Graph Filter: Enabling Efficient Topology Calibration

Lailong Luo, Deke Guo, Jia Xu, Xueshan Luo

Abstract—The topology of a network may change inevitably, due to dynamic behaviors of nodes and links, and failures of hardware and software. Many protocols and applications must be aware of the up-to-date topology of the underlying network. This triggers the topology calibration problem, which means to deduce those different nodes and links between two topologies. The Bloom filter and its variants are efficient to represent and calibrate two general sets. They, however, fail to represent all links and nodes in a topology simultaneously, and thus remain inapplicable to the topology calibration problem. In this paper, we design the graph filter, a novel space-efficient data structure to record not only the node set but also the link set of any given topology. Accordingly, given two topologies we aim to represent them via two respective graph filters, and thereafter deduce those different links in an invertible manner. To this end, we design three essential operations for graph filter, i.e., encoding, subtracting and decoding. Although such operations are sufficient to solve the topology calibration problem, two challenging issues still remain open. First, the XOR traps which occur with low probability at the encoding stage may result in a few miscalculations at the decoding stage. Thus, we propose another augmented decoding algorithm to lessen the impact of XOR traps via terminating illegal decodings. Second, several different links may form cycles in the worst case; hence, we further design a cycle destruction algorithm to make such different links decodable. We implement the graph filter and the associated topology calibration method. Comprehensive evaluations indicate that our method finishes the topology calibration task efficiently with high probability, incurs the least space overhead, and supports invertible decoding reasonably.

Index Terms—topology calibration, graph filter, false positive, false negative.

1 INTRODUCTION

The mission of topology calibration is to deduce those different nodes and links between a given pair of topologies \( G_A \) and \( G_B \). In reality, topology calibration is a common and fundamental task in a variety of networked systems and distributed applications [1]. At the stage of configuration, miswirings or human errors may result in mismatches between the designed blueprint and the actually established network topology [2]. For example, novel topologies are proposed to enhance the network performance of data centers, e.g., DCell [3], BCube [4]. However, the miswired links will ruin the symmetry or hierarchy of these proposals.

Besides, many essential applications or protocols (e.g., multicast, incast) must be aware of the exact topology of the underlying network. The source routing scheme [5] deduces a routing solution for each unicast, multicast, or incast transfer, according to the underlying network topology. Therefore, once the network topology changes, the decision-maker has to track and update its view about the topology. The out-of-date topology may impose a negative impact on routing performance. In the context of Software-Defined Networking (SDN) [8], each of the distributed controllers generates a local view of the network, and then maintains a global and consistent network topology. In this manner, the controllers implement the best flow table entry for any new flow timely. Moreover, beyond the community of computer networks, graphs are employed as a high-level abstraction in the field of graph data mining [9], knowledge mapping domain [10], social networks [11], et al. In these fields, deducing the differences between two graphs is essential to uncover the evolution pattern of graphs, identify their topological similarity, et al. Generally, topologies are represented as adjacency matrices or adjacency tables, which are usually not space-efficient. Let \(|V|\) and \(|E|\) denote the number of nodes and links in a topology. The adjacency matrix, which is a \( |V| \times |V| \) matrix of binary bits, uses 0 and 1 to indicate the existence and absence of the link between any pair of nodes. Therefore, an adjacency matrix costs \( O(|V|^2) \) bits. Adjacency table, on the contrary, maintains a list for each node to record its corresponding neighbours in the topology. The pointers in the lists occupy space resources aggressively. Besides, the value of \(|E|\) goes wild in a highly connected large-scale network, which further aggravates the space overhead of an adjacency table.

Moreover, the general set calibration methods based on Bloom filter and its variants [12] are not applicable to the topology calibration problem. Given two topologies, such methods can separately represent and calibrate nodes and links in theory. However, their incapabilities of representing nodes and links together leads to the following shortcomings. First, they suffer from high false negative or (and) false positive errors due to their potential hash collisions. To ensure a given calibration accuracy, more space resources
must be reserved. Second, they often rely on the membership query to derive out different elements between two sets, which implies a high time-complexity at the level of $O(|V| + |E|)$. In extreme cases, all the nodes and links have to be queried to uncover only one different node or link. This is definitely inefficient. Third, calibrating nodes and links separately may lead to inconsistency between nodes and links. For example, even when the different nodes are identified successfully, the links among them may not be searched out correctly. This mismatch implies the failure of topology calibration.

For the above reasons, we design graph filter, a novel data structure to tackle the topology calibration problem with the following two design rationales: 1) efficiently represent the topological information of a given topology, including nodes and links; 2) inversely decode different links between two topologies without prior knowledge. The previous data structures and methods, however, fail to realize the rationales simultaneously. Typically, a graph filter consists of $n$ cells, each of which records the neighbours of a single node in the given topology. Such a one-to-one mapping between cells and nodes is realized by maintaining a labeling mechanism. A cell records the links of the associated node via two fields, i.e., $idSum$ and $Count$. Specifically, for the $i^{\text{th}}$ cell, its $idSum$ integrates all neighbours of the $i^{\text{th}}$ node by XORing their $ids$, while the $Count$ field counts the number of these neighbours.

The philosophy of using XOR in graph filter is that XOR enables the aggregation, subtraction and deletion of nodes mapped into a cell, in a both computation and storage-friendly manner. For any node $e$, its neighbouring nodes are aggregated into the $idSum$ field. Therefore, the space overhead of a graph filter is constant with the number of nodes, irrespective of the number of links. Besides, XORing the $idSum$ fields of $e$ in two topologies eliminates the common neighbours in once execution. Moreover, by XORing the $id$ of node $e$ with its neighbours’ $idSum$, node $e$ will be deleted from the topology easily. Notably, these bit-wise calculations of XOR can be very fast in hardware.

Furthermore, three dedicated operations (i.e., encoding, subtracting and decoding) are proposed for the graph filter, so as to deduce the different links between two topologies. In the process of encoding, all links in each topology are aggregated into a graph filter. Then, the subtracting operation wipes off the common links in the two topologies, while remains the different ones. Lastly, with the support of anchor cells (whose $idSum\neq 0$ and $Count=1$), we decode the direct links from the subtracting result recursively with high probability.

Two decoding algorithms are proposed to deduce the underlying different links, i.e., the original decoding algorithm and the augmented decoding algorithm. The original decoding algorithm searches anchor cells and deletes the links associated with such cells recursively. However, we note that the potential XOR traps (different pairs of nodes derive the same XOR result when encoding) may lead to a few false positive and false negative errors when decoding. Hence, the augmented decoding algorithm is designed to lessen the impact of XOR trap by illegalizing some anchor cells. Furthermore, in extreme cases, some different links form cycles and no anchor cells will be exposed. Consequently, we further design a cycle destruction algorithm to break up the cycles and decode these links. The core idea is to select and delete a link from the cycle with a suppose-and-check strategy, such that more anchor cells will be exposed.

