Off-chain Execution of IoT Smart Contracts*

Diletta Cacciagrano, Flavio Corradini, Gianmarco Mazzante, Leonardo Mostarda, and Davide Sestili

Abstract Modern blockchains allow the definition of smart contracts (SCs). An SC is a computer protocol designed to digitally ease, verify, or enforce the terms of agreement between users. SCs execution can require high fees when lots of computation is required or a high volume of data is stored. This is usually the case of Internet-of-Things (IoT) systems where a large amount of devices can produce a high volume of data. Off-chain contract execution is a viable solution to decrease the blockchain fees. Users can agree on an on-chain SC which is stored in the main chain. Computation can then be moved securely outside the chain to reduce fees. In this paper we propose DIVERSITY a novel approach that allows off-chain execution of SCs. DIVERSITY provides a novel model for defining on-chain contracts that can be securely executed by using a novel off-chain protocol. We have validate our approach on a novel IoT case study where fees have been greatly reduced.

1 INTRODUCTION

The success of Bitcoin and its increasing use has placed blockchain as a promising solution for enabling trust in a decentralised system. Bitcoin[10] and Ethereum[17] are but a few examples of cryptocurrencies that make use of blockchain in order to enable payments without a trusted party (e.g., a bank). Modern blockchains also allow the definition of complex smart contracts (SCs). These automatically execute, control or document legally important events and actions according to the contract terms or the agreement. SCs are run by miners which can result in slow execution time (e.g., 10 minutes to mine a Bitcoin block while 20 seconds to mine an Ethereum one) and expensive fees [12, 13]. These issues are exacerbated in Internet-of-Things (IoT)

Diletta Cacciagrano · Flavio Corradini · Gianmarco Mazzante · Leonardo Mostarda · Davide Sestili
University of Camerino, e-mail: {name.surname}@unicam.it

* Diversity has been validated in the Italian national project industry 4.0.

systems, where a large amount of devices can produce a high volume of data which require real-time elaboration. There is a voluminous literature about IoT scalability methods and consensus protocols for blockchains [14]. As the authors in [11] show, IoT devices can require high monetary cost for storing and elaborating data and the time required to mine and validate blocks can be too high.

Off-chain computation seems a viable solution to decrease the blockchain fees while reducing the response time [12, 13]. Off-chain approaches are not bound to the transactional speed and high fee limitations of the on-chain transactions since they move the computation outside the main chain. Although off-chain computation can take several forms, it usually requires two or more users to agree on an on-chain SC which is stored in the main chain. Computation can then be moved securely outside the chain. In [7, 6] the authors explore different aspects about off-chaining techniques. In [7] they listed five different off-chaining patterns, such as performing a high number of microtransactions off-chain and then summing them up on a single on-chain transaction (e.g., Lightning Network [13]) or such as storing a large amount of data off-chain and putting only the reference on-chain (e.g., Swarm [16]). In [6] they analyse a couple of generic models for moving computation or storage off-chain. Then they drew up an overview of off-chain computation techniques and compared them with respect to scalability, privacy, security and programmability providing also some examples of implementations. In a different work [3] Eiberhart et al. propose an off-chain model in which SC computation is off-loaded to an external node to improve scalability and to enable private data to be used in SC execution. The main blockchain contains a verification SC which verifies the result of a computation received from the off-chain node along with a zkSNARK proof [9]. Their proposed model achieves greater scalability since on-chain verification is less computationally expensive than on-chain execution.

A noticeable contribution in the literature is the Lightning Network [13] which solves the scalability challenge of the Bitcoin blockchain. Lightning allows users to perform multiple microtransactions over an off-chain peer to peer network without any broadcasting on the main blockchain. Users can take back their balance on the main blockchain by providing a proof that their balance in the lighting network is the one claimed, if the proof is not challenged then they are allowed to take back their balance on the main chain. This protocol allows users to perform several microtransactions that would be otherwise prohibitively expensive in terms of fees if they were performed on the main blockchain. A similar proposal but for the Ethereum network is Plasma [12]. Its structure consists of a tree of blockchains. A root Ethereum blockchain can be connected to multiple Plasma child chains which can themselves have other child chains. The tree structure of the Plasma blockchain enables users to offload transactions and computation to child chains while enforcing correct behaviour of the child chains through fraud proofs. This can be filed to the parent blockchain in case malicious behaviour is detected, allowing them to roll back a block or retrieve their funds.

