This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2023.3281263

DASLog: Decentralized Auditable Secure Logging for UAV Ecosystems

Roozbeh Sarenche, Farhad Aghili, Takahito Yoshizawa, and Dave Singelée

***Abstract*—Rapid technological advancements in Unmanned Aerial Vehicles (UAVs) have revolutionized intelligent platforms such as smart cities. These advancements have paved the way for an emerging class of Internet of Things (IoT) systems called the Internet of Drones (IoD), which has noteworthy security and privacy challenges. In this paper, we tackle the problem of secure logging and design a novel secure logging scheme – DASLog – for the use case of aerial transport of (medical) goods via drones. Our logging scheme provides public auditability of logging records in the setting where all logging components are managed by a single entity. Our secure logging system relies on hash chains and a Merkle tree to generate proofs for stored logging records. These proofs get written on a private blockchain and can be used later by data consumers to verify the integrity and completeness of a set of logging records. We demonstrate the feasibility of our approach via a proof-of-concept prototype relying on Hyperledger Besu and implemented on multiple Amazon EC2 instances. The performance evaluation of our demonstrator shows that up to 8000 logging records per second can be processed.**

***Index Terms*—IoT, UAV, secure logging, blockchain, Hyperledger Besu**

I. Introduction

A. Logging in UAV systems

Nowadays, there are various emerging use cases of Internet of Things (IoT) technology, such as the Internet of Vehicles (IoV), the Internet of Drones (IoD), and the Industrial Internet of things (IIoT). Managing cybersecurity risks is a complex task for many organizations using IoT devices. Popular frameworks such as the NIST Cybersecurity Framework [1] help to identify the necessary security and privacy controls for information systems. One popular and important security control is the secure implementation of logging, where relevant data from daily operations are electronically stored. For example, in drone or Unmanned Aerial Vehicle (UAV) use cases, the logging records typically include UAV flight-related information and all associated events to operate the UAV flight service. Securely storing and managing these logging records is a key enabler for implementing adequate incident response management procedures, for example, to help detecting security violations and flaws or performing forensic analysis.

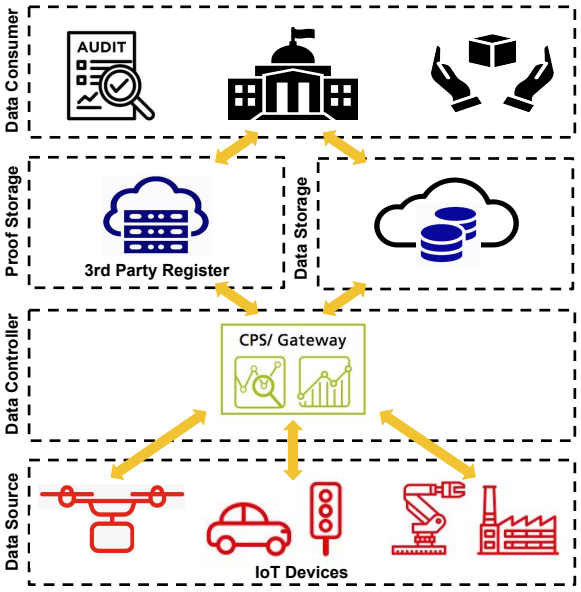


Fig. 1. Logging system in IoT use cases

Figure 1, at a very high level, shows a typical architecture of a logging system for IoT. Going from bottom to top, this figure shows: (i) IoT devices that produce logging data, (ii) a gateway for cyberphysical systems (CPS) that controls and collects data, and forwards it to the cloud, and (iii) an auditor, denoted as a data consumer in our system, downloads log data for auditing. It is also apparent that there exist *proofs* for verifying the correctness and completeness of this log data. Therefore, the gateway also stores these proofs in a dedicated register.

The need and requirements for secure logging are typically well understood by security professionals. For example, it is clear that security-sensitive logging data needs to be encrypted to preserve confidentiality, and backup-ed to ensure availability. Various commercial solutions of secure logging are available (including for IoT); most cloud providers offer secure logging services and multiple stand-alone software tools are present on the market.

B. Motivation

Although commercial security logging solutions are widely available for IoT, they do not almost match well with the

Roozbeh Sarenche, Takahito Yoshizawa, and Dave Singelée are with the imec-COSIC, KU Leuven, Belgium (e-mail: roozbeh.sarenche@esat.kuleuven.be; takahito.yoshizawa@esat.kuleuven.be; dave.singelee@esat.kuleuven.be).
Farhad Aghili is with the Sirris, Leuven, Belgium (e-mail:farhad.aghili@sirris.be), (Corresponding author).

This work was supported in part by CyberSecurity Research Flanders, by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058, and by the ICON project HAI-SCS.

requirements of distributed IoT infrastructures such as UAV systems. One example of IoT systems where there is a mismatch between the state of the art and requirements, are IoT systems where the data sources (i.e., the IoT devices in Fig. 1) and the data controller (i.e., the gateway and cloud storage in Fig. 1) are owned and controlled by the same entity, but where external parties need to verify the authenticity of the logging records (i.e., have *security proofs* such that the external party knows it can trust the data it retrieved from the logging system).

Without loss of generality, we focus in our paper on the case of a UAV system where UAVs are used for the transport of critical (medical) goods from one location to another. All UAVs are controlled centrally by an operator (comparable to an airline company in the case of passenger flights), which is responsible for flight operations and meeting the required safety standards and service level agreements. During these operations, security-sensitive data is generated and stored in logs.

In this use case, there is indeed a need for external parties to access and verify data stored in logging records. For example:

- Usage of logging data to settle disputes with customers about service level agreements.
- Logging data used during audits by government agencies to verify that all safety standards were met.
- Usage of logging data during the investigation of incidents by insurance companies.

The examples above already show the need for adequate security measures to protect this data. Besides conventional security requirements such as confidentiality, integrity, and availability, the secure logging system in place should also offer **public auditability**. Indeed, the logging data could potentially be used as evidence to settle potential disputes. Realizing public auditability is a challenging problem and not offered by conventional secure logging systems. Technical means should be in place to prevent the operator from altering the logging records stored in the secure logging system, deleting certain logging records, or not sending the correct logging records to the external party during an audit. In some cases, also additional security requirements are needed, for example, non-repudiation.

C. Contributions of the paper

The main contributions of our paper are as follows:

- Design of a secure logging system, called DASLog, offering public auditability in the setting where all logging system components are controlled by a single entity (denoted as the operator in the UAV ecosystem). It relies on a private blockchain to provide an immutable and distributed storage service of security proofs.
- Optimization of the DASLog design, allowing to process up to 8000 logging records per second.
- Realization of a proof-of-concept of our proposed secure logging scheme based on Hyperledger Besu, which is a private blockchain platform, and implemented on Amazon EC2 instances.
- Evaluation of the performance of our proof-of-concept implementation and demonstration that our solution can be practically deployed in a UAV ecosystem.

D. Organization of the paper

The rest of the paper is organized as follows. The related work is introduced in Section II. In Section III, we explain the UAV use case which we used as a reference for the design of our secure logging system, including its concepts and terminologies. We also list the main set of security assumptions and security goals in this section. We provide a brief review of blockchain technology and integration of a blockchain in a secure logging system as preliminaries in Section IV. We explain the components of the UTM ecosystem and our secure logging system – DASLog – in Section V. Then, in Section VI, we propose Simple-DASLog as a general solution for implementing a secure logging system for different applications in the IoT network. In Section VII, we present DASLog an improved version of Simple-DASLog that is specifically designed for the UAV ecosystem. We provide a proof-of-concept for our proposed scheme based on Hyperledger Besu private blockchain in Section VIII. We present the security and performance evaluation of the proposed logging system in Section IX. Finally, we present the conclusions and discuss future work in Section X.

II. Related work

The problem of protecting logging records from tampering has been addressed by various works. Schneier and Kelsey presented cryptographic techniques to protect logging records on untrusted storage systems [2]. Holt proposed Logcrypt [3], which provides strong cryptographic assurances that data stored by a logging facility before a system compromise cannot be modified after the compromise without detection. Crosby and Wallach presented novel efficient data structures for tamper-evident logging [4]. Ma and Tsudik [5] proposed an efficient and secure logging approach based on Forward-Secure Sequential Aggregate (FssAgg) authentication techniques. Recently, blockchain is being considered as a security component within logging systems. For example, Rane and Dixit [6] proposed BlockSLaaS: a blockchain-assisted secure logging service for cloud forensics. Putz et al. [7] presented a blockchain-based infrastructure for log integrity preservation that does not depend upon trusted third parties. Paccagnella et al. [8] presented the first kernel-based tamper-evident logging system which addresses the problem of synchronous integrity protection. Ali et al. [9] proposed a blockchain-based log management system where administrators are not able to modify the systems' traces available in audit logs. For more information regarding blockchain-based integrity auditing, readers are referred to [10].

Not all literature focuses solely on tamper-evident secure logging. Various work aims to design secure logging schemes with specific additional security properties. One example is transparency logging. This allows service providers to show that they are compliant with a certain policy that can be

imposed by legislation, sector regulations or internal procedures, or by service level agreements made with customers or subcontractors [11]. Sackmann et al. [12] presented the earliest work on providing transparency of data processing by using cryptographic systems from the secure logging area. Peeters and Pulls proposed Insynd [13], a new cryptographic scheme for privacy-preserving transparency logging. It improves prior work by increasing the utility of the logging data and by relying on a stronger adversarial model.

On the other hand, there exist several research papers that have focused on securing the applications provided in UAV environments using blockchain technologies. In [14], the authors proposed a blockchain-based access control scheme for the UAV system in the IoT environment. Their proposed scheme allows secure communication among the UAVs, and also among the UAVs and Ground Station Servers (GSSs). In [15], the authors proposed a cross-domain authentication scheme for the Internet of UAV systems which is based on blockchain. They employed a local private blockchain to support the registration, authentication, and security audit of UAVs. In [16], the authors proposed an efficient data collection system for UAV-assisted IoT systems based on blockchain. In their proposed system, the UAV provides long-term network access for IoT devices as an edge data collection node. They used blockchain as a distributed, available, and immutable data registry to store collected data from UAVs. The authors in [17] proposed a novel secure access control, data delivery, and collection scheme, which is based on blockchain. Their proposed scheme provides authentication, key agreement, and access control between a UAV and a GSS in each flying zone. The authors describe that their scheme offers more efficiency and better security compared to its predecessors. In [18], the authors proposed a low-latency authentication scheme for UAVs, which is decentralized using blockchain technology. Their proposed scheme is zone-based architecture in a network of UAVs, in which when the system authenticates a UAV in a zone, it is no need to re-authenticate the UAV in the other zones. In [19], the authors present a blockchain-based lightweight authentication service for industrial drones. In this scheme, UAVs can use the smart contracts deployed on the private blockchain to acquire or update the authentication information. In [20], the authors have proposed a blockchain-based scheme to achieve a trustworthy UAV environment. In this paper, a smart contract-based Proof-of-Authentication consensus mechanism is used to verify and validate the communication entities. In [21], the authors provide a review on blockchain for medical delivery in the IoT context. The authors of this survey inspect whether the delivery UAVs for medical applications are suitable for 5G-IoT-assisted blockchain amalgamation. However, in none of the related work above, the need for logging and its security challenges have been discussed.

Our work – DASLog – differs from the state-of-the-art, as this is the first paper – to the best of our knowledge – that aims to design a secure, tamper-evident logging scheme that offers public auditability in the UAV ecosystem where both the internal data sources (i.e., UAVs) and processor (i.e., logging system) are under the control of a single entity (i.e., operator). To realize this property, our work relies on cryptographic hash chains and Merkle trees to generate proofs that are stored in a private blockchain.

III. Logging system for UAV use case

Although our secure logging concept is applicable to any IoT system where all components are controlled and managed by a single entity, in this paper we particularly consider the use case of UAVs that have to transport (medical) items from one customer (e.g., hospital or care center) to another [22]. Below, we briefly elaborate on this use case and the main components and entities in the system.

Critical items, for example, organs or test samples in a healthcare system, often have to be transported from one location (e.g., medical institution) to another one. Ground transport has important limitations, as it is not always the fastest and most reliable option, for example, due to road traffic. Therefore (medical) drone transport is an interesting alternative. In this setting, one or more companies, denoted as operators, control a fleet of UAVs, each with a carriage system beneath. When a customer (e.g., a hospital) makes an order to the operator to transport a (medical) item from A to B, the operator schedules a new UAV flight. Once this is done and approved by the authorities, the UAV will fly to location A, pick up the item, and fly to location B where the item is delivered. During these flights, commands are sent via a wireless channel to the UAV, and UAV data including the UAV's location is received back.

It is evident that such a UAV ecosystem is heavily regulated. Similarly, as for other air traffic, each UAV flight plan has to be approved by the UTM (Unmanned Aircraft System Traffic Management). Besides providing authorizations for flights, the UTM is responsible for real-time monitoring of all unmanned air traffic. Regulations also require operators to be licensed to operate medical drone flights and to follow strict (safety) procedures. Operators that do not meet these requirements risk getting fined or even losing their licenses. Operators will have an insurance to cover their financial losses in case of an accident, assuming that the operator followed all necessary procedures. Finally, operators will negotiate specific service level agreements with their customers, as the safe and timely delivery of some goods is crucial (e.g., in the healthcare system, transporting an organ under the wrong cooling conditions or not within a specific time window creates the risk that the organ can no longer be used). Failing to meet these agreements could result in financial compensation to be paid to the customer.

Data and more particularly logging data is crucial during all the processes mentioned above. Without trustworthy logging data, it is difficult to assess whether all safety procedures and regulations have been correctly followed or to identify the root cause of any safety incident. Moreover, logging data is also needed to settle any potential disputes with customers. Therefore, there is a need for a secure logging system where logging data can be securely stored, is only accessible to authorized entities, and offers security guarantees to external parties that the logging data can be fully trusted.