The comprehensive experiments indicate that graph filter is efficient to calibrate topologies, especially for topologies with a small quantity of different links. Graph filter is more space-saving than other data structures and its time-consumption is acceptable. The augmented decoding operation can significantly decrease the false negative rate, at the cost of a bit higher false positive rate. Moreover, the cycle destruction algorithm can indeed break up the potential cycles and decode the underlying links. We summarize the major contributions of this paper as follows:

- We formulate the topology calibration problem, and propose a novel data structure named graph filter to represent the topological information of any topology. Moreover, the dedicated encoding, subtracting and decoding operations are designed to derive the different links between any pair of topologies.
- We further redesign the decoding operation by terminating potential illegal decodings and attaching a cycle destruction algorithm to it. These enhancements lessen the impact of XOR traps and enable the usage of graph filter in more general scenarios.
- We conduct comprehensive experiments to evaluate the performance of graph filter in the cases of general ER random graphs and two typical data center topologies. The results indicate that graph filter can tackle the topology calibration problem efficiently.

The remainder of this paper is organized as follows. Section 2 summarizes the related work and Section 3 introduces the design methodology of graph filter. After that, based on graph filter, Section 4 designs the associated operations to derive the different links between two topologies. Thereafter, Section 5 reports the probability analysis of the XOR trap, and Section 6 presents the comprehensive evaluations of our proposal. Finally, Section 7 discusses several related issues and Section 8 concludes this paper.

## 2 RELATED WORK

In this section, we summarize the related work about the topology calibration problem in computer networks, including the topology design methodology, the general data structures to represent topologies, and the existing methods to identify difference between topologies.

### 2.1 Topology designs in computer networks

Terminals and network devices are interconnected with a dedicated topology in a computer network. General topologies, i.e., Bus, Star, Ring, Tree, and Mesh, are designed to configure a network with cables. Beyond the wired networks, wireless communication techniques are developed to enable Ad Hoc networks, cellular networks, and wireless sensor networks. The above wireless networks also rely on typical topologies, e.g., Star, Mesh.

Recently, novel topologies are proposed to configure large-scale data centers. In fact, a distinguished topology is of great significance to fault-tolerance, bisection bandwidth,
network diameter, scalability, flexibility, and maintainability of a data center. According to the employed communication techniques, the existing proposals can be categorized as wired topologies and wireless topologies. For wired topologies, devices are connected with general cables or fibers via carefully designed topologies, e.g., Trees [13], Torus, Generalized Hypercubes, Hierarchical structures [3] [4], Random graphs [14] [15], etc. By contrast, wireless topologies introduce various emerging wireless communication techniques (Laser communication, 60 GHz RF communication [16], and Free-Space-Optical communication [17]) into data centers to speed up the links and increase the networking flexibility.

Although state-of-the-art topology designs for networks are proved to function well by either simulations or real deployments, these endeavours can get in vain due to miswirings or failures. The designers hold by default that their blueprints can always be deployed correctly, but this is not always true. Therefore, identifying the difference between the blueprint and real-deployed topology is important to guarantee the design goals of the network topologies.

2.2 Data structures to represent topologies

To represent a topology, generally, adjacency matrix or adjacency table is formed. Adjacency matrix is the most widely-used data structure since it records the node and link information simultaneously. However, its memory cost is \( O(|V|^2) \) bits, where \(|V|\) is the number of nodes in the topology. By contrast, an adjacency table associates each node with the collection of its neighbouring nodes in the graph. An adjacency table consists of \(|V|\) lists, and each list is responsible to record the neighbours of a corresponding node in the topology. For a 32-bit system, the memory cost of an adjacency table is \( 2 \times 32 \times |E| \approx 64|E| \) bits, where \(|E|\) denotes the number of links in the topology. Hence, for highly connected topologies, an adjacency table may cost more memory than the adjacency matrix.

Besides, probabilistic data structures, e.g., Bloom filter [12] and its variants [18] [19] [20] [21], can be employed to record the nodes directly and support fast membership queries with an acceptable false positive rate. However, these probabilistic data structures fail to record the topological information of the topology efficiently, since mapping the massive number of links into Bloom filter calls for vast memory. In effect, to minimize the potential false positives, the length of a Bloom filter is determined as \( kn \cdot \ln 2 \), where \( k \) and \( n \) denote the number of utilized hash functions and the number of recorded elements, respectively. For a connected topology, the number of links \(|E|\approx|V|-1, |V|\times(|V|-1)/2\), which can be extremely large.

The general data structure, like adjacency matrix and adjacency table, can record the topological information precisely, but the resultant space cost is non-trivial for large-scale topologies. The major reason is that the number of links is much more than that of nodes in a highly connected network. Especially, in some oversubscribed network designs, there are even \( O(|V|^2) \) links to guarantee connectivity and fault-tolerance. Consequently, we envision a novel probabilistic data structure, which represents the topological information of a given topology with the memory cost proportional to \(|V|\) instead of \(|E|\).

2.3 Searching the difference between topologies

Let \( G_A(V_A, E_A) \) and \( G_B(V_B, E_B) \) denote two topologies. \( V_A \) and \( V_B \) are the set of nodes (network devices), while \( E_A \) and \( E_B \) are the set of links among the network devices. The task of topology calibration is to extract the different links and nodes between \( G_A \) and \( G_B \) in a timely fashion. Let \( \Delta_V \) and \( \Delta_E \) denote the different nodes and diverse links, respectively. Conceptually, \( \Delta_V \) include the nodes that only appear in \( G_A \), as well as the nodes that only occur in \( G_B \). Similarly, \( \Delta_E \) consists of the links that only exist in topology \( G_A \) and the links which are only contained in topology \( G_B \).

As a typical computer network, data center network has been extensively studied. Especially, many current data center architectures embed locality information into IP address for easy-routing purpose, e.g., BCube [4], DCell [3], PortLand [25]. In these cases, traditional address configuration techniques will be inefficient due to their huge amount of manual deployments. To this end, DAC [2] and ETAC [26] are proposed with the ambition of detecting devices with miswired links. DAC and ETAC formulate the address configuration as a graph isomorphism problem, which is a well-known NP problem. Then efficient algorithms are designed to speed up the calculation. Furthermore, by rational utilization of the correctly wired links, a miswiring tolerant routing protocol named MTR [27] is introduced to increase the network throughput in the presence of miswired links. DAC and ETAC, on the contrary, work only at the stage of implementation, but fail to search out the changed links or nodes in the life-cycle of a data center. That is, they cannot trace the topology evolution of a data center. To this end, in this paper, we employ a graph filter to report the changed links whenever necessary.

For more general situations, one may hold that the widely studied subgraph matching algorithms can be employed to compare the two topologies. However, these algorithms are computation-intensive since they are designed for subgraph matching, which is a proven NP-hard problem [22]. Fortunately, each node in a given network is associated with a unique address (i.e., either the IP address or MAC address). Based on the exclusive addresses, those complicated comparisons in the subgraph matching algorithms are not necessary. Besides, these exclusive addresses also enable the possibility of representing the given topologies with elegant data structures, e.g. Bloom Filter [12] or its variants, and Hash Tables [23]. For example, one may employ Bloom filter to record all nodes in a topology by mapping their IP addresses or MAC addresses into a bit vector. However, Bloom filters have to query the nodes or links one by one to discover the differences, which is surely time-consuming.

In conclusion, the existing methods for address configuration, subgraph matching or set calibration are not applicable to the problem of topology calibration. The inefficiency of existing methods motivates us to design a novel data structure to record the topological information of each topology, and then identify those links in \( \Delta_E \).

