Cryptographic protocol for playing Risk in an untrusted setting

Jude Southworth

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

Cryptographic protocol for playing Risk in an untrusted setting

Submitted by: Jude Southworth

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

We present a modern implementation of the Paillier cryptosystem for the browser, using Jurik's form to optimise encryption. Furthermore, we present an application of this cryptosystem with zero-knowledge proofs to enable peers to maintain private state in a peer-to-peer implementation of the board game Risk. Use of novel zero-knowledge proofs enables peers to verify that the actions of other players adhere to the game's rules without learning additional information about their private state. Finally, we present benchmarks of the implementation.

Contents

1 Outline **5**
- 1.1 Existing solutions . . . 5
  - 1.1.1 Centralised . . . 5
  - 1.1.2 Peer-to-peer networks . . . 6
  - 1.1.3 Untrusted setups . . . 7

2 Literature review **8**
- 2.1 Bit-commitment schemes . . . 8
  - 2.1.1 Commutative cryptography . . . 8
  - 2.1.2 Bit-commitment with one-way functions . . . 8
- 2.2 Zero-knowledge proofs . . . 9
  - 2.2.1 Games as graphs . . . 10
  - 2.2.2 Graphs & zero-knowledge proofs . . . 11
  - 2.2.3 Cheating with negative values . . . 11
- 2.3 Additive homomorphic cryptosystems . . . 12
  - 2.3.1 Paillier cryptosystem . . . 12
  - 2.3.2 Zero-knowledge proofs in Paillier cryptosystem . . . 13
  - 2.3.3 Reducing communication . . . 13

3 Implementation **14**
- 3.1 Message structure . . . 16
- 3.2 Paillier cryptosystem . . . 16
  - 3.2.1 Modular exponentiation . . . 16
  - 3.2.2 Public key . . . 17
  - 3.2.3 Encryption . . . 18
  - 3.2.4 Private key . . . 18
  - 3.2.5 Decryption . . . 19
  - 3.2.6 Implementation details . . . 19
- 3.3 Shared random values . . . 19
  - 3.3.1 Modular bias . . . 20
  - 3.3.2 Application to domain . . . 20
- 3.4 Proof system . . . 21
  - 3.4.1 Proof of zero . . . 21
  - 3.4.2 Proving reinforcement . . . 22
  - 3.4.3 Range proof . . . 24
  - 3.4.4 Proving fortifications . . . 25
  - 3.4.5 Optimising . . . 26

3.4.6 Application to domain 27

4 Discussion **29**
4.1 Theoretic considerations 29
4.1.1 Random oracles 29
4.1.2 Quantum resistance 29
4.1.3 Honest-verifier 29
4.2 Security 30
4.2.1 Soundness 30
4.2.2 Collusion 30
4.3 Efficiency 30
4.3.1 Storage complexity 30
4.3.2 Time complexity 31
4.3.3 Complexity results 32

5 Conclusions **34**
5.1 Contributions 34
5.2 Domain 34
5.3 Wider application 35
5.3.1 Larger scale games 35
5.3.2 Decentralised social media 35
5.3.3 Handling of confidential data 35
5.4 Limitations encountered 36
5.4.1 JavaScript 36
5.4.2 Resources 36

Bibliography **38**

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
$\phi(k)$	Euler's totient function
$\lambda(k)$	Carmichael's totient function
$H(\dots)$	Ideal cryptographic hash function
$\in_R$	Selection at random
$A \parallel B$	Concatenation of A and B

"Never create anything, it will be misinterpreted, it will chain you and follow you for the rest of your life." - Hunter S. Thompson

Artefact available at `https://gitea.jellypro.xyz/jude/Riskless`

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, which is available 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 create rules for the platform through a EULA, and this would be 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.

However, centralised services have a number of major downsides.

- **User freedom.** Users often cannot audit or modify the underlying source code. Furthermore, users must follow rules enforced by the platform, which can be arbitrarily set and enforced.

- **Liability.** In some jurisdictions, platforms are held legally responsible for content on their platforms. This means that consequences of illegal activities are sometimes felt by the platform and not the criminal user, and this encourages platforms to invade the privacy of their users.
- **Ownership.** The service is owned by the operators. If the operators wish to close the service, it is at their own discretion.

1.1.2 Peer-to-peer networks

There are two alternatives to traditional centralised networks: peer-to-peer (P2P) and federated.

In P2P networks, traffic may be routed directly to other peers. In federated networks, servers may be operated by third parties (and in fact, the developers of the service may not run any servers themselves). These network models are still popular in certain games or services, for example BitTorrent is primarily a P2P service; and titles from the Counter-Strike video game series may be considered federated, with a wide selection of third party hosts.

P2P and federated networks address each of the disadvantages listed above.

- **User freedom.** The platform is run by its users. Whilst this doesn't require that the source code is available, it allows the auditing of data collected, and the network may diverge to meet the needs of different groups of users. For example, Bitcoin Cash is a fork from Bitcoin intended to address concerns over transaction fees [6].
- **Liability.** Users are legally responsible for their own behaviours. This results in legal consequence against criminal users, and relative immunity of the platform [42].
- **Ownership.** Games such as Unreal Tournament 99 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 officially supporting the title) [10, 12].

However, general privacy can often be worse in fully P2P networks than that of fully centralised networks. As there is no trusted server, there is no easy way to obscure the traffic of each user or to maintain, validate, and operate on private data. For example, most popular cryptocurrencies (such as Bitcoin and Ethereum) are public-ledger, meaning transactions can be publicly tracked. This is far less private than services such as banks or cash [28, 3].

Some P2P services try to address issues with privacy. Monero and Zcash are privacy coins that use cryptographic protocols to obscure transaction amounts [22, 8]. The P2P file-sharing protocol Soulseek keeps a user's download history private by using direct communication between the two peers [29]. The downside of this approach is that if the first user goes offline, files will no longer be available. BitTorrent overcomes this by pooling peers with the file into a "seed pool". The disadvantage of this approach is that the users who download files is now public knowledge [7].

