Universitat Oberta de Catalunya

uoc.edu

# Desarrollo de un mixer centralizado mediante firmas ECDSA en Ethereum.

**Álvaro Carvajal Castro**

Técnicas de privacidad en Ethereum.

Sistemas de blockchain

**Alberto Ballesteros Rodríguez**
**Victor Garcia Font**

01/2023

**FICHA DEL TRABAJO FINAL**

| | |
|---|---|
| **Título del trabajo:** | *Desarrollo de un mixer centralizado mediante firmas ECDSA en Ethereum.* |
| **Nombre del autor:** | *Álvaro Carvajal Castro.* |
| **Nombre del consultor/a:** | *Alberto Ballesteros Rodríguez.* |
| **Nombre del PRA:** | *Victor Garcia Font.* |
| **Fecha de entrega (mm/aaaa):** | *01/2023* |
| **Titulación o programa:** | Ciberseguridad y Privacidad. |
| **Área del Trabajo Final:** | *Técnicas de privacidad en Ethereum.* |
| **Idioma del trabajo:** | *Castellano.* |
| Palabras clave | *Mezcladores, Blockchain, Ethereum* |


**Resumen del Trabajo**

Documento a modo de guía sobre los mixers, explicar en qué consisten, cómo funcionan y por qué son herramientas interesantes en ciertas redes también se cubren ciertos aspectos éticos sobre el uso de estas herramientas hoy en día.

La metodología se basa en un primer lugar una explicación teórica, explicando el funcionamiento, el motivo de su existencia y los diferentes tipos de mezcladores que existen, todo esto acompañado de un desarrollo de un mixer en Solidity, para profundizar en el aspecto más técnico.

El mixer desarrollado tendrá una modalidad centralizada y hará uso de firmas ECDSA para comprobar la autenticidad de los depósitos. Desarrollado en Go y Solidity.

Como conclusión se obtendrá un documento a modo de guía con una explicación detallada sobre el funcionamiento de los mixers.

**Abstract**

Document as a guide on mixers, explaining on what they consist, how they work and why they are interesting tools in certain networks, this document also covers certain ethical aspects related to the use of this tools.

The methodology is based first on a theoretical explanation, covering the

operation, the reason for its existence and the different types of mixers that exist, all of this accompanied by the development of a real mixer in Solidity, to delve into the more technical aspect.

The mixer developed will have a centralized approach and will use ECDSA signatures to verify the authenticity of the deposists. Developed in Go and Solidity.

As the final product the end user will obtain a document as a guide with a detailed explanation of the operation of mixers.

# Índice de contenidos

# Lista de figuras

# 1. Introducción

## 1.1. Contexto y justificación del Trabajo

Los mezcladores/mixers son herramientas que tienen gran relevancia en el mundo Blockchain donde toda transacción es pública, por lo que la identificación de individuos es una tarea sencilla. Esta característica es para muchos usuarios un inconveniente en ciertos casos a la hora de buscar privacidad en algunas redes (por ejemplo Bitcoin y Ethereum).

Son cada vez herramientas más relevantes, ya que el uso de estas redes está en aumento y al tratarse de un libro público de transacciones, sus usuarios buscan soluciones para proteger su privacidad y en muchos casos su seguridad.

Es necesario dar a conocer estas herramientas y/o protocolos al público para que los usuarios sepan que herramientas tienen a su alcance para poder lograr un cierto grado de anonimato en este tipo de redes. Además explicar su funcionamiento puede ayudar a ciertos usuarios a entender lo que realmente están usando o el motivo por las cuales algunas redes presentan como característica principal el anonimato.

Importante mencionar que brindar un buen servicio de mezcla no es una tarea sencilla, en muchos casos una incorrecta implementación puede resultar en un grado de anonimato menor al esperado o incluso perder su efecto por completo, como se discutirá más adelante en otros capítulos.

Por último, añadir la importancia de desmitificar que los mezcladores solo se necesiten para la realización de actividades delictivas. Existen múltiples usos hoy en día que justifican su existencia.

## 1.2. Objetivos del Trabajo

El objetivo principal es la explicación tanto a nivel teórico y práctico de los mixers, cómo funcionan y por qué son importantes actualmente. Además de explicar posibles casos de usos que pueden ser de interés para cualquier usuario de la red.

Otro objetivo adjunto al principal es de desarrollar un mixer en Ethereum. Este objetivo permite comprender el funcionamiento interno de este tipo de protocolos.

Por último, analizar la posibilidad de censura de estas herramientas (el caso más notable actualmente el de Ethereum) y que consecuencias puede acarrear para todos los usuarios de la red, además mencionar lo ocurrido en un caso más concreto, tornado.cash.

### 1.3. Impacto en sostenibilidad, ético-social y de diversidad

El mayor impacto en sostenibilidad suele darse en la red y en como se mantenga su seguridad (por ejemplo protocolos Proof of Work vs. Proof of Stake), algo que está fuera de esta entrega.

Sin embargo, el impacto ético-social sí tiene cabida dentro de los mixers. Este tipo de herramientas se suelen asociar muchas veces a un uso delictivo o fraudulento, generalmente para entorpecer la traza de fondos de fuentes desconocidas.

En el primer capítulo se cubrirán casos de uso más allá de lo mencionado con anterioridad, con el fin de acabar con la percepción social que los mixers actualmente tienen, por ejemplo, el anonimato a la hora de realizar donaciones [1].

### 1.4. Enfoque y método seguido

La estrategia principal es la explicación detallada, a modo de guía, sobre el funcionamiento de los mezcladores, la explicación técnica se basa en mayor parte en el desarrollo de la aplicación de muestra de un mixer en Solidity.

El desarrollo de una aplicación de muestra es un pilar fundamental para lograr los objetivos, ya que resultará sencillo ir paso por paso sobre el código obtenido para analizar como funciona un mixer.

Se ha optado por desarrollar desde cero en lugar de estudiar un producto ya creado, en primer lugar por la complejidad (casos como tornado.cash resultan más complejos de estudiar sin tener amplios conocimientos previos).

## 1.5. Planificación del Trabajo

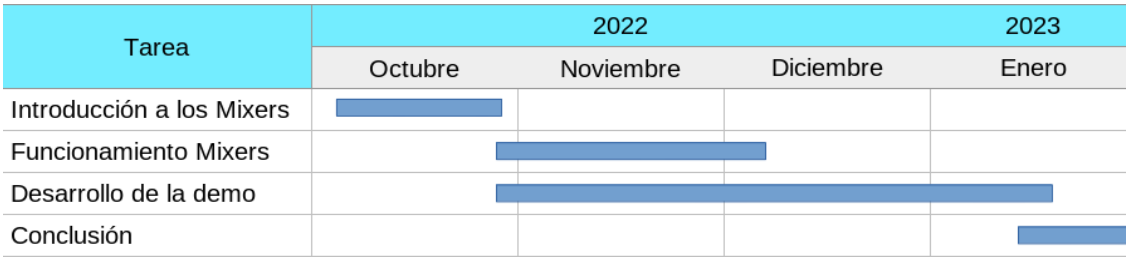



**Figure 1: Diagrama Gantt de la planificación de los bloques.**

Para la planificación del trabajo, cada bloque se ha dividido en su propia tabla de periodización, enumerando las tareas a realizar.

Al tratarse de tres entregas PEC y el trabajo estar dividido en cuatro bloques fundamentales, una entrega contendrá el trabajo de dos bloques.

*Primer bloque – Introducción a los Mixers.*

| PEC 2 – Introducción mixers. |
|---|
| Explicación breve. |
| Casos de uso. |
| Cómo surgen/de donde surgen. |
| En qué redes tienen sentido. |
| Problemas de funcionamiento. |

*Segundo bloque – Funcionamiento de los Mixers.*

| PEC 3 – Funcionamiento mixers. |
|---|
| Funcionamiento. |
| Censura y tornado.cash (bloque 4). |

*Tercer bloque – Desarrollo en Solidity con Hardhat*

| PEC 4 – Desarrollo de la demo. |
|---|
| Desarrollo contratos. |
| Desarrollo front-end. |

*Cuarto bloque – Censura y tornado.cash*

| PEC 4 – Conclusión. |
| --- |
| Sanciones OFAC y MEV-Boost. |
| Censura y tornado.cash. |

Como se puede observar, el último bloque se realiza en la PEC3 junto con el tercer bloque, el tercero es una parte importante del trabajo y el cuarto se considera una conclusión final que no tiene el mismo peso que su bloque anterior.

