B-CoC: A Blockchain-Based Chain of Custody for Evidences Management in Digital Forensics

Silvia Bonomi
Research Center of Cyber Intelligence and Information Security (CIS),
Department of Computer, Control, and Management Engineering "A. Ruberti",
Sapienza Università di Roma, Via Ariosto 25, 00145 Rome, Italy
https://www.cis.uniroma1.it
bonomi@diag.uniroma1.it

Marco Casini
Department of Computer, Control, and Management Engineering "A. Ruberti",
Sapienza Università di Roma, Via Ariosto 25, 00145 Rome, Italy
casini.1724011@studenti.uniroma1.it

Claudio Ciccotelli
Research Center of Cyber Intelligence and Information Security (CIS),
Department of Computer, Control, and Management Engineering "A. Ruberti",
Sapienza Università di Roma, Via Ariosto 25, 00145 Rome, Italy
https://www.cis.uniroma1.it
ciccotelli@diag.uniroma1.it

Abstract

One of the main issues in digital forensics is the management of evidences. From the time of evidence collection until the time of their exploitation in a legal court, evidences may be accessed by multiple parties involved in the investigation that take temporary their ownership. This process, called *Chain of Custody* (CoC), must ensure that evidences are not altered during the investigation, despite multiple entities owned them, in order to be admissible in a legal court. Currently digital evidences CoC is managed entirely manually with entities involved in the chain required to fill in documents accompanying the evidence. In this paper, we propose a Blockchain-based Chain of Custody (B-CoC) to dematerialize the CoC process guaranteeing auditable integrity of the collected evidences and traceability of owners. We developed a prototype of B-CoC based on Ethereum and we evaluated its performance.

2012 ACM Subject Classification Applied computing → Computer forensics; Applied computing → Evidence collection, storage and analysis

Keywords and phrases Digital Forensics, Chain of Custody, Digital Evidence, Private Blockchain, Ethereum

Digital Object Identifier 10.4230/OASIcs.Tokenomics.2019.12

Related Version A technical report is available at https://arxiv.org/abs/1807.10359

Acknowledgements This work has been partially supported by the Sapienza Ateneo 2017 project INOCS.

1 Introduction

One of the main issues in digital forensics is the management of evidences. From the time of evidence collection until the time of their exploitation in a legal court, evidences may be accessed by multiple parties involved in the investigation that take temporarily their ownership. The *Chain of Custody* is the process of validating how any kind of evidence has been gathered, tracked and protected on its way to a court of law. Chain of Custody (CoC) is not a mandatory step in forensic analysis. However, it is extensively used as evidences,

to be acceptable in a court or in legal procedures, must be proved to be not altered during investigations. Thus, a good CoC process should use a standard for dealing and handling evidences (digital or not), regardless of whether the evidence will be used in a trial or not.

The main requirements of a CoC process are:

- **Integrity**: the evidence has not been altered or corrupted during the transferring.
- **Traceability**: the evidence must be traced from the time of its collection until it is destroyed.
- **Authentication**: all the entities interacting with an evidence must provide an irrefutable sign as a recognizable proof of their identity.
- **Verifiability**: the whole process must be verifiable from every entity involved in the process.
- **Security – Tampering proof**: Changeovers of an evidence cannot be altered or corrupted.

Currently, CoC process requirements are met by employing a physical handover of evidences where, at each step, documents are filled in and signed in front of officers. In this paper, we take a step toward the dematerialisation of this process by proposing a Blockchain-based architecture for CoC of digital evidences called *B-CoC*. Leveraging on the features offered by blockchain technologies, we defined an architecture able to support the CoC process. To this aim, we proposed an architecture, namely B-CoC, that is able to realise an Evidence log with integrity checks (i.e., every process is able to verify and detect if there has been an integrity breach that would invalidate the digital evidence). B-CoC integrates together an ordinary database with a permissioned blockchain: the first represents the *Evidence DB* where digital evidences are stored, while the second represents the *Evidence Log* that allows to track digital evidences during their lifecycle. This distinction is done to store each type of information in the most suited kind of distributed storage: digital evidences are quite static and large piece of information and do not need particular support for updates while the evidence log is characterised by a reduced size of record to be stored and is subjected to a high update frequency.

In particular, we set up a private permissioned blockchain and we implemented a smart contract to keep track of the ownership changes during the evidence lifecycle. We implemented our prototype on an Ethereum [9] private network and we evaluated the impact of the system configuration parameters on performance.

2 Background

2.1 Blockchain technology

The blockchain technology implements a decentralized fully replicated append-only ledger in a peer-to-peer network, originally employed for the Bitcoin cryptocurrency [7]. All participating nodes maintain a full local copy of the blockchain. The blockchain consists of a sequence of blocks containing the transactions of the ledger. Transactions inside blocks are sorted chronologically and each block contains a cryptographic hash of the previous block in the chain. Nodes create new blocks as they receives transactions, which are broadcast in the network. Once a block is complete, they start the consensus process to convince other nodes to include it in the blockchain. In the original blockchain technology employed in Bitcoin the consensus process is based on *Proof-of-Work* (PoW) [7]. With PoW nodes compete with each other in confirming transactions and creating new blocks by solving a mathematical puzzle. While solving a block is a computational intensive task, verifying its validity is easy. To incentivize such mechanism, solving a block also results in mining a certain amount of

bitcoins, which is the reward for block creators (usually referred to as *miners*). Sometimes, more than one miner may generate a valid block thus creating forks in the chain. Forks are solved by accepting only the longest branch as the valid continuation of the chain (thus eliminating forks eventually). The main advantage of PoW, over traditional consensus algorithms, is that an attacker would have to control the majority of the computational power of the network, rather than the majority of the nodes, which is considered more difficult and virtually impossible in public large-scale networks.