Intuitively, one might think of using a decentralized blockchain to tackle this problem. However, this would cause latency and reliability problems, as all system entities would have to directly forward all their logging records as transactions to the blockchain. Thus, it is more reliable and efficient to have a logging data controller that collects logging records, stores them securely, and forwards proofs of the records to the blockchain. In the UAV use case, this is a challenging problem, as the logging system (i.e., the logging data controller) and the UAVs are all under full control of the operator, which is not necessarily trusted by the external parties requesting read access.

A. Definitions and notations

In this section, we present the concepts, terminologies, and notations used in our proposed secure logging system, along with the security assumptions and goals for the UAV use case. Table I captures the key acronyms and their definitions used throughout the paper.

TABLE I
ACRONYM LIST

Actonym	Full Name
BFT	Byzantine Fault Tolerance
CFT	Crash Fault Tolerance
DLT	Distributed Ledger Technology
GSS	Ground Station Servers
IoD	Internet of Drones
IoV	Internet of Vehicles
UAV	Unmanned Aerial Vehicle
USP	UTM Service Provider
UTM	Unmanned Aircraft System Traffic Management

Below, one can find the main terminology used in the rest of the paper.

1) Logging information: The content stored in the logging system is organized in a hierarchical fashion, as shown in Figure 2.

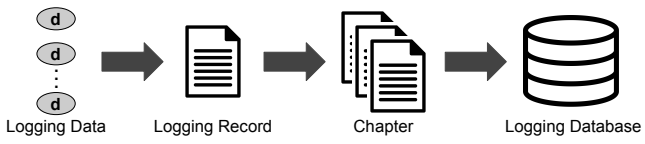


Fig. 2. Log data hierarchy

- **Logging data:** Devices such as UAVs generate individual pieces of data that need to be logged.
- **Logging record:** One or more logging data are combined to form a logging record. A record consists of a meaningful set of information combined together as a unit to be stored in the logging database (see Section VII-B for more details about logging record structure). Therefore, write operations to the logging database are done at the unit of a logging record.
- **Chapter:** A chapter is a collection of one or more logging records that logically belong to each other. One can consider a chapter as a specific view of the logging records. Therefore, reading operations to the logging database is done at the unit of a chapter. A typical example of a chapter is all logging records associated with a single UAV flight from hospital A to B.
- **Logging Database:** All logging records are stored permanently in the logging database.

2) System entities:

- **UTM:** The UAV Traffic Management (UTM) system is a set of automated services that facilitates data exchange between UAVs and other entities such as UAV operators, UAV surveillance sensors, the air traffic management system, etc., to ensure a safe and efficient flight for a considerable number of UAVs [23].
- **UTM Service Provider (USP):** The USP is in charge of monitoring the whole UTM system and is responsible for UAV registration and detection, pre-flight planning, real-time collision avoidance, etc. To monitor the UAV ecosystem, the USP is in need of persistent surveillance information, which is collected using tools such as radars and surveillance sensors [24].
- **UAV Operator:** A UAV operator is an entity that owns a set of UAVs and provides a specific service for its customers, e.g., in the healthcare system, operators perform medical parcel delivery among hospitals. The operator is responsible to manage and monitor the flights of its own UAVs.
- **Data Source:** An entity that generates logging data. This could be an internal entity that is controlled and managed by the operator (e.g., a UAV or any other system component), or an external entity not under the control of the operator (e.g., a hospital).
- **Data Consumer:** Any external entity requesting to read logging records from the secure logging system is denoted as a data consumer.

3) Types of logging data:

- **External logging data:** This is logging data that is generated by an external data source, i.e., not under the control of the operator.
- **Internal logging data:** This is all the logging data that is generated by internal data sources. An example of such type of data is flight data, e.g., location, altitude, and speed of a UAV.

B. Security assumptions

In this section, we list the main set of security assumptions upon which our solution is based.

1) The logging system and its operation is under the full control of the operator. This means that the operator can write/read all the logging records into/from the system and has full control over what is stored in the logging database. This implies that the operator may try to manipulate or alter parts of the logging records, or omit certain logging records during the write and read processes.
2) All internal entities in the UAV ecosystem are managed by the operator and therefore assumed to be under the full control of the operator. Moreover, it is assumed that the operator and all internal entities controlled by the operator are not compromised by an external adversary.

Countermeasures to prevent the compromise of any (software) component or entity in the system by external adversaries are outside the scope of this paper.

3) A secure TLS [25] connection is available between all the components in the logging system and their interaction with the entities in the UAV ecosystem. Therefore, any external adversary cannot read or successfully alter logging records when these are sent from one entity to another.
4) There are criteria in place on which data to log, how to combine logging records into chapters, and how to define the start and end of a chapter. These criteria are chosen by the operator and are assumed to be publicly known to all data consumers. The exact definitions of these criteria are outside the scope of this paper.
5) Read requests to the secure logging system by a data consumer are done at the granularity of a chapter.
6) Security policies are in place to define which data consumer is authorized to access which specific set of logging records. Different access control rules may be applied against data consumers. Similarly, as read requests, security policies are defined at the granularity of chapters.

C. Security goals

A secure logging system should achieve the following security goals [9], [26]:

Confidentiality: Only authorized data consumers can access the content of the logging records in plaintext.

Integrity: Logging records should not be altered or deleted. Note that the term integrity applies not only to the individual logging records but also to the set of logging records as a whole, i.e. chapters in our secure logging system. There exist attacks such as a re-ordering attack and a truncation attack that target the integrity of the whole logging record set. In the re-ordering attack, the attacker tries to manipulate the order in which the logging records have been processed and stored. In the truncation attack, the attacker tries to remove some of the logging records that belong to the same chapter. The attacker may truncate the logging records at the beginning or middle of a chapter (non-tail logging records) or a series of consecutive logging records at the end of a chapter (tail logging records) [26].

Availability: The logging records and their integrity proofs should be available upon request.

Immutability: One should not be able to alter the proofs that preserve the integrity of logging records.

Non-repudiation: The data sources should not be able to deny creating a specific log record.

Public auditability: All data consumers can verify whether or not the received logging records are correct and complete without the need for a **trusted** third party.

Privacy: The process of storing logging records and their proofs should not reveal sensitive information regarding the log contents nor regarding the data sources.

IV. Preliminaries

A. Blockchain

A blockchain is a type of Distributed Ledger Technology (DLT) that provides a distributed peer-to-peer system for storing data without any intermediation from a central authority [27]. The blockchain network consists of multiple users who all have access to the same blockchain ledger. In contrast to the centralized storage systems, in a blockchain network, multiple users can participate in the process of verifying data and writing it on the ledger. In blockchains, data is stored in a chain of consecutive blocks, where any new block contains an immutable cryptographic hash of the previous block to connect them together. Since blocks are connected to each other using a cryptographic hash function, the data recorded on the blockchain cannot be altered, deleted, concealed, or falsified [28].

In a general categorization, blockchains can be divided into two groups: public blockchains and private blockchains. In a public blockchain, anyone can join the blockchain network and contribute to extending the blockchain. However, in a private blockchain, only those users who are authorized by the current participants or validators of the network are allowed to be a part of it [29].

The fundamental part of each blockchain is the consensus mechanism upon which the blockchain is built. In simple words, a consensus mechanism is a mechanism used by the blockchain participants to agree upon a unified ledger without the help of a central authority [30]. The most famous consensus mechanism used in public and permissionless blockchains is called Proof-of-Work (PoW). In a PoW-based blockchain, participants need to solve a cryptographic puzzle to generate a new information block and extend the blockchain. Since solving the cryptographic puzzle needs to consume a considerable amount of energy, PoW-based blockchains can defend against the Sybil attack. However, Sybil attacks are not a threat to private blockchains because only authorized participants are allowed to join the blockchain network [31]. Since we have more levels of trust in private blockchains compared to public blockchains, expensive consensus mechanisms such as PoW are less favorable to be implemented in private blockchains. The consensus mechanism built using the classical fault-tolerant consensus problem, which has been extensively studied in distributed systems since the late 1970s and recently gained popularity in the blockchain community, can be a proper choice to be implemented in private blockchains [30]. Compared to PoW-based blockchains, participants use much less computational power in the fault-tolerant consensus mechanisms. Compared to other recently-emerged proof-based consensus mechanisms such as Proof of Stake, Proof of Space, etc. [32], there is no need for participants in fault-tolerant consensus mechanisms to stake cryptocurrencies or own huge storage space.

The fault-tolerant consensus mechanisms can be divided into two groups: crash fault tolerance (CFT) and byzantine fault tolerance (BFT). CFT mechanisms can only withstand crash failures, where a crashed node simply stops executing any operations. In other words, CFT mechanisms can be

properly implemented in a blockchain network that is built upon this assumption that all the network participants are either online and following the consensus protocol or they are offline and do not participate in the consensus mechanism. In contrast to CFT mechanisms, BFT mechanisms can tolerate Byzantine failures, i.e., arbitrary failures that could be caused by software bugs, hardware errors, and malicious attacks. The BFT-based blockchains are designed in a way that even if there exist a few malicious participants, the honest participants can reach an agreement upon the new blocks that extend the blockchain. In a blockchain network comprising n participants, the BFT-based protocols are able to typically tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ faulty participants [33].

B. Blockchain in secure logging systems

Since blockchain technologies can provide us with an immutable and distributed storage service, they can be used to implement a secure logging system that satisfies the integrity of logging records, non-repudiation, and public auditability. Blockchain immutability means that recorded data on the blockchain ledger cannot be manipulated or modified after being accepted by the blockchain participants [34]. Using a blockchain platform to implement a secure logging system, one should note that integrity of logging records and non-repudiation property of the logging system depend on the immutability of the implemented blockchain. Once immutability is breached, a malicious user can rewrite the blockchain history, and thus, deny the validity of logging records previously stored on the blockchain. Even if logging records are signed, the malicious user can remove some of the logging records and add some new logging records to the database. Although most blockchain platforms are claimed to be immutable, there exist some attacks that can threaten blockchain immutability. As a result, to use a specific blockchain platform in logging applications, one should take a closer look at the assumptions on which the immutability of the blockchain is based to see whether there are any feasible attacks that can break blockchain immutability or not.

There already exist some well-known public blockchains whose ledger has remained immutable over the past few years. Although public PoW-based blockchains can provide a relatively more secure ledger compared to private blockchains, there exist some limitations such as relatively low transaction throughput, long transaction confirmation time, and high transaction fees [35] that hinder public blockchains to be used in logging applications. In CFT-based private blockchains, if just one of the validators decides to behave maliciously or if the private key of one of the validators is leaked, the malicious validator or the attacker can generate a new fork of the blockchain ledger. As a result, a CFT-based blockchain is not a proper choice for implementing the logging applications since a malicious validator can generate multiple forks each containing different logging records. The BFT-based private blockchains are designed in a way that can tolerate malicious validators up to a specific threshold, and thus, can be used to implement auditing applications provided that we can make sure that a malicious user, i.e., a user who has benefits to change the history of logging records, has only control over a minority of validators.

C. Approaches to implementing blockchain-based logging systems

In a general categorization, we can divide blockchain-based logging systems into two groups:

- The logging systems in which all the logging records are stored on the blockchain.
- The logging systems in which the logging records are stored in a secure database and only a small amount of information denoted as integrity proofs are stored on the blockchain.

Although the first approach provides a high level of record availability, it causes latency issues for the logging system. Note that the blockchains that are implemented in IoT or IoD networks need to provide service for a considerable number of entities. The transaction throughput of BFT-based blockchains is in the range of a few hundred transactions per second. However, in the UAV ecosystem where operators need to handle multiple simultaneous flights, the data that needs to be processed may exceed thousands of logging records per second. Therefore, In scenarios where a huge number of logging records are generated, writing all the logging records on the blockchain would be impractical.

In the second approach, a gateway collects the logging records, stores them in a secure central database, and writes the integrity proofs of the logging records on the blockchain. Since the size of proofs is much less than the actual logging records, a lower communication and storage overhead is imposed on the blockchain in the second approach.

D. Hash chain

A hash chain is a successive application of a cryptographic hash function on data. For example, in equation $c = h(h(h(x)))$ the value c contains the result of a hash function being applied three times in a successive manner on x. One of the prominent use of a hash chain is to generate a one-time key or password. There are multiple ways hash chains can be organized. For instance, a hash chain can be structured in the form of a binary tree. In this case, the concatenation of two hash values of two child nodes becomes the input to the hash function of the parent node immediately above them. A well-known such structure is called *Merkle Tree*.

