

Anexo

Descripción técnica

Prueba de Concepto

Blockchain y el derecho de supresión

ÍNDICE

I.	INTRODUCCIÓN	3
II.	DETALLES DE IMPLEMENTACIÓN DE LA PRUEBA DE CONCEPTO (POC) DE LA AEPD	3
A.	Entorno de la PoC y Software y herramientas	3
B.	Consideraciones sobre la ejecución del caso de uso	4
C.	Consideraciones para el procedimiento de detección de registros afectados por la supresión	5
D.	Consideraciones para el procedimiento de generación de una nueva versión de software	8

I. INTRODUCCIÓN

La Prueba de Concepto (PoC) de la AEPD describe como implementar unas medidas de gobernanza y técnicas para facilitar el cumplimiento del RGPD, contemplando, en particular, el derecho de supresión. Incluye la definición de unas medidas de gobernanza, políticas y la implementación de las modificaciones técnicas necesarias para facilitar el cumplimiento del RGPD en una infraestructura Blockchain Ethereum.

Este documento tiene como objetivo únicamente detallar y profundizar en los aspectos técnicos de implementación de la prueba de concepto desarrollada por la AEPD y descrita en la [‘Nota Técnica de la Prueba de Concepto Blockchain y el derecho de supresión’](#) y [video explicativo](#).

Dado que tanto la Nota Técnica como la PoC se han desarrollado con una perspectiva centrada en la protección de datos, es fundamental definir con rigor la terminología y conceptos y comprender sus implicaciones. Por lo tanto, se remite al lector a consultar dicha nota técnica antes de realizar la lectura de este documento. La Nota Técnica proporciona el marco conceptual adecuado sobre las implicaciones de la protección de datos en el contexto de Blockchain. También, la Nota Técnica expone y detalla el diseño de la PoC, los procedimientos, medidas organizativas y técnicas desarrollados, junto con los resultados obtenidos de los casos de uso ejecutados. Es fundamental entender todos estos aspectos para interpretar correctamente los resultados de la PoC y su correcta aplicación dentro de un marco de cumplimiento normativo.

Por lo tanto, el presente anexo solo complementa el detalle de la Nota Técnica y está orientada a aquellas personas que habiendo leído esta desean profundizar en los detalles de la implementación de la PoC. Al estar dirigido a personas con conocimientos técnicos en la tecnología Blockchain, no se incluyen referencias en los términos técnicos utilizados en el documento.

II. DETALLES DE IMPLEMENTACIÓN DE LA PRUEBA DE CONCEPTO (POC) DE LA AEPD

En este anexo se presentan detalles adicionales sobre las opciones técnicas seleccionadas y los procedimientos implementados en la PoC. El objetivo es proporcionar una explicación más profunda de las decisiones tomadas en cuanto a su diseño y ejecución. Asimismo, se detallan los aspectos clave de la implementación de algunos de los procedimientos diseñados para la PoC.

A. ENTORNO DE LA PO C Y SOFTWARE Y HERRAMIENTAS

La ejecución de la PoC consiste en construir una infraestructura Blockchain privada permissionada con dos nodos validadores en modo *archive*. Se ha implementado en un ordenador portátil con Windows 11 (Intel core i5 8th gen con 16Gb de RAM), que ejecuta dos máquinas virtuales con Linux Xubuntu 24.04 configuradas en red local con el host (Virtualbox). Cada máquina virtual ejecuta un nodo validador. La sincronización de un nuevo nodo se realiza ejecutando un tercer nodo desde el sistema anfitrión Windows.

Para la infraestructura Blockchain se usa el software cliente oficial Ethereum *geth* v1.13.15 (lenguaje Go, abril 2024), configurado con base de datos de tipo *leveldb*. Es la última versión que soporta el protocolo de consenso PoA *clique* (que se ha eliminado en versiones más recientes en favor de PoS).

Se ha configurado un bloque génesis sencillo, con 2 cuentas validadoras y protocolo *clique*, 7 cuentas con saldo (entre las que se encuentran las validadoras) y por sencillez se incluye hasta el *London Fork*, con un tiempo de creación de bloques de 30 segundos para

efectuar pruebas en los entornos indicados. Las transacciones que se usan son clásicas o *Legacy*.

