge proofs

A typical example for zero-knowledge proofs is graph isomorphism [11].

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.

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 [1], 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.

The Paillier cryptosystem, which is based on composite residuosity classes express the additive homomorphic property [17]. This is due to the structure of ciphertexts in the Paillier cryptosystem. A public key is of structure (n, g), where n is the product of two large primes and g is a generator of $\mathbb{Z}_n^*$. Under the public key, the encryption c of a message m is computed as

$$c = g^m r^n \mod n^2$$

for some random $r \in \mathbb{Z}_{n^2}^*$.

The Paillier cryptosystem has disadvantages in its time and space complexity compared to other public-key cryptosystems such as RSA. In space complexity, Paillier ciphertexts are twice the size of their corresponding plaintext, as for a modulus n, ciphertexts are computed modulo n^2 for a message in range up to n. This cost can be reduced by employing some form of compression on the resulting ciphertexts.

The main concern is the issue of time complexity of Paillier. Theoretic results based on the number of multiplications performed indicate that Paillier can be 1,000 times slower than RSA encryption. Many optimisations have been presented of the Paillier cryptosystem.

The first is in the selection of public parameter g. The original paper suggests a choice of $g = 2$, however the choice of $g = 1 + n$ is very common, as the exponentiation $g^m = 1 + mn$ by binomial theorem.

Another optimisation is that of Jurik [13, Section 2.3.1]: Jurik proposes that the public-key is instead (n, g, h), where h is the generator of the group $\mathbb{Z}_n^*[+]$ (the group of units with Jacobi symbol $+1$). Then, an encryption c' of a message m is computed as

$$c' = g^m (h^r \mod n)^n \mod n^2$$

for some random $r \in \mathbb{Z}_n^*$.

The optimisation comes in two parts: firstly, the mantissa is smaller, resulting in faster multiplications. Secondly, by taking $h_n = h^n \mod n^2$, we find the following equivalence:

$$(h^r \mod n)^n \mod n^2 = h_n^r \mod n^2$$

Exponentials of the fixed base h_n can then be pre-computed to speed up exponentiation by arbitrary r.

Jurik states that the optimised form can lead to a theoretic four times speedup over Paillier's original form.

Zero-knowledge proofs in Paillier cryptosystem

There exist honest-verifier zero-knowledge proofs for proving a given value is 0 [6, Section 5.2]. 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 [12]. 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 [9] 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. This is not the same in our case, however non-interactive zero-knowledge proofs are still beneficial. The amount of communications can be reduced significantly, and it is easier to write code that can settle a non-interactive proof than an interactive proof.

The downside of using the Fiat-Shamir heuristic in our implementation is that any third party can verify proofs. In some situations, we do not want this to be the case.