The main criticism to PoW is its huge demand of energy, which also prevents its applicability in certain contexts. This has led to the investigation of alternative forms of consensus for the blockchain, such as *Proof-of-Stake* [5]. With PoS, a set of nodes, called *validators*, take turns proposing new blocks and voting on them. Validators put a stake in the network (e.g., a given amount of cryptocurrency) and are incentivized to act honestly so as not to lose the stake. Indeed, the blockchain keeps track of the set of validators, which are ousted if they behave maliciously (thus losing their stake).

A specific type of PoS is *Proof-of-Authority* (PoA) in which individual's identity (rather than cryptocurrency) is at stake. With PoA validators must have been preventively authorized and their identities are known. Thus, acting maliciously results in losing personal reputation and ultimately in being expelled from the validator set.

While PoW is particularly suited for *public* networks, both PoS and PoA may be suitable for *private* networks (where PoW would probably fail short as it would be much easier to control the majority of the computational power). Moreover, PoW and PoS can be used in *permissionless* networks, that is, networks where nodes can freely join the network without previous authorization (e.g., as in Bitcoin and Ethereum). PoA, on the other hand, is typically employed in *permissioned* blockchain networks, that is, networks in which nodes cannot freely join and become validators, but rather they have to be preventively authorized.

2.2 Ethereum and Smart Contracts

Etherium [9] can be seen as a decentralized virtual machine based on the blockchain technology. The Ethereum Virtual Machine (EVM) runs programs, referred to as *smart contracts*, whose state is stored in the Ethereum blockchain. Every node execute a local EVM. When an account wants to execute a function of a smart contract, it issues a transaction which is broadcast to the network. Each node executes the transaction on its local EVM and stores it, along with the new computed state, in the blockchain.

In Ethereum each EVM instruction consumes a virtual resource referred to as *gas*.

Gas can be seen as the fuel of the EVM and is employed to incentivize miners to execute transactions and include them in the blockchain. Indeed, for each transaction, miners are rewarded by the issuer with the payment of fees proportional to the total amount of gas "consumed" to execute that transaction.

To prevent mined blocks from becoming too large, which may severely impact block propagation and processing latency, each block has a *block gas limit*, which is the maximum amount of gas all transactions included in the block are allowed to consume. Thus, an issued transaction may not be included in the current block by a miner because it would exceed the block gas limit. In such case, the issued transaction would have to wait until the next block creation.

The public Ethereum blockchain (often referred to simply as "Ethereum") is a public permissionless networks which adopts PoW as consensus algorithm (even though it is planned to switch to PoS in the future). However, all major Ethereum implementations [1, 3] allow to configure many aspects of the protocol, such as the actual consensus algorithms employed, and allow to build custom public/private permissionless/permissioned blockchain networks.

2.3 Istanbul BFT consensus protocol

Istanbul Byzantine Fault Tolerance (IBFT) [2] is an adaptation of the Practical Byzantine Fault Tolerance (PBFT) [6] algorithm to serve as a PoA consensus algorithm for the Ethereum protocol. IBFT can tolerate at most f faulty validators out of a total of $n = 3f + 1$ validators. The IBFT algorithm proceeds in rounds with a new block created every T seconds, where the *block period* T is a constant configuration parameter. In each round one of the validators is elected as the *proposer*. The proposer creates the new block and broadcasts it to all validators with a *pre-prepare* message. Upon receiving pre-prepare messages, validators enter the *pre-prepared* phase and broadcast *prepare* messages. This, ensures that validators are aligned to the same round and block. Upon receiving $2f + 1$ prepare messages, validators enter the *prepared* phase and broadcast *commit* messages to inform other validators that they accept the proposed block. Finally, upon receiving $2f + 1$ commit messages, validators enter the *committed* phase and insert the block in the blockchain.

3 System Model

CoC model. A digital evidence (or electronic evidence) is any probative information stored or transmitted in digital form that a party may use in a trial to a court case. Digital evidences are collected by authorised parties (usually police officers) that become their temporary (first) owners.

For the sake of presentation and without loss of generality, in the following we will consider a single digital evidence d_ev collected by an authorised entity e_0 that holds its ownership. During investigations, several authorised entities (e.g., police offices, lawyers, judges, magistrates, etc.) may need to access, acquire and/or own temporarily d_ev. The set of authorised entities that can interact with d_ev is denoted with A_{d_ev}. Each authorised entity has a unique identifier known to all and he/she owns credentials that allows him/her to be authenticated and take actions in the CoC process.

At each time t, d_ev can have just one *owner* and the owner must belong to A_{d_ev}. If an authorised entity e_i needs to acquire and own d_ev, the current owner needs to issue a *transfer* request towards e_i. The change of ownership happens if and only if $e_i \in A_{d\_ev}$ and the transfer record is written permanently in the *evidence log*.

Network Model. The system is composed by a set of processes $p_1, p_2, \ldots, p_n$, one for each authorised entity in A_{d_ev}. Each process p_i has a pair of private-public keys that it uses to authenticate itself and to sign messages. Processes are connected trough a peer-to-peer network (authenticated perfect links). We consider that authorised entities are trusted but up to f of them (with $n > 3f$) can be compromised and controlled by an adversary i.e., they may behave as a Byzantine process deviating arbitrarily from the protocol.

