Building a Fair Online Casino on the Blockchain

Student Name: Adam Hodson
Supervisor Name: Dr Ioannis Ivrissimtzis
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—In this work, we build a decentralised, secure and trustless online casino platform by leveraging modern public blockchain networks and tooling. In doing so, we evaluate the technical viability of such a platform and evaluate its technical viability regarding its speed, scalability, and cost. We do this in the context of negative public opinion regarding online gambling, particularly pertaining to its trustworthiness. We implement two different popular casino games; roulette and dice on the blockchain alongside a secondary bank management program. These are accessed via our second deliverable; a lightweight, responsive web app with blockchain integration. Finally, we conclude that, while it is feasible to build a platform that is decentralised, trustless, and secure, it is a mere proof of concept due to its inability to scale in speed or throughput. However, given more mature or better-suited blockchain implementations we find it to be likely to scale significantly better.

Index Terms—Games, Centralization/decentralization

1 Introduction

THE applications of blockchain are now far beyond the scope of an ’electronic cash system’ proposed in Nakamoto’s [1] seminal paper on the blockchain presenting Bitcoin. Recent interest now lies in how classical, centralised systems can be moved to incorporate blockchain.

This work considers one of the practical applications of blockchain as a replacement for the status quo in the industry of internet-based chance games (iGaming).

In doing so, we aim to answer the following questions; ’How viable or practical is it to implement a blockchain-backed online casino?’ and ’To what degree can said implementation leverage the benefits offered by such technologies (i.e. provability, decentralisation, and security)?’.

1.1 Issues with Modern iGaming

Traditionally, there are two things preventing a casino (online or physical) from cheating the player: its reputation and government regulations. Modern casinos are moving their focus to online play due to player demand. This presents a challenge as it is much harder for an online casino to prove they are not cheating the player as their games run on centralised servers where the user will only ever see the outcome of the game.

The industry currently suffers from distrust due to this centralisation of games, with 45.5% of users complaining about ‘unfair software’, 59.1% complaining about not being paid properly and, 25.8% ‘not being paid at all’ [2].

These alarmingly high rates all point to the vendor scamming the user like in the case of non-payment or that users feel the vendor is scamming them. While we know this, in the vast majority of cases, is not true due to strict government regulation - the feelings of being cheated are not soothed by promises of auditing from independent bodies.

The UK Gambling Commission (UKGC) finds that 71% of people in the UK believe that gambling is not conducted fairly [3] between 2020 and 2021. Users prefer to see it to believe it. A blockchain-backed iGaming solution offers a complete solution to this issue as well as the non-payment issue. To explain how this works, we first cover some background on decentralised apps.

1.2 An Overview of Blockchain

First we must establish an understanding of blockchain as an idea. Fundamentally, a blockchain is a chronological, write-only ledger of transactions. In the simplest form, these transactions are simply moving arbitrary tokens from one wallet/user/address to another.

Transactions are grouped into ’blocks’ with a unique identifier in the form of a hash determined when adding the block to the network. Transactions in a block are stored in a Merkle tree [4], [5], [6]. Merkle trees have the ideal property wherein the hash of its root node will encode the data of all child nodes, all the way down to the leaves.

For a new block of transactions to be added to the chain, it must first be ’mined’. To mine a block; the miner (a node running the blockchain software) aims to fill in a nonce (number used once) of the given block, shown in Fig. 1. This is done by constantly incrementing said nonce by 1, starting from 0, placing it in the nonce slot of the block and hashing it. This resultant hash value is compared with the difficulty value set globally in the software run by all nodes. If the resultant value is smaller than the difficulty, then the block is mined.

Next, the network must reach a consensus that this solution is valid. The first proposed consensus protocol, and still the most widely used among blockchains, is Proof of Work (PoW) [7]. This protocol is named as such due to the high number of computations required to find a valid hash.

A miner with a valid solution will propagate their proposed solution throughout the network to avoid reliance on a trusted party [8]. Each node gets one vote on its valid-

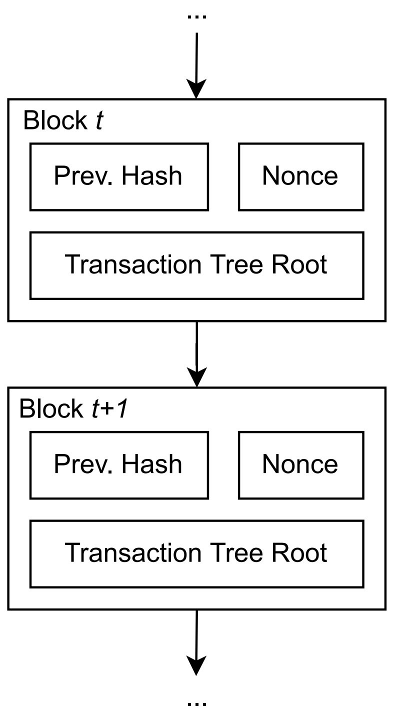

Fig. 1. The anatomy of a block, originally proposed for use in Bitcoin [1].

ity, checking for issues like unsigned transactions, double spending, or improper nonce proposals.

If a majority consensus of block validity is reached then all agreeing nodes with append this to the end of their chain and uses its hash in attempting to mine the next block. From this, it is obvious that the longest, validated chain on the network is the source of truth.

In the case of networks such as Bitcoin, the miner who first submitted a valid block for verification will be rewarded with a fixed amount of cryptocurrency.

Miners often pool their computing power due to the winner-takes-all nature of block rewards [9], [10]. Others simply buy/rent vast amounts of computing power to consistently beat out competing miners.

Other consensus protocols like Proof-of-Stake (PoS) attempt to address these issues by having miners (validators) stake some of their primary cryptocurrency of the network and, in exchange, will probabilistically mine (validate) blocks more often than those who have staked less. This eliminates the profit incentive driving the compute power arms race, which consumes huge amounts of energy every year [11].

1.3 Second Generation Blockchains

Second generation blockchains like Ethereum took the core idea of first-generation networks such as Bitcoin and introduced the concept of smart contracts [12] and the Ethereum Virtual Machine (EVM). These allow Ethereum transactions to do much more than simply move funds like Bitcoin. Understanding the core concepts of Ethereum is integral to understanding all DApps (decentralised applications) [13], including the work we present.

The first of these concepts; smart contracts, are arbitrary computer programs that anyone can deploy to the blockchain. They can be designed to store persistent data on the blockchain and accept transactions which can perform various calculations/operations.

Anyone can deploy a smart contract with their own code but after deployment its code is immutable and permanent. Also, once deployed, a contract is identified by a unique address like any other user of the blockchain. Using this address, anyone can interact with it on-chain, and read completely deterministic, validated results from it in a trustless decentralised network without a central point of failure [14].

The second of these concepts; the EVM, is a classic stack-based machine that operates the same as a Turing machine (see Fig. 2). However, this Turing machine has a very expensive tape, as writing values to the persistent state of the blockchain is rather expensive, as clarified later in Section 1.4. However, writing to the memory is very cheap as it is volatile and its state is not maintained between transactions.

An instance of the EVM runs on each node on the Ethereum network at all times, allowing the hot-swapping of smart contract states to process transactions and reach a consensus on the validity of a mined block (much the same as explained in Section 1.2).

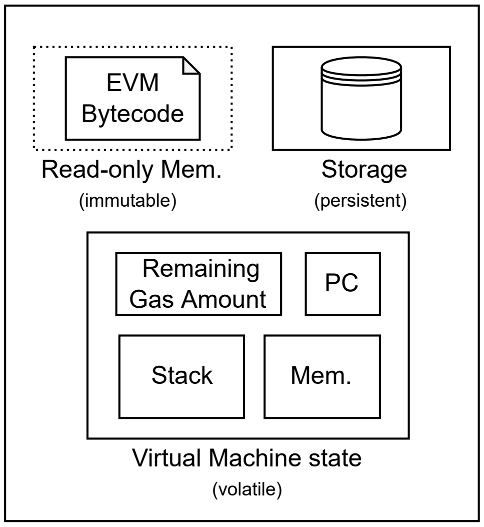

Fig. 2. The structure of the EVM (adapted from EVM illustrated [15]).

This powerful concept has countless possible applications that are just beginning to be explored by allowing developers to take advantage of the transparency, immutability, and cryptographic security explained in Section 1.2 to build trustworthy and arbitrary software.

To interact with the contract, a user can perform two types of transactions on it. Firstly there are 'view' interactions where the user wants to query the current state of the contract. These consume gas but the caller does not pay it. Next, there are write interactions where the transaction will change the state stored by the contract. These require the transaction to be signed with the invoker's private key to verify the identity of the caller. This allows contracts to store per-user information or change its behaviour depending on the caller.

1.4 Gas Fees and Transaction Speeds

All current smart contract-compatible blockchains rely on two things: the concept of gas and a primary (gas) token. For example, ETH is the primary token of the Ethereum network. Gas is a way of measuring how much computational power a particular operation will consume. This could be the transfer of a token from one address to another, deploying a smart contract, or interacting (view/write) with a smart contract, as covered earlier. The invoker of the transaction will pay a certain amount of the network's primary token to perform their transaction on the network. The amount paid is defined by the network fee schedule.

Before signing a transaction and sending it to the network a user will attach a gas limit which acts as an upper bound for how much gas they wish to use. If their transaction exceeds this limit, it will revert and the state of the network will not change. They also specify how much they are willing to pay for each unit of gas. For example, say a user sends θ ETH to their friend. This costs 21000 gas so they would lose $21000 * G_{fee} + \theta$ ETH from their wallet while their friend would gain just θ ETH once the transaction is validated. If G_{fee} was just 50 gwei (which is considered quite low) this would cost the user 0.00105 ETH ($\approx$3 USD as of April 2022) in gas fees.

Gas prices fluctuate wildly especially if you care about your transaction being completed in a timely manner. This price fluctuation is shown in Fig. 3. However, most modern user wallets will have a built-in gas estimator and a gas price tracker for reasonable speed.

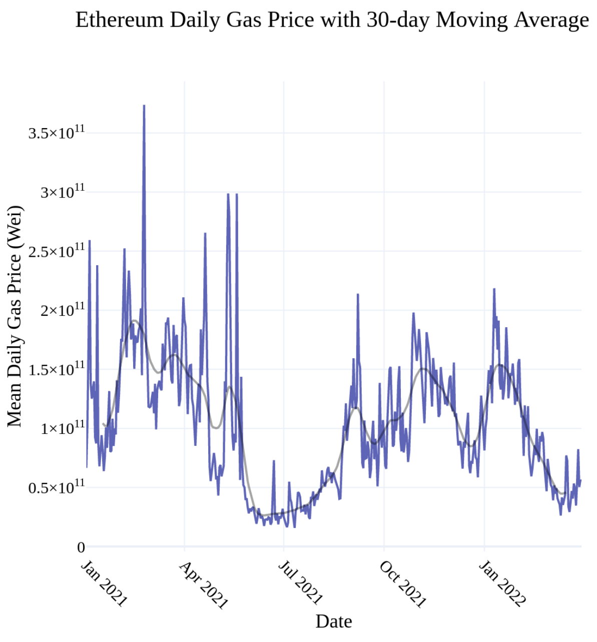

Fig. 3. Ethereum main network daily mean gas fee beginning in 2021 with a 30-day moving average (in wei). This highlights just how volatile unbounded gas prices can be even averaged over an entire 24 hours. Note that 1 wei $= 1 * 10^{-18}$ ETH.

Ethereum was not built with scalability in mind and constantly faces severe congestion causing low transaction throughput and high gas fees. These two issues exacerbate each other, as users will bid each other up on gas fees to have their transaction processed first due to the slow speed of the network, meaning that users are constantly sending new transactions, slowing the network down further.

As a result, Ethereum has an impractically low transaction rate of 15 TPS (transactions processed per second). Bitcoin, as a comparison, can only manage 6 TPS. For reference, a centralised payment provider like Visa can handle 24,000 TPS.