On the other hand, on-chain computation has its representatives in the literature. Since devices with limited storage capabilities cannot store ever-growing blockchains - such as Bitcoin - Richard Dennis et al. [4] proposed the concept of a rolling

blockchain, in which only data stored for a predetermined period of time is stored in the chain and any data older than that is automatically discarded.

In [15], Shahid et al. present two interesting aggregation approaches enabling blockchain footprint to scale gracefully on low memory sensor nodes. The first approach is time-based, and exploits a time window concept. Sensors store blocks for a certain time window, then they aggregate data into that time window and they sum them up in an aggregated block. The second approach consists in dividing the blockchain in geographical cells, where each node is only required to store blocks about nodes in its cell, thus saving storage. Finally, they combine the two approaches in order to optimize the blockchain footprint to its limit. These approaches have two main flaws, unfortunately. The first one is that they require to delete and re-create a new blockchain for each aggregation step. The other one is that even though storage space is preserved, IoT nodes are still required to perform all the computation needed for the maintenance of a blockchain.

In [5] the authors propose a proxy-based approach to the off-chain computation. They implement a blockchain proxy since the use of a blockchain is infeasible for a low-power IoT nodes. A Certification Authority issues an identity certificate for each sensor node, which in turn uses it to sign its blockchain transactions. Then the sensor sends the signed transactions to the proxy, which commits them to the ledger. Also, they prove the potential of their approach proposing an interesting use-case scenario about IoT sensors for cold-chain monitoring. The use of a proxy allows IoT nodes to sign their own transactions without requiring them to store blocks or perform heavy computation. Nevertheless, the proxy is easily identifiable as a single point of failure, and it can create scalability issues with growing sensor networks.

Joshua Elul et al. [8] focus on IoT behaviour definition through blockchain smart contracts. They proposed a split virtual machine architecture to enable IoT device programming through the Ethereum Blockchain. In their proposed architecture an IoT device communicates with a special node in the blockchain which looks for smart contracts containing code to be executed on IoT devices. The code is then sent to the device that will execute it. IoT devices can also request data to the special node when needed. The behaviour of the IoT device is encoded in the contracts bytecodes deployed on the main blockchain, this allows for IoT application code to be developed for the specific smart contract without requiring manual updating of the end IoT devices.

1.1 Contribution of the paper

This paper introduces DIVERSITY ($\mathcal{D}iv$), a novel second layer decentralised network that allows off-chain and secure execution of SCs. $\mathcal{D}iv$ is suitable when SCs require a large amount of CPU computation or high volume of data needs to be processed. In this cases $\mathcal{D}iv$ can lower the high fees required by the main chain and reduce the computation performed by the blockchain peers. The $\mathcal{D}iv$ protocol uses an on-chain SC as a trust and arbitration means in order to securely perform an off-

chain computation and notify any dispute. The on-chain structure also allow the definition of reactions when unanimous consensus is reached (e.g., the transfer of cryptocurrency). $\mathcal{D}iv$ allows users to define on-chain SCs with a well defined structure. A contract specifies sensors, off-chain computation to be performed and the intermediate nodes that will execute it. We assume each user can use its intermediate nodes although delegation mechanisms can be used. The intermediate nodes execute a secure protocol that verifies unanimous agreement on the result of the off-chain computation. An honest node that performs a correct computation will be enough to detect any dishonest ones and open a dispute. While any blockchain which supports the execution of SCs can be enhanced with the $\mathcal{D}iv$ novel approach, its current version supports Ethereum and the application case studies have been performed in the IoT area. In this context we assume there are some IoT devices (in the following refereed to as sensors) producing data plus two or more users that are interested in verifying and enforcing the terms of an agreement. Our contributions can be summarised as follows: (i) $\mathcal{D}iv$ provides a novel combination of a unique model for defining on-chain contracts with a protocol for off-chain secure execution; (ii) $\mathcal{D}iv$ intermediate nodes can run an off-chain code (e.g., Java) that meet the demands of specific use cases, specifically those that are not feasible on the considered blockchain (e.g., Ethereum); (iii) Its peculiar computation model is based on streams of data thus is suitable to IoT data processing; (iv) We have validate our approach on a novel lock case study where fees have been greatly reduced.