1.1.3 Untrusted setups

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

We aim to apply bit-commitment schemes, zero-knowledge proofs, and homomorphic encryption to an online P2P variant of Risk. This will 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: this introduces private state to be operated on.

Chapter 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, another approach must be taken to simulate the randomness, to ensure all peers receive the same random value.

For such randomness, we also 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.

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.

2.1.1 Commutative cryptography

Protocols exist that utilise bit-commitment to play poker [37]. 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 a well-known one-way function, such as SHA [30], with randomly generated salts.

2.1.2 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 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. This is a standard application of bit-commitment to form a "secret sharing" protocol.

2.2 Zero-knowledge proofs

Informally, zero-knowledge proofs can be considered to be protocols between two parties (the prover and the verifier) that operate on some given statement. The protocol holds the following three properties:

- **Completeness.** If the statement is true, an honest verifier will be convinced of its truth by a prover.
- **Soundness.** If the statement is false, a cheating prover cannot convince an honest verifier (except with some small probability).
- **Zero-knowledge.** If the statement is true, the verifier cannot learn any other information besides its truthfulness.

Typically, this protocol will involve the verifier producing a set of challenges, which the prover will respond to.

Formally, 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.

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

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.

Honest-verifier zero-knowledge is a subset of general zero-knowledge, in which 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 behave as to try and attain additional information.

One solution to this is to transform a proof into a non-interactive zero-knowledge proof. The Fiat-Shamir transformation [14] 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. However, this relies strongly upon the random-oracle model [33]. 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.

2.2.1 Games as graphs

Risk's board layout 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 to identify ways to secure this protocol by obscuring the edges and weights, whilst preventing the ability for the player to cheat.

2.2.2 Graphs & zero-knowledge proofs

A typical example for zero-knowledge proofs is graph isomorphism [16].

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.

2.2.3 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 need 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).

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". This can be achieved by using range proofs.

The BCDG range proof proves that a commitment to a plaintext in the interval $[0, \ell]$ lies within the interval $[-\ell, 2\ell]$, where ℓ is some well-known value [5, Section 2].

The distinction between the soundness and completeness intervals in the BCDG proof is important, as 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 may be able to infer a stronger upper or lower bound on m. This is a major downside to this protocol.

The state of the art in range proofs is Bulletproofs [18]. Bulletproofs are utilised by the Monero blockchain ledger, and use linear algebraic properties to allow multiple verifying parties to process a single prover's proof.

Bulletproofs have advantages in size and "batch verification", where a verifier can verify multiple proofs simultaneously. However, the proofs are very complex, and a multi-round approach that borrows some of the concepts used in Bulletproofs could be used instead.

In general, this approach uses a decomposition of the plaintext message m into its bits. This allows a verifying party to reconstruct an encryption of m, and check the bit length,

without discovering m [4, Section 1.2.1].

2.3 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.

2.3.1 Paillier cryptosystem

The Paillier cryptosystem, which is based on composite residuosity classes, express the additive homomorphic property [31]. 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 blinding value $r \in \mathbb{Z}_{n^2}^*$.

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. This is because 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 (although this depends heavily on the key size). Many optimisations to the Paillier cryptosystem have been presented in literature.

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 $g^m = 1 + mn$ directly from the binomial theorem.

Another optimisation is that of Jurik [21, 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.

2.3.2 Zero-knowledge proofs in Paillier cryptosystem

There exist honest-verifier zero-knowledge proofs for proving a given value is 0 [9, Section 5.2]. Hence, proving a summation $a+b=v$ can be performed by proving $v-a-b=0$, which is possible by the additive homomorphic property.

Therefore, using Paillier's scheme to obscure edge weights should enable verification of the edge values without revealing their actual values.

2.3.3 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 [17]. 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 [14], as discussed above, is another way to reduce communication by using a random oracle. For ledgers, non-interactive zero-knowledge proofs are necessary, as the ledger must be resilient to a user going offline. In our case, users do not go offline. However, non-interactive zero-knowledge proofs are still beneficial as the amount of communications can be reduced significantly, resulting in simpler network code.

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 3

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.

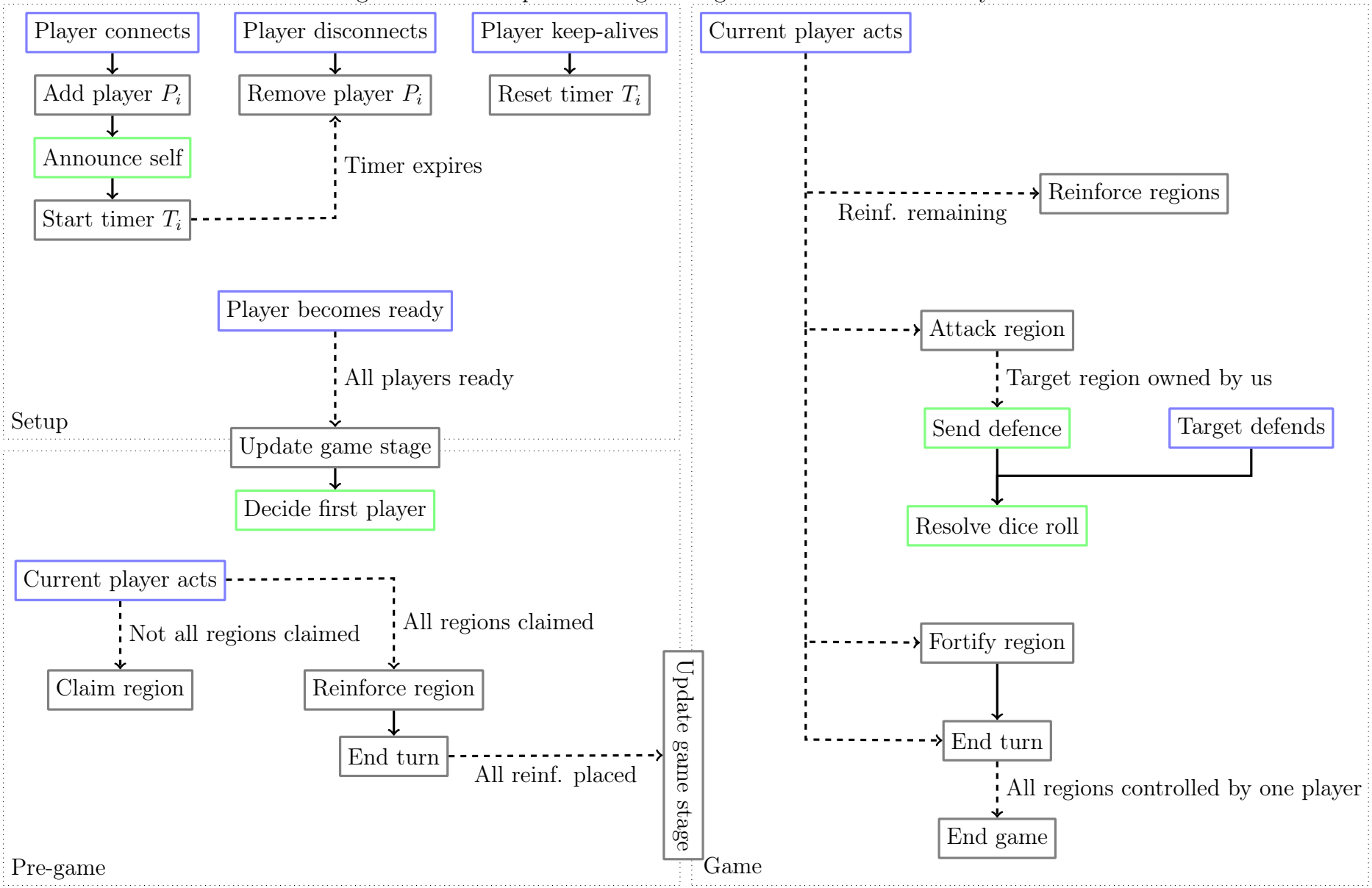
Despite this approach being centralised, it does emulate a fully P2P environment, and has notable benefits:

- There is no need for NAT 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. Boxes in blue are messages received from other players (or transmitted by ourselves). Boxes in green require us to transmit a message to complete.

Figure 3.1: Decomposition of general game structure as a P2P system



3.1 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`, `ACT`, and `RESOLVE`. 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` and `RESOLVE` are used by players to submit actions for their turn during gameplay, and to resolve the outcomes of these actions.

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 [39]. 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.

3.2 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" [40]. 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 [40]. In particular, modular exponentiation is non-constant time. However, as decryption is not performed during the program's runtime, it is unlikely that an attacker could use this to execute a timing attack against another player.

3.2.1 Modular exponentiation

As the implementation of `BigInt`s in V8 does not optimise modular exponentiation itself, we employ the use of addition chaining [35]. 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 b, between $|b|$ and $2|b|$ multiply-and-modulo operations are performed.

In the case of a fixed base, further speedup can be gained through pre-computation of fixed base powers. By pre-computing powers of the powers of two, exponentiation is reduced to at most L multiplications. For some fixed base h and modulus n, let $h[i] = h^{(2^i)} \bmod n$ represent cached values. 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

3.2.2 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 $0 < m < n$.

Generating primes is a basic application of the Rabin-Miller primality test [34]. 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.

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 3.2.1. *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 3.2.2. $1 + n \in \mathbb{Z}^*_{n^2}$.

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 performed, this selection of g also does not require auxiliary memory to store intermediary values using during exponentiation.

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. Using safe primes guarantees this. Safe primes are primes of form $2p+1$ for p prime.

Proposition 3.2.3. *For $p > 5$ a safe prime, $p \equiv 3 \mod 4$*

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

$$q \equiv 1 \mod 4 \implies 2q + 1 \equiv 3 \mod 4$$
$$q \equiv 3 \mod 4 \implies 2q + 1 \equiv 3 \mod 4$$

as required. □

Proposition 3.2.4. *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).

3.2.3 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 $E'(m, r) = E(m, h^r \bmod n)$.

The main speedup as a result of using Jurik's form originates from fixed base exponentiation, as discussed in Section 3.2.1.

3.2.4 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.

3.2.5 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, decryption is not used in the application, and is only useful as a debugging measure. So this optimisation is not applied.

3.2.6 Implementation details

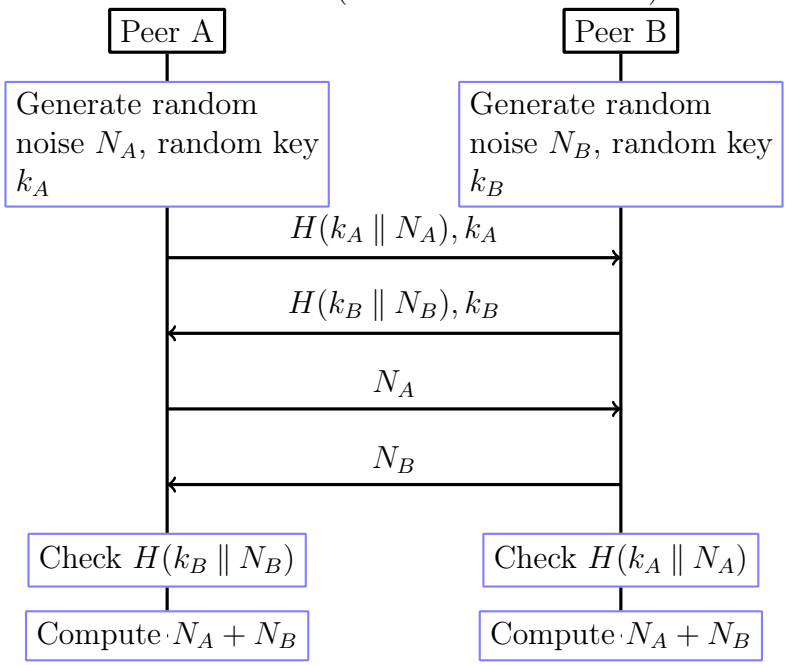
Paillier is implemented by four classes: `PubKey`, `PrivKey`, `Ciphertext`, and `ReadOnlyCiphertext`. `PubKey.encrypt` converts a `BigInt` into either a `Ciphertext` or a `ReadOnlyCiphertext` by the encryption function above. The distinction between these is that a `ReadOnlyCiphertext` does not know 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, which is used later in Protocol 3.4.1.