This severely limits the practical applications of Ethereum in real-time or near real-time applications like iGaming. The upcoming 2.0 update should help to alleviate these issues, but for now, many DApp developers are looking at alternatives. The DApp space is more diverse than its ever been in terms of the blockchains used with 30 active networks supporting DApps in 2021 compared to just 16 in 2020 [16].

1.5 Objectives

In this work, we build a DApp; GlassCasino. This platform allows users to play single and multiplayer casino games such as Roulette and a classic dice game; Chuck-a-Luck [17]. These games are ran entirely on a blockchain to leverage the numerous aforementioned benefits.

Our objectives for achieving this are as follows:

1) Implement two different traditional casino games as smart contracts of varying player counts
2) Integrate verifiable, decentralised randomness into at least one game
3) Build a robust migration chain to deploy our smart contracts to any compatible network
4) Construct a cloud-based, always-online client to operate the flow of games where necessary
5) Create a user-friendly, responsive web-based user interface (UI) to play and watch games take place
6) Add wide-scale standard wallet integration allowing users to play with just a few clicks
7) Include links to mined blocks and transaction records to allow auditing of games

1.6 Achievements

We largely achieved what we set out to do with a key few stipulations that arose during development. Our full achievements are as follows:

1) Implemented a multi-player, synchronised traditional roulette game with a small selection of bet types as a smart contract (objectives 1 and 3)
2) Implemented a single-player, verifiably random Chuck-a-Luck game as a smart contract (objectives 2 and 3)
3) Built out a recorded history of all games recorded on the blockchain via querying event logs (part of objectives 1 and 5)
4) Designed and built a central bank smart contract that works with any compatible deployed game (part of objective 1)
5) Integrated roles-based permissions to allow for the introduction of new games during development, and role renunciation to enforce trustlessness by

removing a central point of failure (part of objectives 1 and 3)

6) Created a user-friendly, responsive web-based user interface with full wallet integration and complete state management allowing for quick switching between different games (objectives 5 and 6)
7) Construction of a cloud-based, always-online client to operate the roulette wheel and synchronise low importance game timing data with the web app (objective 4)
8) Included links to mined blocks and transaction records on a standard block explorer website (objective 7)

Some key achievements to highlight are numbers 4, 5, and 6.

Achievements 4 and 5 were the result of a challenge that presented itself when developing our second game. This was the concept of banking in the context of a casino. All casinos run a simple banking system in the form of chips. The key tenet of this is they can have more chips on the floor of the casino than they have actual money in their vaults.

This means they can pay out any wager in chips even if they haven't made a lot of profit recently and simply tell the player to redeem their chips for cash later. The implemented solution to this is presented in Section 3.6.

Achievement 6 involved working with a large technology stack and combining a lot of moving pieces together; smart contracts status, wallet integration, connection to the house server, state management, and responsive design.

2 Related Work

The following sections will now introduce the technologies and theories that make this work possible. This includes modern blockchain architectures, verifiable randomness, caveats with other proposed solutions, and other work similar to ours.

2.1 Plasma & Polygon

Plasma [18] is a framework for building level 2 blockchain scaling solutions (L2). In essence, a level 2 chain runs on top of an existing blockchain (referred to as the ’rootchain’) as shown in Fig. 4. This means that one does not need to create transactions on the rootchain for every single transaction on the Plasma chain.

All this is to say, a Plasma-based blockchain like Polygon will have much lower gas fees while maintaining the same security and decentralisation benefits as state updates are validated in bulk by the root network (in this case, Ethereum). Second, the transaction bandwidth will be considerably higher as minimal data is passed down to the overloaded Ethereum network.

It does not disclose the contents of the Plasma chain to the rootchain; instead, only block hashes are submitted by Plasma validators utilising proof of stake consensus. If the block cannot be validated by the Ethereum network if, for example, the level 2 network was compromised and invalid blocks were being added, then the block on the Plasma chain will be rolled back and its creator penalized via deducting some of their stake.

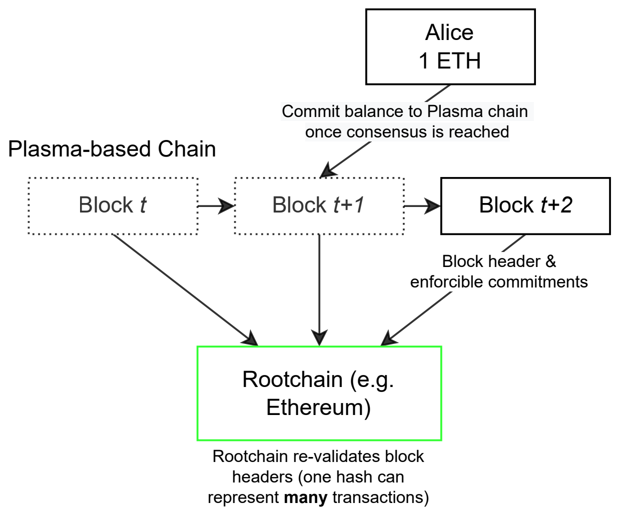
Alice
1 ETH
Commit balance to Plasma chain once consensus is reached
Plasma-based Chain
Block *t*
Block *t+1*
Block *t+2*
Block header & enforcible commitments
Rootchain (e.g. Ethereum)
Rootchain re-validates block headers (one hash can represent **many** transactions)

Fig. 4. Relationship between a Plasma-based L2 chain and its rootchain [18].

Polygon [19] uses an adapted version of the Plasma framework to enable secure off-chain smart contracts and transactions that will still eventually be validated by a reliable root blockchain (in Polygon's case; Ethereum).

Thus, it provides the speed and throughput of a mid-tier centralised system with the security of Ethereum's vast network of validators. All while providing complete interoperability available with the Ethereum network in that any token on the Ethereum network can be exchanged one-to-one with the same token on the Polygon network.

2.2 ChainLink VRF

ChainLink [20] VRF is a provably fair and verifiable source of randomness designed for use in smart contracts. A VRF is essentially the public-key version of a keyed cryptographic hash. Anyone with both the public and private key can compute a hash while anyone with just the public key can verify its result. ChainLink implements Goldberg's verifiable random function [21] which is based on the same elliptic curve cryptography that backs SHA-3. It is ran on off-chain oracle nodes with on-chain proof validator smart contracts acting as guards.

This was previously impossible to do on-chain while being tamper-proof, as older methods, such as using block hash or timestamp to seed a pseudorandom number generator, are vulnerable to attacks from miners by withholding mined blocks to change the outcome of a game [22].

Every random number provided by the ChainLink oracle comes with a cryptographic proof to verify its randomness and this proof is verified on-chain before the number is consumed by the smart contract. This process is outline in Fig. 5. The theory it is based on is elliptic curve cryptography (ECC), specifically the secp256k1 curve [23] and Keccak [24] hashing.

Keccak is the de facto standard hash function on the majority of modern blockchains due to its practically zero chance of collisions and ability to accept an infinite input space. For our purposes, we simply treat it as a black box accepting any data as input and returning a 256-bit unique, positive integer as output.

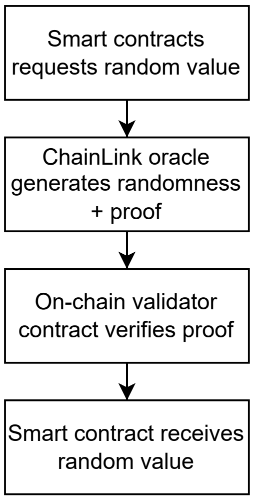
Smart contracts requests random value
ChainLink oracle generates randomness + proof
On-chain validator contract verifies proof
Smart contract receives random value

Fig. 5. Flowchart demonstrating the process of getting a verifiable random number on the blockchain.

ChainLink maintains several oracle nodes, each with a secret key k and a public key pk where p is the field of the secp256k1 curve. This public key is published on the blockchain.

To generate a random number and its proof, the oracle uses its public key and unpredictable block data (i.e. the next n block headers) as inputs to keccak256 - this produces the seed; S. A standard value for n is 10. It repeatedly hashes S until it is less than p. This is then taken as the x coordinate of some point on the secp256k1 curve.

$$V = (x, y) * k \tag{1}$$

Then, it defines V which, when hashed with keccak256, holds our random value.

To build the proof of randomness first the oracle samples a random nonce n in the range $[0, \#secp256k1\text{-}1]$. Next it computes the following values:

$$u = address(n * g) : g = G_{secp256k1} \tag{2}$$

$$v = n * (x, y) \tag{3}$$

$$c = keccak256((x, y), pk, V, u, v) \pmod{\#secp256k1} \tag{4}$$

$$s = n - c * k \pmod{\#secp256k1} \tag{5}$$

The proof is then defined as the tuple (pk, V, c, s, S). This is sent to an on-chain contract to verify, and if it succeeds, it will proceed to send the value V to the consuming smart contract.

This technology is integral to writing any contract that relies on non-deterministic behaviour i.e., the roll of a die, or the spin of a roulette wheel. To call the VRF, the contract must pay some LINK token depending on which chain it was called from.

To see the potential savings of L2 networks one can find that, on the Ethereum network, it costs 2 LINK (an Ethereum-based token to cover gas fees/ChainLink's running cost) per VRF call (~60 USD as of 30/10/2021) whereas on Polygon it only costs 0.0001 LINK (~0.00301 USD as of 30/10/2021).

2.3 Avalanche

Avalanche is a high-performance, scalable, customisable, and secure blockchain platform [25] built from the ground up (unlike Polygon). It is built on a new consensus protocol family called Snow [26], which is part of a family of leaderless Byzantine fault tolerance protocols. This new consensus protocol is also very secure and even resilient to a 51% attack, unlike traditional PoW. Avalanche implements this protocol under the name of Snowman. This protocol is part of what makes Avalanche the fastest EVM-backed, level-1 smart contract platform available for use by developers today.

The avalanche network consists of three chains: X, P, and C shown in Fig. 6. For exchanging, building platforms and deploying/running contracts respectively. In simple terms, this means exchanges are not bogged down by slow smart contracts and vice-versa as each chain is validated by a subset of validators which all have a stake in the primary network.

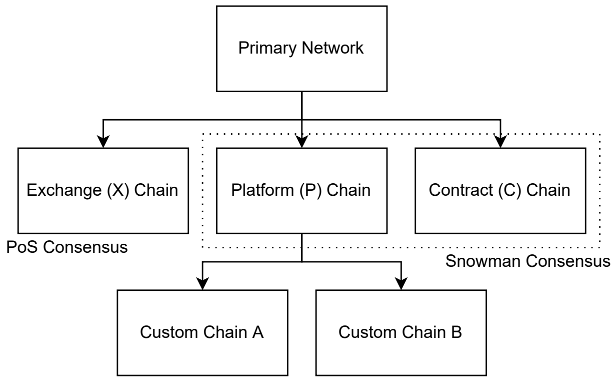

Fig. 6. The relationship between the overall Avalanche networks and its various sub-chains and their respective child chains. Interestingly, the relationship between the P chain and custom platform-specific chains is similar in nature to L2 scaling solutions like Polygon.

Due to its excellent speed and throughput capability thanks to the P chain, Avalanche is a strong contender for DApp developers who still want to focus on security. However, while it would seem Avalanche is perfect for online gambling, it lacks support for Chainlink VRF currently which is essential to building chance-based games in a trustless network.

2.4 Current Blockchain iGaming

Currently, there are very few on-chain iGaming services. Although many platforms like the popular Stake.com [27] offer cryptocurrency deposits and/or wallet integration, this is, for the most part, completely surface level. The games themselves still run on centralised servers which, while offering great response times and thus good user experience (UX), are very easily rigged with the user being none the wiser.

Many modern iGaming platforms try to combat this reputation or stigma with the 'Provably Fair' system [28]. This gives cryptographic proof of a pseudorandom number generation but can be easily circumvented. A vendor can retry multiple different seed values until an outcome that is favourable for the house is reached, as demonstrated in Fig. 7.

