IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 21, 2020, accepted March 11, 2020, date of publication March 30, 2020, date of current version April 15, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984258

Network Traffic Sampling System Based on Storage Compression for Application Classification Detection

SHICHANG XUAN[1], DEZHI TANG[1], ILYONG CHUNG[2], YOUNGJU CHO[3], XIAOJIANG DU[4], (Fellow, IEEE), AND WU YANG[1]

[1]Information Security Research Center, Harbin Engineering University, Harbin 150001, China
[2]Department of Computer Engineering, Chosun University, Gwangju 61452, South Korea
[3]SW Convergence Education Institute, Chosun University, Gwangju 61452, South Korea
[4]Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA

Corresponding author: Wu Yang (yangwu@hrbeu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61802086, in part by the Heilongjiang Provincial Science and Technology Department under Grant GX18A008, and in part by the China Scholarship Council (CSC).

ABSTRACT With the development of the Internet, numerous new applications have emerged, the features of which are constantly changing. It is necessary to perform application classification detection on the network traffic to monitor the changes in the applications. Using RelSamp to sample traffic can provide the sampled traffic with sufficient application features to support application classification. RelSamp separately assigns counters for each flow to record the statistical features and introduces a collision chain into the hash flow table to resolve hash conflicts in the table entries. However, in high-speed networks, owing to the number of concurrent flows and heavy-tailed nature of the traffic, the storage allocation method of RelSamp results in a significant waste of storage on the traffic sampling device. Moreover, the hash conflict resolution of RelSamp causes the collision chains of several hash table entries to be excessively deep, thereby reducing the search efficiency of the flow nodes. To overcome the shortcomings of RelSamp, this study presents a sampling model known as MiniSamp. Based on the RelSamp sampling mechanism, MiniSamp introduces shared counter trees to compress the storage space of the counters during the sampling process and integrates an efficient search tree into the hash table. The search tree structure is adjusted according to the network environment to improve the search efficiency of the flow nodes. The experimental results demonstrate that MiniSamp can effectively aid network operators to classify traffic in the high-speed network.

INDEX TERMS Traffic sampling, application classification, shared counter tree, flow tracking, flow table structure.

I. INTRODUCTION

Since the birth of ARPANET in the late 1960s, following development over half a century, the Internet has achieved great success. During this development, the scale of the Internet has continued to increase. Using the Internet as fertile soil, various practical and novel applications have been produced continually, which has greatly enriched and facilitated people's everyday lives. However, numerous problems have arisen with the development of the Internet. For example, hackers may attack servers and cause them to be paralyzed, intruders may steal sensitive information by implanting Trojans, and the amount of traffic increasing dramatically at one point may place significant pressure on routers and generate network congestion. Some researchers have investigated how to suppress and eliminate the negative effects resulting from the Internet, and they have proposed some solutions [1]–[3]. Moreover, network operators must use traffic measurement technology to monitor the network and handle occurring problems in a timely manner. In recent years, owing to the continual emergence of new applications and constant changes in application characteristics, the measurement of traffic application characteristics has attracted increasing attention from network operators. Based on the measurement results obtained, network operators can detect new undesirable behaviors within applications, such as P2P system

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

vulnerabilities, and re-provision their networks to adapt to major application trends. Performing application classification on traffic is the key step in measuring the traffic application characteristics.

To support traffic application classification, the sampled traffic collected by the sampling algorithm should retain sufficient application features. At present, RelSamp [30] has achieved this target. RelSamp allocates the same counter space for each flow during the sampling process; however, owing to the heavy-tailed nature of the traffic, this allocation method may result in a huge waste of storage space on the traffic sampling equipment if RelSamp is deployed in a high-speed network. Moreover, because many concurrent flows exist in a high-speed network, multiple active flows will inevitably hit the same hash table entry and cause conflicts during the sampling process. RelSamp uses a conflict chain to resolve conflicts within the hash table, but this can cause the conflict chain in certain entries to be excessively deep, thereby decreasing the algorithm searching speed of the sampled flow nodes.

To solve the problems of RelSamp operating in a high-speed network, this study proposes a high-performance sampling model for application classification, known as MiniSamp. MiniSamp optimizes the flow table storage structure of RelSamp to reduce the storage overheads during the sampling process, as well as improves the hash conflict resolution method of RelSamp, with the aim of controlling the depth of the conflict chain in the hash table entry.

The remainder of this paper is organized as follows: Section 2 introduces the related work on traffic application classification and sampling technology. In Section 3, the working principle of MiniSamp is explained in detail. Section 4 presents the verification of the accuracy and feasibility of the proposed model through experiments. Finally, the research contents of this paper are summarized in the concluding section.

II. RELATED WORK

At present, the main methods for classifying traffic are those based on transport layer ports, deep packet inspection, and machine learning methods based on the statistical features of flows.

Application classification based on well-known port numbers registered in IANA is the traditional method. In current networks, numerous applications do not use well-known ports. For example, P2P applications usually listen on dynamic ports [4], while several applications such as embedded audio use well-known ports; for example, 80, for data transmission. Thus, it is not possible to identify the sampled traffic application types based on the port alone. Moore and Papagiannaki [5] and Dainotti *et al.* [6] verified that port-based classification methods cannot accurately classify current traffic.

Most protocols include a protocol characteristic string in the application layer payload to distinguish them from other protocols, and these strings are usually public or easily available. By extracting the payload characteristics from the packet, the deep packet inspection (DPI) method [7] constructs a characteristic rule library. DPI matches the load characteristics of the packets with the rules in the DPI rule library to identify the type of traffic application. DPI generally provides high application recognition accuracy. Sen *et al.* [8] proposed a P2P traffic detection method based on the application signature. This method divides the payload characteristics into fixed and variable offset characteristics, matching first the fixed and then the variable offset characteristics. The experimental results demonstrated that the false positive rate was less than 5% when using five types of popular P2P traffic as input. However, DPI cannot identify encrypted traffic, and once the payload characteristics of the application are changed, the DPI rule library must be updated, the workload of which is enormous. Moreover, the interpretation of payloads involves sensitive issues such as privacy protection. Therefore, DPI is generally used as the basis for judging the accuracy of other classification approaches.

Machine learning approaches are mainly divided into two categories: supervised and unsupervised. Supervised approaches use the training traffic that is labeled with the application types to train the classifier and input the labeled traffic into the trained classifier for application classification. Common supervised approaches include support vector machines (SVMs) [9], k-nearest neighbors (KNN) [10], naive Bayes [11], and decision trees [12]. Unsupervised machine learning approaches take advantage of the similarities between samples to classify unlabeled traffic. Well-known unsupervised approaches include k-means [13], AutoClass [14], and DBSCAN [15]. Machine learning approaches can classify encrypted traffic through the statistical features of the traffic. Dong and Jain [16] improved the Bayesian classifier model and proposed a new approach to identify Skype [17] encrypted traffic, whereby the sampled traffic was input into the classifier in NetFlow [18] flow records. The experiments demonstrated that the accuracy of this approach in recognizing Skype encrypted traffic could reach 93.6%. Jun and Shunyi [19] were the first to extract features from network traffic and to select relevant features from the extracted features through genetic algorithms, using Bayesian networks to identify P2P traffic. The experimental results demonstrated that, compared to traditional traffic classification approaches, the classification speed and accuracy of K2 [20], TAN [21], and BAN were significantly improved. However, this approach is based on probability and relies heavily on the sample space distribution.

Xu *et al.* [22] proposed a traffic classification approach based on the SVM. This method effectively avoided the negative effects caused by unstable factors on the classification accuracy by using two optimization strategies. A comparison of this approach with NB, NBK, and NBK + FCBF [23] revealed that its accuracy remained higher than that of NB and slightly higher than that of NBK + FCBF without the feature selection algorithm.

Machine learning approaches can identify encrypted traffic and provide high accuracy in traffic application classification. Therefore, machine learning has become a research hotspot in the field of traffic application classification.

At present, NetFlow is commonly used to sample traffic, whereby incoming packets are sampled randomly. Researchers have proposed solutions to several shortcomings of the random packet sampling mechanism. Estan *et al.* [24] proposed a method that can adaptively adjust the sampling ratio according to changes in network environments, which can effectively deal with harmful traffic such as DoS attacks [25]. Kompella and Estan [26] proposed a sampling method designed to adjust the memory and CPU utilization of the sampling device in an improved manner. Sekar *et al.* [27] designed a sampling method for monitoring traffic in the local network, which can minimize the redundancy of routers sampling data packets alone. Furthermore, several sampling methods focus on providing network operators with the flexibility to select sampled flows. For example, the sampling method designed by Yuan *et al.* [28] provides a flow set to sample the specified flows, and network operators can add the flows they wish to collect to the flow set. However, it is not clear how to establish the flow set to maintain sufficient application characteristics of the sampled traffic. Ramachandran *et al.* conducted similar work [29] and presented a sampling method that specifies the flows to be sampled by constructing counter-based judgment logic on the five-tuple information in the packet header.