3 THE DESIGN PHILOSOPHY OF GRAPH FILTER

In this paper, we focus on the problem of identifying the links in \( \Delta_E \). To this end, in this section, we first propose a
space-efficient data structure, named graph filter, to represent arbitrary topologies.

3.1 Design of graph filter

A graph filter consists of \( |V| \) independent cells, and the \( i^{th} \) cell records the information of all neighbours of a node whose label is also \( i \) in the topology. More specifically, each cell has two fields, i.e., \( \text{idSum} \) and \( \text{Count} \). The \( \text{Count} \) field counts the number of the neighbouring nodes that are recorded in this cell. The \( \text{idSum} \) integrates all neighbours of the \( i^{th} \) node by Xoring their \( \text{ids} \) together. Note that the \( \text{id} \) of a node is the binary number of its assigned label. Thus, the number of bits for an \( \text{id} \) is determined as \( \lceil \log(|V|+1) \rceil \). The factor \(|V|+1\) indicates that we label the nodes from 1 instead of 0.

Fig. 1 depicts an illustrative example of representing topologies with graph filters. For the given \( G_A \), the node labels range from 1 to 5 since \( n=5 \), and thus \( \lceil \log(5+1) \rceil = 3 \) bits are required to construct the \( \text{ids} \). In the generated \( GF_A \), 5 cells are employed to record the neighbours of each node in the topology. Node 1 has two neighbours, i.e., 2 and 3, therefore \( GF_A[1].\text{Count}=2 \) and \( GF_A[1].\text{idSum}=010 \oplus 011 = 001 \). In the same way, the other cells can be derived out. For a pair of nodes with the same labels, if they share the same \( \text{idSum} \) and \( \text{Count} \) in the graph filters, they share common neighbours with high probability.

Definition 1. For any cell in the graph filter, if the XOR results of diverse combinations of the recorded neighbours are the same, the information of the involved neighbours will be eliminated from the cell. We denote this phenomenon as XOR trap.

In the framework of graph filter, whenever the XOR operations are conducted, XOR traps may occur. For instance, in \( G_A \), if nodes 1011, 1001, and 0010 are all connected with a node 1100, then \( GF_A[12].\text{idSum} \) will be calculated as 0000. Since the XOR result of 1011 and 1001 is 0010 which is exactly a neighbour of node 1100, the generated \( \text{idSum} \) will eliminate the information of 1011, 1001 and 0010. As a probabilistic data structure, graph filter may suffer from miscalculations, but as illustrated in Section 4, we can lessen its negative impact reasonably.

3.2 Recording topological information with graph filter

The existing probabilistic data structures fail to record the topological information efficiently. By contrast, general data structures, e.g., adjacency matrix, adjacency table, can record the topological information of arbitrary topology. However, their memory cost can be unacceptable when representing a large-scale topology.

Graph filter represents the topology by aggregating the information of any node’s neighbours into a cell. In effect, in an undirected topology, each link connects a pair of nodes directly. For a topology with labels from 1 to \( |V| \), the \( i^{th} \) cell in graph filter is responsible to record the neighbours of the node whose label is exactly \( i \). In this manner, any link \( <i,j> \) will be recorded in both the \( i^{th} \) cell and the \( j^{th} \) cell. Every cell in a graph filter integrates the corresponding neighbours together by executing XOR operations in the \( \text{idSum} \) field and increasing the \( \text{Count} \) field. Moreover, unlike an adjacency table, the memory cost of a graph filter is only decided by the number of nodes in the topology, irrespective of the number of links. Based on the graph filter, we tackle the topology calibration problem efficiently by following the method introduced in Section 4.

4 Identifying different links with graph filter

The most difficult challenge of topology calibration is to deduce the different links between \( G_A \) and \( G_B \). Therefore, in this section, we focus on the problem of identifying the links in \( \Delta_E \). For simplicity, we suppose that \( G_A \) and \( G_B \) have the same nodes but diverse links.

4.1 Identifying the different links

Based on the proposed graph filter, three operations are designed to deduce the links in \( \Delta_E \), i.e., encoding, subtracting and decoding. To be specific, the topologies \( G_A \) and \( G_B \) are first represented as \( GF_A \) and \( GF_B \), respectively. Afterward, the subtracting operation eliminates those common links, and hence generate \( GF_C \) that records the different links only. Lastly, the decoding operation lists all links in \( GF_C \) reversely.

Encoding. The basic principle of encoding is that the \( i^{th} \) cell records the neighbours of the node whose label is also \( i \). When a link is added, in the corresponding cell, the \( \text{Count} \) field will be increased by 1, and the \( \text{idSum} \) field executes the XOR calculation. From a high-level of view, the \( \text{idSum} \) performs as a sketch of all neighbours, and the \( \text{Count} \) identifies the number of its neighbours.

Subtracting. The mission of subtracting is to wipe off those common links and remain the diverse ones. To this end, we go through both \( GF_A \) and \( GF_B \) to derive \( GF_C \). For an arbitrary \( i^{th} \) cell, \( GF_C[i].\text{idSum} \) is calculated as \( GF_A[i].\text{idSum} \oplus GF_B[i].\text{idSum} \), and \( GF_C[i].\text{Count} \) is derived as \( GF_A[i].\text{Count} - GF_B[i].\text{Count} \).

In most cases, the resulted \( GF_C \) can represent the links in \( \Delta_E \) correctly. However, errors may occur due to XOR traps. The reason is that, the combination of diverse links may lead to the same \( \text{idSum} \). For instance, both 001 \( \oplus \) 101
Algorithm 1 Decoding operation

Require: The subtracting result $GF_C$, the number of nodes $n$.
1: Let $\Delta_D$ denote the set of decoded links;
2: Let $Flag$ be a bool variable and initialized as $false$;
3: while $Flag == false$ do
4: if There is no anchor cell in current $GF_C$ then
5: $Flag = true$;
6: else
7: Select an anchor cell from current $GF_C$;
8: Denote the location of the anchor cell as $loc$;
9: Translate $loc$ as a binary bits $id_{loc}$;
10: Interpret the $GF_C[loc].idSum$ as an integer tag;
11: Add the link $\langle loc, tag >$ into $\Delta_D$;
12: $GF_C[loc].idSum = 0$;
13: $GF_C[tag].idSum = 0$;
14: $GF_C[tag].idSum = GF_C[tag].idSum \ominus id_{loc}$;
15: if $GF_C[tag].Count > 0$ then
16: $GF_C[tag].Count = -1$;
17: else
18: $GF_C[tag].Count += 1$;
return $\Delta_D$;

Fig. 1 also depicts a walkthrough example of subtracting. The generated $GF_c$ contains three kinds of cells, i.e., type 1, type 2 and type 4. The nodes labeled as 1 in both $G_A$ and $G_B$ possess two neighbours, yet their neighbours are not totally the same. Thus, the resulted $GF_C[1].idSum$ is not 0, while the $Count$ field is 0. The positive (negative) $Count$ field implies there are Count links which only exist in topology $G_A$ ($G_B$).