E. Merkle tree

A Merkle tree is a data structure that commits a set of data or information using a cryptographic operation, thus ensuring the integrity of these data. It uses a binary tree, or a binary hash chain, to organize data blocks with each of the nodes containing the hash value in an organized fashion. Each leaf node at the bottom of the tree holds a cryptographic hash of a specific data block. Then all non-leaf nodes contain a hash of its child nodes, thus subsequently forming a cascading chain of hash values. This way, the top (root) of the tree represents the commitment of the entire data structure represented in

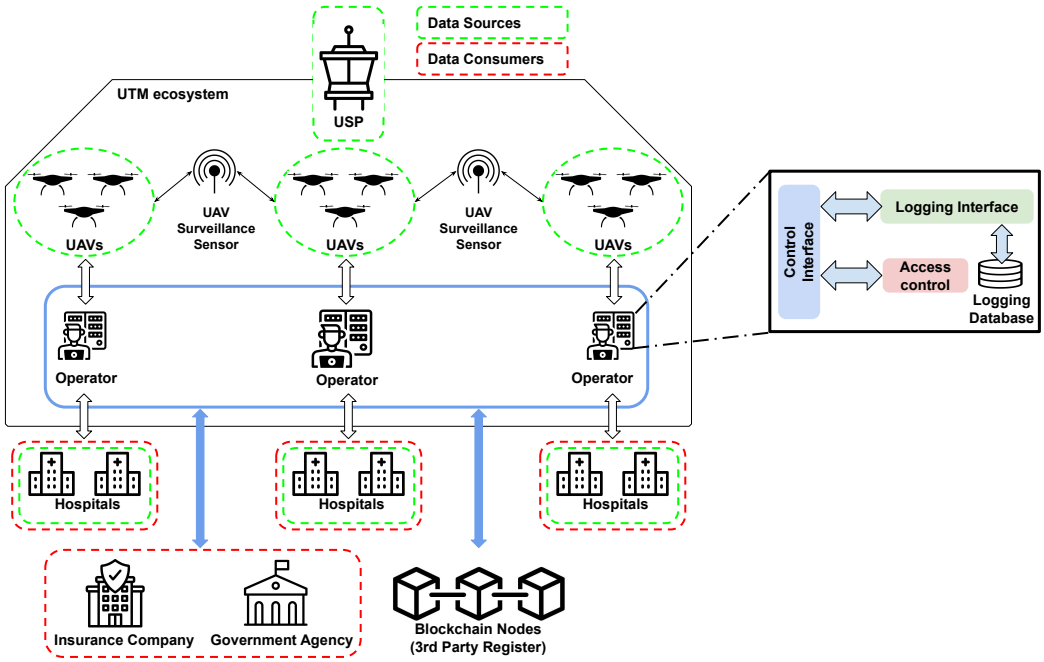


Fig. 3. Secure Logging System Overview for UAV

the aggregation of leaf nodes. Merkle tree, as a tree of hash values, enables efficient verification of the integrity of the data content.

V. DASLog Secure Logging Design

In the UTM ecosystem, there exist two main components called USP and UAV operators.

USP: In our scheme, we assume there is a single UTM service provider (USP) that is in charge of monitoring all the UAV flights performed by multiple operators. The UAV ecosystem is equipped with a set of surveillance sensors that can collect information regarding UAV positioning, navigation, and tracking. This surveillance information is not only important for real-time monitoring of UAVs but also can help detect any conflicts in the internal logging records generated by UAVs, as mentioned before. There is a secure communication channel that connects each UAV, surveillance sensor, and operator to the USP.

UAV Operators: In our scheme, we assume there exist multiple operators that provide a specific service (e.g., medical parcel delivery) to their customers. Each operator owns a set of UAVs and covers a set of customers.

A. DASLog system overview

As depicted in Figure 3, there are four categories of entities in DASLog: (1) central logging system managed by the UAV operator, (2) data source, (3) data consumer, and (4) 3rd party register. We discuss the high-level functionalities of each category below.

Central Logging System managed by the UAV Operator: In our scheme, each UAV operator is equipped with a central logging system. For the sake of convenience, we may use the name operator interchangeably with the central logging system managed by the operator in the rest of the paper. The central logging system consists of four different entities, as shown in Figure 3: Control Interface, Logging Interface, Access Control, and Logging Database.

- **Control Interface:** The control interface interacts with internal and external entities for the purpose of managing and controlling the secure storage and retrieval of logging information. It accepts logging records that are generated and sent by data sources and forwards them to the logging interface for further processing. For retrieval of existing logging information from the logging database, it accepts requests from a data consumer and forwards it to the logging interface when the access control component accepts the read request.
- **Logging Interface:** For storage of new logging information in the logging database, the logging interface accepts logging records from the control interface, encrypts them, and stores them in the logging database. For retrieval of existing chapters from the logging database, the logging interface accepts read requests from data consumers via the control interface, retrieves the relevant chapters, decrypts them, and returns them to the requesting entity. The logging interface is also responsible for interacting with the external 3rd party register to store auxiliary information, associated with the overall integrity of the logged information, denoted as "**proofs**". This interaction with the 3rd party register realizes the public auditability property, as will be discussed later in the paper.
- **Access Control:** The main purpose of the access control component is to decide whether a data consumer is authorized to retrieve specific logging records. This component contains the access control policies defined by the administrator of the logging system. During a read operation, the access control component receives a query from the control interface. This query contains a set of attributes, which fall into two categories: (1)

object attributes, related to the logging records (i.e., identification and type of the logging record), and (2) subject attributes, which are the properties of the data consumer.

- **Logging Database:** The logging database is a database that stores encrypted logging records received through the logging interface.

Data Source: The data source generates the logging records that need to be stored in the central logging system. Upon generation, the logging records are sent to the control interface component. In our scheme, data sources can be either under the control of the operator such as UAVs, or external entities such as the USP and customers.

Data Consumer: The data consumer is the entity that requests one or more chapters from the central logging system. When the data consumer is authorized to read the requested chapters, the central logging system will fetch and decrypt these chapters from its logging database and send the result to the data consumer. Moreover, it will also provide the necessary pointers (to the proofs stored in the 3rd party register, see further) that enable the data consumer to verify the correctness and completeness of the received chapters (i.e., all the logging records that belong to the requested chapter are sent to the data consumer, and these are all legitimate logging records).

3rd Party Register: Since the whole central logging system is managed by the operator, one needs an additional component to ensure public auditability. This entity is denoted as the 3rd party register. It provides a particular storage service to the central logging system. More specifically, all data written into the 3rd party register can no longer be altered or deleted once it is written. All data consumers have read access to the 3rd party register and can read any information stored on it. In our secure logging system, rather than storing all the logging records on the 3rd party register, only a small piece of information, denoted as **proof**, is stored on the 3rd party register. At the writing time, the operator generates a proof for each set of logging records and stores it on the 3rd party register. Later, at reading time, data consumers can use the stored proofs to verify whether or not the retrieved chapters are correct and complete. The 3rd party register that is implemented in our scheme should satisfy two important properties:

(1) Availability: all the data consumers should have access to the 3rd party register whenever needed.

(2) Immutability: once a proof is written on the 3rd party register, it should no longer be possible to modify or delete the proof in the future.

In our scheme, we have used a **private blockchain** to realize the security properties of the 3rd party register.

B. Private blockchain as a 3rd party register

To implement a 3rd party register in our secure logging system, we have used a BFT-based private blockchain. A private blockchain network consists of multiple authorized users who have the right to write/read the information on/from the blockchain ledger. Users in the private blockchain can play the role of either validator nodes or non-validator nodes. The validator nodes should participate in processing the consensus mechanism to generate new information blocks. However, the non-validator nodes only have access to the blockchain information and do not participate in extending the blockchain. In order to preserve immutability in our BFT-based private blockchain, we need to make sure that only a minority of validators behave maliciously. As already mentioned, the operator is responsible to store the proofs of logging records on the private blockchain. Once the proofs are stored, a malicious operator should not be able to modify them. Therefore, the number of validators under the control of the malicious operator should be less than 33% of the whole number of validators. To achieve this we need to distribute the role of blockchain validators among different entities in DASLog, such as customers, the government agency, insurance companies, and UAV operators. In this case, one has the guarantee that the malicious operator cannot modify an existing proof on the blockchain. In the following, we briefly mention the main reasons to choose a BFT-based private blockchain to implement the 3rd party register in our scheme:

(1) A private blockchain can provide a ledger to which data consumers can easily have access at the reading time. If a few nodes get offline, the data consumers can still fetch the proofs from the blockchain and verify the logging records.

(2) Since there exists a strong regulation on authorizing the entities that can participate in the UAV ecosystem, a private blockchain should be used to implement the 3rd party register to prevent unauthorized users from compromising the blockchain information.

(3) A private blockchain enjoys high throughput, high transaction speed, and zero-fee transaction support.

(4) A BFT-based private blockchain can satisfy the property of immutability even in the presence of a malicious operator.

VI. SIMPLE-DASLOG SOLUTION

In the simple version of DASLog solution (Simple-DASLog), we do not impose any assumptions on the logging record structure. In other words, we let logging records have any arbitrary formats. Therefore, Simple-DASLog not only can be applied to the UAV ecosystem but can be used to implement a secure logging system for different IoT applications. In Simple-DASLog, to achieve a secure logging system, we need to consider the following security assumption: *The operator is assumed to be honest at the time of the initial write process of logging records, i.e., when storing a new logging record in the logging database.* This implies:

1) all logging records and their constituting data are valid and correct (i.e., no fake data is being generated).
2) all logging records and their data are complete (i.e., no data is deliberately omitted during the writing process).

However, the operator may not be honest during the reading process of the logging records, i.e., when retrieving logging records from the logging database. This implies that when a data consumer requests to read a specific set of logging records, the operator may try to manipulate or alter parts of the retrieved logging records, omit certain logging records, or even send fake logging records to the data consumer instead of legitimate logging data.

In Section VII, by imposing a specific frame structure on the logging records, we no longer need the aforementioned security assumption of Simple-DASLog.

There are three main steps in the lifetime of a logging record: (1) writing the logging record into the logging system, (2) reading one or more logging records, and (3) verification of the correctness and completeness of the logging records that were fetched from the logging system.

A. Write operation in Simple-DASLog

For each mission that takes place under the supervision of the operator, e.g., a flight operation for parcel delivery, a new chapter is created. All the logging records that are generated during the mission will be added to the created chapter. The first and the last logging records of each chapter respectively declare the start of a new chapter and the ending of that chapter. Each chapter C is uniquely identified by a unique identification number, i.e., C_{ID}. We assume C_{ID} of each chapter is publicly available to all data consumers. During the write operation, the logging records are processed and stored in a central database. Besides, a set of proofs is generated and stored on the private blockchain to preserve the integrity of the logging records. The detailed steps of the write operation are as follows:

(1) We use a smart contract to write the proofs and other auxiliary information on the private blockchain. Our smart contract includes three main functions called the **initializing function**, the **proof function**, and the **finalizing function**. In DASLog, time is divided into epochs whose length is chosen based on the secure logging system specifications, such as the block generation time of the private blockchain used to implement the 3rd party register (see later in this paper). The epoch length can be set in the range of a few seconds. Let e_{start} represent an epoch in which the chapter C_{ID} starts. At the end of epoch e_{start}, the logging interface component in the operator uses the initializing function to write the following message on the blockchain:

$$M^{\text{Init}}_{C_{\text{ID}}} = \Big\{ C_{\text{ID}}, e_{\text{start}} \Big\} \tag{1}$$

$M^{\text{Init}}_{C_{\text{ID}}}$ proves that the operator has accepted the responsibility of the mission and cannot deny performing such a mission in the future. We assume that the first logging record of the chapter, i.e., L_1, which contains general information regarding the mission, is the only logging record created in epoch e_{start}.

(2) A data source sends its logging record L_i to the control interface component of the operator. Here, the index i declares that L_i is the i^{th} logging record of its chapter. Note that the data source and the control interface have no idea about the index of the logging records, and the index i is just used for the sake of convenience in representing further relations.

(3) Upon receiving a logging record, e.g., L_i, the control interface first finds its corresponding C_{ID}, i.e., specifies to which chapter the logging record belongs. Note that it is possible that the same operator performs several missions simultaneously, where each of these missions has its own C_{ID}. Then, the control interface sends the tuple (C_{ID}, L_i) to the logging interface component in the operator.

(4) Once the logging interface receives the tuple (C_{ID}, L_i), it should store the encrypted version of the received logging record, i.e., $\mathsf{Enc}_{k_1}(L_i)$, in the logging database. Here, $\mathsf{Enc}_{k_1}(\cdot)$ denotes a symmetric encryption scheme, and k_1 is the symmetric encryption key that is securely stored in the logging system. Since in our scheme, the encrypted version of the logging records is stored in the database, even if an attacker gets physical access to the logging database, he/she cannot get the content of the logging records.

(5) Prior to sending the encrypted logging record to the logging database, the logging interface uses the relation $C_{\text{PID}} = \mathsf{HMAC}_{k_2}(C_{\text{ID}})$ to generate a pseudo identifier of the chapter. Here, k_2 is the HMAC function key that is securely stored by the operator. Note that all logging records that belong to the same chapter have the same C_{PID}. The reason for storing encrypted logging records in the database with their pseudo identifier instead of their public factor C_{ID} is because if the logging database gets compromised, no identifying metadata would be leaked. Once C_{PID} is generated, the logging interface forwards the tuple $(C_{\text{PID}}, \mathsf{Enc}_{k_1}(L_i))$ to the logging database component in the operator, where the tuple is immediately stored as a new array.

