Received 24 May 2024, accepted 12 June 2024, date of publication 24 June 2024, date of current version 1 July 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3418749

RESEARCH ARTICLE

Combining Git and Blockchain for Trusted Information Sharing

LUCA GRILLI AND PAOLO SPEZIALI
Department of Engineering, University of Perugia, 06125 Perugia, Italy

Corresponding author: Luca Grilli (luca.grilli@unipg.it)

This work was supported in part by the PRIN (Progetti di Rilevante Interesse Nazionale) PNNR (Piano Nazionale di Ripresa e Resilienza) 2022 Project ACOI–Assessing Conflict of Interest, funded by the European Union–Next Generation EU.

ABSTRACT We present *PineSU*, a lightweight system that integrates Git with the Ethereum blockchain for sharing electronic documents, enabling decentralized integrity protection and timestamping. PineSU introduces the concept of *Storage Unit* (*SU* for short), which is essentially a Git repository along with some descriptor files needed to interact with the blockchain. SUs can be *open* or *closed*. Open SUs serve to secure Git repositories whose content may change in the future. At any moment, users can create a *Blockchain Synchronization Point* (*BSP* for short) of their open SUs. This allows for a rigorous integrity and authenticity verification of the corresponding digital documents. Whereas closed SUs are mainly a mechanism to invalidate any change to a Git repository. They are useful when a set of files must be definitively archived and made immutable, while enabling their sharing securely. As shown by a case study on clones of two public repositories on GitHub (owned by the Italian government) containing reports and data about the COVID-19 diffusion, PineSU has proven to be very effective in protecting Git repositories under a few security hypotheses that are easy to guarantee in many circumstances. Furthermore, an experimental and simulated performance evaluation demonstrates that the system scales well for storage units of increasing sizes and structure complexity. Finally, a qualitative comparison with existing solutions shows the strengths of PineSU against state-of-the-art approaches.

INDEX TERMS Blockchain, control version systems, Ethereum, Git, trusted data sharing, COVID-19.

I. INTRODUCTION

Developing effective solutions for sharing data is crucial to speed up digital business transformation. As reported by the Data Governance Act proposal of the European Commission [1], [2], by 2028 the data economy and the economic value of data sharing should rise to €533 billion, under the baseline scenario. This would increase to €544.4 billion if specific actions will be implemented, including mechanisms for the enhanced use of public sector data. Moreover, according to Gartner, Inc. [3], the promotion of data sharing can be a primary competitive factor for organizations, but the lack of adequate security may discourage it.

Although many best practices and recommendations have been issued for secure data sharing [4], [5], several challenges still remain. Besides the CIA triad, ensuring data trustworthiness and high levels of trust in data sources will be essential to achieve business value. In this respect, blockchain and DLT technologies may play a central role in many application domains; a limited list includes IoT [6], financial services [7], healthcare services [8] and digital government [9], [10], [11]. Another challenge is to ensure *data interoperability*, namely, the possibility of easily sharing, gathering and processing data from different IT infrastructures by using standardized formats and protocols [12]. Otherwise, only a few large online platforms could benefit from the economic opportunities of the digital economy. The Solid project, led by Sir Tim Berners-Lee [13], represents one of the most ambitious attempts to provide a set of open standards and technologies for data interoperability.

This work focuses on sharing and tracking changes to digital resources in a secure, trusted, and verifiable manner.

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Sookhak.

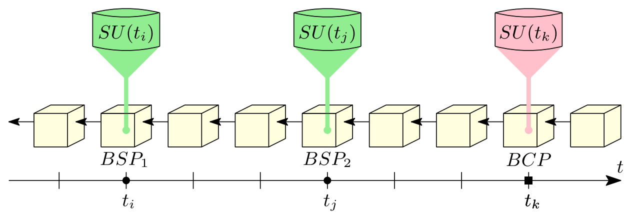


FIGURE 1. Illustration of a series of blockchain registration points for a storage unit.

By digital resources we mean a single file, a group of files, or a portion of file system. The idea is to combine control version systems with blockchain technologies, enabling decentralized integrity protection and timestamping. As control version system we consider Git [14], which is arguably the most effective and versatile one. Millions of software developers use Git for collaborating and sharing their code bases. Additionally, Git's distributed nature facilitates its integration into existing IT infrastructures, promoting data interoperability. However, Git alone is not suitable for applications that require strong protection of integrity and authenticity, such as e-government and health care services.

The aim of this work is to fill this gap by extending Git's functionality with a lightweight software layer, called *PineSU*, which is backed up by a blockchain ledger. PineSU logically wraps a Git repository into a so-called *Storage Unit* (*SU*) by adding specific metadata into the repository itself. SUs can be in one of two basic states: *open* or *closed*. Open SUs are used to secure Git repositories whose content may change in the future. At any moment, users can create a *Blockchain Synchronization Point* (*BSP* for short) of their open SUs (see, e.g., Fig. 1). This allows for common notarization operations, such as proving that a file is unchanged, confirming the origin of a file, confirming that a file already existed at a certain date, an so on. Instead, closed SUs are mainly a mechanism to invalidate any change to a Git repository. They are useful when a set of files must be definitively archived and made immutable, while enabling a blockchain-based integrity and authenticity verification of their content.

PineSU has been designed with the following philosophy in mind:

- The system should be as lightweight as possible, without relying on any DBMS technology. To keep track of all the blockchain operations, only PineSU-dedicated Git repositories should be used.
- SUs should add as little overhead as possible to the Git repositories they wrap.
- There should be mechanisms to reduce blockchain-related costs, minimizing the number of transactions, and the amount of computational resources (memory and CPU usage) of blockchain nodes. In particular, it should be possible to synchronize or close more SUs by executing only one transaction.
- PineSU should be largely blockchain agnostic, and thus easy to adapt to different kind of blockchains, including blockchains with limited programmability (e.g., Bitcoin).
- Git itself should be easily replaceable by any other distributed version control system.

The contribution and the structure of this article are as follows.

- In Section II we provide the necessary background to make the paper self-contained.
- Section III summarizes the existing projects and the scientific literature related to our work.
- Section IV describes a simplified PineSU workflow and illustrates the high-level architecture of our system.
- In Section V, we describe the data model adopted by PineSU to turn standard Git repositories into SUs using PineSU-specific hidden files. We also introduce a tree data structure (called *MerkleCalendar*) that efficiently tracks blockchain data into a PineSU-specific Git repository. It enables efficient calendar-based queries to retrieve SU fingerprints for a specific (registration) timestamp.
- Section VI provides more details on how PineSU interacts with the blockchain, what data are stored locally and what data are stored in blockchain blocks.
- In Section VII we give a more in-depth description of PineSU functionality.
- Section VIII draws some considerations about the level of protection of SUs, under increasingly restricting security hypotheses.
- Section IX describes a case study using PineSU to protect two clones of two real Git repositories owned by the Italian government and containing documents and data about the COVID-19 infection diffusion.
- Section X illustrates the results of an experimental and simulated performance evaluation of PineSU, expressed in terms of memory and time consumption, for storage units of varying sizes and structures.
- Section XI discusses the results of our research by illustrating a qualitative comparison of PineSU against existing state-of-the-art solutions.
- Section XII illustrates the main limitations of the proposed solution.
- Section XIII concludes the paper and suggests further development of our system.

II. PRELIMINARIES

In order to make the paper self-contained, we give the necessary background and terminology on distributed control version systems (Section II-A), Git in particular (Section II-B), integrity and authenticity protection (Section II-C), and blockchain technology (Section II-D).

A. VERSION CONTROL SYSTEMS

A *Version Control System* (*VCS* for short) is a software for sharing, recording and tracking changes to files/directories over time, enabling users to easily recover specific versions later. Resources (i.e., files and/or folders) under version control are organized into *repositories*, which are treated in isolation from one another. A *repository* is a directory in the local file system, called *working directory* or *working tree*, which includes specific metadata that are stored by the VCS software in hidden folders and files. Not all content of a working directory is also part of the corresponding repository, as users may specify resources that must be ignored. Also, whenever a new resource is created, the VCS software must be informed in order to include it into the repository (add command). VCSs can be roughly classified as *centralized* or *distributed*. In *centralized* VCSs, the version control engine runs in a single (central) server. Users manage their local repositories using lightweight clients, which need to interact with the corresponding servers to carry out almost all operations. Basically, clients have only a *snapshot* or a shallow copy of a specific version (typically the latest one) of the full repository stored in the server. Instead, in *distributed* VCSs, each client fully mirrors the repository, and has a complete versioned backup of all the data. Servers are involved to synchronize repositories among clients, i.e., they are the bridge that makes their cooperation possible, but most operations on a client's repository do not require any interaction with the corresponding server.

B. GIT

Git [14] is the most widespread distributed version control system. It models every repository as a *sequence of snapshots* (or *stream of snapshots*) of a miniature file system. Whenever a user saves the state of their project (through a commit operation), Git takes a snapshot of all files and folders under version control at that moment, and stores the snapshot in its local database; files that have been modified (since the last commit) are entirely included in the latest snapshot, while every unchanged file is not duplicated, and just a link to the previous identical copy is inserted in the snapshot. Every resource in a repository is internally identified by its checksum (SHA-1), which allows Git to efficiently detect changes in files, and makes any undetectable change very unlikely. Moreover, nearly all Git operations only add data to its database, including the file removal operation (rm command). This ensures that any change is reversible, with basically no risk of losing data permanently.

Every file in a working directory can be in one of two main states: *untracked* or *tracked*. An *untracked* file is a file that was never added to the repository or that has been removed from the set of tracked files (rm command). A newly created file is therefore in the untracked state. All the other files are *tracked* and can in turn be in one of three states: *unmodified* (or *committed*), *modified* and *staged*. A tracked file is *unmodified* (or *committed*) when it coincides with its latest version in the local database. If any change occurs, the state switches to *modified*. An untracked file or a (tracked) modified file gets *staged* if it is explicitly marked by the user to go into the next snapshot with the add command. More specifically, the add command adds or updates the target resource in the *staging area* (or *index*) of the repository, which is a special hidden file containing information about what will be included in the next snapshot. Basically, the staging area contains all currently tracked files, with a flag for each file that has been modified since the last snapshot. A *commit* operation (commit command) creates a new snapshot that incorporates all the changes specified in the staging area, and stores this snapshot in the local database. As a result, the staging area will be *cleaned*, i.e., flags indicating modified staged files will be removed, and all the previously (modified) staged files will get committed (i.e., unmodified).

Git users share information and collaborate with each other through remote repositories on servers, which can be synchronized with their own local repositories. Pushing and pulling data to and from remote servers are typical tasks when working on a shared project. The main commands to work with remote repositories are: clone, fetch, pull and push. The clone command creates a full-fledged working copy of the target repository, including a remote repository somewhere in Internet. The fetch command can be used to pull down all the resources from a remote project that are not yet in the (local) working directory, without modifying or merging the current local content with the downloaded data. While the pull command is similar to fetch, except that it also tries to automatically merge the downloaded data into the current local content. Finally, the push command uploads the local commit to the server, so that the local and remote projects are in sync. For further details about Git and its usage, we refer the reader to [15].

C. INTEGRITY AND AUTHENTICITY PROTECTION

Integrity is a pillar of information security with a broad meaning, it can be seen as the property that information has not been altered by unauthorized subjects or entities. A stronger property is the *authenticity*, which means that besides the integrity even the *origin* of specific information is guaranteed, where the *origin* is the subject or entity that either created or approved that information. Authenticity therefore implies integrity, but the converse is not true in general. Protecting integrity and authenticity is a fundamental requirement to make data and resources trustworthy. This is accomplished by implementing appropriate security mechanisms, which typically fall into two main classes: *proactive* (or *preventive*) mechanisms and *reactive* (or *detection*) mechanisms. *Proactive* mechanisms try to prevent violations through suitable access control strategies, i.e., their goal is to avoid that unauthorized subjects interfere with protected resources. Whereas *reactive* mechanisms came into play after an integrity breach has occurred, and they should restore the earlier information state. Regardless of the type and level of sophistication, all the reactive mechanisms are triggered only if an integrity violation is detected, making the detection of integrity attacks of paramount importance. Indeed, by integrity protection, it is often meant only the ability to detect integrity attacks.

This work mainly concentrates on blockchain-based security mechanisms to detect integrity attacks on Git repositories and to protect their authenticity. However, by leveraging the built-in features of Git, the recovery of the right state of corrupted repositories is also possible. We remark that the attacker could be the repository owner themselves when they no longer have the write privilege. We now give more technical concepts and tools on this subject that will be extensively used later on.

Let $m = m(t)$ be any digital resource whose *content* may vary over time; by *content*, we mean the low-level sequence of bytes encoding m. Assume that no authorized subject has modified m since a specific instant t_0. Throughout this paper, unless otherwise specified, by *integrity protection* we mean providing a strong and verifiable evidence that m has not been modified since t_0, i.e., $m(t_0) = m(t_1)$ where t_1 represents the current time. We will concentrate on protecting the integrity of stored information; namely, integrity protections for the transmission and processing phases are not considered. A common strategy to protect the integrity of m is to use a compact cryptographic redundancy c_m, of fixed length (no more than 512 bits), such that each bit of c_m depends on every bit of m unpredictably and with uniform probability. It follows that any change to m, even imperceptible, approximately affects half of the bits of c_m (*avalanche effect*). An attack on integrity can be easily detected by testing whether $c_{m(t_0)}$ and $c_{m(t_1)}$ differ, provided that some security assumptions hold. More specifically, there are two possible scenarios depending on whether $c_{m(t_0)}$ (differently from m) can be stored in a secure place, or not. In the first scenario, by the security assumption, an attacker cannot modify $c_{m(t_0)}$, thus they cannot bypass the integrity check, even if they know how to compute c_m for any given m. No cryptographic secret is needed to compute c_m in this case. While in the second scenario, an attacker can modify both m and $c_{m(t_0)}$, thus the redundancy c_m must also depend on a cryptographic secret not known to unauthorized subjects. This prevents an attacker from modifying m and then c_m consistently.

In this work, we consider the first scenario, where the secure place to store $c_{m(t_0)}$ is the blockchain. To produce c_m we use a *message digest* generated by a *cryptographic hash function* (SHA-256 [16]). A *cryptographic hash function* is a deterministic function $h : \{0, 1\}^* \rightarrow \{0, 1\}^b$ that takes in input a binary string m of any length, and returns a fixed-length output $h(m)$ of $b \geq 128$ bits; typical values of b are 128, 160, 224, 384, 256 and 512. The output $h(m)$ is called a *message digest* of m, which depends only on m and on the chosen hash function. Besides the avalanche effect, a cryptographic hash function must satisfy two fundamental properties known as *inversion resistance* (or *pre-image resistance*) and *collision resistance*. The *inversion resistance* means that, for a given hash h, it should be infeasible to find an input m such that $h = h(m)$. In practice, the only strategy for inverting a hash is a brute force search in the input space, which requires exponential time. For this reason, cryptographic hash functions are a particular type of *one-way* functions: they are easy to compute, yet hard to invert. Instead, *collision resistance* means that finding any two distinct inputs m_1 and m_2 such that $h(m_1) = h(m_2)$ should not be feasible; such a pair of inputs is called a *collision*.

The absence of collisions along with the acceptance of arbitrary length inputs make cryptographic hash functions one of the best ways to get *fingerprints* of digital resources, where a *fingerprint* is a compact sequence of bits that identifies a digital resource in the whole cyberspace. The integrity of a resource m can be protected by storing its fingerprint $h(m)$ in a secure storage place. But if we have a large set of resources $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ $(k \gg 1)$, storing k independent fingerprints may be memory inefficient and expensive. In this case, a convenient solution is to store only an *accumulator value* $a_{\mathcal{M}}$ of the whole set $\mathcal{M}$, using a suitable *cryptographic accumulator*. For the purpose of this paper, a *cryptographic accumulator* is a generalization of a cryptographic hash function, which has an arbitrarily large set of inputs, rather than just one. For each input m_i, it also provides a compact *witness* w_i (or *proof*) to verify whether m_i is a member of $\mathcal{M}$. More specifically, to verify whether $\mathcal{M}(t_0)$ coincides with $\mathcal{M}(t_1)$ it suffices to compare $a_{\mathcal{M}(t_0)}$ and $a_{\mathcal{M}(t_1)}$. Whereas, the membership test for a resource m_i can be efficiently performed by an algorithm that takes in input m_i, w_i and $a_{\mathcal{M}}$, without knowing the elements m_j with $j \neq i$; it is typically accomplished by first computing a binary string $a(m_i, w_i)$, and then by verifying the equality $a(m_i, w_i) = a_{\mathcal{M}}$. In particular, if $a(m_i(t_1), w_i(t_0)) = a_{\mathcal{M}(t_0)}$, then $m_i(t_1) = m_i(t_0) \in \mathcal{M}(t_0)$, which guarantees that the integrity of m_i has been preserved.

Cryptographic accumulators were formally introduced by Benaloh and de Mare [17], since then they have been employed in several computer security applications, including authenticity [18] and anonymity [19] protection and, recently, in improving the efficiency of blockchain technology [20], [21]; for further details we refer the reader to [22] and [23]. Many cryptographic accumulator schemes are based on the *Merkle tree* data structure [24], [25]; for a gentle introduction to this subject, we refer the reader to [26].

In PineSU, we use specific versions of Merkle trees that preserve a prescribed ordering of the elements of $\mathcal{M}$. For a given hash function $h(\cdot)$, and for an ordered set of digital resources $\mathcal{M} = (m_1, m_2, \ldots, m_k)$ $(k > 1)$, a *Merkle tree* $mkt(\mathcal{M})$ built on top of $\mathcal{M}$ is a complete binary tree with k leaves such that (*a*) the i-th leaf (in the left-to-right order) is associated with m_i and is labelled with $h(m_i)$; and (*b*) each non-leaf node n is labelled with a hash $h(n)$ obtained by suitably combining the hashes $h(n.l)$ and $h(n.r)$ of its left and right children $n.l$ and $n.r$, respectively; for example $h(n)$ can be the hash of the concatenation of the children's hashes: $h(n) = h(h(n.l)||h(n.r))$. The accumulator value $a_{\mathcal{M}}$ is the hash of the root node of $mkt(\mathcal{M})$. For any $1 \leq i \leq k$, the *witness* w_i of a resource m_i is its *authentication path* π_i, which basically consists of the siblings hashes of all nodes on the path from m_i to the root, along with a "left" or "right" label to specify whether a sibling hash represents the left or right subtree, respectively. More formally, let $\pi_i = h_{i1}h_{i2}\ldots h_{it}$ be the authentication path of m_i, where h_{ij} $(1 \leq j \leq t)$ is the hash of the j-th sibling in the path from m_i to the root. Also, let $s_i = b_{i1}b_{i2}\ldots b_{it}$ be the binary string encoding the left/right position of each sibling, i.e., $b_{ij} = 0$ (resp. $b_{ij} = 1$) if the j-th sibling is a left child (resp. right child). The witness w_i of m_i can be defined as the pair $\langle \pi_i, s_i \rangle$. It is easy to see that w_i requires $O(t) = O(log|\mathcal{M}|)$ space, and that the accumulator value $a_{\mathcal{M}}$ can be computed in $O(t) = O(log|\mathcal{M}|)$ time by knowing only w_i and m_i.

