The cost of immortality: A Time To Live for smart contracts

Dimitri Saingre, Thomas Ledoux, Jean-Marc Menaud

To cite this version:

Dimitri Saingre, Thomas Ledoux, Jean-Marc Menaud. The cost of immortality: A Time To Live for smart contracts. ISCC 2021 - 26th IEEE Symposium on Computers and Communications, Sep 2021, Athènes, Greece. 10.1109/ISCC53001.2021.9631513 . hal-03300144

HAL Id: hal-03300144
https://inria.hal.science/hal-03300144

Submitted on 26 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The cost of immortality: A Time To Live for smart contracts

Dimitri Saingre
IMT Atlantique - Inria - LS2N
Nantes, France
dimitri.saingre@imt-atlantique.fr

Thomas Ledoux
IMT Atlantique - Inria - LS2N
Nantes, France
thomas.ledoux@imt-atlantique.fr

Jean-Marc Menaud
IMT Atlantique - Inria - LS2N
Nantes, France
jean-marc.menaud@imt-atlantique.fr

***Abstract*—Smart contracts, scripts at the heart of blockchain-based applications, are meant to be available forever once deployed. However, this property has a price. The amount of space required to store new contracts keeps increasing. This increase impacts each participating node's performance and makes it inconvenient for low-end devices to participate in the network. Among all contracts deployed in the blockchain, a vast majority will lead to little if any usage. We demonstrate that, in the course of one year, 70% of deployed contracts lead to no use. Unfortunately, unused contracts keep occupying space on the blockchain. To tackle this issue, we propose a new protocol to identify and delete unused contracts. Through simulation, based on Ethereum historical data, we show that deletion of smart contracts after an inactivity period of 90 days could lead to a 66% reduction in the number of contracts stored over a year.**

***Index Terms*—Blockchain, smart-contracts, Ethereum, performances, Time-to-live**

I. Introduction

The concept of blockchain emerged in 2008 with the development of Bitcoin [1]. Bitcoin became the first large-scale currency that did not rely on any central authority. Twelve years later, Bitcoin has a market capitalization of 809 billion dollars[1]. Seeing the potential of blockchain and the limitation of a cryptocurrency-only technology, Ethereum [2] introduced the concept of *smart contracts*. Those scripts, deployed and executed on the blockchain, enable the development of blockchain-based decentralized applications (Dapps). Since then, researchers and companies have proposed a wide range of use cases for Dapps like sharing of computing resources [3], [4], decentralized social networks [5], government services [6], storage solutions [7] and energy trading [8].

Smart contracts are at the heart of the Ethereum community. Tens of new thousands of new contracts are deployed each month. However, the majority of those deployed contracts will rarely, if ever, be used (see section III). Unfortunately, even with no usage, every contract have a continuous cost on the blockchain. Every contract deployment makes the Ethereum State (the data structure storing every Ethereum account) bigger. As this state is read and updated for every new transactions, its growth has an impact on block processing time (see subsection IV-B). Therefore, it is necessary to specify protocols to limit the growth of Ethereum State.

In this paper, we challenge the "immortality" aspect of smart-contracts by lifespan mechanism to every deployed contracts. Through a new protocol (presented in subsection V-A) applied to smart-contracts, we incentivize miners to identify and remove unused contracts. This will ensure that only active contracts will be stored in Ethereum State. Through simulation, based on one year of Ethereum transactions data (see subsection V-F), we illustrate potential significant reduction in the number of contracts stored in the blockchain. For instance, removing every contract after an inactivity period of 90 days could lead to a 66% reduction of deployed contracts at the end of our one-year dataset.

To sum up, we present in this paper the following contributions:

- A Time-To-Tive (TTL) protocol for smart-contracts. After presenting this new protocol, we discuss the impact of different protocol parameters and security considerations. Eventually, we evaluate the potential impact of our proposition on the number of active contracts on Ethereum, based on one year of real-world transactions.
- Motivate research on managing the impact of unused smart contracts by giving insight on the number of unused smart contracts and their impact on the network. This analysis is based on transactions emitted on the Ethereum network over one year.

II. Background

A. Smart contracts

Smart contracts are scripts deployed on the blockchain and called through transactions. They enable the development of decentralized applications. Hoffman et al. [9] discuss the concept of *decentralized applications* and its differences with *distributed computing*. They define distributed blockchain platforms as "*multi-stakeholder Web ecosystems acting as sophisticated support networks enabling peer-to-peer transfers of value and information, including goods and services, in a coordinated manner*".

