Proyecto Final: Casino en Ethereum

Axel Cervantes Henry Martínez Víctor Sánchez Daniel Vargas

1. Introducción

Los casinos en línea suelen cometer distintos tipos de fraudes[1]. Los juegos basados en blockchains brindan una excelente alternativa para evitar el fraude, pues ofrecen probabilidades justas y demostrables, al mantener todo el historial de transacciones. Además, pueden ofrecer a los usuarios tarifas mínimas o nulas porque no hay una autoridad central que administre las ganancias y cobre por este servicio. Pese a lo anterior, los juegos basados en blockchains pueden generar ganancias ecónomicas. Por ejemplo, PoolTogether[2], es un juego de lotería sin pérdidas en la blockchain de Ethereum.

Ethereum es un sistema descentralizado y autónomo, pues no tiene un nodo[3] central que pudiera fallar. En su lugar, Ethereum se ejecuta desde miles de computadoras de voluntarios en todo el mundo, lo cual hace casi imposible desconectar la red y permite que la información personal de los usuarios permanezca en sus propias computadoras, pero el contenido, como aplicaciones, videos, etc., se mantenga bajo control total de sus creadores sin tener que obedecer las reglas impuestas por los servicios de alojamiento como la App Store y los múltiples servicios de Google. La plataforma Ethereum aprovecha todas las propiedades de la tecnología blockchain en la que se ejecuta. Es completamente inmune a las intervenciones de terceros, lo que significa que todas las DApps y DAOs implementadas dentro de la red no pueden ser controladas por un solo ente. Es por esto que decidimos desarrollar una DApp en Ethereum con diversos juegos de casino(ruleta, lotería keno y máquina tragamonedas), donde los usuarios pueden apostar *ether* de manera segura.

2. Desarrollo

2.1. Herramientas y Software

Escribimos los contratos del proyecto en **Solidity**[4], un lenguaje de programación de alto nivel diseñado específicamente para hacer *smart contracts* de Ethereum de manera relativamente sencilla. Utilizamos **Remix**[5] como IDE para escribir, depurar y probar los *smart contracts* directamente desde el navegador. También usamos **Metamask**[6], una extensión del navegador que facilita el manejo de *Ether* entre *DApps* y usuarios de Ethereum. Esta extensión inyecta la API Ethereum Web3 en el ambiente de JavaScript de cada sitio web para que las *DApps* puedan leer la blockchain, que para

[1] https://www.legitgamblingsites.com/blog/online-casino-scams-be-aware-of/

[2] https://www.coindesk.com/this-ethereum-lottery-perfectly-explains-how-facebooks-corporate-backers-will-profit-from-libra

[3] Los nodos son voluntarios que descargan toda la blockchain de Ethereum en sus computadoras y hacen cumplir todas las reglas de consenso del sistema, manteniendo la red honesta y recibiendo recompensas a cambio.

[4] https://solidity.readthedocs.io/en/latest/

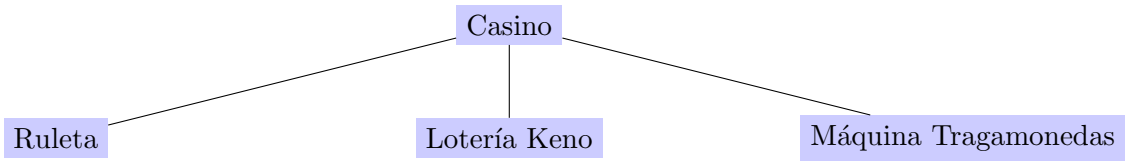
[5] http://remix.ethereum.org/

[6] https://metamask.io/

este proyecto, es la red de prueba **Ropsten**[7]. Todos los contratos se despliegan con el entorno de desarrollo de *DApps* **Truffle**[8] y por medio de la API **web3.js**[9], se generan instancias de éstos sobre la infraestructura de **Infura**[10] en la red de Ropsten, permitiendo la interacción con ellos. Por otro lado, se utilizó **Handlebars**[11] para acceder de manera más sencilla a las funciones del entorno de ejecución **Node.js**[12] y todas las vistas se hicieron con el *framework* de front-end **Bootstrap**[13].