A session of modern applications usually consists of multiple flows. The source IP of each flow in the session must be the same, while the destination IP may be different. If more flows are sampled within an application session, the sampled traffic will retain more application characteristics, thereby helping the classifiers to identify the application of the sampled traffic accurately. Lee *et al.* [30] proposed RelSamp, which samples the flows with a source IP within a certain range. Under the condition whereby the effective sampling ratio is constant, RelSamp can capture more flows within application sessions to retain additional application characteristics by improving the flow sampling probability and decreasing the packet sampling probability.

However, for any statistical features of the flow, such as the flow size, RelSamp must allocate a counter for each flow to record its size. The space allocated for each counter must be the same, and the count range of the counter must cover the maximum flow size. Traffic exhibits a heavy-tailed nature; that is, a small proportion of large flows occupies a large proportion of the traffic. Related studies [31] have demonstrated that, by sorting flows according to size, the top 15% of the flows account for 95% of the total traffic. Allocating the same counter for each flow to store the flow size inevitably causes a huge waste of storage on the traffic sampling device. Moreover, allocating the same counter for each flow to store other statistical features (for example, the number of FIN, SYN, and ACK packets) also wastes storage space, particularly when RelSamp is deployed in a high-speed network environment, which places significant pressure on the sampling device.

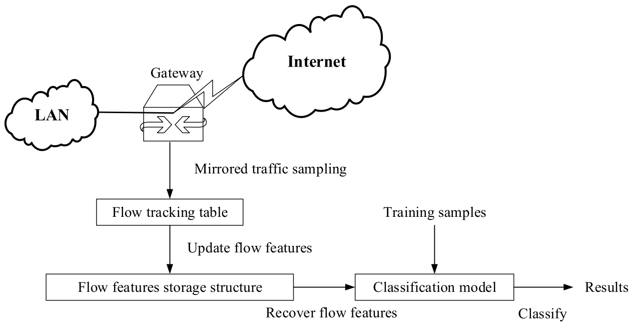

FIGURE 1. Overall framework of MiniSamp.

To date, apart from RelSamp, few sampling algorithms have been developed that can retain sufficient application characteristics within the sampled traffic. However, RelSamp does not take into account the huge storage waste caused by the space overheads for redundant counters in the high-speed network environment.

III. HIGH-PERFORMANCE SAMPLING MODEL FOR APPLICATION CLASSIFICATION DETECTION

As RelSamp allocates the same-sized counter space for each flow, causing a huge waste of storage space and relatively low searching efficiency for flow nodes during the sampling process, based on the RelSamp sampling mechanism, this study proposes a high-performance sampling model for application classification detection, known as MiniSamp. MiniSamp introduces shared counter trees [32] to compress the counter space storing the statistical features of the flows and incorporates an efficient search tree into the hash flow table entries, thereby adjusting the search tree structure according to the network environment in a timely manner. The goal of MiniSamp is less storage space being required and the flow nodes being located at a faster speed during the sampling process, as well as retaining the sampled traffic with sufficient application characteristics to support traffic application classification.

A. OVERALL FRAMEWORK OF MiniSamp

MiniSamp operates in the LAN of an enterprise or campus, with the aim of helping network operators to perform application classification detection of traffic in the LAN. MiniSamp samples the mirrored traffic through the LAN gateway and inputs the sampled flow feature records into the application classification model to generate classification result reports. The overall framework of MiniSamp is illustrated in Figure 1. The overall working process is as follows:

(1) Decide whether to sample incoming packets based on the sampling mechanism.

(2) If it is decided to sample the incoming packet, search the flow node to which the packet belongs in the hash

flow tracking table. Following searching, make optimization adjustments for the search tree structure according to the network environment. If the flow node to which the sampled data packet belongs does not exist, create a new flow node in the flow tracking table for the packet.

(3) The flow feature storage structure is composed of flow feature storage units. The flow feature storage unit consists of a sampling flow node and counters that store the statistical features of the sampled flow in the shared counter tree set. After locating the flow node to which the sampled packet belongs in the flow tracking table, extract the relevant features of the sampled packet, and update the flow features in the flow node as well as the shared counter tree set.

(4) When the sampling for a flow is terminated, recover the features stored in the flow node and the shared counter tree set, and import these into the ordered flow feature records buffer. When the buffer is full, write the sampled flow feature records into a file.

(5) Input the sampled flow feature record file into the pre-trained classifier and generate the classification results in terms of the flows.

B. MIRRORED TRAFFIC SAMPLING

During the sampling process, MiniSamp performs three stages of judgment on each incoming packet and decides whether to sample the packet based on the judgment results. The three phases are the source IP selection phase, flow selection phase, and packet selection phase. Three sampling probabilities exist, namely p_h, p_f, and p_p, where p_h can control the scale of the sampled source IP set in the source IP selection phase, $\mathrm{p_f}$ can control the amount of sampled flows in the flow selection phase, and p_p can control the amount of packets sampled in the sampled flow.

Prior to formal sampling, the sampling probabilities p_h, p_f, and p_p must be determined. Network operators usually configure the effective sampling ratios p_e and p_h ($p_h \geq p_e$) in advance and specify p_f^t according to the sampling purpose. In the process of determining the sampling probabilities, RelSamp uses an iterative algorithm to perform a binary search on the values of p_f and p_p continuously, making p_f approximately p_f^t continuously, under the condition that the effective sampling ratio is p_e and the value of p_p is obtained when the iterative process ends. After determining the sampling probabilities p_h, p_f, and p_p, all of the sampled flow feature records generated previously are discarded and the formal sampling process is started. The pseudocode for determining the sampling probability is presented in Algorithm 1.

The packet capture thread captures the packets on the traffic sampling device constantly and places the captured TCP or UDP packets into the packet buffer queue. A status flag exists in MiniSamp, which is set to 1 during the sampling process.

The sampling mechanism can be explained as follows: Firstly, MiniSamp fetches the packet from the packet buffer queue and assigns two random numbers in the range of 0 to 1 to the packet, which are denoted by r_f and r_p. In the source IP selection stage, the source IP of the packet is hashed using the hash function for this stage, and the hash value is multiplied by p_h to obtain the target value. If the target value falls within the preconfigured range, the flow selection stage is entered; otherwise, the packet is discarded. In the flow selection stage, the five-tuple of the packet (source IP, source port, destination IP, destination port, and transport layer protocol) is hashed using the hash function for locating the hash entry, and the packet flow node is located in the hash flow table based on the hash value. If the flow node is not searched and $r_f \leq p_f$, the packet is sampled, and a flow node is created in the flow table for the packet; if the flow node is not searched and $r_f > p_f$, the packet is discarded; otherwise, the packet selection phase is entered. In the packet selection phase, if $r_p \leq p_p$, the packet is sampled, and the packet flow storage unit is updated; otherwise, the packet is discarded. The process of the MiniSamp sampling is illustrated in Figure 2.

Algorithm 1 Set_Parameter($\mathrm{p_e}$, $\mathrm{p_f}$, $\mathrm{p_f^t}$)

```
Input: $p_e$ Preconfigured effective sampling ratio
       $p_f$ Preconfigured IP sampling probability
       $p_f^t$ Preset flow sampling probability
Output: packet sampling probability $p_p$

1  function Set_Parameter($p_e$, $p_f$, $p_f^t$)
2      $p_f \leftarrow p_e/p_h$;   //initialize flow sampling probability
3      $p_p \leftarrow p_e/p_h$;   //initialize packet sampling probability
4      $t \leftarrow 10^{-5}$;
5      $p_f \leftarrow 0.5 * (p_f + p_f^t)$; //make $p_f$ approximately $p_f^t$
6      $p_x \leftarrow$ current_sampling_ratio(); //get the current sampling ratio
7      while $p_f ! \approx p_f^t$ do
8          while $p_x ! \approx p_e$ do
9              if $p_x > p_e$ then   //if the current sampling ratio is greater than the preconfigured effective sampling ratio
10                 $p_p \leftarrow 0.5 * (p_p + t)$ ;//halve packet sampling probability
11             else   //if the current sampling ratio is less than the preconfigured effective sampling ratio
12                 $p_p \leftarrow 0.5 * (p_p + 1)$ ;   //slightly increase packet sampling probability
13             end if
14         end while
15         $p_f \leftarrow 0.5 * (p_f + p_f^t)$; //make $p_f$ approximately $p_f^t$
16     end while
17     return $p_p$;
18 end function
```