Chapter 2

Implementation

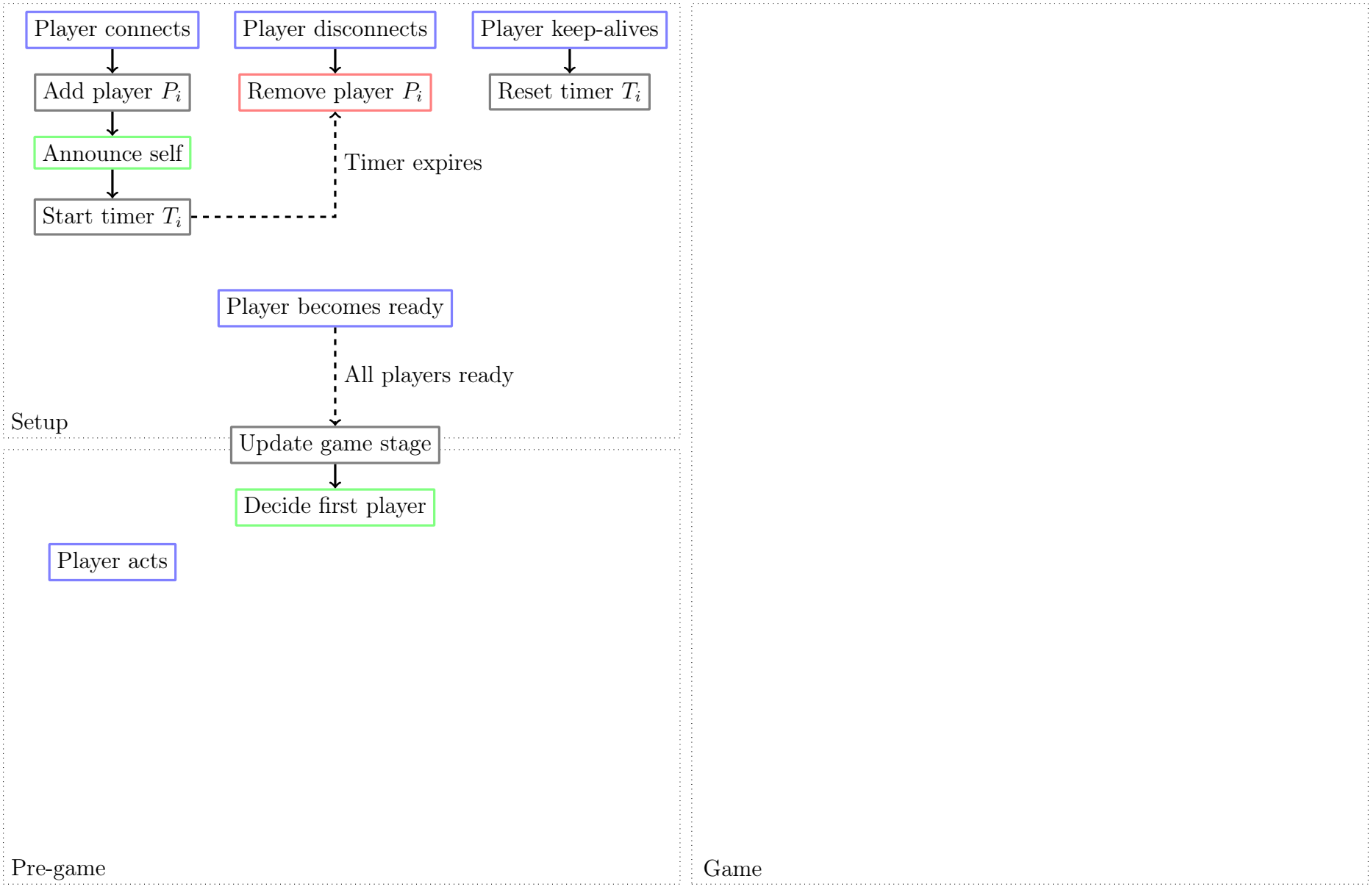
The implementation provided uses WebSockets as the communication primitive. This is therefore a centralised implementation. However, no verification occurs in the server code, which instead simply "echoes" messages received to all connected clients.

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

- 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.

The game is broken down into three main stages, each of which handles events in a different way. These are shown below.


Player connects
Player disconnects
Player keep-alives
Add player P_i
Remove player P_i
Reset timer T_i
Announce self
Timer expires
Start timer T_i
Player becomes ready
All players ready
Setup
Update game stage
Decide first player
Player acts
Pre-game
Game

2.1 Message structure

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

The "action" 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. `ANNOUNCE` is transmitted upon a player joining to ensure the new player is aware of all other players. The `ANNOUNCE` message contains the player's encryption keys and the player's ID.

`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 SHA-3 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 [24]. 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.

2.2 Paillier cryptosystem

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 for the Paillier cryptosystem.

In 2020, ECMAScript introduced `BigInt`, which are, as described in the spec, "arbitrary precision integers" [25]. 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.

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.

2.2.1 Modular exponentiation

As `BigInt`'s V8 implementation does not optimise modular exponentiation itself, we employ the use of addition chaining [20]. Addition chaining breaks a modular exponentiation into repeated square-and-modulo operations, which are less expensive 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.

2.2.2 Generating large primes

Generating primes is a basic application of the Rabin-Miller primality test [19]. 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 some other failure, such as hardware failure.

2.2.3 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 $g \in \mathbb{Z}^*_{n^2}$. The range of plaintexts m is restricted to $0 < m < n$.

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

Proposition 2.2.1. *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 step is to select the public parameter g as $g = 1 + n$.

Proposition 2.2.2. $1 + n \in \mathbb{Z}^*_{n^2}$.

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

Besides reducing the number of operations to perform exponentiation, exponentiation also does not require auxiliary memory to store intermediary values used in the calculation.

In Jurik's form, we also need to compute h, a generator of the Jacobi subgroup, and impose restrictions on p, q. In particular, it is required that $p \equiv q \equiv 3 \mod 4$, $\gcd(p-1, q-1) = 2$, and that $p-1, q-1$ consist of large factors except for 2. One method to guarantee this is to use safe primes, which are primes of form $2p + 1$ for p prime.