Concerning the authenticity protection, as already mentioned, it can be seen as a stronger form of integrity protection which also includes mechanisms to verify the origin of a given resource. This is typically achieved by using asymmetric encryption techniques, in particular digital signature algorithms. In this context, every subject (person or entity), say Bob, has a pair of related keys $\langle PR_B, PU_B \rangle$, known as *private* and *public keys* of Bob. The fundamental assumption is that only Bob knows his private key PR_B, whereas his public key PU_B, as the name suggests, can be known to everyone without exposing the private key. To authenticate a resource m, Bob has to digitally sign it, i.e., he encrypts m using his private key; we may write $[m]_B = E(PR_B, m)$ where $[m]_B$ is the digital signature of m signed by Bob, and $E(\cdot, \cdot)$ is the adopted asymmetric encryption function. Whoever knows Bob's public key PU_B, the resource m and the signature $[m]_B$, can verify the authenticity of m by testing whether the equality $m = D(PU_B, [m]_B)$ holds; $D(\cdot, \cdot)$ is the asymmetric decryption function associated with $E(\cdot, \cdot)$. Functions $E(\cdot, \cdot)$ and $D(\cdot, \cdot)$ are one the inverse of the other, provided that they use different keys of the Bob's pair, i.e., $D(PU_B, E(PR_B, m)) = m$. Due to computational reasons, asymmetric encryption algorithms pose serious restriction on the input length (at most a few thousand bits), it follows that fingerprints or cryptographic accumulators of digital resources are typically signed, rather than the resources themselves.

D. BLOCKCHAIN

Different types of blockchain technologies have been developed since the inception of the Bitcoin system [27]. In this work, we mainly refer to *public* and *permissionless* blockchain like Bitcoin [27], [28], [29] and Ethereum [30], [31], [32]. However, most concepts also apply to other types of blockchains.

Literally, a *blockchain* is a logical data structure consisting of a continuously-growing list of timestamped records, called *blocks*. It is physically stored in every node of a *Peer-to-Peer* (*P2P*) network, i.e., each node (or peer, or *miner*) maintains a whole replica, and no *special* or *central* nodes exist; namely, any two nodes are completely interchangeable. A distributed and decentralized cryptographic protocol ensures synchronization between peers and immutability of blocks. A *block* is a container of information that consists of a *header* and a list of *transactions*. A *transaction* typically describes an ownership transfer of some digital asset (e.g., cryptocurrencies). More in general, by modeling a blockchain as a *state transition system* [31], [32], a *transaction* represents a state transition from one consistent state to a new consistent state, according to a prescribed set of *validation rules*. There also exist mechanisms to include arbitrary data within transactions, with possible constraints on the maximum size. Indeed, there are subtle tricks to attach data to Bitcoin transactions [33], whereas Ethereum does provide an optional *data* attribute for this purpose [34].

Both blocks and transactions are uniquely identified by their cryptographic hash (i.e., by their fingerprint). Blocks can also be identified by their *height*—the integer representing the position in the blockchain, starting from zero—but in case of a temporary fork this does not guarantee uniqueness, as there could be two or more blocks with the same height. The block *header* includes metadata such as the reference to the previous block (i.e., its cryptographic hash), a timestamp, a *nonce*, and a Merkle tree root of all the transactions in the block. The block hash is actually the hash of its header, and it must be lower than a dynamically adjusted threshold, called *target hash*, to be accepted by the P2P network. Nonces have to be chosen so as to guarantee such constraint. Due to the inversion resistance of cryptographic hash functions, this requires a significant, but still feasible, computational effort, called *Proof-of-Work* (*PoW*), which must be remunerated to make the system sustainable. In particular, miners are remunerated in two different ways: with the PoW reward and with transactions fees. The first transaction of each block, called *coinbase transaction*, generates new coins equal to the PoW reward; through this transaction a miner assigns itself both the PoW reward and the fees of all transactions in the block.

Blocks can be considered *immutable*, because any modification, even imperceptible, propagates to all the subsequent blocks, invalidating the corresponding PoWs. Thus, an attacker should recompute all these PoWs and then convince most of the peers to update their database consistently. Transactions are created by software, called *clients* or *wallets*, which are typically controlled by human beings. Every wallet sends transactions to one or more nodes of the P2P network, which, in turn, broadcast them to the whole network. Statistically, once a transaction is stored in a block followed by six or more other blocks, it can be considered definitively valid by every peer.

Validation rules always include the authenticity of transactions; transactions must be digitally signed by the owners of digital assets being transferred, or in general, by those users who are authorized to trigger a specific state transition. However, protecting the authenticity is not enough to guarantee state consistency. Usually, several other validation rules have to be satisfied, some of them depend on the timing of operations, or briefly on the current state of the blockchain. Classical examples are the validation rules to prevent *double* *spending attacks*. In absence of such rules, a legitimate user can *spend* some of its assets at an instant t_0, and just after they can illegitimately *spend* again the same assets, possibly by specifying a fake instant $t_1 < t_0$. Transactions timestamps established by users' wallets are therefore not reliable. This is why blocks are timestamped and chained into a list: once a transaction has been included in a block, a reliable timestamp is that of the corresponding block.

One of the distinctive features of a blockchain is its *consensus mechanism*, which is the way nodes agree on a *single state* or *single truth*, that is on a single chain of blocks with no branches. In a fully decentralized blockchain, like Bitcoin and Ethereum, all nodes of the P2P network compete with each other to create the *next last block* B_n. As there are several thousands of nodes, B_n is created by a two-step procedure: (*S1*) first, one or very few nodes are randomly selected to propose their own B_n; (*S2*) then, the remaining nodes vote the proposed blocks, by expressing their acceptance or rejection. This process may lead to some (temporary) forks involving the ending part of the blockchain. But as soon as one branch gets longer than the others of six or more blocks, then all nodes consider this branch as the true one.

Bitcoin (and Ethereum until September 2022) uses a *Proof-of-Work* (*PoW*) *consensus protocol*, as it strongly relies on the effort for computing the PoW of a block. In particular, Step *S1* is accomplished by rejecting blocks with no or incorrect PoW, which is equivalent to randomly select a very limited number of nodes with uniform probability (i.e., those nodes that have solved the PoW for their block), assuming that all nodes have the same computational power. Concerning Step *S2*, a node working on a block B'_n accepts a block B_n proposed by some other node, by stopping working on B'_n, and by starting working on a new block B'_{n+1} which points to B_n as previous block. Unfortunately, a pure PoW consensus mechanism turns out to be very inefficient in terms of energy consumption, *throughput* (i.e., number of confirmed transactions per second) and *latency* (i.e., confirmation time of a transaction). In general, fully decentralized blockchains suffer from bottlenecks and scalability issues [35], therefore they might not be suitable for applications that require fast or immediate transactions, with low or even no fees. On September 2022, Ethereum switched from Proof-of-Work to *Proof-of-Stake* (*PoS*). *Proof-of-Stake* is a much more efficient consensus mechanism, where validators stake capital, which acts as a security deposit that can be lost if they behave dishonestly or lazily.

Another fundamental feature of a blockchain is the level of *programmability*, that is the possibility to modify existing validation rules, or to introduce new ones, through specific instructions written in a high level language. In a fully programmable blockchain like Ethereum, third-party developers can create decentralized programs that run on top of the blockchain; the P2P network can be viewed as a decentralized virtual machine for the execution of *decentralized applications* (or briefly *dapps*). These programs are called *smart contracts*, as they are typically used to create new digital assets and to manage their ownership. Currently, smart contracts in Ethereum can be implemented in Solidity [36] or Vyper [37]. Solidity is a Turing complete programming language, Vyper is not. Vyper has been introduced to increase the security of smart contracts by preventing bugs and vulnerabilities that are difficult to detect in Solidity.

Unlike the Ethereum platform, Bitcoin has very limited programmability. It does provide only a very simple stack-based scripting system, called *Script* [38], which is not Turing complete. In fact, Script is not intended to support general purpose smart contracts, but rather it provides enough flexibility to implement custom policies for spending Bitcoin.

III. RELATED WORK

Modern data-sharing solutions typically use cloud storage and computing services, which act as intermediaries and may have complete control of the shared information. It makes protecting integrity and confidentiality challenging [39], compared to on-premises solutions, especially when the cloud service may behave dishonestly or is not trusted.

In this section, we give an overview of the relevant literature and projects about how to share data in a trusted, secure, and verifiable manner. Our focus is on the protection of integrity and authenticity. We identified three main approaches, from the more general to the more specific and similar to our work.

A. OFFCHAIN

By *offchain* approaches, we mean those that are not based on a blockchain. They use specific cryptographic protocols to protect the shared data [40] and typically rely on *authenticated data structures* obtained by suitably combining existing data structures with specific cryptographic accumulators [41], [42]. These approaches assume that data sources (i.e., data owners) are trustworthy and never behaves maliciously, while there could be untrusted third-parties. A typical scenario involves a *data owner*, one or more third-party untrusted *publishers* (e.g., data replication servers), and one or more *clients*. Each client accesses the data by submitting a specific query to a publisher, which responds with an answer that includes a verifiable proof. To verify the authenticity of the answer, the client uses the proof and some digest value previously provided by the owner through a secure channel. Clients can reliably detect if a publisher is trying to fool them, and thus they can connect to another publisher. Moreover, if an adversary is unable to change the digest value (sent by the owner to the client), they cannot fool clients, even if they compromise publishers. There are several challenging aspects in the design of effective authenticated data structures. In particular, the verifiable proof should be succinct, efficient to compute and verify, and computationally infeasible to forge. Another critical issue is whether frequent updates by data owners can be efficiently performed, in this case, the authenticated data structure is said to be *dynamic*. Several papers have been devoted to provide various schemes of authenticated data structures. Martel et al. [41] developed a unified framework which generalizes many previously published schemes, by introducing a simple and abstract data model, called *Search DAG*. The Search DAG model makes it possible to turn existing data structures, including hybrid data structures like combinations of trees, arrays, and linked lists, into authenticated data structures. The model has been designed assuming fairly static data sets, however, for some underlying data structures, updates can be efficiently performed. Tamassia and Triandopoulos [43] introduce a new model for distributed data authentication over an untrusted P2P network, and present an efficient construction of a *Distributed Merkle Tree* (*DMT*), which realizes that model. Based on the DMT they realize an efficient *Authenticated Distributed Hash Table* (*ADHT*), which supports authenticated deletions. They also present a *Distributed Authenticated Dictionary*, a more general data authentication scheme, based on the ADHT. Papamanthou et al. [18] present two schemes that implement an efficient authenticated hash table based on cryptographic accumulators. The proposed schemes provide constant verification time, constant proof size and sublinear query or update time.

There is a substantial difference between the solution proposed in our work and the aforementioned papers. All these approaches assume that data owners are trustworthy and have a full control of what data can be written, updated and deleted (when deletion is supported). Instead, in our model, even data owners must not perform undetectable data modifications. This requirement can be guaranteed since blockchain blocks are immutable and timestamped. However, we cannot rule out that the blockchain can be replaced by a platform based on some append-only authenticated data structure, but currently this direction seems not yet viable [44].

B. BLOCKCHAIN-BASED

Several blockchain-based methods have been proposed for trustworthy data sharing [45]. Most of them assume that the actual data are maintained off-chain, whereas only pairs $\langle$**data-identifier, data-hash**$\rangle$ are stored into the blockchain [46] through a mapping in a smart contract. The validation consists of comparing the actual data hash against the hash on the blockchain. Some solutions combine the *InterPlanetary File System* (*IPFS*) [47] and blockchain smart contracts. For example: Nizamuddin et al. [48] propose a method to protect originality and authenticity of digital content published online, by storing such content in IPFS, and by tracking their publication history through an Ethereum smart contract; Khatal et al. [49] introduce a secure decentralized application framework for sharing files and data provenance, which uses IPFS as its data storage layer, and an Ethereum dapp for access control of digital content and storage of provenance data; Azizi et al. [50] provide a decentralized solution to store log data in IPFS, using an Ethereum dapp to protect the integrity. Some other solutions exploit the blockchain to securely share healthcare data produced by IoT devices and stored off-chain on a decentralized directory service [51] or in the cloud [52]. Blockchain-based approaches have also been extensively proposed for integrity auditing in cloud storage. The typical scenario is pretty the same as that mentioned for authenticated data structures: data owners do not trust cloud storage providers, so they need to protect the integrity of their data in the cloud. Often, the integrity auditing process relies on a (supposed trusted) third-party auditor to reduce the user's workload during the verification phase. For a comprehensive overview of integrity auditing schemes in cloud storage, we refer the reader to [53].

Unlike the aforementioned papers, our work focuses on securing Git repositories, irrespective whether they are stored in the cloud or on-premises. Most of the efforts are devoted to design a cost-effective solution based on a lightweight software, without relying on any third-party service, except a blockchain. Furthermore, all the off-chain data are stored in Git repositories, and no DBMS technology is used. Concerning IPFS, PineSU currently does not use it, and a possible adoption of IPFS depends on how compatible it is with Git's standard workflow.

C. COMBINED: DISTRIBUTED VCS AND BLOCKCHAIN

To the best of our knowledge, there are only two scientific articles [54], [55] that deal with blockchain and file version control. The first work [54] describes a decentralized document version control solution obtained by storing digital resources on IPFS, and by leveraging the security and decentralization of the Ethereum blockchain. The proposed solution has two primary actors: the *developers* and the *approvers*. Developers create and upload documents to IPFS, then they request for their approval, from the registered approvers, by providing the corresponding IPFS hashes. Suitable smart contracts manage the document version control functionality among creators and developers. In particular, they enable document version approval through a consensus by approvers.

The second work [55] presents the architecture of a decentralized version control system built on top of a private blockchain (specifically, Hyperledger Fabric[1]) and IPFS. The proposed architecture, named BDA-SCV, uses smart contracts to implement suitable access controls and, in particular, to manage the upload of documents. There are two types of users in BDA-CSV: *developers* and *authorizers*. The former are the classical users of any version control system. Whereas the latter are administrative users who provide permissions to developers, i.e., they are responsible for adding developers to the private blockchain. However, developers cannot add, edit, or delete code files.

The aim of these works significantly differs from ours, as we do not design a version control system from scratch, but we rely on Git. Furthermore, we are not aiming to achieve a decentralized consensus on the content of a

[1]https://www.hyperledger.org/projects/fabric

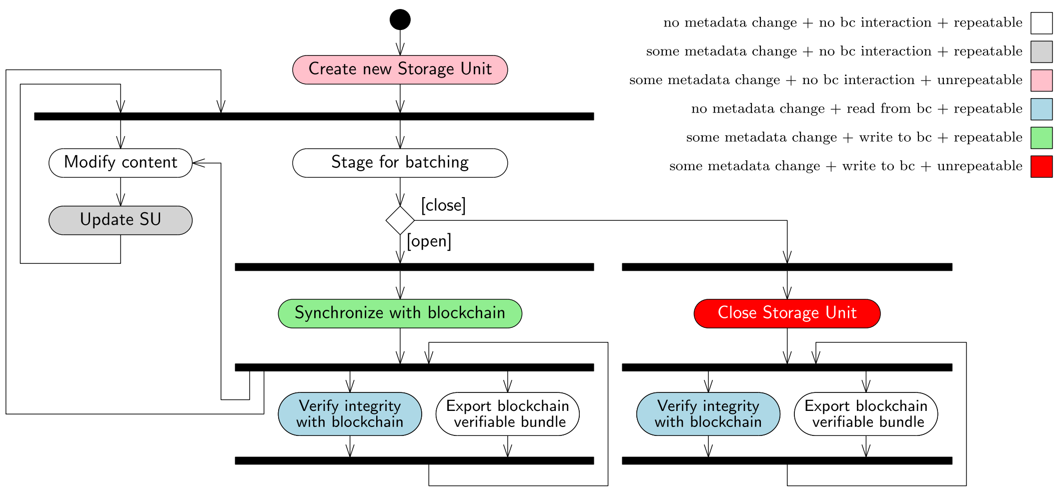

no metadata change + no bc interaction + repeatable
some metadata change + no bc interaction + repeatable
some metadata change + no bc interaction + unrepeatable
no metadata change + read from bc + repeatable
some metadata change + write to bc + repeatable
some metadata change + write to bc + unrepeatable
Create new Storage Unit
Modify content
Update SU
Stage for batching
[close]
[open]
Synchronize with blockchain
Close Storage Unit
Verify integrity with blockchain
Export blockchain verifiable bundle
Verify integrity with blockchain
Export blockchain verifiable bundle

FIGURE 2. Simplified workflow of PineSU illustrated through a UML activity diagram. Create new Storage Unit turns a local directory into an open storage unit, by first making it a Git repository, and then adding PineSU-specific hidden files that appropriately describe the current content of the directory. Update SU updates the SU hidden files to reflect the recent changes in the local directory, since the last updated state. Synchronize with blockchain stores in the blockchain the hash root of a suitable cryptographic accumulator of the whole SU content. Verify integrity with blockchain tests whether the SU is exactly the same as it was just after the last blockchain synchronization. Export blockchain verifiable bundle creates a compressed archive of user-selectable files/folders in the SU, along with a special descriptor file which enables a blockchain-based integrity verification of all the resources in the bundle. Close Storage Unit is similar to Synchronize with blockchain, but also prevents any undetectable change, thus any further modification to the SU will be considered invalid. Colors encode three distinct aspects of the previous commands: whether they change local metadata, whether they involve some interaction with the blockchain, and whether they are repeatable or not.

storage unit. Instead, we exploit the blockchain immutability and verifiability to provide decentralized authentication and timestamping for the state of a storage unit at a given date. It makes unauthorized modifications easily detectable, including retroactive alterations by storage unit owners.

