

Annex

Technical description

Proof of Concept

Blockchain and the right

to suppression

INDEX

I.	INTRODUCTION	3
II.	AEPD PROOF OF CONCEPT (POC) IMPLEMENTATION DETAILS	3
A.	PoC Environment, Software and Tools	3
B.	Use case implementation considerations	4
C.	Considerations for the procedure for detecting affected records for right to erasure	5
D.	Considerations for the procedure of generating a new software version	8

I. INTRODUCTION

The AEPD's Proof of Concept (PoC) describes how to implement governance and technical measures to facilitate GDPR-compliance, taking into account, in particular, the right to erasure. It includes the definition of governance measures, policies and the implementation of the necessary technical modifications to facilitate GDPR compliance on an Ethereum Blockchain infrastructure.

This document aims only to detail and deepen the technical aspects of the implementation of the proof of concept developed by the AEPD and described in the ['Technical Note on the Blockchain Proof of Concept and the right of erasure'](#) and [explanatory video](#).

As both the Technical Note and the PoC have been developed from a data protection perspective, it is essential to rigorously define the terminology and concepts and to understand their implications. The reader is therefore referred to the Technical Note before reading this document. The Technical Note provides the appropriate conceptual framework on the implications of data protection in the context of Blockchain. Also, the Technical Note sets out and details the design of the PoC, the procedures, organisational and technical essential to understand all these aspects in order to correctly interpret the results of the PoC and its correct application within a regulatory compliance framework.

Therefore, this annex only complements the details of the Technical Note and is intended to those who, having read the Technical Note, wish to go into the details of the PoC implementation. As it is addressed to people with technical knowledge in Blockchain technology, no references are provided for the technical terms used in the document.

II. AEPD PROOF OF CONCEPT (POC) IMPLEMENTATION DETAILS

This annex presents additional details on the technical options selected and the procedures implemented in the PoC. The aim is to provide a more in-depth explanation of the choices made in terms of design and implementation. It also details key aspects of the implementation of some of the procedures designed for the PoC.

A. PoC ENVIRONMENT, SOFTWARE AND TOOLS

The PoC execution consists of building a permissioned private Blockchain infrastructure with two validator nodes in archive mode. It has been implemented on a Windows 11 laptop (Intel core i5 8th gen with 16Gb RAM), running two virtual machines with Linux Xubuntu 24.04 configured in local network with the host (Virtualbox). Each virtual machine runs a validator node. Synchronisation of a new node is done by running a third node in the Windows host system.

The Blockchain infrastructure uses the official Ethereum client software geth v1.13.15 (Go language, April 2024), configured with leveldb database. It is the latest version that supports the PoA clique consensus protocol (which has been removed in more recent versions in favour of PoS).

A simple genesis block has been configured, with 2 validator accounts and clique protocol, 7 accounts with balance (including the validators) and for simplicity it is up to the *London Fork* included, with a block creation time of 30 seconds for testing in the indicated environment. The transactions used are of type classic or *Legacy*.

The applications and tools used to modify source code, perform transactions, display *Smart Contracts*, help decode values, and block explorers are:

- Visual Studio Code development environment.

- *Node.js* programming environment¹ with *level*², *merkle-patricia-tree*³, *ethereumjs*⁴, and *web3*⁵ libraries(via *npm* package manager⁶).
- Remix IDE <https://remix.ethereum.org/> to compile *Smart Contracts* and get their bytecode and interfaces (*abi*). Also, for consultation.
- Nethereum Explorer block explorer <https://explorer.nethereum.com/>
- Eternal Explorer block explorer <https://app.tryeternal.com/overview>
- *geth* command line console (*geth attach*)
- RLP decoder/encoder <https://toolkit.abdk.consulting/ethereum>
- Hexadecimal/text/decimal conversion
<https://www.rapidtables.com/convert/number/hex-to-ascii.html>
- Hashes Keccak 256 https://emn178.github.io/online-tools/keccak_256.html

B. USE CASE IMPLEMENTATION CONSIDERATIONS

The process of generating the original Blockchain/table containing the transactions contemplated in the PoC use case has been automated. For this purpose, a programme or script has been developed that takes advantage of the capacity of the official Ethereum *geth* client to execute Javascript code in a non-interactive console.