Proposition 2.2.3. *For $p > 5$ a safe prime, $p \equiv 3 \mod 4$*

Proof. Let q prime and $p = 2q + 1$ the corresponding safe prime. Then,

$$\begin{aligned} q \equiv 1 \mod 4 &\implies 2q + 1 \equiv 3 \mod 4 \\ q \equiv 3 \mod 4 &\implies 2q + 1 \equiv 3 \mod 4 \end{aligned}$$

as required. □

Proposition 2.2.4. *For safe primes $p \neq q$ with $p, q > 5$,* $\gcd(p-1, q-1) = 2$

Proof. As p, q are safe, $\frac{p-1}{2}$ and $\frac{q-1}{2}$ are prime. So

$$\gcd\left(\frac{p-1}{2}, \frac{q-1}{2}\right) = 1 \implies \gcd(p-1, q-1) = 2$$

□

To identify safe primes, first we generate a prime p, and then test the primality of $\frac{p-1}{2}$. Finally, to get the public parameter h, we compute $h = -x^2 \mod n$ for some random $x \in \mathbb{Z}_n^*$. With high likelihood x is coprime to n, and so the Jacobi symbol is computed as

$$\left(\frac{-x^2}{n}\right) = \left(\frac{-x^2}{p}\right)\left(\frac{-x^2}{q}\right) = (-1)^2 = 1$$

This gives us our public key (n, g, h).

2.2.4 Encryption

In the original Paillier scheme, ciphertexts are computed as $E(m, r) = c = g^m r^n \mod n^2$ for $r < n$ some random secret value. In Jurik's form, ciphertexts are computed as

$$E'(m, r) = c' = g^m (h^r \bmod n)^n \equiv g^m (h^n \bmod n)^r \mod n^2$$

Jurik remarks that $E'(m, r) = E(m, h^r \bmod n)$.

To achieve a better speed-up, pre-computation of the fixed base $h^n \bmod n$ is used. By pre-computing powers of the powers of two, exponentiation is reduced to at most $|r|$ multiplications. Let $h[i] = h^{(2^i)} \bmod n$. Then, the following algorithm computes $h^b \bmod n$.

function FIXEDBASEEXP(b)
  $index \leftarrow 0$
  $counter \leftarrow 1$
  while $b \neq 0$ **do**
    if $b \equiv 1 \mod 2$ **then**
      $ctr \leftarrow ctr \times h[i]$
      $ctr \leftarrow ctr \bmod n$
    end if
    $i \leftarrow i + 1$
    $b \leftarrow \lfloor \frac{b}{2} \rfloor$
  end while
end function

2.2.5 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 $\text{lcm}(\phi(p), \phi(q))$. Hence, from the definition of ϕ, and as p, q are equal length, $\lambda = (p-1)(q-1) = \phi(n)$.

We also need 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 easily computable with knowledge of p, q, we get $\mu = \lambda^{(p-1)(q-1)} \mod n$, a relatively straight-forward computation.

2.2.6 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 operation can be optimised by applying Chinese remainder theorem. However, in the application presented, decryption is not used and is only useful as a debugging measure. So this optimisation is not applied.

2.2.7 Implementation details

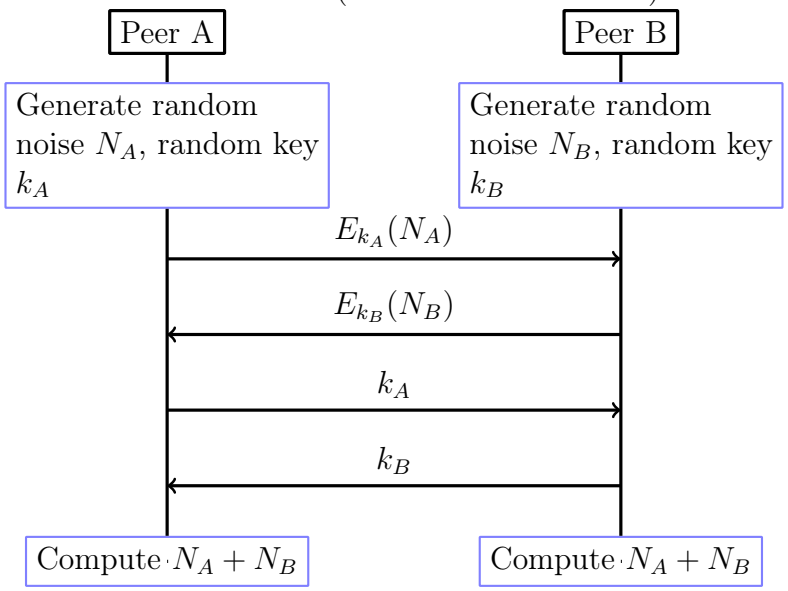
Paillier is implemented by four classes: `PubKey`, `PrivKey`, `Ciphertext`, and `ReadOnlyCiphertext`. `PubKey.encrypt` converts a `BigInt` into either a `Ciphertext` or a `ReadOnlyCiphertext` by the encryption function above. The distinction between these is that a `ReadOnlyCiphertext` does not know 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.