Concerning existing projects aimed at combining Git and blockchain, we found only one by Cardstack,[2] named Gitchain [56], [57], which has important similarities to PineSU. Gitchain is a "chain-agnostic Layer 2 application state synchronization and syndication protocol based on Git" [58]. A group of digital resources to be shared with Gitchain are firstly bundled together into a Git repository. The bundle is then compressed into a single zip-like archive file, obtaining what is called a *packfile*. The packfile is then stored off-chain in some distributed storage system, such as a shared drive on the cloud, a distributed file system like IPFS, or another type of blockchain-driven storage system. Next, a link to the packfile and its hash are stored in some distributed ledger (e.g., on the Ethereum blockchain); whoever gets this link can find the packfile and recreate the original content.

Although PineSU and Gitchain share many aspects, a key difference is that our system exploits a suitable cryptographic accumulator to reduce the data stored on-chain, rather than compute the hash of a single compressed archive. Moreover, much of our work is devoted to define metadata, data structures, protocols and batching techniques that enable a cost-effective blockchain-based authenticity verification of Git repositories through a lightweight software.

[2]Cardstack - The Collaborative OS for Web3 https://cardstack.com/.

IV. PineSU WORKFLOW AND ARCHITECTURE

A simplified workflow of PineSU is depicted in Fig. 2, through a UML activity diagram. As already mentioned, PineSU organizes digital resources into *Storage Units* (*SUs*), where each SU consists of a Git repository along with additional metadata stored in hidden files within the repository itself. The first step of a typical workflow is the creation of a new SU (`create` command), which will be *open* by default. Subsequently, the corresponding (local) working directory can be modified as many times as necessary, but the SU metadata will not automatically update, i.e., they will no longer be consistent with the modified content. To get a new consistent state, the storage unit needs to be explicitly *updated* by running the `update` command. Before a blockchain registration, a storage unit must be *staged*, which adds it to the next group of SUs that will be registered, spending only one transaction (*batching*).

An open SU can be synchronized with the blockchain by creating a *Blockchain Synchronization Point* (*BPS* for short), which enables a blockchain-based integrity verification of the latest updated content. This is obtained by storing in the blockchain (`sync` command) the root hash of a suitable cryptographic accumulator (e.g., the root hash of a Merkle tree) of the SU content, relative to the time of the last update operation. Once an SU has been synchronized, its content can be checked against the blockchain (`checkbc` command); anyone who imports an SU clone can verify its integrity by leveraging the security of the blockchain. It is also possible to *export a bundle* of files with specific metadata (including the whole SU content), so that the integrity of

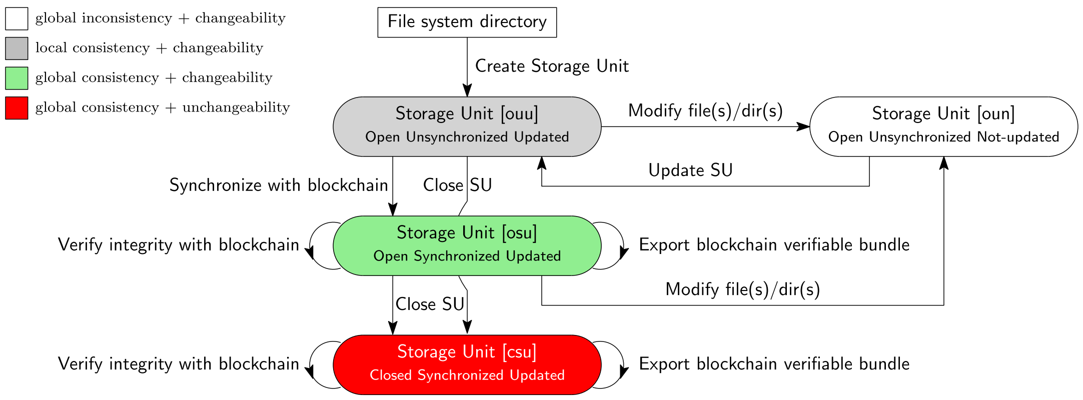

global inconsistency + changeability
local consistency + changeability
global consistency + changeability
global consistency + unchangeability
File system directory
Create Storage Unit
Storage Unit [ouu] Open Unsynchronized Updated
Modify file(s)/dir(s)
Storage Unit [oun] Open Unsynchronized Not-updated
Update SU
Synchronize with blockchain
Close SU
Verify integrity with blockchain
Storage Unit [osu] Open Synchronized Updated
Export blockchain verifiable bundle
Modify file(s)/dir(s)
Close SU
Verify integrity with blockchain
Storage Unit [csu] Closed Synchronized Updated
Export blockchain verifiable bundle

FIGURE 3. Simplified state diagram of a storage unit. Colors encode the degree of consistency of the SU and the possibility of modifying it or not. Consistency can be local or global. Global consistency means that both the content of the (local) descriptor files of PineSU and that of the blockchain reflect the content of the SU. Local consistency means that only the content of the local descriptor files reflects that of the SU. When there is no consistency at all, neither global nor local, there is global inconsistency.

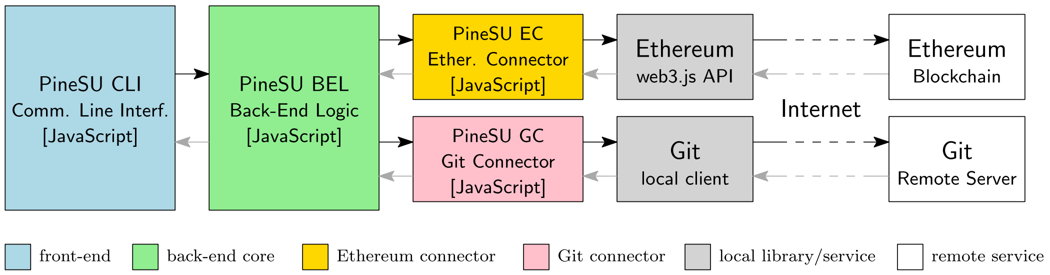


FIGURE 4. High level architecture of PineSU. Black arrows show messages that are triggered by the corresponding source entities, whereas gray arrows show passive reply messages. Software modules of PineSU are highlighted with distinct background colors. White and gray backgrounds indicate libraries or external (local or remote) services that PineSU relies on.

all the resources in the bundle can be verified with the blockchain. Finally, if one wants to definitively freeze an SU, invalidating any further change, it is possible to *close* it by creating a *Blockchain Closing Point* (*BCP* for short). PineSU prevents any change to a closed SU, but of course, it still allows for a blockchain-based integrity verification, as well as the export of verifiable bundles, exactly as it happens for an open synchronized SU. From now on, we will refer to both the Synchronize with blockchain and Close Storage Unit operations as *blockchain registration* operations, or simply *blockchain registrations*. We will see that there are two main approaches for implementing blockchain registrations, depending on the desired trade-off between security and costs.

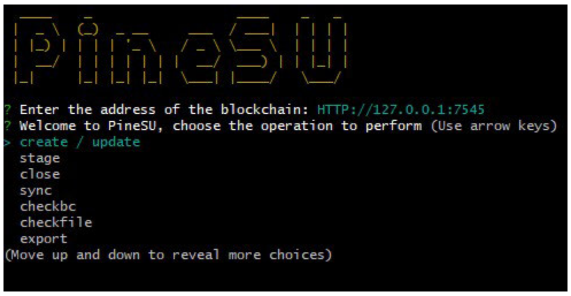
To better understand how PineSU works, a simplified state diagram of an SU is also shown in Fig. 3. The possible states of an SU are determined by three boolean state variables: open/closed, synchronized/unsynchronized and updated/non-updated. However, not all combinations are possible, in fact an SU can only be in one of the following four states: open unsynchronized updated (ouu for short), open unsynchronized not-updated (oun for short), open synchronized updated (osu for short) and closed synchronized updated (csu for short).

PineSU is a lightweight software system[3] that is built on top of a Git local client [14], [15] and of the Ethereum blockchain [31], [59], a high level architecture is shown in Fig. 4. The main architectural components are listed below.

- *PineSU CLI (Command Line Interface)*. Users interact with PineSU through a JavaScript terminal emulator,[4] similar to the original command line of Git. Besides the specific commands of PineSU, also basic Git commands are supported, so users do not need to interact with Git in a typical workflow.
- *PineSU BEL (Back-End Logic)*. This is the core component of PineSU. It manages all the storage units and controls the communication with the blockchain, with the local Git client and, indirectly, with remote Git servers. The back-end logic also takes care of batching blockchain registrations, in such a way that with a single blockchain transaction a whole set of SUs are registered, and not only one. Moreover, it maintains a temporal

[3]A first prototype implementation in JavaScript is publicly available at https://github.com/plspeziali/PineSU.

[4]Based on the Inquirer.js JavaScript library https://github.com/SBoudrias/Inquirer.js.



```
? Enter the address of the blockchain: HTTP://127.0.0.1:7545
? Welcome to PineSU, choose the operation to perform (Use arrow keys)
> create / update
  stage
  close
  sync
  checkbc
  checkfile
  export
(Move up and down to reveal more choices)
```

FIGURE 5. List of PineSU commands shown in the startup screen.

cryptographic accumulator, named *MerkleCalendar*,[5] to efficiently recover specific time-referenced cryptographic strings, for any given blockchain registration, which are necessary to verify the integrity and authenticity of an SU. The MerkleCalendar and its associated information are stored in a special Git repository whose access is restricted to PineSU.

- *PineSU EC (Ethereum Connector)*. PineSU is largely blockchain agnostic, i.e., it can be easily adapted to different blockchains, including blockchains with limited programmability. This is achieved by decoupling the back-end logic from the chosen blockchain, through a blockchain-specific connector. However, at the time of writing, only the connector for Ethereum has been developed by exploiting the web3.js library [60], [61].
- *PineSU GC (Git Connector)*. This component decouples the back-end logic from Git. It takes care of the interaction with the local Git client and, indirectly, with remote Git servers, by using the simple-git library [62].

V. PineSU DATA MODEL

In this section, we describe the data model adopted by PineSU to represent storage units and to support all the operations described in the basic workflow. The data model consists of three main entities: *Storage Unit*, *Storage Group* and *MerkleCalendar*. Below we give a detailed description of each of these entities.

A. STORAGE UNIT

As already said, PineSU logically wraps a Git working directory into a single SU. All the SU resources and other important properties are described in a JSON hidden file, named .pinesu.json, which is stored in the root folder of the working directory and tracked by Git itself; see also Fig. 6 for an illustration. Note that, there are other PineSU-specific hidden resources in the working directory, but they will be described later on.

The SU metadata in .pinesu.json consists of four main properties: `hash`, `header`, `filelist`, and `offhash`. The `hash` property provides the cryptographic fingerprint of the whole SU and some relevant metadata relative to the time of the last update operation. It suitably depends on the content of the `header` properties, which, in turn, depends on that of the `filelist` property. We now describe each of the aforementioned main properties.

1) `offhash`

The content of the `offhash` property, as the name suggests, does not affect the SU hash, since this property includes data that depend on the SU hash itself; so, the only way to break the unavoidable loop of dependencies is to make the SU hash independent of this property. More precisely, `bcregnumber` and `bcregtime` give, respectively, the total number of blockchain registrations for this SU, and the timestamp of the last registration. Whereas, the `closed` flag provides the current open/closed state of the SU. Observe that the values of all these properties depend on the type (synchronization or closing) and result of the last blockchain registration, which needs the SU hash to be performed.

2) `filelist`

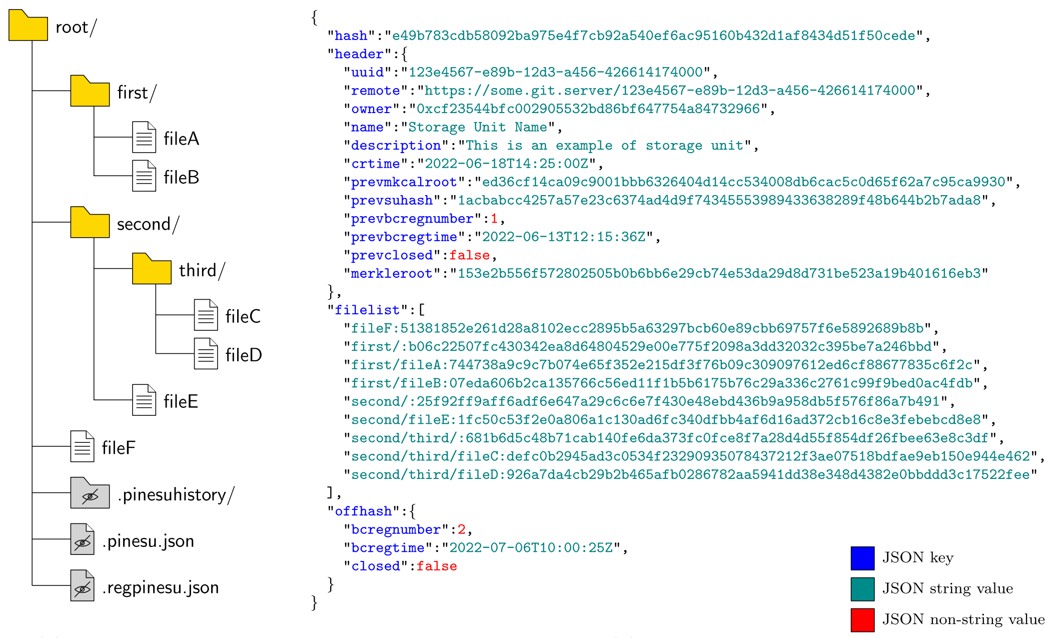
The `filelist` property describes the SU content through a list of strings in lexicographic order, where each item consists of the relative path of a file or folder to the working directory, a colon separator, and the corresponding (*one-way*) *fingerprint*. The *fingerprint* of a file `file` is the cryptographic hash (SHA-256) of the string consisting of the relative path of `file` followed by the cryptographic hash of the file itself (`hash(file)`) in hexadecimal format (see, e.g., Table 1). The fingerprint of an empty folder is the cryptographic hash of its relative path. While the fingerprint of a non-empty folder is the hash of its relative path followed by the fingerprints of its resources in lexicographic order (with respect to their relative paths).

3) `header`

The `header` property contains important SU attributes and other fundamental data that are needed to perform effective integrity checks. In particular, SUs are distinguished by their universal unique identifiers (`uuid` attribute) [63], and they are publicly available at unique URLs (`remote` attribute); the URL format consists of an https server name followed by the uuid string. Each SU has one owner (`owner` attribute) that is identified by a blockchain public address; in the current implementation, the public address of an externally owned account in Ethereum. It is also included a human-readable name, a description and a creation date (`crtime` attribute).

In order to verify the consistency of an SU history, as it will better clarified in the next sections, for each attribute `x` of the `offhash` property, there is the corresponding `prevx` attribute, of the `header` property, which tracks the value of `x` one step back in the past. These attributes are `prevbcregnumber`, `prevbcregtime` and `prevclosed`. For the same purpose, there are also two additional attributes `prevsuhash` and `prevmkcalroot`; namely, `prevsuhash` stores the previous SU hash, whereas `prevmkcalroot`

[5] A recent implementation in TypeScript is publicly available at https://github.com/plspeziali/merkle-calendar.



```
{
  "hash":"e49b783cdb58092ba975e4f7cb92a540ef6ac95160b432d1af8434d51f50cede",
  "header":{
    "uuid":"123e4567-e89b-12d3-a456-426614174000",
    "remote":"https://some.git.server/123e4567-e89b-12d3-a456-426614174000",
    "owner":"0xcf23544bfc002905532bd86bf647754a84732966",
    "name":"Storage Unit Name",
    "description":"This is an example of storage unit",
    "crtime":"2022-06-18T14:25:00Z",
    "prevmkcalroot":"ed36cf14ca09c9001bbb6326404d14cc534008db6cac5c0d65f62a7c95ca9930",
    "prevsuhash":"1acbabcc4257a57e23c6374ad4d9f7434555398943363828 9f48b644b2b7ada8",
    "prevbcregnumber":1,
    "prevbcregtime":"2022-06-13T12:15:36Z",
    "prevclosed":false,
    "merkleroot":"153e2b556f572802505b0b6bb6e29cb74e53da29d8d731be523a19b401616eb3"
  },
  "filelist":[
    "fileF:51381852e261d28a8102ecc2895b5a63297bcb60e89cbb69757f6e5892689b8b",
    "first/:b06c22507fc430342ea8d64804529e00e775f2098a3dd32032c395be7a246bbd",
    "first/fileA:744738a9c9c7b074e65f352e215df3f76b09c309097612ed6cf88677835c6f2c",
    "first/fileB:07eda606b2ca135766c56ed11f1b5b6175b76c29a336c2761c99f9bed0ac4fdb",
    "second/:25f92ff9aff6adf6e647a29c6c6e7f430e48ebd436b9a958db5f576f86a7b491",
    "second/fileE:1fc50c53f2e0a806a1c130ad6fc340dfbb4af6d16ad372cb16c8e3febebcd8e8",
    "second/third/:681b6d5c48b71cab140fe6da373fc0fce8f7a28d4d55f854df26fbee63e8c3df",
    "second/third/fileC:defc0b2945ad3c0534f23290935078437212f3ae07518bdfae9eb150e944e462",
    "second/third/fileD:926a7da4cb29b2b465afb0286782aa5941dd38e348d4382e0bbddd3c17522fee"
  ],
  "offhash":{
    "bcregnumber":2,
    "bcregtime":"2022-07-06T10:00:25Z",
    "closed":false
  }
}
```

(a) working directory (b) .pinesu.json

FIGURE 6. Illustration of a storage unit of PineSU. (a) Files and folders of the SU working directory that are tracked by Git. (b) A possible .pinesu.json hidden file, which is stored in the root folder and describes the SU properties and its content, excluding the PineSU-specific hidden resources. JSON keys, string values and non-string values have been colored in blue, teal and red, respectively.

TABLE 1. Illustration of the procedure for computing fingerprints of files and folders in an SU; the vertical bar symbol | represents the string concatenation operator.

`fp(file)`	=	`hash("rel-path-file"|hash(file))`
`fp(empty-folder)`	=	`hash("rel-path-folder/")`
`fp(non-empty-folder)`	=	`hash("rel-path-folder/"|fp(file1)|fp(file2)|...|fp(fileN))`

contains the previous MerkleCalendar root hash. The (non-hidden) working directory content is succinctly represented by the `merkleroot` attribute, which is the root hash of a complete binary Merkle tree whose leaves are the fingerprints of the SU resources. The left-to-right order of the leaves is given by the lexicographic order of their corresponding relative paths (see, e.g., Fig. 7). It is not hard to see that there is no ambiguity in the definition of this Merkle tree, and that its depth grows logarithmically with the number of resources in the working directory.