Decoding. Based on the subtracting result, a more challenging task is to derive out the remained links to accomplish the topology calibration mission. To this end, we employ those cells with $idSum \neq 0$ and $Count = 1$ in the $GF_C$ as anchor cells. Let $loc$ be the location of the selected anchor cell in $GF_C$, and $tag$ denote the label of node whose $id = GF_C[loc].idSum$. Then, with high probability, the anchor cell only records one link, which connects two nodes with labels $loc$ and $tag$, respectively. Based on this observation, we list the links in $\Delta_D$ in a recursive and invertible manner. Algorithm 1 illustrates the insight of the decoding operation. The four steps are detailed as follows.

- **Step 1:** scan. The algorithm scans $GF_C$ to search out a feasible anchor cell (Line 2 to 7). If no such an anchor cell can be found, exit. Else, move to Step 2.

- **Step 2:** translate. This step deduces the link identified by the anchor cell. For this purpose, the labels of the two nodes that associated with this link should be determined. Note that, one node's label is precisely the location of the selected anchor cell. And the other node can be derived out by translating the binary $idSum$ in the anchor cell into a decimal integer, i.e., $tag$ (Line 8 to 9).

- **Step 3:** record. The above link in Step 2 is added into $\Delta_D$, which denotes the set of decoded links (Line 10 to 11).

- **Step 4:** remove. Since the link deduced above has been added into $\Delta_D$, the information of this link should be removed from $GF_C$ to expose more potential anchor cells. Let $loc$ be the location of the selected anchor cell in $GF_C$, and $tag$ denote the label that interpreted from $idSum$ of the anchor cell. The task can be accomplished via deleting node $loc$ from $GF_C[tag]$, and removing node $tag$ from $GF_C[loc]$ (Line 12 to 18). Then, back to Step 1.

Indeed, in a graph filter, the information of an arbitrary link $\langle i, j >$ is recorded in both $GF_C[i]$ and $GF_C[j]$. Therefore, once one of the two cells is chosen as an anchor cell, link $\langle i, j >$ can be decoded inversely from $GF_C$. Furthermore, deleting the decoded links, in return, will empower the emergence of more anchor cells. Fig. 2 plots an example of the decoding operation towards the $GF_C$ that are generated from the above Fig. 1. First of all, the algorithm travels the $GF_C$ and selects $GF_C[2]$ as an anchor cell. Then, in Step 2, $loc$ and $tag$ are assigned as 2 and 1, respectively. The derived link $\langle 1, 2 >$ is added into $\Delta_D$ in Step 3. Afterward, $GF_C[2]$ erases the information of node 1, and $GF_C[1]$ deletes the information of node 2. In this way, a new candidate for anchor cell occurs, i.e., $GF_C[1]$. By executing the steps for several rounds, all links in $\Delta_D$ will be decoded with high probability.

In reality, the graph filter may miss some different links due to XOR traps. Also, the decoding operation may be terminated in advance, leaving links undecoded in $GF_C$. However, we argue that this method can list the different links without prior knowledge with high probability when the number of different links $|D|$ is relatively small. In fact, the probability that vast of the links in a network fail simultaneously is negligible. Suppose that any link suffers from miswiring or failure during a time slot is independent in networks, and the associated probability is $p$. Let $X$ be a variable and denote the number of changed links. For any integer $0 \leq x \leq |E_A|$, where $|E_A|$ counts the total number of links in $G_A$, we have:

$$p(X = x) = \binom{|E_A|}{x} \times p^x \times (1-p)^{|E_A|-x}.$$

(1)
Then a XOR trap occurs, the above decoding operation may result in both false positive and false negative errors. For instance, given the neighbours of $e$ in $G_A$ as 0101, 1100 and 1011, the $GF_A[i].idSum$ is thus calculated as 0010. If node $e$ in $G_B$ has two neighbours 1001 and 1101, then a XOR trap will be triggered since the XOR result of the neighbours in $G_A$, i.e., 0101 and 1101, is derived as 1000, which is exactly a neighbour of $e$ in $G_B$. Therefore, $GF_C[i].idSum$ will be calculated as 0010$\oplus$1111=1101, and $GF_C[i].Count=3-2=1$. Notice that, in the resulted $GF_C[i]$, the information of neighbours 1001 and 0110 in $G_A$, as well as the neighbour 1001 in $G_B$, have been eliminated erroneously due to the XOR trap. In this way, the false negatives appear during decoding. Besides, the derived $GF_C[i]$ satisfies the constraints of being an anchor cell. When this cell is selected as an anchor, an additional link from $e$ to the node 1101 will be introduced into $\Delta_D$. Then, a false positive will occur.

Note that, once the encoding and subtracting operations have been executed, the potential XOR traps are embedded into the generated $GF_C$, and cannot be eliminated. Fortunately, it is still possible to weaken the negative impact of these XOR traps by augmenting the thereafter decoding operations. In the above example, if $GF_C[i]$ is chosen as an anchor cell, an extra link from $e$ to the node 1101 will be added into $\Delta_D$. Then, the decoding algorithm will “Remove” the information of node $e$ from cell $GF_C[13]$. As a result, $GF_C[13]$ may also satisfy the constraint of being an anchor cell. Thus an additional link started from the node 1101 may also be decoded as a different link. That is, a XOR trap may lead to cascade miscalculations during decoding, thus causing more false positive errors. In other words, we should cease the potential cascade miscalculations to degrade the influence of XOR trap.

Generally, in $GF_C$, most of the cells are empty, i.e., the $idSum$ field is 0 and $Count$ field is 0. Consequently, the cascade miscalculations will mis-decode an extra link from $GF_C[i]$ to an empty cell with high probability. If a candidate anchor cell, which points to an empty cell, is judged as illegal, then it is of great hope to terminate the potential cascade miscalculations. With this insight, we redesign the decoding operation by intensifying the constraint of anchor cell. Specifically, in Algorithm 1, at the stage of translate, i.e., Step 2, if $GF_C[tag].idSum=0$ and $GF_C[tag].Count=0$, then $GF_C[loc]$ is regarded as an illegal anchor cell. As a result, the later Step 3, as well as Step 4, will not be executed. In this way, this augmented decoding algorithm decreases the cascade miscalculations to some extent.

The above strategy does lessen the false positives but may lead to a bit higher risk of false negatives. By ceasing a part of the decoding operations, we also eliminate the chance to decode the hidden different links from Type 4 cells. As stated in Section 4.1, for a Type 4 cell, i.e., a cell whose $idSum=0$ and $Count=0$, the associated two nodes in $G_A$ and $G_B$ have the same number of neighbours, but the neighbours are the same or not remains uncertain due to XOR traps. For example, suppose that the node whose label is 6 has different neighbours in $G_A$ and $G_B$, w.l.o.g., $GF_A[6]$ aggregates the information of nodes 1100 and 0110 in $G_A$, while $GF_B[6]$ aggregates the information of 0100 and 1101. Then the generated $GF_C[6]$ is a typical Type 4 cell. When $GF_C[12]$ (node 12 is a neighbour of node 6 in $G_A$) is chosen as a candidate for anchor cell, according to the above strategy, the decoding process will be terminated since the cell $GF_C[tag]$ belongs to Type 4. Consequently, the different link $<i,12>$ will not be decoded any more, thereby leading to a false negative. The trade-off is detailed and quantified in the latter evaluation section, i.e., Section 6.