2.3 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.

Protocol 2.3.1 (Shared random values)**.**



To generalise this to n peers, we ensure that each peer waits to receive all encrypted noises before transmitting their decryption key.

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 an 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 is 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. To combine noises, then use the XOR operation.

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 2.3.2. *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.

2.3.1 Modular bias

A common approach is to take the modulus of the random noise. This causes modular bias to appear however, where some values are less likely to be generated.

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 $\left(\frac{1}{4}\right)^{21} \approx 2.3 \times 10^{-13}$.

2.3.2 Application to domain

Random values are used in two places.

- Selecting the first player.
- Rolling dice.

As this protocol must run many times during a game, we consider each operation of the protocol as a "session", each of which has a unique name that is derived from the context. This has another benefit as the unique name can then be used with the Web Locks API to prevent race conditions that may occur due to this protocol running asynchronously.

2.4 Proof system

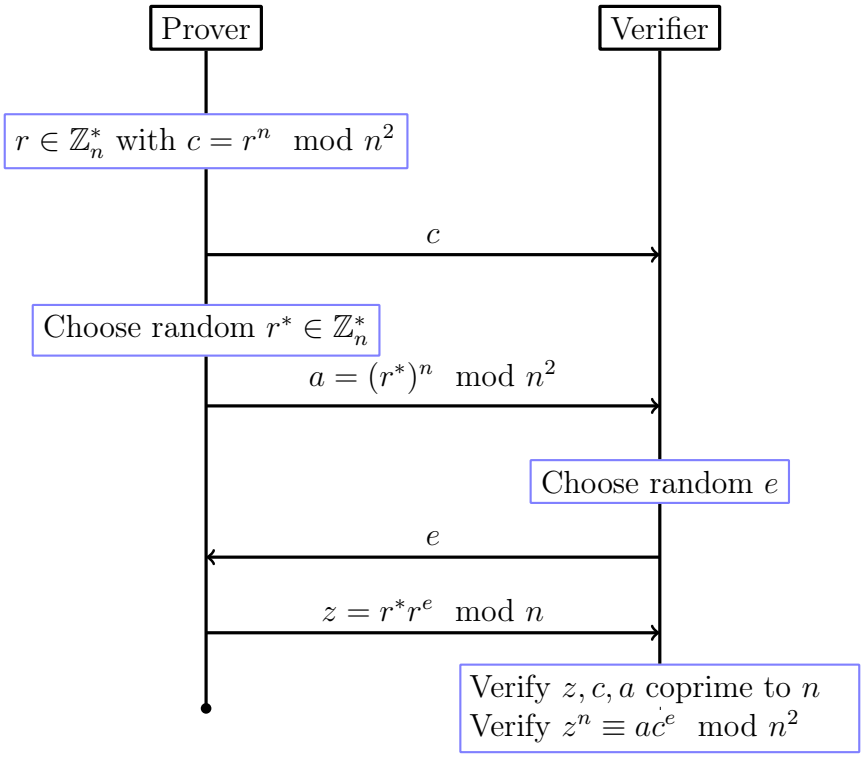
Players should prove a number of properties of their game state to each other to ensure fair play. These are as follows.

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.

These points are referenced in the following sections.

2.4.1 Proof of zero

The first proof to discuss is the honest-verifier protocol to prove knowledge that a ciphertext is an encryption of zero [6, Section 5.2].



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.

This is used in point (2), as one player can then convince a neighbour in zero-knowledge of the number of units within their region.

2.4.2 Range proof

[3, Section 2] demonstrates a proof that an encryption of a plaintext in the interval $[0, \ell]$ lies within the interval $[-\ell, 2\ell]$, where ℓ is some well-known value. So, the soundness and completeness of this proof are not the same.

Through selection of specific private inputs, a prover can create a proof for a plaintext m in the soundness interval and not the completeness interval. In this case, the proof is also not in zero-knowledge, as the verifier can infer more specific information on the value of m.

An alternative approach that is in zero-knowledge with acceptable soundness/completeness is to use a set membership proof for a set of all allowable values. This requires too much processing to be effective in this application however.

The range proof is used in points (3), (4), and (5). In (3), this is to convince other players that the number of units is sufficient for the action. In (4), this is to show that the region is not totally depleted. In (5), this is to ensure the number of units being fortified is less than the strength of the region.

2.4.3 Proving reinforcement

We now need to consider point (1). One option is to prove that the sum of the committed values is 1 by using the additive homomorphic property. However, this allows a player to cheat by using negative values. To overcome this, we want a new protocol that is still in zero-knowledge, but proves additional properties of a reinforce action.