First, the two initialised validator nodes are executed from the genesis block, each of them in a virtual machine, and when a few blocks have been generated (without transactions), the transaction execution script is launched from the Windows host computer.

The transaction execution script in turn consists of different steps or procedures between which waits are carried out, so that the transactions are validated and incorporated into different blocks. At each step, transactions are sent to the nodes, connecting to a non-interactive console of the node in each case. For simplicity, each node has in its *keystore* the accounts involved in the PoC, the first node contains four accounts, and the second node contains the remaining three. In this way, the sending of transactions is facilitated, as no *wallet-type* or other applications are needed for transfers or to interact with *Smart Contracts* (such as Remix IDE).

The steps or procedures for performing *ether* transfer transactions are performed with the native *geth* `eth.sendTransaction` console function, while lightweight scripts have been developed for the deployment of *Smart Contracts* and transactions involving function calls. These scripts are passed to the non-interactive *geth* console, and include the *bytecode* encoding and *abi* interface of the *Smart Contracts* (obtained after compilation in Remix IDE), creating and accessing *Smart Contract* instances with the `eth.contract` function among others.

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/>

which the account address can be derived. In Ethereum three fields are used for this purpose 'r','s','v', the PoC overwrites 'r'¹⁰.

This strategy causes an inconsistency that the existing block explorers or code libraries do not take into account, so they will generate an error. However, this inconsistency is resolved in the PoC with the *Hard Fork*, leaving no trace in the node databases of the deleted (overwritten) account address.

1	Key:	620000000000000008b45659f0463eb114b464797ae523c5d7a8ee03b23dd00eee97e4de3a47bdfba	1	Key:	620000000000000008b45659f0463eb114b464797ae523c5d7a8ee03b23dd00eee97e4de3a47bdfba
2	Value:	c2c0c0	2	Value:	c2c0c0
3			3		
4			4		
5	Key:	62000000000000000957210967687e95e56c26a896999b4a0dc5f6302674918a2443695fed3c064d40	5	Key:	62000000000000000957210967687e95e56c26a896999b4a0dc5f6302674918a2443695fed3c064d40
6	Value:	f901cbf901c7f86502843b9aca008252089498db3b4533c56734e561fb8d5a3fbc0b2813e99f0280820636a04e9702e89f5063317d7fe3aa0b1625f8431adcb4d452d0060faa60b148d69f72a0271dd2f1400489e50871d74faf58e412b603deb4dfea3b5747d83fa93bfa9e06f86501843b9aca008252089417c3b445750221cfc48b1ea6a8d13b1eef1da1970180820636a05c24cfefa3a50e948ccf0c747f4c3b8d9530c5136d3fd822eaa7044a4f7b46f2a0536e28dcb593e4955da593bbc48d484d3abcb7f9f923c2cd3b0732eabcf82e9f86503843b9aca0082520894a7941c445b42e38722ed7a3e3dce04c06b6a94680380820636a0f97d8968b42219b0e4887cbcb949a44c07f9d19bcc7c14d34e9ed75ea10f8820aa056fb32e0ba2fa93d29fd6e9cfe8927db0417bea5c18776cc74a0f0172640feaf89002843b9aca0082fd9894a1e040a43df9ab1b31398ca48a77632a9d2044bd8718de76816d8000a4f088d54700000000000000000000000000007f67ac053d235377fb86cce22e2db438820636a090fe9f649e01914e6c1e7388b44d561e5412bf6bbf5e72a81ac324550083a3a05cd08a18bd36663947ec8ab16ed9e4ad236d3bba2e6925b793f64b26fbc3dfa7c0	6	Value:	f901cbf901c7f86502843b9aca008252089498db3b4533c56734e561fb8d5a3fbc0b2813e99f0280820636a04e9702e89f5063317d7fe3aa0b1625f8431adcb4d452d0060faa60b148d69f72a0271dd2f1400489e50871d74faf58e412b603deb4dfea3b5747d83fa93bfa9e06f86501843b9aca0082520894aaa0180820636a05c24cfefa3a50e948ccf0c747f4c3b8d9530c5136d3fd822eaa7044a4f7b46f2a0536e28dcb593e4955da593bbc48d484d3abcb7f9f923c2cd3b0732eabcf82e9f86503843b9aca0082520894a7941c445b42e38722ed7a3e3dce04c06b6a94680380820636a0f97d8968b42219b0e4887cbcb949a44c07f9d19bcc7c14d34e9ed75ea10f8820aa056fb32e0ba2fa93d29fd6e9cfe8927db0417bea5c18776cc74a0f0172640feaf89002843b9aca0082fd9894a1e040a43df9ab1b31398ca48a77632a9d2044bd8718de76816d8000a4f088d5470000000000000000000000000007f67ac053d235377fb86cce22e2db438820636a090fe9f649e01914e6c1e7388b44d561e5412bf6bbf5e72a81ac324550083a3a05cd08a18bd36663947ec8ab16ed9e4ad236d3bba2e6925b793f64b26fbc3dfa7c0

