Open Access

Open Journal of Internet of Things (OJIOT)
Volume 7, Issue 1, 2021
http://www.ronpub.com/ojiot
ISSN 2364-7108

Hierarchical Data Integrity for IoT Devices in Connected Health Applications

Maryam Karimi, Prashant Krishnamurthy

Department of Informatics and Networked Systems, School of Computing and Information
University of Pittsburgh, 135 N Bellefield Ave, Pittsburgh 15213, PA, USA, {maryam.karimi, prashk}@pitt.edu

Abstract

Internet of things devices are increasingly replacing expensive monitoring devices in many environments such as healthcare. People can eventually own their data, collected from smart personal devices, store them in a variety of cloud services, and make them available to service providers of their choice. In such cases, whenever service providers use these data to provide appropriate services, the data owner may become responsible for ensuring the integrity of data retrieved from multiple points. We present a Hierarchical Data Integrity (HDI) approach to verify if the data, sent by monitoring devices to the cloud, remain unchanged. It is hierarchical as follows: there is a quick verification of the integrity of recent health data (in less than 1 *ms), followed if necessary by a low overhead secure option for verifying the integrity of both recent and historical data (still only in* 6.1 *ms). Further, the hierarchy allows granular identification of data units that fail integrity checks, without requiring any key sharing. It is possible for a data owner to periodically (randomly) use a more secure process to verify the integrity of data. This reduces the computation, storage, and time of integrity verification as shown by analysis, simulation, and hardware implementation.*

Type of Paper and Keywords

Regular research paper: *data verification, data integrity, data ownership, Internet of Things, IoT security*

1 Introduction

Internet of Things (IoT) devices are increasingly being used to supplement or replace expensive monitoring devices. In the near future, the environment around us is likely to see a profound transformation that can lead to a better quality of life, higher efficiencies through less waste, and reduced costs. This transformation (described in [58] for healthcare and paraphrased here) will involve user ownership of data that is generated (mostly) on their premises by a variety of wearable devices ("things" in IoT) and sophisticated, yet not very expensive connected health devices. In examples of healthcare scenarios, the connected health devices could include blood pressure monitoring[1], electrocardiogram sensors[2], ultrasound probes[3] and even inexpensive DNA sequencing chips[4] that can identify inherited genes and chromosomes (that may impact a patient's well being through drugs that are effective or alert them to those that are dangerous to particular classes of patients). Although ensuring the self-integrity[5] of data being transmitted

[1] `https://health.nokia.com/us/en/blood-pressure-monitor`
[2] `https://www.alivecor.com`
[3] see `https://www.lumify.philips.com/web/`
[4] `http://www.thermofisher.com`
[5] The data are correct and there are no flaws in generating or measuring data at any time so that the knowledge that is gained from different parts of the data (measurements of different variables or different times of the same variable) make sense.

from the health monitoring devices is not within the scope of this paper, recent advances show that there is no significant performance degradation between wearable devices and expensive medical tracking devices [60]; therefore, the information coming from an IoT device may be equivalent in merit to the information generated by dedicated expensive medical devices. Much of these data, while they may eventually be owned by a patient in an unforgeable blockchain, are today maintained in separate (perhaps multiple) cloud services that are owned by the vendors of the smart devices. The vendors may themselves be leasing storage, computation, and algorithms from various cloud services like Amazon's AWS or Microsoft's Azure or even lesser-known companies[6].

Data may have been corrupted in one or more of the cloud services either accidentally or intentionally (attacks). In either case, *assessing the integrity of owner-provided data, which resides in a variety of manufacturer or vendor-owned cloud servers is an important challenge*. In [58], one of the envisaged scenarios includes the data owner's (e.g., patient) ability to go to different providers at will, since they own the data and are able to easily provide it to any service provider (e.g., healthcare provider) of choice. While each healthcare provider may set up relationships with some of the device manufacturers, it is easier if the data owner, who is at liberty to pick the manufacturer or provider, can provide an assessment of the integrity of the data retrieved from the cloud.

We present a fresh approach to verify if the data, sent by monitoring devices to the cloud, remain unchanged which we call Hierarchical Data Integrity or HDI. An obvious way to verify the integrity of data from a single source, stored in a single cloud, is to use digital signatures on digests or integrity checks on each data unit [56]. However, this method is not efficient when the data may change rapidly over time. Simple IoT devices may not be able to do the extensive computation, store all data, or participate in sophisticated cryptographic protocols including the key establishment or certification verification. Furthermore, devices from different vendors may send the data to the cloud service associated with those providers – and so any integrity verification method should be able to verify data from multiple sources, stored in multiple cloud servers. As described later, we assume an architecture where the personal devices connect to a trusted *owner gateway* (OGW) that forwards the data to the cloud/server. The integrity of data, retrieved by a service provider through a *user gateway* (UGW) can be verified at the owner's gateway in a hierarchical manner. We interchangeably call this service provider as trusted caregiver or third party service provider below.

The data are maintained at different levels. Briefly, the data are split into blocks, all blocks are inserted into a concatenated Bloom Filter (explained later) and further, a tree is created using the hash values of the blocks. The Bloom Filter is stored in the owner's gateway and has a small footprint. The leaves of the tree are omitted to save on storage, but a copy of the rest of the tree (which includes keyed-hash values (HMAC)) is stored in the cloud. To check if the data remain unchanged, the owner's gateway first checks whether retrieved blocks "belong" using the Bloom Filter. If so, data integrity is assumed (there will be a very small false-positive rate). If the data are found not to belong (due to the concatenated Bloom Filter features/limitations or because the data have been modified accidentally or otherwise), or need to be checked further periodically even if they belong, part of the tree associated with the retrieved blocks will be rebuilt and checked against the stored tree. This method can efficiently distinguish the source of mismatch even with dynamic data. Considering the fact that the hash key remains only in the owner's gateway, this method does not require any key establishment between any parties. This hierarchical scheme of the tree and Bloom Filter reduces the time to verify integrity, storage space, and computational complexity, at the price of introducing False Negatives (FN) (i.e., records that exist but may fail an integrity check).

One may argue that the alternative simple solution for verifying integrity in the cloud is to keep a list of Message Authentication Codes (MACs) in the cloud along with the data. However, it has multiple problems that are solved with HDI. In our scheme, the owner's gateway is the only fully trusted party that can change the data and the temporary access that the trusted caregiver or service provider receives does not allow it to change the data since the key is only kept in the owner's gateway. In contrast, consider storing a list of MACs in the cloud instead of using HDI. If we use unkeyed hash functions, the cloud and whoever has access to the cloud server (e.g., in the proposed scenario, in the presence of secure access control, any caregiver can do this) can change the data. If we use keyed hash functions (without performing any key distribution) the service provider's UGW will have to send the whole data block to the owner's gateway for verification which increases the communication cost between UGW and OGW considerably. Otherwise, the OGW will have to share a key with the UGW which is a complex process (also we do not necessarily trust the UGW with the key) that we are avoiding through HDI.

Security assumptions: HDI relies on a trusted

[6] It has been reported that increasingly companies are relying on multiple cloud providers (on average 8) for services [39].

verifier (OGW) that belongs to the owner of the data (e.g., patient). The OGW stores the data in (many) cloud servers and generates and stores the key that can verify the integrity of data. A trusted third-party (UGW) that wants to retrieve the data, checks with the owner of the data through the gateway (OGW) to ensure that the data remain unchanged compared to what the data owner stored in the cloud. The UGW only shares small metadata that is sufficient for the OGW to verify the integrity of data to be used by the UGW. Under these constraints, HDI is able to detect accidentally corrupted, forged, or fabricated data sent to the third-party retriever (UGW) instead of the original data. We assume that OGW is the trusted secure verifier and since UGW is the third party that requests the data, it only makes sense to trust this entity with sending the *correct* metadata to the verifier.

Contributions: HDI provides a solution that can store the owner's data, gathered from multiple sources, in multiple clouds and verify the data integrity, when a service provider retrieves it. The contributions of this HDI method are:

- Each data block is hashed into an empty fixed-size Bloom Filter. This Bloom Filter is hashed into a Concatenated Generalized Bloom Filter (CBF). The two-level Bloom Filter reduces the communication cost of data integrity verification.
- To resolve the false negatives in the CBF, a hash tree for blocks of data is created and stored in the cloud to identify the corrupted blocks quickly.
- As far as we know, this is the first method that specifies the ability to verify the integrity of data from multiple sources, stored in multiple clouds, without requiring explicit key distribution. All processes are designed in a way to avoid any key establishment. The key remains only in the owner's gateway.
- Analysis and simple hardware implementation show that HDI performs better than or as well as previous methods and is efficient in time, storage and communication overhead.

The rest of the paper is structured as follows: Section 2 discusses the background material on Bloom Filters and hash functions. Section 3 describes the proposed method to solve the problem. Section 4 analyzes our provided solution, Hierarchical Data Integrity (HDI). In this section, we present experimental results and compare our provided method with previous methods. Section 5 provides a literature review and Section 6 discusses the security of gateways, the consequences of the gateways getting compromised, and how to secure the gateways. Finally, Section 7 concludes the paper and summarizes our contributions.

2 Background

We use Bloom Filters and secure hashes to ensure data integrity verification. Here we provide a brief background to keep the paper self-contained. Please check references ([13], [35], [45]) for details.

Bloom Filters: Bloom Filters (BFs)([13]) have been used for set membership queries. A traditional BF begins with an array of all 0s and then hashes all of the set members using k different hash functions (not necessarily secure hashes). Each hash yields a bit in an array called the Bloom Filter. To check if a given element exists in the set, the element is hashed using all k functions and the bit map is compared to the array. If the same bits in the array were set to 1, the element exists in the set. BF allows false positives, but it saves on storage and search space.

BFs have been used for password security checks [55, 40], spell checks [42, 47], speeding up database semi-join operation [47, 12, 59], distributed caching [24, 50], resource discovery, and routing [13, 57], checking image existence [51], etc. Special kinds of Bloom Filters exist for different types of applications such as key-value-supported BF [63, 62] used for checksums, counting BFs [24], and scalable BFs [2] that split the filter space into k hash functions to make them scalable. Also [37, 36, 32], introduced methods to reduce the rate of false-positive errors in Bloom Filters.

Considering that the best number of hash functions k is $k = m \cdot \ln(2)/n$ (see [13]), the minimum required space in the BF for storing n items is m, which is calculated as $m \geq n \log_2 e \cdot \log_2(1/\epsilon)$ [13], where ϵ corresponds to the maximum fraction of the universe of false positives that is tolerable. As an example, if we use $n = 12$ and allow at most 1% false positive, m should be at least 115 bits.

In the Generalized BF (GBF) [35], the initial filter is randomly filled with 0's and 1's. Then, k_1 hash functions set the bit and k_2 other hash functions reset the bits. This method limits false positives, but it introduces false negatives. Concatenation of GBFs (CBF) [45] which consists of multiple GBFs improves robustness and capacity since there is less insertion and therefore fewer false negatives in each sub-filter.

Plain BFs are not enough and do not perform well for three reasons: they are not well structured, nor well organized, and data cannot be deleted or modified in the case of change, because they remember old data. For instance, all of the bits in the BF can eventually become 1 which would defeat the purpose of the BF. GBFs (and therefore CBFs) on the other hand tend to forget old data. Thus CBFs also address the problem of advertising saturated filters (all-one attacks). In this paper, we use CBFs and we assign a dedicated part (subfilter) for each device that generates data; therefore, for each device, the overall BF has the most recent data. Further, a device that generates data more frequently does not overwrite data from other devices that may make less frequent updates. The False Negative probability (FN) and False Positive probability (FP) calculation is discussed in Section 3.6.2, using formulas from [35]).

Hash functions: Simple unkeyed hash functions (e.g., MD-5 - that are also not secure) are used in BFs. We use secure hashes – SHA-3 [8] (unkeyed) and HMAC [34] (keyed) hash functions to build a secure tree which we employ in data integrity verification in this paper.

3 How HDI Works

In this section, we provide a high-level idea of how HDI works. Figure 1 shows the motivating scenario where health tracking devices are connected to a trusted owner's gateway (that can even be the data owner's cellphone) called OGW, which forwards data to the appropriate cloud servers. Later some or all of the data are retrieved by the trusted third party data user (service provider that requires data to provide service to data owner) through a user's gateway (UGW). The UGW and OGW interact to verify the integrity of retrieved data (the onus is on the OGW). In HDI, the OGW hashes data blocks when they come from the health monitoring devices into a concatenated Bloom Filter – CBF which is stored locally in the OGW (as calculated in Section 3.6.2, this can be as small as 10700 bits). The OGW also creates a (keyed) hash tree to keep the (secure) hash values of the blocks in a *structured* manner. The OGW removes the leaves of the tree and sends them to a preferred cloud server for storage. When the need arises to verify the integrity of data, the OGW checks the retrieved blocks using the locally stored CBF. If the data have been forgotten by the CBF, the OGW can retrieve the tree from the preferred cloud server and use the tree to verify the integrity of the associated data blocks. This distinguishes the source of mismatch up to a block of data. If data are forgotten in the CBF due to aging and the tree can verify the data, no action needs to be taken, otherwise, the owner and/or the trusted third party user is alerted to the accidental or malicious change.

3.1 Threat Model

We have the following elements in our scenario:

- Adversary: This entity sends corrupted data to the third party (as an example, in the healthcare scenario, this hurts the patient because the doctor has incorrect information). This adversary can be of two forms: (1) The cloud server storing the data, which may be under attack, itself wants to forge the data or accidentally provides corrupted data blocks that harm the data owner. (2) An adversary that is a man-in-the-middle that sends wrong data to the third party instead of the cloud server and thus harms the data owner.

