STAFF: Automated Signature Generation for Fine-Grained Function Traffic Identification

Yazhe Tang[1], Xun Li[1] and Lishui Chen[2]

[1]*Xi'an Jiaotong University, Shaanxi, China*
[2]*The 54th Research Institute of China Electronics Technology Group Corporation, Shijiazhuang, China*
E-mail: yztang@mail.xjtu.edu.cn; lixun2011@stu. xjtu.edu.cn; 15031150937@139.com

Received 01 May 2018; Accepted 01 November 2018
Publication 26 November 2018

Abstract

Identifying a user operating application function can reflect the user behavior, or even can help to improve the user experience. It is the focus of the real application in big data analytics technology. Unlike Coarse-grained Traffic Identification (CTI) which only identify application/protocol that a packet is related to, Fine-grained Function Traffic Identification (FFTI) maps the traffic packet to a meaningful user operation or an application function. In this paper, our focus is to identify the fine-grained function signature. We propose an automatic and stable signature generation method, so-called STAFF, to identify different application functions. STAFF treats data packets as long strings. The aim of our method is to find all the string fragments whose length is longer than a prescribed length and whose occurrence is higher than a prescribed frequency. The final signature will be presented as pairs of string fragments and their corresponding occurrence frequency. The experimental results show that STAFF can automatically generate fine-grained function signatures in different applications with average 93.65% identification accuracy and the method is noise insensitive.

Journal of Web Engineering, Vol. 17_5, 71–94.
doi: 10.13052/jwe1540-9589.1753

Keywords: automated signature generation, application function signature, application traffic identification.

1 Introduction

How can we follow the network traffic to learn what function in an application a user is operating? Or, how can we know the detected data traffic belongs to which application? We care about these issues because they are related to the user experience. In this paper we focus precisely on this problem – specifically, how can we derive the signatures of different functions out of an application traffic to tell them apart?

Identifying fine-grained application traffic can be used to analyze popular application operations, measure and monitor the network in a deep level, or even improve the user experience. The traditional traffic identification, called *Coarse-grained Traffic Identification* (CTI) [1–4], reports only granularity information at the protocol or application level, *e.g.*, the traffic belongs to Twitter or QQ, but it fails to provide *Fine-grained Function Traffic Identification* (FFTI), i.e., information related to which specific function such as login, data input or output the user is operating. With FFTI, we are able to know the communication details, to evaluate the performance and to make fault discovery and recovery at the function granularity level. So the key problem for FFTI is how to get fine-grained function signature (*FFS*) for every application function.

Signature-based identification has been proposed by previous researchers in different areas. A signature is a portion of payload data that is static and can be described as a sequence of strings. Signatures could be manually extracted by the experts. Such manual process is tedious and time consuming. The quality of signatures also varies due to the level of expertise. Thus, we need a systematic approach to define and extract the signature. Although several algorithms, such as LASER [15] and Behavior Signature [18] can automatically extract protocol format for a data traffic, they are not focus on FFTI.

A new method called *Signature for Traffic of Application Fine-grained Function* (*STAFF*), is proposed in this paper. As shown in Figure 1, a specific function flow in one application is composed of a set of data packets. In STAFF, this data packet set is seen as strings and the signature generation is mapped into the problem of obtaining the frequently occurring substrings and their corresponding occurrence frequency. The idea is based on two observations: first, some portion of the data payload in a functions is invariant