### 4.3 Decoding the links in cycles

Graph filter represents the topological information of a given topology by employing XOR operations. After canceling out the same links, the remained different links will be derived out inversely from the subtracting result. This methodology, however, may terminate its decoding before all cells in $GF_C$ have degenerated into Type 4 cells.

We observe that once some different links between $G_A$ and $G_B$ form a cycle or several cycles, these different links will not be decoded from the generated $GF_C$ by employing...
Algorithm 1 or its augmented edition alone. As depicted in Fig. 3, there are 5 nodes in $G_A$ and $G_B$, and $G_A$ has four more links compared with $G_B$, i.e., $<1, 2>$, $<2, 3>$, $<3, 4>$, and $<4, 1>$. The four different links create a cycle. As a result, the decoding operation, no matter the original one or the augmented one, cannot search out a legal anchor cell from $GF_C$. To tackle this dilemma, the cycle(s) should be broken up such that more candidates for anchor cells will be exposed. However, destroying the cycle directly can be really complicated since no prior knowledge is available. It is also not clear how many cycles are there. What’s worse, nodes or links may get involved with multiple cycles.

To this end, Algorithm 2 is designed to decode the underlying links when only one cycle exists. The basic insight is to assume one link among the remained nodes and then delete its information from graph filter so that additional anchor cells will be uncovered. Specifically, let $GF_{C_1}$ be the remained graph filter after decoding, and $U$ ($U \geq 3$) counts the number of undecoded cells in $GF_{C_1}$. Let $loc_i$ $(1 \leq i \leq U)$ denote the location of an undecoded cell in $GF_{C_1}$. Note that the degree of any node in the cycle is 2, thus the cycle will be broken once any link is deleted from it. Without loss of generality, Algorithm 2 tries to delete a link started from node $loc_1$ by checking all possible links. Consequently, there will be $U-1$ rounds of testing. For each round, a replica of $GF_{C_1}$ is generated as $GF_{C_2}$, and the later operations are conducted upon that replica (Line 5).

A hypothetical link $Test$ is initialized (Line 6). Then the link $Test$ is deleted from $GF_{C_2}$ to expose more anchor cells (Line 7 to 10). Having selected an anchor cell, the decoding operations are conducted to list the hidden links (Line 11). If all the involved cells in the decoding operations are finally transformed as Type 4, the hypothesis that link $Test$ belongs to the cycle is correct (Line 12 to 13). Thus the listed links will be added into $\Delta_D$ and the algorithm will be terminated (Line 14 to 15). Otherwise, the algorithm proceeds to the next round of testing. In effect, the number of such tests follow a typical Geometric distribution with $p = \frac{2}{U-1}$. Thus the expected number of tests is $\frac{U-1}{2}$.

Meanwhile, this suppose-and-check strategy can also be employed to settle the undecodable situations where multiple independent cycles are there in $GF_{C_1}$. By executing Algorithm 2 multiple times, the cycles recorded in $GF_{C_1}$ can be broken up one by one. In this recursive manner, Algorithm 2 derives out all different links in $GF_{C_1}$. Let $C$ count the number of independent cycles in $GF_{C_2}$, and $U = \sum_{i=1}^{C} u_i$ ($u_i \geq 3$) record the total nodes remained in $GF_{C_1}$, where $u_i$ is the number of nodes in the $i$th cycle. Then the expectation number of tests in Algorithm 2 is formed as:

$$\frac{U-1}{2} + \sum_{i=2}^{C} \frac{U-1-S_{j=1}^{i-1} u_j}{2}.$$  

Note that since the cycles are independent, decoding a cycle will not affect others. When decoding the first cycle, there are only two correct neighbours among the $U-1$ nodes. Thus the expected number of tests in Algorithm 2 is $\frac{U-1}{2}$. Similarly, when decoding the $i$th cycle, there are $U-1-S_{j=1}^{i-1} u_j$ candidate neighbours, but only two of them are correct. So the expectation is $(U-1-S_{j=1}^{i-1} u_j)/2$. Thus Eq. 2 can be formulated. That is, the expected execution rounds of Algorithm 2 is proportional to the total number of nodes $U$ and the number of independent cycles $C$.

Actually, when multiple cycles share common links or nodes, Algorithm 2 will fail to decode these different links from $GF_{C_1}$. We argue that, on one hand, this special case seldom happens in a large-scale computer network. On the other hand, we note that the nodes associated with these different links are still recorded in $GF_{C_1}$. Hence, comparing these nodes in $G_A$ and $G_B$ directly can also uncover these different links.

5 Probability analysis of the XOR trap

For an arbitrary node in a topology, graph filter aggregates the information of its neighbours with XOR operations. This compression ideology definitely cuts down the storage cost and lessens the communication overhead of topology calibration, but at the risk of miscalculations during decoding. Therefore, in this section, we quantify the probability of a XOR trap. For a XOR trap, different pairs of nodes generate the same XOR result, leading to negative impacts on the succedent decoding operations. Let $l$ be a constant and denote the number of bits for an id, and $n$ counts the number of nodes in a given topology, respectively. To simplify the analysis, we consider the situations where $n$ is equal to $2^l - 1$ exactly. Note that $l$ bits can identify $2^l$ nodes, but we don’t employ 00 ··· 00 to represent any node, thus the number of nodes is decreased to $2^l - 1$.

For an arbitrary node $e$, let $\Lambda_A$ and $\Lambda_B$ be the neighbours of $e$ in $G_A$ and $G_B$, respectively. We call the XOR result
of the possible combinations of the neighbours as an intermediate result. Let \( \Phi_A \) and \( \Phi_B \) denote the respective intermediate results in \( GF_A[i] \) and \( GF_B[i] \). Note that, different neighbours lead to diverse \( \Lambda_A \) and \( \Lambda_B \). For example, if \( e \) has three neighbours in \( G_A \), i.e., 101, 111 and 011, then there will be four intermediate results of the XOR operations, including 010 which is generated by XORing 101 and 111, which is the XOR result of 101 and 011, 100 which aggregates the information of 111 and 011, and 001 which represents all the three neighbours 101, 111 and 011. For a neighbour \( id \) in \( GF_A[i] \), it incurs a XOR trap if it shares the same binary bit with an intermediate result in \( GF_B[i] \). By contrast, an intermediate result may be generated from different combinations of neighbours. Therefore, an intermediate result in \( GF_A[i] \) suffers from a XOR trap when it conflicts with a neighbour or the same intermediate result is caused by other combinations of neighbours in \( GF_B[i] \).