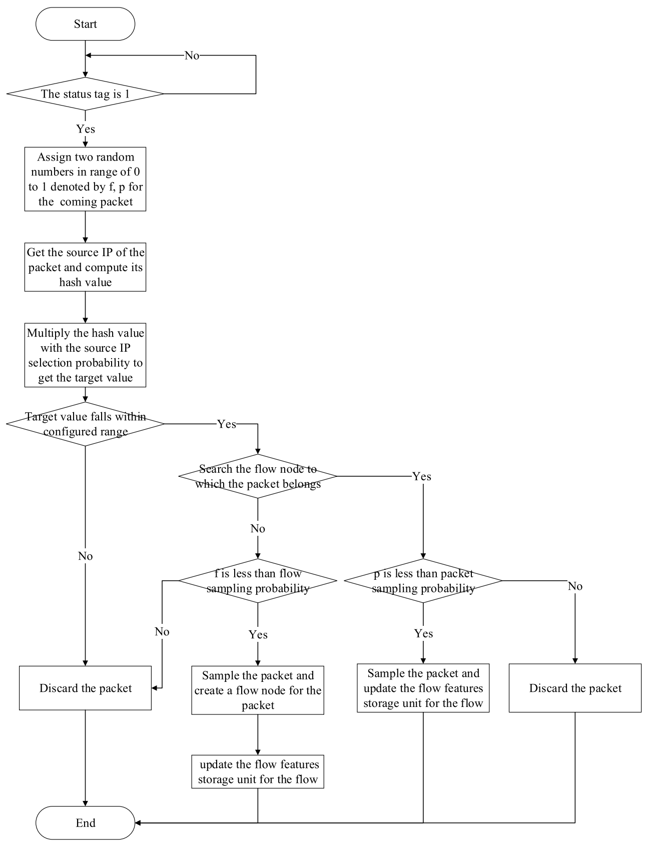
Start
The status tag is 1
No
Yes
Assign two random numbers in range of 0 to 1 denoted by f, p for the coming packet
Get the source IP of the packet and compute its hash value
Multiply the hash value with the source IP selection probability to get the target value
Target value falls within configured range
Yes
No
Search the flow node to which the packet belongs
Yes
No
f is less than flow sampling probability
p is less than packet sampling probability
No
Yes
Yes
No
Discard the packet
Sample the packet and create a flow node for the packet
Sample the packet and update the flow features storage unit for the flow
Discard the packet
update the flow features storage unit for the flow
End

FIGURE 2. Process of MiniSamp sampling.

C. FLOW TRACKING TABLE

In the high-speed network environment, the length of the hash flow tracking table is substantially smaller than the number of active concurrent flows; therefore, multiple active flows will hit the same hash table entry and cause conflicts. To control the depth of the conflict chain in the main hash table entry, an efficient search tree is introduced into the main hash table. This tree is a non-strictly splay tree. MiniSamp does not splay the hit flow node to the root of the search tree every time but uses a certain mechanism to ensure active flow nodes near the root of the search tree. To prevent continuous access to the long branches of the search tree, the flow node is splayed to the root using a double-splay method. The structure of the MiniSamp flow table is presented in Figure 3.

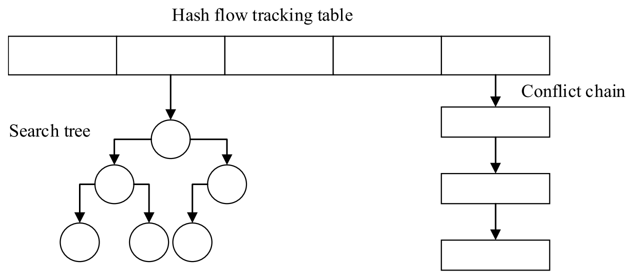

FIGURE 3. MiniSamp flow tracking table structure.

conflict counter	conflict chain pointer	search tree pointer

FIGURE 4. Main hash table entry structure.

In the main hash table entry, a conflict counter is used to record the number of conflicted sampled flows. A threshold t exists, which is far less than the number of concurrent flows in the network. If the number of conflicts recorded by the conflict counter does not exceed t, the conflict flow nodes are organized by the conflict chain; otherwise, the conflict flow nodes are organized in the search tree. The main hash table entry structure is presented in Figure 4.

The work process of searching the flow node to which the sampled packet belongs in the flow tracking table is described in the following, and the process chart is illustrated in Figure 5.

(1) The five-tuple (source IP, source port, destination IP, destination port, and transport layer protocol) of the sampled packet is hashed through the hash function for locating the hash entry and the remainder of the hash value is taken according to the length of the main hash table to locate the entry of the flow to which the packet belongs in the main hash table.

(2) In the located hash table entry, if the number of conflicts recorded by the conflict counter is greater than t, a binary search tree general search algorithm is used to search the target flow node in the search tree. If it is not searched, a new node is created for the flow and inserted into the search tree, using the standard binary search tree insertion algorithm, and the value of the conflict counter is incremented. If the target flow node is hit at the root of the search tree, the search process ends.

(3) In the located hash table entry, if the number recorded by the conflict counter is not greater than t, the flow node is searched linearly in the conflict chain. If it is not searched, a new node is created for the flow and inserted at the head of the conflict chain. The value of the conflict counter is increased by 1, and if the value after increasing is greater than t, the organization method of the conflict flow nodes is switched to search tree, thus ending the search process.

(4) After locating the target flow node or creating the new node for the target flow, MiniSamp requires O(1) time to maintain the longest distance of the search tree. Then, when making a judgment, if the distance is farther than the longest distance, the flow node is splayed to the root of the tree and the search process ends.

(5) If the target flow node is not farther than the longest distance from the root of the tree, MiniSamp requires O(1) time to check whether the packets of the target flow arrive continuously and maintain the relevant data structure required for detection. If the target flow packets have recently arrived constantly, the flow node is splayed to the root of the tree. Then, the search process ends.

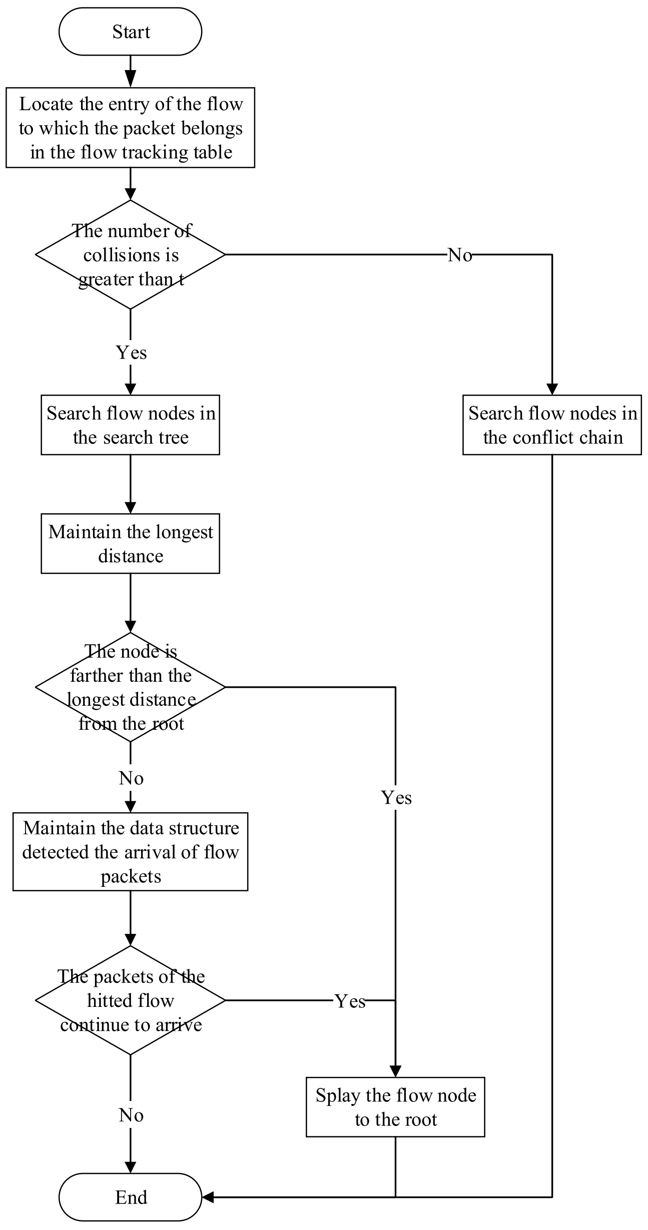
Start
Locate the entry of the flow to which the packet belongs in the flow tracking table
The number of collisions is greater than t
No
Yes
Search flow nodes in the search tree
Search flow nodes in the conflict chain
Maintain the longest distance
The node is farther than the longest distance from the root
No
Yes
Maintain the data structure detected the arrival of flow packets
The packets of the hitted flow continue to arrive
Yes
No
Splay the flow node to the root
End

FIGURE 5. Search flow node in flow tracking table.

Algorithm 2 Distance(tuple,tree,opt_table,counter,upper)

Input: tuple Packet's five-tuple
tree Search tree
opt_table hash table used for maintaining the longest distance of the search tree
counter Packet counter
upper Upper limit set for the packet counter
Output: longest distance of the search tree

```
function distance(tuple,tree,opt_table,counter,up)
  counter ← counter + 1;
  hash_value ← opt_fun(tuple);
  opt_table[hash_value%opt_table.len] ← 1;  // record the incoming packet flow in the hash table opt_table
  if counter < upper then  //the number of packets that hit the search tree recently is less than upper
    return tree.dis;  //no changes are made to the longest distance of the search tree
  else p
    m ← 0;
    for i := 0 to opt_table.len - 1 do //count the number of flows that recently hit the search tree
      if opt_table[i] == 1 then
        m ← m + 1;
      end if
    end for
    c ← log(m);
    n ← ceil(c);  //calculate the longest distance
    tree.dis ← n;  //update the longest distance of the search tree
    counter ← 0;  //reset the value of the packet counter to start a new adjustment
    for i := 0 to opt_table.len - 1 do //reset the entries' values of opt_table to start a new adjustment
      opt_table[i] ← 0;
    end for
  end if
  return tree.dis;
end function
```

The significance of maintaining the current longest distance of the search tree in each item of the main hash table is adjusting the search tree structure at the minimum cost during the search process to ensure that active flow nodes are near the root of the search tree, thereby improving the search efficiency of the flow nodes. MiniSamp can adaptively adjust the longest distance of the search tree according to changes in the network environment. A packet counter is used to record the number of packets that have reached the search tree within a short period, and the upper limit of this counter is set to the value upper. The hash table opt_table is used to record the active flows of the packets that have arrived in a short period. To reduce the probability of hash collisions, the length of opt_table is a prime number that is slightly larger than three times upper. The hash values generated by the hash function opt_fun are evenly distributed in opt_table. The pseudocode that maintains the current longest distance of the search tree is presented in Algorithm 2.

