SmartFly: Fork-Free Super-Light Ethereum Classic Clients for Internet of Things

Pericle Perazzo and Riccardo Xefraj

***Abstract*—The use of blockchains in the Internet of Things (IoT) is extremely promising, as it gives connected things the possibility to send and receive payments or tamper-proof data. In the last years, FlyClient has emerged in the literature as a technique for allowing resource-constrained devices to verify blockchain transactions. FlyClient is based on Merkle mountain ranges (MMRs) and probabilistic sampling, and it allows us to develop blockchain clients whose resource consumption is sublinear with the length of the chain. However, this comes at the cost of a change in the blockchain format, which leads to forks that are politically expensive, because they require 51% consensus. In this article, we explore the possibility of fork-free FlyClient verification methods that leverage smart contract programming. Smart contracts are able to add functionalities to a blockchain without needing forks. This raises several and novel technical issues that we address in the article. We show that fork-free sublinear clients are feasible without trusting the nodes that invoke the smart contract methods, as long as the smart contract language provides a means to access the most recent block or its hash. As a proof of concept we propose SmartFly, a fork-free FlyClient verification system for the ethereum classic (ETC) blockchain. We measure several performance metrics of SmartFly, proving that it is succinct in storage and bandwidth consumption and economically bearable (about 38 euros per day to maintain the whole system).**

***Index Terms*—Blockchain, ethereum classic (ETC), FlyClient, Internet of Things (IoT), Merkle mountain range (MMR), smart contracts.**

I. Introduction

IT IS widely believed that the application of blockchain technology in the Internet of Things (IoT) is not a matter of "if" but "when" [1], [2], [3], [4], [5], [6], [7]. This is because the blockchain technology has several unique features that can address some of the key challenges in IoT, such as security, trust, data sharing, and decentralization. The importance of blockchain-based IoT applications is witnessed by recent standards or draft standards proposed by IEEE [8] and IETF [9]. The possible showstopper is that, to validate data, money transfers, or smart contract executions on the blockchain, clients need to download a huge amount of data, whose size grows linearly with the length of the blockchain itself. Such an operation is extremely burdensome for resource-constrained devices, since they have either to download and verify tens of gigabytes, or rely on some trusted full node to do it for them, which may be unavailable in some applications. This limits the possibility of applying blockchain technologies in the IoT. Bünz et al. [10] proposed FlyClient, an extremely lightweight verification method that uses Merkle mountain range (MMR) data structures and probabilistic sampling on proof-of-work (PoW)-based blockchains. As an example, in ethereum classic (ETC) [11] such a technique would allow a resource-constrained client to verify a money transfer by downloading just 500 kB [10], which is 6600 times smaller than what state-of-the-art clients can do. This is made possible by a probabilistic proving protocol whose size grows only logarithmically with the chain length.

An important limit of FlyClient is that it needs to refactor the block headers, and therefore it necessitates at least a soft fork, which is politically expensive and risky, because it needs 51% consensus to be successful. Among the major cryptocurrencies, only Zcash [12] tackled a similar fork, introducing FlyClient support with the "Heartwood" fork in 2020.[1] Another possibility is to use a velvet fork [13], which does not require consensus majority. However, introducing FlyClient support with a velvet fork can make the system vulnerable to chain-sewing attacks [14].

In this article we explore the possibility to realize FlyClient super-lightweight clients *without any fork*, by means of the smart contract technology. Such a technology is capable of augmenting the features of a blockchain network without needing forks, by programming class-like objects called *smart contracts*. This allows us to realize and maintain an MMR data structure with a smart contract, at the cost of some cryptomoney to pay in order to deploy it and invoke its methods. We show that this approach is generally feasible without trusting the nodes that invoke the smart contract methods, provided that the smart contract programming language allows access to the most recent block or its hash. We further present SmartFly, a system based on smart contracts that realizes FlyClient on ETC without forks. SmartFly takes into consideration the difficulty adjustment mechanism and possible difficulty tampering attacks [15]. We measure the monetary cost of maintaining SmartFly under various tradeoffs, and its performance in terms of client storage and client bandwidth

Manuscript received 15 May 2023; revised 16 August 2023, 15 October 2023, and 8 December 2023; accepted 20 December 2023. Date of publication 10 January 2024; date of current version 25 April 2024. *(Pericle Perazzo and Riccardo Xefraj contributed equally to this work.) (Corresponding author: Pericle Perazzo.)*

Pericle Perazzo is with the Department of Information Engineering, University of Pisa, 56122 Pisa, Italy (e-mail: pericle.perazzo@unipi.it).

Riccardo Xefraj was with the Department of Information Engineering, University of Pisa, 56122 Pisa, Italy. He is now with Cisco Systems, San Jose, CA, USA (e-mail: riccardo.xefraj@ing.unipi.it).

Digital Object Identifier 10.1109/JIOT.2024.3350333

[1] Introducing heartwood, https://electriccoin.co/blog/introducing-heartwood/.

consumption. We prove that SmartFly is succinct in storage and bandwidth consumption and economically bearable (about 38 euros per day to maintain the whole system).

A. Motivation

Some IoT applications that can leverage the benefits of SmartFly are listed in the following.

1) Supply chain management [16], [17], [18], in which blockchain can be used to track the movement of goods from the manufacturer to the end consumer. IoT devices can exchange data on the location, temperature, and humidity of the goods through the blockchain. This can enhance transparency and accountability, reduce the risk of fraud and counterfeiting, and enable more efficient supply chain management.
2) Energy trading [19], [20], [21], [22], in which blockchain can facilitate peer-to-peer energy trading between households or businesses that generate renewable energy. IoT devices can collect data on the energy production and consumption, which can then be recorded on the blockchain and retrieved and used by other IoT devices. Smart contracts can be used to automate the trading of energy between devices, without the need for intermediaries or central authorities.
3) Smart cities [23], [24], [25], in which blockchain can be used to improve the efficiency and the sustainability. Various IoT sensors and actuators can exchange data on traffic, air quality, waste management, and other urban services through the blockchain. This can enable more effective decision making and resource allocation, and facilitate the creation of new services and applications.

B. Contribution

The contribution of this article is summarized as follows.

1) We show that FlyClient super-lightweight clients are realizable without forks by means of the smart contract technology, without trusting the nodes that invoke the smart contract methods.
2) We present SmartFly, a system that realizes FlyClient on ETC without forks, which takes into consideration the difficulty adjustment mechanism of ETC and possible difficulty tampering attacks.
3) We measure the monetary cost of maintaining SmartFly under various tradeoffs, and its performance in terms of client storage and bandwidth consumption.

C. Paper Structure

The remainder of this article is organized as follows. Section II compares with relevant related work. Section III introduces the necessary preliminary concepts. Section IV introduces our reference threat model. Section V explains how we defend against difficulty tampering attacks. Section VI describes the SmartFly system model. Section VII explains our experiments and shows and discusses the results. Finally, we conclude this article in Section VIII.

II. Related Work

The problem of allowing "light clients" that are able to verify the validity of a transaction without storing and processing the entire blockchain was studied since the very beginning of the blockchain technology, by Satoshi Nakamoto in its seminal whitepaper [26]. Nakamoto proposed the simplified payment verification (SPV) technique, which requires to structure each block as a Merkle tree with the transactions as leaves. This allows clients to store only the header of each block, which contains the Merkle tree root. SPV clients can trustworthily verify transactions assuming that 51% of mining power is honest. SPV clients are now provided by the majority of cryptocurrencies [27], [28]. Although the storage saving with respect to full clients is big, SPV clients still need to store an amount of data that linearly grows with the chain length. This means that an SPV client must nowadays store gigabytes of data to work, which is clearly out of bound for the majority of IoT applications.