Listing 1 illustrates an example of a simple smart contract written in Solidity, a popular programming language used to

[1]According to https://coinmarketcap.com/en/currencies/bitcoin/ on 19th February 2020.

978-1-6654-2744-9/21/$31.00 ©2021 IEEE

developed smart contracts for Ethereum. In this example, we defined a simple contract that manage a counter. In Ethereum, a contract's constructor (here used to initialise the counter) is called only once, during deployment. Contracts can then define several functions that can be called through transaction emitted on the network. Here, an *increment* function can increase the counter value. This counter value, stored as a variable, is saved across all transaction calls. Indeed, smart-contracts can save and modify data in a dedicated storage space in the blockchain, through variables. Smart-contracts are compiled and executed in the *Ethereum Virtual Machine* (EVM). The EVM is responsible for all transaction executions.

B. *Ethereum's data storage*

In Ethereum [10], all blockchain data is stored in several modified Merkle Patricia Trees (called *tries*). The *State* trie stores all information related to accounts (including smart-contracts). Each account in the State trie is a collection of information related to a given 20-Bytes hexadecimal address, like its balance (number of Wei[2] possessed by the account) and (for smart-contracts) its *EVM* byte-code.

Each smart-contract can store data in a *Storage* Trie. This Storage Trie is used to store contract variables that will persist across function calls. Contract's account, stored in the state trie, contains the 256-bit hash of its Storage root. The information in the state and storage tries are updated through transaction executions. Figure 1 illustrates the relationship between a given block, the state trie, and the storage trie.

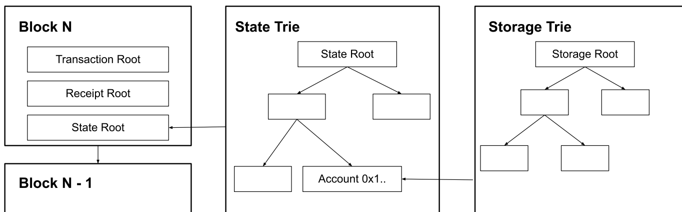

Fig. 1. Ethereum's data storage

Ethereum serialises its different Tries using a *Recursive Length Prefix* (RLP) encoding scheme (see Ethereum *yellow paper* [10]). Serialised data are then stored in a Key-Value database. For instance, the *Geth* Ethereum implementation uses LevelDB [11] to store its data.

C. *Selfdestruct function on Ethereum*

Smart contracts are initially designed to be stored on the blockchain forever. However, in Ethereum, contracts can implement a call to *selfdestruct(address)* function that, once called, will destroy the contract and transfer funds in its balance to the provided address. Contract destruction removes its account (address, balance, bytecode) from the state trie and all stored variables from the storage trie. The contract can't be called anymore, and its address can be reassigned to a new contract. However, all transactions related to the destructed contract still exist in the blockchain transaction history.

[2]The Smallest denomination of Ether. 1 Ether = 10^{18} Wei

Chen et al. [12] investigated the use of *selfdestruct* in Ethereum smart-contracts. They found out that only 5.1% out of 54,739 analysed contracts contain a *selfdestruct* function. Through surveys addressed to smart-contract developers, they defined six reasons for why most contracts does not include *selfdestruct*. Those reasons include *security* and *trust* issues. Indeed, an incorrect contract implementation could lead to an accidental or *malicious* use of *selfdestruct*. Chen et al. also define five reasons for the inclusion of *selfdestruct* (e.g. to destroy a contract when security vulnerabilities are detected). However, numbers show that those reasons are not enough to incentive developers to implement *selfdestruct* in their contracts.

III. Measuring the number of unused contracts on Ethereum

A. *Extracting Ethereum data - Experimentation protocol*

As stated in section II, Ethereum is a public blockchain. Everyone can become a participant in the network and gain access to all data stored in the blockchain. To query these data (blocks, transactions, accounts...) conveniently, we extracted them in a relational database using *Ethereum-ETL* [13]. Ethereum-ETL is an open-source ETL (*Extract-Transform-Load*) developed to extract data from one Ethereum node (through its JSON-RPC API) to different outputs (including Google Cloud, CSV files or PostgreSQL). In our case, we stored the extracted data in an open-source Database Management System (DMS) called PostgreSQL. The data extraction was split into two phases.

First, we installed an Ethereum Node on a Dell PowerEdge R630 server (Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 2 CPUs/node, 8 cores/CPU), 128 GiB of RAM, 600GB HDD SATA Seagate ST600MM0006 storage), part of the Grid5000 [14] research testbed. We chose one of the main open-source Ethereum implementations: OpenEthereum (v3.0.1-stable) [15]. We fully synchronized this node with the rest of the Ethereum Mainnet (The main/"production" Ethereum network). Processing took several days and resulted in the downloading of more than 10 million blocks. The second step was to extract and analyze recent data from this synchronized node, with Ethereum-ETL. We extracted a year of data from September 2019 to August 2020. The resulting PostgreSQL database represented about 300GB of data.