In the network environment, the flow packets may arrive continuously. To improve the search efficiency, this situation should be detected in real time. Once a flow node in the search tree is identified as being hit continuously, the node must immediately be splayed to the root of the tree. During the detection process, a parameter w exists. If w packets of a certain flow arrive continuously, MiniSamp determines that the packets of this flow will continue to arrive constantly. A circular queue of length w, detect_queue, is used

Algorithm 3 Detect(tuple,w,detect_queue,detect_table)

```
Input: tuple Packet's five-tuple
       w Parameter
       queue Circular queue used in the detection process
       detect_table Hash table used during detection
Output: 1 indicates detected; 0 indicates not detected
1  function detect(tuple,w,queue,detect_table)
2    hash_value ← detect_fun(tuple);
3    if !queue.full() then //when the circular queue is not full, the hash value of current packet's five-tuple is enqueued
4      queue.push_back(hash_value);
5      return 0;
6    else p
7      v ← queue.front(); //when the circular queue is full, take the value of the queue head element
8      v ← v % detect_table.len;
9      detect_table [v] ← detect_table[v] - 1; //update the hit number of queue head element's flow in the last w searches in the search tree
10     queue.pop();
11     queue.push_back(hash_value);   //enqueue the hash value of the current packet's five-tuple
12     u ← hash_value % detect_table.len;
13     detect_table[u] ← detect_table[u] + 1; //update the hit number of current packet's flow in the last w searches in the search tree
14     if detect_table[u] == w then //if the flow to which the current packet belongs has been hit w times in the last w searches in the search tree
15       return 1;          //return detected
16     else
17       return 0;          //return undetected
18     end if
19   end if
20 end function
```

to record the hash value of the five-tuples of recent packets. The hash table detect_table is used to record the number of hit times of the current active flows in the last w searches. To reduce the probability of hash collisions in detect_table, the length of detect_table is a prime number slightly larger than five times w, and the hash values generated by detect_fun are evenly distributed in detect_table. The pseudocode for detecting whether the packets of the target flow node arrive continuously and maintaining the data structure used in the detection process is presented in Algorithm 3.

D. UPDATING FLOW FEATURES

MiniSamp is a sampling model that supports application classification. To support the trained classifier in identifying the application types of sampled traffic, MiniSamp records the statistical features of flows that can be used for application classification during the sampling process. Kim *et al.* [33] and Li *et al.* [34] have studied these features for classifying traffic applications. With reference to the traffic application features extracted by these authors, and combined with our own research background, the flow record generated by MiniSamp is composed of the following flow features: transport layer protocol, source port, destination port, minimum payload length, maximum payload length, number of packets (flow size), total data length (flow length), average segment size, and number of ACK, SYN, FIN, and RST packets. Features such as the minimum payload length, maximum payload length, and number of packets can be used to identify UDP applications; the number of ACK, SYN, FIN, and RST packets and average segment size can be used to identify TCP applications; the transport layer protocol, source port, destination port, and total data length can be used to identify both TCP and UDP applications. In the UDP flow record, the statistical features of the TCP flags are set to 0.

MiniSamp stores the application features of the unidirectional flow in the flow node and shared counter tree set. The source IP, destination IP, source port, destination port, transport layer protocol, minimum payload length, maximum payload length, flow arrival time, and flow latest update time are stored in the flow node. The number of packets, total data length, and number of ACK, SYN, FIN, and RST packets are stored in their respective shared counter trees. Therefore, the flow feature storage unit is composed of a flow node and counters that record the flow statistical feature values in the shared counter tree set.

When a new flow feature storage unit is created for the sampled packet, the data packet is parsed. Thereafter, the five-tuple information, time of the flow arrival, time of the latest update of the flow, minimum payload length, and maximum payload length are written to the flow node. The latest update time and arrival time of the flow are both the current time, while the minimum payload length and maximum payload length are the application layer payload lengths of the packet. If the data packet is a TCP data packet, the TCP header is parsed, and the flags ACK, FIN, SYN, and RST are detected as set or not. If the flag is set, the Record function, which will be discussed later in this paper, is used in the corresponding shared counter tree to count the number of incoming flag packets. The Record function is used to count the number of packets for the sampled flow in the shared counter tree that stores the flow size. The length of the packet *len* is calculated, 32B is taken as a data block, and the number of data blocks denoted by c is obtained. The Record function is used to count the flow c times in a shared counter tree that stores the flow data blocks. The calculation formula for c is as follows:

$$c = \left\lceil \frac{len}{32} \right\rceil . \tag{1}$$

When updating the flow features storage unit of the packet, the latest update time of the flow in the flow node is firstly updated, following which the data packet is parsed. The payload length of the packet application layer is calculated and compared with the maximum payload length and minimum

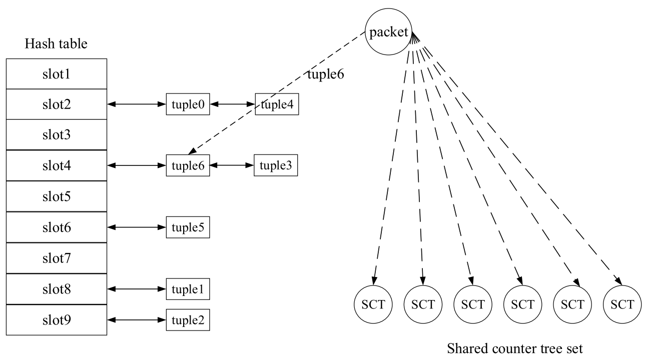

FIGURE 6. Updating flow feature storage unit instance.

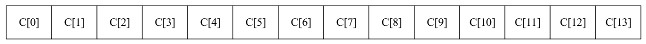

FIGURE 7. Shared counter tree storage structure.

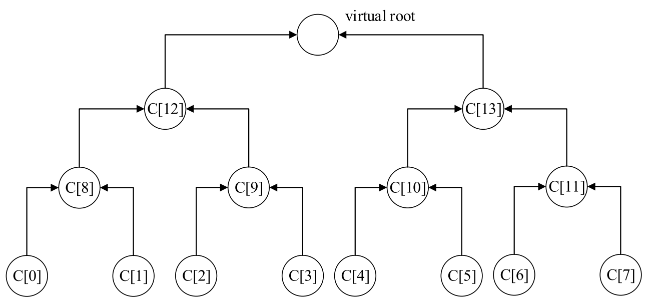

FIGURE 8. Shared counter tree logical structure.

payload length recorded. If the length is greater than the original maximum payload length, the maximum payload length is updated. The update operation for the shared counter tree is the same as that of the flow feature storage unit being established. An example of the MiniSamp update flow feature storage unit is illustrated in Figure 6, where SCT is the abbreviation for shared counter tree.

To record a certain statistical feature of the sampled flow during the sampling process, several shared counters are allocated to all sampled flows. These counters are logically organized into an approximate binary tree to form a shared counter tree in MiniSamp.

The shared counter tree is explained as follows: The storage space allocated for it is N bits, the space allocated for each counter is a bit, and the number of counters in the shared count tree is N/a. The height of the shared counter tree is h, with a total of h layers, wherein the lowest layer is layer 0 and the highest layer is layer h-1. The number of leaf nodes in the counter tree is p, and the degree of non-leaf nodes in the counter tree is 2. If the number of h-1 nodes exceeds 1, a virtual root node is set in the counter tree. A three-tiered shared counter tree storage structure is illustrated in Figure 7, and the logical structure is presented in Figure 8.

In the individual counter, the most significant bit is used as the status bit, and the remaining a-1 bits are used for counting. The storage structure of the counter is presented in Figure 9.

Status bit (1 bit)
Counter bits (a-1 bits)

FIGURE 9. Individual counter storage structure.

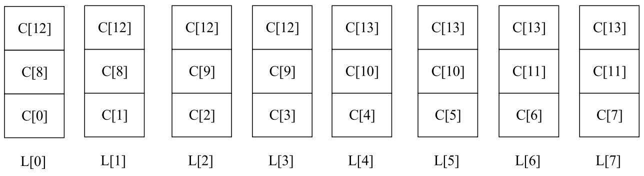

FIGURE 10. Enhanced counters.