2.2. Estructura del proyecto

El Casino es una *Dapp* que consiste de tres *smart contracts*, como se muestra en el siguiente diagrama.



Cada *smart contract* representa un juego de casino diferente y es accesible desde una aplicación web montada sobre Infura en la blockchain de prueba Ropsten. A continuación, se describen todos los juegos.

2.2.1. Ruleta

La ruleta americana se juega con una rueda con 38 casillas enumeradas como 00, 0, 1, 2, 3, ..., 36, cuyos colores están alternados entre rojo y negro. El crupier gira la ruleta en una dirección y tira una pelota sobre ella en la otra dirección. Los jugadores hacen apuestas tratando de adivinar si la bola caerá sobre cierto número, sobre una casilla de cierto color, sobre un número par o impar, entre otras[14].

En nuestra implementación, la ruleta es un *smart contract* que genera un número aleatorio entre 1 y 36, donde cada número tiene asignado un color rojo o negro. El jugador puede apostar a que saldrá un número en particular, un número rojo o uno negro, o un número mayor o menor a 18, y gana una cantidad variable de *Ether* dependiendo del tipo de apuesta que realizó y de si su predicción fue correcta o no.

Para lograr esto, definimos un nuevo tipo de dato `Bet` utilizando `structs` donde almacenamos datos importantes sobre la apuesta como: el usuario que la realiza, la cantidad en *wei*, el tipo (color, número exacto o mayor/menor que), entre otras cosas. Las casillas de la ruleta fueron almacenadas en dos arreglos, cada uno representando un color y cuyos valores eran los números de la ruleta.

En cuanto al funcionamiento del código, destacamos las funciones `wager` y `spin`. La primera función se encarga de llenar la estructura `Bet` con datos proporcionados por el usuario y la segunda simula el movimiento de la ruleta eligiendo un número aleatorio y le notifica al usuario el resultado y su premio.

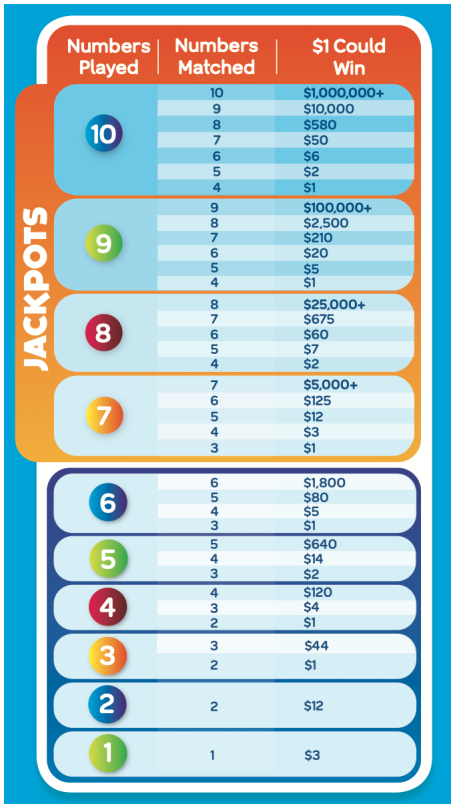
[7] https://ropsten.etherscan.io/
[8] https://www.trufflesuite.com/
[9] https://web3js.readthedocs.io/en/v1.2.9/
[10] https://infura.io/
[11] https://handlebarsjs.com
[12] https://nodejs.org/
[13] https://getbootstrap.com/
[14] http://www.casinocity.com/rule/roulette.htm

2.2.2. Keno (lotería)

En el Keno tradicional, el jugador compra un boleto donde selecciona de 1 a 10 números entre 1 y 80. Luego, el encargado del juego genera 20 número aleatorios sin reemplazo y se otorgan premios al jugador dependiendo de cuántos números de los que eligió coinciden con los obtenidos aleatoriamente[15].