Figure 1. Modified transaction, TO field, block number 9 (raw blockbody data structure, as stored in the key-value database). Transaction number 2 of this block, once decoded, results in the different fields composing the transaction with the values:

["0x01", "0x3b9aca00", "0x5208", "0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa", "0x01", "0x", "0x0636", "0x5c24cfefa3a50e948ccf0c747f4c3b8d9530c5136d3fd822eaa7044a4f7b46f2", "0x536e28dcb593e4955da593bbc48d484d3abcb7f9f923c2cd3b0732eabcf82e9f"].

- For each block with affected transactions, the *State Trie* trees are obtained and the tree-node corresponding to the account to be deleted is searched for. The tree-nodes are stored as key-value pairs, where the key contains information about the account address, in particular the search path not shared by the parent nodes, and the value contains the account status (balance and other account data).

Given the way the states of the accounts are stored in the *State Trie*, with the *Keccak256* hash of the account address being the search path or index, the PoC, for simplicity, performs a search with a reduced depth or number of shared *nibbles*. The depth of the *State Trie* is directly related to the maximum number of accounts (leaf nodes) it can hold, so the greater the number of existing accounts, the greater the depth, with a theoretical maximum depth value of 64 levels¹¹. With a depth value of 10, more than a trillion accounts can be covered. A depth of 5 shared *nibbles* has been used in the PoC. In case there is more than one account with the same 5 shared *nibbles*, which would result in a collision of the first five hash numbers of both accounts, a simple additional check would have to be performed to select the account

¹⁰ The values r and s are the components of the ECDSA elliptic curve that generates the public key, v is an identifier that facilitates the extraction of the public key. If the sender account is derived having overwritten, a different account will be obtained (the one corresponding to the new values r,s,v, with a very low probability that it already exists. in any case, the transaction has no effect on it).

¹¹ The theoretical maximum depth of Ethereum's *State Trie* is 64 levels, as it is based on 256-bit keys divided into 4-bit *nibbles*. Each account has a unique path up to 64 levels in the trie, and the number of accounts that can be stored depends on the paths generated from the 256-bit hash. Although the trie could store 16^{64} theoretical accounts, in practice, this number is limited by the current system structure, the use of address space and the actual existing accounts. Studies estimate an approximate depth of 15 levels.