B. *New smart contract deployment*

Figure 2 illustrates the number of smart contracts deployed over the year. In Ethereum, the deployment of a new smart contract is made by emitting a transaction embedding the contract byte-code and with no recipient. A smart contract deployment cost is roughly similar to the cost of an Ether-transfer transaction. The receipt of this transaction will contain the new smart contract address. By checking the number of transactions used to deploy smart contracts, we can see that, on average, 73 000 new contracts have been deployed each month. Over the year, the number of new contract deployment tend to follow the same evolution as the number of transactions.

Smart contract deployment represents around 0.3% of the total of transactions processed each month. Although the number of smart-contracts created each month may seem impressive, we must put this number into perspective with the actual usage of these new contracts.

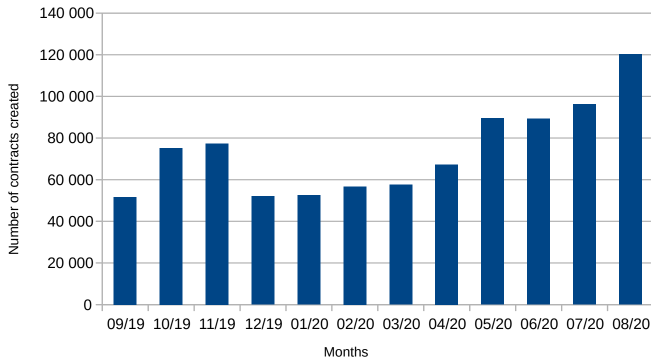

Fig. 2. Number of contracts created over time

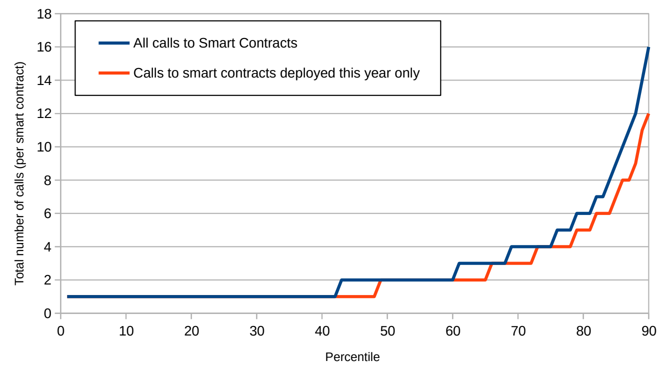

Fig. 3. Distribution of the number of calls per smart contracts - Lowest 90 percentiles

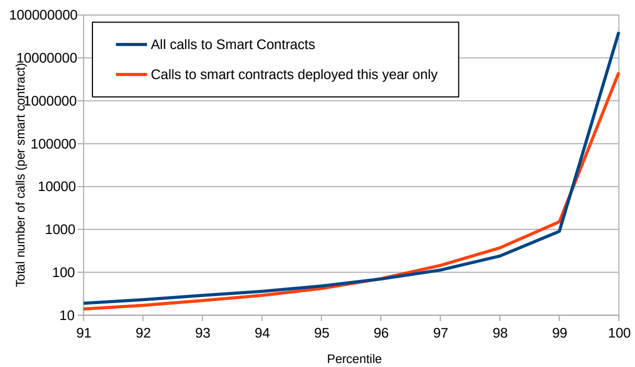

Fig. 4. Distribution of the number of calls per smart contracts - Highest 10 percentiles - Logarithmic scale

C. Quantifying the number of unused smart-contracts

Figure 3 and Figure 4 illustrate the distribution of the number of calls per "active" smart contracts[3], through percentiles (data has been divided into two figures for readability). Percentiles are used to illustrate frequency distribution. Among contracts that have been deployed this year, we can see that 48% have been called only once[4] and 88% have been called less than ten times. Only a few contracts are responsible for most of the traffic. Indeed, 3236 contracts (1.2% of a total of 259 472 "active" contracts deployed this year) are the recipients of 95% contracts calls. If we do not limit the analysis to contracts deployed during the year and consider calls to every smart contract, the trends remain the same. 2851 contracts (0.7% of a total of 412 362 "active" contracts) are the recipients of 95% of all the smart contract calls. Among those 2851 contracts, 1300 (46%) are ERC20 contracts (contracts that implement a standard called "ERC20" that defines a set of functions used to implement new Tokens). The most called contract, Tether[5] (an ERC20 cryptocurrency), accumulates 29% of smart contracts calls. The second most used contract is responsible for 2% of smart contract calls.