4 B-CoC Architecture

The Blockchain-based Chain of Custody (B-CoC) architecture proposed in this paper is based on a *private* and *permissioned* blockchain. This choice has been driven by the *authentication* requirement of the CoC process that does not allow unauthorised and untrusted parties to manage digital evidences and thus to be in the network.

As shown in Figure 1, B-CoC is composed mainly of three components: (i) the *Evidences DB*, (ii) the *Evidence Log* and the (iii) *Frontend* interface. The Evidence DB is an ordinary database and/or file repository where we store the actual digital evidences, while CoC related

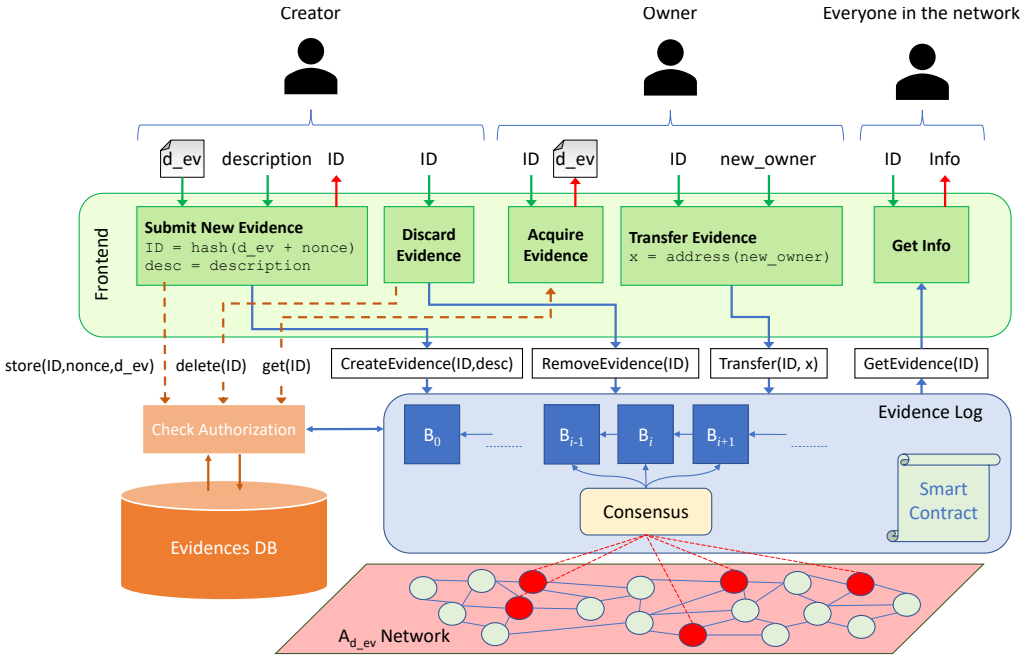

Figure 1 B-CoC architecture.

data are stored in the Evidence Log, which is implemented through the blockchain technology. The reason for this separation is twofold. First of all, evidences can be too large to be efficiently stored in the blockchain (for example, an evidence may be a bit-by-bit copy of a storage device of several TBs of capacity). Secondly, and most importantly, if evidences were stored in the blockchain, every node in the blockchain network would have access to them, while only authorized nodes should be allowed to acquire an evidence. Therefore, we store in the blockchain only the information regarding the CoC process and an hash of the evidence which allows to verify evidences integrity during acquisition.

Evidence DB. The Evidences DB is an ordinary distributed database and/or file repository where the original digital evidence is stored along with an identifier `ID`, obtained as the hash of the evidence and a nonce (to guarantee uniqueness of `ID`s). This database is distributed and is managed by trusted entities (e.g., law court officers). Moreover, each access is executed only if the requesting entity is authorized to perform such access according to its role.

Evidence Log. The Evidence Log is implemented trough the blockchain technology and stores, for each evidence, its `ID`, a description, the identity of the submitter (which we call *creator*) and the complete history of owners up to the current one, including the time at which changes of ownership occurred. Note that while the evidence itself is not stored in the blockchain, the `ID` allows to verify that the evidence has not been tampered with, provided that a robust cryptographic hash function is used to generate it.

The evidence log is implemented on top of a peer-to-peer network composed by all authorised entities. Such network can be decomposed in two sets of nodes:

- *Validator nodes*: they have mainly the following functionalities: (i) storing a copy of the blockchain, (ii) validating transactions and (iii) create, propose and add blocks to the chain (i.e., participate to the consensus protocol). This is the set of nodes that must be preventively authorized with the role of validators in the permissioned blockchain.

- *Lightweight nodes*: they can be seen as clients of the chain since they simply issue transactions and need to rely on validators for adding and validating their transactions.

Taking Italy as a use case, each validator may correspond to the main coordinator of the court of one of the 20 regional capitals. Lightweight nodes, instead, would represent all the other involved parties such as police departments, forensic investigators, forensic consultants and so on. The Evidence Log runs a smart contract which exposes four primitives (see Figure 1):

- `CreateEvidence(ID, description)`: stores a new evidence entry in the blockchain with the specified `ID` and `description`, setting the submitter identity as the *creator* and current *owner* of the evidence.
- `Transfer(ID, newowner)`: transfers the ownership of an evidence (registering the handover). It fails if the issuer is not the current owner.
- `RemoveEvidence(ID)`: removes an evidence entry. It fails if the issuer is not the creator.
- `GetEvidence(ID)`: returns the information in the evidence entry. Namely, the `ID`, description, creator and all owners with the time of each change of ownership.

Implementation details of the Evidence Log and the smart contract are discussed in Section 5.