En nuestra implementación, al momento de desplegar el contrato, se empieza a generar un arreglo con 20 números aleatorios entre 1 y 80, marcando el inicio de una nueva ronda a la que se puede unir cualquier jugador mediante la función `entrarAlJuego`, ingresando los números de su elección.

Para que se puedan consultar los resultados, es necesario que se hayan generado los 20 números aleatorios y que las apuestas de todos los jugadores sumen al menos de 0.02 ETH (lo suficiente para cubrir los costos de la generación de los números aleatorios y tener una ganancia). Una vez que se cumplen estas condiciones, los usuarios pueden consultar sus resultados mediante la función `consultarResultado`, la cual se encarga de verificar las coincidencias entre los números que jugó el usuario y los obtenidos aleatoriamente, y enviar su premio al jugador de acuerdo con la siguiente tabla[16]:



Numbers Played	Numbers Matched	$1 Could Win
10	10	$1,000,000+
	9	$10,000
	8	$580
	7	$50
	6	$6
	5	$2
	4	$1
9	9	$100,000+
	8	$2,500
	7	$210
	6	$20
	5	$5
	4	$1
8	8	$25,000+
	7	$675
	6	$60
	5	$7
	4	$2
7	7	$5,000+
	6	$125
	5	$12
	4	$3
	3	$1
6	6	$1,800
	5	$80
	4	$5
	3	$1
5	5	$640
	4	$14
	3	$2
4	4	$120
	3	$4
	2	$1
3	3	$44
	2	$1
2	2	$12
1	1	$3

Cabe mencionar que, una vez que un jugador consulta sus resultados, ya no se aceptan nuevos jugadores en la ronda y que cuando todos los jugadores de la ronda revisaron sus resultados, se empieza a generar otro arreglo de 20 números aleatorios y empieza una nueva ronda.

[15] https://www.keno.com.au/keno-pdfs/NSW_Game%20Guide.pdf
[16] https://www.keno.com.au/keno-pdfs/NSW_Game%20Guide.pdf

2.2.3. Máquina tragamonedas

En las máquinas tragamonedas clásicas, se tiene una pantalla con tres ruedas con imágenes que giran cuando se activa el juego. Una vez que las ruedas se detienen, detectan las tres imágenes obtenidas y el jugador puede ganar premios dependiendo de los resultados que haya obtenido[17].

En nuestra implementación, existe una función `jugar` que le cuesta exactamente 0.02 *Ether* al jugador. Cuando se ejecuta esta función, se empiezan a generar tres números aleatorios entre 1 y 5. Una vez que se generan estos números, el jugador puede consultar los resultados y recibir su premio con la función `consultarResultado`, la cual calcula dicho premio de acuerdo con la siguiente tabla:

Tabla de relación de resultados y premios		
Números iguales	Valor de los números	Premio (ETH)
0	-	0
2	Los que sean	0.01
3	1	0.25
	2	0.5
	3	1.0
	4	2.0
	5	5.0

2.3. Generación de números aleatorios

Originalmente utilizábamos un *blockhash* para generar los números aleatorios, como se muestra en el siguiente código.

```
1 bytes32 random = keccak256(abi.encodePacked(blockhash(bet.block),id));
```

Sin embargo, esto hace al contrato vulnerable, pues la técnica se basa en el hash de direcciones anteriores en la blockchain, y así, un minero podría calcular el *blockhash* y anticipar el resultado. Al hacer esto, este minero podría tratar de influir en el resultado de una transacción en el bloque actual. Esto resulta especialmente fácil si solo hay un pequeño número de resultados igualmente probables.