2 Motivation scenario

A smart lock is installed on different electric bicycles that are deployed in a smart city. A user can register on the service provider web site in order to buy a token k for renting the bicycles for a certain amount of time T_k. Every time a user u interacts with the locker of a bicycle b a message $m = [b,k,s,t]$ is received by the server where (i) s is equal to either u (i.e., lock unlocked) or l (lock locked); (ii) t is the time at which the operation was performed. Suppose that a token k is used to open a bicycle at time t_1 and the bicycle is closed at time t_2, then we denote with $t(k,t_1,t_2) = t_2 - t_1$ the time between a lock and an unlock operation (i.e., the time the bicycle was used). We denote with $T_d(k)$ the total amount of time the token k was used in a day. This can be defined as $T_d(k) = \sum_i t(k,t_i,t_{i+1})$ where all t_i belongs to the same day. An insurance company will bill the server provider monthly by considering the monthly bicycle rental time. This can be calculated as follows $T_m(k) = \sum_{k \in K_m} T_d(k)$ where K_m are all tokens that were used in the month m.

The first version of the system would store all lock and unlock operations at the service provider database. This would calculate the monthly bicycle rental time at the end of each month in order to pay the insurance company. This solution was immediately found inappropriate by the insurance that would rather prefer the use of the blockchain in order to ensure data immutability and avoid the need of a third trusted part. The main problem of the blockchain solution is that an high volume of

data would be stored in the blockchain. This would cost additional money that would increase the cost of the service of the final user. A SC for calculating the monthly bicycle rental time had also to be run at the end of each month. This would transfer the money from the wallet of the service provider to the insurance one. This would cost money as well. $\mathcal{D}iv$ was used to greatly reduce the amount of money paid by allowing off-chain and secure execution of the terms of agreement between insurance and service provider.

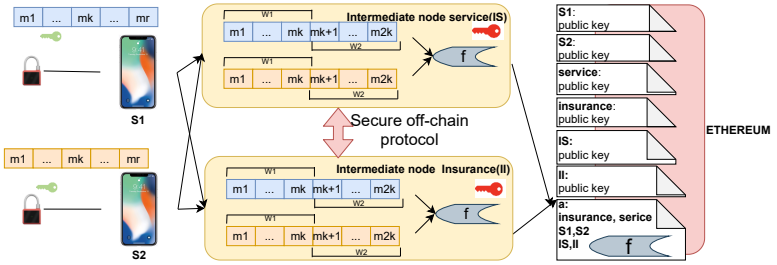

Fig. 1 $\mathcal{D}iv$ applied to the smart lock case study.

Figure 1 outlines the $\mathcal{D}iv$ architecture for the lock case study. The first step when using $\mathcal{D}iv$ is the definition of an on-chain SC. This specifies the following main parts: (i) the set of sensors that produce data; (ii) all users that are involved in the contract; (iii) the off-chain computation to be performed; (iv) a set of intermediate nodes that will perform the off-chain computation; (v) the quality of service. Users, sensors and intermediate nodes must be registered inside the main chain before the on-chain smart contract definition. Although they have different data, their registration always require a unique ID and a public key.

Figure 1 shows the lock case study where two locks (i.e., the sensors s_1 and s_2, the insurance and service intermediate nodes) have been registered. The on-chain SC a that contains the off-chain code f has also been registered. At run time, each sensor generates messages that contains its id, its type, some data and the sensor's signature. This is used in order to ensure integrity, authentication and non-repudiation of the messages. In our trust model we assume that sensors are trusted. Each sensor message is multi-cast to a set of intermediate nodes. Each intermediate node organises the sensor messages into windows that have a fixed-length size. A window is the basic computation unit that is used to compute the terms of agreement that is the off-chain code (in the following denoted with f). This has been agreed between two or more users that do not trust each other. The off-chain code can perform data aggregation and/or event generation and can output a value to be stored in the on-chain SC. $\mathcal{D}iv$ leverages on the use of two or more intermediate nodes for off-chain computation. Users can have their own intermediate nodes but they can also trust external ones. When external intermediate nodes are used fees can be transferred after their correct execution is validated. Intermediate nodes run a secure protocol that ensures the