(6) In addition to storing encrypted logging records in the logging database, the logging interface performs the task of proof generation in parallel. At the end of each epoch, the logging interface fetches all the logging records received during the epoch. Note that the operator may handle several missions simultaneously, and thus, there may exist several ongoing chapters in one epoch. Since the collected logging records during the epoch can belong to different chapters, the logging interface distributes the logging records among separate logging record sets, where each set is assigned to one of the chapters. We use index e to represent the current epoch. Let $\mathcal{L}^e_C$ denote the set of logging records that belongs to chapter C and is collected by the logging interface during the e^{th} epoch. If the operator receives no logging record for the ongoing chapter C in epoch e, we use the representation $\mathcal{L}^e_C = \emptyset$. In the case that $\mathcal{L}^e_C$ is not an empty set, let $\mathcal{L}^e_C = \{L_{q+1}, L_{q+2}, \cdots, L_{q+n}\}$ be the set of n logging records that belongs to chapter C and is collected by the logging interface during the e^{th} epoch. The fact that the index of logging records in $\mathcal{L}^e_C$ starts from $q+1$ declares that logging records $L_1, L_2, \cdots, L_q$ of chapter C have been processed in previous epochs. There exist two types of proof in Simple-DASLog: the **single proof** and the **hash-chain proof**. For each logging record $L_i \in \mathcal{L}^e_C$, the single proof $P^{\mathsf{single}}_{i,C}$ is calculated using the relation $P^{\mathsf{single}}_{i,C} = \mathsf{H}(L_i)$, where $\mathsf{H}(\cdot)$ is a cryptographic hash function. In contrast to the single proof, where a separate proof is generated for each logging record, only one hash-chain proof, i.e., $P^{\mathsf{hash\text{-}chain}}_{e,C}$, is generated for all the logging records in $\mathcal{L}^e_C$. At the end of each epoch, a new hash-chain proof is calculated for each ongoing chapter. The hash-chain proof of chapter C in epoch e can be calculated using Algorithm 1. Since the hash-chain proof is generated using a recursive algorithm, for calculating the hash-chain proof of a specific chapter in epoch e, the logging interface needs to have the hash-chain proof of the same chapter in

Algorithm 1 Hash-chain proof calculation in Simple-DASLog

$P^{\text{hash-chain}}_{e_{\text{start}},C} \leftarrow \mathsf{H}(C_{\text{ID}}||L_1)$ ▷ Hash-chain proof calculation of the starting epoch
for $i = e_{\text{start}} + 1$ **to** e **do** ▷ A loop for hash-chain proof calculation of the e^{th} epoch
  if $\mathcal{L}^i_C = \emptyset$ **then** ▷ In the case that $\mathcal{L}^i_C$ is an empty set.
    $P^{\text{hash-chain}}_{i,C} \leftarrow \mathsf{H}(P^{\text{hash-chain}}_{i-1,C}||\mathsf{Null})$ ▷ Null is a specific message indicating that $\mathcal{L}^i_C$ is empty.
  else ▷ In the case that $\mathcal{L}^i_C$ is not empty: $\mathcal{L}^i_C = \{L_{q+1}, L_{q+2}, \cdots, L_{q+n}\}$
    $v \leftarrow P^{\text{hash-chain}}_{i-1,C}$ ▷ v is a temporary variable to store the latest value of the hash-chain proof.
    for $j = q+1$ **to** $q+n$ **do**
      $v \leftarrow \mathsf{H}(v||L_j)$
    end for
    $P^{\text{hash-chain}}_{i,C} \leftarrow v$
  end if
end for

epoch $e-1$. As a result, in our scheme, the logging interface stores the hash-chain proof calculated in the last epoch in a temporary memory.

Let $\mathcal{P}^e_C = \{P^{\text{single}}_{q+1,C}, P^{\text{single}}_{q+2,C}, \cdots, P^{\text{single}}_{q+n,C}, P^{\text{hash-chain}}_{e,C}\}$ represent the proof set generated for chapter C at the end of epoch e. In total, the logging interface generates $n+1$ proofs for the logging record set $\mathcal{L}^e_C$ in the e^{th} epoch, n single proofs for its n logging records and one hash-chain proof for the whole set. Note that if $\mathcal{L}^e_C = \emptyset$, we have $\mathcal{P}^e_C = \{P^{\text{hash-chain}}_{e,C}\}$. At the end of each epoch, a separate proof set is generated for each of the ongoing chapters. In Section VI-B, we will discuss how these proofs are used by the data consumer to verify the received logging records at the time of reading.

(7) Once all the proofs are generated, the logging interface creates a **Merkle tree** for epoch e, where each leaf represents one of the proofs (the single and hash-chain proofs of all the ongoing chapters in epoch e), and calculates the **Merkle root proof** MR^e. Then, the logging interface extracts the paths for each leaf proof. The path of the single proof for record $L_i \in \mathcal{L}^e_C$ and the hash-chain proof for $\mathcal{L}^e_C$ are respectively denoted by $\mathsf{PATH}^{\text{single}}_{i,C}$ and $\mathsf{PATH}^{\text{hash-chain}}_{e,C}$.

(8) The Merkle root proof MR^e is the only proof in epoch e that gets stored on the private blockchain. There is no need to store single and hash-chain proofs, neither in the logging database nor in the private blockchain, which makes the scheme efficient with low latency. At the end of epoch e, the logging interface uses the proof function in the deployed smart contract to write the following message on the blockchain:

$$M^{\text{Proof}}_e = \left\{\mathsf{MR}^e, e\right\} \tag{2}$$

Once MR^e is written on the blockchain, the logging interface extracts its corresponding transaction address. We use $\mathsf{Addr}^e_{\mathsf{MR}}$ to represent the address of the Merkle root proof MR^e. The address will be used by data consumers to read the Merkle root proof from the blockchain at the time of reading. For each $L_i \in \mathcal{L}^e_C$, the logging interface forwards the tuple $\left(\mathsf{PATH}^{\text{single}}_{i,C}, \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}^e_{\mathsf{MR}})\right)$ to the logging database, where it gets stored within the row data that belongs to L_i. In summary, at the end of the writing process in epoch e, the following array has been stored in the logging database for the logging record $L_i \in \mathcal{L}^e_C$:

$$\left(C_{\text{PID}}, \mathsf{Enc}_{k_1}(L_i), \mathsf{PATH}^{\text{single}}_{i,C}, \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}^e_{\mathsf{MR}})\right) \tag{3}$$

In addition to logging record arrays, there exists a separate array, named hash-chain-proof array, for each chapter in the logging database that stores information regarding the most recent hash-chain proof of the corresponding chapter. At the end of epoch e, the logging interface updates the hash-chain-proof array for chapter C_{ID} as follows:

$$\left(C_{\text{PID}}, \mathsf{PATH}^{\text{hash-chain}}_{e,C}, \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}^e_{\mathsf{MR}})\right) \tag{4}$$

The sequence of the write operation in epoch e is depicted in Figure 4.

(9) Let e_{end} represent an epoch in which the last logging record of the chapter is processed. Similar to the previous epochs, the logging interface performs the task of proof generation for epoch e_{end} and writes $M^{\text{Proof}}_{e_{\text{end}}}$ on the blockchain. Besides,

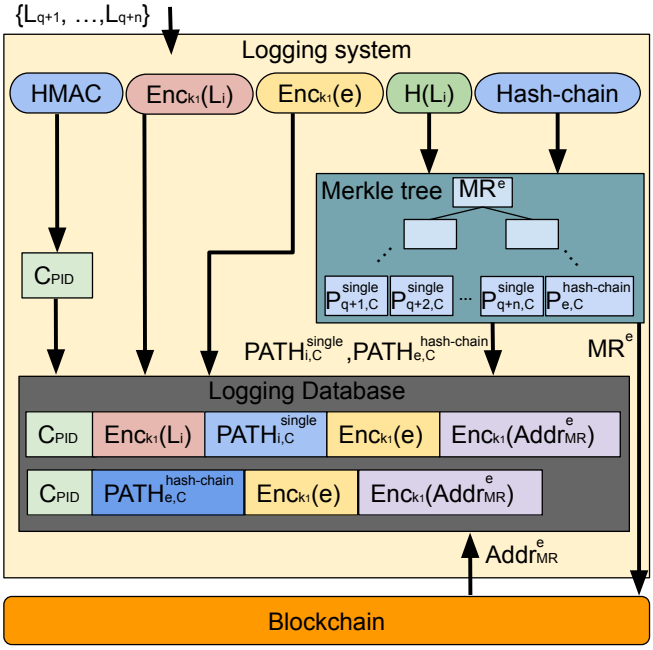


Fig. 4. The sequence of the write operation in epoch e.

the operator uses the finalizing function in the deployed smart contract to write the following message on the blockchain:

$$M_{C_{\text{ID}}}^{\text{Final}} = \Big\{ C_{\text{ID}}, e_{\text{end}} \Big\} \tag{5}$$

$M_{C_{\text{ID}}}^{\text{Final}}$ declares the end of the chapter C_{ID}.

B. Read operation in Simple-DASLog

Recall that read requests are done at the granularity of a chapter. The detailed steps are as follows:
(1) As the first step, the data consumer authenticates itself to an identity and access management (IAM) server and obtains a fresh access token upon successful authentication (see Section VIII-B for more details of the IAM server we used for our proof-of-concept). The authentication process and the structure of the access token are outside the scope of this paper.
(2) Without loss of generality, let us assume that the data consumer wants to access one specific chapter C whose public identifier is denoted by C_{ID}. To do this, after having obtained a fresh access token from the IAM, the data consumer sends a read request for C to the control interface component in the operator. This request consists of two items: $(C_{\text{ID}}, \text{Access Token})$.
(3) Upon receiving a read request from a data consumer, the control interface uses the access token to enforce the access control policies and check whether the data consumer is authorized to retrieve the requested chapter C_{ID}. If the access is denied, the read operation gets terminated. If the access is granted, the control interface forwards C_{ID} to the logging interface.
(4) Upon receiving C_{ID}, the logging interface first generates the corresponding pseudo identifier, i.e., $C_{\text{PID}}^{\mathsf{read}}$, using the relation $C_{\text{PID}}^{\mathsf{read}} = \mathsf{HMAC}_{k_2}(C_{\text{ID}})$. Next, it queries the logging database to retrieve all the arrays belonging to chapter $C_{\text{PID}}^{\mathsf{read}}$.
(5) The logging database will return the array set composed of all the logging records belonging to chapter $C_{\text{PID}}^{\mathsf{read}}$:

$$\Big\{ \Big(C_{\text{PID}}, \mathsf{Enc}_{k_1}(L_i), \mathsf{PATH}_{i\,,\,C}^{\mathsf{single}}, \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^{e}) \Big) \;\Big|\; C_{\text{PID}} = C_{\text{PID}}^{\mathsf{read}} \Big\} \tag{6}$$

Note that in the representation above, the index e represents the epoch in which the corresponding logging record is received and processed, and since the different logging records may be processed in different epochs, the epoch index can vary for different logging records. Let us denote the index of the epoch in which the chapter $C_{\text{PID}}^{\mathsf{read}}$ has ended as e_{end}. In this case, the logging database returns the following hash-chain-proof array for the chapter $C_{\text{PID}}^{\mathsf{read}}$:

$$\Big(C_{\text{PID}}^{\mathsf{read}}, \mathsf{PATH}_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}, \mathsf{Enc}_{k_1}(e_{\text{end}}), \mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^{e_{\text{end}}}) \Big) \tag{7}$$

Using the key k_1 stored in secure memory, the logging interface decrypts all the encrypted logging records $\mathsf{Enc}_{k_1}(L_i)$, the epochs in which the logging records have been generated $\mathsf{Enc}_{k_1}(e)$, and the proof addresses $\mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^{e})$. Then, the logging interface sends the following message to the control interface:

$$\Big\{ \big(L_i, \mathsf{PATH}_{i\,,\,C}^{\mathsf{single}}, e, \mathsf{Addr}_{\mathsf{MR}}^{e} \big) \mid L_i \in C \Big\} \;\|\; \big(\mathsf{PATH}_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}, e_{\text{end}}, \mathsf{Addr}_{\mathsf{MR}}^{e_{\text{end}}} \big) \tag{8}$$

In addition to the logging records, the array above contains the paths, the epochs, and the addresses of all the single proofs as well as only the last hash-chain proof of the chapter.
(6) The control interface forwards this data to the data consumer.

C. Verification phase in Simple-DASLog

Upon receiving the response from the operator, the data consumer needs to verify the correctness and completeness of the received logging records. The verification steps are as follows.
(1) The data consumer needs to verify the only hash-chain proof of the chapter, i.e., $P_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}$. The data consumer uses $\mathsf{Addr}_{\mathsf{MR}}^{e_{\text{end}}}$ to fetch the Merkle root proof of an epoch in which the proof $P_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}$ has been generated from the blockchain. Next, it uses Algorithm 1 and the set of logging records (L_i) received from the control interface to calculate $P_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}$. Then, it uses $P_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}$ and the received path $\mathsf{PATH}_{e_{\text{end}}\,,\,C}^{\mathsf{hash\text{-}chain}}$ to reconstruct the Merkle root. If the computed Merkle root corresponds to the one retrieved from the blockchain, the verification holds. By performing the first verification step, the data consumer can make sure that none of the non-tail logging records has been modified or deleted and that the sequence of received logging records has been preserved.
(2) As the second verification step, the data consumer reads $M_{C_{\text{ID}}}^{\text{Final}}$ from the blockchain and fetches e_{end}. If the epoch in which the last received logging record is processed is the same as e_{end}, the verification holds. Note that to prevent a malicious operator from performing the tail-truncation attack at the reading time, the second verification step is necessary. See Section IX-A for more details regarding the truncation attack.

When these verification steps pass successfully, the data consumer knows the set of logging records it received is correct and complete, i.e., no logging records have been omitted or altered, and the sequence of logging records within the chapter has been preserved. Note that in this case, the verification of the single proofs of each logging record would be redundant and hence not needed. However, if the verification procedure depicted above fails, then the single proofs of the individual logging records can be used to spot which record(s) have been modified or deleted.

VII. DASLog solution

Simple-DASLog was proposed based on the security assumption that the operator is honest during the write operation. In the improved version, which is denoted by DASLog, we introduce a logging record structure that helps us remove the mentioned security assumption. Compared to the Simple-DASLog, DASLog has the following advantages:

1) The operator may not be honest during the write operation.
2) While Simple-DASLog is a general scheme that can be applied to different applications in the IoT network, DASLog is specifically designed for the UAV ecosystem, and thus, considers more details regarding UAV flights.
3) The privacy level is improved compared to the simple version.

The DASLog solution is designed based on the following security assumptions:
1) External logging data is digitally signed by the external data source that generated the data.
2) We assume that the correctness of internal logging data can be verified by external parties (most notably the USP). Evidence of these types of verification by external parties, e.g., flight data monitored by the USP, is digitally signed by this external party and stored together with the logging data. Data consumers are expected to verify these digital signatures when retrieving this type of logging data.

Before we discuss these three main phases (write, read, and verification), we first need to declare how a new chapter starts and specify the logging record structure in DASLog.

A. Start of a new chapter in DASLog

For each flight that takes place under the supervision of the operator, a new chapter is created. All the logging records that are generated during the flight operation will be added to the created chapter. Each chapter C is uniquely identified by a unique identification number, i.e., C_{ID}, which represents the UAV flight number. In the following, we present a scenario as an example to explain how a chapter starts, how a unique identification number is assigned to each chapter, and how the pre-flight information is processed. Note that the writing, reading, and verification steps in DASLog are independent of the starting scenario, and DASLog can be applied to all the scenarios in the UAV ecosystem.