3.3 Shared random values

A large part of Risk involves random behaviour dictated by rolling some number of dice. To achieve this, a 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 with a standard application of bit-commitment, and properties of $\mathbb{Z}_n$. The protocol for two peers is as follows, and generalises to n peers.

Protocol 3.3.1 (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 moved into the required 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, and compute the final value as $N_A \oplus N_B$.

The hash function used must also be resistant to length-extension attacks for the presented protocol. In general, a hash-based message authentication code can be used.

Proposition 3.3.2. *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 $H(k_i \parallel N_i)$. The cheater P_n commits $H(k_n \parallel N_n)$.

The hash function H holds the preimage resistance property: that is, P_i cannot find m given $H(m)$. 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 regardless of the choice of N_n. As P_n cannot reasonably find a collision for $H(k_n \parallel N_n)$, P_n must reveal 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.

3.3.1 Modular bias

Despite restricting each player's random noise to a fixed bit length, we must still avoid the modular bias that occurs when taking the modulus of a bit sequence.

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

3.3.2 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. A benefit of this is that the unique name can be used with the Web Locks API to prevent race conditions that may occur due to this protocol running asynchronously.

To achieve bit-commitment, we use SHA-3 [30], as implemented by jsSHA [41]. SHA-3 is resistant to length-extension attacks, and is considered secure, so it is reasonable to assume that a malicious player will not be able to find a collision.

3.4 Proof system

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

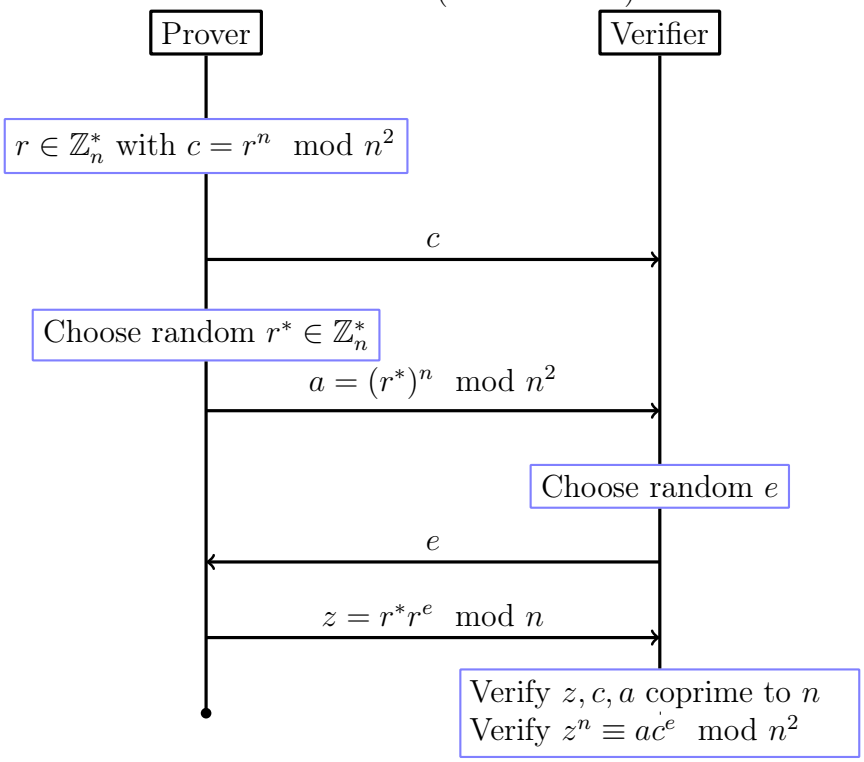
1. The number of reinforcements placed during the first stage of a turn.
2. The number of units on a region neighbouring another player.
3. The number of units available for an attack/defence.
4. The number of units lost during an attack/defence (including total depletion of units and loss of the region).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

These points are referenced in the following sections.

3.4.1 Proof of zero

The first proof to discuss is the honest-verifier protocol to prove knowledge that a ciphertext is an encryption of zero [9, Section 5.2].

Protocol 3.4.1 (Proof 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 $cg^{-m} \equiv r^n \mod n^2$ 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. It is also used throughout the other proofs presented.

3.4.2 Proving reinforcement

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.

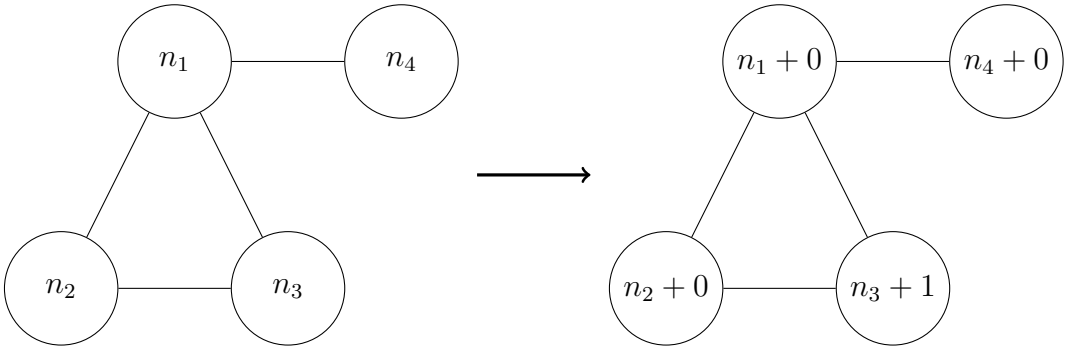


Figure 3.2: Example state change from reinforce action.