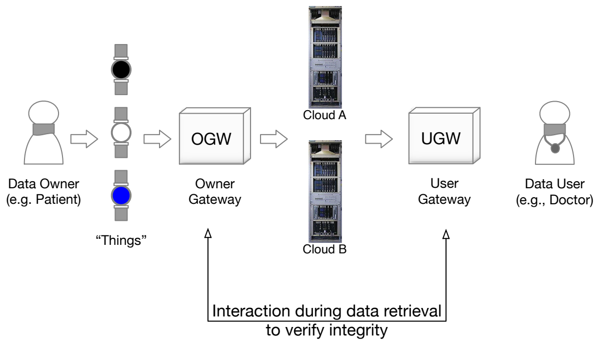

Data Owner (e.g. Patient)
"Things"
OGW
Owner Gateway
Cloud A
Cloud B
UGW
User Gateway
Data User (e.g., Doctor)
Interaction during data retrieval to verify integrity

Figure 1: Motivating scenario

- Fully Trusted OGW: This entity is the data owner's gateway which is secure and trustworthy. It stores the generalized Bloom Filter and the key used for HMACs in building the tree. It performs data updates by storing data in the cloud, hashing them into the Bloom Filter prior to storage, and also hashing them securely in the tree.

- Trustworthy UGW: The UGW is the entity that needs to retrieve and use the data to service the data owner. If the UGW is not trustworthy it does not have to get any data and ask for verification from the OGW. It can harm the data owner directly without any of these hassles; therefore, the only scenario that makes sense is to assume that the UGW is trustworthy.

Some more discussions of the security implications and behavior of the OGW and UGW are included in Section 6.

3.2 Formal Preliminaries

The HDI system's protocols for storing data are shown in Figure 2 and for verifying the integrity of retrieved data in Figure 3 and have the following functions:

- $KeyGen(1^\lambda) \rightarrow \kappa$: The OGW Generates a random private key κ of size of λ from the keyspace K. κ is

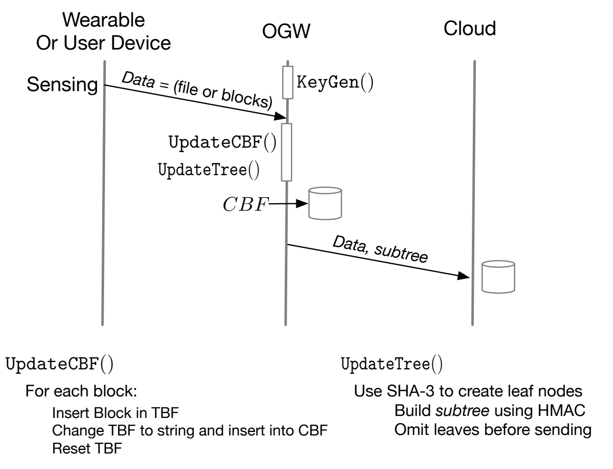


Figure 2: Data storage protocol

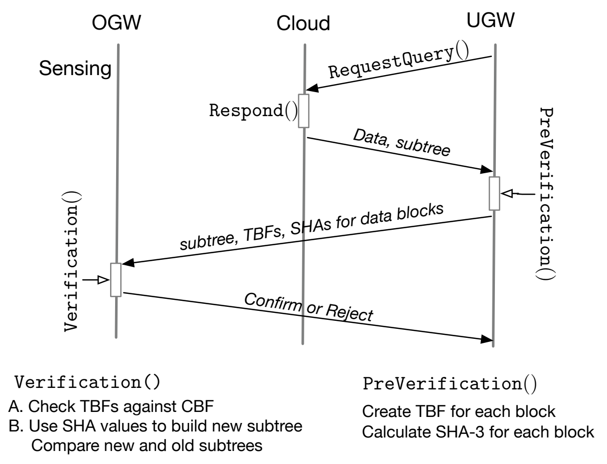

OGW
Cloud
UGW
Sensing
RequestQuery()
Respond()
Data, subtree
PreVerification()
subtree, TBFs, SHAs for data blocks
Verification()
Confirm or Reject
Verification()
A. Check TBFs against CBF
B. Use SHA values to build new subtree
Compare new and old subtrees
PreVerification()
Create TBF for each block
Calculate SHA-3 for each block

Figure 3: Data verification protocol

used for building tree hash values using HMAC [7].

- $UpdateCBF(n, \{F_i\}_{i=0}^{n}, k, \{H_i\}_{i=0}^{k}, CBF, k_0, k_1, \{H_i'\}_{i=0}^{k_0}, \{H_i''\}_{i=0}^{k_1}) \to CBF$: This process is performed in the OGW and it updates the Concatenated Bloom Filter. Here, F is the file that has n blocks (F_is) of data. Each block has information about time and ID of the device that created the data. k is the number of hash functions in ($\{H_i\}_{i=0}^{k}$) which are used to set bits in a simple Temporary Bloom Filter (TBF) (explained in Section 3.3). CBF is the main Bloom Filter, k_0 is the number of hash functions ($\{H_i'\}_{i=0}^{k_0}$) that set bits in the CBF, and k_1 is the number of hash functions ($\{H_i''\}_{i=0}^{k_1}$) that reset the bits in the CBF. This `Update Concatenated Bloom Filter` function is implemented using Algorithm 1.

- $UpdateTree(n, \{F_i\}_{i=0}^{n}, subtree, \kappa, d) \to subtree$: This process is done in OGW. F and n are defined previously. $subtree$ is received from the cloud and will be updated using SHA-3 values of the tree nodes and HMAC values of the cluster of nodes (nodes with similar attributes (device ID, creation time, etc.)). The key κ is used in the HMAC and d is the maximum number of children each node in the tree can have. The `Update Tree` function is implemented using Algorithm 2.

- $RequestQuery(time, ID_{device}) \to ID_{cloud}$: This function finds the corresponding cloud based on the requested data ID_{device} (the ID of the device that generated that data). UGW forwards the request for the data from the specific device in a specific timeline to the corresponding cloud services.

- $Respond(time, ID_{device}) \to \{F_i\}_{i=0}^{n}, n, subtree$: The cloud responds to the UGW with the matched blocks and number of those blocks along with the $subtree$ corresponding to those blocks.

- $PreVerification(n, \{F_i\}_{i=0}^{n}, k, \{H_i\}_{i=0}^{k}, subtree) \to \{TBF_i\}_{i=0}^{n}, \{\text{SHA-3}(F_i)\}_{i=0}^{n}, subtree$: This process is done in the third-party user gateway (UGW). $\{F_i\}_{i=0}^{n}$ includes the set of blocks that are retrieved, n is the number of blocks, k is the number of hash functions and, $\{H_i\}_{i=0}^{k}$ are the hash functions for simple Bloom Filter. The output values includes the simple Bloom Filters ($\{BF_i\}_{i=0}^{n}$) and SHA-3 values ($\{\text{SHA-3}(F_i)\}_{i=0}^{n}$) for each block and the $subtree$ are sent to the OGW. `Pre-Verification` function is implemented using Algorithm 3.

- $Verification(n, \{TBF_i\}_{i=0}^{n}, CBF, k_0, k_1, \{H_i'\}_{i=0}^{k_0}, \{H_i''\}_{i=0}^{k_1}, \{\text{SHA-3}(F_i)\}_{i=0}^{n}, subtree, \kappa) \to \{0, 1\}$: The OGW receives the SHA-3 values, the $subtree$ and simple Bloom Filters ($\{TBF_i\}_{i=0}^{n}$) from the UGW and verifies the data. The OGW uses SHA-3 values, the HMAC function, and the key (κ) to rebuild a tree and compares it to the received $subtree$. The output is reject (0) or verify (1). `Verification` function is implemented using Algorithm 4.

The algorithm for updateCBF, UpdateTree, Pre-verification, and Verification methods are described in the following corresponding sections.

[7] We assume that the key size is commensurate with the needed security.

3.3 Creating and Updating Data, Bloom Filter, and Hash Tree

HDI executes functions to create the CBF, hash tree, and store the very first data from the wearable devices. Further updating the data includes *inserting, deleting*, or *modifying* a block of the data in the cloud. The OGW has a secret key that belongs and is known only to itself. The way data blocks, the Concatenated Bloom Filter, and Tree are created and stored is described in Figure 2. When the owner requests updating the data (e.g., when the wearable device generates new data), the trustworthy OGW, that has access to the HMAC key, starts the process. The OGW updates the CBF using the function `UpdateCBF()` and and the tree using `UpdateTree()`. We recall that the CBF is a coarse rendition for integrity checks while the tree can identify specific data blocks. The OGW receives the data in blocks from a device, passes the data blocks through k hash functions to update the corresponding sub-filter $\in C$ sub-filters in the CBF, updates the tree using SHA-3 values of new data blocks, and sends the data and the updated tree to the cloud. Algorithm 1 and Algorithm 2 show these procedures and in the following, they are explained with an example.

Algorithm 1 is described with the example in Figure 4, Stages 1 and 2. These stages are applicable whenever data blocks arrive, irrespective of whether it is at the start or later. The OGW receives n data blocks from one or more devices (e.g., $n = 12$ blocks). As shown in Line 4 Algorithm 1, these data blocks are first hashed into a fixed-size temporary BF (TBF) that has been previously reset (the fixed size is 20 bits – please see Section 3.6.1 for justification). Then the OGW treats this TBF as a string of 0's and 1's and hashes this string into a CBF (lines 5 and 6). This CBF has subsections (sub-filters) for different devices; therefore, the most recent data from each device is available at any time (we note again that the CBF tends to forget old data). This procedure is performed for the insertion and modification of data blocks. Since the CBF tends to forget old data, we do not explicitly consider updating the BFs in the case of deletion of data in the cloud. The TBF will be cleared to be used by future data blocks (line 7).
In the following, we explain the differences in creation, insertion, deletion, and modification of the tree or subtree, which is more involved than the CBF.

New tree: We first describe how the tree is created using Figure 4 - Stage 3. At the start-up of the entire process when the first data blocks arrive and the tree is null, in order to build the tree, the OGW calculates the SHA-3 value for each block of the data as the leaves of the tree (e.g., Figure 4 Stage 3, node 1, through node 12). The reason for using plain SHA-3 is as follows. Later during data integrity verification, SHA-3 is used in the user's gateway – UGW; therefore, for the first level, we choose an unkeyed, yet secure, hash function to avoid the expensive and complicated process of key establishment/state between an OGW and potentially different UGWs. Next, nodes with the same device ID and year are grouped and hashed using HMAC (e.g., 1,2,3 and 5 from device B and year 2020) and a secret κ and a new node is created as their parent and tagged with the year and device ID (e.g.,2020-B); therefore no entity other than the OGW can manipulate these nodes. Then nodes with the same device ID (e.g., B) are grouped and a new node is created as their parent and tagged with the device ID. Finally, all nodes are merged into a single root node (Peter)[8].

Algorithm 2 is described with an example similar to that in Figure 4 – Stage 3, where the OGW updates the hash tree. The OGW receives new data blocks from devices, the related subtree retrieved from the cloud (the subtree here is the set of nodes in the path from the possible parent of the new node to the root along with its siblings), and sibling data blocks. The node will be added to the tree under the proper device and time (line 4, Algorithm 2). Each node in the tree stores the block ID, date, device ID, and hash value. The OGW starts updating the tree by calculating SHA-3 values for new blocks and requested sibling blocks (line 6). Then, the related subtree is updated using HMAC and the key κ on the calculated SHA-3 values (line 7). It continues using HMAC for updating the upper nodes until reaching the tree's root. The leaves are deleted from the tree to save space.

Subtree creation & storage: The tree is then divided into subtrees (based on the device that created the data. Each subtree includes the data created by one device along with the siblings of the topmost node, e.g., the second subtree includes the nodes created using data blocks from device B (nodes 2020-B, 2021-B, and B) along with B's siblings (nodes A and C).)[9]. The OGW then sends the updated subtree along with the new data to the preferred cloud server (e.g., the second subtree is sent to cloud server B in Figure 5 (b)). The tree helps us to check data integrity and reduce the effect of false negatives (FN) in the local CBF as we will see later. The key κ and the CBF remain in the OGW.

[8] In stage 3, node 6 in the tree is in a rare situation, in which, it is the only child of that branch. In this case, nodes merge together; therefore, node 6 (which replaced A) is consist of the HMAC value of the SHA-3 value for the block 6.