following properties: (i) when all intermediate nodes perform the same off-chain computation on the same window (i.e., unanimous consensus is reached) the on-chain contract is updated and the reaction is run; (ii) if unanimous consensus is not reached a dispute is open. In our lock case study the off-chain code computes the monthly bicycle rental time and stores it inside the related on-chain contract. Insurance and service providers have their own intermediate nodes. This will run the off-chain code which is expected to give the same output for the same windows. Locks are installed by a third-party company.

$\mathcal{D}iv$ provides the following three basic quality of service: (i) no-proof of storage; (ii) off-chain proof of storage; (iii) on-chain proof of storage. These quality of services always store on the on-chain SC the output of a non-null computation and log any dispute that emerges. The quality of services differ for the policy they use to store sensor messages. In the no-proof of storage service, window data are discarded after the unanimous off-chain computation is reached. When a dispute is detected data are locally stored at the intermediate nodes for dispute resolution. This solution minimises the amount of data that are stored in the intermediate nodes and in the main chain. In the off-chain proof storage service, all data are locally stored by the intermediate nodes. When on-chain proof of storage is used data are stored on the on-chain contract when computation is not null or a dispute emerges.

3 $\mathcal{D}iv$ system model

In our system model we assume to have a set U of users that are interested in defining agreements. We denote with $u_1 \dots u_n$ elements in U. We assume to have a set of sensors S that produce data. We denote with $s_1, \dots, s_p$ elements in S. Each sensor s is declared into the main chain by using a declaration of the form d_s=$\{PU_s, c_s, type_1\ x_1, \dots, type_h\ x_h\}$ where PU_s is an asymmetric public KEY, c_s is the type of the sensor (such as temperature or light) while x_i is a variable of the type $type_i$. While running a sensor s must send messages that comply with its declaration. A sequence number is used to totally order the sensor messages. The q-th message is denote with $m_{s,q}$ and is of form $m_{s,q} = (PU_s||c_s||q||d_q) : s$ where q is the sequence number, $d_q = v_{x_1}, \dots, v_{x_h}$ is the array of values assigned to variable $x_1, \dots, x_h$ and $: s$ is the signature of the whole message performed by the sensor s. Each sensor s generates a local trace $Y_s = m_{s,0} \dots m_{s,q} \dots$. We assume to have a set of intermediate node I and we denote with $i_1, \dots, i_m$ elements in I. An intermediate node i can receive messages from various sensors thus it can observe various sensor traces.

We use a window as a basic unit of computation. A window is a sub-string of a sensor trace Y_s. We recall that a sub-string is a contiguous sequence of characters within a string. We define with $w(s,q,z)$ a window of z messages $(m_{s,q} m_{s,q+1} \dots m_{s,q+z-1})$ that belong to the sensor trace Y_s. We call q the starting point of the window. Our execution model is based on a sequence of windows. We denote with $W(s,q_0,sl,z)$ the sequence of windows $w(s,q_0,z), w(s,q_0+sl,z), w(s,q_0+2sl,z), \dots$ where $w(s,q_0,z)$ is the starting window and sl is a positive integer referred to as sliding factor. Effec-

tively, each window can be obtained by sliding the previous of sl messages. For the sake of presentation simplicity we denote with $w_s(t)$, with $t = 0$, the first window $w(s, q_0, z)$ while with $w_s(t)$ the window $w(s, q_0 + t \times sl, z)$ (i.e., the window after t slidings). Effectively, the notation $w_s(t)$ omits the sliding factor sl and window size z. Windows from a set of different sensors $S = \{s_1 \dots s_k\}$ can be organised into an array of windows. We denote with $w_S[t]$, with $t = 0$, the starting array $[w_{s_1,0}, \dots, w_{s_k,0}]$ while with $w_S[t]$ the array $[w_{s_1}(t), \dots, w_{s_k}(t)]$. We use $w[t]$ instead of $w_S[t]$ when the set S is clear from the context. We use $Q_0 = \{q_{s_1,0}, \dots, q_{s_k,0}\}$ to denote the window starting points.