Para evitar esta vulnerabilidad, utilizamos la API de Provable para Ethereum[18] en los tres contratos. Provable funciona como un oráculo para obtener diversos tipos de información externa en *smart contracts* y aplicaciones en blockchains. En nuestra aplicación, utilizamos el **Provable Random Data Source**[19], para generar números aleatorios de manera confiable y evitar previamente mencionadas, aunque con un costo de aproximadamente 0.004 ETH.

Otra ventaja de utilizar Provable es que utiliza una prueba de autenticidad que se incluye junto con el número aleatorio generado. Ésta se verifica dentro de los *smart contract*, para descartar cualquier resultado cuyas pruebas de autenticidad no pasen este proceso.[20], evitando que Provable altere los resultados aleatorios provenientes del TEE (ambiente seguro de ejecución) y protegiendo al usuario de *attack vectors*, como justifican en el documento: "A Scalable Architecture for On-Demand, Untrusted Delivery of Entropy"[21].

[17] https://en.wikipedia.org/wiki/Slot$_m achine$
[18] https://github.com/provable-things/ethereum-api/blob/master/provableAPI_0.6.sol
[19] https://docs.provable.xyz/data-sources-random
[20] https://docs.provable.xyz/#data-sources-computation
[21] https://provable.xyz/papers/random$_d atasource - rev1.pdf$

2.4. Despliegue de la Dapp Web

Para desplegar el contrato utilizamos la suite de desarrollo de *DApps* Truffle e Infura, las cuales permiten conectarse con la blockchain de Ethereum sin necesidad de tener una cartera virtual ni un nodo de Ethereum. Con Truffle, desplegamos los contratos en la red de pruebas Ropsten, de este modo, obtenemos una dirección del contrato dentro de la blockchain, el precio por la creación, el ether con el que se instanció el contrato, entre otros. Truffle nos ofrece la ventaja de utilizar los contratos como objetos, así que en cualquier momento pudimos volver a desplegar los contratos con cambios fácilmente y realizar distintas pruebas antes del despliegue.

Para poder instanciar el contrato en la web, utilizamos la biblioteca web3, que es un framework para interactuar con contratos inteligentes (lectura de variables públicas del contrato, llamadas a funciones o transferencia de ether). Como web3 contiene un constructor de instancias de contratos, basta con crear un objeto tipo Contract con la dirección del contrato y la ABI (Application Binary Interface) para definir completamente el contrato. Adicionalmente, web3 contiene la wallet Metamask que permite interactuar con funciones de los contratos que implican transferencias y costos de gas.

Las vistas se generaron con la biblioteca de HTML, CSS y JS Bootstrap.

3. Conclusión

Las ventajas de los juegos montados sobre blockchains son, principalmente, el bajo costo de implementación y la seguridad que ofrecen a los jugadores de que todos los resultados son confiables y verificables. Éstas características se deben a la misma naturaleza de las blockchains donde todos los nodos tienen acceso a la misma información y se encargan de aprobar o descartar las transacciones, en vez de tener una sola autoridad que lo haga. Sin embargo, esto también puede dar problemas, sobre todo con la generación de números aleatorios confiables y con los tiempos de ejecución. Una solución popular para generar números aleatorios de manera segura y confiable es utilizar oráculos como Provable, pero el uso de métodos y funciones más sofisticadas como éste, hace que las transacciones sean más costosas y tomen más tiempo en agregarse a la blockchain. En el caso de Provable, la generación de un solo número aleatorio cuesta 0.004 ETH y requiere de una transacción adicional, lo cual hace que, en juegos como nuestro Keno, los costos y tiempos de ejecución se eleven mucho. Desgraciadamente, el problema de los tiempos de ejecución lentos en Ethereum aún no está resuelto, por lo que juegos de turnos como el blackjack o el póker serían extremadamente tardados, pero en juegos de una sola ronda como los presentados en nuestro proyecto, el funcionamiento es bastante similar al de los casinos centralizados.

Referencias

Iyer, K. (2018). Building Games with Ethereum Smart Contracts. Apress.