**Theorem 1.** For a binary vector \( \eta \) in \( \Lambda_A \cup \Phi_A \), there are different combinations of neighbours in \( G_B \) which may conflict with \( \eta \). We define such combinations as the conflict combinations. These conflict combinations have diverse number of neighbours, ranging from 1 to \( |\Lambda_B| \). Let \( \Omega(\eta, \alpha, |\Lambda_B|) \) be a function which calculates the number of possible situations where \( \alpha \) conflict combinations are contained in \( GF_B[i] \), leading to XOR trap(s). Let \( p_\lambda(\eta) \) and \( p_\phi(\eta) \) denote the probability that \( \eta \) incurs XOR trap(s) with \( GF_B[i] \) when \( \eta \in \Lambda_A \), and \( \eta \in \Phi_A \) respectively. Then we have:

\[
p_\lambda(\eta) = \sum_{\alpha=1}^{\lfloor |\Lambda_B|/2 \rfloor} \frac{\Omega(\eta, \alpha, |\Lambda_B|)}{2^{(|\Lambda_B|-2)}}, \tag{3}
\]

\[
p_\phi(\eta) = \sum_{\alpha=1}^{\lfloor |\Lambda_B|/2 \rfloor} \frac{\Omega(\eta, \alpha, |\Lambda_B|)}{2^{(|\Lambda_B|-2)}}. \tag{4}
\]

**Proof:** When calculating \( p_\lambda(\eta) \), we consider all possible combinations of neighbours in \( G_B \) whose XOR results are the same as \( \eta \). For a neighbour of \( e \) in \( GF_A[i] \) with \( id=\eta \), the number of neighbours in the conflict combinations must be no less than 2. The reason is that, if \( GF_B[i] \) also has a neighbour with \( id=\eta \), it is not a XOR trap. That is why the value of \( \alpha \) ranges from 1 to \( |\Lambda_B|/2 \) in Eq. 3. Based on the definition of \( \Omega(\eta, \alpha, |\Lambda_B|) \), there are in total \( \sum_{\alpha=1}^{\lfloor |\Lambda_B|/2 \rfloor} \Omega(\eta, \alpha, |\Lambda_B|) \) possible situations where the neighbour in \( G_A \) whose \( id \) is \( \eta \) is in XOR trap(s) with the intermediate results in \( GF_B[i] \). Consider the \( |\Lambda_B| \) neighbours in \( G_B \) may have \( 2^{(|\Lambda_B|-2)} \) possible cases, the denominator can thus be deduced.

As for \( p_\phi(\eta) \), on the contrary, the neighbour in \( G_B \) whose \( id \) is \( \eta \) also causes a XOR trap with the intermediate result \( \eta \) in \( GF_A[i] \). Consequently, the value of \( \alpha \) belongs to the interval \([1, |\Lambda_B|/2]\) instead. For example, when \( |\Lambda_B|=5 \), the value of \( \alpha \) can be \([5/2]=3\), meaning that there are three conflict combinations which contain exactly 2, 2 and 1 neighbour(s), respectively. By contrast, for a neighbour with \( id=\eta \) in \( GF_A[i] \), \( \alpha=1, [5/2]=2 \) because its conflict combinations do not contain the neighbour whose \( id \) is \( \eta \) in \( GF_B[i] \). Similar as \( p_\lambda(\eta) \), the value of \( p_\phi(\eta) \) can be derived out as Eq. 4. Thus Theorem 1 is proved.

According to Theorem 1, we can quantify the probability that no XOR trap is introduced into the cell \( GF_C[i] \) as:

\[
p_c = p_\lambda \times p_\phi = \left( \prod_{\eta \in \Lambda_A} (1 - p_\lambda(\eta)) \right) \times \left( \prod_{\eta \in \Phi_A} (1 - p_\phi(\eta)) \right), \tag{5}
\]

where \( p_\lambda \) and \( p_\phi \) denote the probability that the neighbours and intermediate results in the \( GF_A[i] \) incur no XOR trap with \( GF_B[i] \), respectively. We note that, calculating the exact value of \( p_c \) is not easy, since the function \( \Omega(\eta, \alpha, |\Lambda_B|) \) requires to list all the possible conflict combinations of neighbours for a given \( \eta \). However, intuitively, when the node \( e \) has larger number of neighbours in \( G_A \) and \( G_B \), the graph filter suffers from XOR trap with a higher probability.

In the framework of graph filter, XOR trap is not avoidable. But only when the different links in \( \Delta_E \) get involved into XOR traps, will the calibration accuracy be affected.

**6 Evaluation**

In this section, we conduct comprehensive evaluations of the proposed graph filter. Typically, we compare graph filter with other data structures, including the adjacency matrix and adjacency table, in terms of the storage cost, decoding accuracy, and time-consumption. Without loss of generality, we first generate diverse scale of Erdős-Rényi (ER) random graphs [24] as our input topologies. The connectedness of the topology is determined by a given probability \( p_0 \), which quantifies the probability that a given pair of nodes are interconnected. Thereafter, two representative data center topologies, i.e., Fat-Tree [13] and Scaﬁda [14], are employed to further test the performance of graph filter. Last, the proposed cycle destruction algorithm is evaluated with different parameter settings. Note that the decoding accuracy is qualiﬁed by both the false positive rate and the false negative rate of the decoded results. All algorithms and methods are implemented on a test-bed with 2.6 GHz CPU and 16 GB RAM.

**6.1 Comparing graph filter with other data structures in random graphs**

Except for graph filter, general data structures, e.g., adjacency matrix and adjacency table, are also competent to record the information of a given topology. Hence in this subsection, we compare our graph filter with these data structures in terms of the communication overhead, decoding accuracy, and decoding time-consumption. The parameters we considered include the number of nodes in \( G_A \) or \( G_B \), i.e., \(|V|\), the number of different links remained between \( G_A \) and \( G_B \), i.e., \(|D|\), and the probability \( p_0 \) in ER random graphs.

**6.1.1 Comparison of communication overhead**

To begin with, we compare the storage or communication overhead caused by transmitting graph filter, adjacency matrix, and adjacency table. Since the employed data structures need to be transmitted between two hosts, the communication overheads of topology calibration for them are exactly the storage costs of these data structures. Fig. 4(a) depicts the result when the number of nodes in \( G_A \) and \( G_B \) increases from 100 to 1,500. Given the parameter of ER...
random graph \( p_0 = 0.1 \), we calculate the storage cost when representing the topologies with the above data structures. For legibility, we plot the 10-base logarithm instead of the real value. Definitely, the adjacency table consumes the most storage since two pointers are needed in a node of the adjacency table. By contrast, graph filter needs the least amount of storage. The reason is that adjacency tables call for plenty amount of storage to construct themselves, while the adjacency matrices are not storage-efficient since they can be quite sparse. Thus, graph filter outperforms the other data structures in terms of storage cost.

6.1.2 Comparison of time-consumption

Thereafter, the time-consumption of deducing the \( \Delta_E \) is compared in Fig. 4(b), (c) and (d), when \( |V|, |D| \) and \( p_0 \) varies, respectively. We also plot the 10-base logarithm value. Note that the legends “Matrix” and “Table” represent the caused time-consumption with the methods of the adjacency matrix and adjacency table, respectively. “GF-a” counts the time-consumption of the original decoding operation of graph filter. By contrast, “GF-o” records the resulted time-consumption of the augmented decoding operation.

Obviously, given the same parameters, the caused time-consumption by all the methods increases with the growth of \( |V| \). Clearly, when the value of \( |V| \) increases from 200 to 3000, \( |D| \) and \( p_0 \) are given as 100 and 0.1, the adjacency table based method performs the worst, while deriving \( \Delta_E \) by comparing two adjacency matrices takes the least time. Note that decoding the links in \( \Delta_E \) via graph filter reversely is slightly slower than the adjacency matrix enabled scheme, but much faster than the adjacency table based method. The original and augmented decoding algorithms of graph filter cost similar time to deduce the remained links from \( GF_C \). The augmented decoding operation terminates the decodings towards empty cells to cut down false positive rate. But the number of illegal anchor cells is negligible to affect the total required decoding time in this experiment.