Frontend Interface. The frontend represents the interface between B-CoC and its users. A local instance runs on each node and interacts with the Evidences DB and the Evidence Log (through a local blockchain node). When an authorized user submits a new digital evidence *d_ev* to the system, he/she takes the role of *creator* of *d_ev* (see Figure 1). The frontend generates the `ID` for *d_ev* using a nonce n, sends the command `store(ID, n, d_ev)` to the Evidence DB and issues the `CreateEvidence()` transaction in the Evidence Log. As already discussed the submitter is also registered as the first owner in the blockchain. When the *Check Authorization* component of the Evidence DB receives the `store(ID, n, d_ev)` command, it starts to monitor the Evidence Log for the corresponding `CreateEvidence()` transaction. Only upon confirmation that this transaction has been inserted in the Evidence Log, the Check Authorization component actually stores the pair (ID, n, d_ev) into the Evidence DB.

The creator of an evidence *d_ev* can request to discard it from the system (e.g., because it is no more legally valid). If he/she is authorized to do so, the corresponding entry is removed from the Evidence Log by issuing the `RemoveEvidence()` transaction. If the transaction succeeds, the corresponding evidence is deleted from the Evidence DB by issuing the `delete` command. Upon receiving the `delete` command the Check Authorization component of the Evidence DB checks if the corresponding `RemoveEvidence()` transaction has been inserted in the Evidence Log. If the transaction is not present, the `delete` command fails and sends an error response to the frontend.

When a user wants to acquire an evidence *d_ev*, the Frontend sends a request to the Evidences DB which will serve the request only if the user is the current owner of *d_ev*. This check is performed by the Check Authorization component by interacting with the Evidence Log.

The change of ownership of an evidence *d_ev* is performed by issuing a `Transfer()` transaction specifying the new owner. Note that this operation does not involve the Evidence Log in any way.

Finally, every user in the B-CoC network can query the Evidence Log to get the entry of an evidence (which contains all relevant information except the evidence itself). This is performed by simply issuing the `GetEvidence()` transaction.

5 Evidence Log Implementation

As described in Section 4, B-CoC Evidence Log is designed as a private and permissioned blockchain. The blockchain infrastructure is implemented through *Geth* [1] a popular implementation of a full Ethereum node. Geth allows to setup a private network and configure all aspects of the blockchain and the consensus protocol employed. Given the design of a private permissioned blockchain we adopt a PoA-based consensus. Namely, the IBFT consensus protocol described in Section 2.3. Let us note that, at the time of our development, IBFT was the only Byzantine tolerant consensus protocol available using Geth. On top of this blockchain infrastructure, we run a smart contract implementing the CoC process. The choice of implementing B-CoC using Geth has been driven by a cost-benefit analysis. We considered several blockchain technologies, namely full Ethereum, Geth with PoA consensus and Hyperledger Fabric, and we evaluated them against the requirement of our application. None of them is currently matching perfectly our needs but we believe that Geth with PoA consensus is the most appropriate given its ease of adaptation, deployment and complexity. The implementation of B-CoC Evidence Log involves three steps: (i) the initialization of the private blockchain, (ii) the creation of the private network and (iii) the creation and deployment of the smart contract.

5.1 Private chain initialization

The setup of a new blockchain involves the creation of its *genesis* block. This is the first block of a blockchain and contains the initial parameters. The only configuration parameters that are of interest for the purposes of the following discussion are:

- *Block Period T*: the block period of the IBFT consensus algorithm (see section 2.3);
- *Block Gas Limit G*: Maximum amount of gas transactions in a block are allowed to consume (see section 2.2);
- *Validators*: The Ethereum addresses of the pre-authorized validators.

The genesis block is used to initialize each node of the network.

5.2 Private network setup

First of all, to build the private peer-to-peer network we need to setup the peer discovery service to allow new nodes to enter the network and know other nodes. This is accomplished with the `bootnode` tool (of the Geth tools suite). This tool allows to run special nodes (with known IP addresses) that validators and lightweight nodes will contact when first started to exchange peer information.

Validators and lightweight nodes are Geth nodes. First, we configure the set of validators (which is fixed and known in advance) with the genesis block and we run them through the `geth` command (of the Geth tools suite). Validators are created once at the beginning and they never leave the network, unless they act maliciously and are expelled. Lightweight nodes, instead, can be created and join/leave the network at any time. They are created with the `geth` tool as well, but their addresses are not included in the genesis block.

5.3 Smart contract implementation

The smart contract has been implemented through the Solidity contract-oriented programming language [4]. Due to space constraints, the code of the smart contract is reported in the Appendix. The smart contract manages entries associated to digital evidences (i.e., the entries of the Evidence Log). Each `Evidence` entry (lines 3-10) consists of the `ID`, the Ethereum

Table 1 Size and gas used by each transaction.

TX	size(TX) (bytes)	gas(TX) (units)
`CreateEvidence(0)`	207	170207
...	...	...
`CreateEvidence(1024)`	1233	897367
`Transfer()`	174	80502
`RemoveEvidence()`	142	236478

address of the creator, the address of the owner, a string field to store the description of the evidence and two arrays `taddr` and `ttime` that store, respectively, the evidence handovers and the times at which they occurred. These arrays are chronologically sorted from the creator to the current owner. All evidence items are stored in a map indexed by evidence `IDs` (line 11). The smart contract has a total of four functions implementing the primitives of the Evidence Log described in Section 4. The `CreateEvidence(ID, description)` function creates a new `Evidence` entry with the specified `ID` and `description`, and the address of the related transaction sender as the creator and current owner of the evidence (line 26). The `Transfer(ID, newowner)` function transfers the ownership of the evidence identified by `ID` to the entity identified by the address `newowner` (line 35). Note that only the current owner of an evidence can transfer ownership (`OnlyOwner` modifier). The `RemoveEvidence(ID)` function removes an evidence from the map of evidences (line 41). No further operations can be performed on a removed evidence. Note that only the creator of an evidence can remove the evidence (`OnlyCreator` modifier). The `GetEvidence(ID)` function returns all fields of an evidence entry (line 46).