Cao et al. [29] proposed CoVer, a method alternative to SPV clients in which light clients cooperate with each other to verify a block validity without trusting any full node. CoVer has the advantage of requiring far less trust compared to standard SPV clients. However, CoVer is "sublinear" only with respect to the block size, in the sense that it requires clients to store only a fraction of each block. It is not sublinear with respect to the chain length, therefore, as in the case of SPV clients, it is inapplicable in constrained IoT devices.

The problem of developing truly sublinear light clients has been approached by the literature with three different ways: 1) superblock-based approaches; 2) reputation-based approaches; and 3) MMR-based approaches. Superblocks are blocks that solve a puzzle that is more difficult than the current difficulty target. Due to the nature of the partial hash preimage puzzles employed in PoW-based cryptocurrencies, superblocks are unintentionally produced by miners with a probability distribution that can be foreseen in advance. This allows light clients to check a small set of superblocks instead of the complete chain, and thus to spend a sublinear amount of storage and bandwidth. Superblock-based approaches were proposed in Bitcoin forums and mailing lists since 2012 [30], [31], but the first scientific proposal was by Kiayias et al. [32], which introduced Proofs of Proof of Work (PoPoW). Successive literature proposed NIPoPoW [33], which is a noninteractive version of PoPoW, and a smart contract realization of it [34]. Bünz et al. [10] analyzed the efficiency of superblock-based and MMR-based approaches, and they concluded that the latter ones consume less bandwidth, security level being equal. Moreover, superblock-based approaches are harder to be applied in variable-difficulty blockchains.

In reputation-based approaches, light clients try to identify trustworthy full nodes by exchanging information about good and bad behaviors and maintaining a reputation system. An example is Debe et al. [35], which implements a node reputation system by means of smart contracts. Interestingly, reputation-based approaches are orthogonal to other approaches, so they can be fruitfully applied together with superblocks or MMRs in order to further improve their security. When applied alone, reputation systems are very cheap in terms of resource consumption, but they do not give any formal security assurance to light clients. For example, a full node can accumulate a good reputation by behaving correctly for some time, and then suddenly "go bad" when there is an opportunity to earn much from a malicious behavior.

Todd [36] first proposed to use MMR data structures in the Bitcoin blockchain. Bünz et al. [10] introduced FlyClient, a sublinear client based on MMR and probabilistic sampling. Their proposal is able to work also in blockchains with variable difficulty, but it requires a fork, either soft or velvet [13]. Unfortunately, a soft fork is politically expensive, because it needs 51% consensus to be successful, whereas a velvet fork can make FlyClient vulnerable to the chain-sewing attack [14]. In this article we show that forks are not necessary, and we investigate the technical challenges of realizing MMR-based light clients by means of smart contracts.

It is worth to say that superblock-based and MMR-based approaches are specific for PoW-based blockchains, and they cannot be applied in blockchains based on Proof of Stake (PoS), like the Ethereum one after the "Merge" fork.[2] In general, it is hard to provide for sublinear clients in PoS-based blockchains, basically because they require clients to download and verify account balances stored in previous blocks in order to validate each single block [28].

III. Preliminaries

A blockchain [26], [37] is an append-only ledger constituted by a hashed chain of data blocks, each of which comprises several transactions. A blockchain is typically maintained by a *blockchain network*, which is a peer-to-peer network of mutually untrusted nodes, by means of some consensus protocol. Blockchain networks are primarily used today for implementing decentralized cryptocurrencies, such as Bitcoin and Ethereum.

The traditional consensus protocol for blockchain networks is the *PoW* consensus. In such a protocol, different nodes (*miners*) compete to each other in order to solve a cryptographic puzzle (*mining*). The first miner that solves the puzzle appends a new block to the blockchain, and it gets a monetary reward. In order to take part to the competition, miners must be *full nodes*, that is they must store locally the whole blockchain, which is typically several gigabytes.[3] If a node is not required to mine but only to check that specific transactions are present in the blockchain, it can save space by storing only the *header* of each block and trusting that the majority of miners are honest. In the present paper, we call *light node* a node that stores all the chain of block headers (*headerchain*). Each block header conveys the Merkle tree root of all the transactions of the block.[4] To check that a transaction is inside a given block, a light node must only retrieve the corresponding Merkle proof from an untrusted full node. Although light nodes can save a significant amount of space with respect to full nodes, their storage requirement still grows linearly with the chain length. Thus, light clients are "light" for full-resource PCs or high-end mobile devices, but they are not light at all for more resource-constrained devices, like those that are employed in embedded computing or the IoT. In this article, we call *sublinear nodes* the nodes whose storage, bandwidth, and computation complexity grows less than linearly with respect to the chain length. Of course, sublinear nodes are best suited for embedded computing and IoT applications.

Within the PoW consensus protocol, the block *difficulty* is defined as the expected number of trials needed to solve its puzzle. The difficulty of two or more blocks is the sum of the single blocks' difficulties. According to the protocol, in the presence of two or more alternative valid chains, a node will give its consensus to the most difficult one.

A. Ethereum Classic Difficulty Adjustment

In PoW-based blockchain networks, the difficulty of the blocks is not constant but it is adjusted through time, in order to keep the expected number of appended blocks constant over time. In the following, given a header x we will indicate with $x.D$ its difficulty and with $x.t$ its mining time. Mining times are expressed in seconds passed since a reference date, and they must be strictly increasing block by block. In ETC, the difficulty of each block x is calculated from the previous block x_p as $x.D = D_{\text{next}}(x_p.D, x_p.t, x.t)$, where

$$\begin{aligned} &D_{\text{next}}\big(x_p.D, x_p.t, x.t\big) \\ &\quad = x_p.D + \left\lfloor \frac{x_p.D}{2048} \right\rfloor \cdot \max\left(1 - \left\lfloor \frac{x.t - x_p.t}{10} \right\rfloor, -99\right). \quad (1) \end{aligned}$$

Such a formula aims at keeping the expected interblock time as close as possible to 10 s with respect to the total mining power employed by the ETC network, while at the same time avoiding abrupt difficulty changes.

Note that the difficulty of a block depends on the timestamp of the same block. As a consequence, it is impossible to determine the difficulty of the block that will be mined next until such a block is actually mined and thus its timestamp is decided. This is in contrast with Bitcoin, in which the difficulty of the next block is determined only by the past blocks.

B. Merkle Mountain Ranges

MMRs [10], [36] are extensions of Merkle trees that allow for efficient appends and *proofs of ancestry*. A proof of ancestry proves that a given MMR M_i is an *ancestor* of another MMR M_j, that is M_j has been built from M_i by executing a number of appends. Fig. 1 exemplifies three append operations that change the MMR M_6 to the MMR M_9.

Each append operation changes a logarithmic number of nodes in the MMR, which are represented by blue-filled nodes in Fig. 1. It is always convenient to represent a whole MMR with its root, so we indicate with the symbol $l_i \in r_j$ that the leaf l_i is member of the MMR rooted by r_j, and with the symbol $r_i \longrightarrow r_j$ that the MMR rooted by r_i is an ancestor of the one rooted by r_j.

[2]The Merge, https://ethereum.org/en/upgrades/merge/.

[3]55 GB for ETC at the time of writing: https://bitinfocharts.com/ethereum%20classic/.

[4]More precisely, to index the various transactions inside the block, ETC uses a *Merkle Patricia trie*, which is a data structure similar to a Merkle tree that offers similar proving guarantees and proof sizes.

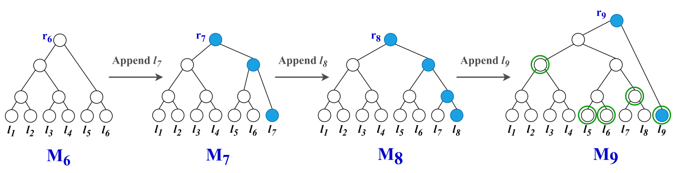

Fig. 1. Examples of MMR append operations and proof of ancestry.