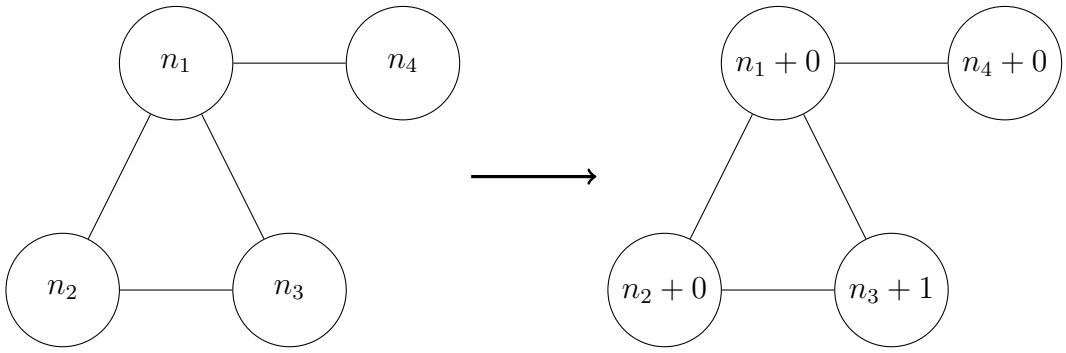


Figure 2.1: Example state change from reinforce action.

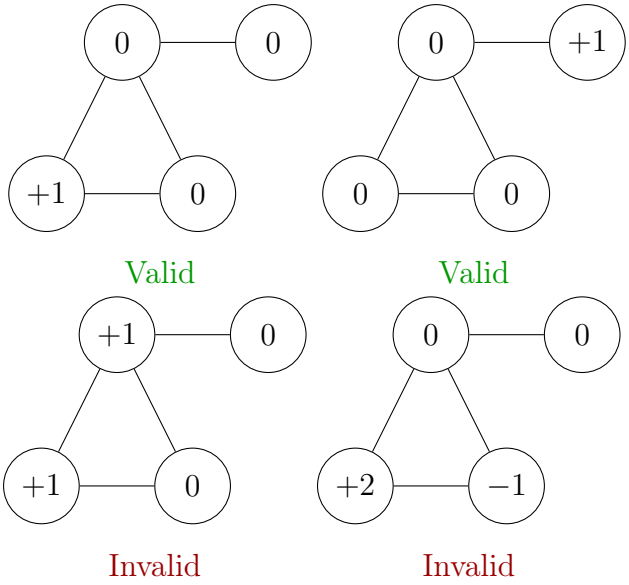


Figure 2.2: Valid and invalid reinforce messages. Notably, the final invalid message would not be caught by the additive homomorphic check.

One consideration is to use a range proof. 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. This requires a large amount of rounds though, and still has the issue of soundness/completeness.

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 known "good" set.

Protocol 2.4.1. 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 request 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 [6].

- **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, this protocol is perfectly simulatable.

Proposition 2.4.2. *Protocol 2.4.1 is perfectly simulatable in the random-oracle model.*

Proof. To prove perfect simulation, 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 simulation [6, Lemma 3].

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

This is as close to perfect zero-knowledge as we can get due to the honest-verifier condition of the zero proof.

In practice, as we are using Jurik's form of Paillier, the best we can hope for is computational zero-knowledge. This is because Jurik's form relies upon the computational indistinguishability of the sequence generated by powers of h to random powers.

2.4.4 Proving fortifications

Point (5) still remains, as the range proof alone only works to prevent negative values from appearing in a fortify action. Fortify actions need to be of form $\{k, -k, 0, \ldots, 0\}$) and the regions corresponding to $k, -k$ amounts must be adjacent.

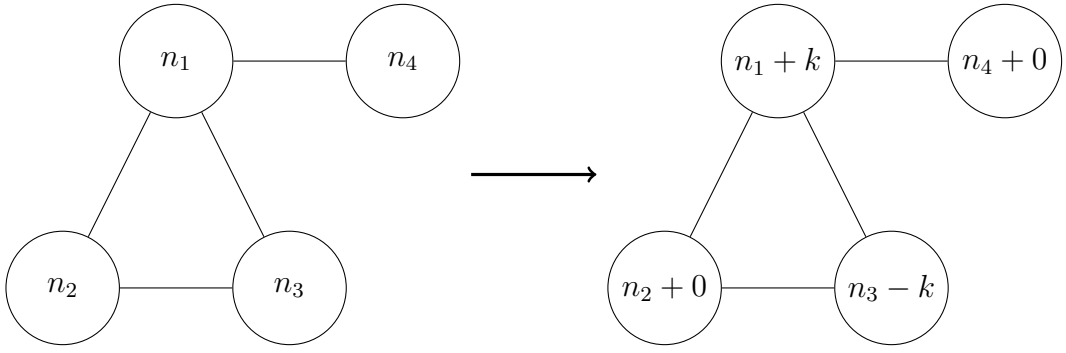


Figure 2.3: Example state change from fortify action.