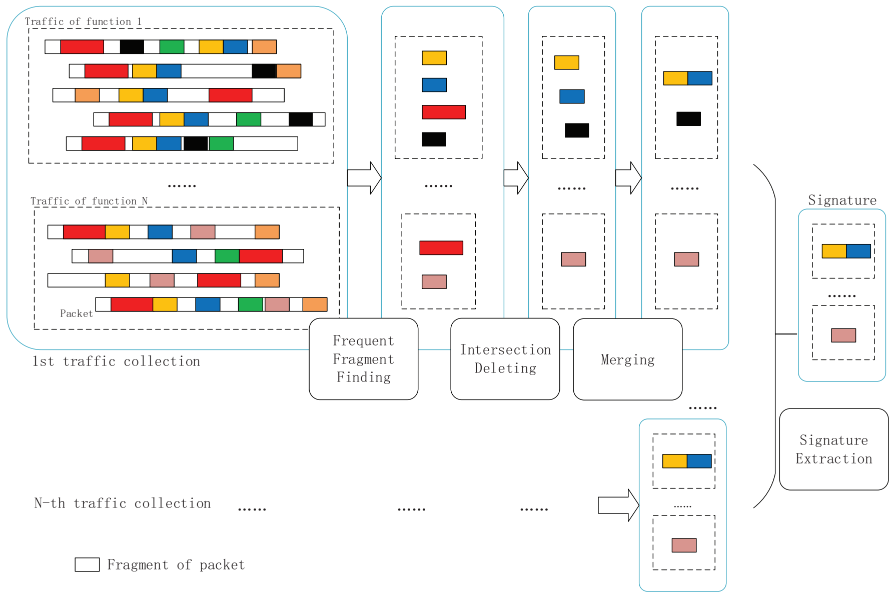


Figure 1 STAFF algorithm overview.

and the signature reflecting the *application function* is atypical in Internet traffic. Second, some portion of the payload are expected to occur much more frequently than random noise. So, the format of signature output from STAFF could be presented by a pair of substring and its occurrence frequency in the form of {(*substring1*, *frequency1*), (*substring2*, *frequency2*) ...}. At the matching stage, when *substring* and its occurrence *frequency* are satisfied at the same time, we consider the token is triggered. To our knowledge, little research has attempted to automatically generate signatures for functions of Internet applications.

Thus, STAFF has the following main advantages:

- **Fine-grained:** We can provide *FFS* to identify different functions in the same application.
- **Noise insensitive:** The condition that input traffic is completely from the same application function is not required.

The remainder of the paper is organized as follows. Section 2 describes the related work in the traffic classification and signature generation area. Section 3 introduces the overview of STAFF and describes the details of the STAFF. We present the experimental results in Section 4. Finally, Section 5 makes a conclusion of the paper and discusses the future work.

2 Related Work

Signature-based identification is not new to the DNA area. However, their focus is to find some sub-sequence [5–7] or PTR (perfect tandem repeat) [8–11] until the support does not go below a certain threshold. Meanwhile, we focus on how to find frequent substring that satisfies the condition, not sub-sequence. Moreover, signature identification can also be used for Internet worm research [12–15]. There are several differences between worm identification with FFTI. In worm area, their focus is limited to identifying the security threatening traffic only. But in FFTI, we focus on the fine-grained application function identification among innocuous traffic which widens the target traffic to be identified. Signatures in FFTI are usually short and some are only several bytes, while the signatures in worms are usually as long as hundreds bytes. Different application function flowing in the same application should have different signatures. So, some methods [12, 13] in worm area are ineffective in FFTI. Furthermore, the problem about unknown protocols identification [16–18] sounds similar to FFTI, but it is coarse-grained and noise sensitive. So the method of unknown protocols identification does not work well on FFTI. We categorize previous methods as follows.

2.1 DNA Domain

DNA sequence patterns can be seen as signatures of specific DNA sequence. In this area [5–7], they denote an *itemset i* by $(i_1 i_2 \ldots i_m)$, where i_j is an item and denote a *sequence s* by $(s_1 s_2 \ldots s_n)$, where s_j is an *itemset*. A *sequence* $\langle a_1 a_2 \ldots a_n \rangle$ is contained in another *sequence* $\langle b_1 b_2 \ldots b_m \rangle$ if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_1 \subseteq b_{i_1},\ a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$. They use the symbol "$\prec$" to denote the "is contained in" relationship. For example, the sequence $\langle (3)\ (4\ 5)\ (8) \rangle \prec \langle (7)\ (3\ 8)\ (9)\ (4\ 5\ 6)\ (8) \rangle$, since $(3) \subseteq (3{\cdot}8)$, $(4{\cdot}5) \subseteq (4{\cdot}5{\cdot}6)$ and $(8) \subseteq (8)$. However, the sequence $\langle (3)\ (5) \rangle$ is not contained in $\langle (3\ 5) \rangle$ and vice versa. In a set of sequences, a sequence *s* is *maximal* if *s* is not contained in any other sequence. So, the problem in this area can be described as: given a DNA database, the problem of mining sequential patterns is to find the maximal sequences among all sequences that have a certain user-specified minimum support. Each such maximal sequence represents a sequential pattern. The search strategy of these methods is that starts from short patterns and then extends them until the support does not go below a certain threshold. The biggest difference between our method and those methods is that we focus on strings not item. For example, we have a database: "*abc*", "*xybac*" and "*xabc*". Minimum support is set as *2*. Our method regards the database as a long string. Frequent substring "*abc*" (2 times) and "*cx*" (2 times) will be found. DNA methods will get the maximal sequence "*ac*" (3 times). Moreover, the DNA methods do not work for FFS, because the DNA sequences are a linear arrangement of four nucleotides, A (adenine), G (guanine), C (cytosine) and T (thymine), not random disorganized strings.