4) `hash`

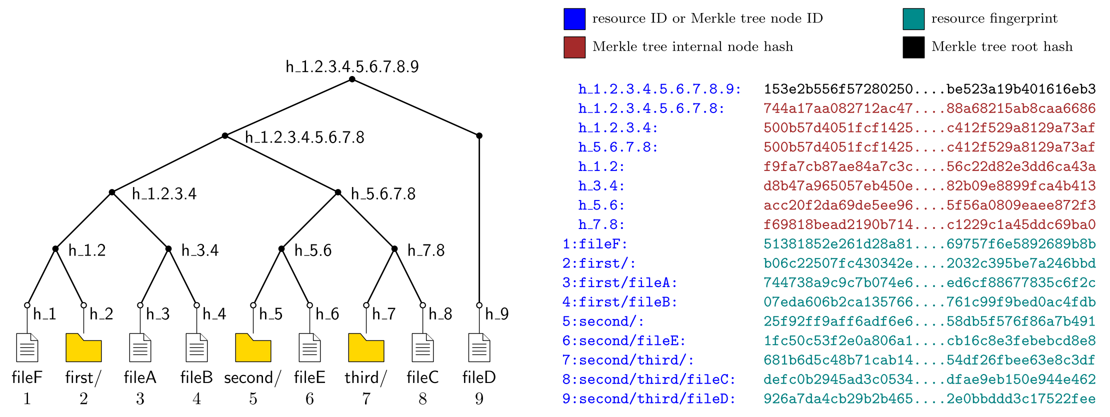
The `hash` property provides the SU (global) hash, which is the cryptographic hash of the `header` content, after removing (if any) line breaks, tab characters, and separating white spaces. We remark that the SU hash depends on its previous hash (through `prevsuhash`), on the previous MerkleCalendar root hash (through `prevmkcalroot`), on the working directory content (through `merkleroot`) at the time of the last update, but does not depend on the `offhash` content.

B. SU RESOURCES BUNDLE

PineSU enables users to export a blockchain verifiable bundle of a subset of SU resources. A bundle consists of a compressed archive, and the included resources must preserve the naming and the hierarchical structure of the SU they are extracted from. A special descriptor file .bundlepinesu.json is also added in place of .pinesu.json. It contains enough information for computing the Merkle root hash of the SU, even if some (or many) resources are not included in the bundle. The .bundlepinesu.json file is indeed obtained by copying the .pinesu.json file, and by replacing the ending name of every entry in the `filelist` array that is not in the bundle with a `$count` placeholder. Observe that if an SU contains a folder that is not present in the bundle, its name and that of its resources will not appear in the .bundlepinesu.json file, but only their fingerprints, which drastically reduces the exposition of information that a user does not want to share.

C. STORAGE GROUP

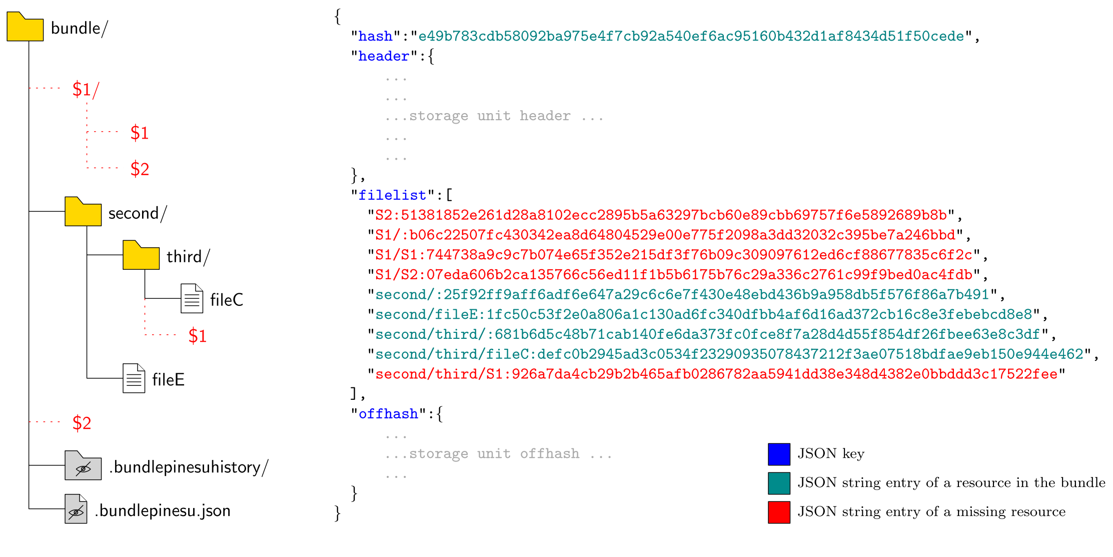
According to the basic workflow depicted in Fig. 2, PineSU provides a **Stage for batching** operation to make blockchain registrations cheaper. Indeed, when two or more SUs must be registered, they can be grouped into a single *Storage Group* so as only a single blockchain transaction is performed, rather than a number of transactions equal to the number of SUs; such a functionality is typically referred to as *batching* [21]. When operating on a single SU, a storage group with only



```
  h_1.2.3.4.5.6.7.8.9:   153e2b556f57280250....be523a19b401616eb3
  h_1.2.3.4.5.6.7.8:     744a17aa082712ac47....88a68215ab8caa6686
  h_1.2.3.4:             500b57d4051fcf1425....c412f529a8129a73af
  h_5.6.7.8:             500b57d4051fcf1425....c412f529a8129a73af
  h_1.2:                 f9fa7cb87ae84a7c3c....56c22d82e3dd6ca43a
  h_3.4:                 d8b47a965057eb450e....82b09e8899fca4b413
  h_5.6:                 acc20f2da69de5ee96....5f56a0809eaee872f3
  h_7.8:                 f69818bead2190b714....c1229c1a45ddc69ba0
1:fileF:                 51381852e261d28a81....69757f6e5892689b8b
2:first/:                b06c22507fc430342e....2032c395be7a246bbd
3:first/fileA:           744738a9c9c7b074e6....ed6cf88677835c6f2c
4:first/fileB:           07eda606b2ca135766....761c99f9bed0ac4fdb
5:second/:               25f92ff9aff6adf6e6....58db5f576f86a7b491
6:second/fileE:          1fc50c53f2e0a806a1....cb16c8e3febebcd8e8
7:second/third/:         681b6d5c48b71cab14....54df26fbee63e8c3df
8:second/third/fileC:    defc0b2945ad3c0534....dfae9eb150e944e462
9:second/third/fileD:    926a7da4cb29b2b465....2e0bbddd3c17522fee
```

(a) node-link diagram (b) hashes

FIGURE 7. **Complete binary Merkle tree of a working directory with nine resources. (a) Node-link diagram with leaves that are aligned from left to right based on the lexicographic order of their relative paths (for space reasons, only the final name of each resource has been inserted). (b) Fingerprints of files and folders in the working directory (colored teal), hashes of non-leaf nodes (colored brown), and Merkle tree root hash (colored black).**



```
{
  "hash":"e49b783cdb58092ba975e4f7cb92a540ef6ac95160b432d1af8434d51f50cede",
  "header":{
      ...
      ...
      ...storage unit header ...
      ...
      ...
  },
  "filelist":[
    "S2:51381852e261d28a8102ecc2895b5a63297bcb60e89cbb69757f6e5892689b8b",
    "S1/:b06c22507fc430342ea8d64804529e00e775f2098a3dd32032c395be7a246bbd",
    "S1/S1:744738a9c9c7b074e65f352e215df3f76b09c309097612ed6cf88677835c6f2c",
    "S1/S2:07eda606b2ca135766c56ed11f1b5b6175b76c29a336c2761c99f9bed0ac4fdb",
    "second/:25f92ff9aff6adf6e647a29c6c6e7f430e48ebd436b9a958db5f576f86a7b491",
    "second/fileE:1fc50c53f2e0a806a1c130ad6fc340dfbb4af6d16ad372cb16c8e3febebcd8e8",
    "second/third/:681b6d5c48b71cab140fe6da373fc0fce8f7a28d4d55f854df26fbee63e8c3df",
    "second/third/fileC:defc0b2945ad3c0534f232909350784372 12f3ae07518bdfae9eb150e944e462",
    "second/third/S1:926a7da4cb29b2b465afb0286782aa5941dd38e348d4382e0bbddd3c17522fee"
  ],
  "offhash":{
      ...
      ...storage unit offhash ...
      ...
  }
}
```

(a) SU bundle (b) .bundlepinesu.json

FIGURE 8. **An example of a storage unit bundle. (a) Missing SU resources are represented with no icon and with a red** `$count` **placeholder. (b) Illustration of the corresponding .bundlepinesu.json hidden file included in the bundle; entries of resources in the bundle are colored in teal, whereas those of missing resources are in red.**

one entry is created. More formally, a *Storage Group* (*SG*) of size n is a collection of pairs $\{\langle uuid(SU_i), hash(SU_i)\rangle : i \in [1, n]\}$, where ($a$) each pair $\langle uuid(SU_i), hash(SU_i)\rangle$ gives, respectively, the UUID and the hash of the i-th storage unit SU_i in the group, and (b) there is no repetition of UUIDs. The *hash sequence* of a storage group is the ordered set of hashes $(h_i)_{i\in[1,n]}$, where $h_i = hash(uuid(SU_i)|hash(SU_i))$ and the hashes are considered in lexicographic order of their corresponding UUIDs; namely, the index is chosen in such a way that $i < j$ if and only if the string $uuid(SU_i)$ precedes $uuid(SU_j)$, in lexicographic order. A complete binary Merkle tree can be built on top of the SG hash sequence. The *storage group hash* is the root hash of this Merkle tree.

As it will be better clarified later, before performing a blockchain registration, PineSU distinguishes storage groups into *Open Storage Group* (*OSG*) and *Closed Storage Group* (*CSG*). The former contains only open SUs, whereas the latter contains only awaiting closing SUs. In other words, an open (resp. closed) storage group is used when a set of open (resp. awaiting closing) SUs need to be synchronized with the blockchain (resp. closed) by performing only a single transaction.

From the user perspective, PineSU provides a **stage**[6] command to add open SUs to a forming storage group. According to the type of the subsequent registration operation, this storage group will be an OSG, a CSG, or it can be partitioned into two subgroups: one open and the other closed.

[6]We used the term "stage" because this operation resembles Git's stage command, even if it is not based on it.

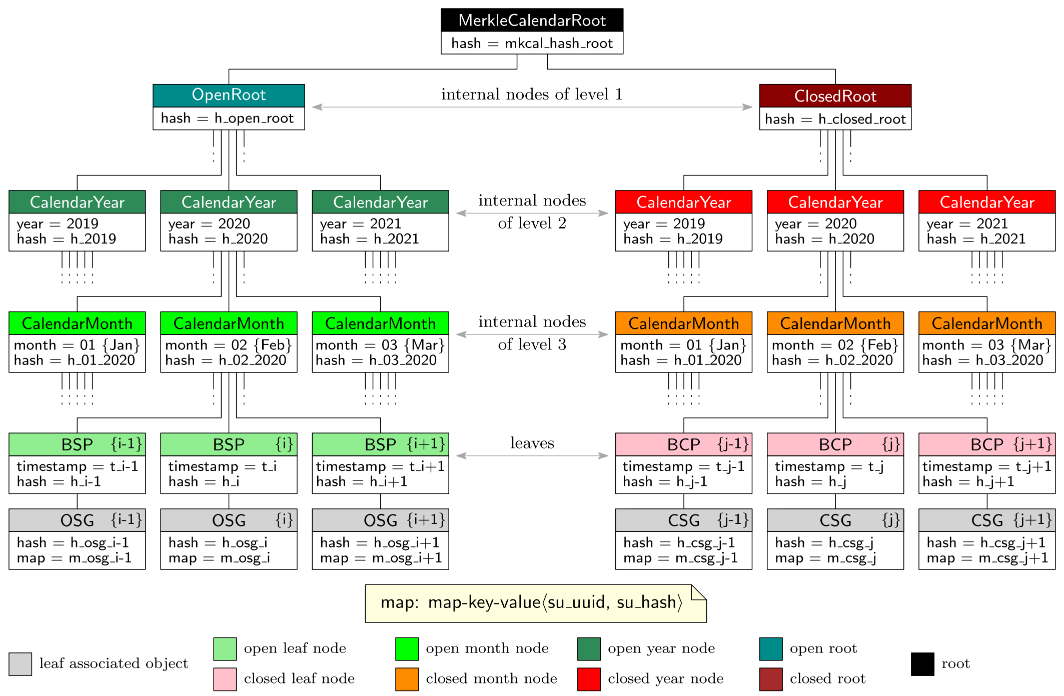


FIGURE 9. Schematic illustration of a MerkleCalendar. The left subtree tracks all the synchronization operations on open storage groups. Each of its leaves represents a BSP, including the timestamp in which the synchronization operation started, and it is associated with the corresponding OSG. Similarly, the right subtree tracks all the closing operations on closed storage groups. The parent of a leaf with a timestamp `YYYY-MM-DDThh-mm-ssZ` represents the corresponding calendar month `MM`, while the grandparent node represents the corresponding calendar year `YYYY`. Children of nodes of level 1, 2 and 3 are left-to-right ordered by increasing years, months and timestamps, respectively. Objects associated with leaves are highlighted in gray, while open and closed internal nodes are highlighted in green and red tones, respectively. Colors get darker moving from bottom to top; the root node is highlighted in black.

TABLE 2. Schematic illustration of how the MerkleCalendar root hash is computed.

`mkcal_hash_root`	`= r.hash = hash(or.hash|cr.hash) = hash(h_open_root|h_closed_root)`
`h_open_root`	`= or.hash = root(bmkt(or.childhash[]))`
`h_closed_root`	`= cr.hash = root(bmkt(cr.childhash[]))`
`h_YYYY`	`= cy_YYYY.hash = root(bmkt(cy_YYYY.childhash[]))`
`h_MM_YYYY`	`= cm_MM_YYYY.hash = root(bmkt(cm_MM_YYYY.childhash[]))`
`h_i`	`= bsp_i.hash = hash(bsp_i.timestamp|osg(bsp_i).hash) = hash(t_i|h_osg_i)`
`h_j`	`= bcp_j.hash = hash(bcp_j.timestamp|csg(bcp_j).hash) = hash(t_j|h_csg_j)`

D. MERKLECALENDAR

A *MerkleCalendar* is a tree data structure that is stored in a special Git repository of PineSU (reserved only to it), which implements a particular cryptographic accumulator having the highest level of hierarchy in PineSU. Its leaves are in bijection with the storage groups, and whenever PineSU writes some SU-related hash string into the blockchain, it actually writes the MerkleCalendar root hash. The general aim of a MerkleCalendar is to efficiently track and recover its root hash and some other hash strings associated with SGs, for a given date or timestamp. In other words, it has been designed to quickly perform temporal queries like "*What was the MerkleCalendar root hash when the BSP of the 5th August 2021 was created?*". Therefore, in perfect accordance with the Git's philosophy, PineSU does not need to interact with nodes in the blockchain P2P network (or with a server on Internet connected to some of them) to recover a hash string written into the blockchain in a specific day in the past.

From a structural point of view, a *MerkleCalendar* is a tree with depth four that is defined as follows (a schematic illustration is shown in Fig. 9). It has exactly two internal nodes of level 1: the *OpenRoot* and the *ClosedRoot*. The former is the root of the subtree whose leaves are in bijection with the open storage groups, while the latter is the root of the subtree associated with the closed storage groups; we will refer to these subtrees as the *open subtree* and the *closed subtree*, respectively. An internal node of level 2 represents a *calendar year*, while an internal node of level 3 represents a *calendar month*. Every leaf in the open subtree describes a BSP created in the month and year corresponding to its parent and grandparent nodes, respectively. Similarly, leaves in the closed subtree represent BCPs added in months and years corresponding to their ancestor nodes. Of course, a specific month/year node is created if and only if at least one blockchain-related operation occurred in that period.

Every node of the MerkleCalendar has the own hash attribute that is computed based on the hashes of its children.

Leaf nodes also include a timestamp attribute that precisely describes the instant of time in which the corresponding BSP (or BCP) operation was initiated. Children nodes are left-to-right ordered based on the level they belong to. More specifically, the open root is to the left of the closed root, while the children of a node of level 1, 2 and 3 are ordered by increasing years, months and timestamps, respectively. It follows that the open and closed subtrees grow incrementally on their "right side", leaving unchanged what is on the left. Therefore, it is easy to identify the subtree of the current MerkleCalendar which was the MerkleCalendar of some past day, as well as to recover the corresponding hash root. We now give more details on how the MerkleCalendar hash root `mkcal_hash_root` is computed. We follow a top-down procedure focusing on the open subtree, the description for the closed subtree is substantially analogous; a schematic illustration is given in Table 2.

The hash `mkcal_hash_root` of the root node is the cryptographic hash of the concatenation of its children hashes. Let `or` denote the open root node, and let `or.childhash[]` be the sequence of hashes induced by its child nodes. The hash `or.hash = h_open_root` is the root hash of the complete binary Merkle tree that is built on top of `or.childhash[]`; more formally, we write `or.hash = root(bmkt(or.childhash[]))`. Similarly, if `cr` denotes the closed root node, the corresponding hash is `cr.hash = root(bmkt(cr.childhash[]))`. The hash `h_YYYY` of a generic calendar year node `cy_YYYY` (the name has been chosen so that `cy_YYYY.year = YYYY`) is `cr_YYYY.hash = root(bmkt(cr_YYYY.childhash[]))`. While the hash `h_MM_YYYY` of a calendar month node `cm_MM_YYYY` is given by `root(bmkt(cm_MM_YYYY.childhash[]))`; note that the name convention follows that adopted in Fig. 9, i.e., `parent(cm_MM_YYYY).year = YYYY` and `cm_MM_YYYY.month = MM`. Finally, if `bsp_i` denotes the leaf node representing the i-th blockchain synchronization point, then `bsp_i.hash = h_i` is defined as the cryptographic hash of the concatenation of its timestamp `bsp_i.timestamp = t_i` with the hash of the associated open storage group `osg(bsp_i).hash = h_osg_i`.

Each BSP (respectively, BCP) node is associated with an object representing the corresponding OSG (respectively, CSG). This object contains the storage group hash (`hash` attribute), and a map (`map` attribute) that stores key/value pairs ⟨`uuid, hash`⟩ for all the SUs in that storage group. Keeping track of the hash of each SU in a storage group is indeed fundamental to verify the integrity of any SU in that group.

In the next section we will see that each BSP/BCP node is also associated with another object containing specific information about the corresponding blockchain transaction and its block, such as the block number, the block hash, the block timestamp, the transaction hash, the transaction addresses, the transaction value, and the transaction data.