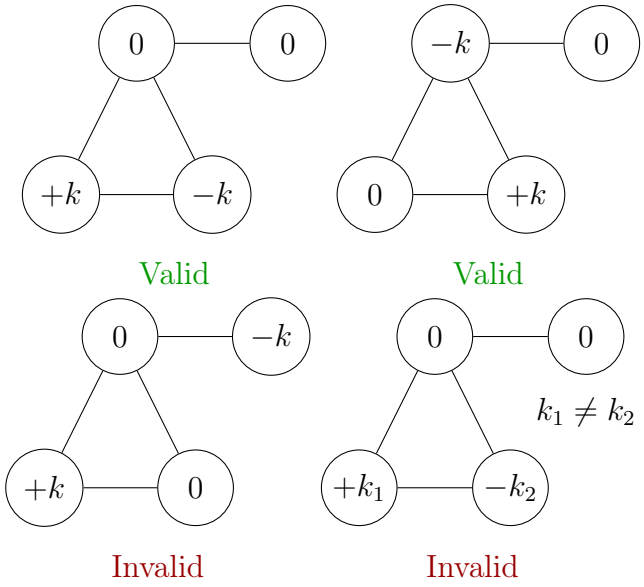


Figure 2.4: Valid and invalid fortify messages.

We combine some ideas from the graph isomorphism proofs with ideas from before to get the following protocol.

Protocol 2.4.3. The prover transmits the set

$$S = \{(R_1, E(k, r_1)), (R_2, E(-k, r_2)), (R_3, E(0, r_3)) \ldots, (R_N, E(0, r_N))\}$$

as their fortify message.

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, and $\{H(R_i, R_j, s_{ij}) \mid R_i \text{ neighbours } R_j\}$ where s_{ij} is a random salt.
2. Verifier chooses a random $c \in \{0, 1\}$.
   (a) If $c = 0$, the verifier requests the definition of ψ and each salt. They check that the resulting graph is isomorphic to the original graph. They then compute $E(n_i, r_i) \cdot E(-n_i, r_i^*)$ for each i and request a proof that each is zero. Finally, they compute each edge hash and check that there are precisely the correct number of hashes.
   (b) If $c = 1$, the verifier requests proofs that $|S| - 2$ are zero and that the remaining pair add to zero. They then request the salt used to produce the hash along the edge joining the two non-zero elements, and test that this hash is correct.

2.4.5 Optimising

It is preferred that these proofs can be performed with only a few communications: this issue is particularly prevalent for protocols requiring multiple rounds to complete. The independence of each round on the next means the proof can be performed in parallel, so the prover computes all of their private state, then the verifier computes all of their challenges. However, still is the issue of performing proofs of zero.

We can apply the Fiat-Shamir heuristic to make proofs of zero non-interactive [9]. 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 can be used [16]. The library jsSHA [4] provides an implementation of SHAKE256 that works within a browser.

Chapter 3

Review

3.1 Theoretic considerations

3.1.1 Random oracles

Various parts of the implementation use the random oracle model: in particular, the zero-knowledge proof sections. The random oracle model is theoretic, as according to the Church-Turing hypothesis, a machine cannot produce infinite truly random output with only finite input.

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 [14].

The second is to associate a non-random value with a random value. In practice, a cryptographic hash function such as SHAKE 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.

3.1.2 Quantum resistance

Paillier is broken if factoring large numbers is computationally feasible [17, Theorem 9]. Therefore, it is vulnerable to the same quantum threat as RSA is, known as Shor's algorithm [23]. Alternative homomorphic encryption schemes are available, which are believed to be quantum-resistant, as they are based on lattice methods (e.g, [10]).

3.1.3 Honest-verifier

The proof of zero is honest-verifier [6, Section 5.2]. However, applying the Fiat-Shamir heuristic converts such a proof into a general zero-knowledge proof [9, Section 5]. This means that, supposing the choice of transform used is appropriate, Protocol 2.4.1 should also be general zero-knowledge. However, the interactive proofs performed as part of

the game are still only honest-verifier, and a malicious verifier may be able to extract additional information from the prover (such as the blinding value used).