Las aplicaciones y herramientas usadas para modificar el código fuente, efectuar transacciones, desplegar los *Smart Contracts*, ayuda para decodificar valores, y exploradores de bloques son:

- Entorno de desarrollo Visual Studio Code.
- Entorno de programación *Node.js*¹ con librerías *level*², *merkle-patricia-tree*³, *ethereumjs*⁴, y *web3*⁵ (mediante el gestor de paquetes *npm*⁶)
- Remix IDE <https://remix.ethereum.org/> para compilar los *Smart Contract* y obtener su codificación (bytecode) e interfaces (abi). También para consulta.
- Explorador de bloques Nethereum Explorer <https://explorer.nethereum.com/>
- Explorador de bloques Ethernal Explorer <https://app.tryethernal.com/overview>
- Consola de comandos de *geth* (*geth attach*)
- Decodificador/codificador RLP <https://toolkit.abdk.consulting/ethereum>
- Conversión hexadecimal/texto/decimal <https://www.rapidtables.com/convert/number/hex-to-ascii.html>
- Hashes Keccak 256 https://emn178.github.io/online-tools/keccak_256.html

B. CONSIDERACIONES SOBRE LA EJECUCIÓN DEL CASO DE USO

El proceso de generación de la cadena de bloques original que contiene las transacciones contempladas en el caso de uso de la PoC se ha automatizado. Para ello se ha desarrollado un programa o script que aprovecha la capacidad del cliente oficial de Ethereum *geth* de ejecutar código Javascript en una consola no interactiva.

En primer lugar, se ejecutan los dos nodos inicializados desde el bloque génesis, cada uno de ellos en una máquina virtual, y cuando llevan unos pocos bloques generados (sin transacciones) se lanza el *script* de ejecución de las transacciones desde el ordenador anfitrión Windows.

El *script* de ejecución de transacciones consta a su vez de distintos pasos o procedimientos entre los que se van realizando esperas, con el fin de que las transacciones se vayan validando e incorporando en distintos bloques. En cada paso se envían transacciones a los nodos, conectándose a una consola no interactiva del nodo en cada caso. Por sencillez, cada nodo tiene en su almacenamiento de claves (*keystore*) las cuentas involucradas en la PoC, el primer nodo contiene cuatro cuentas y el segundo nodo contiene las tres restantes. De esta manera, se facilita el envío de transacciones, al no ser necesarias aplicaciones de tipo *wallet*, o de otro tipo, para las transferencias o para interactuar con los *Smart Contracts* (tales como Remix IDE).

Los pasos o procedimientos para efectuar transacciones de transferencia de *ether* se realizan con la función nativa de la consola *geth* `eth.sendTransaction`, mientras que

¹ <https://nodejs.org/en>

² <https://www.npmjs.com/package/level>

³ <https://www.npmjs.com/package/merkle-patricia-tree>

⁴ <https://github.com/ethereumjs/ethereumjs-monorepo>

⁵ <https://www.npmjs.com/package/web3>

⁶ <https://www.npmjs.com/>

para el despliegue de los *Smart Contract* y las transacciones que involucran llamadas a sus funciones, se han desarrollado *scripts* ligeros. Estos *scripts* se pasan a la consola no interactiva de *geth*, y en ellos se incluyen la codificación *bytecode* y el interfaz *abi* de los *Smart Contracts* (obtenidos tras su compilación en Remix IDE), creando y accediendo a las instancias de los *Smart Contract* con la función `eth.contract` entre otras.

C. CONSIDERACIONES PARA EL PROCEDIMIENTO DE DETECCIÓN DE REGISTROS AFECTADOS POR LA SUPRESIÓN

Este procedimiento consiste en detectar las modificaciones necesarias que hay que efectuar en las bases de datos de los nodos, donde se materializa la tabla de registros y la cadena de bloques, y generar la base de datos modificada (hasta el último bloque necesario)

Para ello en la PoC se sobrescriben los datos pertinentes que forman parte de los bloques, almacenamiento de *Smart Contracts*, recibos de transacciones (*receipts/logs*), estado de balances, etc. Para cada bloque se observan las distintas transacciones que se registran en él, y para cada transacción se comprueba si la cuenta en cuestión es origen, destino de esta, o bien está incluida en datos adicionales de la transacción (*input data*). Cuando una de estas transacciones está dirigida a un *Smart Contract*, se comprueba también su almacenamiento y los posibles logs o recibos de transacción que se hayan podido generar y almacenar.