The Merkle proof (also called "proof of inclusion") $\Pi_{l_i \in r_j}$ proves that l_i is member of r_j, and it is composed by all the sibling nodes of the path that connects the leaf l_i to the root r_j in the MMR. The proof of ancestry $\Pi_{r_i \longrightarrow r_j}$ is composed by the rightmost leaf of r_i (l_i), plus the Merkle proof $\Pi_{l_i \in r_j}$. In the example of Fig. 1, the proof that M_6 is an ancestor of M_9 is constituted by the green-circled nodes in M_9. It is easy to see that from these nodes it is possible to compute both the M_6 root (r_6) and the M_9 root (r_9). Verifying an ancestry proof consists in recomputing both roots and checking if they match the given ones. The proof of ancestry $\Pi_{r_i \longrightarrow r_j}$ does not prove only that an MMR is generically an ancestor of another, but also that the former is exactly the ith version and the latter is exactly the jth version. Note that the size of an ancestry proof grows logarithmically with the number of MMR leaves.

C. MMR-Based Sublinear Nodes

Bünz et al. [10] introduced a method to allow sublinear nodes based on MMR data structures and random sampling of the blockchain. MMR-based sublinear nodes require to modify each block header to include the root of an MMR having the hashes of all the preceding block headers as leaves. In this way, given two block headers containing the MMR roots r_i and r_j, it is easy to prove that the entire blockchain preceding the first block is a prefix of the entire blockchain preceding the other block, by simply providing the proof of ancestry $\Pi_{r_i \longrightarrow r_j}$. Given this possibility, a constrained node can avoid downloading the entire headerchain, by requesting and downloading only a sublinear number of randomly chosen headers from an untrusted resourceful node. Assuming for simplicity that the difficulty remains constant, the untrusted resourceful node must prove to the constrained node to own a correctly mined blockchain with a given difficulty and indexed by a given MMR root (*top MMR root*), which must be declared beforehand.

To do that, the two nodes execute a *proving protocol*, during which the sublinear node is the *verifier* and the resourceful node is the *prover*. For each sampled header, the verifier checks its PoW solution validity and that its contained MMR root is an ancestor of the top MMR root. If the verifier samples the headers according to an increasing reciprocal probability distribution, then the protocol will reach some provable level of security against a dishonest prover capable of correctly mining a given fraction of the last part of the blockchain. Therefore, a prover cannot lie on the difficulty of its locally maintained chain, for example pretending that it is more difficult than the honest one. Like light SPV clients do, a sublinear node must query two or more nodes about the difficulty of their locally maintained chain before checking such a claim, under the assumption that at least one of them is honest and declares an honest headerchain. Bünz et al. [10] also extended this method in the variable-difficulty setting, basing on the variable-difficulty Bitcoin backbone protocol model [38]. In Section V-A, we will explain our MMR-based solution for ETC in the variable difficulty setting, and how it differs from the Bünz et al.'s one.

IV. Threat Model

We consider an adversary that wants to "modify" a transaction to the eyes of a victim constrained node. To do that, she[5] makes the constrained node believe that she owns a correctly mined chain more difficult than the honest one, when in fact she does not. The adversary conveniently forks her chain from the honest one, with a forking point that is somewhere before the block that contains the transaction to modify.

Following the work of Bünz et al. [10], we model the adversary as a (c, L)-*adversary*, that is an adversary that cannot produce a fork more difficult than L with a c fraction (or more) of its difficulty being valid. "Valid" here means that the blocks forming such a difficulty contain valid PoW solutions and they respect the difficulty adjustment rule of ETC (see Section III-A). In order to convince the constrained node about the difficulty of her chain, the adversary must undergo a proving protocol. Due to the fact that the constrained node verifies the proof of ancestry of the MMR root contained in each sampled header, the adversary cannot "borrow" blocks from the honest chain in her fork, or "reconnect" her fork to the honest chain, or reuse multiple times her validly mined blocks. Indeed, the ancestry proof would reveal if a block was moved from its original position on the same fork or through different parallel forks.

A. Difficulty Tampering and Difficulty Raising Attack

The proof of ancestry alone does not detect whether the adversary respected the difficulty adjustment rule in her fork, therefore she could mount a *difficulty tampering attack*. Such kind of attack has been considered by Bünz et al. [10] in the original FlyClient paper. In a difficulty tampering attack, the adversary tampers with the difficulties of her fork by imposing one or more impossible difficulty increases that violate the difficulty adjustment rule. It is convenient for the adversary to fake the difficulty to be impossibly high, rather than faking it in other ways. Although this may appear counter intuitive, it actually increases the success probability of the adversary. Indeed, if the adversary increases the difficulty of her blocks of, say, $\times 2$ then she will mine half of the blocks in average. Of course, this would not affect the average time in which she mines a chain with a given difficulty, but on the other hand it would sensibly increase the *variance* of such a time. It turns out that the more difficult the malicious blocks are, the greater will be the chances that the adversary mines a chain more difficult than L in which a c fraction of blocks has valid PoW solutions, hence breaking the (c, L)-adversary model.

[5]From now on, we will refer to the adversary as "she," reconnecting with the tradition in cryptography that names an active adversary "Mallory."

A light node can easily detect the difficulty tampering attack by simply checking that the difficulty adjustment rule have been respected block by block along the whole headerchain. This is harder to achieve for a sublinear node, since it cannot download the whole headerchain.

Of course, an attacker could increase the difficulty of her fork also *without* violating the difficulty adjustment rule. In this case, we talk about *difficulty raising attack*. Bahack [15] first studied the difficulty raising attack against Bitcoin.

B. (Dis)Trusting the Smart Contract Appenders

Since we want to realize and maintain an MMR via smart contract, it is important to analyze the trust assumptions we put on the nodes that invoke the smart contract methods to append new leaves to the MMR (*appenders*). Indeed, the appenders could invoke the smart contract methods providing malicious input arguments. Basically, we do not make any trust assumption about the appenders, except that they periodically invoke the methods to maintain the MMR. This threat model captures a wide range of applications, for example, those in which the appenders are anonymous. In Section VI, we will show that this is feasible provided that the smart contract programming language allows access to the most recent block or its hash. The smart contract language of ETC allows it.

V. Detecting Difficulty Tampering Attack

In the Bitcoin context, Bünz et al. [10] proposed to defend against a dishonest prover mounting a difficulty tampering attack by checking that the chain declared by the prover respects the hypotheses of the variable-difficulty Bitcoin backbone protocol model [38]. If this happens, then we have some formally proved properties of liveness and persistence that lead us to exclude an adversarial fork more difficult than the honest chain. Unfortunately, we cannot do the same for ETC because, at the time of writing, it still does not exist a formal model of it in the literature, which would allow us to prove security properties. We cannot even apply the Bitcoin backbone protocol model, because the ETC difficulty transition rule is different and does not easily fit into the model.

Since we cannot rely on a formal model, the objective of our verifier is to check that the ETC's difficulty transition rule has been respected by the blockchain proposed by the prover in order to exclude the possibility that the prover mounted a difficulty tampering attack. Note that it is impossible to check that the difficulty transition rule has been respected along the whole blockchain unless the verifier stores it all. Hence, the verifer must check the *plausibility* of the declared difficulty transitions. In this way, SmartFly *inherits* the security of ETC against the difficulty raising attack, that is, under the assumption that ETC is secure, then also SmartFly is. Indeed, if the attacker could successfully mount (in a reasonable time) a difficulty raising attack against SmartFly, then she could do it also against a full node. In other words, ETC itself would be vulnerable, without SmartFly's fault.

In the next section, we propose an extension of the MMR data structure that allows the plausibility check in the ETC blockchain.

```
Input: $\{x_1, x_2, \cdots, x_N\}$
1: $l \leftarrow$ empty node
2: $l.h \leftarrow H(x_N)$
3: $l.n \leftarrow N$
4: $l.w \leftarrow \sum_{i=1}^{N} x_i.D$
5: $l.D_f \leftarrow x_1.D$
6: $l.D_l \leftarrow x_N.D$
7: $l.t_f \leftarrow x_1.t$
8: $l.t_l \leftarrow x_N.t$
9: return $l$
```

Fig. 2. MMR leaf creation.