As we have seen, only a small minority of contracts are responsible for the traffic of Ethereum smart contracts. The majority of smart contracts will also stop being used only a few days after their deployment. Figure 5 illustrates the evolution of the proportion of active contracts depending on the number of days since their deployment. We only consider contracts deployed in the first eight months of our dataset, and consider each contract's first four months of "existence". We can see that 50% of considered smart contracts won't be called a week after their deployment. Only 16.6% of deployed smart contracts will be called after four months.

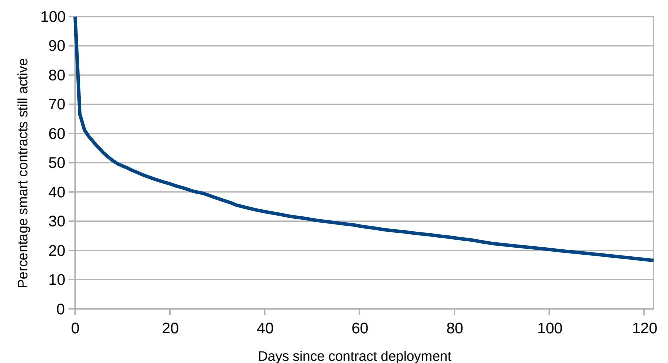

Fig. 5. Evolution of the percentage of active contracts depending on the number of days since their creation (Contracts deployed this year only)

Among all contracts deployed this year, only 259 472 (30%) of a total of 866 508 have been called at least once since their deployment. 607 036 (70%) contracts have been deployed without ever been called. While unused smart contracts induce a one-time (financial) cost for their developers (deployment

[3]We define "active" contracts as contracts that have been called at least once.

[4]This value corresponds to the 48th percentile in Figure 3

[5]https://tether.to/

fees), they cause an ongoing cost (storage and computation) on the chain and, therefore, for all participating nodes. This overhead could be limited by the identification and suppression of those unused smart contracts.

IV. The impact of unused smart-contract on Ethereum

A. Evaluation protocol

After establishing that a large part of deployed smart-contract will rarely (if ever) be used, we aim to evaluate the impact of those unused contracts on Ethereum block processing time (a key point in the overall network performances). To measure this impact, we deployed a minimal two nodes Ethereum network (using Geth v1.10.3, the primary open-source Ethereum implementation) on three testbeds:

- A two-nodes network hosted on a Grid5000 [14] server equipped with 2 x Intel Xeon E5-2630 v3 CPU (8 cores/CPU), 128 GB of RAM and a 600 GB HDD, here called *G5K HDD*;
- A two-nodes network hosted on the same Grid5000 network but with a 200 GB SSD, here called *G5K SSD*;
- A two-nodes network hosted on a Raspberry Pi 4B equipped with a Quad-core Cortex-A72 (ARM v8) CPU, 8 GB of RAM, here called *RPi*.

Our experiment consists of measuring block processing time depending on the state trie size. We define four collected metrics: 1) *execution time* (time spent to modify current state with transactions in the received block) 2) *validation time* (time spent to verify if the newly produced state is valid) 3) *write time* (time spent to write the new state on disk) 4) *insert time* (the overall time spent to insert the received block, includes the three others). We collected those metrics through the standard metrics collection of *Geth* (the *Geth* Ethereum client emits performance metrics to a third-party database). In order to isolate different kind of transactions, we define three loads: 1) *transactions load* (generation of random Ether transfer transactions) 2) *contract creations load* (generation of contract deployment transactions) 3) *contract calls load* (generation of contract calls). The contract used for the two last load generations is a basic implementation of a basic Counter manager contract (illustrated in Listing 1).

```
contract Counter {
    uint256 public counter;
    constructor() {
        counter = 0;
    }
    function increment() public returns(uint256){
        counter = counter + 1;
        return counter;
    }}
```

Listing 1. A simple Counter contract for analysis purpose

B. The impact of unused smart-contract on contract calls processing time

Table I illustrates the impact of State size in Ethereum block processing time. In nearly all cases, an increase in Ethereum State size leads to a rise in block processing time (this increase is noted $+X.XX\%$ in the table). This increase is more significant for the low-end Raspberry Pi platform (because of their lower performances). Among all four collected metrics, an increase in Ethereum state size seems to impact block writing time the most. Out of the three loads, the Ether transfer appears to be the "cheapest" one for the "high performances" Grid5000 nodes. An Ether transfer only updates two balances (creating a new account in the State Trie if the recipient is an unknown address). On the contrary, contract deployments require more time to execute than a simple Ether transfer, as it needs to create a new account in the State Trie, copy the contract's bytecode and execute the constructor function.