Se incorpora este último apartado también en la tabla de tiempo del segundo bloque, donde se podrá empezar a definir el apartado y finalmente acabarlo en la siguiente PEC.

### 1.6. Breve sumario de productos obtenidos

El producto final será una guía detallada sobre el funcionamiento de los mixers. Detallando casos de uso, funcionamiento y características.

Mediante una prueba de concepto se mostrará el funcionamiento de un mixer en Solidity, el cual podrá ser puesto en funcionamiento en alguna red compatible con la máquina virtual de Ethereum, con el fin de lograr una mayor compresión de los mixers.

La prueba de concepto será realizada con el entorno de desarrollo Hardhat, donde se incluirán los contratos desarrollados, los tests y algunos scripts necesarios para su puesta en marcha.

### 1.7. Breve descripción de los otros capítulos de la memoria

Los capítulos se han dividido en cuatro bloques:

1. **Introducción a los mixers:** Se expone una breve explicación (en qué consisten brevemente) y algunos problemas que puede tener su posible implementación:
    1. Casos de uso.

    2. Cómo surgen/de donde surgen.
    3. En qué redes tienen sentido y en cuáles no.
    4. Problemas con los servicios de mezcla centralizados y de funcionamiento.
2. **Análisis de su funcionamiento:** Explicar sobre el funcionamiento concreto, nivel teórico.
    1. Funcionamiento a nivel teórico y de manera general, sin mencionar ningún protocolo en concreto.
3. **Desarrollo del mixer de ejemplo.**
4. **Censura y futuro de los mezcladores:** Revisión de algunos temas de actualidad que se consideran relevantes (tornado.cash) y las posibles censuras que pueden ocurrir en Ethereum.

Todos los bloques tienen relación entre sí, la introducción es necesaria para entender su funcionamiento (además de sus problemas) y para más adelante poder seguir el desarrollo. Con estos tres bloques se espera explicar completamente este tipo de herramientas y por qué son útiles y necesarias.

El último bloque trata algunos temas de actualidad que son necesarios para entender los riesgos que se presentan por la posible ausencia de estas opciones de privacidad. En concreto se cubrirán las recientes censuradas ocurridas en la red de Ethereum, centrándose en el caso del mixer tornado.cash.

# 2. Materiales y métodos

Para los primeros capítulos la metodología empleada es la misma, primero se realiza una serie de sesiones de búsqueda de información además de obtener información sobre los diferentes mezcladores de código abierto que existen.

- Para el estudio de las diferentes técnicas o métodos empleados para medir la trazabilidad de las transacciones se ha ayudado de la publicación “On How Zero-Knowledge Proof Blockchain Mixers Improve, and Worsen User Privacy” [9].

- Para el desarrollo del mezclador, lo más relevante es entender realmente cómo funcionan este tipo de herramientas, para ello la información buscada se encuentra sobre todo en repositorios git, entre ellas se encuentran MicroMix [6] y tornado.cash [8].

- El desarrollo de un mixer en Solidity se divide en dos fases, una primera donde se trabaja sobre un proyecto en Hardhat, desarrollando los smart contract y algunos tests. Una segunda fase donde se creará una sencilla página web a modo de front-end para los contratos, desarrollada en Go.

Se obtendrán un total de dos productos: Un primer producto será este mismo documento, a modo de guía explicando el funcionamiento de los mixers. Un segundo producto será el mezclador desarrollado en Solidity a modo de prueba.

# 3. Introducción a los mixers

## 3.1 Explicación general

Los mixers o mezcladores son herramientas o servicios que permiten a sus usuarios mezclar sus criptomonedas con las de otros usuarios que estén participando.

Tras mezclar las monedas, el usuario obtendrá la misma cantidad que ha depositado (aunque es posible que el mixer cobre una tarifa). La dirección final no tendrá ninguna relación con la dirección usada inicialmente.

Resulta muy complejo seguir el rastro de las monedas que el usuario ha depositado, la conexión entre el origen y el destino queda oculta.

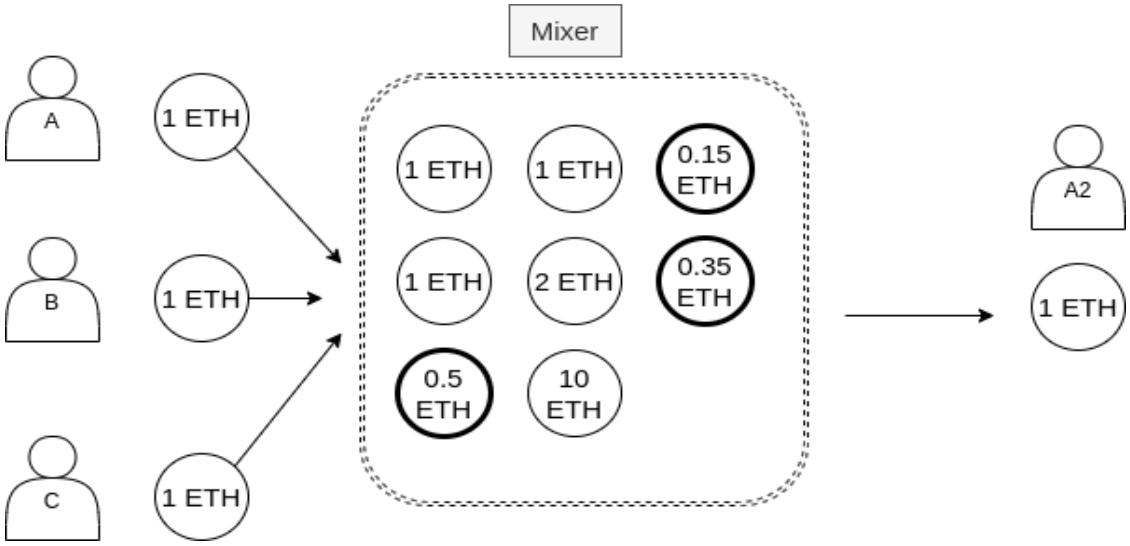



**Figure 2: Esquema funcionamiento de un mixer.**

En la figura anterior se puede ver el funcionamiento general de un mezclador, existen usuarios que van depositando fondos en el mixer (puede ser un contrato o una entidad centralizada por ejemplo). Una vez hay suficientes fondos, los usuarios pueden retirar sus monedas, la cantidad será la misma, pero estará compuesta por una mezcla de distintas cantidades y de distintos orígenes.

El modo de retirada puede ir variando según la implementación del mixer que estemos usando en ese momento, un método muy común es el de generar un mensaje secreto una vez hemos depositado.

Más adelante tendremos que insertar ese mensaje secreto para poder retirar, por ejemplo podemos dar el mensaje secreto a otra persona para que la retirada sea suya.


**Figure 3: Retirada en tornado.cash.**

Como se observa en la figura, el usuario A deposita 1 ETH y más adelante obtiene su 1 ETH de vuelta (en otra nueva dirección), pero este está compuesto por 0.15 ETH, 0.25 ETH y 0.5 ETH, haciendo que trazar el 1 ETH que se ha depositado con anterioridad sea una tarea muy compleja.

Generalmente estos servicios se llevan una pequeña comisión variable, haciendo que las cantidades tampoco coincidan, dificultando aún más su trazabilidad.

Existen dos tipos de mezcladores, los centralizados, los cuales funcionan de igual manera, pero existe una entidad central, la cual procesa los depósitos y los retiros, es decir, todo pasa por un punto central. Normalmente un servidor central se encargará de realizar las transacciones necesarias, al estar todo centralizado esta opción tiene graves problemas de privacidad para el usuario.

La otra modalidad son los mezcladores descentralizados, generalmente funcionan gracias a un smart contract, el cual es el encargado de realizar todas las funciones del mixer y no tiene una entidad central que pueda desaparecer o no cumplir su palabra.

Un mezclador muy popular de esta modalidad es tornado.cash, el cual es en esencia un smart contract en Ethereum.

### 3.2 Casos de uso de los mixers

Este tipo de herramientas resultan muy útiles para mantener cierto nivel de anonimato a la hora de realizar transacciones en una red. Generalmente se han asociado a la realización de actividades ilegales, para ocultar el destino de fondos fraudulentos, pero existen casos de uso que van más allá:

Aceptar cobros en criptomonedas por servicios (por ejemplo, un sueldo), pero a la hora de dar una dirección para que el usuario haga la transferencia, podrá ver todos los movimientos de dicha dirección.