6.1.3 Comparison of decoding accuracy

By comparing the associated adjacency matrices or adjacency tables, the different links between two given topologies can be correctly derived. So the accuracies of both the adjacency matrix and the adjacency table are 100%. Thus in this section, we quantify the false positive rate (FPR) and false negative rate (FNR) of both the original and augmented decoding algorithms for graph filter. Fig. 5, Fig. 6 and Fig. 7 record the generated false positive rate and false negative rate under diverse parameter settings, respectively. Note that, “Missed-a” and “Missed-o” represent the false negative rate of the original and augmented decoding operation, respectively. “Added-o” and “Added-a” count the
false positive rate of the original and augmented decoding algorithms, respectively.

Firstly, Fig. 5(a) and Fig. 5(b) depict the false positive rate and false negative rate when the number of nodes in $G_A$ and $G_B$ is changed from 200 to 3000. With the increase of $|V|$, the value of both “Missed-o” and “Missed-a” decrease drastically. “Added-a” in Fig. 5(a) exhibits a similar downward trend. Theoretically, when $|V|$ increases with fixed $|D|$, the decoding of graph filter suffers from less XOR traps; hence both the false positive rate and the false negative rate will decrease. However, as plotted in Fig. 5(a), “Added-o” incurs unstable performance. It is also clear that the augmented decoding operation suffers from a much lower false positive rate, at the cost of a slight increase of false negative rate.

Then we vary the number of different links in the two topologies from 20 to 280 and record the resulted false positive rate and false negative rate in Fig. 6(a) and Fig. 6(b), respectively. Fig. 6 demonstrates that when $|D|$ grows, the false positive rate, as well as the false negative rate, will rise significantly. The intrinsic reason is that, for graph filter decoding, more different links lead to an increased risk of XOR traps. Surprisingly, the augmented decoding strategy outperforms the original decoding operation in terms of both false positive rate and false negative rate when $|D|$ increases.

Lastly, when the parameter $p_0$ in the ER graph model increases from 0.05 to 0.5, the resultant false positive rate and false negative rate are depicted in Fig. 7(a) and Fig. 7(b), respectively. Larger $p_0$ means more links in the topologies and a higher risk of XOR traps. As illustrated in Fig. 7, graph filter decodes more missed and added links. Clearly, the augmented decoding algorithm outperforms the original decoding strategy in terms of false positive rate but at the cost of a little higher false negative rate.

To conclude Fig. 5, Fig. 6 and Fig. 7, our augmented decoding algorithm can significantly decrease the false positive, but at the risk of missing some different links.

### 6.2 Evaluating the graph filter in data center topologies

Beyond the general ER random graph, we further evaluate the performance of graph filter in two typical data center topologies, i.e., Fat-Tree and Scafida. We consider the number of switch ports in Fat-Tree, i.e., $K$, and the number of different links in the networks, i.e., $|D|$. The metrics include time-consumption of decoding, false positive rate, and false negative rate. The results are specified in Fig. 8, Fig. 9 and Fig. 10. For simplicity, we implement the augmented decoding operation of graph filter in Fat-Tree and Scafida.

Note that, as a representative structured data center topology, Fat-Tree consists of $5K^2/4$ commodity switches which interconnect $K^3/4$ servers. The number of links in a Fat-Tree can be calculated as $3K^2/4$. On the contrary, Scafida is a typical random or irregular topology for data centers. For fairness, in our experiments, the total number of both links and nodes in Fat-Tree and Scafida are set as the same. But they are configured with different rules, and hence have totally diverse topological characteristics (network diameter, degree distribution, and bisection bandwidth).

When $K$ varies from 10 to 24 and $|D|$ is given as 400, Fig. 8(a), Fig. 9(a) and Fig. 10(a) depict the performance of graph filter in Fat-Tree and Scafida, respectively. As plotted in Fig. 8(a), graph filter needs a little more time to accomplish the decoding operations in Scafida than in Fat-Tree. Indeed, Scafida is a highly skewed topology and the majority of nodes in the network are connected with several giant nodes. Accordingly, these giant nodes may associate with multiple different links. Yet this kind of links cannot be decoded directly from the giant nodes since giant nodes will not be selected as anchor cells. Graph filter has to wait until the other ends of this kind of links are searched as legal anchor cells. That is why decoding in Scafida is a bit more time-consuming.

According to Fig. 9(a) and Fig. 10(a), both the false positive rate and false negative rate of graph filter decrease immediately when $K$ increases. Note that when $K=10$, both the false positive rate and false negative rate are very high, which implies the graph filter is not practical in this situation. However, we argue that $K=10$ means 750 links in the networks. Changing 400 links among the 750 links simultaneously is unrealistic since computer networks evolve incrementally rather than aggressively. Besides, compared with Scafida, decoding in Fat-Tree leads to a bit higher
false positive rate but a little lower false negative rate. The reason is that, in the experiments, the different links are generated randomly. The different links in Fat-Tree are more dispersed than those in Scafida, since the giant nodes in Scafida have a higher probability to associate with links in $\Delta_E$. That is, graph filter handles fewer nodes when decoding. Consequently, graph filter may miss more links but add fewer links into $\Delta_D$ in Scafida.

Thereafter, Fig. 8(b), Fig. 9(b) and Fig. 10(b) together reveal the impact of $|D|$ in Fat-Tree and Scafida when $K = 20$. Definitely, when more different links are introduced into the network, graph filter needs more time to derive them out. As shown in Fig. 8(b), decoding in Scafida is still a bit more time-consuming than in Fat-Tree. Moreover, according to Fig. 9(b) and Fig. 10(b), decoding in Scafida results in a higher false negative rate but lower false positive rate. The basic reason is that, unlike in Fat-Tree, in Scafida, the different links are mainly gathered around the giant nodes. Also, when $|D|$ grows, graph filter suffers from increasing false positive rate and false negative rate. But in reality, to ensure availability, a computer network is not allowed to encounter a large number of simultaneous link failures during a time period. This guarantees the practicability of graph filter.

Based on Fig. 8, Fig. 9 and Fig. 10, we can conclude that the graph filter shows disparate performance in differentiated network topologies. And graph filter can be implemented in networks which evolve incrementally.

### 6.3 Evaluation of the cycle destruction algorithm

As stated in Section 4.3, when the different links form cycles in the topology, Algorithm 2 will be employed to break up the potential cycles via a suppose-and-check strategy. If there are no links or nodes participate in multiple cycles, Algorithm 2 can decode the links with neither false positive errors nor false negative errors. Accordingly, we only depict the time-consumption of Algorithm 2 in Table 1.