VI. PineSU INTERACTION WITH THE BLOCKCHAIN

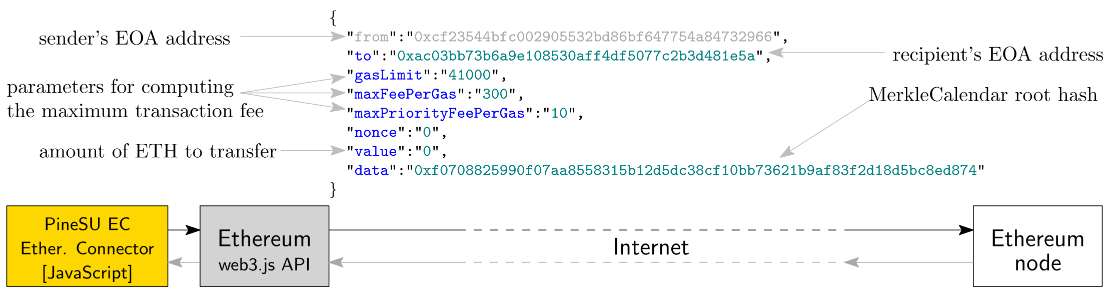
In this section, we describe how PineSU interacts with the blockchain, and what information stores to efficiently recover previous registration data. We preliminarily observe that two main interaction modalities are possible, each with pros and cons. The first one, called *detection mode*, does not make use of smart contracts, but just regular transactions. Namely, the MerkleCalendar root hash is included within the transaction data, so eventually this hash will be written in some block. This modality is easier to manage, is largely blockchain-agnostic, and is also suitable for non-programmable blockchains. On the other hand, at blockchain level, it cannot prevent writing operations that are forbidden by PineSU, such as closing an already closed storage unit. However, such forbidden operations can be detected by PineSU through specific validation checks. The other interaction modality is the *prevention mode*. It requires a specific smart contract to implement the blockchain side of PineSU, which includes an access control logic to verify whether a blockchain interaction, in particular a write operation, can be performed or has to be denied. The prevention mode guarantees a higher level of security, but it is more difficult to manage, is more expensive, and is much more coupled with the adopted programmable blockchain. The current implementation of PineSU is based on the detection mode, which is described hereafter, whereas the prevention mode is left as a future development.

A. DETECTION MODE

Let Bob denote a PineSU user who wants to perform a blockchain registration (synchronization or closing) for a group of his storage units. The registration request consists of performing a regular Ethereum transaction that includes the MerkleCalendar root hash in its `data` field (see, e.g., Fig. 10). The sender and the recipient of the submitted transaction will be the addresses of two externally owned accounts that are both controlled by Bob. In particular, the sender's address must coincide with the value of the `owner` attribute in the **.pinesu.json** file of every storage unit being registered, since only the owner has the right of doing this registration. The amount of ETH to transfer (`value` field) will be set to zero, as the actual aim of the transaction is to store data in the blockchain, and not transferring cryptocurrency. The values of the other transaction fields can be chosen by Bob to get a good compromise between validation time and cost.

Once the transaction has been validated, the corresponding *Blockchain Registration Data* (*BRD*), like the block hash and its timestamp, are suitably associated with the corresponding BSP/BCP node in the MerkleCalendar (see, e.g., Fig. 11a), even if they are not considered in the procedure for computing the root hash.

To enable a blockchain-based integrity verification to non-owner users, a storage unit has to be equipped with additional metadata extracted from the corresponding MerkleCalendar. These metadata are stored in the **.regpinesu.json** hidden file



```
{
  "from":"0xcf23544bfc002905532bd86bf647754a84732966",
  "to":"0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  "gasLimit":"41000",
  "maxFeePerGas":"300",
  "maxPriorityFeePerGas":"10",
  "nonce":"0",
  "value":"0",
  "data":"0xf0708825990f07aa8558315b12d5dc38cf10bb73621b9af83f2d18d5bc8ed874"
}
```

sender's EOA address
recipient's EOA address
parameters for computing the maximum transaction fee
MerkleCalendar root hash
amount of ETH to transfer
PineSU EC Ether. Connector [JavaScript]
Ethereum web3.js API
Internet
Ethereum node

FIGURE 10. Illustration of a regular ETH transaction triggered by a registration operation of PineSU, through its Ethereum connector. All the SUs in the storage group being registered must have the same owner in their .pinesu.json file, which must coincide with the sender's EOA address (`from` attribute). The EOA addresses of the sender and the recipient are both controlled by the same person. The `from` key and its value are colored gray, since they are not explicitly included in the transaction data, but they can be obtained indirectly; namely, every transaction must be signed by the sender, who has to be identified to verify the signature.

(see, e.g., Fig. 11b), and contain a minimal amount of information that makes it possible to verify the integrity against the blockchain, without knowing the whole MerkleCalendar. For each SU of the registered storage group, PineSU writes the corresponding .regpinesu.json file in the root folder of that SU. The content of this file is briefly reported hereafter. The `type` attribute specifies whether the registration is a synchronization or a closing operation. Attributes `mkcalroot` and `mkcaltimestamp` store, respectively, the root hash of the MerkleCalendar, and the timestamp of the BSP/BCP node associated with this operation. The blockchain registration data—the transaction hash, the block hash, the block height, and the block timestamp—are stored in the attributes `txhash`, `bkhash`, `bkheight`, and `bktimestamp`. Of course, the timestamp indicated by `mkcaltimestamp` must strictly precede that of `bktimestamp`. The `witness` and `openstoragegroup` (resp. `closedstoragegroup`) attributes allow users to compute the MerkleCalendar root hash of the storage unit, even if they do not know the whole MerkleCalendar nor the other SUs in the registered storage group. This is accomplished by using only the hashes of all the SUs in the storage group, and the hashes of the BSP node (resp. BCP node) ancestors' siblings along the path to the MerkleCalendar root. In particular, the `closedroot` attribute (resp. `openroot` attribute) contains the hash of the ClosedRoot node (resp. OpenRoot node). Let *BSP** (resp. *BCP**) be the leaf node of the MerkleCalendar that is associated with the registered OSG (resp. CSG), and let CM^* and CY^* be the two ancestors of *BSP** (resp. *BCP**) of level three and two, respectively. The `years` attribute stores the sequence of hashes of all the CalendarYear nodes that strictly precede their sibling CY^*. The `months` attribute stores the hashes of CalendarMonth nodes that are children of CY^* and strictly precede their sibling CM^*. Similarly, the `syncpoints` attribute (resp. `closingpoints` attribute) stores the hashes of BSP nodes (resp. BCP nodes) that are children of CM^* and strictly precede their sibling *BSP** (resp. *BCP**). Finally, the `openstoragegroup` (resp. `closedstoragegroup`) array contains the set of pairs ⟨`uuid`, `hash` ⟩, for each storage unit in the storage group.

Assuming that the number of SUs in a same storage group, the number of registration operations per month, and the number of years in the MerkleCalendar are all bounded by a relative small constant, then the total length of the .regpinesu.json file can be considered small as well, and suitable for many applications.

VII. BASIC PineSU FUNCTIONALITY

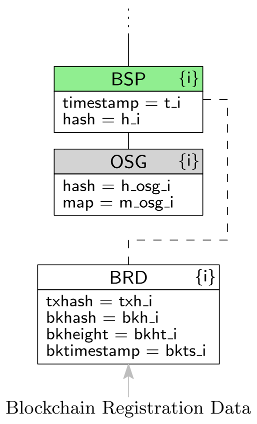
In this section, we give some implementation details of the basic PineSU's functionality and how the system interacts with Git.

A. CREATE NEW STORAGE UNIT

By interacting with the command line interface of PineSU (create command), the user can turn the current working directory into a storage unit. This implicitly creates the corresponding Git repository, if it does not exist yet. PineSU prompts the user to enter the fundamental SU properties that it cannot deduce from itself, such as the name, the URL, and the address of the owner's blockchain account. Other non-fundamental properties are left empty or get some defaults, for example the closed flag is false by default. Optionally, specific files that must be excluded from the SU (and from its Git repository) can be indicated through a standard .gitignore file. To get the SU hash, PineSU computes the fingerprint of every resource in the working directory, then builds an ephemeral Merkle tree, as described in Subsection V-A. If the command is successfully executed, a .pinesu.json descriptor file is created and stored in the root folder, after which a Git commit is done. In case this descriptor file already exists, PineSU deduces that the working directory is already a storage unit. Therefore, if the closed flag is false, PineSU asks the user whether they want to update the SU (see next functionality), otherwise displays an error message, since any operation that may change the SU content is not allowed.

B. UPDATE STORAGE UNIT

This functionality applies only to open SUs, and is required to update the list of files/folders and to recompute the SU hash, whenever some modification occurred, including the creation



(a) BRD

```
{
  "type":"synchronization",
  "mkcalroot":"e79ec5027cd1b51b5b8e41e71d2f7bf4db14fa6bb20587c0d6d276d59bbe1117",
  "mkcaltimestamp":"2022-07-06T09:45:35Z",
  "txhash":"0xc563030328e652d427cd00707d7a0e2ce0bcf6c76b23482469eb497e2dc87d2e",
  "bkhash":"0x041179fa859c5baf3f44c33a707de838b6fb9acb450ffe8b86713515032d73c9",
  "bkheight":15088235,
  "bktimestamp":"2022-07-06T10:00:25Z",
  "witness":{
    "closedroot":"15b893950543da062c810480ed4f328b634af8b6b9803626f087bfbc98498f83",
    "years":[
      "023e33504ab909cf87a6f4e4e545090e40bdc0a2153e5b68b19f7fad2b737904",
      .................................................................,
    ],
    "months":[
      "8e8ba305c0dd3880b69fea846d79dce90153d020fa42c885200f67f5f6c912fa",
      .................................................................,
    ],
    "syncpoints":[
      "4ebbd3fc4913904e16dc16560ba11f6b7215865cb7fedc4487a76dab90332d83",
      .................................................................,
    ]
  },
  "openstoragegroup":[
    {
      "uuid":"123e4567-e89b-12d3-a456-426614174000",
      "hash":"bf9a67b632d438264f2fb0b2735fb0a88ffe33ca324cf6b3676a121fec4ac2db",
    },
    .................................................................,
  ]
}
```

(b) .regpinesu.json

FIGURE 11. Illustration of how the blockchain registration data are stored in PineSU, in case of a synchronization operation. (a) A BRD element is associated with the corresponding BSP node in the MerkleCalendar. (b) Blockchain registration metadata for a specific SU in the synchronized OSG; the metadata are stored in the .regpinesu.json hidden file (in the root folder of the SU) which is tracked by Git.

or removal of resources. Eventually, the .pinesu.json file is updated consistently, and a Git commit is done.

C. STAGE FOR BATCHING

Similarly to Git, PineSU has the own staging area to indicate which open SUs will form the next storage group, for the next blockchain registration. By entering the command stage from the root folder of an open SU, a reference to such SU will be added to the forming storage group; from that moment on, we say that the SU is *staged*. At the low level, PineSU does not distinguish storage groups into open or closed, this distinction is made at the time of the subsequent blockchain registration operation. An SU is staged if a reference to it is included in a temporary file reserved to PineSU, which is not tracked by Git, however. To *unstage* an SU, it suffices to remove its reference from this file, which can be easily done through the CLI of PineSU. We remark that only open SUs can be staged; if the closed flag is true, then the SU cannot be staged. We also remark that the staging area is automatically emptied immediately after a blockchain registration (synchronization or closing) is executed.

D. SYNCHRONIZATION WITH BLOCKCHAIN

By entering the sync command into the PineSU CLI, the staged SUs area synchronized with the blockchain. If the staging area is empty, the command is ignored. This operation is one of the most involved, since it is made up of several steps, besides interacting with the blockchain. We will describe the current implementation, which has been tested on a local blockchain testnet (Ganache [64]) and on the Ethereum Mainnet. However, we will also briefly discuss possible modifications for a faster and more effective interaction with the Ethereum Mainnet. The main steps that PineSU carries out are:

(i) a preliminary check is executed to verify whether all the staged SUs have the same owner, but distinct UUID; in the negative case, the synchronization is aborted and an error message is shown to the user;
(ii) an ephemeral Merkle tree is built on top of the staged SUs, as described in Section V-C, and a new OSG is created;
(iii) a new BSP is created and added to the MerkleCalendar, if necessary even a new CalendarMonth and a new CalendarYear are preliminarily created and added to the open subtree;
(iv) the MerkleCalendar root hash is updated;
(v) a regular ETH transaction is submitted to a blockchain node, attaching the MerkleCalendar root hash in the data field, as described in Subsection VI-A (the other transaction fields are set to default values, even if the user can easily overwrite each of them through the CLI of PineSU);
(vi) PineSU waits for the validation of the transaction, in the negative case, the user can either retry the transaction or abort the synchronization, emptying the staging area and restoring the MerkleCalendar state before step (iii);
(vii) once the transaction has been validated, a BRD element is created and associated with the corresponding BSP node in the MerkleCalendar;
(viii) for each SU in the OSG, a .regpinesu.json descriptor file is written in the root folder, overwriting the

previous file, if any, and a copy is also stored in the **.pinesureghistory/** hidden folder after being renamed as `.regpinesu-YYYY-MM-DDThh-mm-ssZ.json`, where `YYYY-MM-DDThh-mm-ssZ` is the block timestamp;

(ix) for each SU in the OSG, the `offhash` property in the **.pinesu.json** file is updated, and a copy of this file is stored in the **.pinesureghistory/** folder, with a different name like `.pinesu-YYYY-MM-DDThh-mm-ssZ.json`;

(x) for each SU in the OSG, the `header` property of the **.pinesu.json** file is consistently updated (i.e., all the `prevX` fields are updated), the global storage unit hash is computed, and the result is assigned to the `hash` property.

(xi) All the changes in the local Git repository containing the MerkleCalendar are committed.

In our proof-of-concept implementation, step (vi) is executed quickly, but it may take a while when interacting with the Ethereum Mainnet. At the time of writing, a transaction confirmation takes from fifteen seconds to five minutes, on average, depending on the amount of offered tip per gas and on the network congestion. Of course, the longer is the confirmation time, the more is the probability of losing the connection to the blockchain node. A resilient implementation should tolerate even long transaction validation times. This can be done by introducing a pending state, where PineSU enters just after the execution of step (v). To avoid introducing inconsistencies, as long as PineSU is in this pending state, no other blockchain registration operations can be executed. In the meanwhile, with a prescribed frequency and timeout, PineSU can check for the transaction validation. After which, it exits the pending state and goes to either step (vi) or (vii), depending on whether the transaction is confirmed or not.

E. CLOSE STORAGE UNITS

This functionality works much like the previous one, but allows the creation of a blockchain closing point for a set of SUs, rather than a BSP. The steps executed by PineSU are basically the same as that for the synchronization, provided that OSG, BSP, and open subtree, are replaced by, respectively, CSG, CSP and closed subtree. Clearly, at step (ix), the closed flag in the **.pinesu.json** file is set to true.

F. SYNCHRONIZATION AND CLOSING

This functionality is extremely useful when the user wants to synchronize some SUs and close some others, spending only one transaction. By entering the **synclose** command, PineSU displays the list of staged SUs, and asks the user to select which ones should remain open and which ones should be closed. The implementation of this functionality essentially consists of merging each step of the last two.

G. VERIFY INTEGRITY WITH BLOCKCHAIN

A fundamental requirement of PineSU is to enable any user of an SU, especially users who are not the owner, to carry out a blockchain-based integrity check. The command **checkbc** makes it possible to do it easily, exploiting only the **.pinesu.json** and **.regpinesu.json** hidden files in the corresponding Git repository (including those in the **.pinesureghistory/** folder), without knowing the MerkleCalendar. The basic idea is to verify whether the SU content is consistent with the MerkleCalendar root hash stored in the blockchain block indicated in the **.regpinesu.json** file; we will refer to this integrity check and to the corresponding block as the *local block integrity check* and the *local block*, respectively. We will see that there could be subtle attacks to trick the local block integrity check, if none of the following security hypotheses is ensured: (*SH1*) there exists only one SU history, i.e., no forking has been done; (*SH2*) even assuming that the SU history has been forked, there exists only one verifiable true branch, and the local block belongs to this branch. The main steps of the local block integrity check are as follows.

i The existence of the block indicated in the **.regpinesu.json** file, as well as the correctness of all the BRD properties, are verified by querying a blockchain node or some Internet API service connected to it. Moreover, the value of the `mkcalroot` property must equal that of the `data` field in the transaction, as well as the values of the `owner` and `bcregtime` properties in the **.pinesu.json** file must coincide with the sender's EOA address and the block timestamp, respectively. In case of some mismatching, the test outputs false.

ii The Merkle root hash of the SU content is computed by applying the procedure described in Subsection V-A, thus fingerprints of resources in the working directory are recomputed; the result is compared with the value of the `merkleroot` property in the **.pinesu.json** file. In case of mismatching, the test returns false.

iii The SU global hash is computed by hashing the `header` content as described in Subsection V-A1. If the result does not coincide with the value of the `hash` property, then the test outputs false.

iv The SU global hash is also compared with the hash in the array `openstoragegroup` (or `closedstoragegroup`) that is associated with the UUID of the SU. Again, in case of mismatching, the test returns false.

v It is checked whether there are repetitions of UUIDs in the `openstoragegroup` (or `closedstoragegroup`) array. In the affirmative case, the test outputs false.

vi A complete binary Merkle tree is built on top of the array of SUs hashes in the `openstoragegroup` (or `closedstoragegroup`), as described in Subsection V-C. Using the root hash of this Merkle tree and the data associated with the `witness` property, the corresponding MerkleCalendar root hash is computed, and then compared with the value of the `mkcalroot` property. If they coincide, then the local block integrity test returns true, otherwise false.

The local block integrity check does not verify the *history consistency*, that is, the consistency of the SU metadata in the **.pinesureghistory/** folder with the blockchain data and between them. Of course, the global integrity verification test must output true if and only if the local block integrity check yields true, for the last registration, and the history consistency is also verified. However, exploiting the SU metadata, a history consistency check can be easily and efficiently performed. Indeed, the `header` of the **.pinesu.json** file contains fields that refer to the previous blockchain registration, which should match the data in the corresponding registration file in the **.pinesureghistory/** folder. In particular, if `YYYY-MM-DDThh-mm-ssZ` is the value of the `prevbcregtime` field, then there must exist a file `.regpinesu-YYYY-MM-DDThh-mm-ssZ.json`, whose content should be coherent with that of `prevx` fields in **.pinesu.json**. In this way, it is possible to trace back from the last blockchain registration to the first one, while performing a suitable set of consistency checks. To speed up the procedure, the local block integrity checks for blocks preceding the last one are *shallow*, i.e., they do not compute fingerprints of resources (as in step (ii)), but just use the metadata. This means that if the global integrity verification test outputs true, we cannot say that even the integrity of the previous Git versions of the working directory are also verified. In other words, the integrity test must be redone for each previous version of the working directory.