C. Current Ethereum state size

When writing this paper (Late May 2021), contract codes occupy 1.95 GiB of space and State trie occupies 79.50 GiB on an Ethereum node synchronized using Geth v1.10.3. Apart from impacting block processing time, state growth limits the use of low-end devices as participating nodes. Indeed, to process transactions, a node needs to store and process at least the state and recent blocs. As the State grows because of new account creations, it becomes less convenient for low-end nodes to host an Ethereum node.

V. A Time to live mechanism for smart contracts

Due to a smart contract deployment's relatively low financial cost[6], tens of thousands of contracts are deployed each month (see Figure 2). Among all those contracts, only a small portion will generate significant and long-term use. All the remaining unused contracts, still stored in the state forever, participate in increasing the hardware performances required to host an Ethereum node. This section proposes a potential solution to limit the footprint caused by unused smart contracts by restricting their lifetime (otherwise unlimited).

A. Overview

In order to limit the amount of deployed, yet unused, smart contracts on Ethereum, we propose to add a notion of *lifetime* to every contract. Once deployed, a smart contracts can exist for a given time. This default lifetime, here called *Time To Live* is fixed by the blockchain community. After this default time has passed, any miner can request the contract destruction and be rewarded for its action. A contract TTL can be extended in two ways: through interaction (function call by transactions) or through payment.

The overall goal behind this protocol is to add a continuous financial cost for a smart contract's users that would reflect its hosting cost reality. This cost is manifested through direct payment to maintain potentially inactive contracts or through transactions fees for active contracts. Unused contracts with no users willing to pay for its maintenance would be deleted in order to remove the overhead generated otherwise.

[6]We estimate the average cost of contract deployment in our one-year dataset to be around 38 euros with an average gas cost of 520 249 and a gas price of 32gwei (average gas cost from ethgasstation.info)

		Ether transfer		Contract deployment		Contract call	
Platform	Metric	0GB	10GB	0GB	10GB	0GB	10GB
G5K HDD	Execution	4.22 (0.001)	4.41 (0.001) (+4.5%)	6.19 (0.002)	6.14 (0.002) (-0.8%)	7.17 (0.0008)	7.45 (0.001) (+3.91%)
	Validation	1.24 (0.0003)	1.31 (0.001) (+5.65%)	0.95 (0.0008)	0.96 (0.0002) (+1.05%)	1.24 (0.0004)	1.29 (0.0001) (+4.03%)
	Write	1.71 (0.002)	2.14 (0.0005) (+25.15%)	8.67 (0.018)	13.05 (0.006) (+50.52%)	1.88 (0.0008)	2.25 (0.003) (+19.68%)
	Insert	7.38 (0.002)	9.00 (0.004) (+21.95%)	21.19 (0.028)	28.26 (0.009) (+33.36%)	10.55 (0.001)	11.43 (0.005) (+8.34%)
G5K SSD	Execution	4.13 (0.001)	4.44 (0.002) (+7.51%)	6.05 (0.004)	6.46 (0.003) (+6.78%)	6.51 (0.005)	7.18 (0.003) (+10.29%)
	Validation	1.19 (0.0004)	1.32 (0.0009) (+10.92%)	0.92 (0.0001)	0.97 (0.0002) (+5.43%)	1.11 (0.0006)	1.26 (0.0008) (+13.51%)
	Write	1.87 (0.002)	2.41 (0.002) (+28.88%)	8.98 (0.009)	13.37 (0.005) (+48.89%)	1.69 (0.001)	2.27 (0.003) (+34.32%)
	Insert	7.43 (0.004)	8.56 (0.003) (+15,21%)	21.66 (0.019)	29.15 (0.005) (+34.58%)	9.59 (0.007)	11.08 (0.003) (+15,54%)
RPi	Execution	13.11 (0.008)	13.09 (0.012) (-0.15%)	11.00 (0.039)	12.22 (0.005) (+11.09%)	12.74 (0.004)	14.54 (0.004) (+14.13%)
	Validation	1.14 (0.0006)	1.59 (0.0002) (+39,47%)	0.89 (0.0006)	0.94 (0.0002) (+5.62%)	1.04 (0.0003)	1.12 (0.0004) (+7.69%)
	Write	1.91 (0.0003)	14,21 (0.002) (+643,98%)	8.41 (0.023)	23.12 (0.014) (+174.91%)	1.82 (0.0009)	5.08 (0.003) (+179.12%)
	Insert	16.89 (0.008)	49.43 (0.014) (+192,66%)	27.53 (0.081)	49.33 (0.009) (+79.19%)	16.37 (0.0004)	21.95 (0.001) (+34.09%)