3.2 Efficiency

3.2.1 Storage complexity

Let n be the Paillier modulus.

Paillier ciphertexts are constant size, each $2|n|$ in size (as they are taken modulo n^2). This is small enough for the memory and network limitations of today.

The interactive proof of zero uses two Paillier ciphertexts (each size $2|n|$), a challenge of size $|n|$, and a proof statement of size $|n|$. In total, this is a constant size of $6|n|$.

On the other hand, the non-interactive variant needs not communicate the challenge (as it is computed as a function of other variables). So the non-interactive proof size is $5|n|$.

The non-interactive Protocol 2.4.1 requires multiple rounds. Assume that we use 48 rounds: this provides a good level of soundness, with a cheat probability of $\left(\frac{1}{2}\right)^{-48} \approx 3.6 \times 10^{-15}$. Additionally, assume that there are five regions to verify. Each prover round then requires five Paillier ciphertexts, and each verifier round five non-interactive proofs of zero plus some negligible amount of additional storage for the bijection. This results in a proof size of $(10|n| + 10|n|) \times 48 = 960|n|$. For key size $|n| = 2048$, this is $240kB$. This is a fairly reasonable size for memory and network, but this value may exceed what can be placed within a processor's cache, leading to potential slowdown during verification.

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 24 will increase the chance of cheating to $\left(\frac{1}{2}\right)^{-24} \approx 6.0 \times 10^{-8}$, but the size would reduce by approximately half.

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. In JavaScript, encoding a byte string as hexadecimal should yield approximately a four times increase in size, as one byte uses two hexadecimal characters, which are encoded as UTF-16. Results for this are shown in Table 3.3. Some potential solutions are discussed here.

Compression. One solution is to use string compression. String compression can reduce the size considerably, as despite the ciphertexts being random, the hex digits only account for a small amount of the UTF-8 character space. LZ-String, a popular JavaScript string compression library, can reduce the size of a single hex-encoded ciphertext to about 35% of its original size. This will result in some slowdown due to compression time however, but this is somewhat negligible in the face of the time taken to produce and verify proofs in the first place.

Message format. Another solution is to use a more compact message format, for example msgpack [15] (which also has native support for binary literals).

Smaller key size. The size of ciphertexts depends directly on the size of the key. Using a smaller key will reduce the size of the ciphertexts linearly.

3.2.2 Time complexity

Theoretic 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 47,000ms. On the other hand, performing 250 RSA encrypts required just 40ms. Results are shown in Table 3.1.

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

Some potential further optimisations to the implementation are as follows.

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.

Smaller key size. The complexity of Paillier encryption increases with key size. Using a smaller key could considerably reduce the time taken [17].

I tested this on top of the alternative Paillier scheme from above. This resulted in linear reductions in encryption time: encryption under a 1024-bit modulus took a sixth of the amount of time as under a 2048-bit modulus, and encryption under a 2048-bit modulus took a sixth of the amount of time as under a 4096-bit modulus.

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. This process is discussed as a way to allow embedded devices to use Paillier encryption [21].

Protocol 2.4.1 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.

3.2.3 Complexity results

All measurements were taken on Brave 1.50.114 (Chromium 112.0.5615.49) 64-bit, using a Ryzen 5 3600 CPU: a consumer CPU from 2019. Absolute timings are extremely dependent on the browser engine: for example Firefox 111.0.1 was typically 4 times slower than the results shown.

Table 3.1: Time to encrypt

Modulus size	Naïve encrypt	Jacobi encrypt	Jacobi encrypt with pre-computation	RSA encrypt
$\|n\| = 1024$	6.0ms	4ms	1.4ms	0.015ms
$\|n\| = 2048$	34ms	22ms	7.6ms	0.040ms
$\|n\| = 4096$	189ms	128ms	–	0.093ms

Table 3.2: Time[a] to process proofs

Modulus size	Proof-of-zero non-interactive		Protocol 2.4.1 with $t = 24$		Protocol 2.4.1 with $t = 48$		BCDG Range with $t = 24$	
	Prover	Verifier	Prover	Verifier	Prover	Verifier	Prover	Verifier
$\|n\| = 1024$	10ms	18ms	1,420ms	2,140ms	2,900ms	4,270ms	443ms	655ms
$\|n\| = 2048$	44ms	68ms	6,390ms	8,140ms	13,200ms	16,200ms	1,980ms	2,400ms
$\|n\| = 4096$	225ms	292ms	41,500ms	34,400ms	83,200ms	68,400ms	14,300ms	11,400ms

[a] $|n| = 4096$ uses a less-optimised encryption method, as the browser frequently timed out attempting to pre-compute for the more-optimised version.