Let $C[i]$, i $\in$ $[0,\ p)$ represent a certain leaf node of the counter tree. The nodes contained in the path from $C[i]$ to the root node constitute an enhanced counter, which is known as $L[i]$.$L\,[0] = \{C\,[0]\,, C\,[8]\,, C\,[12]\}$ is an enhanced counter. Because p leaf nodes exist in the counter tree, it contains p enhanced counters. The vector L is used to represent the enhanced counter set in the counter tree. As illustrated in Figure 11, $L = \{L\,[1]\,, L\,[2]\,, \ldots, L\,[p-1]\}$.

For any sampled flow f, r (r $\ll$ p) counters are selected from the p enhanced counters through the five-tuple information of the flow and r hash functions h_i, which are independent of one another. The r enhanced counters constitute an enhanced counter vector of the flow f. This vector is represented by L_f, and the i-th element in L_f is represented by $L_f[i]$. The specific calculation formula for $L_f[i]$ is as follows, where i $\in$ $[0,\ r)$, and the range of the hash function h_i is $[0,\ p-1)$:

$$L_f\,[i] = L[h_i(f)]. \tag{2}$$

To reduce the difficulty of designing hash functions, instead of actually designing r independent hash functions, we can design only one main hash function H and use a set S composed of r random elements to simulate r independent hash functions. The formula for the hashing flow f using the hash function h_i is as follows:

$$h_i\,(f) = H\,(f \oplus S[i])\,. \tag{3}$$

Because $r \ll p$, the probability of randomly selecting r different counters from the p enhanced counters of the shared counter tree is

$$\frac{p\,(p-1)\ldots(p-r+1)}{p^r} \approx 1. \tag{4}$$

Therefore, the enhanced counters in the vector L_f are mutually different.

In Figure 11, $r = 3$, the enhanced counter vector of the flow f is $L_f = \{L\,[1]\,, L\,[0]\,, L[4]\}$, and the enhanced counter vector of the flow g is $L_g = \{L\,[3]\,, L\,[4]\,, L\,[6]\}$. Among these, the enhancement counter $L[4]$ is shared by the flow f and g. Because different flows can share the same enhanced counter, the number of flows that can be recorded by the shared counter tree is much larger than the number of its enhanced counters.

The pseudocode for storing the statistical features such as the number of sampled flow packets, and number of ACK,

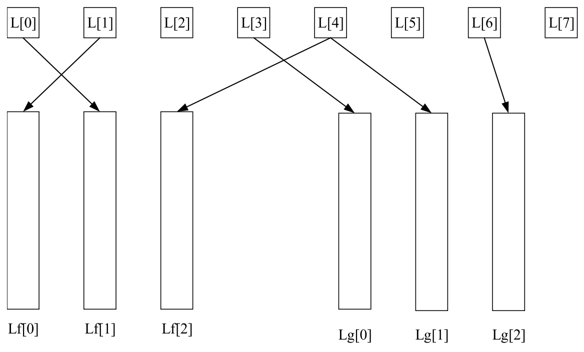

FIGURE 11. Enhanced counter vector.

SYN, FIN, and RST packets in the shared counter tree is presented in Algorithm 4.

The process for storing the sampled flow length in the shared counter tree is as follows.

Step 1: The sampled data packet is parsed, the data packet length is calculated, and 32B are used as a data block to calculate the number of data blocks c of the data packet.

Step 2: c ← c - 1.

Step 3: The Record function is used in the shared counter tree to count the number of data blocks in the flow to which the packet belongs.

Step 4: If c is less than 0, the counting process of the total length of the flow ends; otherwise, go to step 2.

E. RECOVERING FLOW FEATURES

The flow feature recover thread is started every 10 s. When the sampling of a flow ends, the thread recovers the flow features stored in the flow node and shared counter tree into a complete flow record, and this flow record is added to the flow record buffer queue.

The flow feature recovery thread firstly sets the status flag to 0. At this time, no sampling is performed, and the captured packets are temporarily stored in the packet buffer queue. In order not to affect the ongoing sampling operation, 0.25 s are waited before acquiring the mutex of the flow table, and the current time is obtained. After acquiring the mutex of the flow table, the flow nodes in the flow table are scanned in turn. During the process of scanning each flow node, the time interval between the latest update time of the flow and the current time is calculated. If the time interval is greater than 16s, the flow is considered as no longer active, ending the flow sampling. At this time, using the five-tuple information of the flow, the flow arrival time and Recover function, which will be discussed later in this paper, to recover the number of packets, data blocks, and number of ACK, SYN, FIN, and RST packets of the sampled flow in the shared counter tree set. The data blocks of the flow is multiplied by 32 to obtain the flow length, and the flow length is divided by the number of packets to obtain the average segment length of the flow. The five-tuple information, flow arrival time, flow duration, minimum payload length, maximum payload length, and statistical features recovered from the shared counter tree set

Algorithm 4 Record(pkt,root,a,p,r,S,H,F)

Input: pkt Sampled packet
 root Index of the root node of the shared counter tree in the array
 a Counter storage space
 p Number of enhanced counters in the shared counter tree
 r Scale of the enhanced counter vector corresponding to the flow to which pkt belongs
 S A set of r random numbers
 H Main hash function
 F Flow table space
Output: shared counter tree with updated statistical feature values of the flows to which pkt belongs

```
1  function Record(pkt,root,a,p,r,S,H,F)
2    five_tuple ← extract(pkt);  //extract the five-tuple of pkt
3    hash_value ← hash(five_tuple);  //hash the five-tuple
4    f ← locate(hash_value,F);  //locate the flow node in the flow table
5    f_tuple ← extract_flow_tuple(f); //extract the flow labels(five-tuples and flow arrival time) from the flow nodes
6    i ← random()%r;
7    u ← H(f_tuple⊕S[i])%p;  //locate the enhanced counter for this count
8      while u < root do
9        C[u] ← C[u] + 1;
10       if C[u] < pow(2,a-1) then
11         break;
12       else
13         C[u].status_bit ← 1;  //set the status bit of the current counter
14         C[u].num ← 0;  //clear the value of the current counter to 0
15         u ← floor(u/2) + p;  //to count in the parent counter of the current counter; floor is the function of round down
16       end if
17     end while
18     return root
19   end function
```

as well as average segment length are written into the flow record buffer, and the content of flow record buffer is added into the flow record buffer queue. After the relevant features of the current scanned flow are recovered, the flow node is released, and the value recorded by the conflict counter in the main hash table entry is reduced by 1. If the value is not greater than the threshold value t, the organization method of the conflict flow nodes is switched to conflict chain.

If the number of flow records in the flow record buffer queue exceeds n, where n is far smaller than the size of the

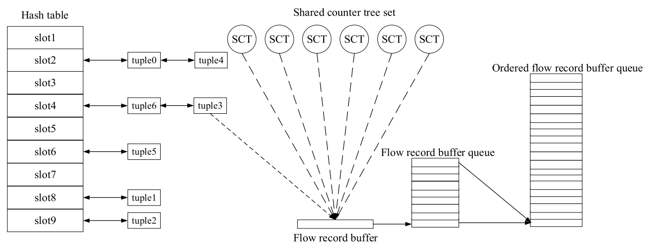

FIGURE 12. Flow features record recovery process.

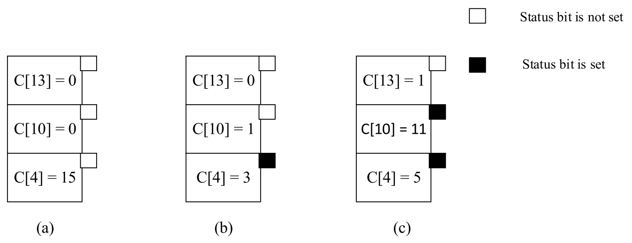

FIGURE 13. Subtree counter.

hash table, a mutex of the ordered flow record buffer queue is obtained, whereby the queue orders the flow records according to the source IP and arrival times. Once all flow records in the flow record buffer queue have been dequeued and added into the ordered flow record buffer queue, the mutex on the ordered flow record buffer queue is released. When the scanning and recovery of all of the flow nodes in the flow table are completed, the mutex of the flow table is released, and the state flag is set to 1. At this point, the flow feature recovery phase ends and sampling is restarted. An example of a flow feature record recovery process with the tuple information of tuple3 is presented in Figure 12.

The concept of the host activity period is used in MiniSamp to replace the concept of the application session. The host activity period is a set of flows with the same source IP. The application types of the flows in the set may differ. The flows in the set are sorted according to arrival time. The time interval between two flows is less than τ seconds. The output thread outputs the flow records from the ordered flow record buffer queue to the sampled flow record file in the units of the host activity period.

To recover certain statistical features of the sampled flows, the concept of the subtree counter is introduced. In Figure 8, the subtree rooted at $C[13]$ contains the nodes $C[13]$, $C[10]$, $C[11]$, $C[4]$, $C[5]$, $C[6]$, and $C[7]$. All nodes in the subtree form a subtree counter. The value of the subtree counter can be calculated as follows, for example, using the value of the subtree counter with $C[13]$ as the root: value(13) $= 2^{2a}C\,[13] + 2^{a}\,(C\,[10] + C\,[11]) + (C\,[4] + C\,[5] + C\,[6] + C[7])$.