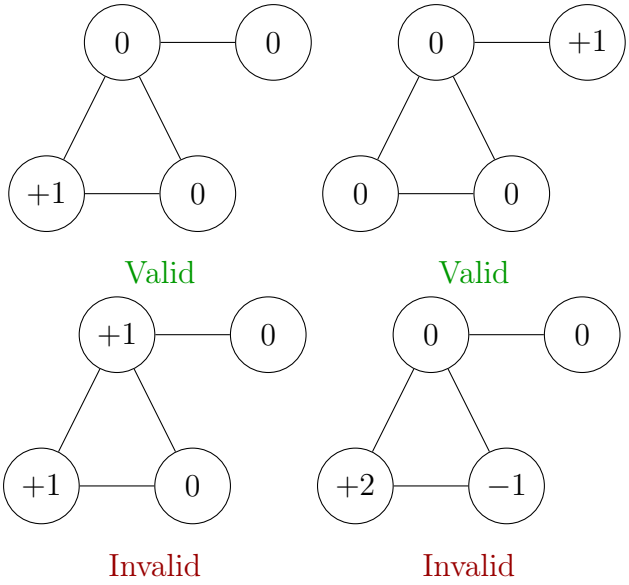


Figure 3.3: Valid and invalid reinforce messages. Notably, the final invalid message would not be caught by the additive homomorphic check.

To prove this, the prover will demonstrate that a bijection exists between the elements in the reinforcement set and a "good" set.

Protocol 3.4.2. 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 picks $c \in_R \{0, 1\}$.
   (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 [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, this protocol is perfectly simulatable.

Proposition 3.4.3. *Protocol 3.4.2 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 [9, 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. □

In practice, as we are using Jurik's form of Paillier, this 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.

3.4.3 Range proof

The range proof we use proves an upper bound on $|m|$ by constructing commitments to the bits of m, and using homomorphic properties [4, Section 1.2.1]. Given valid encryptions of a sequence of bits, we can reconstruct the number: let $b_1, \ldots, b_{|m|}$ be the bits of m (b_1 being the LSB), and $c_i = E(b_i) = g^{b_i} r^n \mod n^2$. Then, we can construct an encryption of m as

$$E(m) = (c_1)^{(2^0)} \cdot (c_2)^{(2^1)} \cdot (c_3)^{(2^2)} \cdot \ldots \cdot (c_{|m|})^{(2^{|m|-1})}.$$

Validating $E(m)$ is done with the proof of zero. Then it remains to prove that each c_i enciphers either a 0 or a 1. This can be done in a similar way to Protocol 3.4.2.

Protocol 3.4.4. The prover transmits $c = E(m)$, and encryptions of the bits $c_1, \ldots, c_{|m|}$, except using -1 in place of 1.

1. The verifier computes

$$c \cdot \prod_i^{|m|} (c_i)^{(2^{i-1})}$$

   and requests a proof of zero.

2. Perform t times for each i:
   (a) Prover transmits $S = \{E(0), E(1)\}$.
   (b) Verifier picks $a \in_R \{0, 1\}$.
      i. If $a = 0$, the verifier requests a proof that $S = \{0, 1\}$.
      ii. If $a = 1$, the verifier requests a proof that $c_i \cdot s_i$ is zero for one of the $s_i \in S$.

The downside of this proof over the BCDG proof [5] is that the time to perform and verify this proof grows linearly with $|m|$. However, in most cases $|m|$ should be "small": i.e, $|m| \leq 6$, as Risk unit counts rarely exceed 64 on a single region. In fact, to prevent revealing additional information from the bit length, this protocol in practice will pad out numbers to a minimum of 8 bits, which in our application makes this protocol constant time.

This proof is still sound, complete, and supposing sufficient padding is used, zero-knowledge. The soundness depends on the number of rounds t performed. Cheating this protocol requires that the prover transmits some $c_j \notin \{0, 1\}$ such that the product computed in step 1 is still zero. Then, there is a $\frac{1}{2}$ chance that, when testing c_j in step 2, the challenge selected by the verifier will be failed. This gives a probability of cheating as 2^{-t} as before.

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. All of these are performed using Protocol 3.4.4 and by using the additive homomorphic property to subtract the lower range from m first.

3.4.4 Proving fortifications

Point (5) still remains, as the range proof alone only works to prevent negative values from appearing in the resolution of 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.

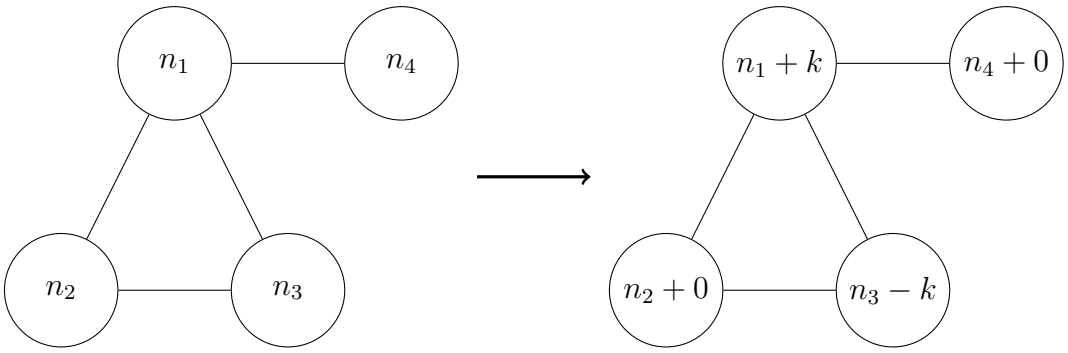


Figure 3.4: Example state change from fortify action.

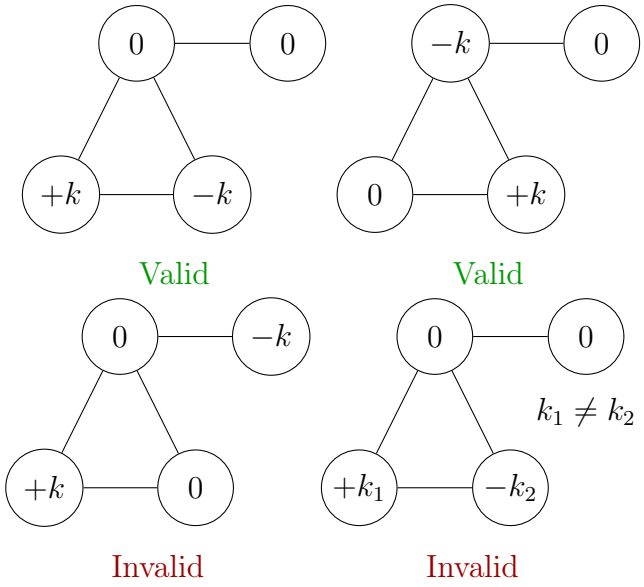


Figure 3.5: Valid and invalid fortify messages.