We assume to have a set A of on-chain smart contracts and we denote with $a_1, \dots, a_n$ elements in A. Each smart contract a is a tuple $(U_a, S_a, Q_0, f_a, I_a, r_a) : U_a$ where: (i) $U_a = \{u_1 \dots u_n\}$, with $n > 1$, is a set of users with $U_a \subseteq U$; (ii) $S_a = \{s_1 \dots s_k\}$ is a set of sensors with $S_a \subseteq S$; (iii) $Q_0 = \{q_{s_1,0}, \dots, q_{s_k,0}\}$ contains the starting points of all windows; (iv) f_a is the off-chain computation function that takes as an input the array $w_{S_a}[t]$ of window $[w_{s_1}(t), \dots, w_{s_k}(t)]$ and outputs a result (i.e., a string) or null; (v) $I_a = \{(u_1, I_{u_1}) \dots (u_n, I_{u_n})\}$ is the set of intermediate nodes where for each tuple (u_i, I_{u_i}) each u_i is a user in U_a and $I_{u_i} \subseteq I$ is a set of intermediate nodes trusted by u_i; (vi) r_a contains a reactions that is a piece of code that is executed when f_a is not null; (vii) : U_a is the signature of all the users.

An SC a specifies a set of users $U_a = \{u_1 \dots u_n\}$ that are interested in the agreement. The terms of agreement concerns the data produced by the sensors in S_a. Each user $u_i \in U_a$ can specify a set of intermediate nodes that trusts for the computation. This is done by adding the tuple (u_i, I_{u_i}) to the contract where I_{u_i} is a set of intermediate nodes trusted by u_i. Each intermediate node can run the function f_a that takes as an input the array of windows $w_{S_a}[t]$ (in the following denoted with $w[t]$). When $f_a(w[t])$ is *null* no data will be written in the on-chain contract a otherwise the result will be store inside a SC associative map map_a. The unique execution entry t will be used for storing a three-tuple $(f_a(w[t]), P_t, w[t])$, where $f_a(w[t])$ is the off chain computation of the windows $w[t]$ and P_t its proof of computations. This ensures that all intermediate nodes in I_a reached unanimous consensus. Otherwise a dispute will be logged. The next section describes the proof of computation P_t in details. When no-proof storage and off-chain proof quality of services are considered the $w[t]$ data will be empty, for the on-chain proof storage service $w[t]$ is stored on the map. The on-chain contract also defines a reaction r_a. This is a piece of code that can be executed when f_a is evaluated on a new window and all the intermediate nodes reach consensus on the result. For instance reactions can include exchange of cryptocurrency between accounts, auditing, or actuation of IoT devices.

Consensus and security: Our main goal is to ensure that the intermediate nodes securely and correctly perform the off-chain SC execution. This means that the following two properties must be ensured: (P1) all nodes agree on the same computation otherwise a dispute is open; (P2) a node cannot learn and copy the computation of another node. This last property tries to avoid lazy nodes. Correct SC execution is a achieved by a secure computation protocol that, for each window, performs the following three phases: (i) proof of computation; (ii) computation agreement checking; (iii) SC update.