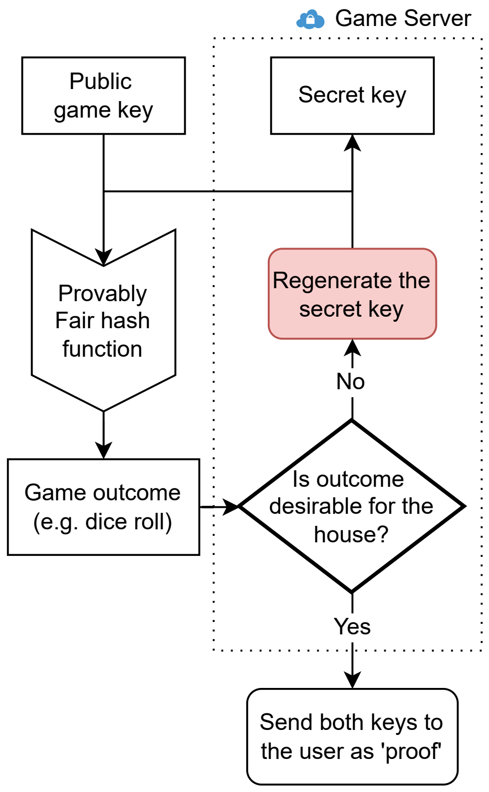
Game Server
Public game key
Secret key
Provably Fair hash function
Regenerate the secret key
No
Game outcome (e.g. dice roll)
Is outcome desirable for the house?
Yes
Send both keys to the user as 'proof'

Fig. 7. An example of a simple malicious feedback loop vendors can use to rig a game and still produce a provably fair result by sending the user both the keys which will hash to produce the outcome.

Even prominent decentralised iGaming platforms such as decentral.games [29] still operate their games off-chain with the only decentralised aspect being the currency itself. In some cases, they also offer benefits for the ownership of non-fungible tokens (NFTs) which are effectively unique digital items.

2.4.1 *Ridotto*

The largest and most similar piece of work to ours is Ridotto [30]. They aim to develop a standard cross-chain gambling and lottery protocol to allow other developers to build out their own casino on the foundations they lay. They set themselves apart from their predecessors in the emphasis on transparency and fair game outcomes as opposed to simply building a decentralised organisation (DAO). decentral.games is one such example of a DAO where the rules of the casino like who can play, who can operate a game etc. are regulated by smart contracts but the games themselves are not.

Ridotto is still in its infancy with no prototype or technical documentation available, but the ideas they propose are very similar to our work with extra details like collective ownership and no-loss games.

2.4.2 *Sunrise Gaming by DAO*

Sunrise Gaming by DAO [31] is another blockchain-backed iGaming platform that proposes some interesting ideas. However, it doesn't propose much of a solution or method to solve the problems with the current iGaming industry outlined in 1.1.

For example, tackling the issue of trust in results is arguably the key driving force in moving iGaming from the status quo to the blockchain. While, Sunrise Gamin claims it will utilise an "Absolute Probability Random Function (APRF)" and that "all the game contents will be recorded on a blockchain to prove to everyone that the numbers are completely random", any detail as to what an APRF is precisely and how it compares against a VRF is not present.

Sunrise Gaming mainly focuses on the idea of collective ownership of an online casino where holders of their NFTs can own specific tables in the casino and skim profits from games played on their tables. They also plan to create a stake pool where holders of their SUNC token can earn passive income and vote on organisational and development decisions. While these ideas are interesting they are inherently human-centric and are solving problems that the average user pays little to no attention to.

2.4.3 *Croissant Games*

Croissant Games [32] is another new project with very similar objectives to the work we present. These include points like 'no sign-ups' and 'transparent game logic'. While this project goes some ways in answering positively to the questions proposed in Section 1 it comes with a few caveats:

1) Randomness is based on the hashing of previous block data - a concept we previously established to be insecure.
2) Despite claims of transparency none of their smart contracts' code is public, verified, or audited.

However, they do make an interesting addition of a "unique gasless model based on EIP-712 [33]", which means that users pay no gas to play.

3 METHODOLOGY

The platform that we built consists of three parts:

1) On-chain games and a bank in the form of smart contracts with verified source code and transaction logs
2) An administrator client with the power to control game flow in games where it is required (e.g. roulette)
3) A front-end web app to allow users to interface with the contracts seamlessly

The final version of our work deviates from the design given in Fig. 8 very slightly. This is discussed in Section 3.6. Furthermore, the final platform consists of just two games. However, the fundamental architecture, design patterns, and structure that we introduce make it easy to extend.

3.1 Selecting a Blockchain

In selecting a blockchain we had three main choices: private self-hosted, public centralised, or public decentralised. We can immediately eliminate a private blockchain as an option, as the very premise of a casino is pitting the house against

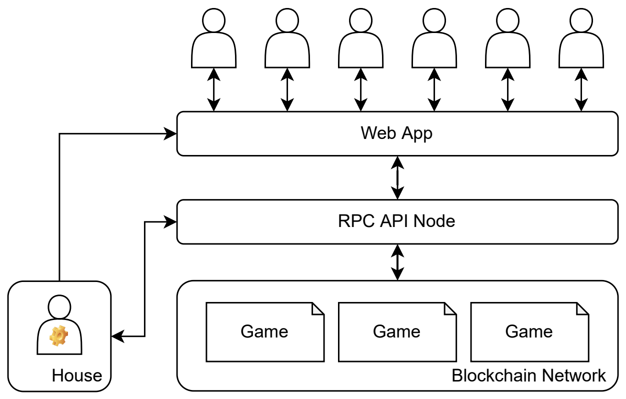

Fig. 8. The project architecture showing smart contracts deployed to a blockchain and how a user or automated user like the admin can interface with said contracts.

the players. Thus, putting the validity of bets and game outcomes in the hands of the players will not create a sustainable platform as players have a financial incentive to behave maliciously.

This leaves public (general-purpose) blockchains. These make up the bulk of well-known blockchains such as Bitcoin, Ethereum, and Polygon. The discussion and comparison can messy when considering blockchains with varying consensus mechanisms. For this reason and its slow speed, we eliminate the only mainstream, smart contract, PoW-backed blockchain available; Ethereum. However, it would be a mistake not to consider its impact and interoperability with our chosen blockchain due to its enormous market share.

Next, we must consider the distribution of blockchains. Centralization versus decentralization is more of a spectrum than a binary category when thinking about non-PoW blockchains. While PoW demands distribution with the enforcement of "one CPU, one vote" more recent consensus protocols like the early discussed PoS or Solana's [34] Proof of History (PoH) [35] can operate at any level of centralization.

Even so-called centralised platforms are distributed to some extent. In the case of Polygon, it has a cap of 100 validator nodes and introduces an auction system that only accepts new validators with larger stakes than currently running ones. Solana is more decentralised with 1,656 validators (as of 27/03/22) as shown in Fig. 9. However, as established earlier, Polygon is backed by Ethereum and thus can claim its validators as its own too.

It is essential to be aware of the degree of decentralisation of a network. For example, one only has to look to the most popular decentralised game in the world (Axie Infinity [38]) to see the repercussions of having just 8 validators backing their network. The 51% attack on Axie's Ronin network in April 2022 [39] lost users over $600 million in various tokens. This demonstrates that it is key to find a healthy compromise between the speed and throughput offered by centralisation and the security and trust offered by decentralisation.

The ideal blockchain for our work would have a good balance of speed (transaction confirmation time), throughput (TPS), wide distribution (well-rounded stake profile)

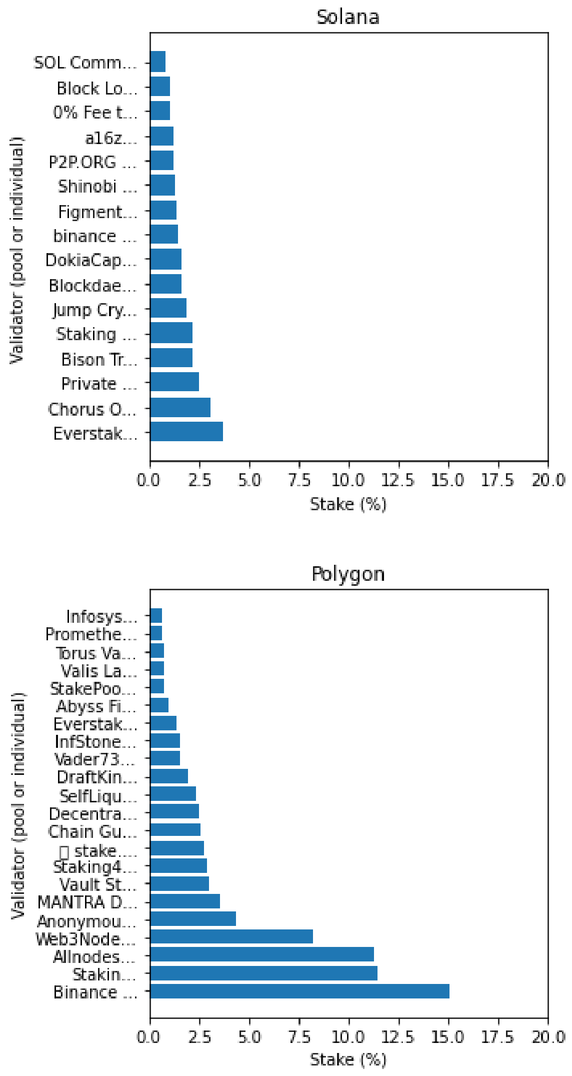

Fig. 9. Top 25 validators on Solana and Polygon [36] [37] (as of 27/03/22) shown against their percentage stake in the network. A less "bottom-heavy" graph indicates better distribution/decentralization as more validators are utilised. This shows Solana is far more distributed than Polygon with its top node staking 3.7% of the network total while Polygon's top node stakes over 15% of the network total.

and, VRF support.

Speed and throughput are obviously vital, as the number one critique of real-time DApps compared to traditional apps is their responsiveness, or rather, lack thereof.

Secondly, wide distribution makes it more difficult for validators to collaborate and perform a 51% attack on the network. While this would jeopardise all DApps on the network, a high-stakes iGaming casino would likely be a primary target if there was a significant amount stored in its bank.

Finally, VRF support is rather self-explanatory - games of chance require randomness and randomness that is susceptible to attack (like hashing block alone data) undermines the principle of a fair, trustless casino.

As shown in Table 1 and Table 2, no blockchain satisfies all our criteria perfectly. Therefore, Polygon was our blockchain of choice to build GlassCasino due to its well-roundedness in relation to all these categories and inherent

TABLE 1
Overview of fees on current blockchains to build near real-time DApps that rely on verifiable randomness as of early 2022. VRF fees are given excluding gas fees incurred when invoking the random function.

Blockchain	Transaction Fee (Transfer)	VRF Fee
Ethereum	0.000546 ETH = 1.41 USD	0.25 LINK = 4.26 USD
Avalanche	0.001 AVAX = 0.09271 USD	N/A
Solana	0.000005 SOL = 0.00056 USD	N/A
Polygon	0.00000004 MATIC = 0.0000000672 USD	0.0001 LINK = 0.001704 USD

TABLE 2
Overview of speed and throughput of the blockchains given in Table 1. Throughput is given in expected peak transactions per second (TPS) if it has not been reached in practice.

Blockchain	Block time (s)	Peak Throughput (TPS)
Ethereum	∼13.5	10-15
Avalanche	2	3400
Solana	0.4	65000
Polygon	2.2	7000

security and strong links to Ethereum as covered in Section 2.1.

Another, less quantifiable, factor in selecting our blockchain was its popularity and therefore the richness of its package ecosystem. Polygon supports practically every mainstream technology that Ethereum supports to write, deploy, manage, and monitor smart contracts as shown in Fig. 10. Polygon also supports multiple vast libraries of pre-written, battle-tested implementations of common data structures or design patterns.

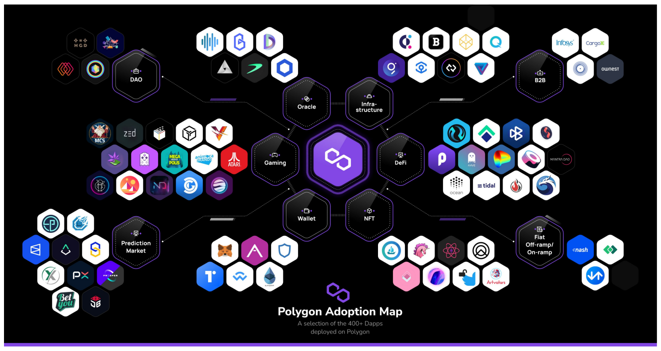

Fig. 10. A snapshot of the vast Polygon ecosystem [40] which, as shown, heavily overlaps with that of Ethereum thus offering excellent tooling, standards, and community discussion.