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

Protocol 3.4.5. 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 $a \in_R \{0, 1\}$.
   (a) If $a = 0$, 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 $a = 1$, the verifier requests proofs that $|S| - 2$ elements 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.

3.4.5 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 proofs, then the verifier computes all of their challenges. However, the issue remains of performing the proofs of zero.

We can apply the Fiat-Shamir heuristic to make proofs of zero non-interactive [14]. 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 [30]. The library jsSHA [41] provides an implementation of SHAKE256 that works within a browser.

We can then apply the Fiat-Shamir heuristic to each higher-level protocol to make them non-interactive. The proving party first generates t proofs (this can be done independently of verification) and serialises them to JSON. Then, use SHA-3 to compute a t-bit hash of the serialised proofs, and use each bit of this hash as the verifier's challenge. It is assumed that the prover cannot compute preimages of SHA-3, so they cannot control which challenges will be "requested". This makes it computationally infeasible for the prover to cheat.

3.4.6 Application to domain

Finally, the following diagram shows how each protocol presented ties into the domain. We highlight the interactions between two particular players: Player 1 is the current player, and Player 2 controls a region neighbouring a region of Player 1.

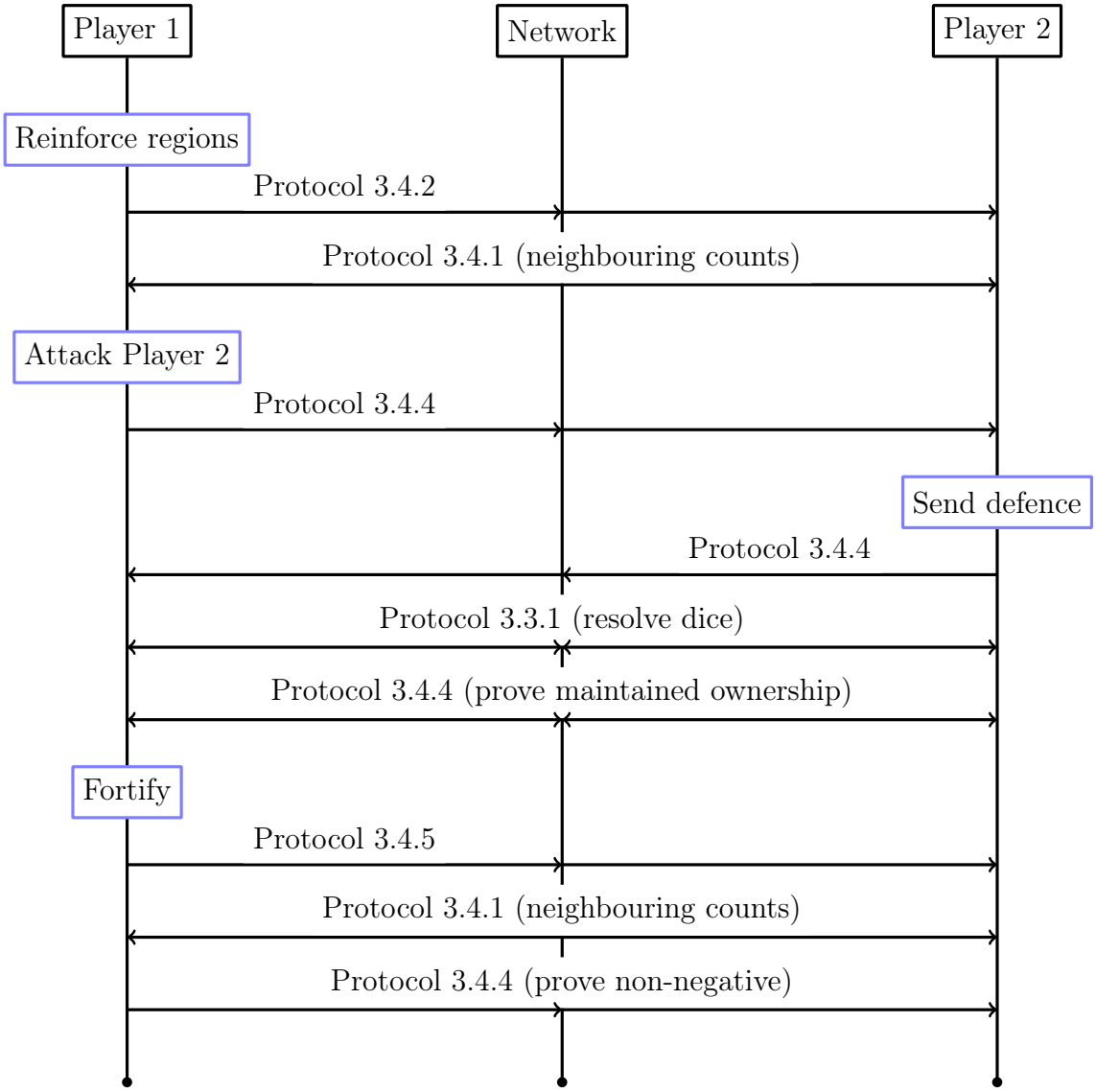


Figure 3.6: An example turn in our P2P implementation of Risk.

Chapter 4

Discussion

4.1 Theoretic considerations

4.1.1 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 thesis, an algorithm with infinite description cannot be computed on a finite machine.

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 [23].

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. This property is necessary when constructing non-interactive proofs, to prevent a prover manipulating the signature used to derive the proof.