A. Difficulty MMR for Ethereum Classic

We enrich the MMR data structure with information necessary to detect a difficulty tampering attack, thus obtaining a *Difficulty MMR* data structure. Our Difficulty MMR differs slightly from the analogous data structure proposed by [10], because it is suitable for the ETC difficulty adjustment algorithm (see Section III-A), which is different from the Bitcoin one. In particular, the node format differs from [10] because it contains the timestamp and the difficulty of the *last* covered block, rather than the *next* block (i.e., the block successive to the last covered one). This is made necessary due to the difficulty adjustment rule of ETC, for which it is impossible to determine the difficulty of the block that will be mined next until such a block is actually mined.

We define our Difficulty MMR in such a way that each of its leaves can cover a different number of blocks. We call *leaf subchain* the group of consecutive headers covered by a leaf. This feature will be important to save smart contract maintenance cost, by aggregating more blocks in a single leaf. It also provides for flexibility on how frequently the appenders can append leafs. Each node a in our Difficulty MMR contains the following pieces of information.

1) The hash of the last covered header for leaves, or the hash of the concatenated child nodes for nonleaves ($a.h$).
2) The number of blocks covered by the node ($a.n$).
3) The total difficulty of the blocks covered by the node ($a.w$).
4) The difficulty of the first and the last blocks covered by the node (respectively, $a.D_f$ and $a.D_l$).
5) The timestamp of the first and the last blocks covered by the node (respectively, $a.t_f$ and $a.t_l$).

The pseudocode to create leaf and nonleaf nodes is detailed, respectively, in Figs. 2 and 3. In Fig. 2 the inputs $x_1, x_2, \ldots, x_N$ are the headers of the blocks that the MMR leaf will cover, and the output l is the created MMR leaf. In Fig. 3 the inputs a_{lx} and a_{rx} are, respectively, the left and right child nodes, and the output a is the created MMR node.

B. Plausibility Checks for Ethereum Classic

With Difficulty MMRs, we can of course provide for ancestry proofs as with plain MMRs. In addition, with the extra information contained in each node of the proof, we can check the plausibility of the difficulty adjustments of the chain declared by the prover. We will refer hereafter to Difficulty MMRs with simply "MMRs." Note that the MMR

```
Input: $a_{lx}, a_{rx}$
1: $a \leftarrow$ empty node
2: $a.h \leftarrow H(a_{lx}|a_{rx})$
3: $a.n \leftarrow a_{lx}.n + a_{rx}.n$
4: $a.w \leftarrow a_{lx}.w + a_{rx}.w$
5: $a.D_f \leftarrow a_{lx}.D_f$
6: $a.D_l \leftarrow a_{rx}.D_l$
7: $a.t_f \leftarrow a_{lx}.t_f$
8: $a.t_l \leftarrow a_{rx}.t_l$
9: return $a$
```

Fig. 3. MMR nonleaf node creation.

```
Input: $x_0, \{a_1, a_2, \cdots\}$
1: $a_0.D_l \leftarrow x_0.D$
2: $a_0.t_l \leftarrow x_0.t$
3: for all $a_i \in \{a_1, a_2, \cdots\}$ do
4:     if $a_{i-1}.t_l \geq a_i.t_f$ or $a_i.D_f \neq D_{next}(a_{i-1}.D_l, a_{i-1}.t_l, a_i.t_f)$ then
5:         return failure
6:     end if
7:     $D_{max} \leftarrow a_i.D_f$
8:     $w_{max} \leftarrow D_{max}$
9:     $t_{min} \leftarrow a_i.t_f$
10:    for $j \leftarrow 1, 2, \cdots, a_i.n - 1$ do
11:        $D_{max} \leftarrow D_{next}(D_{max}, t_{min}, t_{min} + 1)$
12:        $w_{max} \leftarrow w_{max} + D_{max}$
13:        $t_{min} \leftarrow t_{min} + 1$
14:    end for
15:    if $a_i.D_l > D_{max}$ or $a_i.w > w_{max}$ or $a_i.t_l < t_{min}$ then
16:        return failure
17:    end if
18: end for
19: return success
```

Fig. 4. Difficulty plausibility check.

covers the blockchain only from the first block of the first appended leaf subchain. It is impossible to verify transactions *before* that point in a sublinear fashion. From now on, without losing in generality we will neglect the precedent part of the blockchain and we will consider only the part of the blockchain covered by the MMR. We assume that the header of the block precedent to the first leaf subchain (x_0) is well-known by all the verifiers.

Given an ancestry proof $\Pi_{r_i \longrightarrow r_j} = \{a_1, a_2, \ldots\}$ composed of *ordered* MMR nodes, in the sense that they cover the chain from left to right, the *difficulty plausibility check* verifies the adjustments node by node in the following way. The difficulty adjustment between two nodes is checked exactly with (1). On the other hand, the difficulty adjustment within the same node is checked against an upper bound $D_{\max}$, which is computed by considering the greatest possible difficulty raising, corresponding to the case in which each block covered by the node was mined in 1 s. If the difficulty of the last block covered by the node, and the total difficulty and the total mining time of the blocks covered by the nodes are below such a bound, then all the difficulty adjustments are plausible. The pseudocode of the plausibility check is detailed in Fig. 4, where $D_{\text{next}}(\cdot, \cdot, \cdot)$ is the difficulty adjustment function defined in Section III-A. Line 4 implements the difficulty adjustment check between two nodes, whereas line 15 implements that within the same node.

C. Security Against the Difficulty Tampering Attack

It is possible to demonstrate that a SmartFly verifier is secure also in case the difficulty adjustment rule is violated by the adversary. Our proof strategy is to show that a (c, L)-adversary *remains* a (c, L)-adversary also if she performs a difficulty tampering attack, that is, by violating the adjustment rule she cannot produce a fork more difficult than L with a c fraction (or more) of its difficulty being valid. If we manage to prove that, then the security proof in Bünz et al. [10] provides SmartFly for a guaranteed level of security even in case of a difficulty tampering attack. We have to prove that a (c, L)-adversary gains no advantage in violating the adjustment rule, that is she remains a (c, L)-adversary even if she performs a difficulty tampering attack. We prove this by showing that any successful difficulty tampering (c, L)-adversary can be reduced to an equally successful (c, L)-adversary that never violates the adjustment rule and has the same mining power. First of all, let us point out that each leaf subchain in which the adversary violates the adjustment rule is not valid even if it is correctly mined. Indeed, if the verifier samples it, then the proving protocol will fail anyway, since the verifier checks the adjustment rule for each block inside the subchain. Thus, each rule-violating subchain must lie between two sampled subchains, or between the first appended subchain and the first sampled subchain. But in both these intervals the verifier checks the plausibility of the cumulative difficulty transition. If the adversary breaks the plausibility, then the proving protocol will fail, and she will not be successful. On the other hand, if the adversary does not break the plausibility, then a (c, L)-adversary with the same mining power and that never violates the adjustment rule will be able to produce exactly the same valid subchain. This proves that, while the difficulty plausibility check does not prevent an adversary from violating the adjustment rule in one or more blocks, it nevertheless nullifies any real advantage in doing so. To sum up, a (c, L)-adversary remains a (c, L)-adversary also if she performs a difficulty tampering attack, and thus Bünz et al. [10] provided SmartFly for a guaranteed level of security even in case of a difficulty tampering attack. This concludes our proof.

VI. System Model

In this section, we describe the SmartFly system and its components. Fig. 5 shows the SmartFly reference architecture. The components of the system are the following ones.

1) The *SmartFly contract*, which is a smart contract that maintains the MMR and stores the value of the MMR root inside the blockchain.
2) The *SmartFly appenders*, which are nodes that trigger the execution of the MMR append operation of the SmartFly contract, and provide the necessary input information to perform such an operation. Each append operation adds a new leaf to the MMR, covering a leaf subchain.