3.2 Smart Contracts Overview

Most smart contracts are usually written in a high-level language called Solidity [41] which is a classic curly-braced language. This language makes the construction of maintainable large-scale projects viable for developers, as the EVM can only execute the bytecode of simple operations such as `XOR`, `ADD`, or some more complex but integral cryptographic hash functions including the aforementioned `KECCAK256`.

We used Remix [42] IDE using Solidity to write and test all our contracts due to its numerous extensions to test common security vulnerabilities, gas estimation, and one-click deployment to a locally emulated blockchain.

The contracts are the backbone of GlassCasino, in that if just one game contains some exploitable vulnerability, the whole platform is compromised forever due to the permanent nature of smart contracts. Therefore, testing of the smart contracts is key for maintainability. For local development, we used Ganache [43] to run a locally emulated Ethereum-like blockchain. While this is excellent for rapid prototyping it also serves a great tool for running local end-to-end system tests before deploying to any live network.

Deploying a contract in a fault-tolerant manner is rather unwieldy. This is because a large contract costs a large amount of gas to deploy and may take multiple blocks for full confirmation due to a deployment being a very large transaction in itself. Therefore, we use a command line tool called Truffle [44] to write migrations. These migrations are effectively deployment code that keeps track of its own status on the blockchain itself. We write a small deployment chain to deal with some edge cases and post-deployment operations, which are covered later in Section 3.6.

Truffle also includes the Mocha [45] testing library to allow the construction of JavaScript-based unit testing of smart contracts. This is extremely useful for testing high-risk, complex functions of our smart contracts. For example, the pure function that calculates prize distribution must be reliable, even covering edge cases meaning unit testing is the answer.

3.3 EVM Event Logs

Before covering specific game contracts it is important to understand event logging on EVM-based blockchains. EVM bytecode defines five `LOG` operators to write records to any logging nodes. Each record consists of [0, 4] topics.

Topics are just any piece of 256-bit data that the developer of the contract wants to be logged. Solidity reserves one topic for the event/contract name. For example, take the following event definition from our roulette smart contract:

```
event OutcomeDecided(uint256 outcome);
```

Each time this event is emitted, the EVM will invoke the `LOG2` operator to store the tuple (`KECCAK256('OutcomeDecided')`, outcome) in the global blockchain event logs.

These event logs can be queried by topic very quickly using Bloom filters [46]. Included in the query result will be the transaction hash of the transaction that emitted the event, and this can be used to extract further information, such as a timestamp or block number.

This is excellent for DApp development where the state of the app (UI, logic, etc.) depends on the past state of the blockchain as it is obviously not economically viable to record all relevant topics in some ever-expanding array.

A trick that we utilise here is block number caching. This is applicable when a developer wants all events A that have been logged since the last event B was logged. This is achieved by storing the block number n in the persistent state of the blockchain when B is logged and then limiting the query of A to $[n+1$, current block number$]$.

3.4 Dice

The dice game available on GlassCasino is Chuck-a-Luck as mentioned earlier. It is a classic, simple game of chance with a relatively small house edge. A game is played as follows:

1) Player selects a number in the range [1, 6].
2) Three 6-sided dice are rolled randomly.
3) Player is paid out if one or more dice match their roll, if none match they have lost.

The game's simple rules have made this game popular for many years and adjustable payout rates have made it a favourite of casinos themselves. Classic rules dictate a single match is paid at 2:1 odds, two matches are paid at 3:1 while a triple match is casino-dependent.

TABLE 3
Table of expected returns for the casino depending on the odds given for a triple match in Chuck-a-Luck [47].

Payout for a Triple	House Edge
3:1	7.87%
4:1	7.41%
5:1	6.94%
6:1	6.48%
7:1	6.02%
8:1	5.56%
9:1	5.09%
10:1	4.63%
11:1	4.17%
12:1	3.7%

We opted for a payout of 10:1 for simplicity and to force immutability once the game is deployed to the blockchain. This is because, as described in Section 2.2, the VRF waits for 10 block confirmations between the request for randomness and its fulfilment. Therefore a mutable house edge could be changed by the house to an unfair extreme when a big enough game is started but before it concludes, granted they pay enough gas to get their transaction confirmed promptly.

While this could be circumvented with locks on the manipulation of the odds and fixing upper and lowers bounds, it is overly complex and would against the tenet of writing legible, understandable contracts that can be trusted.

Developing the dice contract presented some challenges. The first of these comes from the way the VRF contract works. First, our dice contract puts in a request for randomness and (given the contract has enough LINK) will get an ID of that request. This is because $\sim$10 blocks later, the VRF contract will invoke the `fulfillRandomness` method of our dice contract with that ID. We must therefore maintain a mapping of player address to ID so as to not mix games together.

The second issue also came from the `fulfillRandomness` function. ChainLink recommends consuming minimal gas. Therefore, we tried an approach `fulfillRandomness` won't distribute any prize but will just simply write its value to another mapping that the user can access. This introduced a usability issue wherein players would need to invoke a transaction to claim their winnings, costing time and money (in gas fees).

Writing a new value to a mapping costs 20,000 gas, but, in practice, ChainLink sets the gas limit of `fulfillRandomness` to 200,000. Therefore we were able to avoid the cumbersome prize distribution system outlined by consistently keeping gas usage under 200,000.

Fortunately, calculating winnings can be implemented as a pure function meaning it doesn't read or write to any non-volatile state in the blockchain and is therefore cheaper. Next, the single random value from the VRF must be fairly expanded to three random values [1,6], to represent each die in Chuck-a-Luck. This can also be done in a pure function by rehashing the random value encoded with the index of the die (mod 6) + 1. The final UML structure of the Chuck-a-Luck contract is given below in Fig. 11.

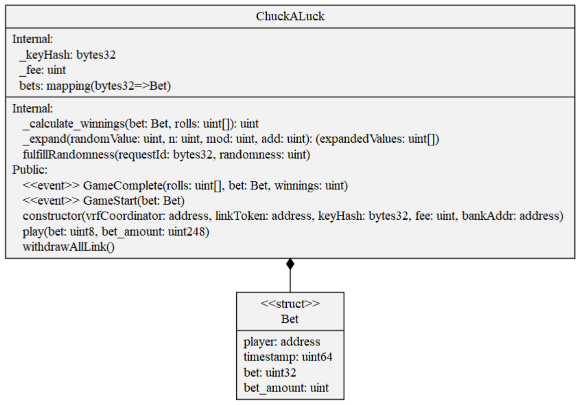
ChuckALuck
Internal:
_keyHash: bytes32
_fee: uint
bets: mapping(bytes32=>Bet)
Internal:
_calculate_winnings(bet: Bet, rolls: uint[]): uint
_expand(randomValue: uint, n: uint, mod: uint, add: uint): (expandedValues: uint[])
fulfillRandomness(requestId: bytes32, randomness: uint)
Public:
<<event>> GameComplete(rolls: uint[], bet: Bet, winnings: uint)
<<event>> GameStart(bet: Bet)
constructor(vrfCoordinator: address, linkToken: address, keyHash: bytes32, fee: uint, bankAddr: address)
play(bet: uint8, bet_amount: uint248)
withdrawAllLink()
<<struct>>
Bet
player: address
timestamp: uint64
bet: uint32
bet_amount: uint

Fig. 11. UML class diagram of the `ChuckALuck` smart contract.

3.5 Roulette

The version of roulette implemented in this work follows a set of simplified European-style rules. The wheel has just one zero giving a house edge of 2.7%. The assignment of dividends to the owner is more simple than Chuck-a-Luck as all bets can simply be given to the owner when the wheel lands on the zero. However, since the game is not zero-sum, when player winnings exceed player losses on certain rounds, money will have to be paid out of the bank. Lastly, there are just four types of bet: odd, even, black, and red and this is easily extensible.

Our implementation of the roulette smart contract, unlike the dice game, uses a vulnerable form of random number generation discussed earlier; hashing block data. This was done for simplicity and speed to implement, as well as to serve as a benchmark for just how unfortunately slow ChainLink VRF is in practice.

The roulette contract also has a few other minor differences from a single-player game. That is, we must account for an upper amount of bets available in a single round. This is done to reduce gas costs for users as writing to a dynamic array costs 4x as much gas. Secondly, there must be some trusted party to actually 'spin' the roulette wheel at a steady frequency. This is discussed further in Section 3.7.

The final UML structure of the Chuck-a-Luck contract is given below in Fig. 12.

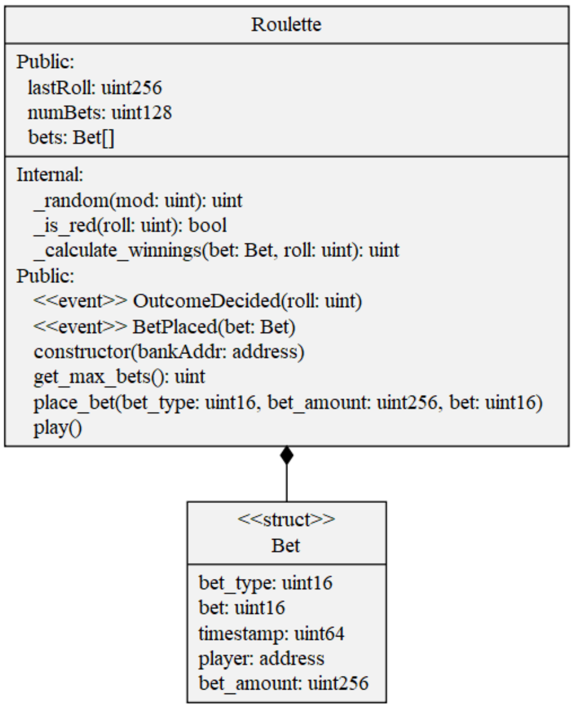
Roulette
Public:
lastRoll: uint256
numBets: uint128
bets: Bet[]
Internal:
_random(mod: uint): uint
_is_red(roll: uint): bool
_calculate_winnings(bet: Bet, roll: uint): uint
Public:
<<event>> OutcomeDecided(roll: uint)
<<event>> BetPlaced(bet: Bet)
constructor(bankAddr: address)
get_max_bets(): uint
place_bet(bet_type: uint16, bet_amount: uint256, bet: uint16)
play()
<<struct>>
Bet
bet_type: uint16
bet: uint16
timestamp: uint64
player: address
bet_amount: uint256

Fig. 12. UML class diagram of the `Roulette` smart contract.

3.6 Central Banking

Our initial design had each game's smart contract implement its own internal bank. This internal bank obviously requires elevated permissions that should not be afforded to any other user i.e. the ability to create or delete chips. Thus, an internal bank where the contract itself is the only entity that can perform these actions seemed an ideal solution.

However, when building a second game, a key usability problem arose; each user would have a different number of chips on each table. This means if a user wins x chips on game i they cannot use their winnings on game k without first withdrawing the x chips to their wallet and then re-depositing them to the internal bank of game k. This is overly cumbersome and slow.

The obvious solution here is to de-couple the banking logic from the game logic and maintain a list of games in a central bank that should be allowed to add or subtract funds.

Fig. 14 obviously presents a far less cumbersome experience for the end-user than Fig. 13. However, it introduces the issue of a central authority; the bank. This directly goes against the grain of the underlying objective of this work.

The solution we implemented was a permission system based on assigning certain pre-defined permissions to addresses. This allowed us to maintain trustlessness despite now having a central authority.