[9] It is OGW's responsibility to update node A and C when they changed in the corresponding cloud server

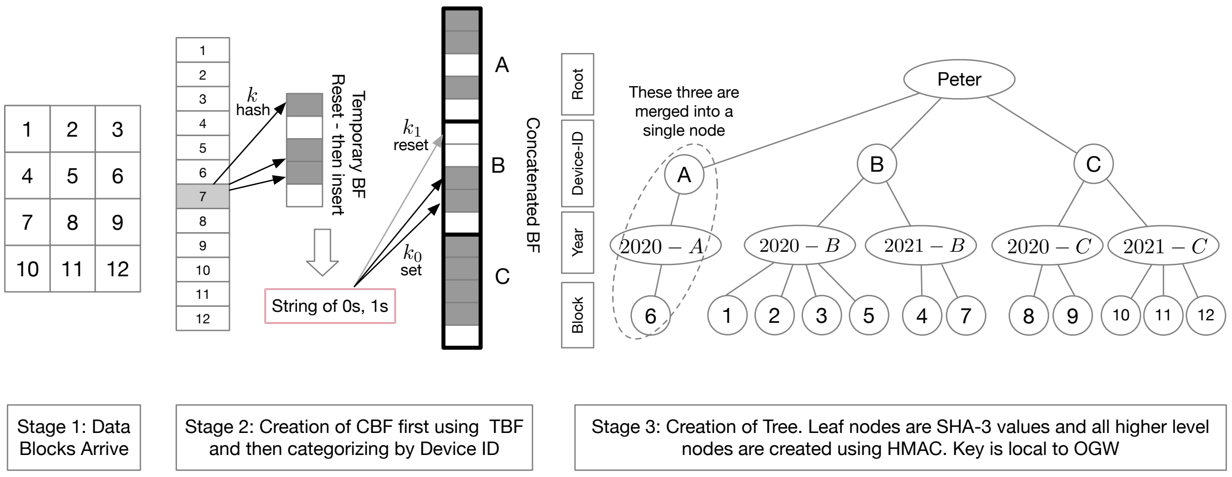
Algorithm 1: `Update Concatenated Bloom Filter`

```
1: input: $(n, \{F_i\}_{i=0}^{n}, k, \{H_i\}_{i=0}^{k}$, CBF, $k_0, k_1, \{H'_i\}_{i=0}^{k_0}, \{H''_i\}_{i=0}^{k_1})$
2: output: CBF
3: for each $i \in \{1 \dots n\}$ do
4:     $TBF : \{\forall j \in \{1 \dots k\} : \upsilon(H_j(F_i)) \leftarrow 1\}$    ▷ Hash $i^{th}$ block of data in the temporary Bloom Filter (TBF)
5:     $CBF : \{\forall j \in \{1 \dots k_0\} : \upsilon(H'_j(TBF)) \leftarrow 1\}$    ▷ Hash TBF in Concatenated Bloom Filter (set bits to 1 for $k_0$ number of hash functions)
6:     $CBF : \{\forall j \in \{1 \dots k_1\} : \upsilon(H''_j(TBF)) \leftarrow 0\}$    ▷ Hash TBF in Concatenated Bloom Filter (reset bits for $k_1$ number of hash functions)
7:     $TBF : \{\forall j \in \{1 \dots size_{TBF}\} : \upsilon(j) \leftarrow 0\}$    ▷ Reset the TBF for future blocks
8: end for each
```

*$\upsilon(i)$ denotes the value of bit i in the corresponding Bloom Filter

Algorithm 2: `Update Tree`

```
1: input: $(n, \{F_i\}_{i=0}^{n}, subtree, \kappa, d)$
2: output: subtree
3: for each $f \in \{F_i\}_{i=0}^{n}$ do    ▷ For each new block
4:     insert the node $n$ under the proper node    ▷ Proper node is leaves' parent (second layer) with proper device ID and date
5:     split the branch if the number of nodes exceeds $d$    ▷ Split process is shown in Figure 7 and Section 3.5
6:     s ← SHA-3(f)    ▷ Calculate SHA-3 value for the new block
7:     $p \leftarrow HMAC((s|\forall x \in \{siblings\} : s = s\oplus$ SHA-3$(x)), \kappa)$    ▷ Recalculate HMAC value for parent using children's SHA-3
8:     for each np in the path from n to subtree root do
9:         $np \leftarrow HMAC((s|\forall x \in \{npsiblings\} : s = s \oplus x), \kappa)$    ▷ Update HMAC value for nodes in the path to the root
10:    end for each
11: end for each
12: omit leaves from the subtree
```



1 2 3 4 5 6 7 8 9 10 11 12
k hash
Temporary BF Reset - then insert
String of 0s, 1s
k_1 reset
k_0 set
A
B
C
Concatenated BF
Root
Device-ID
Year
Block
These three are merged into a single node
Peter
A
B
C
$2020-A$
$2020-B$
$2021-B$
$2020-C$
$2021-C$
6 1 2 3 5 4 7 8 9 10 11 12
Stage 1: Data Blocks Arrive
Stage 2: Creation of CBF first using TBF and then categorizing by Device ID
Stage 3: Creation of Tree. Leaf nodes are SHA-3 values and all higher level nodes are created using HMAC. Key is local to OGW

Figure 4: Data/CBF/Tree creation

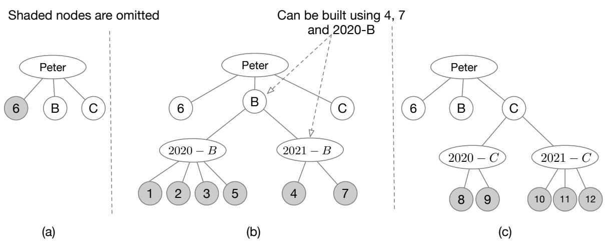


Figure 5: Subtrees that are stored in cloud servers A, B & C

Insertion: In the case that the tree already exists and we want to update it, different scenarios are possible that we explain using (Figure 6). In this figure, all nodes that are involved in the recalculation of the tree are shaded grey. The nodes that actually change are shown with a yellow background. To insert a node (say node 13 in Figure 6.top), the SHA-3 of node 13 should be calculated, the nodes in the path to the root (nodes 2021-B, B, Peter) should be updated, and HMAC values should be recalculated. Consequently, the OGW requires all the siblings, including the SHA-3 values of nodes 4 and 7, as well as the node 2020-B, and nodes 6 and C.[10] After insertion, if the number of children exceeds d ($d = 4$ in this example), the tree is split as described in Split Procedure in Figure 7 and Section 3.5.

Deletion & modification: Figure 6.d depicts deletion. If there is a need to delete node 5, the gateway (OGW) needs to recalculate the HMAC values to re-create a node along the path from 5 to the root which includes 2020-B, B, and Peter; therefore, the OGW requests their Siblings (data blocks 1, 2 and 3, as well as node 2021-B, and nodes 6 and C) from the cloud server. In the case of modification, the same procedure is followed, but the OGW updates node 5 instead of removing it, and here, the gateway needs to recalculate the SHA-3 value for node 5 as well.

3.4 Data Integrity Verification

Verification of data integrity occurs when a third-party user of the data retrieves it from the cloud servers. Integrity verification is performed by the gateways

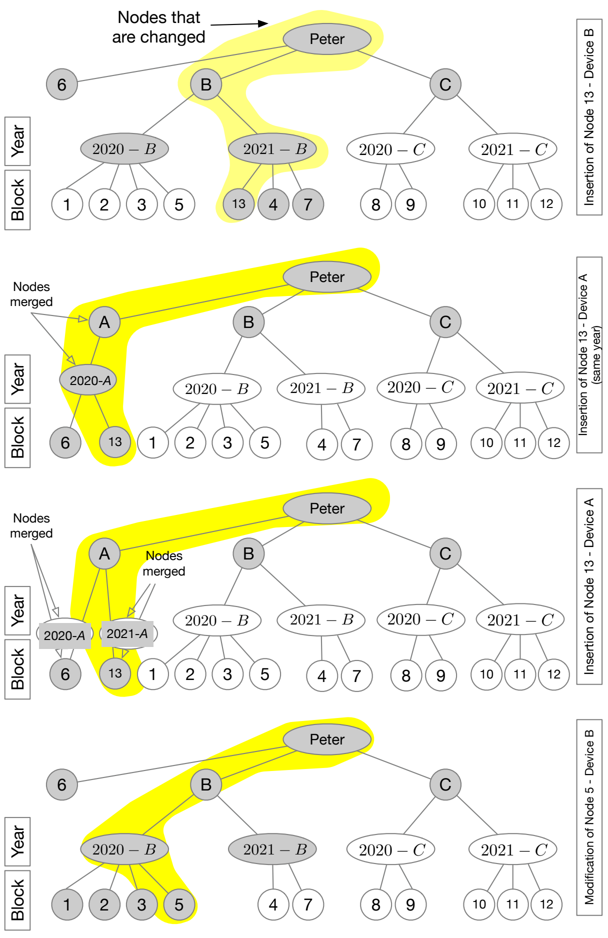

Nodes that are changed
Nodes merged
Peter
6
A
B
C
2020-A
2021-A
2020 – B
2021 – B
2020 – C
2021 – C
Year
Block
Insertion of Node 13 - Device B
Insertion of Node 13 - Device A (same year)
Insertion of Node 13 - Device A
Modification of Node 5 - Device B

Figure 6: Updating the tree: shaded nodes are needed for the update and nodes with the yellow background are changed

(OGW and UGW) at this time. We assume that the OGW has authenticated the UGW to the cloud servers [11].

The procedure for verifying the integrity of the retrieved data blocks is described in Figure 3. The user's gateway – UGW – requests data from a cloud provider. The cloud provider sends the requested data blocks along with sibling data blocks *and* the subtree associated with the data to the user's gateway. The subtree includes the HMAC values of parent nodes that are in the path between the requested block to the root of the tree along with the siblings of the nodes in the path. As an example, Figure 8 shows the requested data block, sibling data

[10] If the node is being inserted to a node that was previously merged with the parent, the tree splits into two branches, and different split points can happen based on the features of the new data. As an example, in Figure 6.b, the new node (13) and node 6 are both created in 2020, so both nodes will be children of 2020-A (2020-A merges with A). In a different situation, for example in Figure 6.c, the new node is created in 2021. In this case, node A will have two children (2020-A and 2021-A) that each of them will have one child and are merged with their child (6 and 13 respectively).

[11] To authorize and control access of the user, the owner's gateway may act as a temporary certificate authority and generate a one-time password and share it with both parties (user and server). That password can be used as an initial temporary key that helps the cloud server to authenticate the user. It also can be used as an initial key to set up connection keys and establish a secure connection between the user's gateway and the server or using TLS. This is outside the scope of this paper

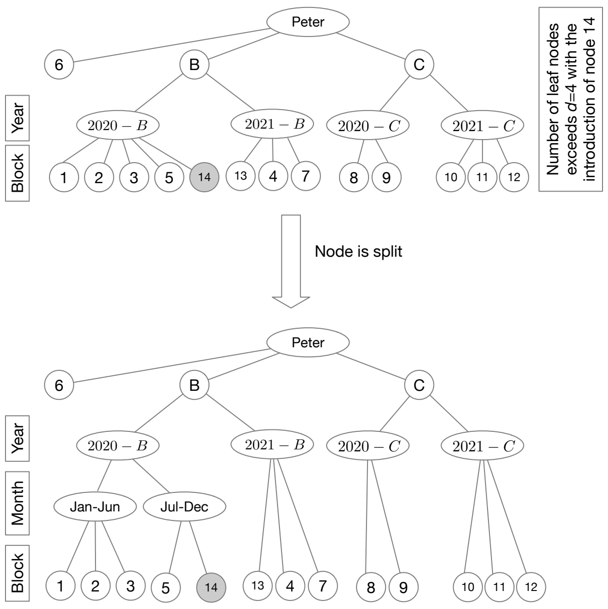


Figure 7: Splitting nodes when the number of children exceeds d (here $d = 4$)

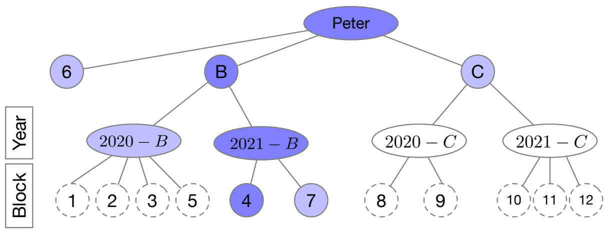


Figure 8: Subtree for retrieving block number 4. The darker nodes are in the path between node 4 and the root. Lighter nodes are the siblings of the path nodes. Dashed nodes are SHA-3 values that are omitted from the tree; however, the cloud sends the data blocks (not the SHA-3 values) of nodes 4 and 7 along with the subtree.

block, and the subtree when data block 4 is requested by the UGW.

The UGW and the OGW share the same hash functions that are used to create the CBF (these are unkeyed and can be public). As shown in Algorithm 3, there is a pre-verification process at the UGW. In line 4 Algorithm 3, for each block of data, the UGW creates a simple Bloom Filter – this process is identical to the TBF creation. It then calculates the SHA-3 values of retrieved data nodes (line 5) and sends them to the OGW for verification along with the subtree. The SHA-3 values of the data nodes need to be sent to the OGW for it to rebuild the subtree if necessary. Note that the entire data need not be sent from the UGW to the OGW - this saves on communication costs. Next, the OGW verifies the integrity of data as shown in Algorithm 4. The OGW checks the received TBF's against the locally stored CBF (lines 3 to 6) . If there is no match (and this may happen if the data are old), then the gateway rebuilds a new tree with the received SHA-3 values and compares it with the received tree (lines 8 to 11). If they match, a confirmation is sent otherwise the data is rejected and an alert is issued. In the following, this procedure is explained in more detail with an example.

In Figure 9, the user requests the owner's data for the year 2021 from devices A and B (blocks 4, 6, and 7 – see the top of the figure). The UGW sends the request to the cloud (steps 1,2). The cloud sends the data along with the related subtree to the UGW (step 3). The UGW hashes the received data into three TBFs (for the three blocks) and also calculates the SHA-3 values (step 4) of the data blocks. The UGW sends the TBFs, hash values, and the subtree to the OGW (step 5). The OGW verifies the integrity of data by comparing the received BFs' strings against their dedicated parts (sub-filter) of the locally stored CBF (based on the device ID) (step 6-A). New data is verified with a higher probability. If the data are very old, the CBF may have forgotten them. Alternatively, if the data are old, corrupted, or cannot be verified using the CBF, the OGW uses a rebuilt tree to verify the data (step 6-B). For this, the OGW uses the hash (SHA-3) values to rebuild the tree. The OGW compares the rebuilt tree with the received subtree. If the comparisons do not match, the OGW sends a rejection message to the UGW; otherwise, the OGW sends a confirmation message to the UGW (step 7). The UGW sends the blocks to the user (Doris) (step 8).