Lo mismo ocurre si se realizan pagos, el usuario destino puede realizar las mismas observaciones, por ejemplo, queremos donar a alguna causa, pero no queremos dar a conocer todos nuestros movimientos en la red, tampoco queremos que se nos asocie con la donación en cuestión.

Un mezclador juega aquí un papel importante, si el usuario paga mediante algún servicio de mezcla no podrá conocer todos los movimientos pasados y futuros.

A continuación, se muestran algunos casos de uso comunes en el uso de mezcladores y ninguno relacionado con temas fraudulentos:

- Realizar pagos o donaciones de manera anónima.
- Aceptar un salario en criptomonedas.
- Una dirección ha sido “descubierta” y el usuario quiere volver a ser anónimo.

En este último caso, si la dirección fuese descubierta, es decir, asociada a una persona física, se necesitaría hacer “reset”, en este caso los mixers son una herramienta muy útil.

En un posible escenario donde las criptomonedas tengan más adopción a nivel global, si se asocia una dirección a una persona es posible ver todos los movimientos que ha realizado con sus monedas, desde el inicio, por ejemplo, haciendo uso de algún explorador, por ejemplo, Etherscan [7].

### 3.3 Porque surgen los mezcladores

En la blockchain todos los nodos poseen una copia de todas las transacciones que han ocurrido, con sus cantidades, fuentes y destinos (aunque esto puede variar entre redes, este trabajo se centra en la red Ethereum).

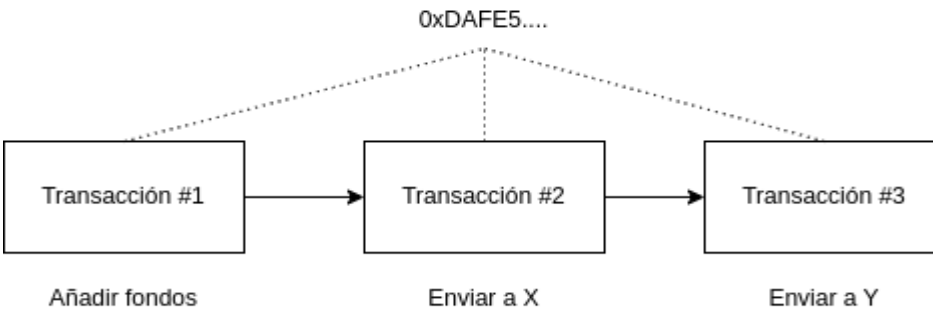



**Figure 4: Misma dirección para todas las transacciones.**

Una dirección no tiene ninguna relación con un nombre, por lo que de forma general no es posible asociarla a ninguna persona física.

Con exploradores como Etherscan [7] es posible fácilmente visualizar las transacciones que una dirección ha realizado, este tipo de herramientas resultan útiles a la hora de visualizar actividad en la red (bloques generados, direcciones, interacciones con smart-contracts).

Debido a este tipo de herramientas se puede apreciar el tipo de anonimato al que un usuario está expuesto, por un lado, una dirección no lleva asociado ningún nombre, pero si se logra crear esta unión resulta entonces sencillo poder ver toda la actividad de esa persona en la blockchain:

| Txn Hash | Method | Block | Age |
|---|---|---|---|
| 0x7a5f51fea0b7495ab72… | Transfer | 15563328 | 50 days 16 hrs ago |
| 0x395b5993dff11aba777… | Transfer | 15562599 | 50 days 18 hrs ago |
| 0x089e7b3b1fb8afa2af0… | Set Name | 15387436 | 78 days 12 hrs ago |
| 0xd960345ad2ab5ebc3c… | Register With Co... | 15387420 | 78 days 12 hrs ago |
| 0xb9f02336aae0047c92f… | Commit | 15387399 | 78 days 13 hrs ago |
| 0x95fb3e9ade2d4a52b4… | Transfer | 15085388 | 125 days 15 hrs ago |

**Figure 5: Explorador de bloques Etherscan**

Al tener todas las transacciones públicas y visibles a todo el mundo, si en algún caso se logra asociar una dirección a una persona física, resultará muy sencillo poder ver todas sus transacciones.

De este modo surgen los mixers, herramientas útiles para quienes quieren permanecer anónimos, algo que no es posible en muchas de las redes que más uso tienen hoy en día (por ejemplo en Ethereum y Bitcoin, aunque en esta última existen más herramientas que ayudan a lograr un nivel de anonimato).

### 3.4 En qué redes tienen sentido

Generalmente, este tipo de herramientas tendrán sentido en redes las cuales no tienen ninguna herramienta intrínseca de anonimato a la hora de hacer transacciones.

Hay que recordar que Bitcoin ni Ethereum nacen para ser anónimas, nacen como un medio en el que poder hacer transferencias sin un intermediario. Por lo que en este tipo de redes sí tiene sentido que este tipo de herramientas existan.

También hay que añadir, que en Bitcoin es posible tener cierto grado de anonimato, existe la generación de nuevas direcciones, con lo que es posible no recibir futuras transacciones a la misma dirección.

Además existen ciertas redes que prestan a sus usuarios ciertas medidas para ser más o menos anónimos, por ejemplo, Monero (Stealth Address, Signaturas Ring), Zcash (zk-SNARKs), Litecoin (MWEB), etc.

### 3.5 Problemas de funcionamiento

Usar un mixer no asegura que la dirección final sea completamente anónima, es posible que tras usar un mixer aún se puedan trazar las monedas que han sido depositadas.

En primer lugar, si se usa un mixer centralizado existe un gran riesgo, la entidad central puede guardar de alguna manera ciertos «logs» de las acciones que se han realizado en el mezclador.

Con entidades centralizadas, datos como la dirección de entrada y salida, dirección IP o un fingerprint del dispositivo que ha realizado las interacciones) pueden ser guardadas.

Con mezcladores descentralizados generalmente no existen los mismos problemas que con una entidad central, pero, aun así, su uso no es garantía de anonimato. Generalmente la trazabilidad tras usar un mezclador se basa en medidas heurísticas, las cuales intentan “averiguar” la relación entre direcciones, algunos ejemplos como MixEth [3], tornado.cash [4] o MicroMix [6].

En primer lugar, se debe usar un nodo privado y no conectarse a un nodo público. En ningún momento se puede conocer que datos están siendo grabados en el nodo público.

Existen ciertos filtros heurísticos que se pueden aplicar para intentar relacionar transacciones tras el paso de un mezclador:

- Relacionar los tiempos en las que fueron realizadas las transacciones.
- Relacionar las cantidades de entrada y salida.
- Intentar buscar patrones similares con la misma dirección de entrada que ha interactuado con el mezclador.
- Examinar la cantidad de entrada y buscar salidas con la misma cantidad.