4.1.2 Quantum resistance

Paillier is broken if factoring large numbers is computationally feasible [31, Theorem 9]. Therefore, it is vulnerable to the same quantum threat as RSA is, known as Shor's algorithm [38]. Alternative homomorphic encryption schemes are available, which are believed to be quantum-resistant, as they are based on lattice methods (e.g, [15]).

4.1.3 Honest-verifier

The proof of zero is honest-verifier [9, Section 5.2]. However, applying the Fiat-Shamir heuristic converts such a proof into a general zero-knowledge proof [14, Section 5]. This means that, supposing the choice of transform used is appropriate, Protocol 3.4.2 should also be general zero-knowledge. However, the interactive proofs performed as part of

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

4.2 Security

4.2.1 Soundness

Assuming $t = 24$, the chance of an undetected cheater in a single execution of a multi-round protocol is $2^{-24} \approx 6.0 \times 10^{-8}$.

It is possible that even if a prover cheats a proof at one point, the cheat would be detected later on in the game. For example, suppose a player cheated the range proof during an attack, and as a result won the attack. This instance of cheating is likely to be detected in the final range proofs to prove that regions are non-negative in value at the end of a turn.

We previously discussed the soundness issues relating to the BCDG proof [5]. These issues are overcome by Protocol 3.4.4, which instead aims to fix an upper bound on the bit length of a value, rather than prove a value falls within a specific range.

From the additive homomorphic property, this proof can be easily manipulated to cover other ranges without affecting the soundness: for example, to prove a value v is in the range $1 \leq v \leq 256$, each party first multiplies the ciphertext by $g^{-1} \mod n^2$, and then proceeds with the proof as normal.

4.2.2 Collusion

Assuming n players, we discussed that Protocol 3.3.1 is resistant to $n-1$ colluding parties. Similarly, as the Fiat-Shamir heuristic is used for most proofs, colluding parties cannot agree beforehand to use specific challenges, which would allow cheating of proofs.

The only instance of a zero-knowledge proof that doesn't use Fiat-Shamir is the proof of neighbouring values. However, this proof is not important to the integrity of the game, as all state-changing actions are verified by the other non-interactive multi-round proofs.

Colluding players could agree to not verify each other's proofs. However, this would just result in the colluding players' games diverging from the games of the players who are playing fairly, effectively ending the game session for the non-colluding players.

4.3 Efficiency

4.3.1 Storage complexity

For this section, 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 within today's memory and network limitations.

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 need 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 3.4.2 requires multiple rounds. Assume that we use 24 rounds, and additionally assume that there are five regions to verify. Each prover round then requires five Paillier ciphertexts, and each verifier round requires five non-interactive proofs of zero plus some negligible amount of additional storage for the bijection. This results in a proof size of $(10|n| + 10|n|) \times 24 = 480|n|$. For key size $|n| = 2048$, this is 120kB. This is a reasonable size for memory and network, but risks exceeding 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 decreasing the soundness. 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 12 will increase the chance of cheating to $\frac{1}{4096}$, but the size would reduce to approximately half.

Each of these calculations is in an ideal situation without compression or signatures. In the implementation presented, the serialisation of a ciphertext is larger than this for two main reasons. First, each value serialises to a string of its hexadecimal representation, and secondly each message 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 the actual sizes of each proof are given in Table 4.3. 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 [32], 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, this is 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 [27], 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 shorter key will reduce the size of the ciphertexts linearly.

4.3.2 Time complexity

Theoretic timing results versus RSA are backed experimentally by the implementation. Performing 250 Paillier encrypts required 47,000ms. On the other hand, performing 250 RSA encrypts required just 40ms. Results are shown in Table 4.1.

Potential further optimisations to the implementation are considered below.

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 concern 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 considerably reduces the time complexity [31].

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 [36].

Protocol 3.4.2 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.

However, the other proofs do not translate so trivially to this structure. In fact, in some contexts the proofs required may be considerably more complicated, 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 [20].

Another approach is to use a web extension to communicate with a system daemon providing the relevant functionality. This is language-agnostic (except that the extension itself must be JavaScript), and the daemon could take advantage of other system features such as multiple cores. The multi-round proofs in particular are embarrassingly parallel, as each round is independent of the other rounds.

4.3.3 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 4.1: Time to encrypt

Modulus	Paillier encrypt	Jurik encrypt	Jurik encrypt with pre-computation	RSA encrypt
$\|n\| = 1024$	6ms	4ms	1.4ms	0.015ms
$\|n\| = 2048$	34ms	22ms	7.6ms	0.040ms
$\|n\| = 4096$	190ms	130ms	–	0.093ms

Table 4.2: Time[a] to process non-interactive proofs

Modulus	Protocol 3.4.1		Protocol 3.4.2 with $t = 24$		BCDG Range with $t = 24$		Protocol 3.4.4 with $t = 24$		Protocol 3.4.5 with $t = 24$	
	Prover	Verifier	Prover	Verifier	Prover	Verifier	Prover	Verifier	Prover	Verifier
$\|n\| = 1024$	10ms	18ms	1,420ms	2,140ms	443ms	655ms	3,530ms	5,310ms	1,350ms	2,070ms
$\|n\| = 2048$	44ms	68ms	6,390ms	8,140ms	1,980ms	2,400ms	15,800ms	19,000ms	5,800ms	7,790ms
$\|n\| = 4096$	225ms	292ms	41,500ms	34,400ms	14,300ms	11,400ms	112,000ms	79,300ms	40,500ms	29,100ms

[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 4.3: Byte size[b] of encoded non-interactive proofs

Modulus	Protocol 3.4.1		Protocol 3.4.2 with $t = 24$		BCDG Range with $t = 24$		Protocol 3.4.4 with $t = 24$		Protocol 3.4.5 with $t = 24$	
	JSON	with LZ-String	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	123,354B	34,857B	895,474B	248,420B	322,946B	92,042B
$\|n\| = 2048$	3,153B	1,050B	662,233B	187,333B	252,230B	70,868B	1,746,017B	485,787B	620,206B	176,854B
$\|n\| = 4096$	6,226B	1,999B	1,315,027B	368,646B	484,117B	135,990B	3,458,376B	964,913B	1,206,657B	341,028B

[b] 1 UTF-16 character, as used by ECMAScript [11, Section 6.1.4], is 2 or more bytes.