Para efectuar estas operaciones la PoC, por simplicidad, ejecuta un nodo auxiliar que contiene una copia de la base de datos original. Así se facilita la obtención de bloques, transacciones y sus datos mediante consultas a la API que expone el nodo. Adicionalmente, mediante la ejecución de un programa desarrollado en NodeJS, compatible con Windows y Linux, se usan librerías de lectura y escritura en las bases de datos del nodo (de tipo *leveldb*) para acceder al resto de datos necesarios y modificarlos. Este enfoque permite realizar las modificaciones necesarias sin afectar directamente la red principal hasta que se ejecute la actualización del *Hard Fork*.

Cabe recordar que la base de datos *leveldb* de un nodo almacena las diferentes estructuras de datos de la cadena de bloques en la forma de pares clave-valor codificadas con RLP. Los valores clave representan distintas estructuras de datos⁷, entre ellas, los nodos de los árboles de Merkle-Patricia⁸ que se almacenan (*State Trie* y almacenamiento de *Smart Contracts*), los bloques y los *receipts/logs*, en donde puede aparecer la dirección de una cuenta (*address*).

A continuación, se detallan los pasos (automatizados mediante un *script*) que se han de realizar para eliminar el rastro de una dirección de una cuenta (*address*) que ha solicitado la supresión mediante operaciones de lectura y escritura en sus bases de datos *leveldb*:

1. En el intervalo de bloques correspondiente, se consulta para cada bloque las transacciones en donde la cuenta aparece en los campos origen, destino o datos, respectivamente FROM, TO, INPUT/DATA, y a su vez se comprueba si se trata de una transacción a un *Smart Contract*⁹. Para simplificar esta consulta se arranca el nodo sin opciones de minado/validación y sin sincronizarse con otros, aprovechando así las librerías de Node.js disponibles para interactuar con un nodo Ethereum.

⁷ <https://github.com/ethereum/go-ethereum/blob/master/core/rawdb/schema.go>

⁸ <https://ethereum.org/es/developers/docs/data-structures-and-encoding/patricia-merkle-trie/#merkle-patricia-trees>

⁹ Se comprueba el campo CodeHash del World State de la dirección de la cuenta destino de la transacción (*to*). Un *Smart Contract* tiene un valor no nulo.

máximo teórico de la profundidad de 64 niveles¹¹. Con un valor de profundidad de 10 pueden abarcarse más de un billón de cuentas. En la PoC se ha usado una profundidad de 5 *nibbles* compartidos. En el caso de que existiera más de una cuenta con esos 5 *nibbles* compartidos iguales, lo que supondría una colisión de los cinco primeros números del hash de ambas cuentas, habría que efectuar una sencilla comprobación adicional para seleccionar entre ellas la cuenta que se pretende eliminar (comprobando el camino de búsqueda restante, aunque la PoC no contempla esta comprobación por no ser necesaria dado el reducido número de cuentas)

Un nodo en un Merkle Patricia trie es uno de los siguientes:

1. NULL (representado como la cadena vacía)
2. *branch* Un nodo de 17 elementos [v0 ... v15, vt]
3. *leaf* Un nodo de 2 elementos [encodedPath, value]
4. *extension* Un nodo de 2 elementos [encodedPath, key]

Figura 2 Tipos de nodos en un árbol Merkle-Patricia-Trie

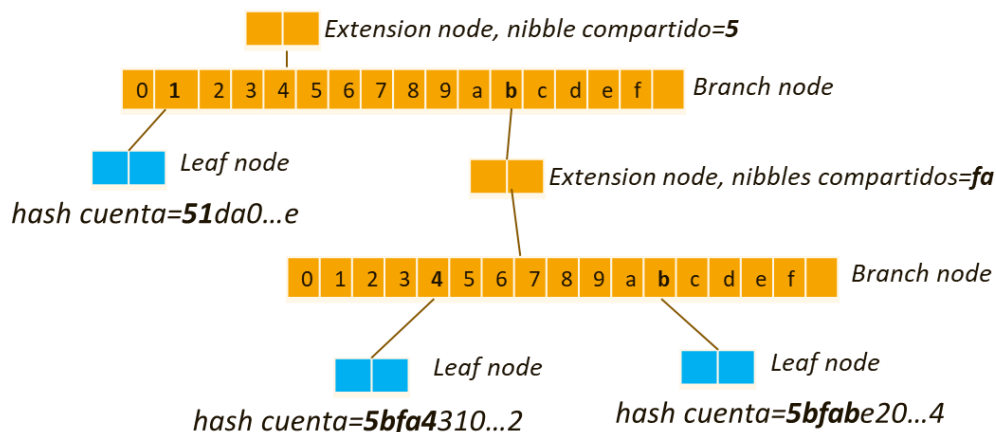


Figura 3. Ejemplo de búsqueda de cuentas en el árbol de Merkle del State Trie. Se muestran los caminos de búsqueda de tres cuentas distintas (aleatorias). La estrategia empleada por PoC sustituye los valores del nodo hoja correspondientes a la cuenta del usuario que ejerce el derecho de supresión por un par de valores nulos.