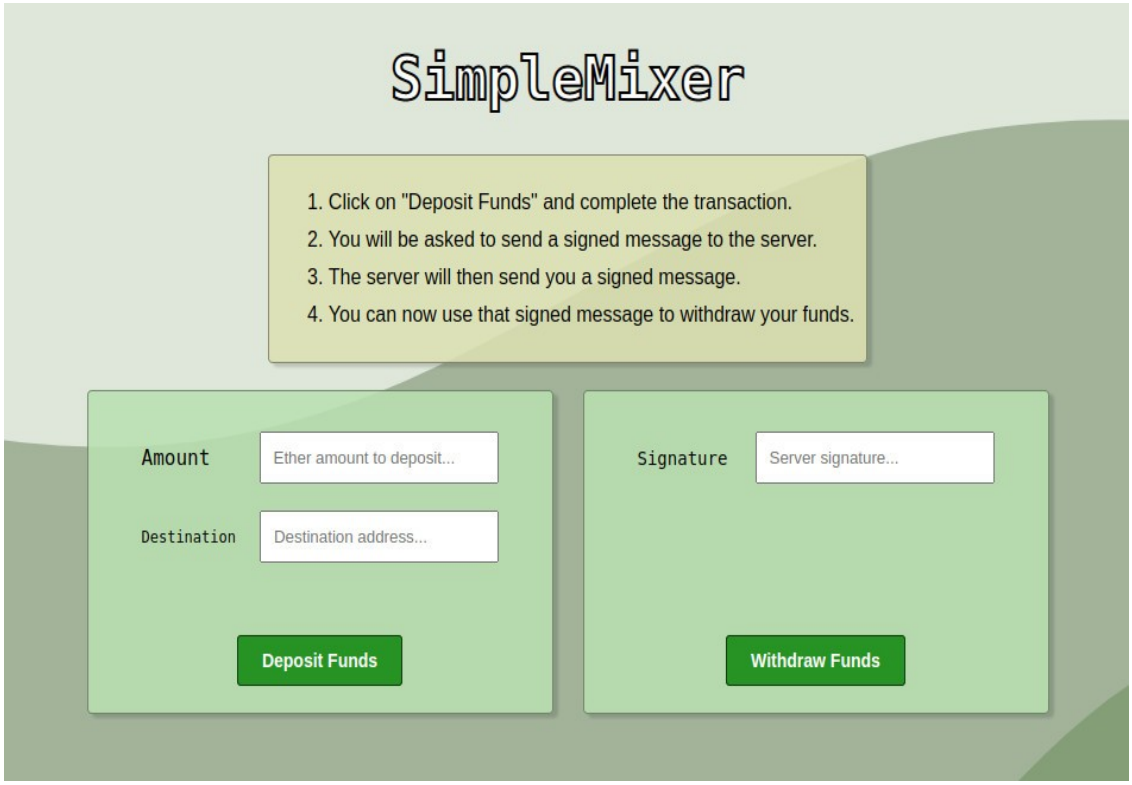
- Examinar los bloques en los que fueron incluidas las transacciones.
- Determinar si el usuario ha usado algún nodo público que pueda dar información (Infura, etc.).

La mayoría de estos filtros se basan en la aleatoriedad del mezclador, cuanto más aleatorios sean los resultados, más complicado resulta relacionar las direcciones de entrada y salida.

Por ejemplo, no usar las mismas cantidades, cargar una tasa aleatoria, no hacer las transacciones al mismo tiempo, usar diferentes valores para el gas de cada transacción, usar un nodo propio para interactuar con la red.

Para finalizar, si se utiliza un mezclador el cual proporcione un buen nivel de aleatoriedad, dará como resultado una serie de transacciones las cuales serán complicadas de trazar.

# 4.Desarrollo del mezclador SimpleMixer




**Front-end del mezclador desarrollado SimpleMixer.**

Se trata de un mezclador centralizado, el usuario primero tendrá que depositar los fondos en el mezclador y contactar con la entidad central mandando una firma generada por la misma clave privada que ha realizado la transacción.

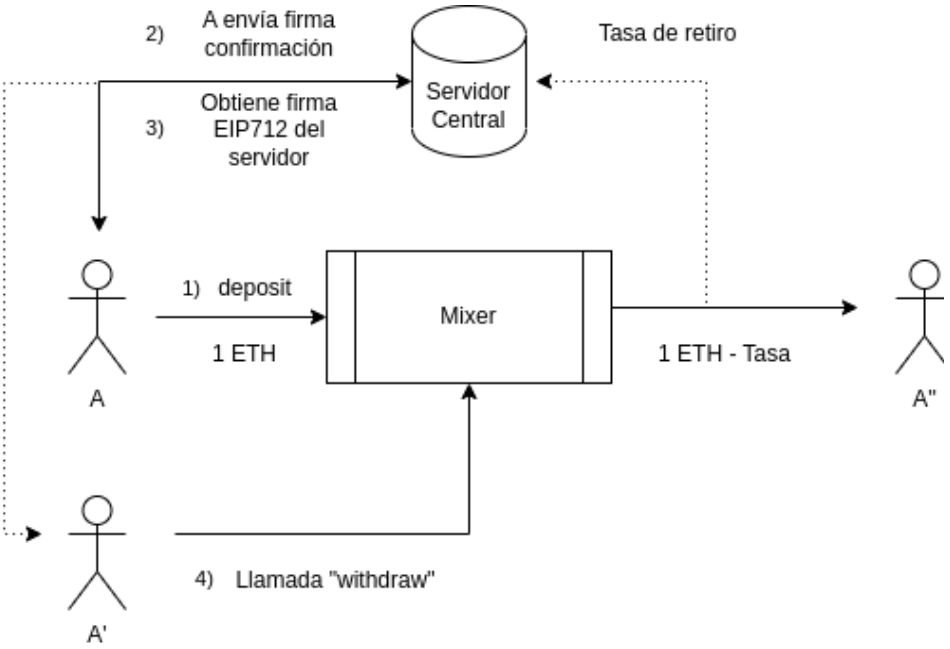



**Figure 6: Funcionamiento de SimpleMixer**

Esta firma generada por el servidor central sirve a modo de “llave” o “clave secreta” para poder obtener los fondos que han sido depositados con anterioridad.

Si la firma es correcta, el servidor central comprueba la transacción realizada y manda de vuelta al usuario otra firma, esta vez firmada con la clave privada de la entidad central, esta firma hace uso del estándar EIP-712 [10]. La firma tiene entonces la siguiente estructura:

| | |
|---|---|
| Separador | `\x19\x01` |
| Dominio | `EIP712Domain(string name,address verifyingContract)` |
| Struct | `WithdrawAction(uint256 amount,uint256 salt,address to)` |

El usuario podrá entonces interactuar con el contrato (método withdraw), mandando la estructura mencionada en la tabla y la firma obtenida, si todo es correcto (la clave pública de la firma coincide) entonces se mandarán los fondos mezclados.

Como se puede apreciar, no existe una relación directa en la blockchain entre la clave pública que ha depositado los fondos (con la función deposit) y la clave pública que ha interactuado con la función withdraw, pues estas pueden ser direcciones diferentes.

En este caso, solamente la entidad central conoce la relación entre las direcciones, ya que es el quien ha generado la firma final y puede comprobar que otra dirección la ha usado para retirar los fondos, además en la firma existe el campo “to” que también se podría guardar.

También existe el riesgo de que la entidad central desaparezca, en este caso cualquier usuario que no tenga su firma no podrá acceder los fondos sin la clave privada de firma central.

### 4.1 Elección entre centralizado o descentralizado

Para esta aplicación a modo de ejemplo, se ha decidido optar por un mezclador con un componente centralizado.

El desarrollo de un mezclador completamente descentralizado no es una tarea sencilla, ya que es necesaria alguna manera de comunicación donde se pueda compartir un secreto sin desvelar este y además tener la capacidad de comprobarlo sin tener acceso al secreto.

Generalmente este tipo de mezcladores usan pruebas Zero-Knowledge, en concreto emplean pruebas zk-SNARK [11] para llevar a cabo esta comunicación de la información secreta. Un concepto que no es trivial.

La opción de un mezclador con un componente centralizado no es una mala decisión, ya que disminuye la complejidad de este y hace que sea más sencillo entender el funcionamiento básico de como mezclar criptomonedas.

En el mundo real, siempre hay que tener en cuenta que a la hora de usar un mezclador centralizado existe el riesgo (como ya se ha explicado) de que la información sea guardada y realmente el usuario no sea anónimo.

### 4.2 Medidas de seguridad

En primer lugar, el usuario tiene que enviar un mensaje firmado al servidor central para confirmar que se tiene acceso a la clave privada que ha realizado la transacción.

```
bytes32 h = getWithdrawTypedDataHash(_action);
address pub = recover(h, _signature);
require(pub == authPublicKey, "Invalid provided
signature");
```

**Verificación de la firma en SimpleMixer.**

Para generar la firma del servidor, el usuario debe enviar la dirección final donde los fondos serán depositados, esto tiene un riesgo debido a la naturaleza centralizada del mezclador, donde el servidor puede guardar dicha información, en este caso ficticio no supone un riesgo.

#### 4.2.1 Prevención de front-running usando la Mempool

Es necesario que el usuario envíe la dirección final para generar la firma por parte del servidor, en caso contrario donde no fuese necesaria la dirección final, existen problemas de front-running.

Si el usuario pudiera mandar la firma y la dirección final (como argumentos a la función del contrato), otros usuarios tendrían acceso a esta información en la mempool de transacciones pendientes.

Es posible entonces copiar la información (firma), usar una dirección final distinta y mandar la misma transacción con más gas, haciendo que esta tenga más prioridad a la hora de ser añadida en un bloque final.