3.5 Refinements for Efficiency

In order to reduce the amount of data required to be fetched from the server for verification, we suggest choosing d (the number of children for each parent in the tree) between 2 and $\log(n)$. The lower bound is 2, in view of the fact, that it is more efficient to merge the child and parent into a single node if a node has only one child. The upper bound is $\log(n)$, since it limits the number of extra sibling data blocks that have to be fetched when data is being retrieved and verified; therefore, the communication complexity between the cloud server and UGW would be more efficient and is limited to $O(\log(n))$ (considering that subtree size complexity is also $O(\log(n))$ -discussed in Section 4.1). As shown in Figure 7, if any node has more than d children, the tree splits into two branches (in this example, we selected d to be 4). Here we have shown this split happening between the first six months and the next six months of the year for illustration only - the split can be accomplished using

Algorithm 3: `Pre-Verification in UGW`

```
1: input: $n, \{F_i\}_{i=0}^{n}, k, \{H_i\}_{i=0}^{k}, subtree$
2: output: $\{TBF_i\}_{i=0}^{n}, \{\text{SHA-3}(F_i)\}_{i=0}^{n}, subtree$
3: for each $i \in \{1 \dots n\}$ do                                  ▷ For each new block
4:     $TBF_i : \{\forall j \in \{1 \dots k\} : \upsilon(H_j(F_i)) \leftarrow 1\}$   ▷ Hash $i^{th}$ block of data in the temporary Bloom Filter
5:     $SHA\text{-}3_i \leftarrow SHA\text{-}3(F_i))$                  ▷ Calculate SHA-3 values
6: end for each
```

*$\upsilon(i)$ denotes the value of bit i in the corresponding Bloom Filter

Algorithm 4: `Verification in OGW`

```
1: input: $n, \{TBF_i\}_{i=0}^{n}, CBF, k_0, k_1, \{H_i'\}_{i=0}^{k_0}, \{H_i''\}_{i=0}^{k_1}$,
   $\{\text{SHA-3}(F_i)\}_{i=0}^{n}, subtree, \kappa$
2: output: $\{0, 1\}$
3: for each $i \in \{1 \dots n\}$ do
4:     $result_i \leftarrow min(\{\forall j \in \{1 \dots k_0\} : min\{\upsilon(H_j'(TBF_i))\}\}, \{\forall j \in \{1 \dots k_1\} : min\{1 - \upsilon(H_j''(BF_i))\}\})$
       ▷ Check if the received TBF exists in CBF
5: end for each
6: $resultBF \leftarrow \{\forall i \in \{1 \dots n\} : min\{result_i\}\}$   ▷ resultBF is 1 if all blocks' TBF exist in CBF
7: if $resultBF \neq 1$ $Or\ randomly$ then   ▷ if the data are old or higher accuracy is required and 1% false negative is not tolerable
8:     Build a new-subtree using SHA-3 values:
9:         Merge the SHA-3$_i$ values that have the same device-ID and time interval and create HMAC parents
10:        Merge HMAC parents with the same device-ID and create a parent for them
11:    $resultTree \leftarrow compare(subtree, new\text{-}subtree)$ ▷ Compare the built subtree and received subtree, output is 1 if they are equal
12: end if
```

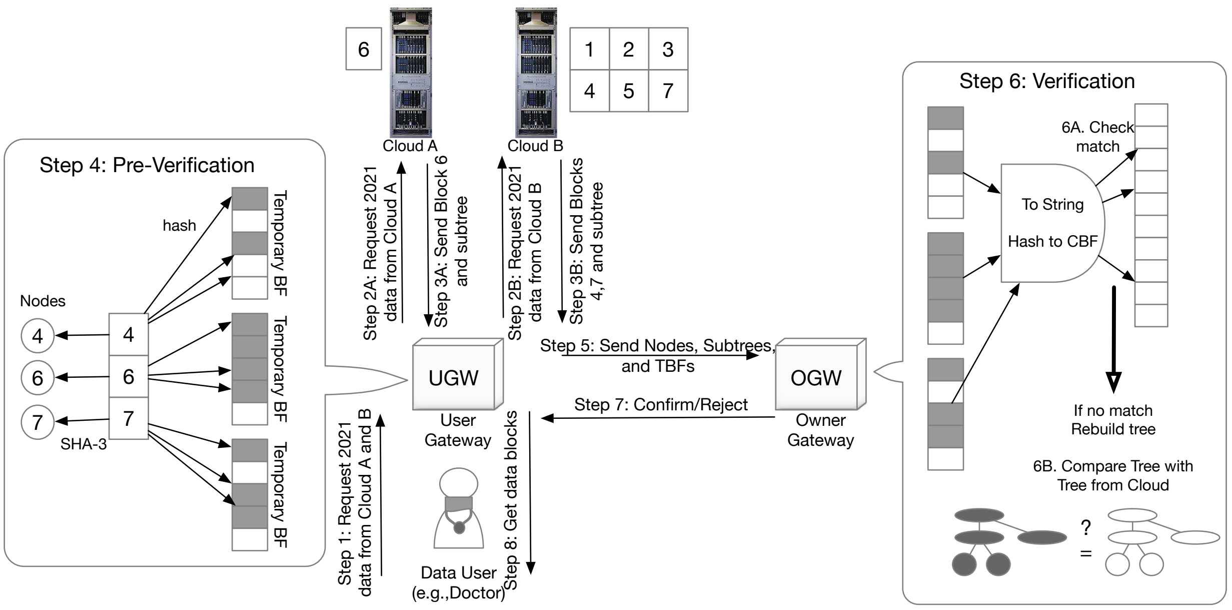


Figure 9: Retrieving and verifying the integrity of data

other factors.

Another refinement that can be done to reduce the communication complexity during the verification process is to avoid sending the middle nodes of the subtree that can be rebuilt using the received SHA-3 values; therefore, in such a case, it is sufficient to send SHA-3 values and some high-level common parents. As an example in Figure 5(b) nodes "$2021 - B$" and "B" can be rebuilt using current information (SHA-3 for blocks 4 and 7 and HMAC values for 2020-B) and sending them as a part of the subtree will not add to the security of the integrity verification. Furthermore, in the verification phase step $6 - B$, it is not required for those nodes (2021-B and B) to be compared to the rebuilt tree.

Yet another possibility that reduces the communication overhead is to send the SHA-3 values and the lowest common parent(s). As an example, in Figure 8 to retrieve and verify the integrity of node 4, only the SHA-3 value of nodes 4 and 7 and the $HMAC$ value of the node $2021 - B$ are needed [12].

3.6 Tuning Bloom Filters

In this section, we discuss how to tune the size of Bloom Filters to reduce the overhead and still have tolerable false positive and false negative rates.

3.6.1 False-Positive Probability for Simple Temporary Bloom Filter

As mentioned in Section 2, the size of the Bloom Filter is m and the number of data blocks inserted into the Bloom Filter is n; however, only one data block ($n = 1$) is inserted into the Temporary Bloom Filter. For such a simple Bloom Filter, the false positive probability (that a block that does not exist is believed to be correct) is calculated as $FP = (1 - e^{(\frac{-k \times n}{m})})^k$ [35]. Based on this formula, our analysis shows that for $m = 20$ the false positive probability would be less than 0.01 when k is between 2 and 50. As an example, the FP for $m \in (0, 20)$ and $k = 3$ is shown in Figure 10. For $k = 3$ and $m = 20$ we have $FP = 0.0027$.

[12] As another example, in Figure 8, to retrieve and verify the integrity of nodes 5 and 7, UGW sends SHA-3 values for 1,2,3,5,4,7 and HMAC for B to OGW. In a more complicated example to retrieve and verify the integrity of nodes 7 and 8, UGW sends SHA-3 for 4,7,8,9, and to create node Peter we also need the HMAC of node A. There is no need to send node B since it can be created using SHA-3 values. However, OGW still needs 2020-B (there is no need to transmit node 2021-B since it can be created using SHA-3 values), the same situation is true for node C that needs node 2021-C for verifying data integrity.

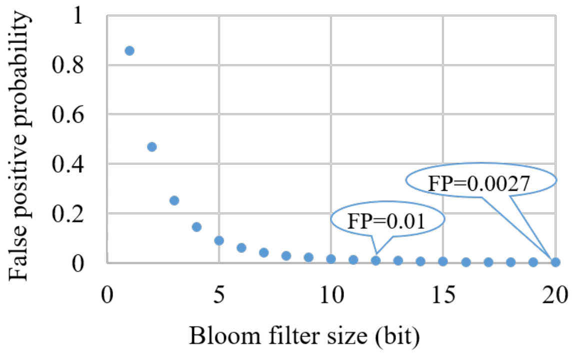


Figure 10: False positive probability for temporary simple Bloom Filter: $n = 1$ & $k = 3$

3.6.2 False-Positive and Negative Probability for a General Bloom Filter - GBF

The false-negative probability for a GBF is calculated in [35] and summarized in this section. Let us assume that k_0 is the number of hash functions that turn the bit to 1 and k_1 is the number of hash functions that change the bit to 0. The false-negative rate for a GBF is calculated as shown in the set of equations below (Equation Set 1) with comments:

Equation set 1: False Negative probability in GBF [35].

$q_0 = (1 - e^{(-k_0/m)})$

$\triangleright$ q_0 determines the probability of a bit reset

$q_1 = (1 - e^{(-k_1/m)}) \times e^{(-k_0/m)}$

$\triangleright$ q_1 determines the probability of a bit set

$b_0 = m \times q_0$

$\triangleright$ b_0 is the probability of resetting m bits

$b_1 = m \times q_1$

$\triangleright$ b_1 is the probability of setting m bits

let's define Q as$(1 - q_0 - q_1)$

$$p_{00}(n-i) = [Q^i + \frac{q_0}{q_0 + q_1} \times (1 - Q^i)]^*$$

$$p_{11}(n-i) = [Q^i + \frac{q_1}{q_0 + q_1} \times (1 - Q^i)]$$

$$F_n = \frac{1}{n} \sum_{i=0}^{n-1} [(1 - p_{00}(n-i)^{b_0}) \times (p_{11}(n-i)^{b_1})]$$

$\triangleright$ F_n is false negative probability

* $p_{00}(n - i)$ is the probability of a bit that is reset in the $n - i$-th insertion remains 0 after the ith insertion. $p_{11}(n - i)$ is the same for a bit that is set.

Similarly, the calculation of the false positive for GBF is calculated in Equation set 2 below with comments:

Equation set 2: False positive probability in GBF [35].

$$p = p_0(1 - q_0 - q_1)^n + \frac{q_0}{q_0 + q_1} \times (1 - (1 - q_0 - q_1)^n)$$

$\triangleright$ p is probability of a bit remaining zero after n insertions

$$F_p = (p^{b_0}) \times ((1 - p)^{b_1})$$

$\triangleright$ F_p is the false positive probability

Based on Equation sets 1 and 2, and assuming that each owner generates one data block (each data block is 16 KB as suggested in [19]) each day and also assuming that in most cases users may require the data from the past year, in order to keep false positive as low as 1%, we calculate the minimum required size of the Bloom Filter for $n = 365$. With $m = 10700$ the false positive probability is less than 0.01 for $k \geq 3$[13] (where $k = k_1 + k_2$); therefore, $m \geq 10700$ bits satisfies $FP \leq 0.01$. Figure 11 and 12 show the FN and FP rates for $k \in \{3, 5, 7, 10\}$ as example.[14] As shown in Table 1, in the case the owner is storing more data blocks each day, if we maintain $m = 10700$ bits, in order to keep FP less than 0.01, k should increase. As an example, if the owner (e.g., a patient) is creating 10 data blocks each day of the year (3650 data blocks), in order to satisfy $FP \leq 0.01$, k should be 7. $k = 7$ also satisfies $FP \leq 0.01$ for 100 data blocks a day (36500 data blocks each year). The reason that we do not select a higher value of k is the FN. The false-negative probability increases with increasing k with a small number of blocks (check $k = 7$ FP and $k = 7$ FN in both Figures 11 and 12). Although FN is addresses by the tree (if the data cannot be verified by the Bloom Filter it will be checked against the tree as explained in Section 3.4), it is more efficient to optimize k based on the application requirements as this reduces the time needed for integrity verification.

4 Analytical and Experimental Results and Comparisons

We first analyze HDI's security and complexity in time, for storage and communication. Then we simulate the HDI process and analyze the results and compare HDI with DPDP [19, 21]. Finally, we implement HDI in Java using two Raspberry Pis as gateways (OGW and UGW) and a Desktop computer as a cloud server to see how HDI may perform in a real-world experiment.

[13] The upper bound is an 8 digit number.

[14] In HDI, it is more important to keep the false positives low, since false negatives can be addressed using the rebuilt tree.