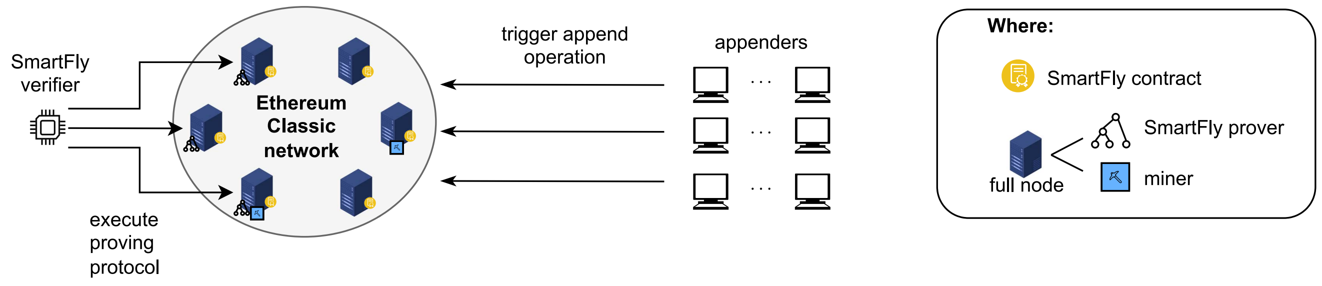

Fig. 5. SmartFly reference architecture.

3) The *SmartFly provers*, which are nodes able to execute instances of the proving protocol with the verifiers. Provers must store off-chain the full MMR.
4) The *SmartFly verifiers*, which are low-resource devices that want to verify the presence of a given transaction in the blockchain. To do that, they execute the proving protocol with one or more provers.

The appenders bear the cost of executing the MMR append operations, so it is necessary to provide an incentive for them to do that. To this aim, it is convenient that the appenders are run by the same companies that control one or more verifiers and enjoy the proving service. The provers themselves could give incentives for such companies to pay for the MMR maintenance, for example by refusing to run proving protocols with the ones that performed no or too few append operations in the recent past.

A. MMR Representation

In ETC, smart contracts are written in the *EVM bytecode*, which is a Turing-complete language. Each EVM instruction has an associated cost, which is measured in *gas*. In turn, gas has a price (*fee*) measured in Gwei, which a node must pay in order to get a smart contract method executed. In the EVM bytecode, the most convenient place where to store the MMR is the *storage*, which is a read/write persistent memory [39]. It is possible to implement MMR append operations by simply storing the *peak set*, which is the minimal set of MMR nodes that cover all the leaves and whose subtrees are complete. Indeed, from the peak set it is possible to append a new leaf, recompute a new peak set (which may be smaller than the old one), and from that recompute a new root. This allows us to put only the peak set into the SmartFly contract storage, and to program the append operation in such a way that it modifies a constant number of peaks, thus minimizing the maintenance cost. After having updated the peak set, the append operation computes the new root (r) and writes it inside the blockchain. This operation has logarithmic complexity since it depends on the number of changed MMR nodes, which is logarithmic (cfr. Section III-B). To save money, it is convenient to store only the root hash ($r.h$) inside the blockchain, which is equivalent in terms of security. Since the SmartFly contract does not need to read again such a root hash afterwards, we store it inside *event logs*, which are cheap and write-only memory that represents the output of smart contract executions [39].

B. MMR Append Operation

In order to put the least possible trust on the appender, the safest thing would be that the append operation of the smart contract accepts no input, any part of which could be malicious. All the information to build the new leaf should be retrieved from storage or from built-in functions of the smart contract language. This is possible if the smart contract language lets us access the fields of the past block headers. Unfortunately, the EVM bytecode lets us access only the *hashes* of the most recent 256 block headers. To overcome this, the SmartFly contract receives in input from the appenders all the headers of the leaf subchain. Upon receiving the block headers, the contract checks their validity by recomputing their hashes, and by comparing them with the ones accessible via the EVM bytecode. For the headers precedent to the 256 most recent ones, the smart contract checks that they are concatenated to the ones accessible via the EVM bytecode, by using the block header field containing the hash of the previous block. If the check fails, then the append operation is reverted with a `REVERT` instruction. This prevents that malicious appenders update the MMR with invalid blocks, or with valid blocks belonging to forks different from the one that miners gave consensus to. Also, the execution revert makes the malicious appender spend money uselessly, so this constitutes a disincentive for attacks.

Note that the aforementioned technique is feasible also if the smart contract language would allow us to access the most recent block's hash only, instead of the most recent 256 ones. This proves that fork-free sublinear clients are feasible without trusting the appenders in case the smart contract language provides a means to access the most recent block or its hash.

It is also possible that an append operation from an honest appender gets reverted. This can happen for example when the blockchain maintained by the appender differs from the one maintained by the miners due to a temporary fork caused by network delays. In this case, the cost of the execution revert constitutes an incentive to append only stable blocks.

Except for the first appended leaf subchain, the contract also checks that one of the input block headers contains the previous MMR root. This is required in order the proving protocol to work (see Section VI-C). The check is done by simply storing in the smart contract the index of the block in which the append operation has been requested and by checking, during the following invocation, that the block with that index has been provided in input. Even here, if the check fails then the append operation is reverted. If all the checks pass, then the contract builds a new MMR leaf from the received leaf subchain, and it appends it to the MMR.

Each append operation appends a single MMR leaf, but an MMR leaf can cover a subchain of multiple consecutive blocks. On the other hand, each mined block may not contain append operations. This gives a great flexibility to the appenders, because the growth rate of the MMR can temporarily lag behind that of the blockchain. Of course, the part of the blockchain which is not yet covered by the MMR cannot be involved in the proving protocols.

Even if the MMR is updated sporadically like it happens in a velvet-fork-based deployment of FlyClient [14], the smart-contract solution does not give space to the same vulnerabilities. The reason is that a velvet fork divides the network in *updated* miners (i.e., miners that adopt the new protocol) and *nonupdated* miners (i.e., miners that do not). To maintain backward compatibility, a block mined by an updated miner is valid for a nonupdated miner as well. Therefore, a malicious updated miner can put arbitrary data in the field containing the MMR root, and all honest nonupdated miners will continue mining over it. Such an unchecked MMR root can be used to perform a chain-sewing attack. A smart-contract solution avoids the attack at its root, because it makes no such a distinction between updated and nonupdated miners. Every miner executes the smart contract in the same way, so no honest miner will mine over an invalid MMR root.

C. Proving Protocol

Suppose that a constrained node wants to check the presence of some transaction in the blockchain. First of all, the verifier retrieves some public list of active SmartFly provers. The node then chooses k provers to query either randomly or by some reputation system, which is outside the scope of this article. The constrained node asks to each of these k provers to declare the root of their locally maintained MMR (r_{tot}), which contains the root hash ($r_{\text{tot}}.h$) and the difficulty of their locally maintained headerchain ($r_{\text{tot}}.w$). After that, the constrained node starts querying the prover that declared the most difficult chain with a proving protocol, in which the constrained node plays the role of the verifier. If the prover fails the proving protocol, then the constrained node queries the prover that declared the second most difficult chain, and so on. The proving protocol aims at convincing the verifier that the prover actually stores a valid chain of the declared difficulty. The verifier downloads from the prover a sublinear number of headers, together with various proofs that convinces the prover about their validity. If all the downloaded headers are valid, then the prover will consider the whole chain declared by the prover as valid. In particular, assuming that we want to guarantee λ bits of security level against a (c, L)-adversary, the verifier downloads the most recent n headers (*deterministic sampling*), with n being the smallest number of headers necessary to: 1) make a total difficulty greater than or equal to L and 2) cover the block conveying the MMR root hash declared beforehand by the prover. Then, the verifier downloads m leaf subchains at randomly chosen heights of the total difficulty (*probabilistic sampling*), with

$$m = \frac{\lambda}{\log_{0.5}\left(1 - 1/\log_c\left(\frac{L}{r_{\text{tot}}.w}\right)\right)}. \quad (2)$$