Note that while calling the first three functions results in issuing transactions to the blockchain that modify the state of the smart contract, the `GetEvidence` function only returns an entry and does not modify the state. In the context of the Solidity language these are called *constant* functions or *views*. Calling views does not result in the issuing of transactions, but rather they are executed locally by the node's local EVM.

6 Evaluation

In this section we evaluate how the parameters of B-CoC, namely the block period T and the block gas limit G, affect its performance. This analysis allows to guide the choice of the most appropriate configuration parameters in each scenario, as discussed in Section 7. Section 6.1 reports an analysis of the transaction latency, Section 6.2 evaluates the space overhead due to block headers and Section 6.3 discusses the growth rate of the blockchain.

Notation. In the following sections we will use the notation TX to refer to a *transaction type* (i.e., a non-constant function of a smart contract) and tx to refer to an *execution of a transaction*. For example, TX may refer to the transaction type `Transfer()`, while tx may refer to an actual execution of a `Transfer()` of an evidence. We will use gas(tx) and size(tx) to indicate, respectively, the gas consumed by the execution of a transaction tx and the size (in bytes) of tx when included in a block. Note that, in general, both gas(tx) and size(tx) depend on the particular execution of tx. In practice, for our smart contract, transaction types `Transfer()` and `RemoveEvidence()` have constant size and gas used, while for `CreateEvidence()` such parameters depend exclusively on the length ℓ of

the `description` parameter. Thus, for ease of presentation we will consider a different transaction type `CreateEvidence(`ℓ`)` for each value of ℓ. Since we limit the length of the `description` parameter to 1024 characters, we consider 1025 different transaction types ($\ell = 0, \ldots, 1025$). Thus, each transaction type has constant size and constant consumed gas and, therefore, we will consider $\text{size}(TX) = \text{size}(tx)$ where tx is an execution of TX and use the two members of the equation interchangeably, as well as $\text{gas}(TX) = \text{gas}(tx)$. We will refer to $\mathcal{SC} = \{TX_1, \ldots, TX_n\}$ as the set of transaction types of the smart contract. Table 1 reports the size and gas of the transaction types in our smart contract. Due to space constraints Table 1 only shows `CreateEvidence(`ℓ`)` for $\ell = 0$ and $\ell = 1024$, but the size and gas used by such transaction types increase with ℓ.

6.1 Transaction latency

The transaction latency $L(tx) = L_{\text{B}}(tx) + L_{\text{C}}(b)$ is the time elapsed from the issue of the transaction to its inclusion in the blockchain. It is the sum of the *block inclusion latency* $L_{\text{B}}(tx)$, that is the time required by tx to be included in a block b of the current proposer, and the *consensus latency* $L_{\text{C}}(b)$, which is the time required to reach consensus on block b and include it in the blockchain: In the next two sections we will analyze these two terms separately.

6.1.1 Block inclusion latency

The block inclusion latency $L_{\text{B}}(tx)$ is the time required for a transaction tx to be included in a block. Indeed, whenever a new transaction is issued it may not *fit* in the block of the current proposer due to the block gas limit G. In such case, the transaction is reissued in the next block period.

More formally, let $\text{block}(tx)$ be the block in which, eventually, transaction tx is included, and $\text{time}(tx)$, $\text{time}(b)$ be, respectively, the time at which tx is issued and the time at which a block b was created (i.e., the beginning of b's block period), then:

$$L_{\text{B}}(tx) = \text{time}(\text{block}(tx)) + T - \text{time}(tx) \tag{1}$$

where T is the block period.

The block inclusion latency is affected by the block period parameter T, the gas limit G and the workload, i.e., the rate of transactions issued to the system in the unit of time. Suppose that we are able to precisely characterize the workload the system is subject to and to set G such that every issued transaction is included in the block of the current proposer. In such ideal conditions, $L_{\text{B}}(tx) \in [0, T]$. That is, the maximum block inclusion latency is the block period T. Clearly, setting $G = \infty$ would meet the ideal conditions for every possible workload volume, but on the other side would negatively impact consensus latency, as blocks could increase indefinitely (see next section). Thus, we would like to set G as small as possible (to reduce consensus latency), but large enough so that (at least on average) every transaction is included in the block of the current proposer. Thus the ideal value of G depends on the workload. However, rather than the number of transactions per seconds, it depends on the *gas rate*, i.e., the amount of gas consumed by the transactions issued in a given block period. Indeed, the minimum value of G that guarantees the ideal conditions for the block inclusion latency is the maximum gas rate.