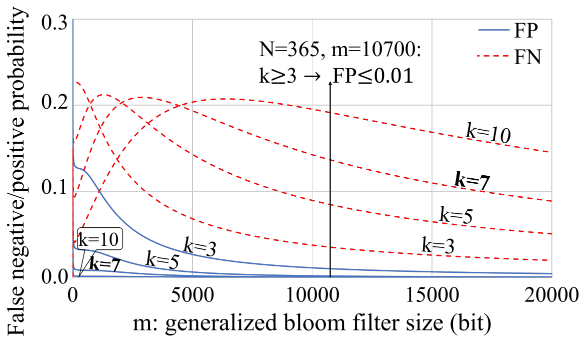


Figure 11: False-positive and negative probability for GBF for $n = 365$

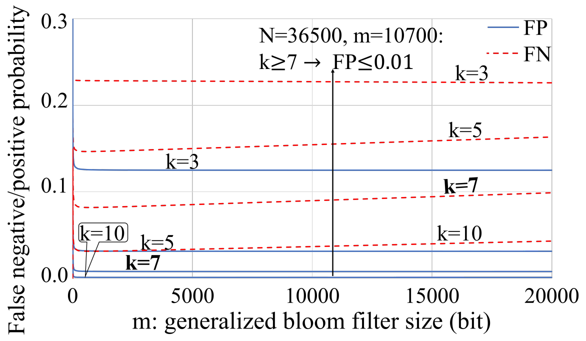


Figure 12: False-positive and negative probability for GBF for $n = 36500$

4.1 Analysis

We discuss security and evaluate time, storage, and communication complexity in this section.

4.1.1 Security

We discuss trust, hash functions linearity property, and collision to analyze the security in this section.

- **Trust:** Trustworthiness of entities is defined in the threat model in Section 3.1. The most important security assumption is that the owner's gateway - OGW - should be fully trustworthy. No key is shared with the UGW; therefore, the UGW will not be able to change the data without such changes being detected. One may argue that the cloud may keep the old correct SHA-3 values and then tamper with the data. The UGW cooperates with it to send those pre-calculated SHA-3 values to OGW instead of calculating SHA-3 values for the data it received currently; however, if the UGW is malicious in this way, it would be much easier for it to not even communicate with the OGW and use unverified data. On the other hand, this scenario does not make sense since the UGW starts the process and asks

Table 1: False-positive and negative probability for the different number of data blocks each day for CBF size of 10700 bit. 365 blocks (one block per day),3560 (10 blocks per day), and 36500 (100 blocks per day)

	$k = 3$		$k = 5$		$k = 7$		$k = 10$	
Number of data blocks	FP	FN	FP	FN	FP	FN	FP	FN
365	0.0100	0.0351	0.0014	0.0843	0.00026	0.13627	0.000029	0.1917
3560	0.1016	0.1784	0.02876	0.2106	0.0076	0.1682	0.00097	0.0972
36500	0.1250	0.227	0.03127	0.155	0.0078	0.0904	0.00098	0.037

OGW to verify the data for it. Therefore, we also assume that the UGW should be trustworthy enough to verify the data with the OGW's help. After all, the UGW needs the correct data so that the service provider can offer the best service to the owner.

- **Linearity:** One of the advantages of HDI is that the hash functions **do not have to be linear or homomorphic** as in most (if not all) provable data possession (PDP) and proof of retrievability (POR) provided solutions (PDP and POR are discussed in Section 5). Here we chose SHA-3 and HMAC but other secure hash functions can be used as well (e.g., SHA-512).

- **Collision:** As mentioned in Section 3, in HDI, data is first hashed into a simple temporary Bloom Filter (TBF). Then the TBF will be transformed into a string, and the string is hashed into a concatenated Bloom Filter. The probability of detecting a forged element as a valid one is the same as the probability of false-positive, which in HDI would be equal to $FP_t = P_1 + (1 - P_1) \times (P_2)$ where P_1 is the probability of having a collision in the TBF and P_2 is the probability of having a collision in the CBF. As discussed in Section 3.6.1, the false positive for TBF is calculated as $FP = (1 - e^{(\frac{-k \times n}{m})})^k$. For $n = 1$, $k = 3$ and $m = 20$ bits, we have the FP or P_1 of 0.0027 (Figure 10). As discussed in Section 3.6.2 in Equation sets 1 and 2, in CBF, for $n = 365$ (one year data, assuming that each day 1 data block is created.), $k = 7$ and $m = 10700$ bits the FP or P_2 would be 0.0002 therefore, the overall false positive rate would be $FP_t = P_1 + (1 - P_1) \times (P_2) = 0.0027 + (1 - 0.0027) \times 0.0002 = 0.0029$. This value will change with time – for example, after two years the data is doubled and this value would be $FP_t = P_1 + (1 - P_1) \times (P_2) = 0.0027 + (1 - 0.0027) \times 0.0014 = 0.004$ when the number of blocks that are hashed into the Bloom Filters are doubled. Therefore, it is suggested to use the tree more frequently for older data[15]. In the tree, the collision happens based on the collision probability in SHA-3, which is negligible ([48]).

4.1.2 Storage

The cloud services store the data and the tree. The number of children each node of the tree can have is between 2 and d $(2 \leq d \leq \log(n))$. The number of nodes in the tree can be calculated using a geometric progression sum as $S = \frac{(1-d^h)}{1-d}$, in which h is the height of the tree (considering the value of d, h is between $\log_d(n)$ and $\log_2(n)$). Assuming we have n data blocks, the number of nodes in the worst case, when each node has only 2 children, is $(n - 1)$ (note that the leaves are omitted). Thus, the cloud storage of the tree is $O(n)$. The OGW stores the CBF with a fixed size (m) which has a storage complexity of $O(1)$. As shown in Section 3.6.2, with $n = 36500$, to get a FP of 0.01 or less (with $k = 7$), m should be 10700 $bits$. In summary, the client storage has the complexity of $O(1)$ (HMAC key and CBF) and cloud storage has the complexity of $O(n)$ (tree) in addition to the data blocks.

4.1.3 Time

Storing the data includes the process of inserting the data into the CBF and creating the tree. For storing n data blocks, the time complexity of inserting data nodes into a CBF using k hash functions is $O(kn)$. To create the tree, all blocks need hashing (i.e., a calculation of SHA-3 or HMACs values) which is fast to compute [48, 34]. For a tree with $2n - 1$ nodes,[16] this process has the

[15] We tuned out variables so carefully (e.g., $k = 7$ and m=10700) that with n increasing, the FP would not exceed 0.01. As an example, for $n = 36500$, the FP is $0.0027 + (1 - 0.0027) * (0.0078) = 0.0104$ and from this point, increasing n does not affect FP. Choosing efficient k and m, based on the application requirement, can make a huge difference.

[16] Building the tree involves n number of $SHA - 3$ computations and at most $n - 1$ number of $HMAC$ computations. Although the $SHA - 3$ values are omitted from the tree, they affect the time complexity of computation.

time complexity of $O(n)$. Therefore, the overall time complexity for storing n data blocks including filling the CBF and creating the tree is $O(kn)$.

The integrity check process includes Bloom Filter comparisons and sub-tree rebuilding. A comparison between BFs and CBF is done in $O(1)$, and then there is $\log(n) - 2$ number of HMAC calculations to rebuild the sub-tree; therefore, the overall verification process time complexity is $O(\log(n))$

To update the tree (change, insert, delete), the SHA-3 or HMAC for the siblings is retrieved and the nodes along the path to the root should be updated $(\log(n)-2)$. The CBF does not have to be updated since it tends to forget old data. The overall time complexity for updating process is $O(\log(n))$.

4.1.4 Communication

In the process of retrieving the data and verifying the integrity of data, two places have communication overhead. The communication between the cloud server and UGW (fetching the data) and communication between the UGW and OGW (for verifying the integrity of the retrieved data blocks).

In fetching the data, the cloud server sends the data $(O(n))$ and the sub-tree to the UGW $O(\log(n))$. The sub-tree is the certain overhead in HDI, which has the complexity of $O(\log(n))$ (n is the size of the retrieved data). Based on the nature of the user's query, the data may need to include not only the requested data but also sibling blocks of the requested data. In the scenarios, we considered, this situation happened rarely in experiments, where the user requested data from a random time period and random device and mostly included all of the children of one parent node. In rare cases that sibling data blocks are required, in a worst-case scenario, the number of overhead sibling data blocks would be $d - 1$ data blocks (each block is 16 KB as suggested in [19] and $2 \leq d \leq \log(n)$ as mentioned in Section 3.5.); therefore, the complexity of communication overhead between the cloud and the UGW including sibling data blocks $(O(\log(n)))$ and subtree $(O(\log(n)))$ still would be $O(\log(n))$.

In verifying the data, the UGW sends the temporary BF $(O(1))$, sub-tree $(O(\log(n)))$ and SHA-3 of blocks ($O(1)$ per block – we have at most d blocks due to the limited number of children in the tree $(2 \leq d \leq \log(n))$; therefore, the complexity is $O(\log(n))$ to the OGW. Hence, the entire communication overhead complexity between OGW and UGW is $O(\log(n))$.

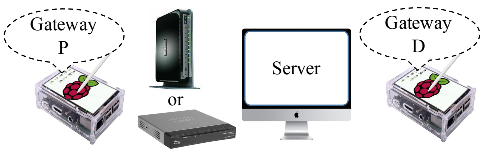


Figure 13: Elements in the implementation

4.2 Experimental Results

An implementation in hardware can help us to make a better sense of the performance of the HDI method in the real world. As shown in Figure 13, we implemented a server (to emulate the cloud) on a desktop computer using a Mac operating system (with 2.66 GHz Quad-Core Intel Xeon processor and 8 GB 1066 MHz memory), and Java version 8. We also implemented the OGW and UGW on two Raspberry Pis version 3 Model B, with a Raspbian operating system, running Java 8. Our justification is that a Raspberry Pi can emulate an IoT gateway (inexpensive but perhaps running a full OS).

The elements (server, OGW, and UGW) communicate with each other using Java sockets[17]. We tested this implementation on different networks including a wireless network (WiFi with a data rate of 28 Mbps using NET GEAR N750 wireless dual band gigabit router), wired Ethernet network (using Cisco gigabit smart switch SG 200-08), and finally, in order to eliminate the effect of network connection speed and processing power of Raspberry Pis, we also performed experiments on a single system (PC) for comparison.

We used crowd-sourced fit-bit data sets from [26]. The devices in our experiment include HRM (heart rate monitor), activity tracker, and calorimeter. We assumed that each device stored its data in a corresponding server (on the desktop). We assumed 36500 data blocks (each block 2^{14} bytes as suggested in [19]) as information from a single year (each day generates 100 blocks of data) to select the tuned Bloom Filter size (m value) for tolerable FP and FN rates. As discussed in Section 3.6.2, to satisfy an FP rate of at most 0.01, in CBF the value of $m = 10700$ bits is enough (considering that k should be chosen based on the number of data), as shown in Figures 11 and 12. In the temporary BF, as discussed in Section 3.6.1 and Figure 10, the size as small as $m = 20$ bits gives us the overall FP of less than 0.01.

In order to evaluate the performance, we measured the execution time of updating and verification processes. We repeated each experiment 100 times and calculated the average with a 95% confidence interval as shown in Figure 14 and Figure 15.

The experiment outcome (time) for the updating

[17] The implementation is available on Github and can be reached at https://github.com/Maryam-mary-karimi/HDI

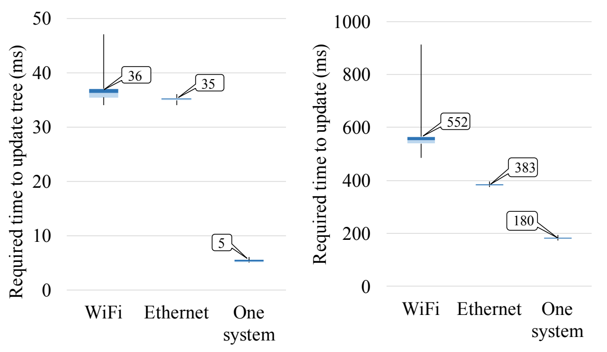


Figure 14: Updating time: (Left) The required time for requesting the tree from the server and updating it. (Right) The entire updating process including updating CBF, updating the tree, sending the data and the tree to the server

process is shown in Figure 14. The required time for updating the tree and the entire updating process was measured. Updating the tree involves requesting the related part of the tree from the server, creating new leaves (SHA-3 values) for new blocks of data, and rebuilding the tree. This process took 36 ms using the wireless network, 35 ms in a wired network, and 5 ms in a single system, on average. The entire updating process including receiving data, updating CBF, updating the tree, and sending the new data and the tree to the server took 552 ms using WiFi connection, 383 ms in wired connection, and 180 ms in a single system, on average.

The verification process is shown in Figure 9. In this experiment, we measured the verification time (steps 4,5,6,7), starting from the time that the UGW receives the data and metadata from Servers. The UGW then starts negotiating with the OGW and the OGW verifies/rejects the data and replies to the UGW. We also measured the required processing time for the OGW to verify the data both with CBF or the tree.

The averages in the experiments indicate that the number of nodes in the tree is 377.7 and the average size of the sub-tree that should be sent to the UGW contains 50.19 nodes. On average, in 75% of the cases in each experiment, the CBF could verify the data without requiring the usage of the tree. In all cases, the integrity verification time using only the CBF took less than 0.5 ms.