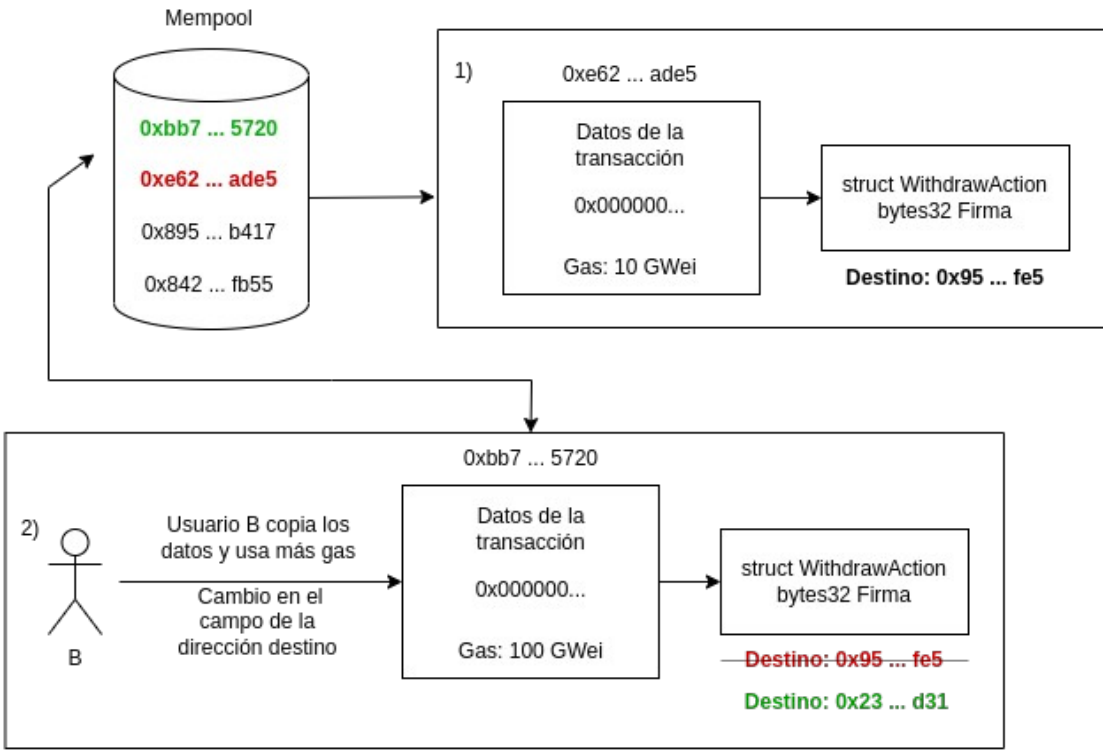



**Figure 7: Uso de la Mempool con SimpleMixer**

Es una limitación debido a la naturaleza centralizada del mezclador, la entidad central sabrá a donde serán destinados los fondos depositados.

El envío de la dirección final por parte del usuario (y más adelante en la comprobación de la firma) hace que cualquier usuario deba usar esa misma dirección final para recibir los fondos, lo que invalida por completo este tipo de ataques.

### 4.2.2 Prevención de ataques reentrancy

La firma generada por el servidor central, tiene un atributo llamado “salt” el cual es un valor aleatorio (puede ser un hash keccak256 de un número cualquiera), el cual sirve para diferenciar cada firma generada.

```
struct WithdrawAction {
    uint256 amount;
    bytes32 salt;
    address to;
}
```

**Estructura de la firma WithdrawAction.**

El atributo salt se guarda en cada llamada al método withdraw, lo que impide que se vuelve a utilizar en futuras llamadas con la misma firma. Impide también que, al traspasar los fondos, se ejecute cualquier callback usando los mismos datos de entrada (en caso de que el destino fuera otro smart-contract).

```
require(!usedSalt[_action.salt], "Salt already used");
```

**Guardar cada salt usada.**

Es importante señalar que solo se transfieren los fondos una vez todos los cambios de estado han sido efectuados, un aspecto importante en la prevención de este tipo de ataques.

### 4.3 Firmas en el mezclador

Todas las firmas se generan usando el algoritmo ECDSA [12] (Elliptic Curve Digital Signature Algorithm), el cual es el empleado en redes como Ethereum.

Se trata de un algoritmo de clave pública interesante el cual permite tamaños de números menores para obtener claves seguras, un requisito perfecto para la máquina virtual de Ethereum, la cual soporta tamaños de 256 bits.

Algoritmos como pueden ser RSA quedan descartados, ya que generalmente para obtener seguridad se recomiendan tamaños de clave superiores a 1024 bits, algo que no es posible en la EVM [13].

#### 4.3.1 Generación de las firmas

Para la generación de firmas en el front-end del mezclador, sencillamente se ha optado por usar el método que la wallet del usuario exporte al navegador, de manera general, basándose en MetaMask [14], se usa el método “signTypedData” el cual implementa el firmado de EIP712.

Es necesario pasar una estructura de dominio y una estructura WithdrawAction, la misma que la usada en el smart-contract. Esta firma será generada por el servidor, donde el usuario final podrá usarla para recuperar los fondos.

```
async function generateMainServerSignature(signer,
_contract, _to, _value) {
  const domain = {
    name: "SimpleMixer",
    verifyingContract: _contract
  };

  const types = {
    WithdrawAction: [
      { name: "amount", type: "uint256" },
      { name: "deadline", type: "uint256" },
      { name: "salt", type: "bytes32" },
      { name: "to", type: "address" },
    ]
  };

  const value = {
    amount: _value,
    deadline: 2670810385,
    salt:
ethers.utils.keccak256(ethers.utils.toUtf8Bytes("random_sa
lt_test")),
    to: _to,
  };

  const s = await signer._signTypedData(domain, types,
value);
  return [value, s];
}
```

**Generación de la firma EIP-712 en Hardhat.**

Por la parte del back-end usando la librería go-ethereum se generan las firmas basadas en el EIP712 usando ECDSA:

```
// Generate a valid EIP712 signed message
func signWithdraw(to, contract string, salt []byte, priv
*privateKey) ([]byte, []byte, error) {
     typedData, salt, err := hashTypedData(salt, to,
contract)
     if err != nil {
          return nil, nil, fmt.Errorf("Unable to
hashTypedData: %v", err)
     }
```

```
     // Generate signature
     signature, err := crypto.Sign(typedData,
priv.Private)
     if err != nil {
          return nil, nil, err
     }

     // Legacy V
     signature[64] += 27

     return signature, salt, err
}
```

**Generación de la firma EIP-712 en Go.**

Tanto en Solidity como en el backend se debe generar el hash a la hora de comprobar o crear una firma.

En el backend hay que tener en cuenta el formato antiguo de firmas, por lo que el valor V de la curva debe ser 27 o 28, en este caso incrementamos el valor añadiendo 27 (para convertir 0 o 1 en 27 o 28) [15].

```
// Legacy V
signature[64] += 27
```

**Modificación para firmas legacy en Go.**

Este formato ya no se recomienda para nuevas aplicaciones actualmente, pero ciertas wallets aún lo usan.

### 4.3.2 Comprobación de las firmas

Para la recuperación de las firmas en Solidity, lo primero es recrear la estructura de la firma, usando el mismo algoritmo de hash (keccak256) y recreando tanto como la estructura del dominio como la estructura WithdrawAction:

```
function getWithdrawTypedDataHash(WithdrawAction memory
_action) private view returns (bytes32) {
```

```
    // Hash struct
    bytes32 structHash = keccak256(
      abi.encodePacked(
        WITHDRAW_ACTION_IDENTIFIER,
        _action.amount,
        _action.deadline,
        _action.salt,
        uint256(uint160(_action.to))
      )
    );

    return keccak256(
      abi.encodePacked(
        "\x19\x01",
        DOMAIN_IDENTIFIER,
        structHash
      )
    );
}
```

**Recreación del hash en Solidity.**

Una vez se tiene el hash del contenido firmado y la firma proporcionada, lo primero es obtener los valores r,s y v, necesarios para obtener la clave pública de un mensaje firmado, finalmente obtener la clave pública, usando la llamada ecrecover [16].

```
function recover(bytes32 _hash, bytes memory _signature)
internal pure returns (address) {
    require(_signature.length == 65, "Invalid signature
length");

    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
      r := mload(add(_signature, 0x20))
      s := mload(add(_signature, 0x40))
      v := byte(0, mload(add(_signature, 0x60)))
    }

    return ecrecover(_hash, v, r, s);
}
```

**Recuperación de la clave pública de un mensaje firmado en Solidity.**

Para recuperar la firma en el contrato, se ha utilizado código directamente en ensamblador, en concreto “mload” para cargar una palabra (32 byte) de la memoria (usando la posición de la memoria donde reside nuestra firma + 0x20) a la variable especificada.