Chapter 5

Conclusions

5.1 Contributions

This project has contributed an implementation of an optimised form of Paillier that is compatible with modern web browsers. Benchmarks show that, considering current hardware, Paillier in Jurik's form can be a viable cryptosystem for occasional use. However, additional work is required to make it efficient enough for large amounts of encryptions, as seen in Protocol 3.4.4.

The Paillier implementation provides capability for Schnorr-style proofs of knowledge and also multi-round proofs of knowledge, which serialise to JSON. These are made non-interactive by applying the SHAKE cryptographic hash suite.

Multi-round proofs combining set membership and graph isomorphism are among the implementations, and have strong zero-knowledge properties once used with the Fiat-Shamir transform.

5.2 Domain

The protocols devised are effective in the target domain of online games. With multi-round proofs using 24 rounds, players can be reasonably confident that other players are not cheating.

For the most part, the proposed protocols run in a time-frame that would not disrupt the experience, with the exception of the bit length proof. With additional work, this proof could be replaced with a Bulletproof [18], which may use less bandwidth and perform faster.

A large outstanding problem with the implementation is conflict resolution. Currently, if a player submits proofs that do not verify, other players simply ignore the message. However, a better solution would be to allow other players to remove a misbehaving player from the protocol.

5.3 Wider application

P2P software solutions have many benefits to end users: mainly being greater user freedom. The content presented here shows clear ways to extend P2P infrastructure, and reduce dependence on centralised services.

We propose some further ideas which could build off the content here.

5.3.1 Larger scale games

Many other games exist that the ideas presented could be applied to. Games of larger scale with a similar structure, such as Unciv [25], could benefit from P2P networking implemented similarly. In particular, similar protocols to Protocol 3.4.4 would form an intrinsic part of such games, as they have a similar graph structure which requires guarantees of adjacency for many actions.

The downsides of this are that the complexity of P2P networking is far greater than in a centralised model. This would be a considerable burden on the developers, and could hurt the performance of such a game. Additionally, some modern routers no longer support NAT hole-punching or UPnP due to security concerns [13], which makes accessing P2P services more difficult for end users.

5.3.2 Decentralised social media

The schemes presented here could be applied 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 [19]. This poses an advantage that users can store their own data, 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.

Decentralised platforms promote user privacy, as users can control their own data. Additionally, decentralised platforms promote standardisation of common operations such as instant messaging. This can include end-to-end encryption, and so confidentiality is then a choice of the user rather than the platform. Furthermore, the consequences of security issues in individual configurations or legislation targetting platforms is reduced.

Some P2P messaging standards already coexist that could be used here, for example Matrix and XMPP [24, 43].

5.3.3 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. To allow a second party to process data, some guarantee of the correctness of the data is required.

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.

5.4 Limitations encountered

5.4.1 JavaScript

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

JavaScript, in its most common implementations, is a slow language for number processing. Prime generation takes a considerable amount of time, and this extends to encryption being slower than in an implementation in an optimising compiled language.

Table 5.1: Time to generate safe primes

	Our implementation	`openssl dhparam 512`
$\|p\| = 512$	8,660ms	66ms

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

JavaScript is an asynchronous, but single-threaded language: this means that the interpreter uses an event loop to handle new events [26]. This introduces the possibility of race conditions despite no explicit threading being used. The asynchronous nature is 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. Using a language with explicit threading would allow for speed up in prime generation and proof construction, as these can be parallelised trivially.

Using a language that can interact with the operating system would also have advantages, as key generation can be performed by standard tools such as OpenSSL and stored in the system keychain, and platform features such as SIMD could be utilised for parallelism.

5.4.2 Resources

The P2P implementation requires more processing power and more bandwidth on each peer than a client-server implementation would. This is the main limitation of the P2P 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 computers in use today. Using greater processing power increases power consumption,

which is undesirable. In a client-server implementation, the power consumption should be lower than the P2P 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.

Final word count: 9,355

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] John Bohannon. Why criminals can't hide behind Bitcoin. 03 2016.

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

[5] 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.

[6] Bitcoin Cash. Bitcoin Cash – Peer-to-peer Electronic Cash. `https://bitcoincash.org`.

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

[8] Electric Coin Company. Zcash Basics – Zcash Documentation. `https://zcash.readthedocs.io/en/latest/rtd_pages/basics.html`.

[9] 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.

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

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

[12] Epic Games. Epic is turning off online services for some older games. https://www.epicgames.com/site/en-US/news/epic-is-turning-off-online-services-and-servers-for-some-older-games.

[13] Shadi Esnaashari, Ian Welch, and Peter Komisarczuk. Determining home users' vulnerability to universal plug and play (upnp) attacks. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 725–729, 2013.

[14] 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.

[15] 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.

[16] 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.

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

[18] Jahid Hasan and Minghai Xu. *Bulletproofs: A Non-Interactive Zero Knowledge Proof Protocol For Blockchain Security.* PhD thesis, 06 2020.

[19] IPFS. Ipfs specifications. `https://github.com/ipfs/specs`, 2023.

[20] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

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

[22] Monero Research Lab. What is Monero (XMR)?

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

[24] Matrix Spec Core Team. Matrix Specification. `https://spec.matrix.org/latest`.

[25] Yair Morgenstern. Unciv - Civ V remake for Android & Desktop. `https://github.com/yairm210/Unciv`, 2023.

[26] Mozilla. JavaScript language overview. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Overview`.

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

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 10 2008.

[29] "Nir". Frequently asked questions | soulseek. `http://www.soulseekqt.net/news/faq-page#t10n606`.

[30] 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.

[31] 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.

[32] Pieroxy. lz-string: Lz-based compression algorithm for javascript. `https://github.com/pieroxy/lz-string`, 2013.

[33] 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.

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

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

[36] 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.

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

[38] 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.

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

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

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

[42] Christopher Williams. UK cops arrest six alleged BitTorrent music uploaders. *The Register*.

[43] XMPP. Xmpp specifications. `https://xmpp.org/extensions/`.