H. EXPORT A BLOCKCHAIN VERIFIABLE BUNDLE

A user of an SU, even a non-owner, can export a subset of resources, that are packaged into a compressed archive, while preserving the possibility of checking the integrity with the blockchain. After entering the **export** command from the root folder of an SU, PineSU prompts the user to select a subset of resources that they desire to export. Then, a compressed .zip archive of the SU subset is created. The hidden file **.pinesu.json** is replaced with the **.bundlepinesu.json** file, as described in Subsection V-B. Moreover, the **.pinesureghistory/** folder is renamed as **.bundlepinesureghistory/**, and in each of its files the `filelist` array is removed. Observe that this does not prevent a (shallow) history consistency check. A special option of the **checkbc** command makes it possible to verify the integrity of an SU bundle.

VIII. SECURITY CONSIDERATIONS

Although an in-depth security analysis of PineSU has not been conducted yet, we can draw some important considerations. Assume that the following security hypothesis is satisfied: (*H1*) *every user knows the URL and the owner's address (i.e., the EOA) of a storage unit*. Then, under the *H1* hypothesis, a non-owner user cannot modify an SU and create a new BSP/BCP without being detected, since the corresponding transaction must be signed by the SU's owner. It follows that undetectable BSPs/BCPs can only be created by the owners of SUs. A first attack that we identified, named *double independent history*, can be summarized as follows. The owner of an SU can craft two (or more) versions of it, which are completely isolated to one another since their creation. Each SU version has the own evolution with registration points that never are stored in a same block. Therefore, a non-owner user, say Bob, may be tricked into thinking that only the first variant exists. Conversely, another user, say Alice, may be convinced that only the second variant exists. Most importantly, with no additional security hypotheses, the blockchain-based integrity test does not detect this attack. Of course, a comparison between Alice and Bob's versions may reveal the attack, and if so, they have a verifiable proof that the attack was conducted by the SU owner, which constitutes a strong reason for deterrence. But in many circumstances, such a deterrence may not be enough. However, a double independent history attack can be detected if a second security hypothesis is satisfied: (*H2*) *every SU user knows the block of the first registration*. Indeed, the **checkbc** command includes a history consistency check of an SU. By tracing back along the history, the first registration block is identified and compared with that known to the user. Since the two SU versions do not share any registration block, the attack is detectable. Unfortunately, this is not true in case of a *forking attack*, where the SU owner crafts two (or more) versions that share the beginning of their history, i.e., at least the first registration point, after which they evolve into two independent branches. To detect a forking attack, another security hypothesis must hold: (*H3*) *every SU user knows the block of the last registration*. Observe that if (*H3*) is satisfied, then (*H2*) is no longer necessary. In general, due to time delay issues, guaranteeing (*H3*) may not be always easy, especially when the latest registrations involve consecutive blocks, or blocks that are close to each other. However, these time-related issues can be overcome by a suitable usage policy of PineSU; namely, by imposing that blockchain registrations associated with a specific address must have a minimal temporal distance, e.g., six hours at least. Observe that this is perfectly inline with the PineSU's philosophy, according to which the system should be as much cost-effective as possible.

Summarizing, exploiting the blockchain immutability, PineSU adds strong authenticity and integrity protections to Git repositories, provided that non-owner users know the right owner's address and URL of the corresponding SUs, and registrations performed by a same owner occur sufficiently far away from each other.

IX. CASE STUDY

In this section, we describe an experience using PineSU on clones of two real Git repositories owned by the Italian Presidency of the Council of Ministers (Presidenza del Consiglio dei Ministri,[7] PCM for short) and the Civil Protection Department (Dipartimento della Protezione Civile,[8] DPC for short). We will refer to these repositories as the *pcm-dpc repositories*. They are publicly available on

[7]Presidenza del Consiglio dei Ministri https://www.governo.it.
[8]Dipartimento della Protezione Civile https://www.protezionecivile.it.

GitHub[9] and contain official data, including digital scans of documents with legal value. Sharing these repositories while protecting their authenticity is fundamental to guaranteeing a high level of trust and security. At the time of writing, GitHub offers only weak mechanisms for authenticity protection that are much less secure than those provided by traditional notarization services. More specifically, GitHub allows associating the repositories of one organization to a website domain and can verify whether that organization controls that domain. Since the identity of (https) websites is certified, this association provides a weak form of authenticity verification.[10] In particular, pcm-dpc repositories are associated with the Civil Protection Department domain: https://www.protezionecivile.it. But there is no security mechanism against retroactive attacks by corrupted officers or hackers who have gained access to GitHub. Indeed, the authenticity protection is bound to the GitHub website, and there is no way to verify the authenticity of one or more files in a local repository. Moreover, attackers who gain ownership of repositories can spoof their metadata [65], [66], including the history of old commits [67], [68], [69]. In summary, the authenticity protection is weak and centralized, whereas we want strong and decentralized protection.

We have cloned two pcm-dpc repositories from GitHub to our local machine, which we will refer to as the *CTS repository*[11] and the *Pandemic Monitoring repository* (*PM repository*, for short).[12] The former contains the official reports of the Scientific Technical Committee (Comitato Tecnico Scientifico, CTS for short), whereas the latter mainly contains monitoring data about the infection diffusion on a national, regional, and provincial basis. Examples of data are the number of positive subjects, the number of deaths, the number of hospitalizations, the number of tampons, and so on. A CTS report is published (approximatively) 45 days after the corresponding meeting, since 5 February 2020, when the committee was established. For each report, two PDF files are stored: a shorter version with selectable text (called "accessibile", i.e., "accessible" in English) and without any attachment, and a scanned version of a print of it including all the attachments. The last CTS meeting was held on 30 March 2022, after which the committee was disbanded, due to the end of the Covid-19 state of emergency. Instead, the PM repository is still regularly updated. Initially, DPC officers published data daily, but since 28 October 2022 the repository is updated weekly (typically on Friday afternoon).

As a local machine, we used a VM Lubuntu v1CPU 8GB RAM, on VirtualBox on Windows 11, running on a Lenovo ThinkPad E15, with a 10th Gen. Intel®Core™ i7-10510U processor, 16GB LPDDR5 RAM, and a 512GB PCLe NVMe x2 solid-state hard drive.

[9]pcm-dpc repositories are available at https://github.com/pcm-dpc.

[10]GitHub for organizations - Website domain verification or approval https://docs.github.com/organizations/managing-organization-settings/verifying-or-approving-a-domain-for-your-organization.

[11]https://github.com/pcm-dpc/COVID-19-Verbali-CTS

[12]https://github.com/pcm-dpc/COVID-19

In our case study, we have simulated the data sharing process carried out by the Civil Protection Department using PineSU, while adding some blockchain registrations to increase security. For a more realistic test, we connected PineSU to the Ethereum Mainnet[13]: the public Ethereum network where actual-value transactions take place. For each transaction, the sender and the recipient addresses were as follows:

```
From: 0xc7bb2808c959be4b9b76533601a1c60b147674a0
To:   0x55e3b8ffea6c77ec880297e7185a924e20b8b920
```

both addresses were assumed to be public and securely associated with the owners of the two SUs. Below we give the main phases of our case study.

(i) On Friday, 30 June 2023, at around 1:35 pm (UTC time), we cloned the CTS and the PM repositories from GitHub to our local machine.
The CTS repository contained 490 resources, 26 directories and 464 files, mostly digital scans in PDF format of CTS meeting paper reports, for a total of 5.56GB.
The PM repository contained 5139 resources, 22 directories and 5117 files, including recent updates and resources added by DPC officers, just around 12 pm of the same day. The repository was made up from files in CSV, JSON and PDF format, for a total of 2.4GB.
We then created two PineSU storage units, one for each repository. It took less than 20 seconds for the PM repository, and less than 85 seconds for the CTS repository. The **.pinesu.json** descriptor files of the CTS and PM storage units were 57580 and 665478 characters long, respectively.

(ii) At around 3:00 pm, we staged the two SUs and performed a single blockchain registration by using the **synclose** command; the registration comprised a synchronization and a closing of the PM and CTS storage units, respectively. We have decided to close the CTS storage unit because the last commit on GitHub was on May 17, 2022. Details of the Ethereum transaction associated with this registration are shown in Table 3. The transaction was included in the block with height `17592893` and a timestamp `Jun-30-2023 03:09:47 pm UTC`. Within 10 minutes the transaction was confirmed and the block was `Unfinalized`, after other 8 minutes the status of the block was `Unfinalized (Safe)` and, finally, after other 8 minutes, the status was `Finalized`. PineSU has consistently updated the descriptor files **.pinesu.json** and **.regpinesu.json** of both SUs. In particular, both the **.regpinesu.json** files included the attributes of block `17592893`, and the value of the `mkcalroot` key was equal to the input data of the transaction.

(iii) After one week, on Friday, July 7, around 12:15 pm (UTC time), we modified the PM repository by reproducing the same changes as those performed by the DPC officers on the original version. We then

[13]https://ethereum.org/en/developers/docs/networks/

TABLE 3. Details of the Ethereum transaction associated with the first blockchain registration (triggered by the synclose command) on June 30, 2023.

Transaction Hash:	`0xd68f62698bbfab1ff84939dcf29af48b58ef90ce7839a5ae9f21ae389505522f`
Block:	`17592893`
Timestamp:	`Jun-30-2023 03:09:47 pm UTC`
From:	`0xc7bb2808c959be4b9b76533601a1c60b147674a0`
To:	`0x55e3b8ffea6c77ec880297e7185a924e20b8b920`
Value:	`0 Ether`
Transaction Fee:	`0.001037187244363592 Ether` (value on day of Txn \$1.92 €1.76)
Gas Price:	`0.000000048214356841 Ether (48.214356841 Gwei)`
Gas Limit & Usage by Txn:	`21.512   21.512 (100%)`
Input Data:	`0x4b177cb88642c6d132c46a550af9dc8f3ebd53eaaec0413dc5d7cfbd56a1836a`

TABLE 4. Details of the Ethereum transaction associated with the second blockchain registration (triggered by the sync command) on July 7, 2023.

Transaction Hash:	`0x99e12d1389bc4caa99d9d7cadae4d39d2c64a2b48c023f51b6318b739a9472f9`
Block:	`17641910`
Timestamp:	`Jul-07-2023 12:25:11 pm UTC`
From:	`0xc7bb2808c959be4b9b76533601a1c60b147674a0`
To:	`0x55e3b8ffea6c77ec880297e7185a924e20b8b920`
Value:	`0 Ether`
Transaction Fee:	`0.000458285236326888 Ether` (value on day of Txn \$0.85 €0.78)
Gas Price:	`0.000000021303701949 Ether (21.303701949 Gwei)`
Gas Limit & Usage by Txn:	`21.512   21.512 (100%)`
Input Data:	`0x99b145504bcf826d7657d192035f628c3b8bbaf11d102d293d8b3cc5e052f6e2`

updated and staged the PM storage unit. The update operation took less than 80 seconds as before. Finally, we synchronized the PM storage unit with the blockchain using the **sync** command. Table 4 shows details of the transaction associated with this synchronization. The height of the transaction block is `17641910`, with timestamp `Jul-07-2023 12:25:11 pm UTC`. After about 15 minutes, the status of the block was set to `Finalized`, and files **.pinesu.json** and **.regpinesu.json** were updated accordingly; in particular, file **.regpinesu.json** was updated after a copy of it was saved in folder **.pinesureghistory/** under the name **.regpinesu-2023-06-30T15-09-47Z.json**.

A. ATTACK MODEL

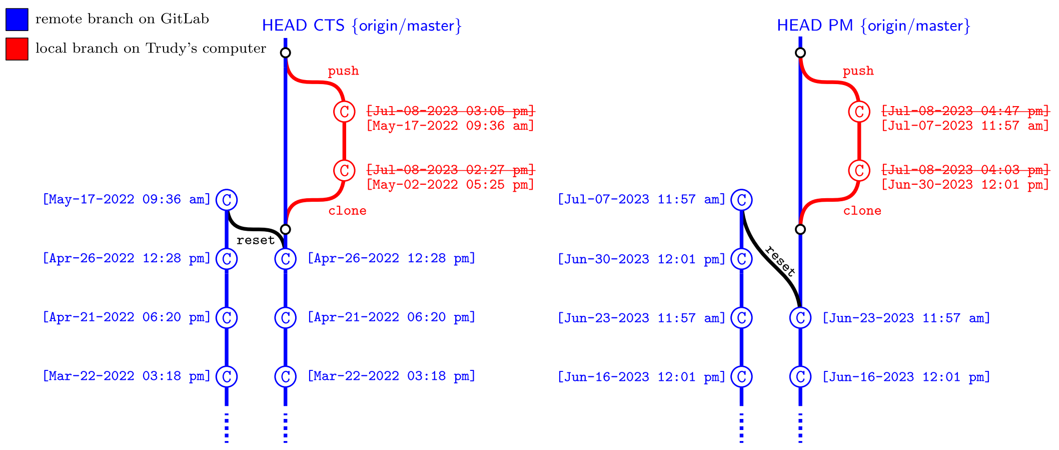
On Saturday, July 8, the following attacks by Trudy (a fancy name) were simulated, assuming that she gained access to the pcm-dpc repositories on GitHub. We accomplished this by cloning the two repositories from the original GitHub location to a personal GitLab account.[14], [15]

- Trudy cloned the CTS repository to her local machine. Then, she crafted a fake report to be added to folder `2022-03/`, between the penultimate (report 64) and the last (report 65), specifying March 18, 2022, as the date. She numbered the fake report as 65, which shifted the last one to 66. An illustration of folder `2022-03/`, before and after the attack, is shown below.

  . . . [before the attack]
  ```
  covid-19-cts-verbale-063-20220304.pdf
  covid-19-cts-verbale-064-20220311.pdf
  covid-19-cts-verbale-065-20220330.pdf
  covid-19-cts-verbale-accessibile-063-20220304.pdf
  covid-19-cts-verbale-accessibile-064-20220311.pdf
  covid-19-cts-verbale-accessibile-065-20220330.pdf
  ```
  . . . [after the attack]
  ```
  covid-19-cts-verbale-063-20220304.pdf
  covid-19-cts-verbale-064-20220311.pdf
  covid-19-cts-verbale-065-20220318.pdf
  covid-19-cts-verbale-066-20220330.pdf
  covid-19-cts-verbale-accessibile-063-20220304.pdf
  covid-19-cts-verbale-accessibile-064-20220311.pdf
  covid-19-cts-verbale-accessibile-065-20220318.pdf
  covid-19-cts-verbale-accessibile-066-20220330.pdf
  ```
  Therefore, the attack involved four PDF files. The two red files did not exist before the attack, whereas the orange ones are a slightly modified version of the old last report (the file name and its numbering were changed). To make the attack detection more complex, Trudy managed it by preserving the date of the last commit. In particular, she acted as follows. She undid all the changes of the last legitimate commit (performed on May 17, 2022) by executing a git **reset**. This operation removed both PDF files of the last original report (old number 65). Then, she added the two PDF files of the fake report (numbered 65) and ran a first commit. Next, she re-added the two slightly modified versions of the last original report (now number 66) and ran a second commit; an illustration of the Git commit graph is shown in Fig. 12a.

  Of course, Trudy named the fake files according to the CTS numbering and naming convention. Moreover, to conceal the attack, she suitably altered the commit metadata, using, where possible, the same metadata as the repository version just before the reset operation. In particular, she spoofed the timestamps of her first and second commit as `May-02-2022 05:25:17 pm UTC`

[14]https://gitlab.com/rfjp6hn181/COVID-19-Verbali-CTS
[15]https://gitlab.com/rfjp6hn181/covid-19



(a) CTS repository (b) PM repository

FIGURE 12. Git commit graph of Trudy's attacks on (a) CTS and (b) PM repositories. The blue color represents the remote branch on GitLab, whereas the red color represents local branch on Trudy's machine. Overwritten timestamps are crossed out, and corresponding spoofed values are shown beneath.

and `May-17-2022 09:36:00 am UTC`, respectively. After that, Trudy pushed to GitLab her two commits with spoofed metadata.

- With a similar attack, Trudy tampered with a CSV file uploaded to the PM repository on June 16, 2023. In particular, she cloned the PM repository and undid all the changes of the last two legitimate commits, until the commit of June 23, 2023. Then, she modified the file `dati-regioni/dpc-covid19-ita-regioni-20230614.csv` that was uploaded on June 16. Next, she re-added all the documents changed on the commit of June 30 and ran a first Git commit. After that, she re-applied all the changes of the last legitimate commit of July 7 and ran a second commit. Finally, she spoofed the timestamps of commits to make the attack difficult to detect and pushed her changes to GitLab (see Fig. 12b for an illustration of the Git commit graph).

Attacks of this type can be very challenging to detect using only the integrity protection mechanisms of Git and GitHub/GitLab. Also, even if someone had a correct local copy of a poisoned repository, they would be unable to prove it, which could lead to costly disputes to resolve.

B. ATTACK DETECTION

We now show how the previous attack on the CTS and PM repositories can be easily detected by using PineSU.

CTS Repository Poisoning Detection. Let Alice be any non-owner user of the CTS repository who knows that the block with height 17592893 is the block of the last registration involving this repository. In other words, the security hypothesis H3 holds.

On Tuesday, 11 July 2023, i.e., after the attack, Alice cloned (or pulled) the CTS repository from GitLab to her local machine. To verify the repository integrity against the blockchain, she ran the **checkbc** command of PineSU. In less than two minutes, PineSU returned false in output. Indeed, the hash of the actual SU content differed from that of the last legitimate commit. This inconsistency propagates to the MerkleCalendar root hash computed using the metadata in the **.regpinesu.json** file of the local repository copy. Hereafter we report the CTS storage unit hash and the MerkleCalendar root hash immediately before and after the attack.

… [before the attack]

```
CTS-SU-hash:      0 × 6f0a746be4015cb5ee...2f4df909870b550e86
Mkcalroot-hash:   0 × 4b177cb88642c6d132...3dc5d7cfbd56a1836a
```

… [after the attack]

```
CTS-SU-hash:      0 × 7cc7de57c69cd3b998...448ee8976973c44133
Mkcalroot-hash:   0xee37162c5403e1a6be...38a9f04f8a4a340a60
```

Eventually, PineSU returned false because the computed MerkleCalendar root hash differed from that stored in the blockchain.