2.2 Worm Research

Signature-based identification has been studied in Internet worm detection area. EarlyBird system [12] generates overlapping fixed-length content blocks over each byte offset for each sample flow, and extracts the content blocks appearing many times as the whole signature. Since common substrings in application protocols are usually very short, the method cannot be directly used in fine-grained function signature extraction. Autograph system [13] divides payload of flows into variable-length content blocks using Content-based Payload Partitioning Schema. The number of generated strings is reduced, but short common strings can be missed using the partition schema. Redundant and short content blocks will increase the false positive when they are used directly as signatures. In worm area, their focus is limited to identifying the security threatening traffic only. But in FFTI, we focus on the fine-grained application function identification among innocuous traffic which widens the target traffic to be identified. Different *function flows* in the same application should have different signatures. This fact separates our work from the previous worm signature generation research.

2.3 Unknown Protocols Identification

Most of the proposed methods that generate signatures automatically stay focus on detection of application's unknown protocol format. Little attention was paid to FFS. There are many methods to identify network traffic using flow statistical information [19–21]. It is well know that the statistical characteristics used in identification are not stable since delay and packet loss ratio of the network are dynamic and not all the flows of a specific application have obvious and special traffic characteristics. In addition, flows belonging to different applications could have similar per-flow statistics. It is hard to distinguish these *function flows* in the same application by only using flow properties.

Byung-Chul et al. [15] proposed the LASER algorithm which tries to find the longest common subsequence among samples. The algorithm is similar to Polygraph system [14]. The longest common subsequence is extracted from sample flows and is treated as the signature of the given application. The algorithm compares two samples to get the longest common subsequence between them, and then compares it with other samples iteratively to refine it. This method is noise sensitive in the samples and the comparing order. Moreover, there is a big challenge in generating signature if different common substrings exist in the given application. So, it is hard to distinguish application functions. Yoon et al. [18] proposed behavior signature as a unique traffic behavior pattern appearing in the first few packets of plural traffic flows when a specific function is conducted by an application with a combination of various optional traffic features. However, if a single function makes a single flow, the behavior signature method cannot be applied. Moreover, this method has difficulty in distinguishing applications generated in a single host due to the assumption that the traffic on a single host is generated by a particular application. In our method, in order to avoid the impact of other applications, we run the target application for many times in different operating environment and collect target *function flow* many times. The final signature is produced by the intersection information of all the signatures generated from all these multiple results.

In brief, although the existing methods have shown a reliable performance in their specific application areas, the limitations that mentioned above prevent such methods from using in the FFTI area.

3 Signatures Generation Using Staff

This paper proposes a new method called STAFF. Multiple traffic packets will be generated when a user performs a specific *application function*. For example, several data packets that are involved in authorization and update are generated in the login phase. To put everything in context, let us first define some basic concepts. As shown in Figure 2, *application function* indicates a single operation that a user can click in the application, such as a login, data input, file transfer, and so on. Then *function flow* indicates network traffic that is triggered by using one function of the application. It is always composed of a set of data packets that share the identical source IP, destination IP, source port, destination port and protocol.

The aim of STAFF is to find all the fragments whose length is longer than a given length and whose occurrence is higher than a given frequency. These fragments along with their occurrence number will be taken as an FFS candidate in the *function flow*. After that, we extract FFS from all the FFS' candidate list. In other words, the problem that we want to solve can be abstracted into