Table 3.3: Byte size[b] of encoded proofs

Modulus size	Proof-of-zero non-interactive		Protocol 2.4.1 with $t = 24$		Protocol 2.4.1 with $t = 48$		BCDG Range with $t = 24$	
	JSON	with LZ-String	JSON	with LZ-String	JSON	with LZ-String	JSON	with LZ-String
$\|n\| = 1024$	1,617B	576B	338,902B	95,738B	673,031B	186,857B	123,354B	34,857B
$\|n\| = 2048$	3,153B	1,050B	662,233B	187,333B	1,315,463B	365,086B	252,230B	70,868B
$\|n\| = 4096$	6,226B	1,999B	1,315,027B	368,646B	2,609,131B	721,891B	484,117B	135,990B

[b] 1 UTF-16 character, as used by ECMAScript [8, Section 6.1.4], is 2 or more bytes.

Chapter 4

Wider application

Peer-to-peer software solutions have many benefits to end users: mainly being greater user freedom. I believe that the content presented here shows clear ways to extend peer-to-peer infrastructure, and reduce dependence on centralised services.

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

4.1 Larger scale P2P games

Many other games exist that the ideas presented 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. In particular, Protocol 2.3.1 would form an intrinsic part of such games.

The downsides of this are 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.

4.2 Decentralised social media

The schemes presented here could be applies to the concept of a decentralised social media platform. Such a platform may use zero-knowledge proofs as a way to allow for "private" profiles: the content of a profile may stay encrypted, but zero-knowledge proofs 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.

To store data, IPFS could be used. IPFS is a P2P data storage protocol. This poses an advantage that users can store their own data, if they have a large amount, but other users can mirror data to protect against outages or users going offline. The amount of effective storage would also grow as more users join the network.

4.3 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. For example, common statistical functions such as regression can be performed on data that is encrypted under fully homomorphic encryption schemes.

Chapter 5

Limitations

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

5.1 JavaScript

JavaScript was the incorrect choice of language for this project. Whilst the event-based methodology was useful, I believe overall that JavaScript made development much more difficult.

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. This was seen in the results shown before.

JavaScript's type system makes debugging difficult. It is somewhat obvious that this problem is far worse in systems with more interacting parts. 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).

JavaScript is a re-entrant language: this means that the interpreter does not expose threads or parallelism to the developer, but it may still use threads under-the-hood and switch contexts to handle new events. This introduces the possibility of race conditions despite no explicit threading being used. The re-entrant nature is however beneficial to a degree, as it means that long-running code won't cause the WebSocket to close or block other communications from being processed.

5.2 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. Taking modulo n of a negative number tends to return a negative number, rather than a number within 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. Using a non-numerical type (such as a byte array) may resolve this issue in general.

5.3 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 undesirable. In a client-server implementation, the power consumption should be lower than the peer-to-peer implementation presented as no processing time is spent validating proofs or using the Paillier cryptosystem, which is less efficient than the hybrid cryptosystems used in standard online communication.

Bibliography

[1] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

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

[3] Ernest F. Brickell, David Chaum, Ivan Damgård, and Jeroen van de Graaf. Gradual and verifiable release of a secret. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, page 156–166, Berlin, Heidelberg, 1987. Springer-Verlag.

[4] Caligatio. jsSHA: A JavaScript/TypeScript implementation of the complete Secure Hash Standard (SHA) family. `https://github.com/Caligatio/jsSHA`, 2022.

[5] Bram Cohen. Bittorrent.org, Feb 2017.

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

[7] EatSleepUT.com. EatSleepUT, Feb 2022. Archive: https://archive.ph/Gp0Ou.

[8] ECMA. ECMAScript 2024 language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,.*

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

[10] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] Oded Goldreich, Silvio Micali, and Avi 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.

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

[13] Mads Jurik. Extensions to the paillier cryptosystem with applications to cryptological protocols. In *BRICS Dissertation Series*, 2003.

[14] Linux man-pages project. *random, urandom - kernel random number source devices*, September 2017.

[15] msgpack. MessagePack: Spec. `https://github.com/msgpack/msgpack`, 2021.

[16] National Institute 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.

[17] Pascal 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.

[18] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 387–398, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

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

[20] Bruce Schneier. *Applied cryptography*. John Wiley, 1996.

[21] Hossein Shafagh, Anwar Hithnawi, Andreas Droescher, Simon Duquennoy, and Wen 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.

[22] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. *Mental Poker*, pages 37–43. Springer US, Boston, MA, 1981.

[23] Peter 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.

[24] Solderpunk. Project Gemini: Speculative specification, 2022.

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