<https://hackmd.io/@sign/geth-mpt-analysis>, <https://arxiv.org/pdf/2408.14217>

to be deleted (by checking the remaining search path, although the PoC does not foresee this check as it is not necessary due to the small number of accounts).

A node in a Merkle Patricia trie is one of the following:

- 1.NULL (represented as the empty string)
- 2.Branch: A 17-item node [v0 ... v15, vt]
- 3.Leaf: A 2-item node [encodedPath, value]
- 4.Extension: A 2-item node [encodedPath, key]

Figure 2 Types of nodes in a Merkle-Patricia-Trie

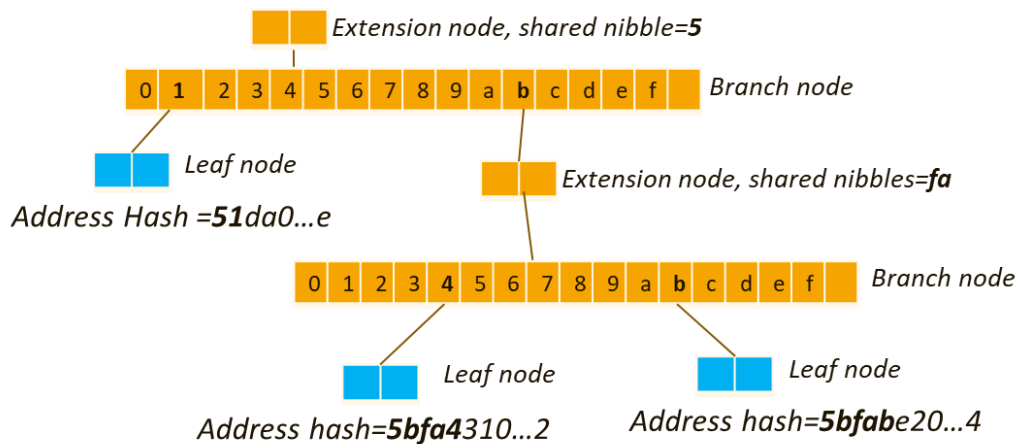


Figure 3. Example of an address account search in the Merkle tree of the State Trie. The search paths of three different (random) accounts are shown. The strategy employed by PoC replaces the leaf node values corresponding to the account of the user exercising the right to erasure by a null key-value pair.

The leaf node contains two fields, the first field contains the last Keccak256 hash values of the account address, which is the remaining search path, and the second field contains the state of that account (balance and other data). In the PoC, the leaf node corresponding to the account to be deleted is replaced by null values (0xc28080 which corresponds to the RLP encoded of the null key-value pair ['0x', '0x']).

3. If the transaction also corresponds to a *Smart Contract* call, it is necessary to check whether the account also appears in the internal storage of the *Smart Contract*¹². To do this, the *storage trie* of this *Smart Contract* is also obtained for each block concerned. We proceed in a similar way to the *State Trie* but taking into account that the search indexes in the Storage Trie depend on the type of variable and its order of appearance in its declaration in the source code¹³. It is therefore necessary to know in detail the source code of each *Smart Contract* concerned. Two *Smart Contracts* have

¹² Although a *Smart Contract* can implement functions to erase its stored data at a certain point in time (by calling an erase function), this data still exists on the blockchain as part of the stored information resulting from the execution of transactions in previous blocks. Given the structure of a *Smart Contract*'s data storage on the blockchain, *Smart Contracts* need to be known (source code) in order to be able to effectively erase their stored data.

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

been used in the PoC, one of them with an *address* type variable and the other with a *mapping* type variable (in where the account in question will appear in these variables).

Key in *leveldb*, where *p* is the order of appearance in the source code (not counting constant variables and counting from 0):

Simple variable: *address*, Key = Keccak256(*p*)

mapping (*address*=>uint256), Key = Keccak256(Keccak256(*address* concatenated with *p*))

Figure 4. Key in the Storage Trie depending on variable type and order in the Smart Contract

4. In each affected block, we obtain the *receipts* of the transactions, and overwrite the value of the account in question by the hexadecimal string 'aaa...a' in a similar way to the transactions in the *blockbody*.
5. Finally, we remove the *keystore* account from the *keystore* storage of the nodes that include it.