In DASLog, each entity owns a pair of public and private keys. Let $\sigma_{\mathsf{E}}(M)$ represent the signature of entity E on message M created using E's private key. Assume customer A wants to transfer a parcel to customer B. They create a contract $\mathsf{CNT}_{\mathsf{A,B}}$ which comprises necessary information regarding the parcel aerial delivery. This contract consists of four parts: $\mathsf{CNT}_{\mathsf{A,B}} = \mathsf{CNT}^1_{\mathsf{A,B}}||\mathsf{CNT}^2_{\mathsf{A,B}}||N^1_{\mathsf{A,B}}||N^2_{\mathsf{A,B}}$. $\mathsf{CNT}^1_{\mathsf{A,B}}$ includes general information regarding the flight such as the origin, the destination, and the time at which the flight will take place – USP needs to have access to $\mathsf{CNT}^1_{\mathsf{A,B}}$ to arrange the flight operation. $\mathsf{CNT}^2_{\mathsf{A,B}}$ includes privacy-sensitive information such as parcel specifications, and $N^1_{\mathsf{A,B}}$ and $N^1_{\mathsf{A,B}}$ represent two cryptographic nonces. Let the operator O be chosen by customers to operate the flight. Customer A sends the message $\{\mathsf{CNT}_{\mathsf{A,B}}, \sigma_{\mathsf{A}}(\mathsf{CNT}_{\mathsf{A,B}}), \sigma_{\mathsf{B}}(\mathsf{CNT}_{\mathsf{A,B}})\}$ to the operator O, where the message contains the contract and the signatures of customers A and B on the contract. The operator sends the pre-flight request $\{\mathsf{FP}, \mathsf{CNT}^1_{\mathsf{A,B}}, N^1_{\mathsf{A,B}}\}$ to USP, where FP represents the technical description of the flight plan. If airspace requirements are met and there is no conflict with the other flights, USP generates the unique identification of the flight, i.e., C_{ID}, and returns the message $\{C_{\mathrm{ID}}, \sigma_{\mathsf{USP}}(C_{\mathrm{ID}}||\mathsf{H}_1(\mathsf{CNT}^1_{\mathsf{A,B}}||N_{\mathsf{A,B}})), \sigma_{\mathsf{USP}}(\mathsf{FP})\}$ as flight confirmation to the operator O, where $\mathsf{H}_1(\cdot)$ is a cryptographic hash function.

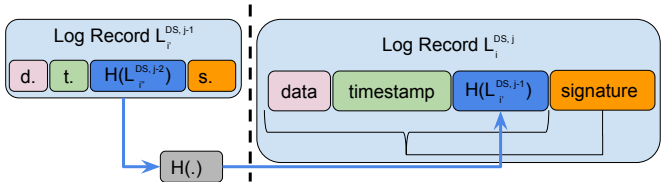


Fig. 5. The structure of logging records generated by the same data source in DASLog

B. Logging record structure in DASLog

Recall that logging records are generated by a data source and that logging records are logically combined into chapters. Therefore, the **first** and the **last** logging records of each chapter respectively declare the start of a new chapter and the ending of that chapter. All the remaining logging records are generated by one of the data sources (represented by DS), namely customers, UAVs, and USP.

A logging record in DASLog is represented by $L_i^{\mathsf{DS},j}$. Here, the index i declares that $L_i^{\mathsf{DS},j}$ is the i^{th} logging record of its chapter. Note that the data source has no idea about the index of the logging records, and the index i is just used for the sake of convenience in representing further relations. The upper index j shows that $L_i^{\mathsf{DS},j}$ is the j^{th} logging record generated by DS in this chapter. Whenever there is no point in referring to the generator of the logging record, we simply use L_i to represent the logging record.

All the logging records, except the first and the last ones, consist of four fields: the actual logging data, a time stamp, the hash of the previous logging record generated by the same data source, and the signature(s). The main part of each logging record is the actual logging data generated by the data source. The format of this data is not important for our solution and hence is out of the scope of this paper. Each logging record contains the time stamp at which the logging record is generated by the data source. Besides, each logging record, e.g., $L_i^{\mathsf{DS},j}$ for $j \geq 2$, created by the data source DS contains the hash of logging record $L_{i'}^{\mathsf{DS},j-1}$, where $L_{i'}^{\mathsf{DS},j-1}$ is the last logging record generated by the same data source DS in the current chapter. Therefore, each data source needs a temporary memory to store the hash of its last logging record. Moreover, the data source also signs all the fields using its unique private signing key. This proposed log record structure is shown in Figure 5.

If DS is a data source that is not under the control of the operator, i.e., DS is either a customer or the USP, the logging record generated by DS has the following format:

$$L_i^{\mathsf{DS},j} = \left\{\mathsf{data}, \mathsf{ts}, \mathsf{H}\left(L_{i'}^{\mathsf{DS},j-1}\right), \sigma_{\mathsf{DS}}\left(\mathsf{data}||ts||\mathsf{H}\left(L_{i'}^{\mathsf{DS},j-1}\right)\right)\right\}, \quad (9)$$

where data is a piece of external logging data, and $\mathsf{H}(\cdot)$ is a cryptographic hash function. Upon generating a new logging

record, the data source DS sends its logging record $L_i^{\mathsf{DS},j}$ to the control interface component in the operator O.

However, if a UAV is the generator of the logging record, the UAV itself doesn't directly send the logging record to the operator. In our scheme, the operator is responsible to equip and control the UAVs, and thus, can find access to the private keys stored inside the drones. If logging records of UAVs were sent directly to the operator, the operator could have forged the UAV logging records. To prevent forging UAV logging records, UAVs send their logging records to USP. Let $RL_i^{\mathsf{UAV},j}$ denote a raw logging record generated by the UAV. We have:

$$RL_i^{\mathsf{UAV},j} = \Big\{ \mathsf{data}^{\mathsf{UAV}}, \mathsf{ts}^{\mathsf{UAV}}, \mathsf{H}\big(RL_{i'}^{\mathsf{UAV},j-1}\big), \sigma_{\mathsf{UAV}}\Big(\mathsf{data}^{\mathsf{UAV}}||\mathsf{ts}^{\mathsf{UAV}}||\mathsf{H}\big(RL_{i'}^{\mathsf{UAV},j-1}\big)\Big)\Big\}, \tag{10}$$

where $\mathsf{data}^{\mathsf{UAV}}$ is a piece of verifiable internal logging data. Upon generating the raw logging record $RL_i^{\mathsf{UAV},j}$, the UAV sends $RL_i^{\mathsf{UAV},j}$ to USP. As already mentioned, we assume that the UTM ecosystem is equipped with drone surveillance sensors that are in contact with USP. These sensors can detect UAVs and collect information regarding the UAV operations such as location, altitude, and speed of a UAV. This information can help USP detect technical conflicts in logging data sent from UAVs. Let $\mathsf{data}^{\mathsf{sensor}}$ denote technical information regarding the UAV flight measured by surveillance sensors at time $\mathsf{ts}^{\mathsf{UAV}}$. Upon receiving $RL_i^{\mathsf{UAV},j}$, USP generates the logging record $L_i^{\mathsf{UAV},j}$ as follows and forwards it to the control interface component in the operator O:

$$L_i^{\mathsf{UAV},j} = \Big\{RL_i^{\mathsf{UAV},j}, \mathsf{data}^{\mathsf{sensor}}, \sigma_{\mathsf{USP}}\big(RL_i^{\mathsf{UAV},j}||\mathsf{data}^{\mathsf{sensor}}\big)\Big\} \tag{11}$$

C. Write operation in DASLog

During the write operation, the logging records are processed and stored in a central database. Besides, a set of proofs is generated and stored on the private blockchain to preserve the integrity of the logging records. The detailed steps of the write operation are as follows:

(1) Similar to Simple-DASLog, in DASLog, we use a smart contract to write the proofs and other auxiliary information on the private blockchain. Let e_{start} represent an epoch in which the chapter C_{ID} starts. At the end of epoch e_{start}, the logging interface component in the operator uses the initializing function of the smart contract to write the following message on the blockchain:

$$M_{C_{\text{ID}}}^{\text{Init}} = \Big\{C_{\text{ID}}, e_{\text{start}}, \mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}), \sigma_{\mathsf{O}}\big(\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}})\big), \mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}}^1), \sigma_{\mathsf{USP}}\big(C_{\text{ID}}||\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}}^1)\big)\Big\} \tag{12}$$

$M_{C_{\text{ID}}}^{\text{Init}}$ is the initializing message that declares the start of the chapter C_{ID}. Customers A and B can listen to the events emitted from the smart contract to find whether the initial message including $\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}})$, i.e., $M_{C_{\text{ID}}}^{\text{Init}}$, is added to blockchain or not. Once $M_{C_{\text{ID}}}^{\text{Init}}$ is written on the blockchain, customers A and B can find access to the corresponding C_{ID} of the chapter and verify the signatures of both operator O and USP. The signature $\sigma_{\mathsf{O}}\big(\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}})\big)$ proves that operator O has accepted the responsibility of the flight and cannot deny performing such a flight in the future. The signature $\sigma_{\mathsf{USP}}\big(C_{\text{ID}}||\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}}^1)\big)$ shows that this flight is authorized and will take place under the supervision of USP.

(2) In our scheme, the operator O is responsible to generate the first and the last logging records of each chapter. The starting log of the chapter contains general information regarding the flight and is created in epoch e_{start} as follows:

$$L_1 = \Big\{\mathsf{CNT}_{\mathsf{A,B}}, \mathsf{FP}, \sigma_{\mathsf{A}}\big(\mathsf{CNT}_{\mathsf{A,B}}\big), \sigma_{\mathsf{B}}\big(\mathsf{CNT}_{\mathsf{A,B}}\big), \sigma_{\mathsf{O}}\big(\mathsf{CNT}_{\mathsf{A,B}}\big), \sigma_{\mathsf{USP}}\big(C_{\text{ID}}||\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}})\big), \sigma_{\mathsf{USP}}\big(\mathsf{FP}\big)\Big\} \tag{13}$$

We assume L_1 is the only logging record created in epoch e_{start}.

(3) As already mentioned in Section VII-B, all the logging records, except the first and the last logging records of each chapter, are generated by one of the data sources. Once a logging record, e.g., L_i, is generated, it is sent to the control interface component in the operator O. The steps that the operator performs in DASLog to process the logging records are almost the same as Simple-DASLog with the following difference:

- Since all the logging records in DASLog are signed by the data sources, there is no need for generating single proofs. Therefore, the proof set of chapter C in epoch e only contains the hash-chain proof, i.e., $\mathcal{P}_C^e = \{P_{e,C}^{\mathsf{hash\text{-}chain}}\}$.
- To encrypt the arrays that are stored in the database, a secure encryption mode is used, i.e., one which uses an initialization vector (IV)[1].

(4) Similar to Simple-DASLog, at the end of epoch e, the logging interface uses the proof function in the deployed smart contract to write the following message on the blockchain:

$$M_e^{\text{Proof}} = \Big\{\mathsf{MR}^e, e\Big\} \tag{14}$$

Once MR^e is written on the blockchain, the logging interface extracts its corresponding transaction address. At the end of the writing process in epoch e, the following array has been stored in the logging database for the logging record $L_i \in \mathcal{L}_C^e$:

$$\Big(C_{\text{PID}}, \mathsf{Enc}_{k_1}(L_i), \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^e), \mathsf{IV}\Big) \tag{15}$$

At the end of epoch e, the logging interface updates the hash-chain-proof array for chapter C_{ID} as follows:

$$\Big(C_{\text{PID}}, \mathsf{PATH}_{e,C}^{\mathsf{hash\text{-}chain}}, \mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^e), \mathsf{IV}\Big) \tag{16}$$

(5) Once the flight operation and subsequently the chapter ends, the operator O requests all the data sources (except the UAV) to send the signature $\sigma_{\mathsf{DS}}\big(C_{\text{ID}}||\mathsf{LAST}||\mathsf{H}(L_i^{\mathsf{DS,last}})\big)$, where LAST is a pre-defined message indicating the last logging record, and $L_i^{\mathsf{DS,last}}$ represents the last logging record

[1]Using IV ensures the same plaintexts will not encrypt to the same ciphertext. More details regarding the effect of IV on privacy are mentioned in section IX-B.

generated by DS in chapter C_{ID}. The UAV sends the hash of its last raw logging record, i.e., $\mathsf{H}(RL_i^{\mathsf{UAV,last}})$ to USP, and USP sends the signature $\sigma_{\mathsf{USP}}\big(C_{\text{ID}}||\mathsf{LAST}||\mathsf{H}(RL_i^{\mathsf{UAV,last}})\big)$ to the operator. The operator O generates the last logging record of chapter C_{ID} as follows:

$$L_{\mathsf{last}} = \Big\{\sigma_{\mathsf{USP}}\big(C_{\text{ID}}||\mathsf{LAST}||\mathsf{H}(RL_i^{\mathsf{UAV,last}})\big), \sigma_{\mathsf{DS}}\big(C_{\text{ID}}||\mathsf{LAST}||\mathsf{H}(L_i^{\mathsf{DS,last}})\big)\big|_{\text{for all DS except UAV}}\Big\} \tag{17}$$

L_{last} contains the signatures of all the data sources on the hash of their last logging record. Similar to all the logging records, the logging interface encrypts L_{last} and adds it to the logging database. Let e_{end} represent an epoch in which L_{last} is generated. Similar to the previous epochs, the logging interface performs the task of proof generation for epoch e_{end} and writes $M^{\text{Proof}}_{e_{\text{end}}}$ on the blockchain. The last hash-chain proof for chapter C is the one generated in epoch e_{end}.