Proof of computation: Each intermediate node i calculates the off-chain code $f(w_i[t])$ by considering its local window $w_i[t]$. Instead of revealing the computation $f(w_i[t])$ the intermediate node calculates the proof of computation $p_{t,i} = H(f(w_i[t])||S_{t,i}||ID_i)||t||a||H(w_i[t])$ where H and $||$ denote a cryptographic hash function and the concatenation operation respectively; $S_{t,i}$ is a fresh random secret that uniquely identifies the t-th computation of f; ID_i is the identity of i; a and t the contract id and the window unique identifier; and $H(w_i[t])$ the hash of the window data. This is used by an intermediate node j to check that its local window has the same hash $H(w_j[t])$. When $H(w_i[t]) \neq H(w_j[t])$ j notifies a dispute on the on-chain contract. The intermediate node i signs and sends its proof of computation to all other nodes for signature. At the end of the proof of computation phase each intermediate node must have the signed proof of computation of all nodes. In the following we denote with $M : i$ the signature of the message M by the node i. Figure 2.A shows a way to implement the the proof of computation protocol. The proof of computation is performed by a node i_1 for an SC that includes three nodes that are i_1, i_2 and i_3. The node i_1 computes the off-chain code $f(w_{i_1}[t])$ and generates the prof of computation $p_{t,i_1} = H(f(w_{i_1}[t])||S_{t,i_1}||ID_{i_1})\ ||a||t||H(w_{i_1}[t])$. This is signed by i_1 and is sent to i_2 for signature. This will append its signature and forwards the message to i_3. This signs the message and sends it back to i_1. At this point i_1 has its proof signed by all nodes. This is broadcast to all other nodes. The same steps will be performed by i_2 and i_3. Each node performs a ring based signature protocol that can be further optimised by using more efficient multi-signature approaches or by electing a leader.

Computation agreement checking and smart contract update: Each intermediate node signs and reveals its secret plus its computation after receiving all proof of computations signed by all nodes. More precisely, a node i reveals its computation by broadcasting the message $c_{t,i} = [f(w_i[t])||S_{t,i}] : i$. This can be used by other intermediate nodes for verifying the off-chain code computation. More precisely, a node j that receives $c_{t,i} = f(w_i[t])||S_{t,i}$ can now verify the hash $H(f(w_i[t])||S_{t,i}||ID_i)$ sent by i and compare its computation with the one of i. When $f(w_j[t]) \neq f(w_i[t])$ a dispute is open by setting a notification on the on-chain contract. A node terminates with a successful on-chain contract update when its computation $f(w_i[t])$ its equal to all the other nodes. In this case the node sends its proof of computation $p_{t,i}$ and $c_{t,i}$ signed by all nodes.Figure 2A shows the message c_{t,i_1} sent by i_1 and the proof of computation sent by all nodes (denoted with P_t). This is register on the on-chain SC.

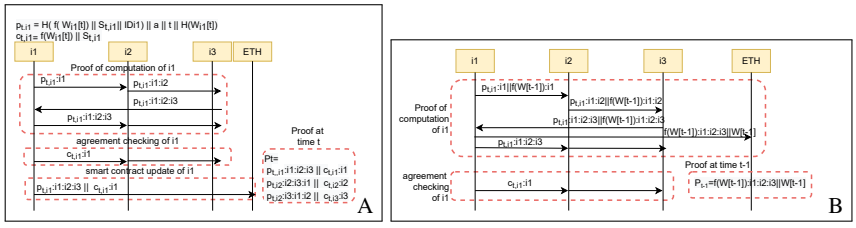

Fig. 2 A) Intermediate node unanimous consensus protocol, B) Fast protocol

We have designed and implemented an optimisation of the protocol (fast protocol). This decreases the amount of messages and reduces the size of the data that are stored in the on-chain SC during the SC update phase. While the proof of computation and agreement checking phases are the same, a leader node is entrusted to produce a compact proof of computation for the previous round of the protocol. More precisely, a leader node i en reaches its proof of computation message $p_{t,i}$ with a message $P_{t-1} = f(w[t-1])$ that contains the computation agreed in the previous run $t-1$. This is sent by the leader node and signed by all the nodes. When the leader gets all the signatures, the on-chain SC is updated. Figure 2B shows an example of the fast protocol. We can see that the leader node i_1 sends together with its proof of work the computation agreed in the previous round. In the example an on-chain proof of storage is implemented since the window data are also stored in the main chain. It is worth mentioning that the leader node can be statically designated during the contract definition or can dynamically change over the time. A dispute must be notified to the users that will try to analyse the window data and try to solve it. When the off-chain proof storage service is used the data will be available in the intermediate nodes. When the on-chain proof of storage is used the data will be available in the main chain.

Implementation: The blockchain stores a SC called *mainContract*. This SC allows Ethereum accounts to register sensors, intermediate nodes, users and on-chain contracts inside the blockchain. Each time a user registers an entity by calling the appropriate method of *mainContract* a new SC representing that entity is created. A reference to this new SC is kept in the main SC.