This was achieved with an extension of OpenZeppelin's `AccessControlEnumerable` contract [48]. OpenZeppelin is an extensively tested, open-source repository of commonly used utilities for smart contracts, and thus serves as

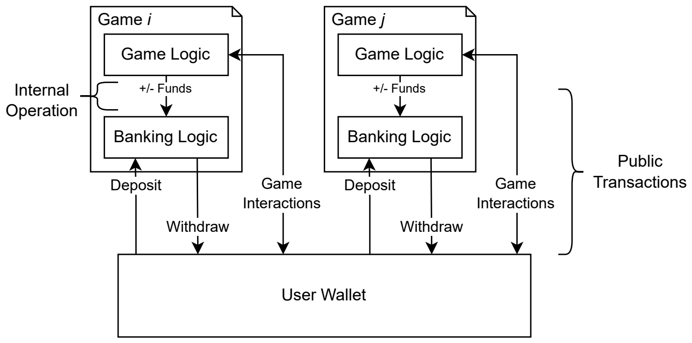

Fig. 13. A diagram of game-independent, internal banks as initially implemented and designed. This clearly demonstrates the inherent usability issue as users must jump through many hoops to withdraw their winnings and play another game.

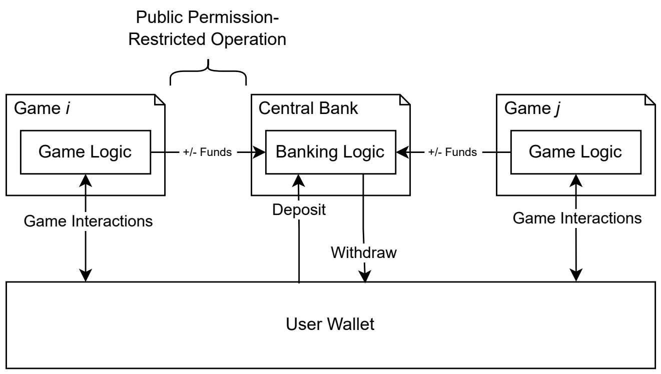

Fig. 14. Demonstration of how the game contracts and central bank contract interact with each other and in relation to the user.

an excellent resource for building out robust, maintainable systems.

The contract we specifically utilise here; `AccessControlEnumerable`, allows addresses to hold arbitrarily created roles defined by the contract developer. Each role has an administrator role that can add or remove users from its child role. Addresses can be a member of infinitely many roles. The enumerable aspect of the contract is a simple extension that allows iteration over all addresses that hold a given role. The simplified structure of this contract is given below in Fig. 15.

Transactions defined in the contract can specify that they require the caller of the method to hold a specific role, or else it will revert and will not mutate the state of the contract on the blockchain. This is key for us, as we want to protect the methods of the bank that create or delete funds so that only the games and no one else can perform these actions. Then, since we assume the game code to be safe and fair, we maintain trustlessness.

To achieve this, we define two roles in the central bank; `ADMIN` and `OPERATOR`. In the contract's constructor (the code that is run once in its lifetime at deploy time) we set `ADMIN` as the super-role of `OPERATOR`. This means that any address that holds the `ADMIN` role can assign the `OPERATOR` role to any address, at any time.

Initially, the only address holding either role is the contract deployer as defined in the contract's constructor. They hold the `ADMIN` role. This means that they can then

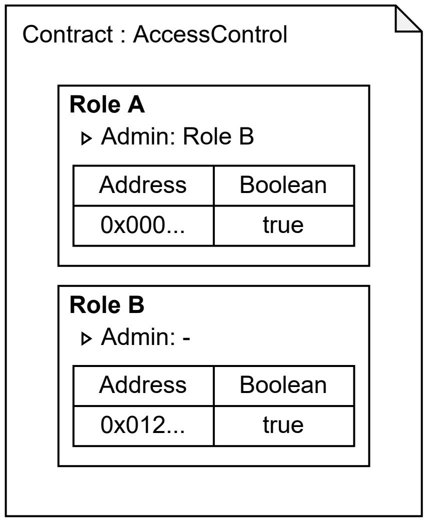

Fig. 15. The general structure of the `AccessControl` roles. This shows the reliance on Solidity's `mapping` data structure allowing the mapping of an infinite input space to a finite output space. In the boolean case here, any address not present in the mapping will default to false. This means the mapping is simply acting as a hash set.

deploy a game contract with a reference to the deployed bank contract's address and allocate the `OPERATOR` role to the game contract's address. This creates the two-way link we need where:

1) The stand-alone bank contract must have a set of `OPERATOR` addresses who can manipulate funds (the game contract addresses) **AND**
2) The game contract must have a reference to the address of the bank contract in order to call the methods that manipulate the funds

Once all games have been deployed and tested, the deployer should renounce their `ADMIN` role, and thus no one will hold this role. This can be verified via enumeration as introduced earlier. Renunciation of the `ADMIN` role permanently freezes the list of addresses that hold the `OPERATOR` role as there are no `ADMIN`s to add or remove addresses from this role. With this action, as in Fig. 13, the user only needs to trust the code of the game and the bank, as no other address has the ability to create or destroy funds.

In a development/testing environment, however, it may be desirable to allow the hot-swapping of games, and thus on development networks or builds, the deployer can skip their renunciation of `ADMIN` and add/remove games in the future. This should never be the case in a production app though as there is a serious security risk of the `ADMIN` allocating themselves or a malicious contract the `OPERATOR` role, allowing them to empty the bank's real balance into their wallet. A full logic flowchart of this process is given in Fig. 16.

The final action that we perform is the verification of the source code of the contracts on polygonscan [49]. This is a web-based block explorer for the Polygon network which allows users to query any block number or hash, transaction hash, wallet address or, contract address and easily read all available, relevant information on the queried item.

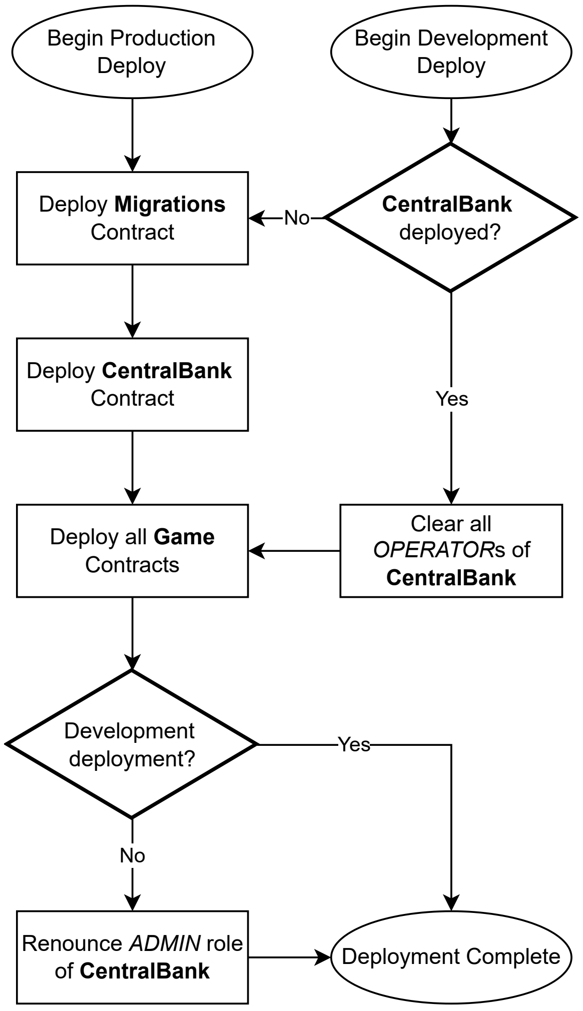
Begin Production Deploy
Begin Development Deploy
Deploy Migrations Contract
No
CentralBank deployed?
Deploy CentralBank Contract
Yes
Deploy all Game Contracts
Clear all OPERATORs of CentralBank
Development deployment?
Yes
No
Renounce ADMIN role of CentralBank
Deployment Complete

Fig. 16. Flowchart of our migration chain. It's important to note that each rectangular box represents a distinct migration step and is committed to the `Migrations` contract set up in the initial step. This means if the flow is interrupted for any reason (e.g. running out of gas) the deployment is restarted at that step after remedying the interruption.

Polygonscan provides an API to upload contract source code in Solidity and compiler parameters (software version, optimizer passes etc.). This will be compared against the EVM bytecode stored on-chain and if it is a match the verified human-legible Solidity code will be stored on the contract's block explorer page. GlassCasino takes advantage of this API to further bolster the transparent nature of the casino.

3.7 House Server

The house server is made up of two connected components; a WebSocket server and a blockchain-connected scheduler. Both of these components are written in Node.js. The full structure of the house server and how it interacts with the rest of the platform is shown in Fig. 17.

Firstly, it is vital to establish how to interact with any state on the blockchain like checking balances or submitting signed transactions. This is done by making RPC (Remote Procedure Call) requests to an RPC API node. There are two types of RPC nodes; public general use and developer cloud-hosted. General-use nodes have a lower rate limit, do not require an API key, and are typically used by users to connect their wallet apps to the blockchain. Cloud-hosted nodes handle higher throughput, have a faster response time and can allow for access control. This access control can be configured to restrict access to certain contracts, IPs or source domains.

The option of self-hosting an RPC node was not explored in this work due to its immense cost in time and money with a negligible performance edge. Instead, GlassCasino uses an RPC node provided by Alchemy [50].

Integration with Alchemy was seamless and instantly offered speed improvements of several seconds over a general public RPC node like Polygon-RPC [51] which we originally used.

However, interacting with the RPC nodes directly is unnecessarily difficult. Instead, most developers utilise a library to abstract away the raw RPC method names, HTTP requests and packaging of operands. GlassCasino uses ethers.js [52] for the house server and the web app.

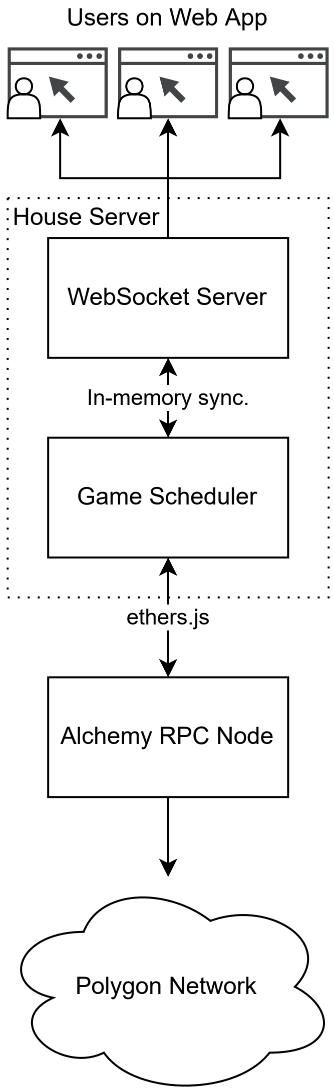

Fig. 17. How data from the blockchain is processed and used to authoritatively dictate game scheduling information and relay it to the web clients.

To schedule an event there must always be some timer. In classical centralised systems, this is trivial to implement but it is unreasonable to expect validators to constantly run through a list of scheduled transactions at every iteration.

Projects like the Ethereum Alarm Clock Service [53] work by providing upfront funds to cover future gas fees and a tip for invoking the function at a fixed interval. However, it still relies on an external network of task runners to be completing these tasks, thus it is not much better than a central task runner.

The existence of a central authority that controls game flow does not completely undermine the trustless nature of the casino, as the worst-case scenario is that the admin can effectively hold users' bets in limbo for which there is obviously no financial incentive.

A solution to this would be time-locking the `play` function by storing the timestamp in the contract's persistent state and verifying a certain amount of time has elapsed before any address can call it again. However, block timestamps are rather spaced out due to the multi-second block time of most networks and thus offer little precision which is important in building a robust UI. There is also still no guarantee that anyone would invoke the `play` transaction at all.

In the end, we opted for an off-chain authority to manage the flow of roulette. The scheduler is a simple timer implementation that listens for roulette bets and counts down a set interval to invoke the `play` transaction to spin the wheel. It is designed to be online at all times and to consume little to no computing power when no one is playing roulette. While this is not ideal, as it adds a central authority to what is otherwise a completely distributed casino, it is a necessary component.

3.8 Web App

The final component to make this platform complete is, of course, some user interface to allow for interaction with the general public. Our work, in this regard, consists of a web application for maximum user reach.