The probability distribution according to which the verifier randomly chooses the heights w to sample is

$$f(w) = \frac{r_{\text{tot}}.w}{(w - r_{\text{tot}}.w)\ln\left(\frac{L}{r_{\text{tot}}.w}\right)} \text{ with: } w \in [0, (r_{\text{tot}}.w) - L]. \quad (3)$$

The proof that such deterministic and probabilistic samplings guarantee λ bits of security level against a (c, L)-adversary is similar to the one in [10]. The verifier downloads the leaf subchains that include the blocks at such difficulty heights. During the deterministic sampling, the verifier downloads also the Merkle proof that the event log conveying the MMR root hash is actually inside one of the blocks. The verifier checks the validity of the downloaded headers (i.e., it checks the validity of the PoW solutions and the block-by-block difficulty adjustments), and the Merkle proof. During the probabilistic sampling, for each sampled leaf subchain, the prover downloads also: 1) the relative MMR hash root, which is inside the event logs of one of the blocks; 2) the Merkle proof that proves that such event log is actually inside one of the blocks; and 3) the ancestry proof that proves that the MMR hash root is actually a previous version of the MMR locally maintained by the prover. The verifier checks the validity of the leaf subchain (i.e., it checks the validity of the PoW solutions and the block-by-block difficulty adjustments), the Merkle proof, the ancestry proof, and the difficulty plausibility of the ancestry proof. To decrease the amount of data downloaded by the verifier, we apply simple optimizations that avoid the verifier to download twice the same data. In particular, if the verifier randomly chooses a difficulty height included in a leaf subchain that has been already sampled before, the sampling is avoided totally. Moreover, the verifier avoids downloading twice the same MMR nodes, or MMR nodes that are recomputable from leaf subchains or from other MMR nodes that it already downloaded.

We can obtain a noninteractive version of the proving protocol just described by applying the Fiat–Shamir heuristic [40]. In particular, which blocks are sampled is determined not at random, but according to a secure hash function applied to the MMR root r_{tot} that the prover declared at the beginning of the protocol. The noninteractive version is more practical than the interactive one, because it allows the prover to be offline while the verifier executes the protocol. For example, the prover could publish the noninteractive proof on a website, or the noninteractive proof could be sent to the verifier by another verifier in a peer-to-peer fashion.

After the proving protocol is concluded successfully, the constrained device can permanently store only the MMR root hash, so the SmartFly system is very succinct in memory. When the device wants to verify a transaction, it downloads from some untrusted prover the leaf subchain of the block containing the transaction, the Merkle proof that such a transaction is inside a block of the leaf subchain, and the Merkle proof that such a leaf subchain is inside the MMR.

VII. Experiments

In this section we experimentally evaluate the SmartFly contract monetary cost and the proof size, which is the total amount of data downloaded by the verifier during an execution of the proving protocol. Table I shows a summary of the parameters employed in the experiments with their values.

TABLE I
PARAMETERS OF THE EXPERIMENTS

Parameter	Value
Average fee	10 Gweis/gas
Gwei/euro exch. rate	$25 \cdot 10^{-9}$ euros/Gwei
# of blocks per leaf	$\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$
Chain length	$\{2048, 46080, 92160, 138240, 184320, 230400\}$
c	$\{0.1, 0.3, 0.5\}$
L	$30D$
Security level (λ)	$\{50, 80, 100\}$

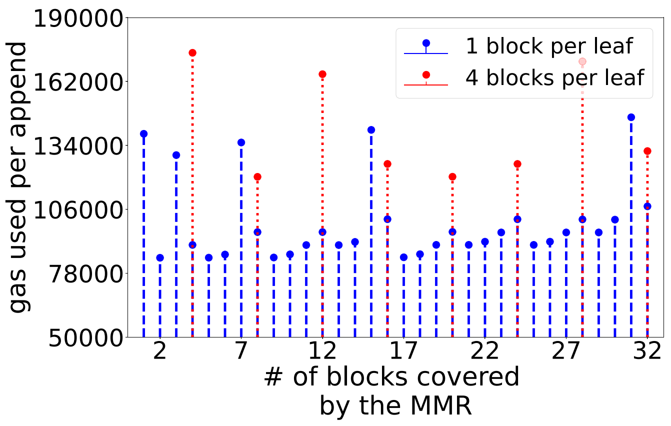

Fig. 6. Costs of an example sequence of append operations.

A. SmartFly Contract Monetary Costs

In order to evaluate the SmartFly contract costs we implemented it in the Solidity[6] language, and we statically validated it using Mythril[7] to eliminate possible bugs. We then created a private ETC blockchain using Ganache,[8] and we compiled and deployed the contract on the chain using Truffle.[9] In order to evaluate the cost of SmartFly deployment and maintenance we extracted the gas consumed from the receipt of the relative transactions. The cost of deploying the SmartFly contract on the chain is 1 842 811 gas, which is quite cheap. In order to get a concrete idea of the cost, we convert gas into Gweis and then in turn into euros, by applying, respectively, the average fee[10] and the average Gwei-euro exchange rate[11] over the year 2021. Using these exchange rates, the SmartFly contract deployment costs only 0.46 euros.

Fig. 6 shows the cost of an example sequence of MMR append operations with one block per leaf and four blocks per leaf. The abscissas show the number of blocks covered by the MMR. We can notice that there are some high-cost peaks, whose frequency decreases as the number of covered blocks increases. This phenomenon can be observed when the number of blocks covered by the MMR are 1, 3, 7, 15, and 31 (with one block per leaf), and 4, 12, and 28 (with four blocks per leaf). This happens when the size of the peak set reaches a new maximum, so the contract needs to allocate new storage to accommodate it. Excluding the cost peaks, we can also notice an approximate saw-tooth shaped trend, which is due to the cost of accessing the peak set stored in the blockchain. Indeed, due to how the MMR is structured, at each append the peak set grows or shrinks with a saw-tooth trend.

[6]Solidity, https://soliditylang.org/.
[7]Mythril, https://github.com/ConsenSys/mythril/.
[8]Ganache, https://trufflesuite.com/ganache/.
[9]Truffle, https://trufflesuite.com/.
[10]1 gas equals 10 Gweis, https://explorer.bitquery.io/ethclassic/.
[11]1 ETC (i.e., 1 000 000 000 Gweis) equals 25 euros, https://finance.yahoo.com/quote/ETC-EUR/.

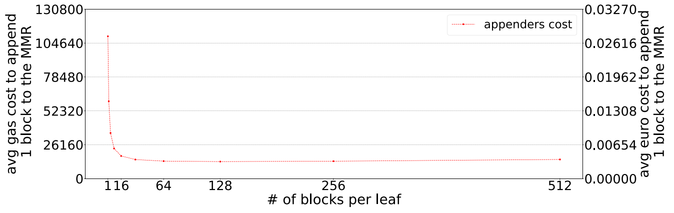

Fig. 7. Average gas and euro cost to insert one block in the MMR with different values of blocks per leaf and a chain of 2048 blocks.

Fig. 7 shows the average gas and euro cost per block to append the first 2048 blocks in the MMR with different values of blocks per leaf. With 16 blocks per leaf, an appender spends 0.0044 euros per block. This means that the appenders would collectively spend about 38 euros per day to maintain the whole system. The per-block cost paid by appenders does not approach zero, because the input size of a single append operation grows linearly with the number of blocks per leaf. Moreover, we experienced that the cost slightly increases after the limit of 128 blocks per leaf. This occurs because the cost of allocating storage in the EVM bytecode increases quadratically after ≈ 724 allocated bytes, which is a design choice of the EVM bytecode to defend against memory DoS attacks [27].