As shown in Table 2 and Figure 15, considering a real-world implementation (using WiFi as the communication network protocol and Raspberry Pis as gateways), verifying the integrity of data blocks using CBF took only 0.37 ms on average in the OGW, and in this situation, the entire verification process took only 207 ms (considering the PC experiment which eliminates the

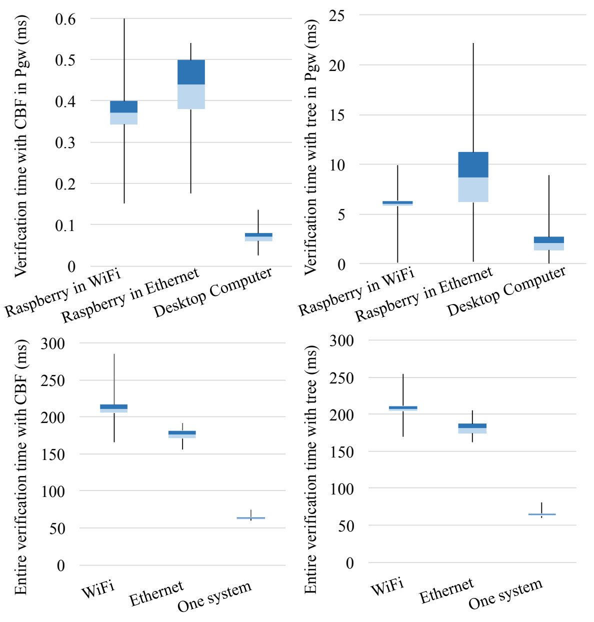


Figure 15: Verification time: the top-left chart shows the required time to verify the data in the owner's gateway, using only CBF (step6A in verification), the top-right chart shows the required time to verify the data in the owner's gateway, using the hash tree (step6B in verification), the bottom charts show the entire verification time

network delays this time would be 63 ms). In the cases that the CBF failed to verify the data, it takes 6.1 ms for the OGW to verify the data using the tree. However in most cases (more than 75%), in which the CBF is enough for verifying the integrity of data, the verification time in the OGW would be much less than 1 ms.

4.3 Comparison with HDI and DPDP

We also compared our method, Hierarchical Data Integrity (HDI), with Dynamic Provable Data Possession (DPDP) [19, 21], both analytically and experimentally with $1GB$ file of 2^{16} blocks of data. We compared the storage in the cloud, storage on the client-side (client and

Table 2: Experimental evaluation for verification time

	Using CBF		Using tree	
Process	in OGW	all	in OGW	all
Step	6A	4,5,6A,7	6B	4,5,6,7
WiFi	0.37 ms	207 ms	6.1 ms	208 ms
Eth	0.44 ms	176 ms	8.7 ms	181 ms
PC	0.06 ms	63 ms	1.5 ms	63 ms

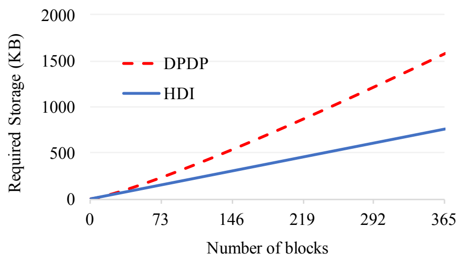


Figure 16: Cloud storage in HDI and DPDP

gateway), and communication overhead.

In DPDP [20, 22, 19, 21], clients store data in a rank-based authentication Skiplist (a data structure that keeps the meta-data of n blocks as leaves and ranks upper layers as the number of accessible leaves) in an untrustworthy server. The nodes in the search path are affected in the case of the insertion, modification, or deletion of blocks. The client stores the root of the list. Please note that as discussed in Section 5, DPDP was not developed for the scenario we have considered, where a trusted third-party data user needs integrity checks by a data owner, not in the cloud. Further, DPDP does not consider multiple data sources and multiple cloud stores. Yet, it is a reasonably close integrity verification approach.

As shown in Figure 16, DPDP stores its Skiplist in the cloud with the complexity of $O(n^{\epsilon}\log(n))$ (ϵ is expected amortized and is between 0 and 1 [19, 21].) while HDI stores the tree with the complexity of $O(n)$. Storing 2^{16} blocks in DPDP consumes $1.0084GB$ while in HDI it takes at most $1.0035GB$. HDI requires less storage than DPDP in the cloud.

Storage in the client (OGW in HDI) in both methods has the complexity of $O(1)$. DPDP stores the root node and HDI stores the CBF. Although they both have the same complexity of $O(1)$, experiments show that for 2^{16} blocks DPDP needs 18.13 KB, while, HDI requires 4.231 KB (as discussed in Section 3.6.2, the required size of the CBF is 10700 $bits$), to make sure that the integrity of one whole year worth of data is verifiable with a false positive probability of less than 0.01, without requiring to use the tree for verification. HDI requires less storage than DPDP in client storage as depicted in Figure 17.

Considering that DPDP was implemented in C++ and HDI is implemented in Java and they used different platforms, comparing the experimental execution time may not be valid; however, the analytical comparison can give us a good estimation. The integrity verification time in both methods (DPDP and HDI with tree) has the complexity of $O(log(n))$; however, experiments showed that in HDI, in 75% of the cases, the tree is not required

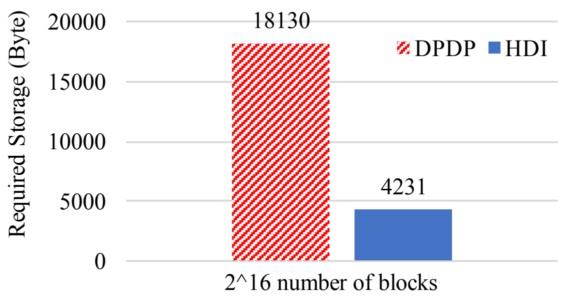


Figure 17: Client-side storage in HDI and DPDP

for data verification and the CBF alone can verify the integrity of data. In this case (using Bloom Filters), the verification has the time complexity of $O(1)$ and has a better performance than DPDP. Of course, we cannot compare the security implications completely in the case of using Bloom Filters. HDI and DPDP both have the time complexity of $O(log(n))$ for updating the data.

We compared communication overhead as well. Communication complexity in DPDP includes sending the ”proof” (the path from the root to leaf) and the key which has the complexity of $O(\log(n))$ which would be 415.5 KB overhead in experiments [19]. As mentioned in Section 4.1, in HDI the communication complexity is divided into two parts: the communication complexity between cloud and UGW and the communication complexity between UGW and OGW.

In our experiments, the user requests data blocks from a random device and random time period. In assessing the communication complexity between the cloud and UGW, based on such experiments, the query includes all leaf-level children of one parent (recall that leaves are SHA values that are omitted and only pointers remain to determine which data blocks belong to this lowest-level parent.); therefore, the communication overhead between the cloud and UGW includes only the subtree (with the size complexity of $O(\log n)$ which was 480 bytes on average in the experiments– n is the number of nodes in the tree and each node size is 32 bytes); however, imagine the case where the user queried data that does not include all children of the lowest parent. In this case, the cloud server should also send the requested sibling blocks to the UGW. In the worst-case scenario, the number of sibling data blocks is $(d-1)$ and each block is 16 KB. In this case, considering the communication between cloud and UGW, with $d \leq 28$, HDI is still better than DPDP. Consider also that DPDP is assumed to be working with a single source of data stored in a single cloud; hence the assumption is to have one skip list for all data. However, if we apply DPDP

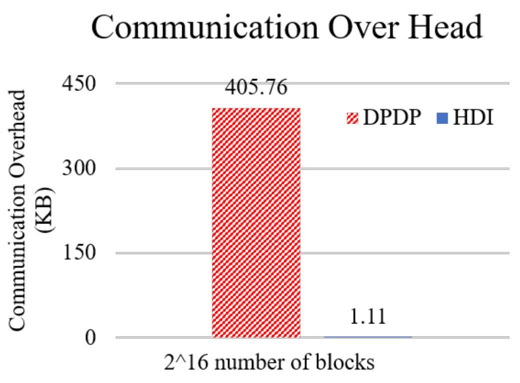


Figure 18: Communication Overhead in HDI and DPDP

to the multiple clouds scenario we may need multiple skip lists which may also increase the communication overhead by a factor of c which is the number of cloud servers.

The communication complexity between UGW and OGW includes temporary BFs $(O(n(1))$, sub-tree $(O(\log(n)))$ and SHA-3 values $(O(1))$ which is $O(\log(n))$ in overall. Therefore HDI and DPDP have the same communication complexity; however, as shown in Figure 18, experiments showed that for 2^{16} blocks, DPDP has 415.5 KB overhead while HDI has only 1.11 KB, overhead. The number of siblings does not change this communication overhead considerably, since each extra sibling will add only 32 bytes (size of SHA-3 output value) to the communication overhead.).

Analytical and experimental results showed that HDI performs as well or somewhat better than DPDP in time, storage, and communication overhead. Analytically, HDI has the same or less time complexity than DPDP in verifying the data. In addition, experiments showed that HDI surpasses and was better than DPDP in requiring less storage and communication overhead.

5 Related Work

This section discusses methods from the literature that provide data integrity verification, provable data possession (PDP), and proof of retrievability (POR). In experiments explained in Section 4.3, we compare HDI with DPDP [19] in terms of time complexity, memory complexity, and communication cost. The procedures used with HDI and their complexity are explained in detail in Sections 3 and 4. Here, we provide a summary that helps to compare HDI with other previous methods. Then we review some previous methods in integrity verification, PDP, and POR and compare them with HDI.

5.1 Summary of HDI

We assume that there is an owner's gateway that stores data in different clouds and later the trusted third party's gateway retrieves the data and requests the owner's gateway to verify the data integrity. In HDI, we verify the integrity of the data using a hierarchical structure of nested Bloom Filters (simple Bloom Filters inside a generalized Bloom Filter) and a tree (that keeps HMAC values which are derived from SHA-3 values). Hashing the data in the Bloom Filter and creating the tree has the time complexity of $O(1)$ and $O(n)$ respectively. Updating the tree has the complexity of $O(\log\ n)$. Then, we store a Bloom Filter and one key in the client's gateway with a storage complexity of $O(1)$. The key remains in the owner's gateway and there is no requirement for key sharing. The tree is stored in the cloud with the storage complexity of $O(n)$ (up to $n-1$ number of hash values). The communication between the User's gateway and owner's gateway, during verification, includes SHA, sub-tree, and simple Bloom Filter which has the complexity of $O(\log\ n)$.

The verification process, in most cases, uses the Bloom Filter which has the complexity of $O(1)$. If data blocks are not verified with the BF, in order to avoid false negatives, the data integrity will be verified using the tree which has the time complexity of $O(\log\ n)$. In the following, we highlight the differences between previous approaches (mostly provable data possession and proof of data retrievability) and HDI.

5.2 Integrity Verification in Literature

Providing data integrity, privacy, and trust in IoT networks has attracted much attention. In order to balance user privacy and integrity in cloud servers, and computational cost, authors in [61] added some user-arbitrary weights while calculating the mean and used users' identity and biometric elliptic curve cryptography to authenticate them [61]. Authors in [25] and [1] modeled trust and reputation. Authors in [38] used a data integrity monitored method to detect and isolate failures in a sensor system.

In HDI we discuss how to assure that the data being stored in the cloud, is the same data that is received from sensors and IoT devices. It is out of the scope of this work to check the integrity of the data that sensors and IoT devices created at the time of creation; however, we briefly discuss it here. Ontology Evaluation (OE) is used to assure that IoT devices' measurements and information were correct and the data are coherent. OE can be used to address the heterogeneity and diversity of data created by devices in IoT networks. It addresses the integrity verification problem semantically. It can be used to assure that the data coming from the sensors and IoT devices were correct in the first place. Ontology can be based on rule, evolution, metric, application, data-driven, evaluation by humans, gold standard (compare with a high standard reference), and task. OE can be done with different aspects such as syntax, structure, vocabulary, semantics, representation, and context. It is important to build the ontology correctly (verification) in both lexical and structural features and build the correct ontology (validation) [44].

The authors in [29] used a cryptographic one-way hash to detect up to d defective items in a set of n items. They proposed a digital watermarking technique, to encode authentication information in a data structure *D*. They did this by modifying non-data fields, in a way that they should not be immediately identifiable by an adversary. In their model, the adversary modified the values of *D* but not the topology of *D*'s pointers. The adversary had the knowledge of the algorithm but not the cryptographic master key. The authors built a program that identified up to *'d'* number of changes and made it probabilistically difficult for an adversary to reproduce the database structure. The idea in [29] came from blood testing in which a test consists of selecting a sample including *'t'* items and performing a single experiment that determined if the sample contained bad blood. They produced a $t \times n$ matrix, where for any $d + 1$ columns there was one designated, with non-adaptive combined group testing scheme performed on each row. The column with a negative test result had 1 in a row and was removed. The remaining columns corresponded to bad elements.

In the method described in [29], the only quantity that is needed to remain at the client-side was a key. In our method (HDI), a Bloom Filter and a hash key, both of fixed size, are needed to be stored in the gateway; both methods have the same client-side storage complexity of $O(1)$. In [29], the authors add some watermarks to the data in the cloud that has the complexity of $O(d^3 \log\ n\ \log\ d)$ where *'d'* determines how many defective items can be identified and n is the number of bits. In HDI, considering each item as a block of data, the extra memory required in the cloud has the complexity of $O(n)$. Depending on d, HDI can either perform better in storage or better in detecting more modifications to data blocks. The work in [29] did not consider updating the data, or retrieval by a third party.

Multi-party fair exchange and blockchains have been used for exchanging items that are efficiently and verifiably encrypted [33]. These methods are not practical to be used for large files and are more useful towards keeping track of transactions, signing agreements, etc. In FairSwap [18], authors used optimistic blockchain-based fair exchange -in which a trusted third party verifies the data only in cases where two parties cannot reach an agreement and for signing contracts between two parties. In recent work in [3], authors used coin-based fair exchange for exchanging large files between multiple parties. While it was unnecessary for all parties to receive the file in previous schemes, in coin-based fair exchange all parties receive the file at the same time (requires synchronization) and if anybody does not, the sender that did not send its share needs to compensate the rest; broadcasting the message here has a complexity of $O(n^2)$. Further, in [3], the key belongs to the trusted third party, the prover encrypts the message and the receiver verifies the ciphertext message. The process has 3 phases: (1) Parties generate a public key and obtain their share of the file (2) They encrypt their share of the message with a symmetric key and forward it to anyone requires it. (3) They generate a decryption and a proof for every party that requires it.