Figure 2 shows the results of three experiments that confirm the previous claim. In each experiment we set a different value of the block gas limit (G_1, G_2 G_3) and we progressively increased the gas rate from the start to the end of the experiment. From the figure we

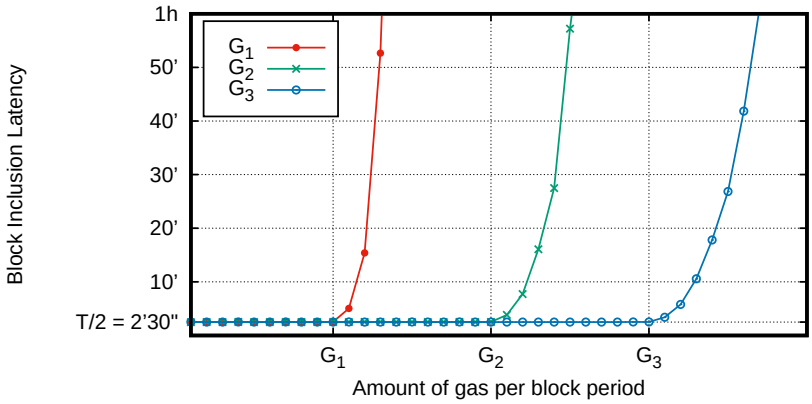

Figure 2 Mean block inclusion latency varying the *gas rate*, i.e., the amount of gas consumed by transactions in a block period.

can clearly see that, in each experiment, when the gas rate is less than or equal to the gas limit the average block inclusion latency is approximately equal to the expected value of $T/2$ (because transactions are issued uniformly distributed in each block period) while the maximum latency is T (not shown in the figure). However, as soon as the gas rate exceeds the block gas limit the average block inclusion latency starts increasing indefinitely as expected.

This analysis provides a lower bound for the value of the block gas limit (i.e., the maximum gas rate), that allows to minimize the maximum block inclusion latency to T. However, determining such value may be difficult. Section 7 reports a more general and detailed discussion on setting the parameters of B-CoC.

6.1.2 Consensus latency

Given the consensus protocol described in section 2.3, the consensus latency, i.e., the time required to propagate a block b between the validators and reach consensus, can be approximated by the formula $L_{\mathrm{C}}(b) \approx \frac{s_{\mathrm{PP}}(b)+s_{\mathrm{P}}+s_{\mathrm{C}}}{R}$, where $s_{\mathrm{PP}}(b)$ is the size of the *pre-prepare* message, s_{P} is the size of the *prepare* message, s_{C} is the size of the *commit* message and R is the bandwidth of the slowest communication channel between two validators nodes (bytes/sec). While s_{P} and s_{C} are constant, the pre-prepare message piggybacks the block b and thus $s_{\mathrm{PP}}(b)$ depends on size(b). Since R is typically a constant that depends on the infrastructure connecting the validator nodes, the only factor that we can adjust to control the latency is the size of a block.

The size of a block is the sum of the size of the transactions in it plus the size of the block header s_{H} (which is constant). In our prototype implementation of B-CoC, $s_{\mathrm{H}} = 1909$ bytes. The actual number and type of transactions in a block depends on many factors, including the block period T, the block gas limit G and ultimately the particular set of transactions sent during a given time period. Thus, in general, different blocks have different sizes. However, we can control the maximum block size $S^{\max}$, and thus the maximum consensus latency $L_{\mathrm{C}}^{\max}$, by adjusting the block gas limit G.

For a given value of G, the maximum block size $S^{\max}$ can be computed by solving the following optimization problem:

▶ **Problem 1** (UKP).

$$\begin{aligned} \text{maximize} \quad & \sum_{TX_i \in \mathcal{SC}} \text{size}(TX_i) \cdot x_i \\ \textit{subject to} \quad & \sum_{TX_i \in \mathcal{SC}} \text{gas}(TX_i) \cdot x_i \leq G \\ & x_i \in \mathbb{N}, \quad i = 1, \ldots, n \end{aligned}$$

where x_i is the number of times a transaction of type TX_i appears in the block of maximum size. The optimal solution $\{x_1^*, \ldots, x_n^*\}$ leads to the maximum block size:

$$S^{\max} = s_{\text{H}} + \text{OPT}_{\text{UKP}}(G) = s_{\text{H}} + \sum_{TX_i \in \mathcal{SC}} \text{size}(TX_i) \cdot x_i^*$$

Problem 1 is an instance of the well-known unbounded knapsack problem [8], where transaction types correspond to the items to fit in the knapsack, while transactions' size and consumed gas correspond, respectively, to items' value and weight. The block gas limit parameter G corresponds to the knapsack maximum weight.

While the general unbounded knapsack problem is NP-hard (with time complexity $O(nG)$), this particular instance turns out to be trivial. Indeed, it is easy to see that `Transfer()` *dominates* all other transaction types [8]. That is, given any block containing at least a transaction tx of type in $\mathcal{SC} \setminus \{\texttt{Transfer()}\}$, we can always replace tx with a sufficient number of `Transfer()` so as to obtain a better solution to Problem 1. For example, we can always replace a transaction of type `RemoveEvidence()` with a single `Transfer()` and obtain a solution that consumes less gas but have larger size. The same occurs if we replace `CreateEvidence(0)` with 2 `Transfer()`, or `CreateEvidence(1024)` with at least 8 `Transfer()`. This implies that the optimal solution of this instance of the unbounded knapsack problem corresponds to a block consisting of only `Transfer()` transactions. Therefore, let $g_{\text{T}} = \text{gas}(\texttt{Transfer()})$, $s_{\text{T}} = \text{size}(\texttt{Transfer()})$, the solution is simply given by:

$$S^{\max} = s_{\text{H}} + \left\lfloor \frac{G}{g_{\text{T}}} \right\rfloor \cdot s_{\text{T}} \tag{2}$$

Note that $S^{\max}$ cannot be an arbitrary integer, but only one such that $S^{\max} = s_{\text{H}} + k \cdot s_{\text{T}}$, $k \in \mathbb{N}$.