Whether a node in the shared counter tree can be used as the root node of its subtree is related to the status bit of this counter node. In Figure 13, the enhanced counter $L[4]$ is used as an example. In (a), because $C[4]$ does not overflow, the status bit is not set to $C[4]$; therefore, $C[4]$ can be the root node of its subtree, there is no need to judge whether the ancestor nodes of $C[4]$ can be the root. Only one leaf node, $C[4]$, exists in this subtree, and the height of the subtree is L = 1. In (b), $C[4]$ overflows, while $C[10]$ does not overflow. The status bit is set to $C[4]$, and $C[10]$ can be the root node of its subtree, there is no need to judge whether the ancestor nodes of $C[4]$ can be the root. Only two leaf nodes, namely $C[4]$ and $C[5]$, and a non-leaf node, $C[10]$, exist in this subtree, and the height of the subtree is L = 2. In (c), both $C[4]$ and $C[10]$ overflow. The status bits are set to $C[4]$ and $C[10]$, and $C[13]$ can be the root node of the subtree. There are four leaf nodes and three non-leaf nodes in the subtree, and the height of the subtree is L = 3.

Chen *et al.* [32] previously derived Equation (5) through statistical principles, and its result is the unbiased estimator of the statistical feature value. In this equation, X_i is the value of i-th subtree counter, n is the total value of the shared counter tree, and k_i refers to the result of basing 2 and powering path value of the i-th subtree counter. The first term in the equation records the number of total packets stored by r subtree counters, and the second term records the noise produced during the counting process.

$$s = \sum_{i=0}^{r-1} X_i - \sum_{i=0}^{r-1} \frac{nk_i}{p} \tag{5}$$

The pseudocode for recovering the statistical features, such as the numbers of data packets, ACK packets, SYN packets, FIN packets, and RST packets from the shared counter tree is presented in Algorithm 5.

To recover the total length of the flow f from the shared counter tree, the Recover function is firstly used to restore the total number of data blocks occupied by flow f, following which Equation (6) is applied to calculate the total length of the flow f.

$$len \leftarrow 32 \times c \tag{6}$$

IV. EXPERIMENTS

A. EXPERIMENTAL TARGET

MiniSamp and RelSamp were compared from the perspectives of sampling flow application recognition accuracy and flow table storage space consumption, thereby verifying that MiniSamp effectively reduces the storage space required for the flow table during sampling and can ensure that the sampled traffic contains sufficient application characteristics to support application classification. Compared to the flow tracking table organized by the hash list and hash splay, it was verified that MiniSamp can search the flow node at a faster speed.

B. EXPERIMENTAL ENVIRONMENT

The experimental environment is illustrated in Figure 14. The traffic set used in the experiment was obtained from the mirror server. MiniSamp and RelSamp were deployed in the sampling server for sampling of the experimental traffic, while an application identification classifier was deployed

Algorithm 5 Recover(f,root,a,p,r,S,H,F)

```
Input: f Ended sampled flow
       root Index of the root node of the shared counter tree in the array
       a Counter storage space
       p Number of enhanced counters in the shared counter tree
       r Scale of the enhanced counter vector corresponding to the flow f
       S A set of r random numbers
       H Main hash function
       F Flow table space
Output: statistical features of the flow f
1  function Recover(f,root,a,p,r,S,H,F)
2    f_tuple ← extact_flow_tuple(f);   //extract flow labels(five-tuples and flow arrival time) from flow nodes
3    for i := 0 to r-1 do   //locate r enhanced counters of flow f
4      u[i]←H(f_tuple ⊕ S[i])%p;
5    end for
6    for i := 0 to r-1 do
7      X[i] ← 0;
8    end for
9    for i := 0 to r-1 do
10     v[i] ← get_subtree_root(u[i]);   //locate the root node of the i-th subtree of flow f
11     X[i] ← get_subtree_value(v[i]);   //calculate the value of the i-th subtree counter of flow f
12   end for
13   for i := 0 to r-1 do   //calculate the path lengths of the r subtrees of flow f from the root node to the leaf nodes
14     l[i] ← get_path_len(i);
15     k[i] ← pow(2,l[i]-1);
16   end for
17   n ← get_total_packet(root);   //calculate the total value of the shared counter tree
18   X_sum ← 0;
19   for i := 0 to r-1 do   //calculate the sum of the counter values for each subtree of flow f
20     X_sum ← X_sum + X[i];
21   end for
22   noise ← 0;
23   for i := 0 to r-1 do
24     noise ← noise + (n × k[i]/p);   //calculate the noise sum of the subtrees of f
25   end for
26   S ← X_sum – noise;   //recover the statistical features of the flow f
27   return S;
28 end function
```

in the classification server. The operating system version of the mirror server and sampling server was CentOS Linux release 7.3.1611, while the operating system version of the classification server was Windows 10. The language used in the experiment was C++. The packet replay tool Tcpreplay version 4.1.2, deep packet inspection tool Tstat version 3.1.1, and classification software Weka version 3-6-13 were used.

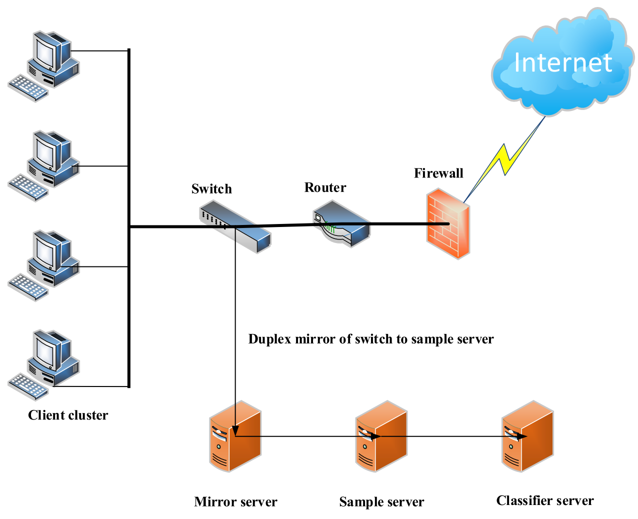

FIGURE 14. Experimental environment.

C. EXPERIMENTAL DATA SOURCE

The traffic captured at the laboratory's LAN gateway was used as the experimental data source. The traffic flowing through the gateway is the traffic of all users' PCs in the LAN accessing to the Internet during a certain time period. The specific means of obtaining the experimental data was implementing a duplex mirror between the sample server and the LAN switch.

The mirror traffic was captured in the mirror server twice to obtain the experimental data for the two experiments, the identification accuracy comparison, and the flow table space cost comparison. A total of 9 GB of traffic was captured the first time, and this traffic set was used as the test traffic set. The second time, a total of 20 GB of traffic was captured, and this traffic set was used as the training traffic set. The deep packet detection tool was used to label the training and test traffic with the application types in the flows.

To obtain the data of the average search time overhead comparison experiment, five sets of 2 GB traffic sets were captured on the mirror server as test traffic sets 1 to 5.

D. EXPERIMENTAL PROCESS

(1) Comparison of sampling flow application recognition accuracy

In this experiment, the SVM classifier and C4.5 decision tree classifier were used to identify the application types of the sampled flow records of RelSamp and MiniSamp, and to compare the two sampling algorithms in terms of application recognition accuracy.

The SVM and C4.5 machine learning algorithms were trained with the sampling flow records of RelSamp as the training data, and the SVM classifier and C4.5 decision tree classifier were obtained. The training steps were as follows:

Step 1: On the sampling server, the traffic replay tool was used to replay the training traffic on the listened network card at a transmission rate of 100 Mbps.

Step 2: RelSamp was used to sample the training traffic. To obtain sufficient flows, the effective sampling ratio p_e was set to 0.15 and the source IP selection probability p_h was set to 0.3. As the sampling flow had to maintain sufficient application characteristics, the flow sampling probability had to be increased and p_f was set to 1.0. The packet sampling probability p_p was calculated by Algorithm 1 as 0.5.

Step 3: The sampled flow record was labeled with the application type of the training traffic according to the five-tuple information in the sampled flow record.

Step 4: The sampled flow record file labeled with the application type was input as training data into the SVM and C4.5 machine learning algorithm for training and obtaining the SVM classifier and C4.5 decision tree classifier.

The experimental steps were as follows:

Step 1: On the sampling server, the traffic replay tool was used to replay the traffic in the test traffic set on the listening network card at a transmission rate of 100 Mbps.

Step 2: MiniSamp was used to sample the traffic on the monitoring network card and the sampled flow record was generated. The effective sampling ratio, source IP sampling probability, stream sampling probability, and packet sampling probability were set to the same values as those when using RelSamp to sample the traffic during the training process. Considering the size of the test traffic set used in the experiment, the size of the shared counter tree that stores the statistical features of the flow in MiniSamp was set to $h = 4$, $r = 100$, $a = 8$, and $p = 2{,}000$.

Step 3: According to the five-tuple information in the MiniSamp sample flow record and application label of the tested traffic, labels were applied to the MiniSamp sampled flow records.