The protocol in [3] includes exchanging of many messages and the communication cost has a complexity of $O(n^2)$ for each participant (blockchains typically have large communication costs). HDI has a complexity of $O(1)$ for the number of communications and each communication has only the size complexity of $O(log(n))$. Further, HDI does not involve any key sharing versus [3] that includes both symmetric keys sharing between the parties and an additional asymmetric key with a trusted third party [3]. HDI does not require an additional trusted third party.

5.3 Provable Data Possession in Literature

In this section, we review the literature in PDP. The work in [4] provides provable data possession at untrusted stores, using fully additive homomorphic signatures; however, since it is not secure, they added one-time indices to make it secure. In homomorphic message authentication, the user generates a set of tags that authenticates some values using a secure key. This method is able to dynamically add blocks without re-tagging the entire file, support unlimited verification, and with some variations, it also supports public verifiability; however, even the publicly verifiable modified version of this work requires sharing the key with other entities which we avoid in HDI. Other work such as [6, 27, 15, 5, 52] all tried to provide more efficient PDPs. In [28] a Diffie-Hellman Key Exchange and Merkle hash tree is used which has a storage overhead as large as the file itself to reduce the server computation; however, this method increased communication. The works in [27, 16] are RSA-based PDP approaches that have a communication complexity and client storage complexity of $O(1)$; but, they have heavy computation on the server, and performing RSA over a file is slow. In

[52], authors provide a method in which the signature of the parity blocks is equal to the parity of the signature of the data blocks and use this method to provide a PDP approach with a communication complexity of $O(n)$. In [53, 64], authors used Diffie-Hellman-based approaches; however, in both approaches, the client has to store n bits per data block; therefore, these methods would not be efficient if data blocks are small. All these methods involve public-key cryptography and they all require key sharing which we avoid in HDI. Some PDP schemes are based on identity to eliminate the requirement for public key certificates [49]. They eliminated PKI (public key infrastructure); therefore when the third party verifier wants to very the data for data owner it should receive the key from PKG (public key generator) by using the identity of the data owners. HDI eliminates the process of key sharing completely. Authors in [30] designed a PDP scheme that performs multiple updates at the same time by using a Merkle hash tree that enables updating the values of multiple leaves and their parents up to the root which also requires key sharing [30].

Authors in [7] discussed an amortized verifiable computation in which the client provided a function and an input to the server. The server replies with the answer and proof of the correctness of the result. The evaluation of polynomials was derived from very large data sets. Initially, the client stores the data on the server with some authentication information and keeps a short secret key. The server computes a result with an authentication code. The client keeps the clear text polynomial P and a vector of coefficients. The server has a vector of groups of the form $g_i^{ac} r_i$ in which r_i is the i^{th} coefficient of polynomial R and was calculated using a pseudorandom function. When queried, the server replies with $y = P(x)$ and $t = g^{aP(x)+R(x)}$ and the client accepts y if $t = g^{ay+R(x)}$. One application of [7] is in verifying outsourced computation to make predictions based on polynomials fitted to many sample points in an experiment. Another application is updating data and performing verifiable keyword searches and securing proofs of retrievability. For an n variable for polynomial of degree d, assuming Decisional Diffie-Hellman the required time for the setup is $O((n+d)^d)$. After the setup, the required time is $O(nd)$ in the client and is $O((n+d)^d)$ in the server; while in HDI, the set up time is $O(n)$ and it avoids key sharing while [7] requires key sharing.

In the work in Catalano-Fiore [14] a value m is encoded into 1-degree polynomial y, where $y(0) = m$ and $y(\alpha) = Fk(L)$, where F randomizes a label. The server creates a new MAC with n authentication polynomials $(y1, y2, ...yn)$ that authenticate m as result of $f(m1, ...mn)$ and also $y(\alpha) = f(Fk(L1), Fk(L2), ..., Fk(Ln))$. Before this work, existing verification algorithms were not time efficient. The server sends m' to the client that could test whether m' is the result of $f(m1, m2, ..., mn)$ by checking if $y(0) = m'$ and $y(\alpha) = f(Fk(L1), Fk(L2), ..., Fk(Ln))$. The authors avoid the time consuming $y(\alpha) = f(Fk(L1), Fk(L2), ..., Fk(Ln))$ part by safely reusing labels. They constructed a pseudo-random scheme that pseudo-computes a piece of the label. They split the labels into 2-dimensions: the data set identifier and input identifier represented as (Δ, τ) that allows the same τ in labels. Also with a pseudo-random function F using new amortized closed-form efficiency, if a user pre-computes some information wf with same τs and different Δs it is possible to use wf to compute $W = f(Fk(\Delta, \tau), Fk(\Delta, \tau), ..., Fk(\Delta, \tau))$ in constant time. The client either should store the labels, which is a 2-dimensional matrix, and/or compute them, which has time and communication complexity of at least $O(n^2)$ and it is not efficient compared to HDI which has the complexity of $O(n)$. In addition, this scheme requires public key sharing.

In Dynamic Provable Data Possession (DPDP) [20, 22, 19, 21], clients store data and a Skiplist in an untrustworthy server. The skip list is a data structure that keeps the meta-data of n blocks as leaves and ranks upper layers as the number of accessible leaves. This method is based on rank-based authenticated skip lists. The nodes in the search path are affected in the case of insertion, modification, or deletion of blocks. The client keeps the label of the top-leftmost element in the skiplist which is the root of the list. The validation of data integrity consists of the hash of nodes in the verification path (leaf to root) with the size of $O(\log(n))$. For updating the data, the client verifies the new proof and computes the new label of the root node after the update. The updating process affects nodes along the verification path with the length of $O(\log(n))$. As shown in Section 4.3, HDI outperforms DPDP in storage: $O(n^{\epsilon}\log(n))$ in DPDP comparing to $O(n)$ in HDI, time: both $O(\log(n))$ if HDI uses tree; however, HDI is $O(1)$ if the Bloom Filter is used and communication costs: both have the same complexity of $O(\log(n))$: however, experiments show that HDI's communication cost for a 1G file verification is $1.11KB$ while in DPDP it is $450KB$. The DPDP scheme is based on homomorphic verifiable tags combined with authenticated data structures, such as Merkle trees, skip lists, hash tables, etc. to store them. There are other variations of DPDP that may serve specific requirements such as blockless DPDP, RSA-tree-based DPDP, and FLexlist DPDP. Authors also developed a Generalized DPDP that separates the process of creating tags and building the authenticated implicitly-ordered data structure. Therefore, after proving the security of data structure, their framework can be used to create the basic DPDP scheme[23].

5.4 Proof of Retrievability in Literature

In [31, 54, 17, 11] proof of retrievability (POR) is provided which applies to encrypted files only and supports limited numbers of challenges. High Availability and Integrity Layer (HAIL) [10] provided proof of retrievability (POR) for a trusted verifier that checks the integrity of data and *corrects the errors* where the file is distributed across multiple servers with redundancy across servers and the threat model has a Byzantine adversary that can corrupt multiple servers at a time. In HAIL the client stores only the key. It assures granularity of a full file by detecting server faults in a challenge-response reactive cryptographic system and recovers corrupted files using cross-server redundancy. The file is publicly verifiable even if it is encrypted. They build an IP-ECC (Integrity protected error-correcting code) which combines MAC and parity and aggregate responses by combining MACs across multiple blocks. In HAIL the client is the one that verifies the data and it does not support public verifiability. In HAIL the server code overhead is 9% of the file size for a 1G data while in HDI as explained in Section 4.3 the storage overhead in cloud servers is only 0.35%. In addition in HDI, the user is the one that receives the data and the gateway verifies the data without sharing any key with any party; however, HAIL requires sharing the public key with the cloud but provides error correction and redundancy.

6 Discussion

In this paper, we develop a hierarchical data integrity (HDI) verification method for connected health devices, that can be used with other IoT services, when we have multiple sources and data is being stored in multiple clouds. In this method, there is a gateway that exists on the data owner's side (OGW) that stores the data in multiple could servers. We verify the integrity of the data using a hierarchical structure of nested Bloom Filters (simple Bloom Filters inside a generalized Bloom Filter) and an encrypted hash tree (that stores keyed-HMAC values which are derived from SHA-3 values). We store a Bloom Filter and one key in the owner's gateway and there is no requirement for key sharing. The hash tree is stored in the cloud.

A gateway on the user (third-party) side (UGW) retrieves the data and requests the OGW to verify the data integrity. The communication between the UGW and OGW during verification, includes SHA values, the sub-tree, and a simple Bloom Filter. The verification process, in most cases, uses the Bloom Filter; however, if the data are not verified to be unchanged, to avoid false negatives arising with the Bloom Filter, the data integrity will be verified using the encrypted hash tree. Results show that this scheme reduces the complexity of time, storage, and communication required for verifying the integrity of data without requiring explicit key sharing.

In this scheme, we have two security assumptions. First, OGW is trustworthy and secure. Second, UGW is trustworthy and secure. We clarify these assumptions in the following:

1. OGW has to be trustworthy. This is the strict minimum security requirement for this scheme. Every security scheme depends on the secret key, and in HDI the secret encryption key for the tree is stored in OGW. All the processes are managed so that OGW never shares this key with any entity. The secret key is used in the hash tree (hash values are keyed-HMAC values) and this cannot be compromised without having the key (assuming secure encryption and hashing schemes). We can assume that OGW gets partially compromised, e.g., the CBF gets manipulated; in this case, we can still use the encrypted tree for verification. However, if we assume that OGW is totally compromised then the secret key is revealed and the tree can be compromised. If we cannot trust the OGW with either the CBF or the key, then there is no way to verify the integrity of data. Therefore it is a strict necessity for the scheme to make sure that OGW is secure and can store the key securely.

2. If UGW gets compromised, the problems are different. A service provider does not require communication with the OGW and can hurt the patient (or data owner) directly; the compromised UGW does not ask for integrity verification of the data since it intends to compromise the data. Therefore, it does not make sense to assume that a UGW is compromised and would still use this scheme.

To assure the security of OGW and UGW authors in [41] describe a platform based on the combination of software defined network (SDN) [43] and software defined perimeter (SDP) [9, 46]. The combination of SDN and SDP assures the safety and security of OGW and UGW through a global network view and the flexibility of SDN and security mechanisms of SDP. This software defined platform has a single controller that authenticates and checks the security parameters of each entity before allowing it to communicate with other entities in the network. Each authenticated entity should send a request for communicating with another entity, the controller authorizes the connection, and creates secure communications between them [41].

7 Conclusions

In this paper, we develop a solution that can store the owner's data, gathered from multiple sources, in multiple clouds and verify the data integrity, when the service provider retrieves it. We assumed a scenario where the data owners (e.g., patients) own their data (e.g., medical data) and replace expensive monitoring devices with wearable devices. Since the cloud services are provided by different vendors and may not be reliable, trustworthy, or secure, the data owner should be able to verify the integrity of their data with local information but not have significant storage, communication, or computational overhead. Therefore, we suggest and create a hierarchical model that uses Bloom Filters and a Hash Tree to verify the integrity of data.

The methodology is hierarchical – this feature helps us to trade-off between the trustworthiness of retrieved data and speed. Each data block is hashed into an empty fixed-size Bloom Filter. This Bloom Filter is hashed into a concatenated generalized Bloom Filter (CBF). The two-level Bloom Filter reduces the cost of data integrity validation. To resolve the false negatives in the CBF, a hash tree for blocks of data is created, encrypted, and stored in the cloud. The hash tree is trustworthy but it is slower; however, the Bloom Filter is speedy, but it introduces false negatives and false positives and may not be completely secure. If the application's security is more important, we can set a flag to occasionally check the tree, as an extra verification step, even in cases that the Bloom Filter was enough to verify the integrity of data. One can imagine a challenge-response-like protocol in such situations.

As far as we know, this is the first method that specifies the ability to verify the integrity of data from multiple sources, stored in multiple clouds, without requiring explicit key distribution. The key remains only in the data owner's gateway. This method has the advantage of having the capability to verify the integrity of data that is coming from multiple sources and is stored in multiple servers, and as experiments and analytical comparisons show, HDI still surpasses or is comparable to the methods that work with a single source and single server. Results showed that HDI is efficient in time, storage, and communication overhead.

This method may apply to applications such as healthcare as well as storing data from smart homes in cloud servers, smart city data, smart cars, smart farming, etc. and the parties that need access to those data can be banks (who wants to give loans), insurance companies, city departments, or other service provider companies. Gateways will have to mediate the verification of data integrity.

We plan to examine and analyze security attacks in detail in the future and investigate how they may interfere with HDI and how we should make HDI secure against those attacks. For example, we have not carefully considered a curious cloud provider or a malicious adversary. We plan to compare this method with blockchain integrity check approaches. We also plan to improve this approach to be able to perform error correction by improving either the Bloom Filter structure or improving hash functions used in the tree. In addition, one possibility for future work is to integrate an ontology layer to detect errors and check integrity of the data at the source. Ontology has been widely used to examine devices' measurements, information correctness and the data coherency and we plan to add another integrity checking layer this way.

Acknowledgments

The authors would like to thank Prof. Vladimir Zadorozhny for valuable relevant discussions. We would like to express our deep gratitude to Alptekin Kupcu for providing the DPDP implementation in C++.