ETC imposes a maximum amount of gas that all the transactions inside a block are allowed to consume (*block gas limit*), which is currently fixed at 8 500 000 gas. This poses a theoretical limit to the number of blocks per leaf that can be used in an append operation, which is approximately 560 blocks per leaf.

B. Proof Size

To evaluate the proof size, we simulated a number of proving protocols between a prover and a verifier. For simplicity, we suppose that the blocks held by the prover have constant difficulty D, whose value is irrelevant to the simulation results. Note that this is a pessimistic assumption from the performance point of view, since real variable-difficulty blockchains tend to have much more difficulty distributed on recent blocks rather than on old ones. Thus, the verifier will sample recent leaf subchains with much more probability, and this allows it to leverage more effectively the optimizations to reduce the proof size described in Section VI-C. Regarding the adversary, we fix $L = 30D$. This choice stems from the common assumption that no adversary is capable of correctly mining a fork more difficult than $30D$, which is implicitly done by ETC community when recommending 30 confirmation blocks before considering a transaction immutable.[12] To obtain more meaningful statistical results, we performed 30 independent repetitions of each simulation.

[12]ETC Frequently Asked Questions, https://bitni.com/page/ETC-etc-faq.

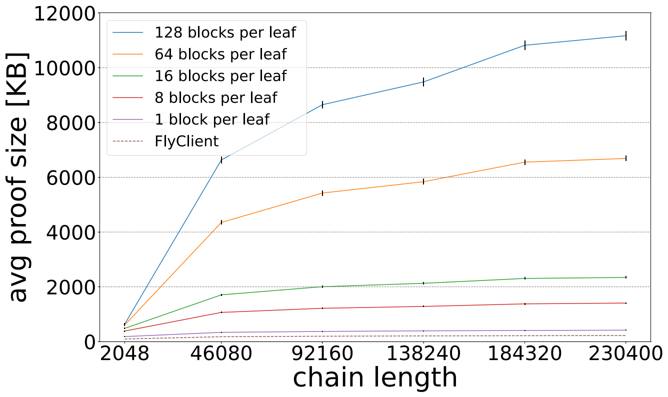

Fig. 8. Average proof size w.r.t. number of blocks in the chain with $\lambda = 50$ and $c = 0.5$. 95%-confidence intervals are displayed in error bars.

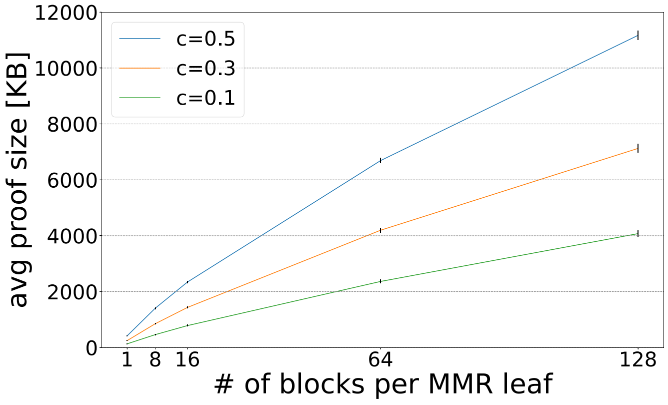

Fig. 9. Average proof size w.r.t. number of blocks per leaf, with $\lambda = 50$ and a chain of 230 400 blocks.

Fig. 8 shows the average proof size of SmartFly compared with that of FlyClient as the chain length grows, with $\lambda = 50$, $c = 0.5$, and different numbers of blocks per leaf. As expected, the proof size grows logarithmically with the chain length. There is also a constant component that depends on L, which becomes prevalent as the number of blocks per leaf decreases. The SmartFly proof size approaches that of FlyClient in case of one block per leaf, but FlyClient remains more efficient. This is because in FlyClient the MMR root hash is directly inside the block header, whereas in SmartFly it is inside an event log. Therefore, the SmartFly proving protocol needs an additional Merkle proof to prove that such an event log is covered by the downloaded blocks, as said in Section VI-C.

Fig. 9 shows the average proof size with respect to the number of blocks per leaf with a chain of 230 400 blocks, corresponding to an average mining time of 640 h. We fixed $\lambda = 50$ and tested different values of c. As expected, the proof size grows with the number of blocks per leaf. This defines a tradeoff between the cost of the append operation and the proof size. In other words, the more the appenders pay, the less data the verifiers download, and vice versa. Note that the size grows only sublinearly, due to the proving protocol optimizations. Also, the size greatly depends on the adversarial power expressed by c: the more powerful the adversary is (i.e., the greater is c), the more data have to be downloaded by the verifiers.

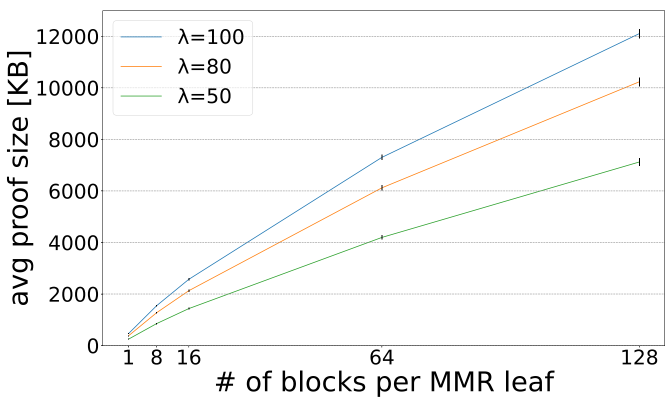

Fig. 10. Average proof size w.r.t. number of blocks per leaf, with $c = 0.3$ and a chain of 230 400 blocks.

Fig. 10 shows again the average proof size with respect to the number of blocks per leaf with a chain of 230 400 blocks, but now fixing $c = 0.3$ and varying λ. Note that the size depends roughly linearly on the desired security level. The more security the verifier wants, the more data it has to download. Of course, different verifiers can choose different security levels in the same SmartFly system, depending on the criticality of the particular IoT application they realize.

VIII. Conclusion

In this article we explored the possibility to realize FlyClient super lightweight clients for PoW-based blockchains without forks, by means of the smart contract technology. This comes at the cost of some cryptomoney to pay in order to deploy the smart contract and invoke its methods. We showed that this approach is generally feasible without trusting the nodes that invoke the smart contract methods, provided that the smart contract programming language allows access to the most recent block or its hash. We further presented SmartFly, a system based on smart contracts that realizes FlyClient on ETC without forks. SmartFly is applicable also to the newly created EthereumPoW cryptocurrency.[13]

Acknowledgment

The authors thank Mariano Basile for helpful discussions in the early stages of this work.

References

[1] M. A. Khan and K. Salah, “IoT security: Review, blockchain solutions, and open challenges,” *Future Gener. Comput. Syst.*, vol. 82, pp. 395–411, May 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17315765

[2] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, “On blockchain and its integration with IoT. Challenges and opportunities,” *Future Gener. Comput. Syst.*, vol. 88, pp. 173–190, Nov. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17329205

[3] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, “Blockchain for IoT security and privacy: The case study of a smart home,” in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, 2017, pp. 618–623.

[4] O. Novo, “Blockchain meets IoT: An architecture for scalable access management in IoT,” *IEEE Internet Things J.*, vol. 5, no. 2, pp. 1184–1195, Apr. 2018.

[13]EthereumPoW, https://ethereumpow.org/.

[5] Z. Baozhi, Y. Junyan, L. Rongsheng, and S. Shanting, "Research on the application of blockchain technology in ubiquitous power system Internet of Things," in *Proc. Proc. 2nd Int. Conf. Blockchain Technol. Appl.*, 2020, pp. 118–123.

[6] C. Chen, M. Liu, P. Mo, C. Yuan, and P. Dai, "LBLCO: A lightweight blockchain with low communication overhead for Internet of Things," in *Proc. 4th Blockchain Internet Things Conf.*, 2022, pp. 92–99.