Once chosen the appropriate value of $S^{\max}$ (one that allows to limit the maximum consensus latency to an acceptable bound), G can be set to any value such that:

$$G = \frac{S^{\max} - s_{\text{H}}}{s_{\text{T}}} \cdot g_{\text{T}} + r = k \cdot g_{\text{T}} + r, \qquad r = 0, \ldots, g_{\text{T}} - 1 \tag{3}$$

Equations 2 and 3 allows to determine an upper-bound for G so as to bound the maximum consensus latency $L_{\text{C}}^{\max}$. A more general discussion on how to properly set B-CoC's parameters is reported in section 7.

6.2 Block headers overhead

As discussed in section 6.1.1, the block period T affects transactions block inclusion latency. A longer block period implies higher latency. On the other hand, a shorter period results in a higher number of blocks created per time interval (since a block for each block period is

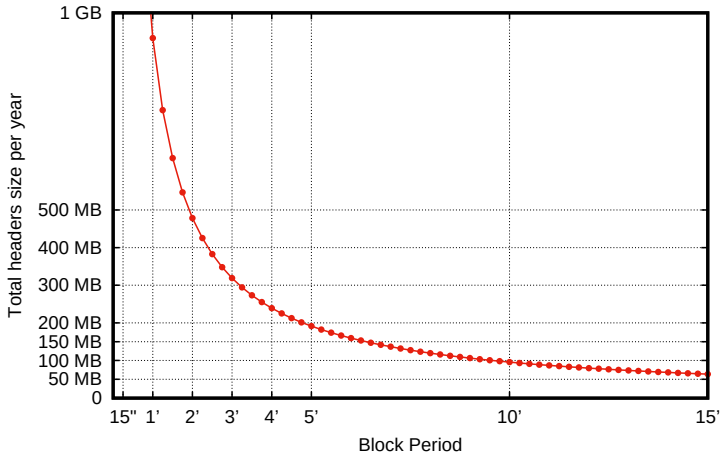

Figure 3 Total headers size per year for different block periods T.

created). Since each block has a fixed size header, the larger the number of blocks created, the higher the space occupied by block headers compared to transactions in the blockchain, that is, the higher the space overhead.

The headers size overhead, i.e., the total size of block headers, at any time t is:

$$\mathrm{OH}(t) = s_{\mathrm{H}} \cdot \frac{t}{T} \tag{4}$$

Note that this value only depends on the number of blocks in the chain at time t, and not on the number of transactions. Figure 3 shows the space overhead per year ($s_{\mathrm{H}} = 1909$ bytes in our prototype implementation), that is, how much blockchain's space is taken up by block headers every year. For example, for $T = 5$ minutes the space overhead is around 191 MB per year. We find this value of T a good trade-off between transaction latency and block headers overhead for this particular application of the blockchain.

6.3 Blockchain growth rate

The blockchain can be seen as an append-only database. That is, its size cannot shrink over time. If $I_{\mathcal{SC}}(t)$ is the set of transactions included in the blockchain at time t, then the blockchain total size at time t is:

$$\mathrm{size}_{\mathrm{bc}}(t) = s_{\mathrm{g}} + \mathrm{overhead}_{\mathrm{bc}}(t) + \sum_{tx \in I_{\mathcal{SC}}(t)} \mathrm{size}(tx)$$

where s_{g} is the size of the genesis block. Therefore, the growth rate over a time interval $[t_1, t_2]$ is $\mathrm{size}_{\mathrm{bc}}(t_2) - \mathrm{size}_{\mathrm{bc}}(t_1)$, that is:

$$\mathrm{GR}(t_1; t_2) = s_{\mathrm{H}} \cdot \frac{t_2 - t_1}{T} + \sum_{tx \in I_{\mathcal{SC}}(t_1; t_2)} \mathrm{size}(tx) \tag{5}$$

where $I_{\mathcal{SC}}(t_1; t_2) = I_{\mathcal{SC}}(t_2) \setminus I_{\mathcal{SC}}(t_1)$.

Obviously, how fast a blockchain grows over time depends mainly on the transaction rate. Another factor that affects the growth rate is the block period T. As already shown in section 6.2, this parameter affects the headers overhead and thus the first term of equation 5. The block gas limit parameter G may also affect the growth rate, as, if not properly

Table 2 Growth rate for different classes of workloads (n `CreateEvidence(1024)`, n `RemoveEvidence()`, $10n$ `Transfer()` per year).

Workload	GR − OH	GR with $T = 5'$	OH / GR (%)
$n = 10000$	29.7 MB/year	221.08 MB/year	86.56%
$n = 100000$	297 MB/year	488.45 MB/year	39.18%
$n = 1000000$	2.9 GB/year	3.09 GB/year	6.05%

dimensioned, it may increase latency spreading the incoming transaction rate over a larger time period, thus, decreasing the growth rate (i.e., G would affect the number and type of transactions included in $I_{\mathcal{SC}}(t_1; t_2)$ and thus the second term of equation 5). However, the analysis detailed in section 6.1, should allow to set the value of G so as to bound transaction latency. In practice, G should be set greater than the average gas rate to avoid an ever increasing latency. In this conditions, if the growth rate is computed over a large enough interval of time (to hide the effects of potential peak gas rate periods), the block gas limit parameter should not affect the growth rate significantly (that is, if properly set, G should not affect $I_{\mathcal{SC}}(t_1; t_2)$). Otherwise, G should be set to a larger value.