En la parte del backend también existe una porción para obtener la clave pública de un mensaje firmado, con el mismo funcionamiento, en concreto esta función se usa para verificar la transacción que el usuario ha realizado a la hora de depositar los fondos inicialmente:

```
func verifySignature(signedMessage, message string)
(string, error) {
     // Create hash of the message
     hashedMessage := []byte("\x19Ethereum Signed
Message:\n" + strconv.Itoa(len(message)) + message)
     hash := crypto.Keccak256Hash(hashedMessage)

     decodedMessage := hexutil.MustDecode(signedMessage)

     // Handle EIP-115 not implemented
     if decodedMessage[64] == 27 || decodedMessage[64] ==
28 {
          decodedMessage[64] -= 27
     }

     // Recover public key
     pub, err := crypto.SigToPub(hash.Bytes(),
decodedMessage)
     if err != nil {
          return "", err
     }
     if pub == nil {
          return "", errors.New("Unable to get a public
key from the message")
     }
     return crypto.PubkeyToAddress(*pub).String(), nil
}
```

**Recuperación de la clave pública de un mensaje firmado en Go.**

### 4.3.3 Usando una tercera dirección a modo de proxy

Una opción a la hora de retirar los fondos es la de usar la dirección final directamente, este caso no presenta ningún problema de privacidad, pues no existe relación entre la dirección inicial y la final.

Debido al uso de la firma es posible usar una tercera dirección a modo de proxy para añadir una capa de privacidad más, de este modo se puede realizar la retirada de fondos desde una tercera dirección.

### 4.4 Clave privada del mezclador

La clave privada del mezclador, la cual se usa para la generación de los mensajes firmados se pasa en texto plano y hexadecimal al mezclador. Al tratarse de una prueba de concepto es una acción aceptable.

En producción este método no es aceptable ni seguro, pues un usuario podría simplemente por los parámetros del proceso obtener la clave privada en texto plano.

Opciones posibles para este caso puede ser el uso de un signer externo o algún mecanismo donde la clave esté encriptada (por ejemplo, AES-GCM).

Además, añadir mecanismos dentro de la aplicación para evitar que la clave resida en la memoria más tiempo del necesario, por ejemplo una vez la operación ha acabado poner todos los bits a 0 en memoria.

# 5.Repositorio del mezclador

Todo el código del mixer se encuentra en el siguiente repositorio git: https://github.com/Raggaer/simple-mixer. En el repositorio se puede encontrar el archivo README.md el cual tiene instrucciones detalladas sobre el funcionamiento y ejecución del proyecto.

### 5.1 Smart-contracts y proyecto en Hardhat

En la carpeta “hardhat” se encuentra todo lo relacionado con los smart-contracts empleados en el mezclador, en concreto se encuentra el archivo “SimpleMixer.sol”.

En la carpeta “scripts” se encuentra un sencillo script que despliega el smart-contract. Puede ser ejecutado con el comando:

```
npx hardhat run scripts/deploy-mixer.js
```

**Ejecución de script deploy en Hardhat**

También existen una serie de tests para el contrato dentro de la carpeta “tests”. Los cuales se pueden ejecutar con el comando:

```
npx hardhat test
================

Compiled 1 Solidity file successfully


  SimpleMixer test
       Mix some coins properly (924ms)
       Try to re-use same signature salt (106ms)
       Use a signature signed by other invalid key (68ms)
       Use a different amount at withdraw (63ms)


  4 passing (1s)
```

**Ejecución y resultado de los tests en Hardhat**

## 5.2 Frontend (HTML y JS)

El frontend se encuentra en el directorio “web”, en concreto en los directorios “web/views” y “web/static”, dentro de views se encuentra un archivo .html para la página principal del mezclador.

Tras realizar la primera transacción el usuario debe firmar un mensaje (el hash de la transacción) y mandarlo al servidor mediante una petición AJAX.

La firma que el servidor genera tiene varios campos (destino, cantidad, salt), para que sea más sencillo compartir o guardar esta información en el cliente se agrupan todos los campos (json) y se codifican con base64, para todo sea una cadena de texto únicamente.

### 5.3 Backend (Go)

El backend está escrito en el lenguaje de programación Go, en SimpleMixer el backend se encarga de validar mensajes firmados por parte de usuarios que han depositado fondos en el contrato y finalmente producir una firma EIP-712 para que el mismo usuario pueda retirar los fondos desde otra dirección.

Se hace uso de la librería go-ethereum para poder acceder a funciones relacionadas con Web3.

### 5.4 Puesta en marcha

Para la puesta en marcha total del mixer es necesario realizar los siguientes pasos:

1. Ejecutar un nodo local o tener acceso a algún nodo de la red. Recomendado usar ganache-cli.
2. Desplegar el contrato en la red local. Recomendado usar el script del repositorio.
3. Iniciar el servidor web.
4. Interactuar con el contrato mediante el front-end, en la dirección web http://localhost:8080.

En primer lugar, es necesario tener algún nodo en ejecución para poder conectarse a la red, una opción popular es el uso de Ganache [17].

Una vez el nodo está en ejecución, el siguiente paso es desplegar el contrato, en la carpeta “hardhat/scripts/” se encuentra un script para ayudar en el lanzamiento, una vez se ejecuta podemos obtener la dirección en la cual ha sido lanzado:

```
> npx hardhat run --network localhost scripts/deploy-
mixer.js
```

```
Compiled 1 Solidity file successfully

SimpleMixer mixing fee: 10%
SimpleMixer fee destination address:
0xca6a54142a48f98d99f2057103917D2f374136CD
SimpleMixer deployed to:
0x246322B86e1b39D6eD02660AC1310C2fDb9E5d82
```

**Deployment del contrato SimpleMixer.**

Una vez desplegado, es necesario ejecutar el servidor web, pero antes es necesario compilarlo para el sistema que se esté usando. La compilación se realiza con el siguiente comando, es necesario tener el lenguaje de programación Go instalado:

```
go build
```

**Compilación del servidor web en Go.**

Se creará un artefacto ejecutable en la misma carpeta llamado “simple_mixer”, el cual se podrá ejecutar para iniciar el servidor web.

El siguiente paso es ejecutar el servidor web, el servidor tiene dos flags que son necesarias a la hora de iniciar la aplicación:

```
> ./simple_mixer -h

Usage of ./simple_mixer:
  -contractAddress string
     Address where the mixer contract is deployed (default
"0x0")
  -privateKey string
     Private key for message signing (default "0x0")
```

**Ejecución del servidor web.**

- -contractAddress: Dirección donde el contrato está desplegado, en hexadecimal.
- -privateKey: Clave privada en hexadecimal encargada de generar los mensajes firmados.

Es importante mencionar que la flag “privateKey” debe coincidir con el atributo que el contrato espera (la misma clave que hizo el despliegue del contrato) para verificar las firmas.

Una vez el servidor web está en ejecución se puede acceder al front-end en “http://localhost:8080” y empezar a interactuar con el contrato.

Para la interacción en el front-end es necesario que el navegador en uso tenga alguna wallet instalada y que esta sea compatible con la API Web3, el desarrollo y las pruebas han sido realizadas con la wallet MetaMask versión 10.23.2.