[7] P. Urien, "Blockchain IoT (BIoT): A new direction for solving Internet of Things security and trust issues," in *Proc. 3rd Cloudification Internet Things (CIoT)*, 2018, pp. 1–4.

[8] *IEEE Standard for Framework of Blockchain-Based Internet of Things (IoT) Data Management*, IEEE Standard 2144.1-2020, 2021.

[9] P. Urien, "Blockchain transaction protocol for constraint nodes," Internet Engineering Task Force, Internet-Draft draft-urien-core-blockchain-transaction-protocol-10, Jun. 2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-urien-core-blockchain-transaction-protocol/10/

[10] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "FlyClient: Super-light clients for cryptocurrencies," in *Proc. IEEE Symp. Security Privacy (SP)*, 2020, pp. 928–946.

[11] M. Beck, "Into the ether with Ethereum classic," Grayscale, Stamford, CT, USA, White Paper, 2017.

[12] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, *Zcash Protocol Specification*, GitHub, San Francisco, CA, USA, 2016.

[13] A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, and W. J. Knottenbelt, "A wild velvet fork appears! Inclusive blockchain protocol changes in practice," in *Financial Cryptography Data Security*. Heidelberg, Germany: Springer, 2019, pp. 31–42.

[14] T. Nemoz and A. Zamyatin, "On the deployment of FlyClient as a velvet fork: Chain-sewing attacks and countermeasures," IACR, Bellevue, WA, USA Rep. 2021/782, 2021. [Online]. Available: https://eprint.iacr.org/2021/782

[15] L. Bahack, "Theoretical Bitcoin attacks with less than half of the computational power (draft)," 2013, *arXiv:1312.70133*.

[16] A. E. C. Mondragon, C. E. C. Mondragon, and E. S. Coronado, "Feasibility of Internet of Things and agnostic blockchain technology solutions: A case in the fisheries supply chain," in *Proc. IEEE 7th Int. Conf. Ind. Eng. Appl. (ICIEA)*, 2020, pp. 504–508.

[17] S. Madumidha, P. S. Ranjani, S. S. Varsinee, and P. Sundari, "Transparency and traceability: In food supply chain system using blockchain technology with Internet of Things," in *Proc. 3rd Int. Conf. Trends Electron. Inform. (ICOEI)*, 2019, pp. 983–987.

[18] A. Arena, A. Bianchini, P. Perazzo, C. Vallati, and G. Dini, "BRUSCHETTA: An IoT blockchain-based framework for certifying extra virgin olive oil supply chain," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, 2019, pp. 173–179.

[19] M. J. A. Baig, M. T. Iqbal, M. Jamil, and J. Khan, "IoT and blockchain based peer to peer energy trading pilot platform," in *Proc. 11th IEEE Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON)*, 2020, pp. 402–406.

[20] H. Wang, S. Ma, C. Guo, Y. Wu, H.-N. Dai, and D. Wu, "Blockchain-based power energy trading management," *ACM Trans. Internet Technol.*, vol. 21, no. 2, p. 43, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3409771

[21] A. Kumari, R. Gupta, S. Tanwar, S. Tyagi, and N. Kumar, "When blockchain meets smart grid: Secure energy trading in demand response management," *IEEE Netw.*, vol. 34, no. 5, pp. 299–305, Sep./Oct. 2020.

[22] A. Kumari et al., "Blockchain-driven real-time incentive approach for energy management system," *Mathematics*, vol. 11, no. 4, p. 28, 2023. [Online]. Available: https://www.mdpi.com/2227-7390/11/4/928

[23] M. A. Rahman, M. M. Rashid, M. S. Hossain, E. Hassanain, M. F. Alhamid, and M. Guizani, "Blockchain and IoT-based cognitive edge framework for sharing economy services in a smart city," *IEEE Access*, vol. 7, pp. 18611–18621, 2019.

[24] H. Mora, J. C. Mendoza-Tello, E. G. Varela-Guzmán, and J. Szymanski, "Blockchain technologies to address smart city and society challenges," *Comput. Human Behav.*, vol. 122, Sep. 2021, Art. no. 106854. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563221001771

[25] A. Kumari, R. Gupta, S. Tanwar, and N. Kumar, "A taxonomy of blockchain-enabled softwarization for secure UAV network," *Comput. Commun.*, vol. 161, pp. 304–323, Sep. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366420318545

[26] S. Nakamoto (Bitcoin, Las Vegas, NV, USA). *Bitcoin: A Peer-to-Peer Electronic Cash System*. (2008). Accessed: May 31, 2022. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[27] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," Ethereum, Zug, Switzerland, White Paper, 2014. Accessed: May 31, 2022. [Online]. Available: https://bit.ly/3aaiR9x

[28] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "SoK: Blockchain light clients," in *Financial Cryptography Data Security*, I. Eyal and J. Garay, Eds. Cham, Switzerland: Springer Int., 2022, pp. 615–641.

[29] S. Cao, S. Kadhe, and K. Ramchandran, "CoVer: Collaborative light-node-only verification and data availability for blockchains," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, 2020, pp. 45–52.

[30] "The high-value-hash highway." 2012. Accessed: May 31, 2022. [Online]. Available: https://bitcointalk.org/index.php?topic=98986.0

[31] M. Friedenbach. "Compact SPV proofs via block header commitments." 2014. Accessed: May 31, 2022. [Online]. Available: https://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg04318.html

[32] A. Kiayias, N. Lamprou, and A.-P. Stouka, "Proofs of proofs of work with sublinear complexity," in *Financial Cryptography Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds. Heidelberg, Germany: Springer, 2016, pp. 61–78.

[33] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," in *Financial Cryptography Data Security*, J. Bonneau and N. Heninger, Eds. Cham, Switzerland: Springer Int., 2020, pp. 505–522.

[34] S. Daveas, K. Karantias, A. Kiayias, and D. Zindros, "A gas-efficient superlight Bitcoin client in solidity," in *Proc. 2nd ACM Conf. Adv. Financ. Technol.*, 2020, pp. 132–144.

[35] M. Debe, K. Salah, R. Jayaraman, I. Yaqoob, and J. Arshad, "Trustworthy blockchain gateways for resource-constrained clients and IoT devices," *IEEE Access*, vol. 9, pp. 132875–132887, 2021.

[36] P. Todd, "Merkle mountain ranges." 2012. Accessed: May 31, 2022. [Online]. Available: https://bit.ly/3x4d5iJ

[37] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton Univ. Press, 2016.

[38] J. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin backbone protocol with chains of variable difficulty," in *Advances in Cryptology*, J. Katz and H. Shacham, Eds. Cham, Switzerland: Springer Int., 2017, pp. 291–323.

[39] P. Kostamis, A. Sendros, and P. S. Efraimidis, "Exploring Ethereum's data stores: A cost and performance comparison," 2021, *arXiv:2105.10520*.

[40] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology*, A. M. Odlyzko, Ed. Heidelberg, Germany: Springer, 1987, pp. 186–194.

Pericle Perazzo received the master's (cum laude) degree in computer engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 2010 and 2014, respectively.

During his Ph.D. studies, he was a Visiting Researcher with the Institute for Parallel and Distributed Systems of Stuttgart, Stuttgart, Germany. Since 2017, he has been a Researcher with the Department of Information Engineering, University of Pisa. His research interests include the area of security and privacy in the IoT, with special emphasis on attribute-based encryption, post-quantum cryptography, and blockchain technologies.

Riccardo Xefraj received the master's (cum laude) degree in computer engineering from the University of Pisa, Pisa, Italy, in 2021.

In 2022 won a scholarship, founded by the University of Pisa, to study and implement cryptographic solutions applied to constrained environments. He currently works as a Software Engineer with Cisco Systems, San Jose, CA, USA. His research interests include communication security, cryptography, and blockchain security.

Open Access funding provided by "Università di Pisa" within the CRUI CARE Agreement