PM Repository Poisoning Detection. Consider a non-owner user, say Bob, of the PM repository. Bob cloned the PM repository from GitLab to his computer on Wednesday 12 July. Suppose that Bob knows that the block with height 17641910 is the block of the last registration. As above, we give the hashes of the PM storage unit and the root of the MerkleCalendar, both before and after the attack.

… [before the attack]

```
CTS-SU-hash:      0 × 7ac8e949f017b99c5e...82a0ad1eace71b98b2
Mkcalroot-hash:   0 × 99b145504bcf826d76...293d8b3cc5e052f6e2
```

… [after the attack]

```
CTS-SU-hash:      0 × 2d81b955385a7277c1...9820f00ccbcf339bbd
Mkcalroot-hash:   0 × 070fd499afa6317bbd...4d5fd33b4e635905e4
```

By running the **checkbc** command, which took less than one minute, Bob easily detected the integrity attack.

In both cases, Alice and Bob have verifiable proof that the repositories on GitLab are not authentic. Also, if some other user has a copy of a repository just before the attack, they have verifiable proof that their copy is authentic. Of course, this can be very helpful for recovering corrupted repositories.

X. PERFORMANCE EVALUATION

In this section, we provide a performance evaluation of PineSU, expressed in terms of memory and time consumption, for storage units of varying sizes and structures. The aim is to show how PineSU scales well, under assumptions that are very common in practice. We limit our analysis to the create and update operations, which are the most critical, since they require the creation/update of the whole file .pinesu.json. (Observe that the registration operations may take a longer time, but this is due to the latency of the blockchain and not to PineSU.) In particular, we evaluated the length of the .pinesu.json file, expressed as the number of characters, and the time needed for its creation/update, in milliseconds. As dataset, we considered a collection of repositories described by three integer attributes: b, n and c. These attributes take values in the sets D_b, D_n and D_c, respectively, where $D_b = \{10^5, 10^6, 10^7\}$, $D_n = \{5, 10, 25, 50, 100, 250, 500, 10^3, 10^4, 10^5, 10^6\}$ and $D_c = \{1, 2, 5, 10, 25, 50, 100\}$. For each triple (b, n, c), there is a repository $R(b, n, c)$ that satisfies the following conditions.

(i) each file consists of a pseudo-randomly generated sequence of b bytes;
(ii) each file name has a length of 60 characters;
(iii) each folder name has a length of 30 characters;
(iv) there are exactly n files (folders are not considered files);
(v) there are no empty folders;
(vi) every folder can contain at most $2c$ children, with at most c files and, in suborder, with at most c sub-folders (we will refer to c as the *children threshold*);
(vii) the hierarchical structure of a repository has the minimum depth among those compatible with n and c.

The last three conditions are inspired by behavioral rules that users usually adopt when archiving their files. For example, the triple $(b, n, c) = (10^6, 110, 10)$ describes repositories with 110 files of 1 MB each, such that the root folder contains 10 files and 10 sub-folders, and each sub-folder contains exactly 10 files. Observe that c affects the depth of the repository, i.e., the higher the c the lower the depth. Also, for $c > 1$ the repositories have a well-balanced hierarchical structure, since a folder can have two or more sub-folders. Instead, for $c = 1$, the repositories are strongly unbalanced.

A. LENGTH OF THE .PINESU.JSON DESCRIPTOR FILE

Let L denote the length of the descriptor file .pinesu.json expressed as total number of characters. Apart from a constant contribution, L increases as the number of files in the repository and the length of their relative paths increase. Therefore, taking into account that files in deeper folders have longer relative paths, we expect L to increase with n but decrease with c. Note that the parameter b is irrelevant,

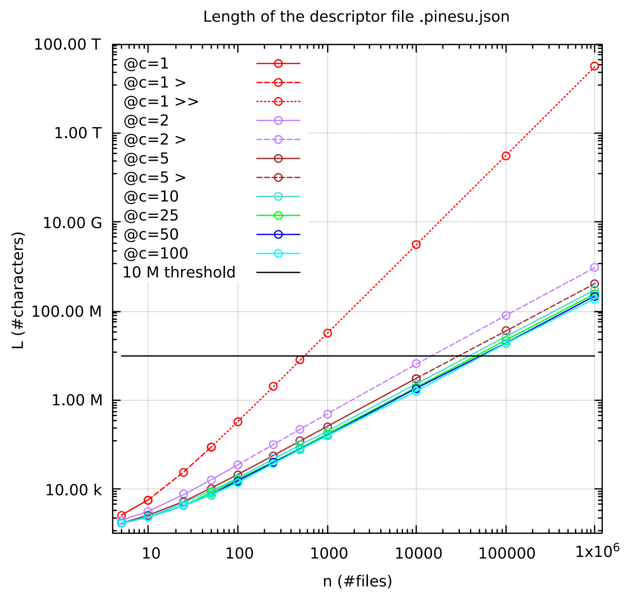


FIGURE 13. Length L of the descriptor file .pinesu.json, expressed as number of characters, with respect to the number of files $n \in D_n$, for different children thresholds $c \in D_c$. The line colors encode the different values of c, whereas the line styles encode the maximum length of the resources' relative paths: solid, dashed and dotted lines correspond to normal, long and very long lengths, respectively.

since fingerprints of files have a fixed length, irrespective their memory occupation.

We performed a simulation to estimate how much L varies with $n \in D_n$, for each value of $c \in D_c$. Fig. 13 shows the simulation results as a log-log plot (raw data are given in the Appendix, in Table 6). Each line shows L as a function of n, for a fixed value of c (encoded by the line color). The line style represents the maximum length l of the resources' relative paths, according to the following scale: solid line for *normal* length ($l \leq 260$ characters), dashed line for *long* length ($260 < l \leq 1000$) and dotted line for *very long* length ($l > 1000$). We use the labels $>$ and $\gg$ to denote repositories whose parameter l is, respectively, long and very long.

We recall that equations of the form $y = \alpha x^e$ correspond to straight-line segments in a log-log plot, where the exponent e determines the slope and the factor α represents the intercept on the vertical axis. Also, considering that the x- and y-axes have different scales, in particular, a same variation $\Delta_x = \Delta_y$ is twice longer along the x-axis than the y-axis, we can draw the following considerations:

- for any fixed $c \geq 2$, L grows linearly with n, as the corresponding lines have approximately slope 0.5 (i.e., 26.6°);
- for $c = 1$, L grows (approximately) as a quadratic function of n, indeed, the corresponding line has slope 1 (i.e., 45°);
- for $c = 1$, there is an increasing number of resources with very long relative paths (dotted red line), but in practice this cannot happen because modern operating systems do not accept file names so long.

- though L decreases as c increases (for a fixed n), for $c > 5$, higher values of c lead to relative low decrements of L;
- for $c \geq 5$, L is less than 10^7 (black horizontal line) for repositories with up to tens of thousands of files, and only for $n > 10^4$ there are resources with a long relative path.

To summarize, PineSU adds only a lightweight overhead to the memory consumption, in all practical situations, which scales linearly with the number of files.

B. TIME FOR COMPUTING/UPDATING THE .PINESU.JSON DESCRIPTOR FILE

Let T denote the time (in milliseconds) needed for creating/updating the .pinesu.json file. Of course, T increases as the size b and the number n of files increase, whereas it does not appreciably change with the children threshold c.

We developed a simple mathematical model to estimate T, which does not require the execution of the create and update operations nor the execution of cryptographic hash algorithms on files in the dataset. According to our model, T is the sum of four contributions:

- T_{lf}: the time needed for computing the list of fingerprints of all the SU resources;
- T_{mr}: the time spent for computing the hash of the Merkle tree root built on top of the list of fingerprints;
- T_{hh}: the time taken to compute the header of the .pinesu.json file and its hash;
- T_{wd}: the time needed to write the .pinesu.json file to disk.

The contributions T_{lf}, T_{mr} and T_{hh} exploit a simple linear function to estimate the running time t_h of SHA-256 on a file or string: $t_h = m_h \cdot size + t_{h0}$. For a file in the dataset, *size* coincides with b; whereas, for a generic string, we assume an encoding where each character takes an average of one byte. Similarly, we use a linear function $t_w = m_w \cdot size + t_{w0}$ to estimate T_{wd}, where *size* represents the number of bytes of the .pinesu.json file. By performing a few pilot tests on our hardware, the parameters m_h, m_w, t_{h0} and t_{w0} were set as follows:

- $m_h = 3$ nanoseconds per byte;
- $m_w = 5$ nanoseconds per byte;
- $t_{h0} = 340$ microseconds;
- $t_{w0} = 2750$ microseconds.

In Fig. 14, the lines with a circle point type show the simulation results of our model in a log-log plot, for $c = 10$. As the x- and y-axes have the same scale, thus a slope 1 represents a linear function, it follows that the running time T grows approximately linearly in the number of files. For repositories with up to 1000 files, we experimentally validated our model, by measuring the running time of the create/update operations on our hardware. The results are shown by lines with a triangle point type in the log-log plot of Fig. 14. Both simulated and experimental results show that even the running time of PineSU scales linearly with the size of repositories. In particular, the running time is less than 1 minute for repositories with up to 1000 files with size at most 10 MB each, and less than 10 minutes for repositories with up to 10 thousands of files.

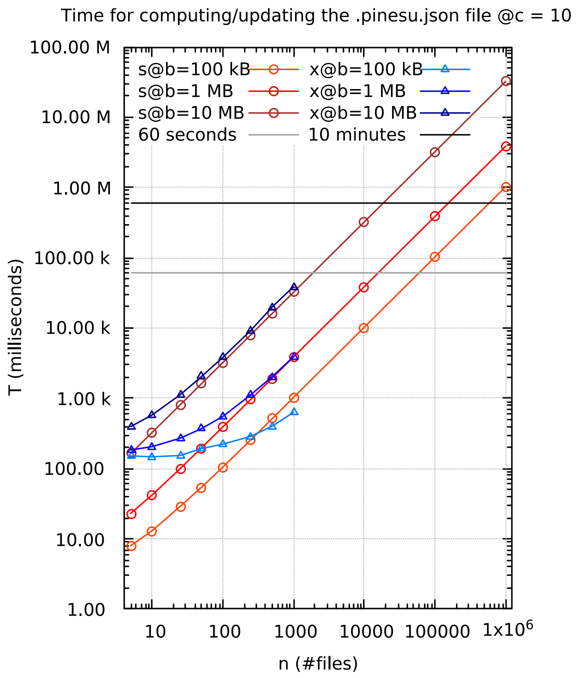


FIGURE 14. Experimental and simulated execution time of the create/update operations of PineSU. Each line chart represents how the time T grows with the number of files $n \in D_n$, for a fixed file size $b \in D_b$ and for $c = 10$. The line charts of simulated (resp., experimental) results have a circle (resp., triangle) as point glyph and are identified by an 's' (resp. 'x') prefix in the legend labels. The experimental tests were performed on the repositories with up to 1000 files of our dataset, for $c = 10$.

XI. DISCUSSION

As already mentioned in Section III, several approaches have been proposed for secure and trusted data sharing. In this section, we provide a qualitative comparison of PineSU with known methodologies or systems. We remark that a rigorous quantitative analysis should embrace several metrics, which would require a whole paper. Therefore, we consider a simple attribute-based comparison. We identified the following set of attributes that are categorized into three groups: *Security*, *File Sharing Functionality*, and *Cost* (see Table 5). The attributes are defined so that their possession represents a positive feature.

A. SECURITY

- *no trusted third party*: the security does not require any trusted third party or central authorization entity.
- *no fully trusted source*: the security does not require fully trusted data sources (i.e., data owners) since there are security mechanisms to prevent or, at least, detect attacks by malicious sources.
- *no data owner-consumer secure channel*: the security does not require sending a verification code through a secure channel between the data owner and the data consumer.

TABLE 5. Qualitative attribute-based comparative analysis of PineSU with known methodologies or systems.

		methodologies or systems															
		off-chain						blockchain-based						VCS&blockchain-based			
	attributes	[18]	[40]	[41]	[42]	[43]	[44]	[46]	[48]	[49]	[50]	[51]	[52]	[54]	[55]	[56]	PineSU
security	no trusted third party	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	no fully trusted source						✓	●	●	●	●	●	●	✓	●	●	✓
	no data owner-consumer sec. channel	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	prevention of retroactive alteration							✓	✓	✓	✓	✓	✓	✓	●	○	●
	detection of retroactive attacks	○	○	○	○	○	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	immutability of verification codes	○	○	○	○	○	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	interminability of verification codes							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
file sharing functionality	version support	●		●	●	●	✓	○	✓	✓	✓			✓	✓	✓	✓
	folder support	●		●	●	●	●										✓
	Git or (distributed) VCS integration	●	○	○	●	●	○									✓	✓
	IPFS integration	○	○	○	○	○	○	●	✓	✓	✓	○	●	✓	✓	✓	○
	decentralized multi-user collaboration													✓	●		
cost	computational practicability	✓	●	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	admin-less	●	●	●	●	●	●	✓	✓		✓		✓	✓		✓	✓
	cost-effective blockchain interaction	n/a	n/a	n/a	n/a	n/a	n/a	✓	○	○	✓	○	○	○	✓	✓	✓
	batching	●	●	●	●	●	●										✓

LEGEND
- ✓ (check mark symbol) the solution exhibits the attribute in the corresponding row
- ● (full circle) the solution does not posses the corresponding attribute but can have it with a limited effort
- ○ (empty circle) similar to the previous case, although much more effort is required to make the solution possess the attribute
- ■ (gray text) the solution is mainly theoretical or describes a very abstract methodology
- ■ (black text) the solution describes an implemented system or a well-detailed application scenario

- *prevention of retroactive alteration*: there are security mechanisms that prevent attackers, including the data owners themselves, from modifying previously shared data without deleting them (i.e., deleting previously shared data remains a possible attack).
- *detection of retroactive attacks*: there are security mechanisms to quickly detect any change to previously shared data, including alteration and deletion by data owners.
- *immutability of verification codes*: no attacker (including data owners) can modify the integrity verification codes once issued without being detected.
- *Interminability of verification codes*: the integrity verification codes are stored on some public permissionless ledger forever; no one can delete them.

B. FILE SHARING FUNCTIONALITY

- *version support*: there are mechanisms to track all changes to shared files over time.
- *folder support*: there is support for managing both single files and whole file system folders at once.
- *Git or (distributed) VCS integration*: the proposed approach for trusted data sharing can be easily integrated with Git or any (distributed) VCS software.
- *IPFS integration*: the proposed approach stores files on IPFS.
- *decentralized multi-user collaboration*: the content of shared files is determined through a proposal and approval process involving distinct users.

C. COST

- *computational practicability*: each operation of the proposed solution requires a reasonable computational cost (expressed in terms of memory consumption and CPU usage). Therefore, the solution may be suitable for many practical applications.
- *admin-less*: the proposed solution does not need any administrator user to work appropriately.
- *cost-effective blockchain interaction*: the cost of a single blockchain state-changing interaction is comparable to that of a regular transaction (i.e., a transaction to transfer ETH from one wallet to another).
- *batching*: the proposed approach allows multiple independent data-sharing operations by performing a single blockchain transaction or database transaction.

Table 5 summarizes the qualitative comparison of PineSU against a selection of existing approaches. Columns in gray indicate that the corresponding solutions are mainly theoretical or illustrate very abstract approaches, i.e., they do not describe any implemented system nor focus on some specific application scenario. Due to the breadth and heterogeneity of the "trusted data sharing" topic, we did our best to investigate whether existing solutions were adaptable to protect the integrity and authenticity of VCS repositories. In particular, the check mark symbol indicates that the corresponding solution, identified by the column, exhibits the feature identified by the intersecting row. A full (resp. empty) circle means that the solution does not explicitly possess the corresponding feature but can be adapted to have it with a limited (resp. significant) effort. It turns out that PineSU is the

TABLE 6. Length of the descriptor file .pinesu.json expressed as number of characters (*L*) with respect to the number of files *n* of a repository, for different children thresholds *c*.

n	$c=1$	$c=2$	$c=5$	$c=10$	$c=25$	$c=50$	$c=100$
5	$2.62 \cdot 10^3$	$2.02 \cdot 10^3$	$1.72 \cdot 10^3$	$1.73 \cdot 10^3$	$1.73 \cdot 10^3$	$1.73 \cdot 10^3$	$1.73 \cdot 10^3$
10	$5.77 \cdot 10^3 >$	$3.20 \cdot 10^3$	$2.62 \cdot 10^3$	$2.37 \cdot 10^3$	$2.37 \cdot 10^3$	$2.37 \cdot 10^3$	$2.37 \cdot 10^3$
25	$2.45 \cdot 10^4 >$	$7.61 \cdot 10^3$	$5.30 \cdot 10^3$	$4.95 \cdot 10^3$	$4.29 \cdot 10^3$	$4.29 \cdot 10^3$	$4.29 \cdot 10^3$
50	$8.68 \cdot 10^4 \gg$	$1.60 \cdot 10^4$	$1.05 \cdot 10^4$	$9.12 \cdot 10^3$	$8.36 \cdot 10^3$	$7.49 \cdot 10^3$	$7.49 \cdot 10^3$
100	$3.28 \cdot 10^5 \gg$	$3.52 \cdot 10^4 >$	$2.13 \cdot 10^4$	$1.76 \cdot 10^4$	$1.65 \cdot 10^4$	$1.55 \cdot 10^4$	$1.39 \cdot 10^4$
250	$1.98 \cdot 10^6 \gg$	$1.00 \cdot 10^5 >$	$5.72 \cdot 10^4$	$4.77 \cdot 10^4$	$4.10 \cdot 10^4$	$3.97 \cdot 10^4$	$3.80 \cdot 10^4$
500	$7.83 \cdot 10^6 \gg$	$2.22 \cdot 10^5 >$	$1.21 \cdot 10^5$	$9.84 \cdot 10^4$	$8.17 \cdot 10^4$	$8.00 \cdot 10^4$	$7.79 \cdot 10^4$
10^3	$3.12 \cdot 10^7 \gg$	$4.88 \cdot 10^5 >$	$2.55 \cdot 10^5$	$2.00 \cdot 10^5$	$1.74 \cdot 10^5$	$1.60 \cdot 10^5$	$1.58 \cdot 10^5$
10^4	$3.10 \cdot 10^9 \gg$	$6.44 \cdot 10^6 >$	$3.10 \cdot 10^6 >$	$2.33 \cdot 10^6$	$1.93 \cdot 10^6$	$1.84 \cdot 10^6$	$1.60 \cdot 10^6$
10^5	$3.10 \cdot 10^{11} \gg$	$7.99 \cdot 10^7 >$	$3.57 \cdot 10^7 >$	$2.67 \cdot 10^7$	$2.22 \cdot 10^7$	$1.92 \cdot 10^7$	$1.88 \cdot 10^7$
10^6	$3.10 \cdot 10^{13} \gg$	$9.50 \cdot 10^8 >$	$4.16 \cdot 10^8 >$	$3.01 \cdot 10^8 >$	$2.46 \cdot 10^8$	$2.20 \cdot 10^8$	$1.91 \cdot 10^8$

LEGEND

░	$10^6 \le L < 10^7$
▓	$10^7 \le L < 10^8$
🟥	$10^8 \le L < 10^9$
🟪	$10^9 \le L$
$>$	$260 < l \le 1000$, where l is the maximum length of the resources' relative paths
$\gg$	$1000 < l$

only solution that tracks the changes and synchronizes with the blockchain a whole file system folder (i.e., a repository) at once. All the other solutions only work on one file at a time. Furthermore, it possesses almost all security features and is one of the most cost-effective.