El nodo hoja contiene dos campos, el primero de ellos contiene los últimos valores del hash Keccak256 de la dirección cuenta, que es el camino de búsqueda restante, y el segundo campo contiene el estado de esa cuenta (balance y demás datos). En la PoC se reemplaza el nodo hoja correspondiente a la cuenta que se va a eliminar por valores nulos (0xc28080 que se corresponde con el valor codificado RLP del par clave-valor nulo ['0x', '0x']).

¹¹ La profundidad máxima teórica del State Trie de Ethereum es de 64 niveles, dado que se basa en claves de 256 bits divididas en *nibbles* de 4 bits. Cada cuenta tiene una ruta única de hasta 64 niveles en el trie, y la cantidad de cuentas que pueden almacenarse depende de las rutas generadas a partir del hash de 256 bits. Aunque el trie podría almacenar 16⁶⁴ cuentas teóricas, en la práctica, este número es limitado por la estructura actual del sistema, el uso del espacio de direcciones y las cuentas realmente existentes. Hay estudios que estiman una profundidad aproximada de 15 niveles. <https://hackmd.io/@jsign/geth-mpt-analysis>, <https://arxiv.org/pdf/2408.14217>

3. En caso de que la transacción se corresponda además con una efectuada a un *Smart Contract*, entonces es necesario comprobar si la cuenta aparece también en el almacenamiento interno del *Smart Contract*¹². Para ello se obtiene también el *Storage Trie* de dicho *Smart Contract* en cada bloque afectado. Se procede de manera análoga al *State Trie*, pero teniendo en cuenta que los índices de búsqueda en el *Storage Trie* dependen del tipo de variable y de su orden de aparición en la declaración en el código fuente¹³. Es necesario, por tanto, conocer con detalle el código fuente de cada *Smart Contract* afectado. En la PoC se han empleado dos *Smart Contract*, uno de ellos con una variable tipo *address* y el otro con una variable tipo *mapping* (en donde la cuenta en cuestión aparecerá en dichas variables).

Key/Clave en *leveldb*, siendo *p* el orden de aparición en el código fuente (sin contar variables constantes y contando desde 0):

Variable sencilla *address*, Key = Keccak256(*p*)

mapping (*address*=>*uint256*) , Key = Keccak256(Keccak256(*address* concatenado con *p*))

Figura 4. Clave en el *Storage Trie* según el tipo de variable y orden en el *Smart Contract*

4. En cada bloque afectado obtener los *receipts*, recibos de las transacciones, y sobrescribir el valor de la cuenta en cuestión por la cadena hexadecimal 'aaa...a' de manera similar a lo realizado en las transacciones del cuerpo del bloque (*blockbody*)
5. Por último, eliminar la cuenta del almacenamiento *keystore* de los nodos que la incluyan.

La base de datos modificada en este punto será la base de la gestión de inconsistencias y se utilizará para implementar el *Hard Fork* de la cadena de bloques.

D. CONSIDERACIONES PARA EL PROCEDIMIENTO DE GENERACIÓN DE UNA NUEVA VERSIÓN DE SOFTWARE

Este procedimiento consiste en generar la nueva versión del software de la infraestructura Blockchain, en donde se incorporan los pares clave-valor de la base de datos modificada, así como el mecanismo de acuerdo de nodos validadores, inspirado en el BIP-0009 de Bitcoin, como se ha descrito anteriormente. En la PoC se implementa una simplificación de éste en el código de *geth*.

Puesto que en el protocolo *clique* el campo *MixDigest* de los bloques no se usa, y siempre tiene un valor vacío, será el que se utilice para señalar el apoyo de los nodos validadores al *Hard Fork*. Una vez alcanzado el acuerdo, dicho campo se modificará para reflejar también este hecho, que será lo que compruebe un nodo que se re-sincroniza o incorpora a la red.