This works well as the standard for DApps is to be web-based and rely on some kind of browser-installed (or, in the case of mobile devices; local app) wallet extension. The most well known of these is MetaMask [54] - a browser extension/mobile app wallet built for EVM-based blockchains such as Ethereum or Polygon.

The user interface (UI) itself was built with Vue 3 [55]. This allows the separation of the app into views which consist of components. Components can be easily re-used which is particularly useful when lots of instances of the same UI element exist independently from one another. For example, a fundamental component is that of a `BalanceBox` which takes a value in MATIC and formats it correctly with the token symbol after. A final rendered version of is component shown below in Fig. 18.

However, this component has some extra functionality and styling because all values of ETH (or MATIC in our case) are given in wei. Therefore, all libraries including ethers.js store currency values as 256-bit integers, much like the EVM. This requires some special processing in validating (i.e., more than, less than) and displaying. To display a value it must be converted to a string, sliced along

Fig. 18. Example of a `BalanceBox` with the extra 'Wallet' header used on GlassCasino to indicate a user's current wallet balance without requiring the constant checking of their wallet provider.

the desired number of decimals and measured to make sure it fits its bounding box. Finally, the token image must be appended to signify the value is a currency.

Some components require the sharing of state. This is where a global state manager comes in. GlassCasino uses Vuex [56]. This state contains:

1) The global reference to the Alchemy RPC provider
2) The transaction signer (assuming a wallet could be detected and its connection to GlassCasino was approved by the user)
3) Extensive wallet detection and cookie-based memorisation of prior approval (this includes adding Polygon to the user's wallet and switching networks if required)
4) Reference to the `CentralBank` contract
5) Reference to the currently open `Game` contract
6) Current wallet & `CentralBank` balances (given a signer exists), along with methods to refresh them
7) WebSocket client to connect to the house server
8) Dynamically updated game flow information from the house server

We bind to the on-chain contracts by instantiating an instance of ethers.js's `Contract` utility class. To do this, the application binary interface (ABI) is required along with the address of the deployed contract. Truffle saves this information as a JSON file each time the migration chain (Fig. 16) is run. Thus, this JSON file is simply bundled into the web app.

The process of client-side contract interfaces binding values it receives to UI elements is shown below in Fig. 19.

The rest of the UI is built using local and global states, reusing component logic, and styling as much as possible. The final work presented here consists of two full-screen views with some common elements between them, such as navigation bars.

As shown in Fig. 20 and Fig. 21; the web app is designed to respond to viewports of all sizes and detect wallets on any mobile or desktop device. This is important, as 50% of all online gambling happens on mobile devices as of 2019 [3]. In the future, adding progressive web app (PWA) support would be desirable to create a native-like app feel with very little development overhead.

Some more interesting components to note are the roulette's current bets, game history, and Chuck-a-Luck's game history. All three of these components use event log querying to populate their UI with 'current bets' using the block number caching trick discussed in Section 3.3.

Each UI component that relates to a transaction (i.e. a specific bet or a game outcome) will also display a polygonscan URL along with it, allowing the user to verify the legitimacy of the game with one click.

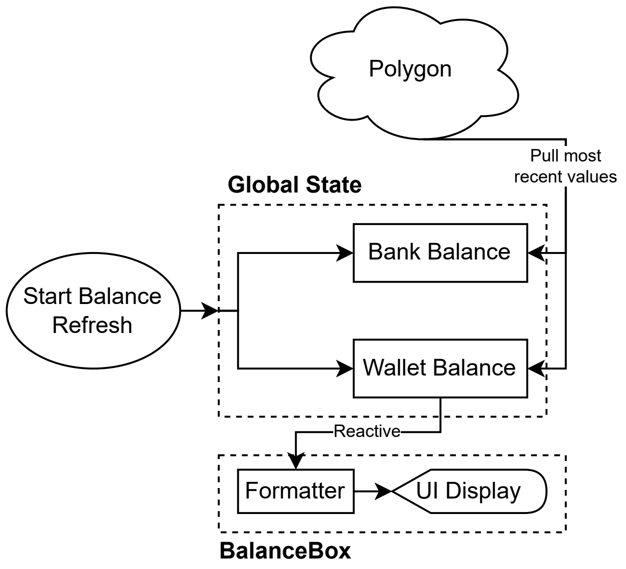

Fig. 19. A basic example of how the UI is bound to values fetched from Polygon's global state and how user interaction can force these values to be refreshed thus updating the UI elements.

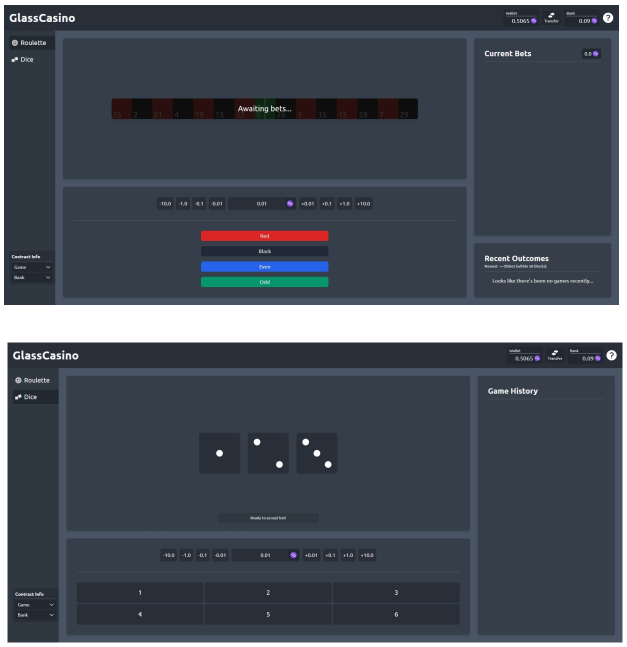

Fig. 20. Full views of the roulette UI (top) and the Chuck-a-Luck UI (bottom) on a standard 16:9 1920x1080 desktop monitor.

4 Results

Our results are recorded with respect to the questions presented in Section 1, particularly we focus on measuring the technical viability of implementing a blockchain-backed online casino.

We break technical viability down into the following three sections:

1) Speed
2) Cost of operation (for the user and owner)
3) Scalabity

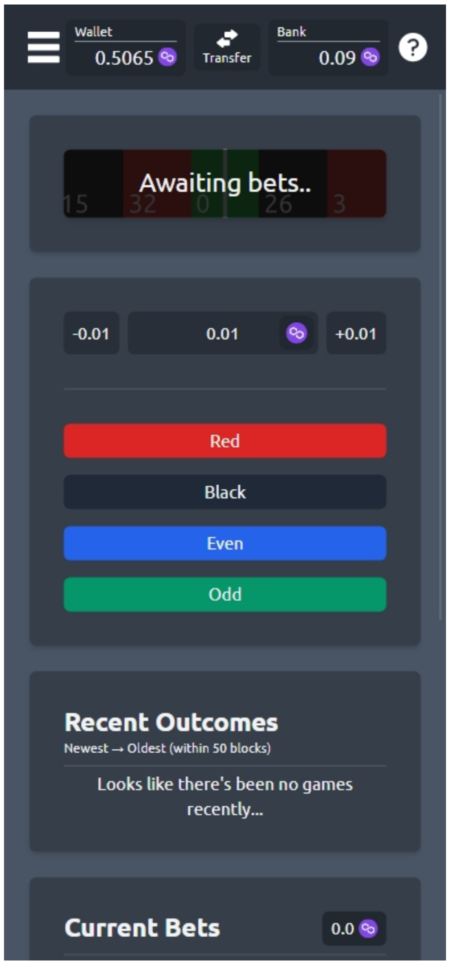

Fig. 21. Full view of the roulette UI displayed at the full-screen resolution of an iPhone XR (414px, 896px)

In the following section we focus on listing the results and describing the way in which they were gathered. A more detailed exploration of the impact of these results is given in Section 5.

4.1 Overall Platform Speed

Our results pertaining to speed are split into two parts to measure separately:

1) Smart contract speed
2) Client speed (initial load, download size & responsiveness)

We measure the speed of execution of smart contracts by executing the most important transaction on each of our smart contracts 100 times and timing how long it takes to receive confirmation. These results are displayed in Fig. 22.

Interestingly, depositing to the central bank and placing a roulette bet have very similar execution times. This is likely due to their similar gas fees as in Table 6 and Table 7.

It's worth noting that all smart contract testing was conducted on the Polygon Mumbai test network. This is a one-for-one copy of the Polygon main network but is designed for ease of development. The main point is that MATIC is worth nothing and is given away freely.

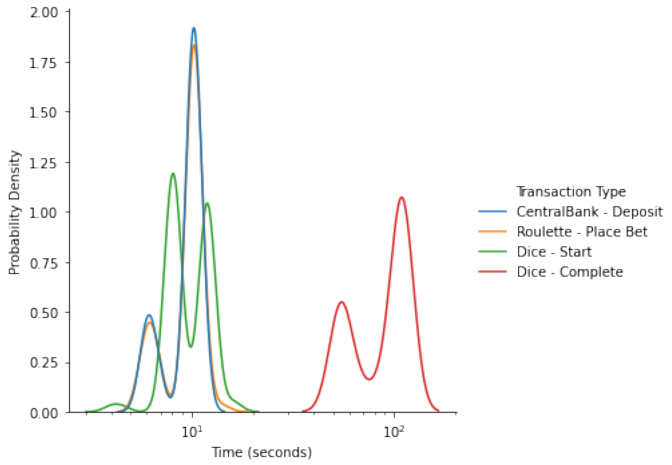

Fig. 22. Distribution of timings of entry-point transactions for each of our smart contracts against the density at which they occurred when executed 100 times. These were measured on the Polygon Mumbai test network with automatic gas fee estimation from ethers.js.

When measuring our client app's performance, we consider it in the context of two other popular iGaming platforms: Croissant Games and bc.game. The former was introduced earlier as a similar DApp to ours whereas the latter is a much more popular, centralized iGaming platform supporting the betting of cryptocurrency. These two apps serve as excellent references when considering web app performance as they are the accepted standard in both paradigms of web-based gambling.

To quantify client app performance, we use WebPageTest.org [57] to record bundle size (that is, how much data the user must download to load a page), render progress, and other omitted metrics. We plot these results in Fig. 23 and their corresponding speed indices in Table 4. This speed index metric is defined by the following equation:

$$\mathrm{SI} = \int_0^{end} 1 - \frac{\mathrm{VC}}{100} \tag{6}$$

Here we effectively take the area above the visual completeness (VC) line in Fig. 23, therefore a higher number is worse as it indicates a longer time to reach full visual completeness.

TABLE 4
Speed indices of the same three web apps as Fig. 23 and under the same test conditions.

Web App	Speed Index (ms)
bc.game	1601
GlassCasino	1609
Croissant Games	18272

Finally, we list each web app's bundle size (the total size of JavaScript files, images, HTML and other documents downloaded when loading a page) in Table 5.

4.2 Cost of Operation

First we calculate the guzzle rate of each game; this rate measures how many units of gas are used per minute by

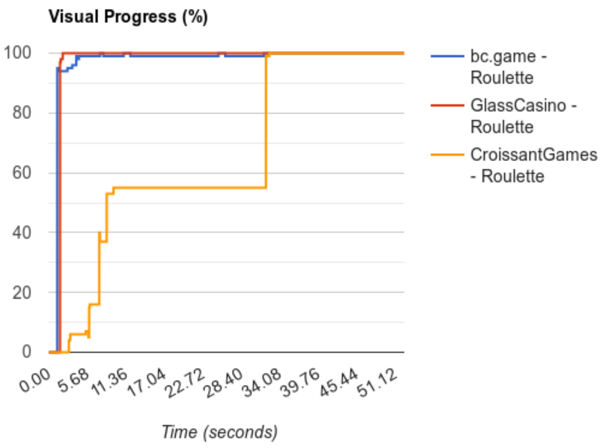

Fig. 23. Visual progress graph of three gambling web apps, including GlassCasino. Measured on Firefox v98 for desktop, in Germany using a standard cabled internet connection (5000 Kbps down/1000 Kbps up, 28ms latency).