The database modified at this point will be the basis for inconsistency management and will be used to implement the *Hard Fork* of the blockchain.

D. CONSIDERATIONS FOR THE PROCEDURE OF GENERATING A NEW SOFTWARE VERSION

This procedure consists of generating the new version of the Blockchain infrastructure software, where the key-value pairs of the modified database are incorporated, as well as the validator node agreement mechanism, inspired by Bitcoin's BIP-0009, as described in the Technical Note. The PoC implements a simplification of this in the *geth* code.

Since in the *clique* protocol the *MixDigest* field of the blocks is not used, and always has an empty value, it will be the one used to signal the support of the validator nodes to the *Hard Fork*. Once agreement is reached, this field will be modified to reflect this fact as well, which will be what a node that re-synchronises or joins the network will check.

```
// 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")
```

Figure 5. Source code of *geth* showing the calculation of the *MixDigest* field of a block header and an error function if it is not null. In the PoC the field is modified to include the reference to support for the new version or agreed new version. The error function is removed in the PoC.

The PoC implements the necessary functions in the validator nodes to keep track of a certain number of the last blocks added to the chain, where the *MixDigest* field contains the value corresponding to the acceptance or support of the new version (in the PoC the value of this field is the version number, e.g. 1). In addition, it is necessary to implement also the modification in the local database of the validator node when the agreement on the new version is reached. This agreement is reached when the count of the number of blocks accepting or supporting the new version reaches a majority (4 blocks out of the last 5 in the PoC, with 2 validator nodes).

On the other hand, it is also necessary to implement the synchronisation of a new node, or a node that re-synchronises from a previous state of the Blockchain/table. In both cases, such a node runs the new software version and checks the *MixDigest* value of the last block

to be downloaded, which, if it already contains the value of the accepted and executed modification, the node then modifies its local database.

The *geth* functions that have been modified can be found in the following files:

- `aepd.go`: Auxiliary file to implement some variables and write functions in the node's local database and consensus checking on the blocks being added.
- `db.go`: Auxiliary file to incorporate the changes in the local database of the node that execute the deletion right. I.e., the affected and necessary values in which the original value of the address of the account in question has been overwritten by a constant value 'aaaa...a' and those corresponding to the balance of said account.
- `cmd/geth/main.go`: Implements the functionality of a node that re-synchronises with the modified network, modifies the local database and restarts with those changes applied. To do this, a listening process is added to the events of the *downloader* component, which checks the blocks to be downloaded/synchronised, and the *MixDigest* field.
- `consensus/clique/clique.go`: Modifies the *MixDigest* field, with a value equal to the current software version or an accepted version indicator when agreement is reached from the validator nodes.
- `core/blockchain.go`: Implements an interface to clear the node's cache when updating the database (after the validators' agreement or by resynchronisation of the new chain). In this way, any console or application connected to the node will not retrieve the original transactions from the cache, but the modified ones already present in its local database.
- `eth/api_backend.go`: Implements an interface to access the node's local Blockchain/table.
- `eth/downloader/downloader.go`: Checks the values of the *MixDigest* field of the headers of the blocks being downloaded, when it finds that the modification has been made (accepted and executed on the new Blockchain/table) then sends a message to the node of this fact, so that it modifies its local database, and restarts synchronising with the new Blockchain/table.
- `eth/downloader/events.go`: Add a new event to indicate if the change has occurred.
- `miner/miner.go`: Implements an interface to expose the Blockchain/table of the validator node.
- `miner/worker.go`: Call the `aepd.go` function that implements the BIP-0009 approach after validating a new block.