(6) Finally, the operator O uses the finalizing function in the deployed smart contract to write the following message on the blockchain:

$$M^{\text{Final}}_{C_{\text{ID}}} = \Big\{\mathsf{H}_1(\mathsf{CNT}^1_{\mathsf{A,B}}||N^2_{\mathsf{A,B}}), e_{\text{end}}, \mathsf{H}(L_{\mathsf{last}})\Big\} \tag{18}$$

$M^{\text{Final}}_{C_{\text{ID}}}$ declares the end of the chapter C_{ID}. Customers A and B can listen to the events emitted from the smart contract to find whether the final message including $\mathsf{H}_1(\mathsf{CNT}^1_{\mathsf{A,B}}||N^2_{\mathsf{A,B}})$, i.e., $M^{\text{Final}}_{C_{\text{ID}}}$, is added to blockchain or not. $M^{\text{Final}}_{C_{\text{ID}}}$ includes the hash of the last logging record, which can be served as the integrity proof of signatures included in L_{last}.

D. Read operation in DASLog

The read operation in DASLog is similar to the read operation in Simple-DASLog. Upon receiving a read request from an eligible data consumer to fetch the logging records of chapter C_{ID}, the operator sends the following message to the data consumer:

$$\Big\{\big(L_i, e, \mathsf{Addr}^e_{\mathsf{MR}}\big) \mid L_i \in C\Big\} \parallel \big(\mathsf{PATH}^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}, e_{\text{end}}, \mathsf{Addr}^{e_{\text{end}}}_{\mathsf{MR}}\big) \tag{19}$$

E. Verification phase in DASLog

Upon receiving the response from the operator, the data consumer needs to verify the correctness and completeness of the received logging records. The verification steps are as follows.

(1) Recall that the logging record set generated by the same data source in each chapter creates a hash chain. As the first verification step, the data consumer verifies the hash chain generated by each of the data sources. Then, the data consumer finds the last logging record of each data source in the set of received logging records and verifies the correctness of the signatures included in L_{last}, and checks whether or not $\mathsf{H}(L_{\mathsf{last}})$ is included in $M^{\text{Final}}_{C_{\text{ID}}}$. By performing the first verification step, the data consumer can make sure that not only it has received the complete logging record set generated by each data source in chapter C_{ID}, but the order of the logging records generated by the same data source has been preserved. Note that this step is necessary to prevent a malicious operator from performing both tail and non-tail truncation attacks. See Section IX-B for more details regarding the truncation attack.

(2) As the second verification step, the data consumer needs to verify the main hash-chain proof of the chapter generated by the operator, i.e., $P^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$. The data consumer uses $\mathsf{Addr}^{e_{\text{end}}}_{\mathsf{MR}}$ to fetch the Merkle root proof of an epoch in which the proof $P^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$ has been generated from the blockchain. Next, it uses Algorithm 1 and the set of logging records (L_i) received from the operator to calculate $P^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$. Then, it uses $P^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$ and the received path $\mathsf{PATH}^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$ to reconstruct the Merkle root. If the computed Merkle root corresponds to the one retrieved from the blockchain, the verification holds. By performing the second verification step, the data consumer can make sure that the sequence of received logging records has been preserved.

(3) The data consumer verifies the correctness of the digital signatures embedded in each of the logging records to assess the integrity of individual logging records.

(4) For the logging records that are generated by UAVs, the data consumer can compare logging data created by the UAV, i.e., $\mathsf{data}^{\mathsf{UAV}}$, with the data measured by the surveillance sensors, i.e., $\mathsf{data}^{\mathsf{sensor}}$, to verify whether or not $\mathsf{data}^{\mathsf{UAV}}$ is correct. Note that $\mathsf{data}^{\mathsf{UAV}}$ and $\mathsf{data}^{\mathsf{sensor}}$ should not be exactly the same due to the measurement error.

(5) The data consumer can compare the generation time of the logging records, i.e., ts, stored inside the logging record, with the generation time of the block in which the corresponding Merkle root proof is stored in the blockchain. This step can help the data consumer make sure that the operator has processed the logging records immediately after receiving them. A malicious operator may keep a specific logging record for a long time to perform the re-ordering attack.

When these verification steps pass successfully, the data consumer knows the set of logging records it received is correct and complete, i.e., no logging records have been omitted or altered, and the sequence of logging records within the chapter has been preserved.

VIII. Proof-of-concept implementation

A. Private blockchain implementation

There exist multiple platforms that can provide private blockchains. In our proof-of-concept, we have used Hyperledger Besu [36] to implement the 3rd party register. Hyperledger Besu is an open-source project developed under the Apache 2.0 license and written in Java. Hyperledger Besu is built on top of the Hyperledger framework [37] and is compatible with the Ethereum network. Hyperledger Foundation is an organization that provides resources and infrastructure to develop software blockchain projects. In fact, Hyperledger is a well-known framework on top of which a considerable number of private blockchains have been designed. As well as compatibility with the Hyperledger framework, Hyperledger Besu is also compatible with the Ethereum network, which is a well-known and widely-used blockchain technology that

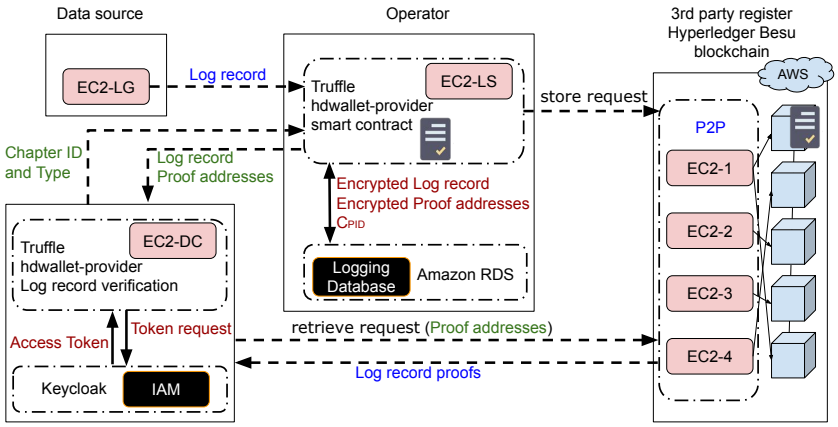


Fig. 6. Proof-of-concept prototype, high-level overview

can provide decentralized data storage with smart contract functionality.

One of the main reasons that make us choose Hyperledger Besu to implement the 3rd party register is the fact that Hyperledger Besu can provide us with BFT-based consensus mechanisms. In our secure logging system, by design, we do not want the operator to be able to modify or remove the proofs stored on the private blockchain. As a result, we should use a BFT-based private blockchain that can withstand malicious behavior. By using a BFT-based private blockchain, the operator cannot behave in a malicious manner to modify or remove the proofs already written on the blockchain. The consensus mechanism that has been used in our proof-of-concept is called IBFT 2.0, which is a BFT-based consensus protocol that ensures immediate finality of the information blocks and is robust in an eventually synchronous network [38].

In our proof-of-concept, we have used smart contracts to realize a storage program. This program takes a value as input, which in our case are the Merkle root proofs, and stores these values on the blockchain. In this project, the storage smart contract is written in the Solidity language. Another tool that is used in our implementation is a development tool named Truffle [39]. With the help of Truffle, nodes can interact with the implemented private blockchain to deploy smart contracts on the blockchain or read information from the blockchain. Besides, we have used HDWalletProvider [40] to sign the transactions prior to sending them to the blockchain.

B. Demonstration of proposed secure logging system

Our demonstration of the DASLog secure logging system is composed of four entities: (1) Operator, (2) Data Source, (3) Data Consumer, and (4) 3rd Party Register. To ensure that our solution can be easily (re)deployed in practice, we have opted for a cloud-based implementation. We used seven Amazon Elastic Compute Cloud (Amazon EC2 [41]) instances, as illustrated in Figure 6, to implement the four entities mentioned above. More specifically:

- EC2-LS is an EC2 instance realizing the operator component. This instance stores encrypted log records and addresses of the proofs in a MySQL database. We used Amazon RDS [42] to implement the MySQL database.
- EC2-LG is an EC2 instance realizing the data source component. This instance forwards log records to the operator component via a REST (Representational State Transfer) interface.
- EC2-DC is an EC2 instance realizing the data consumer component. This instance requests log records via a REST interface. It interacts with the 3rd party register to obtain the necessary proofs to audit the log records.
- EC2-1, .., EC2-4 are EC2 instances realizing the 3rd party register, based on a Hyperledger Besu private blockchain network. These four EC2 instances are the blockchain validator nodes in our demonstration, where EC2-1 is the bootnode.

All EC2 instances are "Amazon Linux, t2.micro" located in different virtual private clouds (VPCs). To deploy the necessary cryptographic functions, we used the Python cryptography library[2]. We also used Flask-Python micro framework [43] built on top of Python for implementing all the REST interfaces [44][3]. We also used the OpenSSL library to generate public and private key pairs [45]. More details on configuring the Amazon RDS and EC2 instances can be found in Appendix A.

In our demonstration, the data source component (EC2-LG) first generates a sample log record, containing C_{ID} factor, and sends it to the operator component (EC2-LS) via a REST interface. Afterward, EC2-LS calls a mapping function to compute the pseudo identifier using the relation $C_{\text{PID}} = \mathsf{HMAC}_{k_2}(C_{\text{ID}})$, where HMAC is a hash-based message authentication code using SHA-256 and a secret key k_2.

The EC2-LS instance then encrypts the log record and stores the result in the RDS database. The encryption algorithm used in EC2-LS is the Advanced Encryption Standard (AES) in CBC mode with a 128-bit key k_1 for encryption, using PKCS7 padding. The SHA-256 hash function is also used in our demonstration to compute the hash-chain and single-hash proofs. In parallel, the EC2-LS reads the latest logging records that no proof has been generated for them. Then, it generates proofs for these logs (see Section VI-A for more

[2]https://cryptography.io/en/latest/

[3]The source code of our demonstrator can be found at https://github.com/logging-system/DASLog.

details) and forwards the Merkle root proof to the blockchain as the message $M_e^{\text{Proof}} = \left\{\mathsf{MR}^e, e\right\}$. The other program in our demonstration regularly reads the transaction hash (addresses) where the Merkle root proofs are stored ($\mathsf{Addr}^e_{\mathsf{MR}}$). The EC2-LS instance finally writes the encrypted address of the Merkle root proofs as an array into an Amazon RDS database.

When the data consumer (EC2-DC) wants to access the log records, it first sends an access token request to the IAM server. In our demonstration, we used the Keycloak IAM, which is an open-source framework [46]. If Keycloak successfully authenticates the data consumer, it generates a fresh access token and sends this back to the EC2-DC instance. Finally, EC2-DC sends this access token along with the chapter type and identification of the chapter it wants to retrieve to EC2-LS via a REST interface. Note that for simplicity, we have integrated Keycloak in the EC2-DC. More details about the Keycloak IAM demonstration can be found here[4].

After checking the access token and the access control policies, the EC2-LS instance uses the HMAC function and the key k_2 to generate the $C^{\mathsf{read}}_{\text{PID}}$. Using $C^{\mathsf{read}}_{\text{PID}}$, EC2-LS fetches the encrypted logging records of the requested chapter(s) along with the encrypted proof addresses from the Amazon RDS database, decrypts them using k_1 and forwards the result to EC2-DC as a REST response. See Section VI-B for more details regarding the read operation steps.

Upon receiving the REST response from EC2-LS, the EC2-DC instance uses the proof addresses to fetch the related proofs from the blockchain. Finally, the EC2-DC instance uses these proofs to verify the correctness and completeness of the received logging records, following the verification process presented in Section VI-C.

IX. Security and performance evaluation

In this section, we evaluate the security and performance of our proposed DASLog logging system.

A. Security analysis of Simple-DASLog

In this section, we assess how Simple-DASLog satisfies the security goals mentioned in Section III-C.

Confidentiality: In our scheme, the encrypted version of the logging records, i.e., $\mathsf{Enc}_{k_1}(L_i)$, are stored in the logging database; thus, if an attacker manages to get physical access to the logging database, he/she cannot find access to the contents of logging records. Besides, in the reading phase, a data consumer first needs to authenticate itself to an IAM server to get a fresh access token. Using the access token, the control interface component in the operator applies the control access policies and prevents an unauthorized data consumer from accessing the logging records. Moreover, as mentioned earlier, a secure TLS connection is available between all the entities; hence, an external attacker who is eavesdropping on the communication channels cannot find access to the transferred data.

Integrity: Prior to start of a chapter, an initial message $M^{\text{Init}}_{C_{\text{ID}}}$ is stored on the private blockchain. Therefore, the operator cannot deny the existence of such a chapter. In Simple-DASLog, we assume that the operator is honest during the write operation. Thus, a malicious operator may try to tamper with the integrity of logging records at the reading time. Since a Merkle root proof is stored on the blockchain as the integrity proof for the hash-chain proof $P^{\mathsf{hash\text{-}chain}}_{e_{\text{end}}, C}$, any modification or deletion in/of non-tail logs at the reading time can be detected. However, the hash-chain proof is not enough to stop the tail-truncation attack, in which the operator can delete a set of consecutive logging records at the end of the chapter. To prevent the tail-truncation attack, once a chapter ends, a final message $M^{\text{Final}}_{C_{\text{ID}}}$ is stored on the blockchain, which includes e_{end}. e_{end} is the epoch in which the last logging record of the chapter has been processed. If logging records in the received chapter end before reaching the epoch e_{end}, the tail-truncation attack can be detected. Note that, since we are using a BFT-based blockchain in our scheme, a malicious operator cannot modify a piece of data that is already stored on the ledger.

Availability: The logging records are stored in a central database controlled by the operator. Therefore, the availability of the logging records in our scheme depends on the operator's availability. However, the proofs are stored in the private blockchain, and thus, enjoy a high level of availability. Once a data consumer receives a set of logging records, there is a guarantee that the data consumer can find access to the proofs to validate the logging records.