Step 4: The labeled MiniSamp sample flow records were input into the SVM classifier and C4.5 decision tree classifier. The application types of the MiniSamp sample flow records were identified in units of flows, and the application recognition accuracies of the two classifiers for each application were recorded separately.

Step 5: RelSamp was used to sample the traffic on the monitoring network card and the sampled flow record was generated. The process of identifying the application types of the RelSamp sampled flow records was the same as that of MiniSamp. Finally, the accuracies of the two classifiers were recorded for each application.

(2) Flow table space overhead comparison

This experiment tested and compared the amount of flow table storage space required by MiniSamp and RelSamp during the sampling process.

The experimental steps were as follows:

Step 1: In the RelSamp sampling process, a statistics thread was started, the amount of storage space required for the flow table was calculated every 2 min, and the statistics were written to a file; a log was generated thereafter.

Step 2: During the MiniSamp sampling process, a statistics thread was started, the amount of storage space required by the flow table and storage space occupied by the shared counter tree set were counted every 2 min, and the statistics were written to a file, following which logs were generated.

(3) Comparison of average searching time cost

This experiment tested and compared the average time overhead for searching a flow node for a single packet in the hash collision chain flow tracking table (Hash_List), hash splay tree flow tracking table (Hash_Splay), and MiniSamp flow tracking table. The experiment was divided into five groups, with test flow sets 1 to 5 as inputs.

The experimental steps for each group were as follows:

Step 1: On the sample server, the traffic replay tool was used to replay the test traffic set on the monitoring network card at a transmission rate of 100 Mbps. Traffic flow tracking was performed in the flow tracking tables of Hash_list, Hash_splay, and MiniSamp. To reduce the depth of collisions in the hash table entries, the hash table length was set to 1,000. For improved adaptation of MiniSamp to the experimental network environment, the threshold t of MiniSamp was set to 10, the parameter w was set to 4, and the parameter upper was set to 100.

Step 2: A packet counter was present in the flow tracking process, which was incremented each time a packet was obtained from the input buffer circular queue.

Step 3: When the flow node was about to be searched in the flow tracking table, the time stamp recorded at this time was time1; when the target flow node was searched in the flow tracking table and the adjustment to the search tree was completed, or when the target flow node was newly created, the time stamp recorded at this time was time2. The difference between time2 and time1 was the time overhead for searching the flow node in the flow tracking table.

Step 4: The total time cost of searching the flow node for each packet during the experiment was accumulated. When the flow tracking was completed, the total time cost divided by the value of the packet counter was the average time cost of searching the flow node for a single packet in the flow tracking table.

E. EXPERIMENTAL RESULTS AND ANALYSIS

(1) Comparison of sampling flow application recognition accuracy

Table 1 presents the application recognition accuracy of the SVM classifier on the flow records collected by RelSamp and MiniSamp. Table 2 presents the application recognition accuracy rate of the C4.5 classifier for RelSamp and MiniSamp.

It can be observed from Tables 1 and 2 that, compared to the sampling traffic of RelSamp, when the SVM classifier was used to identify the applications of MiniSamp for sampling traffic, in each of the applications identified, the application recognition accuracy was reduced by at least 1.1% and at most 5.3%. The application identification accuracy of the

TABLE 1. Accuracy of classifying sampled flow using SVM.

Application type	RelSamp accuracy	MiniSamp accuracy
WWW	0.826	0.815
MAIL	0.772	0.742
FTP	0.741	0.721
P2P	0.898	0.845
DATABASE	0.698	0.679
MULTIMEDIA	0.805	0.768

TABLE 2. Accuracy of classifying sampled flow using C4.5.

Application type	RelSamp accuracy	MiniSamp accuracy
WWW	0.891	0.874
MAIL	0.930	0.917
FTP	0.821	0.819
P2P	0.910	0.895
DATABASE	0.748	0.739
MULTIMEDIA	0.871	0.852

MiniSamp for sampling traffic when using the C4.5 classifier was reduced by a minimum of 0.2% and a maximum of 1.9%. It verifies that MiniSamp can accurately record and recover the flow features of the sampled traffic using shared counter trees. It can be observed that, compared to the SVM classifier, the C4.5 classifier could identify the application types of the sampled traffic more accurately. Therefore, using MiniSamp to sample the traffic can ensure that sufficiently accurate application features are retained in the sampled traffic, and MiniSamp decreases the accuracy of the sampled traffic's application classification very little compared with RelSamp.

(2) Flow table space overhead comparison

Figure 15 illustrates the changes in the amount of storage space required by the flow table over time when using MiniSamp and RelSamp.

It can be observed from Figure 15 that, during the sampling process, the flow table space required by MiniSamp was always less than that of RelSamp. MiniSamp saved up to 268 kB of flow table storage space and at least 111 kB of flow table storage space compared to RelSamp. Therefore, compared to RelSamp, using MiniSamp to sample the traffic can effectively reduce the storage space required by the flow table during the sampling process.

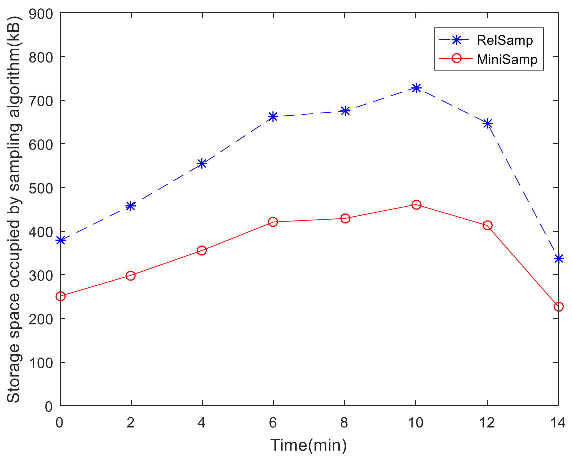

FIGURE 15. Flow table space overhead comparison.

TABLE 3. Flow node average search cost.

	Hash_List	Hash_Splay	MiniSamp
1	0.0903 μs	0.0757 μs	0.0643 μs
2	0.1040 μs	0.0955 μs	0.0788 μs
3	0.1074 μs	0.0761 μs	0.0607 μs
4	0.1036 μs	0.0841 μs	0.0749 μs
5	0.0929 μs	0.0764 μs	0.0683 μs

TABLE 4. Relative search time percentages.

	Hash_List	Hash_Splay	MiniSamp
1	100%	83.8%	71.2%
2	100%	91.8%	75.8%
3	100%	70.9%	56.5%
4	100%	81.2%	72.3%
5	100%	82.2%	73.5%

(3) Comparison of average searching time cost

Table 3 presents the average time required to search the flow node for a single packet in the flow tracking tables of Hash_List, Hash_Splay, and MiniSamp. Table 4 presents the relative percentage of time for searching the flow node for a single packet in the flow tracking table of Hash_Splay and MiniSamp based on Hash_List.

According to the experimental results in Tables 3 and 4, it can be concluded that in most cases, within the same network environment, the time overhead required to search the flow node for a single packet in the MiniSamp flow tracking table was 73.2% of the time required to complete the same operation in Hash_List, and 86.4% of the time required to complete the same operation in Hash_Splay. Therefore, compared to Hash_List and Hash_Splay, MiniSamp requires the lowest average time to search flow nodes, and possesses the highest searching efficiency.

V. CONCLUSION

The memory allocation method of RelSamp may cause a huge waste of storage space on the traffic sampling device and long conflict chains, thereby reducing the searching efficiency of the flow nodes in the high-speed network. Based on storage compression, this paper has proposed a high-performance sampling model known as MiniSamp for application classification detection.

MiniSamp compresses the counter space that records the statistical features of the sampled flow by introducing shared counter trees into the flow table, and it integrates a search tree into the flow table to resolve hash conflicts in the hash table entries. Furthermore, the search tree structure can be adjusted according to the network environment in real time to improve the searching efficiency of the flow nodes.

It was verified that MiniSamp can effectively reduce the storage space required during the sampling process without decreasing the accuracy of the application classification. Moreover, MiniSamp can search the sampled flow nodes at a faster speed than RelSamp in the high-speed network.

Nevertheless, MiniSamp still has a deficiency. In the process of adjusting the longest distance within the search tree and detecting whether the current flow node is consistently hit, two parameters exist, upper and w. If these parameters can be adjusted according to the network environment changes in real time, the structure of the search tree would be adjusted more optimally to locate flow nodes.

To further improve the search efficiency of flow nodes, our future work will focus on how to adjust the parameters upper and w during the sampling process.

REFERENCES

[1] X. Du, Y. Xiao, S. Ci, M. Guizani, and H.-H. Chen, "A routing-driven key management scheme for heterogeneous sensor networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Glasgow, Scotland, Jun. 2007, pp. 3407–3412, doi: 10.1109/ICC.2007.564.

[2] L. Xiao, Y. Li, X. Huang, and X. Du, "Cloud-based Malware detection game for mobile devices with offloading," *IEEE Trans. Mobile Comput.*, vol. 16, no. 10, pp. 2742–2750, Oct. 2017, doi: 10.1109/TMC.2017.2687918.