La secuencia que el frontend ejecutará es la siguiente:

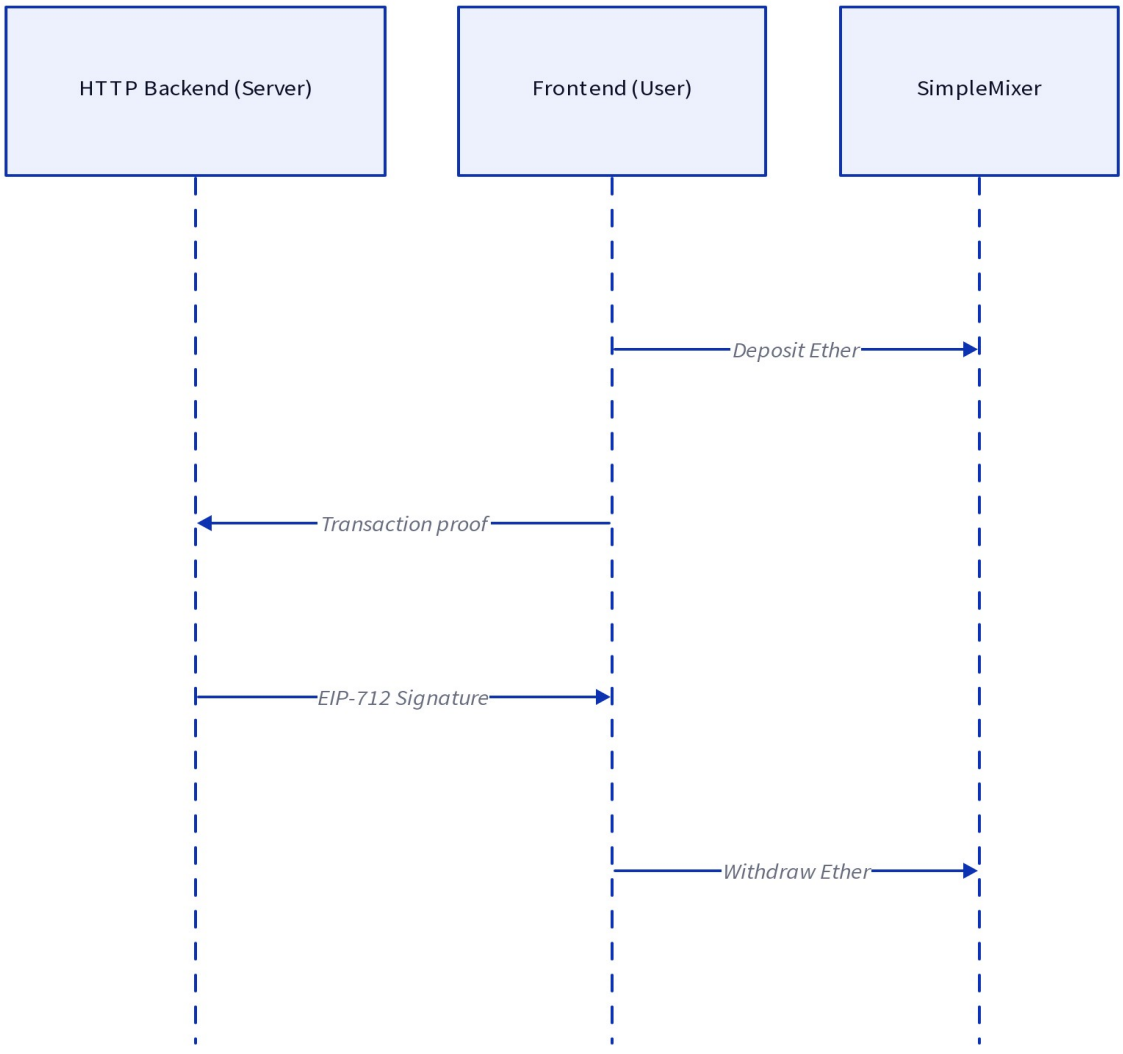



**Figure 8: SimpleMixer diagrama de secuencia**

# 6.Conclusión

El uso de mixers va más allá de actividades delictivas (por ejemplo, intentar ocultar la procedencia de dinero), en redes donde sea posible perder el anonimato que la blockchain nos ofrece y siempre serán una herramienta de importante peso.

Los mezcladores dan la posibilidad de realizar transacciones realmente anónimas (siempre que la implementación del mezclador sea la correcta) en redes que no permiten transacciones confidenciales de manera nativa como por ejemplo Ethereum y Bitcoin.

Gracias a la naturaleza de cómo funcionan los balances de criptomonedas nativas (y tokens) en redes como Ethereum es sencillo realizar su mezcla, pues no existe ningún mecanismo diferenciador entre las diferentes cantidades que una cuenta pueda tener, todas las “monedas” están mezcladas entre sí. Por ejemplo, los balances de tokens en Ethereum suelen guardarse en un simple mapping o integer.

## 6.1 Censura en Ethereum Proof of Stake

Desde el merge de Ethereum con el paso de PoW a PoS la censura en Ethereum ha aumentado de forma considerable.

En primer lugar, con Proof of Work la censura era más complicada pues la generación de bloques era más descentralizada, los usuarios podían fácilmente cambiar de pool si esta decidía censurar bloques.

Tras el paso a PoS han nacido multitud de protocolos centralizados que permiten a sus usuarios hacer staking de su Ethereum, por lo que todo este Ether que se está almacenando en los diferentes nodos controlados por la misma organización está ayudando a la censura de bloques, en concreto censura comandada por la OFAC (Oficina de Control de Activos Extranjeros de los Estados Unidos).

Se pueden utilizar herramientas como mevWatch [18] donde poder visualizar la cantidad de bloques que siguen las medidas de la OFAC (usando la implementación MEV-Boost, algo que la mayoría de nodos usan para maximizar sus ganancias):

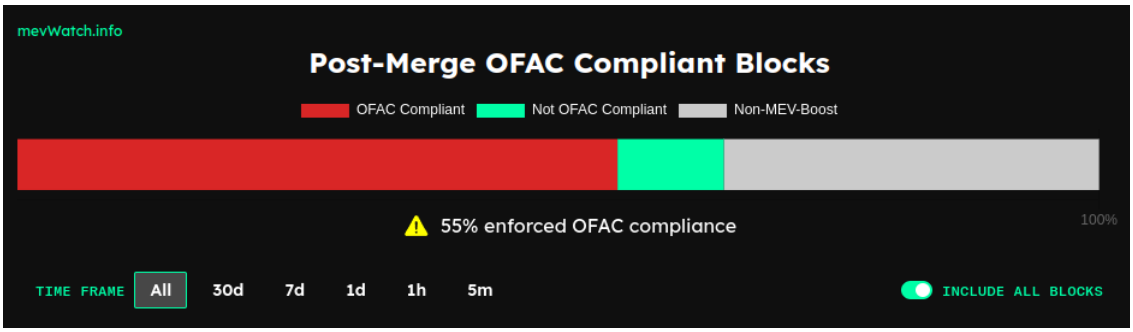



**Figure 9: Ethereum PoS bloques censurados**

Desde el cambio a Proof of Stake entidades como Binance, Celsius, Lido, Huobi, Kraken ofrecen servicios de staking a sus usuarios, estas entidades usan el Ether proporcionado por sus usuarios para ejecutar sus nodos validadores en la red.

También proveedores API como Infura y Alchemy están ya censurando ciertas transacciones, estos proveedores tienen mucho peso en la red de Ethereum, en concreto, herramientas como MetaMask (una de las wallets más usada actualmente) usan Infura para interactuar con la red.

### 6.1.1 El problema de Flashbots MEV-Boost

La censura en Ethereum viene en gran medida debido a la implementación MEV-Boost de la organización Flashbots [19], el problema reside en la lista de relés que Flashbots ofrece.

Flashbots es el relay más grande actualmente en Ethereum por lo que es normal que la gran mayoría de validadores estén usando Flashbots. El problema reside en que Flashbots acepta las censuras de la OFAC y por lo tanto todos los usuarios que usen alguno de sus relays estarán censurando transacciones.

El usuario que maneja un nodo validador tiene la capacidad de cambiar de proveedor y usar uno sin censura, en la actualidad existen varias opciones con diferentes implementaciones, como se puede observar en la siguiente lista [19].

Exchanges centralizados que ofrecen servicios de staking también censuran bloques y estas opciones son las más atractivas para usuarios que simplemente quieren poder hacer staking de sus Ethers sin necesidad de cumplir los requisitos (y tener el conocimiento) para poder ejecutar su propio nodo validador (los cuales no son triviales).

### 6.1.2 Censura en el mezclador tornado.cash

La OFAC recientemente registró tornado.cash en su lista de sanciones, desde entonces el uso de tornado.cash es ilegal en los Estados Unidos y cualquier usuario que interactúe con este protocolo puede tener problemas legales.

Se podría pensar que censurar en una red como Ethereum no tiene ningún sentido, pues las direcciones públicas no tienen ninguna asociación con una persona física, pero hoy en día sí existe censura como hemos visto con anterioridad.