TABLE I

BLOCK PROCESSING TIME (IN MILLISECONDS) COMPARED TO STATE TRIE SIZE ON THREE PLATFORMS (STANDARD DEVIATION BETWEEN PARENTHESIS)

B. Details

We introduce the notations used in this section in Table II

Notation	Details
TTL	Time after which a contract's next rent can be redeemed.
TE	Time Extension - Duration in seconds for which a contract TTL is extended at *interaction* or *extension payment*.
P	Price to pay to extend TTL by T.
R	Reward gave to miner for unused contract destruction.

TABLE II

DETAILS OF VARIABLES USED IN OUR PROTOCOL

At deployment time, a smart-contract has an initial expiration date of $TTL = TE$. Without any extension, miners will be able to request its destruction at time TTL and be financially rewarded by R tokens. Those R tokens come from a fixed deposit made during contract deployment by the account requesting the deployment. This new *destruction deposit* will serve as an incentive for miners to purge unused contracts.

If any user interact with this contract by sending a transaction to its address, the contract's TTL will be extended if its remaining lifetime is shorter than TE (see algorithm 1). In other words, a contract is considered active TE seconds after its last call.

Algorithm 1: Function called for every interaction with a contract

```
Function ExtendTTL(addr):
    if CurrentDate >= TTL(addr) − TE then
        TTL(addr) = CurrentDate + TE
    end
```

To ensure that a contract will still be deployed on a certain date, regardless of its activity, any participant can pay in advance for its hosting fees (see algorithm 2). By paying $N \times P$ tokens, the participant will increase the contract's TTL by $N \times TE$. Some contracts may not be designed for intensive usage. For instance, a contract specifying ownership of a particular asset (e.g. housing) may still be valuable even without new transactions. In this case, the contract community could pay in advance using this functionality to maintain the contract active despite no usage.

Algorithm 2: Function called to increment a contract TTL in advance

```
Function PayForTTL(addr, tokens):
    TTL(addr) = TTL(addr) + TE × tokens/P
```

Once a contract's expiration date has been reached, it becomes eligible for destruction. Any miner will be able to emit a *call for destruction* transaction. This new kind of transaction will destroy a given contract if and only if it's TTL is greater or equal to the current date. As for blocs' timestamp, we allow here a certain lag between current date and TTL as blockchain networks does not include any central clock. A *call for destruction* produces two effect: 1) destroy the specified contract and 2) financially reward the caller. This reward of R tokens correspond to the *destruction deposit* made during the contract's deployment. In theory, anyone can send a *call for destruction* but in practice we consider that, as miners have power on which transaction they include in new blocs, miners have a financial incentive to manage contract destruction themselves.

In order not to flood the network with *call for destruction* transactions, we define its input as a list of contracts to be deleted. This way, a miner can request the destruction of unused contracts through batches. For a list of contracts to be destroyed, only contracts available for destruction will actually be destroyed (others will be ignored). The reward granted to the miner will correspond to the sum of each individual contract reward.

C. Contract destruction and data retrieval

Through a *call for destruction* transaction, a miner can destroy an unused contract. In Ethereum, destructing a contract is equivalent to removing its account (address, balance, bytecode) and related storage from the state trie. However, a contract destruction does not impact a blockchain transaction history. Transactions related to a contract could still be retrieved and replayed after its destruction for later use (e.g. for later analysis). The impact of our proposed protocol lie in State trie management.

D. Discussion on determining parameters value

As we descried our protocol for smart-contracts with Time To Live, we have presented several protocol parameters (detailed in Table II). We did not specify any fixed values, as we wanted to describe a general protocol that could be adapted to any blockchain network. We will now discuss on some considerations regarding the choice of specific values.

1) Setting TE: TE represent the unit of time used for contract lifetime and its payment. A value too high would reduce the gains on state management, taking longer to purge unused contracts. On the other hand, a value too low could be too intrusive and lead to the destruction of a contract that could be still used. We detail this trade-off in subsection V-F.

2) Setting P: We believe that paying in advance should be roughly equal to basic transaction fees. If set too high, this could create an incentive for the creation of undesired "uptime services" that could maintain contracts through interactions for cheaper ($< P$) fees.