By using equation 5 we computed the annual growth rate for different classes of workloads. Since we were not able to find any publicly available statistics about evidence collection and transfer, we considered different classes of synthetic workloads with n new evidence creations and removals and $10n$ transfers per year. The results of this analysis are reported in Table 2. The second column of Table 2 reports the annual growth rate without considering the headers size overhead, while the third one includes the overhead term computed for $T = 5'$. Finally, the fourth column shows the overhead percentages. Even in presence of a very large number of evidence collection (1 million per year) and transfers (10 millions per year) the growth rate is around 3 GB per year, which seems acceptable given the capacities of todays storage devices.

7 Discussion on the configuration of the parameters

Section 6 discusses how the parameters of B-CoC affect its performance with respect to different aspects, namely the transaction latency, the block headers overhead and the blockchain growth rate. Here we give a comprehensive discussion on how to set B-CoC parameters appropriately.

7.1 Setting the block period T

The block period T affects transactions block inclusion latency (see section 6.1.1) and the block headers overhead (section 6.2), that ultimately affects the blockchain growth rate. As already discussed in section 6.2, a shorter block period results in a lower maximum block inclusion latency, but also in a higher block header overhead. To find the best trade-off one can use equation 4. For example, with our prototype implementation of B-CoC we consider $T = 5'$ to be a good trade-off between latency and block header overhead. Indeed, any further increase of T would result in a small improvement in terms of overhead reduction compared to the increase of latency (as shown in Figure 3).

7.2 Setting the block gas limit G

The block gas limit affects both term of the transaction latency. In particular in section 6.1, we describe an analysis that allows to derive a lower bound G_{L} for G, to limit block inclusion latency and an upper bound G_{U} to limit consensus latency.

When $G_L \leq G_U$ it is safe to set G equal to any value in $[G_L, G_U]$ to obtain a maximum block inclusion latency bound by T and the desired maximum consensus latency. On the other hand, if $G_L > G_U$, it is not possible to have both terms of transaction latencies bounded by the desired values. In such case, one should set G to the best trade-off between block inclusion latency and consensus latency. A good strategy may be to discard the lower bound G_L in favor of a new lower bound G_L^{avg} which is set to the average gas rate rather than the maximum gas rate. Setting $G = G_L^{avg}$ would result in block inclusion latency bounded by T on average, with possible periods of increasing latencies, e.g., during peak loads. In this case, if $G_L^{avg} \leq G_U$ one should set $G = G_U$, otherwise $G = G_L^{avg}$. Indeed, setting a value of G less than the average gas rate would result in ever increasing transaction latencies.

8 Conclusion

This paper presented B-CoC, a blockchain-based architecture to dematerialise the CoC process in digital forensics. We also provided a prototype of the B-CoC architecture based on the Geth implementation of Ethereum nodes. Based on the performance evaluation, B-CoC showed to be an effective support for the CoC process as it is able to sustain realistic workload with an acceptable overhead in terms of memory used to store the chain.

The current implementation assumes that the set of validators node is fixed and that validators are available to sacrifice their privacy when participating in the consensus process. As a future work, we are investigating how it is possible to manage a dynamic set of validators and most important we are studying alternatives that allow to increase the level of privacy for validators not altering other dependability and security attributes.

References

1 Geth. `https://github.com/ethereum/go-ethereum/wiki/geth`. [Online; accessed 30-May-2018].

2 Istanbul BFT. `https://github.com/ethereum/EIPs/issues/650`. [Online; accessed 17-July-2018].

3 Parity. `https://parity.io`. [Online; accessed 20-July-2018].

4 Solidity. `https://solidity.readthedocs.io`. [Online; accessed 11-June-2018].

5 Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies Without Proof of Work. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2016.

6 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=296806.296824`.

7 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: `http://www.bitcoin.org/bitcoin.pdf`.

8 Vincent Poirriez, Nicola Yanev, and Rumen Andonov. A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization*, 6(1):110–124, 2009. `doi:10.1016/j.disopt.2008.09.004`.

9 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

A Smart Contract Code

```
pragma solidity ^0.4.22;
contract ChainOfCustody {
  struct Evidence {
    bytes32 ID;
    address owner;
    address creator;
    string description;
    address[] taddr;
    uint[] ttime;
  }
  mapping(bytes32 => Evidence) private evidences;

  modifier OnlyOwner(bytes32 ID) {
    require(msg.sender == evidences[ID].owner); _;}
  modifier OnlyCreator(bytes32 ID) {
    require(msg.sender == evidences[ID].creator); _;}
  modifier EvidenceExists(bytes32 ID, bool mustExist) {
    bool exists = evidences[ID].ID != 0x0;
    if (mustExist)
      require(ID != 0x0 && exists);
    else
      require(!exists);
    _;}

  function CreateEvidence (bytes32 ID, string description)
      public EvidenceExists(ID, false) {
    evidences[ID].ID = ID;
    evidences[ID].owner = msg.sender;
    evidences[ID].creator = msg.sender;
    evidences[ID].description = description;
    evidences[ID].taddr.push(msg.sender);
    evidences[ID].ttime.push(now);
  }
  function Transfer(bytes32 ID, address newowner)
      public OnlyOwner(ID) EvidenceExists(ID, true) {
    evidences[ID].owner = newowner;
    evidences[ID].taddr.push(newowner);
    evidences[ID].ttime.push(now);
  }
  function RemoveEvidence(bytes32 ID)
      public OnlyCreator(ID) EvidenceExists(ID, true) {
    delete evidences[ID];
  }
  function GetEvidence(bytes32 ID)
      view public returns(bytes32, address, address,
        string, address [], uint []) {
    return(evidences[ID].ID, evidences[ID].owner,
      evidences[ID].creator, evidences[ID].description,
      evidences[ID].taddr, evidences[ID].ttime);
  }
}
```

Listing 1 Smart contract code.