A sensor is represented by a SC containing the *publicKey* used for signing messages and its id. Intermediate nodes are represented as SC containing an Ethereum address and its ip address.
A *User* is represented as a SC containing an Ethereum address and an id with the relative getters. Only registered users can be referenced by an on-chain SC.
On-chain SC specifies the off-chain code that should be performed by intermediate nodes on the data provided by the specified sensors and the intermediate nodes and sensors interested in the contract. On-chain SC contain the following attributes:

- *intermediateNodes*: it is the list of the intermediate nodes allowed to partake in the execution of the agreement SC;
- *parties*: it is a list containing the users interested in the SC execution;
- sensors: a list of the sensors that provides data used in the execution of the off-chain SC by the intermediate nodes;
- *offChainCode*: it is a JSON string containing the off-chain code that should be executed by the intermediate nodes.
- *reward*: it is the amount of wei that should be rewarded to nodes when they agree on the result of a computation.

The agreement SC contains also methods that allows intermediate nodes to perform reactions (e.g., log data or to move funds from the SC to a *party*) if they are provided with the signature of all the intermediate nodes referenced by the on-chain SC.

4 Performance Evaluation

In this section we tested a simplified version of our proposed model on a smart lock case study. The main objective here is to compare three approaches in terms of fees spent. In the first one, the data produced by a smart lock is directly committed to the blockchain. In the second one, the data produced is first sent to an intermediate node that aggregates them and commits the result of the aggregation to the blockchain. In the third approach, both the result and the data used in the aggregation are sent to the blockchain.

Experimental setup: The experimental setup consists of two virtual machines. One who runs Ubuntu 18.04.4 LTS and four validator nodes of a private Hyperledger Besu network [2]. The blockchain is set up with the IBTF 2.0 proof of authority consensus protocol.
Another Ubuntu 18.04.4 LTS Virtual machine acts as an intermediate node and performs two tasks: (i) complex event processing through Flink [1] and (ii) blockchain interaction. The java application running flink reads data streams on an artificially generated dataset. When a Flink pattern is applied to the stream an aggregation function is executed. The combination of the flink pattern and the aggregation function acts as the off-chain smart contract. The result is then sent as a JSON object to a NodeJS app through a socket. The app prepares a transaction for the blockchain using the received JSON as argument and signs the transaction.

Smart lock case study: This example is a simulation of an hypothetical scenario where a smart lock sends messages related to its opening and closing to an intermediate node that will aggregate the data received into a single transaction containing the amount of time the lock remained open in a month. The intermediate node reads the message data on 10 artificially generated datasets. The first dataset contains data for 10 opening and 10 closing messages in a month [2], each subsequent dataset contain 10 opening and closing messages more than the previous one, up to the last dataset containing 100 opening messages and 100 closing messages. Messages contain two attributes: *state* and *timestamp*. *state* is a boolean and determines if the message refers to the opening of the smart lock (true) or it is a closing message (false). *timestamp* is the value that represents the moment in time when the event occurred. The applications tested simulate a stream of events based on the messages of the datasets and applies to it a Flink pattern that matches whenever the difference between the timestamp of the first element of its trace and the last one is more than one month. A transaction is sent to a logging smart contract on the blockchain whenever the pattern is matched. The content of the transaction depends on the application tested:

- The session application (SA) sends only the value of the session. A session is an integer representing the amount of time a lock remained open in a month. The gas spent executing SA is equivalent to the amount of gas spent when the quality of service is set either to *no-proof of storage* or *off-chain proof of storage*

[2] We denote with *OCM* the amount opening and closing in a month.

- The session application with trace (SAT) stores in the blockchain both the session and all the opening and closing events that triggered the pattern along with the SHA256 with ECDSA signature of the event. This corresponds to the quality of service *on-chain proof of storage*

An additional application called *raw data application* (RDA) has been tested for comparison reasons. The RDA application sends a transaction containing the data in a message each time an opening or closing message is generated.