3) Setting R: A higher *destruction deposit* serves as a better incentive for miners to purge unused contracts, but makes it more expensive for developers to deploy new contracts. However, too high a value could be an incentive for undesired behaviour (see subsection V-E).

E. Discussion on potential attacks

With this proposition, we enable the destruction of deployed (unused) contracts by miners once their TTL has been outdated. The main potential attack we conceive would be the censorship of some contract-related transactions by miners in order to reach a contract's TTL, destroy this contract and redeem the destruction reward. We believe that the plausibility of such an attack lies in the Byzantine fault tolerance of the blockchain consensus algorithm. Indeed, as long as enough miners stay honest, contract-related transaction will be still processed. Moreover, this potentially undesired behaviour could be limited by avoiding setting the destruction reward R too high.

F. Protocol impact depending on TTL duration

As stated in subsection V-D, TE is the principal protocol parameter that will determine the duration of an inactivity period before determining that a contract is eligible for destruction.

Figure 6 illustrates the evolution of the number of deployed contracts during our one-year dataset and potential gains depending on TE value. In this evaluation, we consider that a contract will be destructed right after an inactivity period of TE days. We do not consider any payment in advance in order to extend contract lifespan. As expected, a small TE (e.g. 10 days) leads to significant reduction in the number of deployed contracts (90% reduction, from 945 116 contracts at the end of the year to 85 485). A larger value of TE still brings significant impact (77% reduction for $TE = 60$ days) while limiting overhead for contract users.

Concerning the potential overhead of this proposed protocol, Table III illustrates the number of contracts called after an

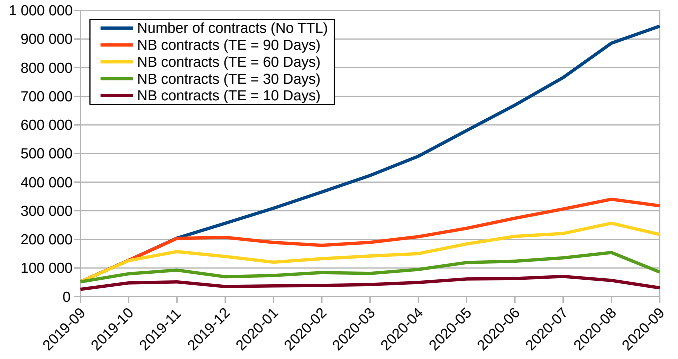

Fig. 6. Evolution of the number of contracts depending on TE value

inactivity period of TE days. We see that, in this one year dataset, only 7.1% (66 709) of all 945 116 contracts have been used after a 10 days inactivity period. This percentage drops to 1.1% (10 236) for a 90 days inactivity period. Among all contracts that have been called after a certain period of inactivity, only a small minority have lead to significant usage. For instance, 90% of contracts used after an inactivity period of 10 days have been called at most 10 times after this inactivity period. Users of such contracts that could be used after an inactivity period greater than TE could pay in advance to ensure that their contract will be still deployed for later use.

VI. Related work

To the best of our knowledge, very few academic papers have addressed the issue of removing data on blockchains. Dennis et al. [16] proposed a formal analysis of a temporal "rolling" blockchain, in order to maintain a stable blockchain size. In their proposed model, transactions are stored for a pre-set period (older transactions are removed). In their paper, Dennis et al. considered a Bitcoin-like blockchain. Bitcoin differs from Ethereum at least in two points: 1) it does not have smart-contracts, 2) it uses a different model (called *UTXO*) for transactions. With UTXO, coins have a transaction history: the balance of a given account is the sum of coins addressed to this account minus coins emitted from this account. Ethereum uses a different approach by updating different data structures (as detailed in section II) through each transaction. The use of those different structures makes it more difficult to determine which data to delete with a "rolling" blockchain. Our proposed protocol focuses on state trie and smart-contract management. We consider both approaches to be complementary: The approach proposed by Dennis et al. could be used to manage transaction history, whereas ours could be used to manage smart-contracts accounts.

We also find a few exchanges on related topics outside academia. In an article[7], Vitalik Buterin (co-founder of Ethereum) discuss the issue of state management in Ethereum. He discusses several potential strategies for state expiry. Our

[7]We decided to include the following reference for exhaustiveness and as its content helped us in our thoughts, but we remind readers that this does not constitute any peer-reviewed content.