Immutability: The immutability of the private blockchain on which the proofs are written guarantees the immutability of our logging system. To make logging records immutable, we should make sure that their proofs are immutable, i.e., the operator cannot alter the proofs. Since we have used a BFT-based private blockchain to store the log proofs, we can make sure the blockchain history cannot be deleted or altered as long as the operator has not colluded with a considerable number of blockchain validators.

Public auditability: Once a data consumer receives a set of logging records, it can use the proofs stored on the private blockchain to verify the correctness and completeness of the logging records. The proofs stored on the blockchain can be accessed by all the data consumers and be verified without the need for a trusted third party.

B. Security analysis of DASLog

In this section, we assess how DASLog satisfies the security goals mentioned in Section III-C. The analyses for properties confidentiality, availability, immutability, and public auditability are the same as the analyses for Simple-DASLog in Section IX-A.

Integrity of an individual logging record: The integrity of each individual logging record is protected by the digital signature(s) included in the logging record. Note that it is computationally infeasible for the operator to forge a valid signature of an external data source that is not under its control. However, since UAVs are equipped and controlled by the operator, a malicious operator may try one of the following scenarios to tamper with the UAVs' logging records. In the first scenario, the operator may try to forge the received UAV

[4]https://github.com/logging-system/DASLog#:~:text=Keycloak

logging records since the operator is able to find access to UAVs' private keys and forge their signatures. To prevent this scenario, in our scheme, the UAVs' logging records are signed by USP prior to being sent to the operator. Therefore, to forge the UAVs' logging records, the operator should forge USP's signature too, which is impossible. Although the operator cannot forge the logging record received from USP, it can still send a fake raw logging record to UAV and ask UAV to forward it to USP. To prevent this scenario, the UAVs' logging record contains two pieces of data, namely $\mathsf{data}^{\mathsf{UAV}}$ and $\mathsf{data}^{\mathsf{sensor}}$. The former is generated by UAVs and can contain false information. However, the latter is measured by the surveillance sensors which are not controlled by the operator. Since $\mathsf{data}^{\mathsf{UAV}}$ is a piece of internal verifiable data, the data consumer needs to compare $\mathsf{data}^{\mathsf{UAV}}$ with $\mathsf{data}^{\mathsf{sensor}}$ in the verification process. If these two pieces of data do not match together, the verification does not hold.

Integrity of a whole chapter: Prior to the start of a chapter, an initial message $M_{C_{\text{ID}}}^{\text{Init}}$ is stored on the private blockchain, which includes the operator's signature on the contract created by the customers. Therefore, the operator cannot deny the existence of such a chapter. In addition, each data source uses a hash chain to create its logging records; hence, the operator can neither perform a reordering attack on a series of logging records sent from the same data source nor delete one of the non-tail logging records. However, these hash chains are not enough to stop the tail-truncation attack, in which the operator can delete a set of consecutive logging records at the end of the hash chain. To prevent the tail-truncation attack, once a chapter ends, a final message $M_{C_{\text{ID}}}^{\text{Final}}$ is stored on the blockchain, which includes $\mathsf{H}(L_{\mathsf{last}})$. L_{last} is the last logging record of the chapter and contains the signature of each data source on the hash of its last logging record, i.e., $\sigma_{\mathsf{DS}}\big(C_{\text{ID}}||\mathsf{LAST}||\mathsf{H}(L_i^{\mathsf{DS,last}})\big)$. To perform the tail-truncation attack at the reading time, the operator needs to modify or remove the data sources' signatures in/form L_{last}. The operator cannot forge the signatures since it is computationally infeasible. However, a malicious operator may try to remove one of the signatures from L_{last} and claim the corresponding data source has not sent the signature. To prevent this scenario, the hash of the last logging record $\mathsf{H}(L_{\mathsf{last}})$ is included in $M_{C_{\text{ID}}}^{\text{Final}}$ and stored on the blockchain. Removing any signature from L_{last} leads to a change in $\mathsf{H}(L_{\mathsf{last}})$, which can be detected by calculating the hash of the last logging record and comparing it with the one stored on the blockchain. Note that, since we are using a BFT-based blockchain in our scheme, a malicious operator cannot modify a piece of data that is already stored on the ledger.

Non-repudiation: Since a data source needs to sign all of its logging records using its private key, the data source cannot repudiate creating and signing those logging records in the future.

Privacy: In DASLog, the logging records are stored in the operator database with their pseudo identifier $C_{\text{PID}} = \mathsf{HMAC}_{k_2}(C_{\text{ID}})$ instead of their public factor C_{ID}. This is because if an attacker finds physical access to the logging database, he/she cannot identify which rows belong to which chapters. Note that, we have assumed keys k_1 and k_2 which are respectively used to encrypt the logging records and generate C_{PID} are stored in secure registers. Besides, to improve the privacy level in DASLog, an initialization vector is used to encrypt the arrays. Let's assume a scenario in which IV is not used. In this case, the tuple $\big(\mathsf{Enc}_{k_1}(e), \mathsf{Enc}_{k_1}(\mathsf{Addr}_{\mathsf{MR}}^e)\big)$ would encrypt into the same ciphertext for all the logging records that have been processed in the same epoch. Therefore, an attacker who has found access to the logging database can find some privacy-sensitive information such as the number of logging records processed in each epoch and the duration of the flight. However, by using IV, we can prevent the mentioned information leakage. In DASLog, for each chapter, one initial message $M_{C_{\text{ID}}}^{\text{Init}}$ and one final message $M_{C_{\text{ID}}}^{\text{Final}}$ are stored on the blockchain. Besides, for each epoch, a proof message M_e^{Proof} is stored on the blockchain. Since all the entities in the IoD network can have access to the private blockchain, it is of huge importance that the malicious entities cannot extract any privacy-sensitive information from data stored on the blockchain. In our scheme, chapter C is represented by a random identification number C_{ID}. The initial message of chapter C includes the random identification C_{ID}, which reveals nothing about the content of the chapter. Besides C_{ID}, the initial message includes $\mathsf{H}(\mathsf{CNT}_{\mathsf{A,B}})$ and $\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}}^1)$. Since contracts have a predefined format, an attacker may try the brute force attack to find $\mathsf{CNT}_{\mathsf{A,B}}$ in a way that satisfies the mentioned hash results. However, due to using a random nonce $N_{\mathsf{A,B}}^1$ in each contract, this attack would be impossible. Similarly, the final message includes $\mathsf{H}_1(\mathsf{CNT}_{\mathsf{A,B}}^1||N_{\mathsf{A,B}}^2)$, which reveals no information regarding the contract. Besides, since two different nonces $N_{\mathsf{A,B}}^1$ and $N_{\mathsf{A,B}}^2$ are used in the initial and final messages, an attacker cannot link the initial and final messages of the same chapter. Note that linking the initial and final messages of the same chapter can reveal some privacy-sensitive information such as the duration of the flight.

C. Performance evaluation

In this section, we evaluate the performance of our proposed logging system considering the number of log records on both the operator and data consumer sides. Without loss of generality, we consider that all the logging records belong to the same chapter. We obtained the benchmarks on a Linux 20.04.03 laptop with AMD Ryzen 5 PRO 3500U 2.10 GHz CPU and 8GB RAM.

Log processing time (LPT) versus the total number of log records (Fig 7(a)): This graph displays the time required to encrypt logging records and generate their corresponding proofs. In Fig 7(a), we have two graphs: one that is for the DASLog scheme and considers the time consumption of the encryption of the logging record and the hash-chain (SHA-256) proof generation; the second graph is related to the Simple-DASLog scheme and shows the time consumption of the logging record encryption, the hash-chain proof generation, as well as the single proof generation. In our implementation, the hash algorithm SHA-256 is used for implementing the Merkle tree. If we assume $n+1$ is the number of leaves in the

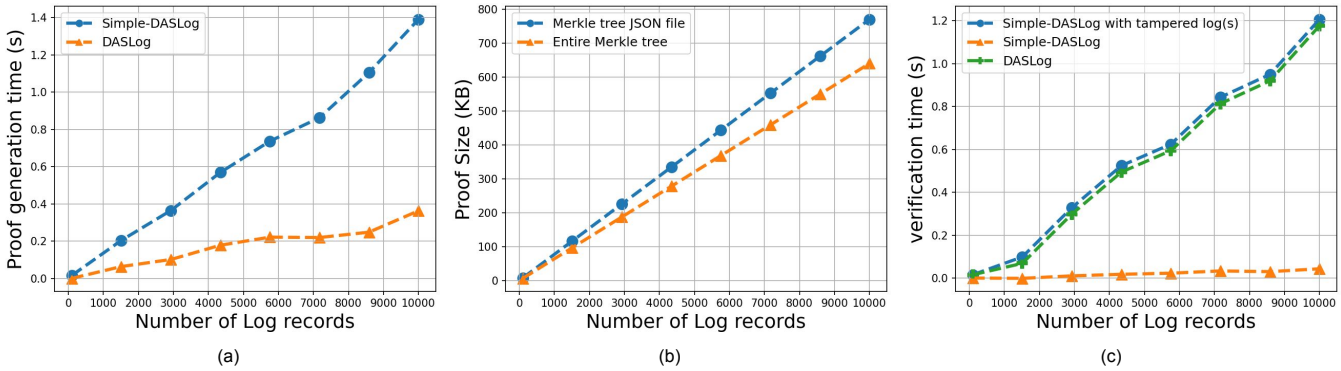


Fig. 7. a) Log processing time, b) Merkle tree size, and c) Verification time

Merkle tree, the first n leaves are single proofs and the last leaf is the hash-chain proof. As can be seen, the second graph has an overhead which is due to the single proof generation and Merkle tree process. Based on the results, the proposed system can handle about 8000 log records in LPT = 1 second and LPT = 220 milliseconds using Simple-DASLog and DASLog, respectively.

Merkle tree size (MTS) versus the total number of log records (Fig 7(b)): In this graph, if we look at the size of the tree over the total number of the logging records, we can see that this relationship is linear. In our proof-of-concept, we store the Merkle tree (JavaScript Object Notation) JSON file with the minimum required information for calculating the root value and verifying the membership. Looking at the graph, one can see that in our system, MTS grows linearly from ≈ 0.2 (KB) to ≈ 770 (KB) for 1 to 10000 log records, respectively. In Table II, we list the size of all data that needs to be stored in the operator database. The result shows that our proposed scheme consumes less than 611 KB to store information needed for the log verification.

Verification time (VT) versus the total number of log records (Fig 7(c)): This graph presents the time the data consumer must spend verifying logging records. For the Simple-DASLog scheme, there are two possible scenarios in the verification phase: (i) no log records have been tampered with, and the data consumer only needs to verify the hash-chain, (ii) log records have been tampered with, and the data consumer needs to verify all the received log records. As shown in the graph, if the hash-chain proof is verified successfully, the verification time is negligible. In the case that one or more logs are tampered with (worst case), the time to verify 10000 log records is almost 1.2 sec for Simple-DASLog. However, the verification time when using DASLog scheme is slightly less than the worst case of Simple-DASLog. As shown in the graph, the verification time of DASLog consists of the time consumption of the signature verification (RSA signature is used) and the hash-chain proof.

TABLE II
THE MEMORY USAGE OF AN RDS DATABASE FOR PROOFS OF 8000 LOG RECORDS

Tree JSON file	hash-chain	$\mathsf{Enc}_{k_1}(e)$	$\mathsf{Enc}_{k_1}(\mathsf{Addr}^e_{\mathsf{MR}})$
610 KB	256 bit	0.184 KB	0.184 KB

Discussion: As mentioned above, our proposed logging system is capable of processing up to 8000 log records in $LPT = 1\ sec$. Log processing consists of encrypting and storing the logs, generating the corresponding proofs, and writing the Merkle root proof on the blockchain. We configured the block generation time (BGT) as one second in our Hyperledger Besu network consisting of 4 validators and accordingly set the epoch length to one second. Note that for a blockchain network with a higher number of validators, one should increase the block generation time. Thus, in the proposed logging system, the operator can handle up to 8000 logging records in each epoch. As already mentioned, it is possible to fetch the proof addresses (i.e., $\mathsf{Addr}^e_{\mathsf{MR}}$) from the blockchain and store it in the database after a while, which is less than 3 seconds in our configured blockchain network. Thus, in our demonstration, the operator fetches each Merkle root proof address from the blockchain 3 seconds after writing it. Since in our scheme fetching the proof addresses and writing the proofs to the blockchain are independent operations, the operator could process up to 8000 log records per epoch without delay. Therefore, one can conclude that our proof-of-concept meets the performance requirements imposed by the UAV ecosystem for which we have designed the logging scheme.

X. Conclusion and Future Work

Security logging is well understood by ICT professionals and widely deployed in commercial systems. Most solutions available merely focus on achieving the conventional requirements considered in the security triad (confidentiality, integrity, availability) and on the reliable collection of relevant logging data. However, some systems require additional security guarantees that are typically not offered by state-of-the-art logging systems. Our work aimed to develop a secure logging scheme for UAV-integrated IoT systems where drones transport (medical) goods from one location to another. One of the aspects which makes this UAV system unique from a security point of view is that the internal data sources (for example, the UAVs) and the logging system itself are all managed and controlled by a single entity – the operator. Yet, external parties requesting read access to logging records for

auditing purposes, need the necessary security guarantees that the set of log records they retrieved from the logging system is complete and not altered by the operator during reading time.

In this paper, we addressed this need and proposed DASLog, a novel secure logging scheme that provides public auditability in the setting mentioned above. The design relies on the generation of Merkle tree security proofs stored in a distributed private blockchain. To demonstrate the feasibility of our approach, we implemented a proof-of-concept of our logging scheme on multiple Amazon EC2 instances and used Hyperledger Besu to realize the private blockchain. Performance evaluations of our demonstrator showed that up to 8000 logging records could be processed per second. These performance metrics vastly exceed what is required by the UAV system for which the secure logging scheme was designed.