TABLE 5
Bundle sizes of the same three web apps as 23 and under the same test conditions.

Web App	Bundle Size (KB)
bc.game	6035
GlassCasino	399
Croissant Games	17638

a given contract. Block explorers like polygonscan will list the top gas guzzlers and spenders. The guzzle rate (gas per second) is defined as follows for a set of transactions $T : |T| = k$ that occurred over s seconds.

$$G/s = \frac{\sum_{i=1}^{k} GasUsed(T_i)}{s} \quad (7)$$

The following tables show the measured gas costs for important transactions on each smart contract ($GasUsed(T_i)$). These were measured using a locally deployed Ganache blockchain and the Truffle command-line tool. The contracts were compiled with Solidity 0.8.7 with 200 optimizer passes.

These results were obtained through manual analysis of transaction history and taking maximum/minimum values or, in the game of consistent values, a mean. These fluctuations are difficult to predict due to the optimisation performed by Solidity to save gas in particular edges cases.

TABLE 6
Table of gas fees for the write transactions of the central bank smart contract.

Method	Gas Used
Deposit (from 0)	43848
Withdraw (to 0)	38394
Deposit	28848
Withdraw	23394

4.3 Scalability

Scalability is dependent on the gas guzzle rate of the contracts. We calculate this by taking the total block gas limit of Polygon (20 million gas per block) and the gas amounts presented in Table 7 and Table 8 to visualise network utilisation against the transaction throughput supported. These results are shown in Fig. 26 and Fig. 27.

TABLE 7
Table of gas fees for the write transactions of the roulette smart contract. We assume all betters have non-zero balances and that the contract has been deployed and used for a while (i.e. all relevant fields are no longer at their initial value of zero).

Method	Gas Used
PlaceBet	[55868, 90068]
Play (0 betters)	~30948
Play (n betters, n losers)	23302 + ~3149n
Play (n betters, ≥1 winner)	30948 + ~3149n

TABLE 8
Table of gas fees for the write transactions of the Chuck-a-Luck smart contract. These results were measured on Polygon's Mumbai test network as the Chuck-a-Luck implementation uses ChainLink VRF.

Method	Gas Used
Play	[178286, 212486]
FulfillRandomness	[86993, 92445]

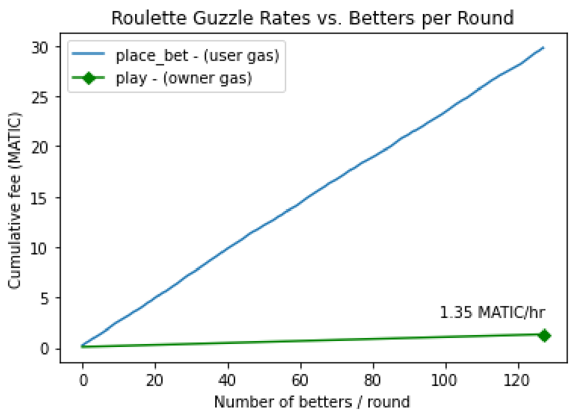

Fig. 24. Simulated guzzle rate of the roulette contract. We assume a maximum bet count of 128, 45-second rounds, and a gas fee of 70 gwei (gigawei) per unit of gas.

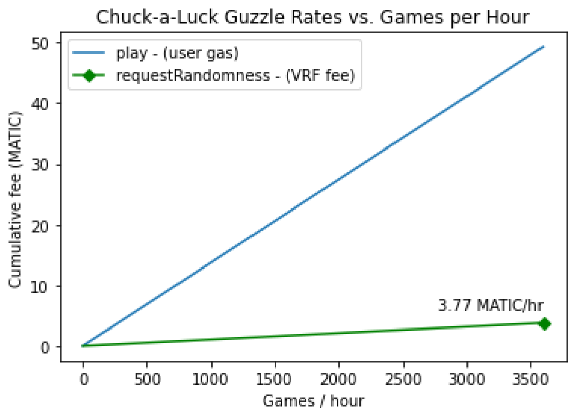

Fig. 25. Simulated guzzle rate of the Chuck-a-Luck contract. We assume a VRF fee of 0.0001 LINK per call, a MATIC to LINK exchange rate of 0.09541955, and a gas fee of 70 gwei per unit of gas.

These results are obtained by interpolating between a small number of games/betters and extrapolating up, assuming the cumulative fee is linear. This is a fair assumption as there is no exponential time or space operations committed to the blockchain in any part of the final smart contracts.

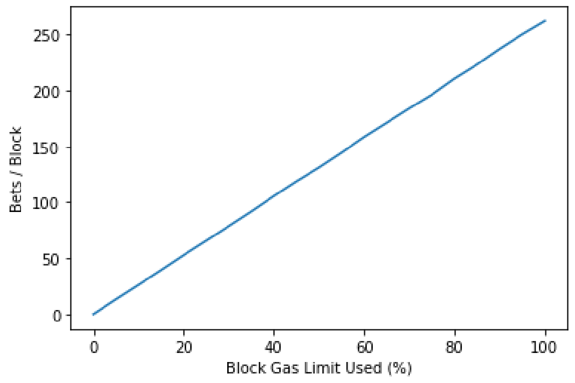

Fig. 26. Percentage of all gas used on the network (%) against the number of betters that proportion of gas can support (normalised to bets/block rather than bets/round).

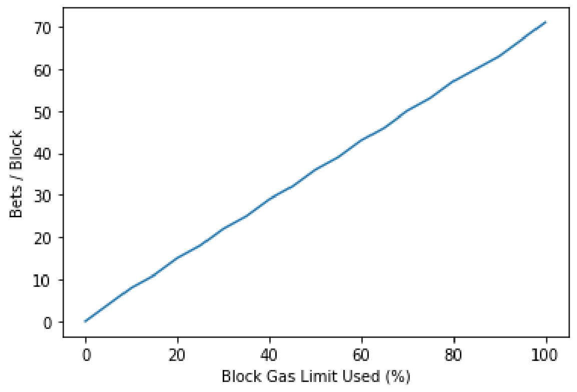

Fig. 27. Percentage of all gas used on the network (%) against the number of dice games that will fit per block.

5 Evaluation

We evaluate our work with respect to the three categories of technical viability presented in Section 4. However, we also consider the second question presented in Section 1 in analysing how well we leverage the benefits offered by blockchains such as provability, decentralisation, and security.

5.1 Overall Platform Speed

As Fig. 22 shows, the transaction speeds of the three methods we implemented were consistent.

However, their performance is still lacklustre with the majority of cases taking 6 or 10 seconds. Interestingly, we see large dips in the middle of all of these distributions. This occurs as a transaction must be placed into a block and blockchains all have a fixed block time.

Therefore, it's preferable to get into the earliest block possible. In our tests, this appears to occur $\sim$ 6 seconds after invoking the transaction.

These speeds are almost entirely dependent on external factors such as network congestion or how much gas the user pays when invoking the transaction. However, even if all outside factors were in our favour a best-case scenario of $\sim$6 seconds is not good enough to hold user interest, as research states that most users will mentally switch context when waiting for more than a second, and usually quit out after 10 [58].

Chuck-a-Luck's time to completion is abysmal due to the process of generating verifiable randomness as described in Section 2.2. Secure, decentralised DApps that depend on randomness will require a more creative usage of VRFs until a better alternative presents itself as wait times of nearly two minutes are unacceptable for the modern internet user.

As for the web app; our fundamental goal was to build a light, performant platform to partially make up for the poor speeds of transaction confirmation on the blockchain.

The HTTP archive [59] performs a monthly measurement of speed indices of the top one million websites as ranked by themselves. In 2022, they found the median speed index of a page loaded on a desktop to be 3500 ms.

GlassCasino's excellent score of 1609 ms puts it just outside the 10th percentile, which sits at 1533 ms. The indistinguishable result from that of bc.game is also very promising, as it is a staple centralised iGaming platform. Our work also outperformed Croissant Games by an order of magnitude in this metric.

Our bundle size of 399 KB also fairs favourably compared to the findings of the HTTP archive and the benchmark web apps we measured against. Not only is our work 44x lighter (download-wise) than its closest peer, but sits comfortably within the 10th percentile of total bundle sizes, which is 455.1 KB.

However, it's important to note that this is not a like for like comparison as bc.game is a much more mature, richer app with many more features even on the page of a single game. While the good load times and bundle size of GlassCasino seem positive, it is vital to keep them at this level when attempting to reach feature-parity with an app like bc.game.

5.2 Cost of Operation

Our operational cost results shown in Fig. 24 and Fig. 25 are rather promising for realistically small numbers of users/games. The owner is expected to pay minimal fees with both games that, on the upper end of users, would easily be covered by their house edge. Thus, due to the linear scaling of fees to users, both games are sustainable as long as the average bet is large enough.

For example, to break even on Chuck-a-Luck the average bet S must be at least $\sim$0.023 MATIC:

$$S \geq \frac{0.0001 \text{ LINK} \approx 0.00106 \text{ MATIC}}{0.0463} \tag{8}$$

The average stake required to break even on Roulette is even lower despite its inferior house edge because, as shown in Fig. 24 the house fee still scales linearly but with a smaller gradient.

Costs also do not increase for users as more users join in, as the contracts are written to be user count-agnostic when it comes to gas fees.

5.3 Scalability

For scalability we only consider smart contracts. This is because it is known to be the bottleneck of all DApps. Infrastructure to scale classical web applications through content delivery networks (CDNs) is ubiquitous with the modern web so it's not considered here.

Our scalability results in Fig. 26 and Fig. 27 are promising. We find that Chuck-a-Luck can support up to ∼35 bets / second when using all the gas on the network. This is obviously infeasible to sustain; thus a realistic at-scale number of bets / hour to maintain would be >10 bets / second.

Roulette, on the other hand, can support ∼3.5x the number of bets per block as Chuck-a-Luck at any given network utilisation. This is primarily due to its multi-player nature combined with the lack of VRF use in its implementation. Since just one number needs to be generated every s seconds to satisfy multiple users it is much more gas efficient.

Our throughput results are unsurprisingly poor when compared against those of traditional, centralised iGaming. With very conservative estimates of their throughput we find that at least 20% of the Polygon network's global gas limit would be used by our two games alone .

To produce this estimate, we only consider games that UKGC recognises as 'casino (remote)' (making up 22.5% of all gambling in the UK). These are 72.5% slots, 13.2% roulette, 4.9% blackjack, and 5.9% 'other' as of 2021 [60].

Using this, we can begin an assumption of the number of active online gamblers in the UK alone. There were 24 million gamblers in the UK in 2020, 22%, of whom play more than twice a week [61]. Thus, we can assume that there are $24,000,000 * 0.22 * 0.225 = 1,188,000$ twice-weekly online gamblers in the UK alone. Assuming, conservatively, that each player uses online casinos for an average of an hour, twice a week: the iGaming industry in the UK handles $\frac{1,188,000}{168} * \frac{1}{60} \approx 236$ bets/minute. Converting this to Polygon's block time is $\frac{236*2.2}{60} \approx 9$ bets/block. While it is obvious that no single app in particular would hold 100% of this market share it is important to consider the goal of implementing all iGaming on the blockchain and thus this number is the one that matters.

Fig. 26 shows roulette can comfortably handle this utilising less than 5% of the network's total gas. On the other hand, Fig. 27 shows that Chuck-a-Luck would use nearly 15% of the network's total gas under a similar load.

Although these results seem relatively reassuring, even a small surge in usage would push network usage far too high. Also, these results also do not account for other platform-related transactions like deposits/withdrawals or for the concurrency between different games.

Even with our very conservative estimates (single account per user, just 2 hours per week of usage, perfectly evenly distributed demand), our platform does not compare well to modern centralised systems or even to blockchains built with the purpose of high throughput in mind. If we even slightly tweak our estimates by, for example, assuming that people only gamble when they are awake or that usage will rise when people finish work, we go far beyond the upper limits of 250 or 70 bets/block at 100% network usage.