Implementation Code

The HDI implementation is available online. It includes three Java projects for the three elements including server, DGW, and PGW. They communicate through sockets. It also includes Matlab code for calculating false positive and negative rates. The code is on GitHub and can be reached at https://github.com/Maryam-mary-karimi/HDI

References

[1] N. B. Akhuseyinoglu, M. Karimi, M. Abdelhakim, and P. Krishnamurthy, "On automated trust computation in iot with multiple attributes and subjective logic," in *2020 IEEE 45th Conference on Local Computer Networks (LCN)*. IEEE, 2020, pp. 267–278.

[2] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007, https://www.sciencedirect.com/science/article/pii/S0020019006003127.

[3] H. K. Alper and A. Küpçü, "Coin-based multi-party fair exchange," in *International Conference*

on Applied Cryptography and Network Security. Springer, 2021, pp. 130–160.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 598–609.

[5] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, 2008, pp. 1–10.

[6] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 863–874, https://dl.acm.org/citation.cfm?id=2516681.

[7] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable delegation of computation over large datasets," in *Annual Cryptology Conference*. Springer, 2011, pp. 111–131, https://link.springer.com/content/pdf/10.1007/978-3-642-22792-9_7.pdf.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 313–314, https://link.springer.com/content/pdf/10.1007/978-3-642-38348-9_19.pdf.

[9] B. Bilger, A. Boehme, B. Folres, Z. Guterman, M. Hoover, M. Iorga, J. Islam, M. Kolenko, J. Koilpilla, G. Lengyel *et al.*, "Sdp specification 1.0," 2014.

[10] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 187–198.

[11] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 43–54.

[12] K. Bratbergsengen, "Hashing methods and relational algebra operations," in *Proceedings of the 10th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1984, pp. 323–333, https://dl.acm.org/citation.cfm?id=671296.

[13] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004, https://projecteuclid.org/download/pdf_1/euclid.im/1109191032.

[14] D. Catalano and D. Fiore, "Practical homomorphic macs for arithmetic circuits," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 336–352, https://link.springer.com/content/pdf/10.1007/978-3-642-38348-9_21.pdf.

[15] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *2008 the 28th international conference on distributed computing systems*. IEEE, 2008, pp. 411–420.

[16] Y. Deswarte, J.-J. Quisquater, and A. Saïdane, "Remote integrity checking," in *Working Conference on Integrity and Internal Control in Information Systems*. Springer, 2003, pp. 1–11.

[17] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Theory of Cryptography Conference*. Springer, 2009, pp. 109–127.

[18] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 967–984.

[19] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 4, p. 15, 2015, https://user.eng.umd.edu/~cpap/published/cce-alp-cpap-rt-09.pdf.

[20] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 213–222. [Online]. Available: https://doi.org/10.1145/1653662.1653688

[21] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and Ö. Özkasap, "Flexdpdp: Flexlist-based optimized dynamic provable data possession," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, pp. 1–44, 2016, https://crypto.ku.edu.tr/wp-content/uploads/2019/05/flexdpdp.pdf.

[22] E. Esiner, A. Küpçü, and Ö. Özkasap, "Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession,"

in *International Conference on Intelligent Cloud Computing*. Springer, 2014, pp. 65–83.

[23] M. Etemad and A. Küpçü, “Generic dynamic data outsourcing framework for integrity verification,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–32, 2020.

[24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000, http://www.cs.utexas.edu/users/lam/396m/papers/SummaryCache.pdf.

[25] G. Fortino, L. Fotia, F. Messina, D. Rosaci, and G. M. Sarné, “Trust and reputation in the internet of things: state-of-the-art and research challenges,” *IEEE Access*, vol. 8, pp. 60 117–60 125, 2020.

[26] R. Furberg, J. Brinton, M. Keating, and A. Ortiz, “Crowd-sourced fitbit datasets 03.12.2016-05.12.2016,” May 2016, https://doi.org/10.5281/zenodo.53894. [Online]. Available: https://doi.org/10.5281/zenodo.53894

[27] D. L. Gazzoni Filho and P. S. L. M. Barreto, “Demonstrating data possession and uncheatable data transfer.” *IACR Cryptology ePrint Archive*, vol. 2006, p. 150, 2006.

[28] P. Golle, S. Jarecki, and I. Mironov, “Cryptographic primitives enforcing communication and storage complexity,” in *International Conference on Financial Cryptography*. Springer, 2002, pp. 120–135.

[29] M. T. Goodrich, M. J. Atallah, and R. Tamassia, “Indexing information for data forensics,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 206–221, https://link.springer.com/content/pdf/10.1007/11496137_15.pdf.

[30] J. He, Z. Zhang, M. Li, L. Zhu, and J. Hu, “Provable data integrity of cloud storage service with enhanced security in the internet of things,” *IEEE Access*, vol. 7, pp. 6226–6239, 2018.

[31] A. Juels and B. S. Kaliski Jr, “Pors: Proofs of retrievability for large files,” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 584–597.

[32] G. Ç. Kayaturan and A. Vernitski, “Routing in hexagonal computer networks: How to present paths by bloom filters without false positives,” in *Computer Science and Electronic Engineering (CEEC), 2016 8th*. IEEE, 2016, pp. 95–100, https://ieeexplore.ieee.org/abstract/document/7835895.

[33] H. Kılınç and A. Küpçü, “Optimally efficient multi-party fair exchange and fair secure multi-party computation,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2015, pp. 330–349.

[34] H. Krawczyk, M. Bellare, and R. Canetti, “Hmac: Keyed-hashing for message authentication,” 1997.

[35] R. P. Laufer, P. B. Velloso, and O. C. M. Duarte, “A generalized bloom filter to secure distributed network applications,” *Computer Networks*, vol. 55, no. 8, pp. 1804–1819, 2011, https://www.gta.ufrj.br/ftp/gta/TechReports/LVD11.pdf.

[36] H. Lim, J. Lee, and C. Yim, “Complement bloom filter for identifying true positiveness of a bloom filter,” *IEEE Communications Letters*, vol. 19, no. 11, pp. 1905–1908, 2015, https://ieeexplore.ieee.org/abstract/document/7264999.

[37] H. Lim, N. Lee, J. Lee, and C. Yim, “Reducing false positives of a bloom filter using cross-checking bloom filters,” *Appl. Math*, vol. 8, no. 4, pp. 1865–1877, 2014, https://pdfs.semanticscholar.org/aa1f/2b857e93dc5d76c010fc091c6e83e7631625.pdf.

[38] G.-X. Liu, L.-F. Shi, and D.-J. Xin, “Data integrity monitoring method of digital sensors for internet-of-things applications,” *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4575–4584, 2020.

[39] A. Loten, “CIOs contend with ever-expanding range of cloud services,” *The Wall Street Journal*, December 1st 2017. [Online]. Available: https://blogs.wsj.com/cio/2017/12/01/cios-must-manage-ever-expanding-range-of-cloud-services/

[40] U. Manber and S. Wu, “An algorithm for approximate membership checking with application to password security,” *Information Processing Letters*, vol. 50, no. 4, pp. 191–197, 1994, http://webglimpse.net/pubs/password.pdf.

[41] P. K. Maryam Karimi, “Software defined ambit of data integrity for the internet of things,” in *The 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. IEEE/ACM, 2021.

[42] M. McIlroy, “Development of a spelling list,” *IEEE Transactions on Communications*, vol. 30, no. 1, pp. 91–99, 1982, www.acooke.org/spell.pdf.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[44] S. Mishra and S. Jain, “Ontologies as a semantic model in iot,” *International Journal of Computers*

and Applications, vol. 42, no. 3, pp. 233–243, 2020.

[45] M. D. D. Moreira, R. P. Laufer, P. B. Velloso, and O. C. M. Duarte, “Capacity and robustness tradeoffs in bloom filters for distributed applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2219–2230, 2012, https://ieeexplore.ieee.org/abstract/document/6171165.

[46] A. Moubayed, A. Refaey, and A. Shami, “Software-defined perimeter (sdp): State of the art secure solution for modern networks,” *IEEE Network*, vol. 33, no. 5, pp. 226–233, 2019.

[47] J. K. Mullin, “A second look at bloom filters,” *Communications of the ACM*, vol. 26, no. 8, pp. 570–571, 1983, https://dl.acm.org/citation.cfm?id=358167.

[48] C. Paar and J. Pelzl, “Sha-3 and the hash function keccak,” *Understanding Cryptography A Textbook for Students and Practitioners, www. crypto-textbook. com*, 2010, http://professor.unisinos.br/linds/teoinfo/Keccak.pdf.

[49] S. Peng, F. Zhou, Q. Wang, Z. Xu, and J. Xu, “Identity-based public multi-replica provable data possession,” *IEEE Access*, vol. 5, pp. 26 990–27 001, 2017.

[50] A. Rousskov and D. Wessels, “Cache digests,” *Computer Networks and ISDN Systems*, vol. 30, no. 22, pp. 2155–2168, 1998, https://www.sciencedirect.com/science/article/pii/S0169755298002517.

[51] A. Salvi, S. Ercoli, M. Bertini, and A. Del Bimbo, “Bloom filters and compact hash codes for efficient and distributed image retrieval,” *arXiv preprint arXiv:1605.00957*, 2016, https://arxiv.org/pdf/1605.00957.pdf.

[52] T. S. Schwarz and E. L. Miller, “Store, forget, and check: Using algebraic signatures to check remotely administered storage,” in *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*. IEEE, 2006, pp. 12–12.

[53] F. Sebe, A. Martinez-Balleste, Y. Deswarte, J. Domingo-Ferrer, and J. Quisquater, “Time-bounded remote file integrity checking,” *Technical Report 04429*, 2004.

[54] H. Shacham and B. Waters, “Compact proofs of retrievability,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2008, pp. 90–107.

[55] E. H. Spafford, “Opus: Preventing weak password choices,” *Computers & Security*, vol. 11, no. 3, pp. 273–278, 1992, https://www.sciencedirect.com/science/article/pii/016740489290207B.

[56] R. Tamassia, “Authenticated data structures,” in *European Symposium on Algorithms*. Springer, 2003, pp. 2–5, https://link.springer.com/chapter/10.1007/978-3-540-39658-1_2.

[57] J. Tapolcai, J. Bíró, P. Babarczi, A. Gulyás, Z. Heszberger, and D. Trossen, “Optimal false-positive-free bloom filter design for scalable multicast forwarding,” *IEEE/ACM Transactions on Networking*, vol. 23, no. 6, pp. 1832–1845, 2015, http://real.mtak.hu/22032/1/06877748.pdf.

[58] E. Topol, “The smart-medicine solution to the health-care crisis,” *The Wall Street Journal*, July 7 2017. [Online]. Available: https://www.wsj.com/articles/the-smart-medicine-solution-to-the-health-care-crisis-1499443449

[59] P. Valduriez and G. Gardarin, “Join and semijoin algorithms for a multiprocessor database machine,” *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 133–161, 1984, https://dl.acm.org/citation.cfm?id=318590.

[60] M. P. Wallen, S. R. Gomersall, S. E. Keating, U. Wisløff, and J. S. Coombes, “Accuracy of heart rate watches: Implications for weight management,” *PloS one*, vol. 11, no. 5, p. e0154420, 2016, https://www.ncbi.nlm.nih.gov/pubmed/27232714.

[61] T. Wang, M. Z. A. Bhuiyan, G. Wang, L. Qi, J. Wu, and T. Hayajneh, “Preserving balance between privacy and data integrity in edge-assisted internet of things,” *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 2679–2689, 2019.

[62] S. Xiong, F. Wang, and Q. Cao, “A bloom filter based scalable data integrity check tool for large-scale dataset,” in *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. IEEE Press, 2016, pp. 55–60, http://www.pdsw.org/pdsw-discs16/papers/p55-xiong.pdf.

[63] S. Xiong, Y. Yao, S. Li, Q. Cao, T. He, H. Qi, L. Tolbert, and Y. Liu, “kbf: Towards approximate and bloom filter based key-value storage for cloud computing systems,” *IEEE Transactions on Cloud Computing*, 2014, https://ieeexplore.ieee.org/document/6995996.

[64] G. Yamamoto, S. Oda, and K. Aoki, “Fast integrity for large data,” in *Proc. ECRYPT Workshop Software Performance Enhancement for Encryption and Decryption*, 2007, pp. 21–32.

AUTHOR BIOGRAPHIES

Maryam Karimi is a Ph.D. candidate at the University of Pittsburgh in the Department of Informatics and Networked Systems, School of Computing and Information. She is working on the Internet of Things, wireless networks, security, cryptography, and machine learning. Her thesis is on security in the internet of things for which she achieved second place in the ACM graduate research competition at Grace Hopper conference 2019. She passed two internships in Medtronic working on product security in the software department. She got her M.Sc. and B.Sc. in Information Technology Engineering from the Shiraz University of Technology, during which she was participating in RoboCup international competitions and she achieved first place in the international competition of RoboCup Iran Open 2014. Her publications are mostly in the fields of security, data integrity in the internet of things, wireless software defined networks, and machine learning.

Prashant Krishnamurthy is a professor in the Department of Informatics and Networked Systems, School of Computing and Information. He teaches at both the graduate and undergraduate levels, offering introductory and advanced courses on wireless networks and cryptography. He also was one of the co-founders of the school's Laboratory for Education and Research in Security Assured Information Systems (LERSAIS), a national Center of Academic Excellence (CAE) in Information Assurance Education (IAC) and Research (IAR). During his time at Pitt, Krishnamurthy has developed new courses and programs of study for the school, particularly addressing the wireless and security curricula, (for which he served as either the principal investigator or co-principal investigator) from the National Science Foundation and the Commonwealth of Pennsylvania.