Como se ha comentado antes, desde el cambio a Proof of Stake en Ethereum, validadores que usan ciertos relays ya están censurando transacciones relacionadas con tornado.cash, además proveedores como Infura y Alchemy también están censurando llamadas RPC que interactuan con tornado.cash.

Aún tras la censura el código sigue en la blockchain de Ethereum, aún es posible interactuar con el contrato, pero el front-end ya no es accesible. Aunque como se ha comentado antes, ciertos nodos probablemente no acepten transacciones que interactúan con este protocolo.

# 7.Conclusiones y trabajos futuros

## 7.1 Conclusiones del trabajo

La principal conclusión es la de explicar cómo el uso de los mezcladores no se basa solamente en actividades que se puedan considerar delictivas, existen muchos más casos de uso como se ha mostrado en el trabajo.

Además, el funcionamiento de estas herramientas queda explicado y demostrado con un ejemplo práctico, empleando un método basado en firmas ECDSA que puede resultar novedoso frente a las soluciones existentes.

El uso de firmas abre la puerta a una seria de posibilidades (verificación de un servidor central), su uso a modo de llave puede ser interesante para otros propósitos (por ejemplo, verificación de bots con contratos centralizados).

## 7.2 Reflexión crítica sobre la consecución de los objetivos planteados

El objetivo principal, mostrar casos de uso para los mezcladores en redes como Ethereum se logra en los primeros capítulos del trabajo.

Mostrar el funcionamiento de los mezcladores se logra en los siguientes apartados, en concreto con el desarrollo del mezclador SimpleMixer, en el cual se puede observar el funcionamiento de un mezclador centralizado.

## 7.3 Análisis crítico sobre el seguimiento de la planificación y metodología

La planificación se ha seguido casi en su totalidad, puntos sobre el uso de ciertos mixers han sido omitidos pues cada mixer tiene su protocolo, por lo que no ayudan a su explicación más de lo que ya ayuda el desarrollo de SimpleMixer.

En cuanto a la metodología, sí se ha seguido lo planteado. En los puntos más teóricos primero se realiza una fase de información y luego se desarrolla. En cuanto al desarrollo del mezclador, primero se empezó por la parte de Solidity y finalmente la parte en Go.

## 7.4 Impactos no previstos

Durante el desarrollo de SimpleMixer su complejidad ha sido un poco más elevada de la esperada, en concreto por problemas de front-running se ha tenido que modificar ciertos aspectos del mezclador que lo hacen menos “user-friendly”.

La necesidad de una dirección final no ayuda a su descentralización y hace que el proceso sea en general menos sencillo para el usuario.

La sanción al mezclador tornado.cash en un principio afectó el ritmo de trabajo, la documentación sobre el protocolo no estaba disponible, más adelante se restauró el repositorio en git.

## 7.5 Líneas de trabajo futuro

En esta entrega no ha podido explorarse al completo el funcionamiento de uno de los mezcladores más relevantes en la actualidad, tornado.cash, en concreto por su elevada complejidad y longitud.

Queda pendiente profundizar sobre su funcionamiento, en concreto explicar cómo logra compartir información sensible sin revelar dicha información usando zk-SNARKs.

# 4. Glosario

- Mixer: Traducción de mezclador en Ingles.
- ABI: Application Binary Interface, interfaz para interactuar con contratos en Ethereum.
- API: Interfaz de programación de aplicaciones, software que ofrece una biblioteca que puede ser usada por otro software.
- Back-end: Rama del desarrollo que se encarga de la lógica.
- Front-end: Rama del desarrollo que se encarga de la parte visual, interfaz de usuario.
- RSA: Algoritmo criptográfico de clave pública.
- ECDSA: Algoritmo de clave pública que emplea curvas elípticas.
- RPC: Llamadas a procedimiento remoto, programas que puede utilizar programas en otros servidores remotos.
- EVM: Máquina virtual de Ethereum (Ethereum Virtual Machine).
- Solidity: Lenguaje de programación para implementar contratos inteligentes en Ethereum.
- Golang: Lenguaje de programación también conocido como Go.
- zk-SNARK: Pruebas de conocimiento cero no interactivas.

# 5. Bibliografía


1. Hilo de Twitter de Jeff Coleman (@technocrypto). Disponible en https://twitter.com/technocrypto/status/1556884107040378880. [Consulta 10 de octubre de 2022].
2. Solidity Team. Documentación Solidity 0.8.15. Disponible en: https://docs.soliditylang.org/en/v0.8.17/. [Consulta: 16 de octubre de 2022].
3. István András Seres, Dániel A. Nagy, Chris Buckland y Péter Burcsi (2019). “MixEth; efficient, trustless coin mixing service for Ethereum”. Disponible en: https://eprint.iacr.org/2019/341.pdf. [Consulta: 13 de octubre de 2022].
4. Alexey Pertsev, Roman Semenov y Roman Storm (2019): "Tornado Cash Privacy Solution Version 1.4”. Disponible en: https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf. [Consulta: 10 de septiembre de 2022].
5. U.S. Department of the Treasury. U.S. “Treasury Sanctions Notorious Virtual Currency Mixer Tornado Cash (2022)”. Disponible en: https://home.treasury.gov/news/press-releases/jy0916. [Consulta: 10 de septiembre de 2022].
6. Koh Wei Jie. Código fuente de MicroMix. Disponible en: https://github.com/weijiekoh/mixer. [Consulta 8 de septiembre de 2022].
7. Etherscan. The Ethereum Blockchain Explorer. Disponible en: https://etherscan.io/. [Consulta 6 de noviembre de 2022].
8. Código fuente de Tornado.cash. Disponible en: https://github.com/tornadocash/tornado-core. [Consulta 7 de noviembre de 2022].
9. Zhipeng Wang, Stefanos Chaliasos, Kaihua Qin, Liyi Zhou, Lifeng Gao, pascal berrang, Ben Livshits y Arthur Gervais (2022): “On How Zero-Knowledge Proof Blockchain Mixers Improve, and Wosen User Privacy”. Disponible en https://arxiv.org/pdf/2201.09035.pdf. [Consulta 7 de noviembre 2022].
10. Remco Bloemen, Leonid Logvinov, Jacob Evans. EIP-712: Typed structured data hashing and signing. Disponible en: https://eips.ethereum.org/EIPS/eip-712. [Consulta 21 de noviembre de 2022].

11. Zcash. What are zk-SNARKs. Disponible en: https://z.cash/technology/zksnarks/. [Consulta 3 de diciembre de 2022].
12. ECDSA Wikipedia. Elliptic Curve Digital Signature Algorithm. https://es.wikipedia.org/wiki/ECDSA. [Consulta 22 de noviembre de 2022].
13. Ethereum. Máquina Virtual de Ethereum. Disponible en: https://ethereum.org/es/developers/docs/evm/. [Consulta 3 de diciembre de 2022].
14. MetaMask. https://metamask.io/. [Consulta 22 de noviembre de 2022].
15. go-ethereum. https://github.com/ethereum/go-ethereum/blob/3775e198dfc9fec312f44fd563dcc525cefa1e5d/signer/core/signed_data.go#L40. [Consulta 13 de diciembre de 2022].
16. Soliditydeveloper. What is ecrecover in Soldity?. Disponible en: https://soliditydeveloper.com/ecrecover. [Consulta 3 de diciembre de 2022].
17. Ganache. One Click Blockchain. https://trufflesuite.com/ganache/. [Consulta 21 de diciembre de 2022].
18. MEVWatch. https://www.mevwatch.info/. [Consulta 5 de diciembre en 2022].
19. Flashbots. Disponible en: https://www.flashbots.net/. [Consulta 5 de diciembre de 2022].
20. Eth-Educators. MEV relay list for Mainnet. Disponible en: https://github.com/eth-educators/ethstaker-guides/blob/main/MEV-relay-list.md. [Consulta 5 de diciembre de 2022].

# 6. Anexos

1. Configuración e instalación de un entorno de desarrollo Hardhat. Disponible en: https://hardhat.org/hardhat-runner/docs/config.
2. Configuración e instalación del lenguaje de programación Go. Disponible en: https://go.dev/doc/install.
3. Elliptic Curve Digital Signature Algorithm. Explicación del algoritmo ECDSA. Disponible en: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.