XII. LIMITATIONS

Although PineSU has proven to be very effective in protecting the integrity and authenticity of Git repositories, there are some limitations to be aware of, which are listed below.

- *Denial Of Service attacks*: as with all the blockchain-based solutions, only the verification codes are stored in the ledger, while the actual content of repositories is stored off-chain. It means that DoS attacks aimed at deleting or encrypting files may irremediably lead to a loss of information unless correct copies of corrupted repositories are securely stored elsewhere.
- *Correct data recovery*: while PineSU can easily detect attacks on the integrity of Git repositories, it does not include built-in mechanisms to recover the correct data version. To guarantee such a security requirement too, the owners of repositories should take care of securely storing copies of them. A good practice could be to store on-premise a copy of each publicly shared repository.
- *Real-time applications*: PineSU has evident limitations in protecting real-time application data. Specifically, if the authenticity of data, including the corresponding timestamp, needs to be verified every minute or less, then the number of blockchain transactions (and thus fees) would significantly increase. Furthermore, the attacks described in Section VIII would be feasible, which reduces the system's security.

XIII. CONCLUSION AND OPEN PROBLEMS

We presented PineSU, a lightweight software system that adds strong integrity and authenticity protection to Git repositories, by leveraging the security of the Ethereum blockchain. PineSU wraps Git repositories into storage units, through suitable hidden files that are stored in the corresponding working directories. At any time, users can store the state of a group of their storage units into the blockchain, by performing a regular transaction that includes the root hash of a calendar-based cryptographic accumulator. This accumulator efficiently combines and keeps track of storage units fingerprints for each blockchain registration date. Non-owner users can clone storage units and verify the integrity against the blockchain, by using only their PineSU-specific hidden files, which has a very little overhead. From a scientific perspective, the contribution of our research is to show how to combine known ingredients like cryptographic accumulators, trees data structures with chronologically-ordered nodes, and blockchain technologies to design a lightweight software that is very effective in protecting the authenticity of VCS repositories under a few security hypotheses that are easy to guarantee in many circumstances. However, there is still room for improvements, and we plan to enhance the system by extending its functionality as follows.

- *Multi-ownership*. Currently, every storage unit has a single owner, who is the only party authorized to perform registrations on the blockchain. To enable distinct users, or a group of users, to perform registrations of a same storage unit, a multi-ownership model should be adopted by suitably using multi-signature schemes, which are supported by many blockchains.
- *Multi-paradigm*. The detection mode is the security paradigm adopted by PineSU, which cannot prevent (at blockchain level) invalid registration operations like closing an already closed storage unit. The security of PineSU can be strengthened by adopting even the prevention mode, which requires the development of a smart contract that implements a suitable access control logic.
- *Storage Unit Composability*. PineSU enables users to export their storage units or subsets of them (i.e.,

bundles). However, there is no mechanism to include a reference to one or more external SUs within an SU, whereas Git does provide a feature like that for repositories (the submodules).

APPENDIX

In Table 6, we report the exact values of the line charts depicted in Figure 13 that represent the length L of the descriptor file **.pinesu.json** expressed as the total number of characters w.r.t. the number of files n of a repository (for different children thresholds c).

REFERENCES

[1] Eur. Commission. (2020). *Commission Staff Working Document Impact Assessment Report Accompanying the Document Proposal for a Regulation of the European Parliament and of the Council on European Data Governance (Data Governance Act)*. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52020SC0295

[2] Eur. Commission. (2020). *Proposal for a Regulation of the European Parliament and of the Council on European Data Governance (Data Governance Act)*. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52020PC0767

[3] Gartner, Inc. (2021). *Data Sharing is a Bus. Necessity to Accelerate Digital Business*. Accessed: Apr. 5, 2024. [Online]. Available: https://www.gartner.com/smarterwithgartner/data-sharing-is-a-business-necessity-to-accelerate-digital-business

[4] *Managing the Security of Information Exchanges*, Nat. Inst. Standards Technol., Washington, DC, USA, 2021.

[5] Data Spaces Support Centre. *Knowledge Base*. Accessed: Apr. 5, 2024. [Online]. Available: https://dssc.eu/page/knowledge-base

[6] M. S. Ali, M. Vecchio, M. Pincheira, K. Dolui, F. Antonelli, and M. H. Rehmani, "Applications of blockchains in the Internet of Things: A comprehensive survey," *IEEE Commun. Survey Tuts.*, vol. 21, no. 2, pp. 1676–1717, 2nd Quart., 2019, doi: 10.1109/COMST.2018.2886932.

[7] Deloitte. (2021). *Deloitte's 2021 Global Blockchain Survey: A New Age of Digital Assets*. Accessed: Apr. 5, 2024. [Online]. Available: https://www2.deloitte.com/content/dam/insights/articles/US144337_Blockchain-survey/DI_Blockchain-survey.pdf

[8] A. Haleem, M. Javaid, R. P. Singh, R. Suman, and S. Rab, "Blockchain technology applications in healthcare: An overview," *Int. J. Intell. Netw.*, vol. 2, pp. 130–139, Sep. 2021, doi: 10.1016/j.ijin.2021.09.005.

[9] D. Cagigas, J. Clifton, D. Diaz-Fuentes, and M. Fernández-Gutiérrez, "Blockchain for public services: A systematic literature review," *IEEE Access*, vol. 9, pp. 13904–13921, 2021, doi: 10.1109/ACCESS.2021.3052019.

[10] D. Allessie, M. Sobolewski, and L. Vaccari, "Blockchain for digital government," F. Pignatelli, Ed., Publications Office of the European Union, Luxembourg, EUR 29677 EN, 2019, doi: 10.2760/93808. [Online]. Available: https://publications.jrc.ec.europa.eu/repository/handle/JRC115049

[11] C. G. Reddick, M. P. Rodriguez-Bolivar, and H. J. Scholl, *Blockchain and the Public Sector—Theories, Reforms, and Case Studies*. Cham, Switzerland: Springer, 2021.

[12] Eur. Commission. (2020). *Communication From the Commission to the European Parliament, the Council, the European Economic and Social Committee and the Committee of the Regions: A European Strategy for Data*. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52020DC0066

[13] Community Group. *Solid Technical Reports*. Accessed: Apr. 5, 2024. [Online]. Available: https://solidproject.org/TR/

[14] (2022). *Git Software Freedom Conservancy*. Accessed: Apr. 5, 2024. [Online]. Available: https://git-scm.com

[15] S. Chacon and B. Straub. (2014). *Pro Git*. [Online]. Available: https://git-scm.com/book/en/v2

[16] *Secure Hash Standard (SHS)*, Nat. Inst. Standards Technol., Washington, DC, USA, 2015.

[17] J. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *Advances in Cryptology—EUROCRYPT '93* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 1994, pp. 274–285.

[18] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables based on cryptographic accumulators," *Algorithmica*, vol. 74, no. 2, pp. 664–712, Feb. 2016, doi: 10.1007/s00453-014-9968-3.

[19] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed E-cash from Bitcoin," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2013, pp. 397–411, doi: 10.1109/SP.2013.34.

[20] B. Bailey and S. Sankagiri, "Merkle trees optimized for stateless clients in Bitcoin," in *Financial Cryptography and Data Security* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2021, pp. 451–466.

[21] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to IOPs and stateless blockchains," in *Advances in Cryptology–CRYPTO 2019* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2019, pp. 561–586.

[22] D. Derler, C. Hanser, and D. Slamanig, "Revisiting cryptographic accumulators, additional properties and relations to other primitives," in *Topics in Cryptology—CT-RSA* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2015, pp. 127–144.

[23] I. Ozcelik, S. Medury, J. Broaddus, and A. Skjellum, "An overview of cryptographic accumulators," in *Proc. 7th Int. Conf. Inf. Syst. Secur. Privacy*, 2021, pp. 661–669, doi: 10.5220/0010337806610669.

[24] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. IEEE Symp. Secur. Privacy*, Apr. 1980, p. 122, doi: 10.1109/SP.1980.10006.

[25] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology—CRYPTO' 89 Proceedings*. Cham, Switzerland: Springer, 1989, pp. 218–238.

[26] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2nd ed. Boca Raton, FL, USA: CRC Press, 2742.

[27] S. Nakamoto. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[28] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2017.

[29] 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.

[30] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018.

[31] Ethereum.org. *Ethereum Whitepaper*. Accessed: Apr. 5, 2024. [Online]. Available: https://ethereum.org/en/whitepaper/

[32] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Accessed: Apr. 5, 2024. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[33] A. Sward, I. Vecna, and F. Stonedahl, "Data insertion in Bitcoin's blockchain," *Ledger*, vol. 3, p. 3, Apr. 2018, doi: 10.5195/ledger.2018.101.

[34] P. Kostamis, A. Sendros, and P. Efraimidis, "Exploring Ethereum's data stores: A cost and performance comparison," in *Proc. 3rd Conf. Blockchain Res. Appl. Innov. Netw. Services (BRAINS)*, Paris, France, Sep. 2021, pp. 53–60, doi: 10.1109/BRAINS52497.2021.9569804.

[35] A. Hafid, A. S. Hafid, and M. Samih, "Scaling blockchains: A comprehensive survey," *IEEE Access*, vol. 8, pp. 125244–125262, 2020, doi: 10.1109/ACCESS.2020.3007251.

[36] Solidity Team. *Solidity Programming Language*. Accessed: Apr. 5, 2024. [Online]. Available: https://soliditylang.org/

[37] Vyper Team. *GitHub—Vyperlang/Vyper: Pythonic Smart Contract Language for the EVM*. Accessed: Apr. 5, 2024. [Online]. Available: https://github.com/vyperlang/vyper

[38] Bitcoin Wiki. *Script*. Accessed: Apr. 5, 2024. [Online]. Available: https://en.bitcoin.it/wiki/Script

[39] S. K. Kotha, M. S. Rani, B. Subedi, A. Chunduru, A. Karrothu, B. Neupane, and V. E. Sathishkumar, "A comprehensive review secure data sharing cloud environment," *Wireless Pers. Commun.*, vol. 127, no. 3, pp. 2161–2188, 2022. Accessed: Apr. 5, 2024.

[40] G. Zhao, C. Rong, J. Li, F. Zhang, and Y. Tang, "Trusted data sharing over untrusted cloud storage providers," in *Proc. 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 97–103.

[41] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, May 2004, doi: 10.1007/s00453-003-1076-8.

[42] R. Tamassia, "Authenticated data structures," in *Proc. 11th Annu. Eur. Symp. Algorithms*, 2003, pp. 2–5.

[43] R. Tamassia and N. Triandopoulos, "Efficient content authentication in peer-to-peer networks," in *Proc. 5th Int. Conf. Appl.*, 2007, pp. 354–372.
[44] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1299–1316, doi: 10.1145/3319535.3345652.
[45] L. T. Nguyen, L. Duc Nguyen, T. Hoang, D. Bandara, Q. Wang, Q. Lu, X. Xu, L. Zhu, P. Popovski, and S. Chen, "Blockchain-empowered trustworthy data sharing: Fundamentals, applications, and challenges," 2023, *arXiv:2303.06546*.
[46] R. Kalis and A. Belloum, "Validating data integrity with blockchain," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2018, pp. 272–277, doi: 10.1109/CloudCom2018.2018.00060.
[47] J. Benet, "IPFS–content addressed, versioned, P2P file system," 2014, *arXiv:1407.3561*.
[48] N. Nizamuddin, H. R. Hasan, and K. Salah, "IPFS-blockchain-based authenticity of online publications," in *Proc. 1st Int. Conf. Blockchain*, 2018, pp. 199–212.
[49] S. Khatal, J. Rane, D. Patel, P. Patel, and Y. Busnel, "FileShare: A blockchain and IPFS framework for secure file sharing and data provenance," in *Algorithms for Intelligent Systems*. Cham, Switzerland: Springer, 2021, pp. 825–833.
[50] Y. Azizi, M. Azizi, and M. Elboukhari, "Log data integrity solution based on blockchain technology and IPFS," *Int. J. Interact. Mobile Technol.*, vol. 16, no. 15, pp. 4–15, Aug. 2022, doi: 10.3991/ijim.v16i15.31713.
[51] B. Shen, J. Guo, and Y. Yang, "MedChain: Efficient healthcare data sharing via blockchain," *Appl. Sci.*, vol. 9, no. 6, p. 1207, 2019.
[52] K. Hasan, M. J. M. Chowdhury, K. Biswas, K. Ahmed, M. S. Islam, and M. Usman, "A blockchain-based secure data-sharing framework for software defined wireless body area networks," *Comput. Netw.*, vol. 211, Apr. 2022, Art. no. 109004.
[53] H. Han, S. Fei, Z. Yan, and X. Zhou, "A survey on blockchain-based integrity auditing for cloud data," in *Digital Communications and Networks*. Beijing, China: KeAi Communications Co., 2022, doi: 10.1016/j.dcan.2022.04.036. [Online]. Available: https://www.keaipublishing.com/en/journals/digital-communications-and-networks/ and https://www.keaipublishing.com/en/ and https://www.scopus.com/sourceid/21100823476
[54] N. Nizamuddin, K. Salah, M. Ajmal Azad, J. Arshad, and M. H. Rehman, "Decentralized document version control using Ethereum blockchain and IPFS," *Comput. Electr. Eng.*, vol. 76, pp. 183–197, Jun. 2019, doi: 10.1016/j.compeleceng.2019.03.014.
[55] M. Hammad, J. Iqbal, C. A. U. Hassan, S. Hussain, S. S. Ullah, M. Uddin, U. A. Malik, M. Abdelhaq, and R. Alsaqour, "Blockchain-based decentralized architecture for software version control," *Appl. Sci.*, vol. 13, no. 5, p. 3066, Feb. 2023, doi: 10.3390/app13053066.
[56] L. Melia. *GitHub—Cardstack/Githereum: Gitchain Implementation in Ethereum*. Accessed: Apr. 5, 2024. [Online]. Available: https://github.com/cardstack/githereum
[57] A. Speller. *GitHub—Cardstack/Gitchain*. Accessed: Apr. 5, 2024. [Online]. Available: https://github.com/cardstack/gitchain
[58] C. Tse. *Introducing Gitchain—Cardstack's Protocol for Syncing Application States*. Accessed: Apr. 5, 2024. [Online]. Available: https://medium.com/cardstack/introducing-gitchain-add61790226e
[59] *Ethereum*. Accessed: Apr. 5, 2024. [Online]. Available: https://ethereum.org
[60] *GitHub—Web3/Web3.JS: Ethereum JavaScript API*. Accessed: Apr. 5, 2024. [Online]. Available: https://github.com/web3/web3.js
[61] *Web3.JS—Ethereum JavaScript API*. Accessed: Apr. 5, 2024. [Online]. Available:https://web3js.readthedocs.io/
[62] S. King. *GitHub—Steveukx/GIT-JS: A Light Weight Interface for Running Git Commands in Any Node.JS Application*. Accessed: Apr. 5, 2024. [Online]. Available: https://github.com/steveukx/git-js
[63] P. J. Leach, M. Mealling, and R. Salz, "A universally unique identifier (UUID) URN namespace," in *Proc. RFC*, vol. 4122, 2005, pp. 1–32.
[64] ConsenSys Softw. Inc. (2022). *Ganache—Truffle Suite*. Accessed: Apr. 5, 2024. [Online]. Available: https://trufflesuite.com/ganach
[65] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos, "On omitting commits and committing omissions: Preventing Git metadata tampering that (re) introduces software vulnerabilities," in *Proc. 25th Secur. Symp.*, 2016, pp. 379–395.
[66] L. Gershon and T. Folkman. *Unverified Commits: Are You Unknowingly Trusting Attackers' Code?*. Accessed: Apr. 5, 2024. [Online]. Available: https://checkmarx.com/blog/unverified-commits-are-you-unknowingly-trusting-attackers-code/
[67] (2022). *Git Tools—Rewriting Hist*. Accessed: Apr. 5, 2024. [Online]. Available: https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History
[68] H. D. Francesco. (2018). *Change Date a Git Commit*. Accessed: Apr. 5, 2024. [Online]. Available: https://codewithhugo.com/change-the-date-of-a-git-commit/
[69] Stackoverflow.com. (2014). *How Can I Undo Pushed Commits Using Git?*. Accessed: Apr. 5, 2024. [Online]. Available: https://stackoverflow.com/questions/22682870/how-can-i-undo-pushed-commits-using-git

LUCA GRILLI received the Ph.D. degree in information engineering, in 2007. He is currently an Assistant Professor with the University of Perugia, where he teaches data security and blockchain in the Computer Engineering and Robotics degree course. He has started teaching a doctoral module on blockchain and DLT technologies as part of the Ph.D. course in industrial and information engineering with the University of Perugia. His research interests include network visualization, information visualization, computational geometry, software design and experiments, and information security. In these research areas, he collected about 40 publications and served as an external referee for international conferences and journals. He has been involved in several national and international research projects, including projects supported by the Italian Ministry of Education and the European Community. From May 2009 to November 2014, he was a co-founder member of the academic spin-off Vis4 Srl. He has recently taken an interest in the blockchain and cryptocurrency phenomenon. He has been involved in blockchain related projects. He was a member of the organizing committees of international conferences.

PAOLO SPEZIALI received the B.S. degree in computer and electrical engineering from the University of Perugia, where he is currently pursuing the M.S. degree in computer engineering and robotics. His research interests include blockchain programming, causal inference, and data science.

Open Access funding provided by 'Università degli Studi di Perugia' within the CRUI CARE Agreement