A potential solution could be creating "subnets" on the aforementioned Avalanche P-chain as introduced in Section 2.3. This would reasonably allow for near 100% utilisation of the network, which is not feasible for public networks. This would make reaching the upper end of users displayed in Fig. 26 and Fig. 27 more realistic without compromising security. At this upper-band, the platform compares slightly more favourably to the estimated numbers we deduce earlier.

5.4 Decentralization and Security

It is difficult to record results for the level of security as it is always a game of attack and defence. However, auditing the completed smart contracts with polygonscan and Remix IDE did not reveal any severe security warnings.

As for decentralisation, we know that, by virtue of the 100 validators of Polygon, there is little chance of a 51% attack. Therefore, to maintain this level of decentralisation, smart contracts must have no single trusted entity. This is not completely possible with any iGaming platform due to the inherent need for a house to provide funding and infrastructure.

Our implementation of Chuck-a-Luck is as close to as fully decentralised as possible. It is completely governed end-to-end by the immutable, transparent code deployed to the blockchain. It, however, is not without some authority - that being the CentralBank which is an unfortunate necessity to minimize gas fees and maximize usability.

Roulette is, of course, limited in its level of decentralisation by the existence of the game flow management server.

Finally, the web application used to interface with these smart contracts is distributed by a CDN, and thus controlled by a single governing body, the web host. While the contracts can be transacted with directly via sending raw signed transactions to any RPC node or even polygonscan, most users will simply use the web app. Thus, if the web host is hacked, the malicious party can edit the contract addresses baked into the web app to point to malicious contracts instead. This is probably the most dangerous single point of failure but it is the reality of our current centralised internet.

5.5 Software Engineering

The software engineering process we adopted when building GlassCasino was a slightly modified Agile approach with continuous integration and continuous deployment (CI/CD). Due to the number of developers working on the project collaborative design obviously does not make sense and so traditional the Agile approach did not apply.

Our work flow consisted of approaching one of our objectives at a time, planning out a general structure and design before beginning a code sprint. This was interwoven with testing and upon completion of the objective it would be integrated to the rest of the project and deployed. Continual deployment was done by updating the main branch on our version control system. This would trigger the web app and house server to reload from the newly updated scripts on their respective servers.

While the approach did work and allow for rapid development it could have benefited from a more rigorous structure. For example, test-driven development would likely have been the best choice for writing smart contracts, ensuring 100% coverage.

While version control software like git was key in the construction of GlassCasino to track smaller changes made in code sprints, the organisation could have definitely been improved. In the future it would be beneficial to use a progress tracking tool like Kanban boards, particularly for incremental objectives like the web app.

6 Conclusion

In summary, we have shown explored the various aspects of technical viability in regard to building a trustless, secure blockchain-backed iGaming platform.

We find some positive results; namely in the existence of truly secure, trustless systems with verifiable randomness. Verifiable, decentralised, and affordable games of chance are possible to implement on current EVM-based blockchains with some novel smart contract tricks and special considerations.

We also found modern DApp tooling like Truffle and ethers.js to be excellent at what they do. Truffle allows complete control over a robust and reliable blockchain deployment chain. It also provides an excellent general-purpose toolbox for debugging and testing DApps. Ethers.js on the other hand is an excellent, lightweight and intuitive library for creating traditional web interfaces connecting with the blockchain and user wallets.

However, we also find many technical limitations with the technology stack that underpins our platform. Our work suggests that while Polygon is an excellent L2 scaling solution for general-purpose transactions where speed is not a concern, it simply does not scale to the level required by even simple games like Chuck-a-Luck or roulette. Its gas fees are too prohibitive to scale to the size of even a small/medium enterprise, and its block time and confirmation times are too slow to retain users long-term.

6.1 Future Extensions

There are several future extensions to this work that would improve its technical viability when it comes to speed and throughput. One such example would be the introduction of VRF nodes on a network like Avalanche, which supports EVM-compatible smart contracts.

Another smaller area of research would be gas-less betting, similar to that of Croissant Games mentioned in Section 2.4.3. This completely eliminates one of the three major pain points for DApps being gas fees, slow speeds and, understanding.

More substantially, it would be interesting to see a study into the effects of completely rebuilding the blockchain side of the system to a more technically radical blockchain like Solana and migrating tooling to that ecosystem. We expect there would be significant speed and throughput improvements at the cost of using a slightly less popular blockchain. This would also depend on the implementation of verifiable randomness like a VRF node on Solana.

Furthermore, the implementation of player-versus-player (PvP) games like any variety of poker would be the next step to flesh out the catalog of viable, decentralised games. Both games presented in this work are player vs. house. While creating a smart contract to govern fund allocation and pooling seems trivial, the interesting section to study would be that of drawing and verifying each player's cards while keeping them a secret from the other players.

Finally, the exploration of the viability of a houseless casino with scheduled games such as roulette would remove the only central point of failure in this work. While the house server is not a devastating point of failure, it goes against the grain of this work and eliminating it would go a long way in proving the viability of a completely blockchain-based ecosystem for online gaming.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] S. Gainsbury, J. Parke, and N. Suhonen, "Consumer attitudes towards internet gambling: Perceptions of responsible gambling policies, consumer protection, and regulation of online gambling sites," *Computers in Human Behavior*, vol. 29, 2013.

[3] "Overview of the british gambling sector - gambling commission." [Online]. Available: https://www.gamblingcommission.gov.uk/manual/annual-report-and-accounts-2020-to-2021/annual-report-20-21-performance-report-overview-of-the-british-gambling

[4] S. Haber and W. S. Stornetta, "Secure names for bit-strings," in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, 1997, pp. 28–35.

[5] H. Massias, X. S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirement," in *the 20th Symposium on Information Theory in the Benelux*. Citeseer, 1999.

[6] R. C. Merkle, "Protocols for public key cryptosystems," in *Secure communications and asymmetric cryptosystems*. Routledge, 2019, pp. 73–104.

[7] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–16. [Online]. Available: https://doi.org/10.1145/2976749.2978341

[8] W. Dai, "b-money," 1998. [Online]. Available: http://www.weidai.com/bmoney.txt

[9] "Ethereum (eth) blockchain explorer - top 25 miners by blocks." [Online]. Available: https://etherscan.io/stat/miner

[10] "Ethereum (eth) mining pool." [Online]. Available: https://ethermine.org/

[11] J. Li, N. Li, J. Peng, H. Cui, and Z. Wu, "Energy consumption of cryptocurrency mining: A study of electricity consumption in mining cryptocurrencies," *Energy*, vol. 168, 2019.

[12] V. Buterin, "A next-generation smart contract and decentralized application platform," 2014.

[13] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, "Decentralized applications: The blockchain-empowered software system," *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018.

[14] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[15] T. Tani, "Ethereum evm illustrated," Mar 2018. [Online]. Available: https://github.com/takenobu-hs/ethereum-evm-illustrated

[16] "2021 dapp industry report." [Online]. Available: https://dappradar.com/blog/2021-dapp-industry-report

[17] E. T. Noone, "Chuck-a-luck: Learning probability concepts with games of chance," *The Mathematics Teacher MT*, vol. 81, no. 2, pp. 121 – 123, 1988. [Online]. Available: https://pubs.nctm.org/view/journals/mt/81/2/article-p121.xml

[18] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," 2017.

[19] J. Kanani, A. Arjun, S. Nailwal, and M. Bjelic, "Polygon lightpaper," *Polygon*, 2021.

[20] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz *et al.*, "Chainlink 2.0: Next steps in the evolution of decentralized oracle networks," 2021.

[21] D. Papadopoulos, D. Wessels, S. Huque, M. Naor, J. Včelák, L. Reyzin, and S. Goldberg, "Making nsec5 practical for dnssec," Cryptology ePrint Archive, Report 2017/099, 2017, https://ia.cr/2017/099.

[22] P. J. Piasecki, "Gaming self-contained provably fair smart contract casinos," *Ledger*, vol. 1, pp. 99–110, Dec. 2016. [Online]. Available: http://ledger.pitt.edu/ojs/ledger/article/view/29

[23] S. SEC, "2: Recommended elliptic curve domain parameters," *Standards for Efficient Cryptography Group, Certicom Corp*, vol. 5, 2000.

[24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 313–314.

[25] K. Sekniqi, D. Laine, S. Buttolph, and E. G. Sirer, "Avalanche platform whitepaper," 2020. [Online]. Available: https://www.avalabs.org/whitepapers

[26] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," 6 2019. [Online]. Available: http://arxiv.org/abs/1906.08936

[27] "Stake.com: Bitcoin casino & sports betting." [Online]. Available: https://stake.com

[28] K. Poduszló, "Introduction to provably fair gaming algorithms (5th draft)," 2017.

[29] decentral.games team, "decentral.games whitepaper," 2019.

[30] "Ridotto - reinventing the gambling experience," 2022. [Online]. Available: https://ridotto.io

[31] "Sunrise gaming by dao," 2021. [Online]. Available: https://sunrisegaming-dao.com

[32] "Croissant games - croissant games." [Online]. Available: https://croissant-games.gitbook.io/croissant-games

[33] "Eip-712: Ethereum typed structured data hashing and signing." [Online]. Available: https://eips.ethereum.org/EIPS/eip-712

[34] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," *Whitepaper*, 2018.

[35] "Proof of history: A clock for blockchain — solana - medium." [Online]. Available: https://medium.com/solana-labs/proof-of-history-a-clock-for-blockchain-cf47a61a9274

[36] "Solana validators — www.validators.app." [Online]. Available: https://www.validators.app/?network=mainnet&order=stake

[37] "Polygon web wallet v2 - staking." [Online]. Available: https://wallet.polygon.technology/staking

[38] "Whitepaper — axie infinity." [Online]. Available: https://whitepaper.axieinfinity.com

[39] "Community alert: Ronin validators compromised." [Online]. Available: https://roninblockchain.substack.com/p/community-alert-ronin-validators

[40] "Polygon — $matic (@0xpolygon) / twitter." [Online]. Available: https://twitter.com/0xpolygon

[41] "Solidity programming language." [Online]. Available: https://soliditylang.org

[42] M. Verma, "The study on blockchain-based library management and its characterization," 2021.

[43] "Ganache - truffle suite." [Online]. Available: https://trufflesuite.com/ganache/

[44] "Truffle - truffle suite." [Online]. Available: https://trufflesuite.com/truffle/

[45] "Mocha - the fun, simple, flexible javascript test framework." [Online]. Available: https://mochajs.org

[46] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[47] "Chuck a luck - wizard of odds." [Online]. Available: https://wizardofodds.com/games/chuck-a-luck

[48] "Access - openzeppelin docs." [Online]. Available: https://docs.openzeppelin.com/contracts/3.x/api/access

[49] "Polygonscan." [Online]. Available: https://polygonscan.com

[50] "Why use alchemy - alchemy documentation." [Online]. Available: https://docs.alchemy.com/alchemy/introduction/why-use-alchemy

[51] "Your instant rpc gateway to polygon." [Online]. Available: https://polygon-rpc.com

[52] "Ethers.js." [Online]. Available: https://docs.ethers.io/v5

[53] "Ethereum alarm clock documentation." [Online]. Available: https://media.readthedocs.org/pdf/ethereum-alarm-clock-service/latest/ethereum-alarm-clock-service.pdf

[54] "Metamask: The crypto wallet & gateway to web3 blockchain apps." [Online]. Available: https://metamask.io

[55] "Vue.js - the progressive javascript framework — vue.js." [Online]. Available: https://vuejs.org

[56] "What is vuex? — vuex." [Online]. Available: https://cite.me/Gp7DulB

[57] "Webpagetest - website performance and optimization test." [Online]. Available: https://www.webpagetest.org

[58] M. Schofield, "Meaningfully judging performance in terms of user experience," *Weave: Journal of Library User Experience*, vol. 1, no. 4, 2016.

[59] "Http archive." [Online]. Available: https://httparchive.org/

[60] "Industry statistics - november 2021 - gambling commission." [Online]. Available: https://www.gamblingcommission.gov.uk/statistics-and-research/publication/industry-statistics-november-2021

[61] "Latest statistics on the uk gambling industry - the european ..." [Online]. Available: https://www.europeanbusinessreview.com/latest-statistics-on-the-uk-gambling-industry