Results: The results of the simulations are shown in Figure 3 and Table 1. The Y-axis shows the amount of gas spent for storing data relative to one month of activity. The X-axis shows the amount of opening and closing events in the dataset taken into consideration. The amount of gas spent in the SA application is constant and always lower than the gas spent in the other applications. This is due to the fact that only a single transaction containing an integer representing the session is stored in the blockchain no matter how many events are in the pattern trace. The increase in gas spent in the SAT application is linear, this happens because each transaction stores every event matched in the pattern, this means that the higher the amount of events are in the dataset the higher the amount of data to be stored. In the RDA application the amount of gas spent increases linearly because it generates a transaction for each event. The reason why the amount of gas spent in the RDA application is significantly higher than the gas spent in the SAT application is because the constant cost of sending a transaction is higher than the cost of adding more data in a transaction, thus, although the amount of data sent by the two applications to the blockchain is almost the same, the amount of transactions performed are way higher in the RDA application than in the SAT application.

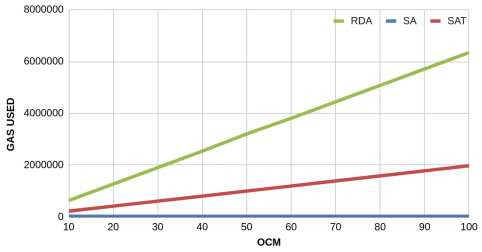

Fig. 3 Gas consumption chart for different approaches

Table 1 Gas consumption table for different approaches

OCM	RDA	SA	SAT
10	634876	23952	222607
20	1269688	23952	416244
30	1904628	23952	610379
40	2539952	23952	803924
50	3206367	23952	998992
60	3809896	23952	1193854
70	4444836	23952	1388638
80	5079584	23952	1584176
90	5714780	23952	1780853
100	6349208	23952	1976492

5 Conclusion and future work

In this paper we present *Diversity*, a second layer decentralised network for off-chain smart contract execution. *Diversity* allows users to define on-chain contracts that

are automatically and securely executed off-chain by intermediate nodes. An honest intermediate node that performs a correct off-chain computation will be enough to detect any dishonest ones and open a dispute. This is achieved by *Diversity* novel protocol. Off-chain smart contract code is not bound the main chain contracting language and can be written in any language. We have validate our approach on a novel lock case study where fees have been greatly reduced.

References

1. Apache flink. https://flink.apache.org/, 2020.
2. Hyperledger besu. https://besu.hyperledger.org/en/stable/, 2020.
3. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 185–200, 2019.
4. R. Dennis, G. Owenson, and B. Aziz. A temporal blockchain: A formal analysis. In *2016 International Conference on Collaboration Technologies and Systems (CTS)*, pages 430–437, 2016.
5. G. Dittmann and J. Jelitto. A blockchain proxy for lightweight iot devices. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 82–85, 2019.
6. J. Eberhardt and J. Heiss. Off-chaining models and approaches to off-chain computations. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL'18, page 7–12, New York, NY, USA, 2018. Association for Computing Machinery.
7. J. Eberhardt and S. Tai. On or off the blockchain? insights on off-chaining computation and data. In F. De Paoli, S. Schulte, and E. Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, pages 3–15, Cham, 2017. Springer International Publishing.
8. J. Ellul and G. J. Pace. Alkylvm: A virtual machine for smart contract blockchain connected internet of things. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–4, 2018.
9. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
10. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.
11. Y. K. Peker, X. Rodriguez, J. Ericsson, S. J. Lee, and A. J. Perez. A cost analysis of internet of things sensor data storage on blockchain via smart contracts. *Electronics*, 9(2):244, Feb. 2020.
12. J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.
13. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.
14. M. Salimitari and M. Chatterjee. A survey on consensus protocols in blockchain for iot networks, arxiv, 1809.05613, 2019.
15. A. R. Shahid, N. Pissinou, C. Staier, and R. Kwan. Sensor-chain: A lightweight scalable blockchain framework for internet of things. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1154–1161, 2019.
16. V. Trón, A. Fischer, D. A. Nagy, Z. Felföldi, and N. Johnson. Swap, swear and swindle - incentive system for swarm. 2016. https://ethersphere.github.io/swarm-home/ethersphere/orange-papers/1/sw%5E3.pdf.
17. G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.