TE (in Days)	10	30	60	90
Number contract called after inactivity period of TE days	66 709	36 731	18 226	10 236
% Total (945 116)	7.1	3.9	1.9	1.1
Number of calls after inactivity period of TE days (max)	358	274	254	218
Number of calls after inactivity period of TE days (average)	6	4	3.5	3
Number of calls after inactivity period of TE days (median)	2	2	1	1
Number of calls after inactivity period of TE days (90th percentile)	10	7	6	5

TABLE III

NUMBER OF CONTRACTS CALLED AFTER AN INACTIVITY PERIOD

approach can be seen as a mix between approaches here called *rent via time-to-live* (where users can pay to extend the lifetime of a state object) and *refresh by touching* (where a state object's lifetime is extended through interactions). Unfortunately, the article does not include any evaluation of any discussed approaches.

VII. FUTURE WORK

Through this paper, we hope to challenge some aspects of current blockchain systems. Complete immutability of stored data comes with the cost of an ever-growing database. We believe our proposed protocol is an interesting trade-off to lower the footprint of blockchain systems. We identify room for improvement in order to develop a new framework for low-footprint blockchains:

- Investigate the possibility to dynamically determine the TTL extension value TE, based on account creation rate in order to limit contract suppression if state size stabilises;
- Investigate how this protocol could be adapted to *Externally Owned Accounts* (accounts that does not belong to smart-contracts). It appears hazardous to delete accounts that could belong to human users, as they could be using their account for long-time investment;
- Further investigate theoretical values that protocol parameters TE, P, R can take, their implication on one another and their implications on the network behavior.

VIII. CONCLUSION

This paper tackles the issue of the ever-growing space required by unused smart-contracts. As we saw in section III, a large majority of deployed smart-contracts will lead to limited usage. Over the course of one year, we determined that 70% of deployed contracts were never called. Half of the remaining 30% won't be called a week after their deployment. This ever-growing amount of unused contracts is not without consequences for the Ethereum community. Indeed, each contract deployment participate in the increase of Ethereum state size. This increase impacts the performances of the network and makes impractical the use of low-end devices as participating nodes.

In order to identify and remove unused smart contracts, we propose a novel "Time-To-Live" protocol for smart contracts. This protocol incentives miners to identify and remove unused contracts from the blockchain. We simulated that, over the course of one year, removing contracts after an inactivity period of 90 days could lead to the reduction of the number of contracts in Ethereum state by 66%.

ACKNOWLEDGEMENTS

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

This paper has been financed by the *HYDDA FSN* project.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *Journal for General Philosophy of Science*, vol. 39, no. 1, pp. 53–67, 2008.

[2] E. Foundation, "A Next-Generation Smart Contract and Decentralized Application Platform," https://github.com/ethereum/wiki/wiki/White-Paper, 2013, whitepaper.

[3] J. Zawistowski, P. Janiuk, A. Regulski, A. Skrzypczak, A. Leverington, P. Bylica, M. Franciszkiewicz, P. Peregud, A. Banasiak, M. Stasiewicz, R. Zagórowicz, and W. Davis, "The Golem Project," https://golem.network/crowdfunding/Golemwhitepaper.pdf, 2016, whitepaper.

[4] Iexec, "IExec: Blockchain-Based Decentralized Cloud Computing ," https://iex.ec/wp-content/uploads/pdf/iExec-WPv3.0-English.pdf, 2018, whitepaper.

[5] Steem, "Steem - An incentivized, blockchain-based, public content platform," https://steem.com/steem-whitepaper.pdf, 2016, whitepaper.

[6] "E-Estonia," https://e-estonia.com/.

[7] P. Labs, "Filecoin: A Decentralized Storage Network," https://filecoin.io/filecoin.pdf, 2017, whitepaper.

[8] E. Mengelkamp, J. Gärttner, K. Rock, S. Kessler, L. Orsini, and C. Weinhardt, "Designing microgrid energy markets: A case study: The brooklyn microgrid," *Applied Energy*, vol. 210, pp. 870–880, 2018.

[9] M. R. Hoffman, L.-D. Ibáñez, and E. Simperl, "Toward a formal scholarly understanding of blockchain-mediated decentralization: A systematic review and a framework," *Frontiers in Blockchain*, vol. 3, p. 35, 2020.

[10] W. Gavin, "Ethereum: a secure decentralised generalised transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf, 2020, yellowpaper.

[11] S. Ghemawat and J. Dean, "Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values." https://github.com/google/leveldb, 2021.

[12] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *arXiv preprint arXiv:2005.07908*, 2020.

[13] M. Evgeny, "Ethereum-etl," https://github.com/blockchain-etl/ethereum-etl, 2020.

[14] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[15] Openethereum, "Openethereum," https://github.com/openethereum/openethereum, 2020.

[16] R. Dennis, G. Owenson, and B. Aziz, "A temporal blockchain: a formal analysis," in *2016 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2016, pp. 430–437.