¹² Aunque un *Smart Contract* puede implementar funciones de borrado de sus datos almacenados en un determinado momento (mediante la llamada a una función de borrado), esos datos siguen existiendo en la cadena de bloques como parte de la información almacenada resultante de la ejecución de transacciones en los bloques previos. Dada la estructura del almacenamiento de los datos de un *Smart Contract* en la cadena de bloques, es preciso que los *Smart Contract* sean conocidos (código fuente) para poder efectuar un borrado eficaz de sus datos almacenados.

¹³ https://ethereum.org/en/developers/docs/apis/json-rpc/#eth_getstorageat

```
// Mix digest is reserved for now, set to empty
header.MixDigest = common.Hash{}

// errInvalidMixDigest is returned if a block's mix digest is non-
// zero.
errInvalidMixDigest = errors.New("non-zero mix digest")
```

Figura 5. Código fuente de *geth* donde se muestra el cálculo del campo *MixDigest* de la cabecera de un bloque y una función de error si este no es nulo. En la PoC el campo se modifica para incluir la referencia al apoyo a la nueva versión o nueva versión acordada. La función de error se elimina en la PoC.

La PoC implementa las funciones necesarias en los nodos validadores para llevar la cuenta de un determinado número de los últimos bloques añadidos a la cadena, en los que el campo *MixDigest* contiene el valor correspondiente a la aceptación o apoyo de la nueva versión (en la PoC el valor de este campo es el número de versión, p.ej. 1). Además, es necesario implementar también la modificación en la base de datos local del nodo validador cuando el acuerdo sobre la nueva versión se alcanza. Este acuerdo se alcanza cuando la cuenta del número de bloques que aceptan o apoyan la nueva versión alcanza una mayoría (4 bloques de los 5 últimos en la PoC, con 2 nodos validadores).

Por otra parte, también es necesario implementar la sincronización de un nuevo nodo, o un nodo que se re-sincroniza desde un estado anterior de la cadena de bloques. En ambos casos, dicho nodo ejecuta la nueva versión de software y comprueba el valor *MixDigest* del último bloque que se tiene que descargar, que, si contiene ya el valor de la modificación aceptada y ejecutada, el nodo modifica entonces su base de datos local.

Las funciones de *geth* que se han modificado se encuentran en los siguientes archivos:

- *aepd.go*: Archivo auxiliar para implementar algunas variables y funciones de escritura en la base de datos local del nodo y la comprobación del consenso en los bloques que se van añadiendo.
- *db.go*: Archivo auxiliar para incorporar los cambios en la base de datos local del nodo que ejecutan el derecho de supresión. Es decir los valores afectados y necesarios en los que se ha sobrescrito el valor original de la dirección de la cuenta en cuestión por un valor constante 'aaa...a' y los correspondientes al balance de dicha cuenta.
- *cmd/geth/main.go*: Implementa la funcionalidad de un nodo que se re-sincroniza con la red modificada, modifica la base de datos local y reorganiza con esos cambios. Para ello se añade un proceso de escucha a eventos del componente *downloader*, que es quien chequea los bloques que se quieren descargar/sincronizar, y el campo *MixDigest*.
- *consensus/cliq/clique.go*: Modifica el campo *MixDigest*, con un valor igual a la versión actual del software o un indicador de versión aceptada cuando se llega al acuerdo de los nodos validadores.
- *core/blockchain.go*: Implementa una interfaz para borrar la caché del nodo en el momento de actualizar la base de datos (tras el acuerdo de los validadores o por resincronización de la nueva cadena). De esta manera, cualquier consola o aplicación conectada al nodo no recuperará las transacciones originales de la caché, sino las modificadas presentes ya en su base de datos local.
- *eth/api_backend.go*: Implementa un interfaz para tener acceso a la Blockchain local del nodo.
- *eth/downloader/downloader.go*: Comprueba los valores del campo *MixDigest* de las cabeceras de los bloques que se descarga, cuando encuentra que la modificación se ha efectuado (aceptada y ejecutada en la nueva cadena)

entonces envía un mensaje al nodo de este hecho, para que modifique su base de datos local, y re arranque sincronizándose con la nueva cadena.

- `eth/downloader/events.go`: Añade un nuevo evento para indicar si la en la cadena se ha producido el cambio.
- `miner/miner.go`: Implementa una interfaz para exponer la cadena de bloques del nodo validador.
- `miner/worker.go`: Llama a la función de `aepd.go` que implementa la aproximación al BIP-0009 tras validar un nuevo bloque.