Typically, we consider the impact of number of cycles $C$, the number of nodes $|V|$ and the number of different links in the cycles $|D_C|$. Table 1 records the resultant time-consumption of breaking up the cycles when $C$, $|V|$ and $|D_C|$ increase, respectively. As calculated in Eq. 2, the expected execution rounds of Algorithm 2 is proportional to the total number of nodes $U$ (for independent cycles, $U = |V|$), and the number of cycles $C$. Therefore, as shown in Table 1, when $C$ and $|D_C|$ increase, the caused time-consumption grows apparently. Besides, given $C$ and $|D_C|$ as $15$ and $100$ respectively, we vary the total number of nodes $|V|$ in $GFC$, from $500$ to $5500$. Table 1 demonstrates that larger $|V|$ will slow down the decoding process, since searching anchor cells must check all the cells. That is, by following the suppose-and-check strategy, Algorithm 2 can correctly destruct the cycles in $GFC$, at the cost of additional time-consumption.

As a summary of the whole evaluation section, graph filter is more space-saving than other data structures and its time-consumption is acceptable. The augmented decoding operation can significantly decrease false negative rate, at the risk of a bit higher false positive rate. Moreover, our graph filter has disparate performance in differentiated data center topologies. Lastly, the proposed cycle destruction algorithm can indeed break up the potential cycles and decode the underlying links. Thus, we believe that graph filter is practical for networks where a large number of simultaneous failures are not allowed.

### 7 Discussion

We discuss the following issues for further understanding.

**Searching the different nodes.** Graph filter can decode the different links from the subtracting results. When $V_A \subset V_B$ or $V_B \subset V_A$, graph filter can be employed to derive the different links, as well as the different nodes between $G_A$ and $G_B$. In effect, an implicit assumption of graph filter is that, in the given topologies, the same labels imply the same nodes. When subtracting, graph filter can identify the different nodes between $G_A$ and $G_B$ via comparing $GFA$ and $GFB$. Furthermore, the subtracting operation will remain the different links root from the nodes in $\Delta_V$. Therefore, the decoding operation can also inversely decode the different links from $GFC$. Note that, when $V_A \cup V_B \neq V_A$ and $V_A \cup V_B \neq V_B$, identifying the different nodes can be a bit more challenging. By introducing another field to label each cell, graph filter can solve this problem. The redesign is left as our future work.

**Calibrating graphs with diverse scales.** Note that, for simplicity, we assume that the two topologies to be
calibrated has the same $|V|$. Actually, the graph filter is also capable of calibrating topologies with diverse scales. For instance, given $G_A$ and $G_B$ with $|V_A|$ and $|V_B|$ nodes respectively and $|V_A| < |V_B|$, the generated $GF_A$ will attach $|V_B| > |V_A|$ empty cells to enable the thereafter subtracting and decoding operations. This ensures the generality of graph filter in real use.

Undecodable situations. In this paper, we propose a cycle destruction algorithm to decode the links remained in $GF_C$, which form independent cycles. Especially, when links or nodes participate in multiple cycles, Algorithm 2 will fail to break these cycles. In this case, the graph filter will miss these different links, but the associated nodes are determined by comparing these nodes in $G_A$ and $G_B$. The remained different links can finally be derived out. But in fact, the probability that a node or link participates into multiple cycles is insignificant and negligible.

False positive and false negative errors. The XOR operations are employed to eliminate the same neighbours for a node in $G_A$ and $G_B$. However, during encoding and subtracting, diverse pairwise of nodes may result in the same XOR result. Consequently, when decoding, graph filter may miss a few links in $\Delta_F$. Besides, as stated in Section 4, the potential XOR traps may also cause false negative errors via adding extra links into the decoding results. However, the missed different links may be derived out in the next round of calibration. Also, the added links will be finally found and eliminated from $\Delta_B$ when network operators try to fix the failed or miswired links.

Flow transmission during topology calibration. During topology calibration, the applications route their flows with the existing version of topology. In the case of an unconnected path, retransmission or rerouting of these packets is triggered. This issue is beyond the scope of this paper, but a dozen of schemes or protocols are designed for such a puzzle [28] [29] [30] [31]. In wireless networks, C. Xu et al. optimizes the concurrent multipath transfer by reducing the reordering delay and unnecessary fast retransmissions [28]. Especially, the decoupling of control plane and data plane in SDNs enables the fast retransmission and recovery of packets with multiple paths in networks [29] [30] [31].

Generality of graph filter. In effect, graph filter may lose the generality of calibrating topologies but is competent to represent any irregular topology. Graph filter represents a topology by aggregating the neighbours of any node with the XOR operations, which makes this data structure space-efficient. With this design philosophy, graph filter can represent any irregular topology efficiently, including topologies with self-loops and directed links. Especially, to represent the nodes with self-loops, graph filter just needs to XOR the id of those nodes into their cells respectively. For the directed links, graph filter just aggregates the in-degree neighbours with XORs. However, graph filter cannot calibrate topologies with self-loops or directed links. The reason is that graph filter decodes the different links by recursively exposing anchor cells. The self-loops and directed links may interrupt that recursion since their information is stored only in one single cell. However, by comparing the generated graph filters, it is still possible to locate the different nodes in two irregular topologies.

Deploying graph filter in networks. To calibrate topologies, graph filter is deployed in the topology module of the network. For source-routing multicast, the graph filter is deployed in the source server to establish the multicast tree correctly. In SDN scenarios such as data centers, the distributed controllers implement graph filters to calibrate topologies from others, so that they can locate the difference among their topology views.

8 Conclusion
In this paper, we characterize and tackle the topology calibration problem in DCNs, to ensure the QoS of topology enabled protocols and applications in computer networks. The existing data structures, however, fail to represent the topological information of a graph and call for non-trivial memory cost and time-consumption to deduce the links in $\Delta_F$. And the existing methods for general set calibration is not applicable for topology calibration problem. Consequently, we design a novel data structure, named graph filter, to represent and deduce the different links. Graph filter executes the associated encoding, subtracting and decoding operations to derive the different links recursively and inversely. Furthermore, to enable successful decoding of circular different links, we propose a cycle destruction algorithm for graph filter. The experiments demonstrate that graph filter achieves the design rationales simultaneously.

Acknowledgment
The authors would like to thank anonymous reviewers for their constructive comments. This work is partially supported by National Program for Support of Top-Notch Young Professionals of National Program for Special Support of Eminent Professionals, Tianjin Science and Technology Foundation under Grant No. 18JYJCGX0290, and National Natural Science Foundation of China under Grant No.61772544.

References


Lailong Luo received his B.S. and M.S. degree at the school of information system and management from National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively. He is currently working toward a Ph.D degree in College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include computing networks and probabilistic data structures.

Deke Guo received the B.S. degree in industry engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008. He is currently a Professor with the College of System Engineering, National University of Defense Technology, and is also with the College of Intelligence and Computing, Tianjin University. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a senior member of the IEEE and a member of the ACM.

Jia Xu was born in 1984. She received the Ph.D. Degree in Computer Science from Northeastern University of China in 2013. She is currently an associate professor and master supervisor at School of Computer, Electronics and Information in Guangxi University. Her research interests cover data center management, data privacy protection, big graph query optimization and distributed and parallel data processing.

Xueshan Luo received his B.E. degree in information engineering from Huazhong Institute of Technology, Wuhan, China, in 1985, and his M.S. and Ph.D degrees in system engineering from the National University of Defense Technology, Changsha, China, in 1988 and 1992, respectively. Currently, he is a professor of Information System and Management, National University of Defense Technology. His research interests are in the general areas of information system and operation research.