[3] L. Wu, X. Du, W. Wang, and B. Lin, "An out-of-band authentication scheme for Internet of Things using blockchain technology," in *Proc. IEEE ICNC*, Maui, HI, USA, Mar. 2018, pp. 769–773, doi: 10.1109/ICCNC.2018.8390280.

[4] A. W. Moore and D. Zuev, "Internet traffic classification using Bayesian analysis techniques," *ACM SIGMETRICS Perform. Eval. Rev. ACM*, vol. 33, no. 1, pp. 50–60, 2005, doi: 10.1145/1064212.1064220.

[5] A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *Proc. Int. Workshop Passive Act. Netw. Meas.* Berlin, Germany: Springer, 2005, pp. 41–54, doi: 10.1007/978-3-540-31966-5_4.

[6] A. Dainotti, F. Gargiulo, L. I. Kuncheva, A. Pescape, and C. Sansone, "Identification of traffic flows hiding behind TCP port 80," in *Proc. IEEE Int. Conf. Commun.*, May 2010, pp. 1–6, doi: 10.1109/ICC.2010.5502266.

[7] L. Ye, *Research on Network Traffic Identification Technology Based on DPI* Jining, China: Qufu Normal Univ., 2017.

[8] S. Sen, O. Spats, and D. Wang, "Accurate, scalable in-network identification of p2p traffic using application signatures," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 512–521, doi: 10.1145/988672.988742.

[9] J. Cao, Z. Fang, G. Qu, H. Sun, and D. Zhang, "An accurate traffic classification model based on support vector machines," *Int. J. Netw. Manage.*, vol. 27, no. 1, Jan. 2017, Art. no. e1962, doi: 10.1002/nem.1962.

[10] D. Wu, X. Chen, and C. Chen, "On addressing the imbalance problem: A correlated KNN approach for network traffic classification," in *Proc. Int. Conf. Netw. Syst. Secur.* Cham, Switzerland: Springer, 2015, pp. 138–151, doi: 10.1007/978-3-319-11698-3_11.

[11] L. Chuangfeng, L. Yunlong, and S. Wei, "Android malicious application detection algorithm based on CNN and naive Bayes method," *Inf. Secur. Res.*, vol. 5, no. 6, pp. 470–476, 2019.

[12] H. Cunge and Y. Qiusun, "Research and improvement of C4.5 algorithm in decision tree classification algorithm," *Comput. Syst. Appl.*, vol. 28, no. 6, pp. 198–202, 2019.

[13] J. Ma and L. Wu, "An improved accelerated K means clustering algorithm," *J. THz Sci. Electron. Inf.*, vol. 17, no. 5, pp. 885–891, 897, 2019, doi: 10.11805/TKYDA201905.0885.

[14] R. Thupae, B. Isong, N. Gasela, and A. M. Abu-Mahfouz, "Machine learning techniques for traffic identification and classifiacation in SDWSN: A survey," in *Proc. IECON 44th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2018, pp. 4645–4650, doi: 10.1109/IECON.2018.8591178.

[15] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: Why and how you should (Still) use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, pp. 1–21, Jul. 2017, doi: 10.1145/3068335.

[16] S. Dong and R. Jain, "Flow online identification method for the encrypted Skype," *J. Netw. Comput. Appl.*, vol. 132, pp. 75–85, Apr. 2019, doi: 10.1016/j.jnca.2019.01.007.

[17] X. Wang, Y. Zhang, W. Zhang, X. Lin, and Z. Huang, "Skype: Top-k spatial-keyword publish/subscribe over sliding window," *Proc. VLDB Endowment*, vol. 9, no. 7, pp. 588–599, Mar. 2016, doi: 10.14778/2904483.2904490.

[18] Y. Hua, *Analysis and Classification of Network Traffic Based on NetFlow*. Zhengzhou, China: Zhengzhou Univ., 2018.

[19] L. Jun and Z. Shunyi, "Peer-to-peer traffic identification using Bayesian networks," *J. Appl. Sci.*, vol. 27, no. 2, pp. 124–130, 2005.

[20] V. R. Tabar, F. Eskandari, S. Salimi, and H. Zareifard, "Finding a set of candidate parents using dependency criterion for the k2 algorithm," *Pattern Recognit. Lett.*, vol. 111, pp. 23–29, Aug. 2018, doi: 10.1016/j.patrec.2018.04.019.

[21] A. Meehan and C. de Campos, "Averaged extended tree augmented naive classifier," *Entropy*, vol. 17, no. 12, pp. 5085–5100, Jul. 2015, doi: 10.3390/e17075085.

[22] P. Xu, Q. Liu, and S. Lin, "Internet traffic classification using support vector machine," *J. Comput. Res. Develop.*, vol. 46, no. 3, pp. 407–414, 2009.

[23] H. Duan, Q. Zhang, and M. Zhang, "FCBF algorithm based on normalized mutual information for feature selection," *J. Huazhong Univ. Sci. Technol. (Natural Sci. Edition)*, vol. 45, no. 1, pp. 52–56, 2017, doi: 10.13245/j.hust.170110.

[24] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, p. 245, Oct. 2004, doi: 10.1145/1030194.1015495.

[25] P. Dong, X. Du, and H. Zhang, "A detection method for a novel DDoS attack against SDN controllers by vast new low-traffic flows," in *Proc. IEEE Int. Conf. Commun.*, May 2016, pp. 1–6, doi: 10.1109/ICC.2016.7510992.

[26] R. R. Kompella and C. Estan, "The power of slicing in Internet flow measurement," in *Proc. 5th ACM SIGCOMM Conf. Internet Meas., USENIX Assoc.*, Oct. 2005, pp. 105–118.

[27] V. Sekar, M. K. Reiter, and W. Willinger, "cSamp: A system for network-wide flow monitoring," in *Proc. 5th USENIX Symp. Netw. Syst. Des. Implement.*, vol. 8, pp. 233–246, Apr. 2008.

[28] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: Towards programmable network measurement," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 115–128, Feb. 2011, doi: 10.1109/TNET.2010.2066987.

[29] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast monitoring of traffic subpopulations," in *Proc. 8th ACM SIGCOMM Conf. Internet Meas. Conf. IMC*, 2008, pp. 257–270, doi: 10.1145/1452520.1452551.
[30] M. Lee, M. Hajjat, R. R. Kompella, and S. G. Rao, "A flow measurement architecture to preserve application structure," *Comput. Netw.*, vol. 77, pp. 181–195, Feb. 2015, doi: 10.1016/j.comnet.2014.11.005.
[31] J. Mikians, A. Dhamdhere, and C. Dovrolis, "Towards a statistical characterization of the interdomain traffic matrix," in *Proc. Int. Conf. Res. Netw.* Berlin, Germany: Springer, 2012, pp. 111–123, doi: 10.1007/978-3-642-30054-7_9.
[32] M. Chen, S. Chen, and Z. Cai, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1249–1262, Apr. 2017, doi: 10.1109/TNET.2016.2621159.
[33] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, "Internet traffic classification demystified: Myths, caveats, and the best practices," in *Proc. ACM CoNEXT Conf. CONEXT*, 2008, pp. 1–12, doi: 10.1145/1544012.1544023.
[34] W. Li, M. Canini, A. W. Moore, and R. Bolla, "Efficient application identification and the temporal and spatial stability of classification schema," *Comput. Netw.*, vol. 53, no. 6, pp. 790–809, Apr. 2009, doi: 10.1016/j.comnet.2008.11.016.

SHICHANG XUAN received the B.S., M.S., and Ph.D. degrees in computer science and technology from Harbin Engineering University, in 2007, 2010, and 2017. He is currently an Associate Professor with Harbin Engineering University. His main research interests include information security and blockchain.

DEZHI TANG received the bachelor's degree in computer science and technology from Shenyang Aerospace University, in 2016. He is currently pursuing the master's degree with Harbin Engineering University. His main research interests include social networks and information security.

ILYONG CHUNG received the M.S. and Ph.D. degrees in computer science from The City University of New York, in 1987 and 1991, respectively. Since 1994, he has been a Professor with the Department of Computer Engineering, Chosun University, Gwangju, South Korea. His research interests are in computer networking, security systems, and coding theory.

YOUNGJU CHO received the master's and Ph.D. degrees in electronic calculation from Chosun University, specializing in information technologies, education, and mobile ad hoc networks. She is currently the SW Education Research Professor with the SW Convergence Education Institute, Chosun University. Her interests include network security, the Internet of Things, information protection, mobile ad hoc networks, software education, AI, VR, and AR.

XIAOJIANG DU (Fellow, IEEE) received the B.S. and M.S. degrees in electrical engineering from Tsinghua University, China, in 1996 and 1998, respectively, and the Ph.D. degree in electrical engineering from the University of Maryland at College Park, in 2003. He is currently a Professor with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA. His research interests include security, wireless networks, and systems. He has authored over 400 journal articles and conference papers in these areas. He is a Life Member of ACM.

WU YANG received the Ph.D. degree in computer system architecture specialty from the Computer Science and Technology School, Harbin Institute of Technology. He is currently a Professor and the Ph.D. Supervisor with Harbin Engineering University. His main research interests include wireless sensor networks, peer-to-peer networks, and information security. He is a member of ACM and a Senior Member of CCF.