In our scheme, the logging records are stored in a central database monitored by the operator, and only the proofs get stored on the blockchain. As future work, we can investigate how to store the logging records on the blockchain too without compromising the efficiency and throughput. By doing so, we can reduce the threat of a malicious operator, and thus, further improve security.

REFERENCES

[1] NIST, "NIST Special Publication 800-53: Security and Privacy Controls for Information Systems and Organizations," https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf.

[2] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, A. D. Rubin, Ed. USENIX Association, 1998.

[3] J. E. Holt, "Logcrypt: forward security and public verification for secure audit logs," in *The proceedings of the Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006) and the Fourth Australasian Information Security Workshop (Network Security) (AISW 2006), Hobart, Tasmania, Australia, January 2006*, ser. CRPIT, R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo, Eds., vol. 54. Australian Computer Society, 2006, pp. 203–211.

[4] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, F. Monrose, Ed. USENIX Association, 2009, pp. 317–334.

[5] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Trans. Storage*, vol. 5, no. 1, pp. 2:1–2:21, 2009.

[6] S. Rane and A. M. Dixit, "Blockslaas: Blockchain assisted secure logging-as-a-service for cloud forensics," in *Security and Privacy - Second ISEA International Conference, ISEA-ISAP 2018, Jaipur, India, January, 9-11, 2019, Revised Selected Papers*, ser. Communications in Computer and Information Science, S. Nandi, D. Jinwala, V. Singh, V. Laxmi, M. S. Gaur, and P. Faruki, Eds., vol. 939. Springer, 2018, pp. 77–88.

[7] B. Putz, F. Menges, and G. Pernul, "A secure and auditable logging infrastructure based on a permissioned blockchain," *Computers & Security*, vol. 87, p. 101602, 2019.

[8] R. Paccagnella, K. Liao, D. Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1551–1574.

[9] A. Ali, A. Khan, M. Ahmed, and G. Jeon, "Bcals: Blockchain-based secure log management system for cloud computing," *Transactions on Emerging Telecommunications Technologies*, vol. 33, no. 4, p. e4272, 2022.

[10] H. Han, S. Fei, Z. Yan, and X. Zhou, "A survey on blockchain-based integrity auditing for cloud data," *Digital Communications and Networks*, 2022.

[11] T. Pulls, R. Peeters, and K. Wouters, "Distributed privacy-preserving transparency logging," in *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*, A. Sadeghi and S. Foresti, Eds. ACM, 2013, pp. 83–94.

[12] S. Sackmann, J. Strüker, and R. Accorsi, "Personalization in privacy-aware highly dynamic systems," *Commun. ACM*, vol. 49, no. 9, pp. 32–38, 2006.

[13] R. Peeters and T. Pulls, "Insynd: Improved privacy-preserving transparency logging," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, Eds., vol. 9879. Springer, 2016, pp. 121–139.

[14] B. Bera, D. Chattaraj, and A. K. Das, "Designing secure blockchain-based access control scheme in iot-enabled internet of drones deployment," *Computer Communications*, vol. 153, pp. 229–249, 2020.

[15] C. Feng, B. Liu, Z. Guo, K. Yu, Z. Qin, and K.-K. R. Choo, "Blockchain-based cross-domain authentication for intelligent 5g-enabled internet of drones," *IEEE Internet of Things Journal*, vol. 9, no. 8, pp. 6224–6238, 2021.

[16] X. Xu, H. Zhao, H. Yao, and S. Wang, "A blockchain-enabled energy-efficient data collection system for uav-assisted iot," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp. 2431–2443, 2020.

[17] B. Bera, S. Saha, A. K. Das, N. Kumar, P. Lorenz, and M. Alazab, "Blockchain-envisioned secure data delivery and collection scheme for 5g-based iot-enabled internet of drones environment," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 8, pp. 9097–9111, 2020.

[18] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, H. Karimipour, G. Srivastava, and M. Aledhari, "Enabling drones in the internet of things with decentralized blockchain-based security," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6406–6415, 2020.

[19] Y. Tan, J. Wang, J. Liu, and N. Kato, "Blockchain-assisted distributed and lightweight authentication service for industrial unmanned aerial vehicles," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16 928–16 940, 2022.

[20] P. Kumar, R. Kumar, A. Kumar, A. A. Franklin, and A. Jolfaei, "Blockchain and deep learning empowered secure data sharing framework for softwarized uavs," in *2022 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2022, pp. 770–775.

[21] P. P. Ray and K. Nguyen, "A review on blockchain for medical delivery drones in 5g-iot era: Progress and challenges," in *2020 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*. IEEE, 2020, pp. 29–34.

[22] Helicus, https://helicus.com/.

[23] N. Samir Labib, G. Danoy, J. Musial, M. R. Brust, and P. Bouvry, "Internet of unmanned aerial vehicles—a multilayer low-altitude airspace model for distributed uav traffic management," *Sensors*, vol. 19, no. 21, p. 4779, 2019.

[24] D. Carramiñana, I. Campaña, L. Bergesio, A. M. Bernardos, and J. A. Besada, "Sensors and communication simulation for unmanned traffic management," *Sensors*, vol. 21, no. 3, p. 927, 2021.

[25] Internet Engineering Task Force, "RFC8446 – The Transport Layer Security (TLS) Protocol Version 1.3," https://datatracker.ietf.org/doc/html/rfc8446.

[26] A. A. Yavuz, P. Ning, and M. K. Reiter, "Baf and fi-baf: Efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 2, pp. 1–28, 2012.

[27] A. Collomb and K. Sok, "Blockchain/distributed ledger technology (dlt): What impact on the financial sector?" *Digiworld Economic Journal*, no. 103, 2016.

[28] R. Yang, R. Wakefield, S. Lyu, S. Jayasuriya, F. Han, X. Yi, X. Yang, G. Amarasinghe, and S. Chen, "Public and private blockchain in construction business process and information integration," *Automation in construction*, vol. 118, p. 103276, 2020.

[29] C. Mohan, "State of public and private blockchains: Myths and reality," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 404–411.

[30] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[31] C.-W. Chen, J.-W. Su, T.-W. Kuo, and K. Chen, "Msig-bft: A witness-based consensus algorithm for private blockchains," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 992–997.

[32] G.-T. Nguyen and K. Kim, "A survey about consensus algorithms used in blockchain," *Journal of Information processing systems*, vol. 14, no. 1, pp. 101–128, 2018.
[33] X. Wang, S. Duan, J. Clavin, and H. Zhang, "Bft in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, 2021.
[34] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert, "The immutability concept of blockchains and benefits of early standardization," in *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*. IEEE, 2017, pp. 1–8.
[35] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, "Solutions to scalability of blockchain: A survey," *Ieee Access*, vol. 8, pp. 16 440–16 455, 2020.
[36] Hyperledger Besu, https://www.hyperledger.org/use/besu.
[37] Hyperledger, https://www.hyperledger.org/.
[38] R. Saltini and D. Hyland-Wood, "Ibft 2.0: A safe and live variation of the ibft blockchain consensus protocol for eventually synchronous networks," *arXiv preprint arXiv:1909.10194*, 2019.
[39] Truffle, "trufflesuite," https://trufflesuite.com/.
[40] HDWallet, "Truffle HDWalletProvider," https://github.com/trufflesuite/truffle/tree/develop/packages/hdwallet-provider.
[41] A. W. Services, "AWS EC2," https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc.
[42] ——, "AWS RDS," https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.DBInstance.html.
[43] Flask, "Flask-Python micro framework," https://flask.palletsprojects.com/en/2.0.x/.
[44] Crypto, "Python crypto library," https://cryptography.io/.
[45] OpenSSL, "Cryptography and ssl/tls toolkit," https://www.openssl.org/.
[46] Keycloak, "Keycloak Identity and Access Management," https://www.keycloak.org/.

APPENDIX
EC2 INSTANCE AND AMAZON RDS CONFIGURATIONS

A. EC2 instance configurations

In our demonstration, we have used four AWS EC2 instances as blockchain nodes and three AWS EC2 instances as Operator (EC2-LS), Data Consumer (EC2-DC), and Data Source (EC2-LG) respectively.

The general process of setting up an AWS EC2 instance was done as follows.

- The instance type must be selected as "Amazon Linux, t2.micro".
- Create a new VPC. Note that, in our configuration, each node is in a different VPC. Using different VPCs proves that, when the system would be deployed in a real-life setting, we can have each instance in a different account.
- Configure the security group and add inbound configuration.

Blockchain node inbound configurations: In our demonstration, the bootnode should have the minimum inbound configurations as listed in Table III. The UDP/TCP port 30303 is used to enable peer discovery and UDP/TCP port 8545 is used to accept inbound traffic sent from the client nodes (i.e., EC2-LS and EC2-DC instances in our proof-of-concept).

TABLE III
THE INBOUND CONFIGURATIONS FOR THE BOOTNODE

Type	Protocol	Port range	Source
Custom TCP	TCP	30303	0.0.0.0/0
Custom UDP	UDP	30303	0.0.0.0/0
Custom TCP	TCP	8545	0.0.0.0/0
Custom UDP	UDP	8545	0.0.0.0/0
SSH	TCP	22	My IP

In our demonstration, all other nodes except the bootnode should have the minimum inbound configurations as listed in Table IV. The UDP/TCP port 30303 is also used to enable peer discovery. Note that, in all the inbound configurations above, Secure Shell (SSH) is used to communicate with an AWS EC2 instance on port 22.

TABLE IV
THE INBOUND CONFIGURATIONS FOR THE OTHER NODES OF THE BLOCKCHAIN

Type	Protocol	Port range	Source
Custom TCP	TCP	30303	0.0.0.0/0
Custom UDP	UDP	30303	0.0.0.0/0
SSH	TCP	22	My IP

Inbound configurations of the other AWS EC2 instances: The EC2-LS settings are given in Table V. As is shown, this instance accepts MYSQL database traffic on port 3306. EC2-LS also accepts traffic from port 5000, which is for the Flask REST API. The EC2-DC settings are given in Table VI. As mentioned, this instance also implements the Keycloak IAM, which listens on port 8080. Note that the EC2-LG instance only needs an SSH inbound on port 22.

Installing Hyperledger Besu on EC2 instances: Using the steps below, one can install Hyperledger Besu on EC2 instances as nodes of the blockchain.

- To upload the Besu version besu-21.7.2m, run wget https://hyperledger.jfrog.io/artifactory/besu-binaries/besu/21.7.2/besu-21.7.2.zip.
- unzip besu-21zip
- sudo amazon-linux-extras install java-openjdk11 -y
- sudo ln -s /home/ec2-user/besu-21.7.2/bin/besu /usr/local/bin

Installing Truffle on the operator and data consumer EC2 instances: Using the steps below, one can install Truffle on the operator and data consumer EC2 instances.

- curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash
- . ∼/.nvm/nvm.sh
- nvm install node
- npm install -g truffle
- npm install –save @truffle/hdwallet-provider

The readers are referred to the readme file in https://github.com/logging-system/DASLog#readme for the rest of the configurations and steps that must be done in the operator, data consumer and data source components.

B. Amazon RDS configurations

When setting up the MySQL database on AWS RDS, one should consider the following:

- After creating and running the Mysql database, be sure that the database is located in a VPC with a security group that accepts traffic to TCP port 3306.
- Create an empty database and table.

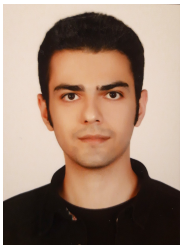
The readers are referred to the readme file in https://github.com/logging-system/DASLog#readme for more information.

TABLE V
THE INBOUND CONFIGURATIONS FOR THE EC2-LS

Type	Protocol	Port range	Source
MYSQL/Aurora	TCP	3306	0.0.0.0/0
Custom TCP	TCP	5000	0.0.0.0/0
SSH	TCP	22	My IP

TABLE VI
THE INBOUND CONFIGURATIONS FOR THE EC2-DC

Type	Protocol	Port range	Source
Custom TCP	TCP	30310	0.0.0.0/0
Custom TCP	TCP	8080	0.0.0.0/0
SSH	TCP	22	My IP



Roozbeh Sarenche received his BS degree in electrical engineering from University of Tehran (UT) in 2016 and his Master's degree in secure communications from Sharif University of Technology (SUT) in 2019. He is currently pursuing his Ph.D. degree at COSIC, KU Leuven, Belgium. His research interests are focused on studying blockchain applications, conducting security analysis of the consensus layer in blockchains, and designing secure protocols.

Farhad Aghili received his Master's degree in Electrical Engineering from Shahid Rajaee Teacher Training University (SRTTU) in 2013. He received his PhD degree in Information Technology Engineering from the Faculty of Computer Engineering, University of Isfahan, in 2019. He did a Post Doctoral Research Fellow, COSIC, KU Leuven, Belgium for three years. He is currently a research expert in SIRRIS https://www.sirris.be/en/expert/farhad-aghili. His current research interests include access control models, authentication protocols, applied cryptography, blockchain and IoT systems security.

Takahito Yoshizawa received BS degree in information and computer science from Georgia Institute of Technology in 1992 and MS degree in Telecommunication from Southern Methodist University in 2002. He has over 30 years of industry experience in mobile communication systems. His experience includes engineering work in the entire product development lifecycle of mobile communication systems, standardization in 3GPP and holds over 10 granted patents. He is currently pursuing his PhD degree at COSIC group in KU Leuven in Belgium.

Dave Singelée received the Master's degree of Electrical Engineering and a PhD in Applied Sciences in 2002 and 2008 respectively, both from KU Leuven (Belgium). He is currently an industrial research manager at the research group COSIC. His main research interests are cryptography, security and privacy of wireless networks, key management, distance bounding, cryptographic authentication protocols, and security solutions for medical devices.