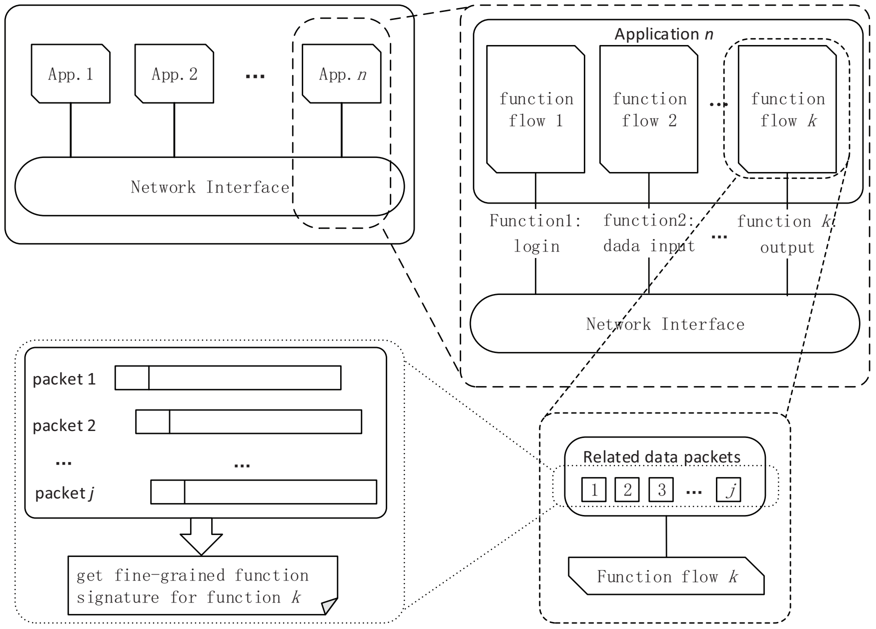


Figure 2 Fine-grained function signature overview.

the following model. We have a stream $\sigma = \langle s_1, \ldots, s_n \rangle$, with each s_n is a character. Fragment $\sigma' = \langle s_i, \ldots, s_j \rangle$, with $i, j \in (1, \ldots, n)$ and $i < j$. $F\text{Size} = j - i + 1$. In addition, $F\text{Size}$ is defined as size of fragment σ'. FN is defined as how many times fragment σ' appears in stream σ. Our task can be formally described as: If $\exists\, \sigma'$: $F\text{Size} \geq f\_size$ and $FN \geq min\_st$, then insert key-value pair (σ', FN) into the candidate FFS, where f_size (fragment size) and min_st (minimum support threshold) are the predefined thresholds. Otherwise, output "NULL". The final FFS can be extracted from all the FFS's candidates.

Sequential pattern discovery consists in finding patterns in sequences, where each element is directly followed by another. But in our case, STAFF regards all data packets of one *function flow* as strings. We search for frequent fragment in strings. A fragment is frequent if the fragment is present at least min_st times in a set of data packets, where min_st is a predetermined threshold. As shown in Figure 1, STAFF has four steps namely frequent fragment finding, intersection deleting, merging and signature extraction. Main algorithm of STAFF acquires all the FFS for every *function flow*. In a first step, the algorithm establishes a candidate list for the frequent fragment set (***Step 1***). Secondly, deleting intersection will be executed for reducing calculation and improving identification accuracy (***Step 2***). In a third step, fragment merging is done to limit the number of results without any loss of information (***Step 3***). In the last step, FFS are extracted from the candidate list for the frequent fragment set (***Step 4***).

To illustrate the method, a simple example is used here. Assume that we have a function flow as follows:

- $Pack_1$*: 'xian39%duxjtustu420lan'*
- $Pack_2$*: 'xingqingxian&xjtustu98420http'*
- $Pack_3$*: 'xian123xjtustu'*
- $Pack_4$*: 'xian2356xjtustu#@420hao'*
- $Pack_5$*: 'xiantangxjtustulee420li'*

In the test, the substring length is set to 3, the minimum string occurrence number is set to 4 and merging criterion is set to 0.5, i.e., f_size=3, min_st=4 and mt=0.5. After frequent fragment finding, we will get a set of frequent fragment candidate {(xia;5), (ian;5), (xjt;5), (jtu;5), (tus;5), (ust;5), (stu;5),(420;4)}. In the second step, the candidate set for the other functions could also have the same items such as (xia; 5), (ian; 5). So, we should remove this intersection list from all frequent fragment sets. Item (xjtustu; 5) will be merged after phase merging. At last, FFS of function1, {(xjtustu; 5), (420; 4)}, will be extracted. As shown in Figure 1, the red rectangular block represents 'xian'. Black one is '420'. Yellow one denotes 'xjtu' and blue one for 'stu'. We will delete the red rectangular block, because other function flows also have this fragment. Moreover, the yellow block and the blue block can be merged. After that we get the signature of the Function1. Such process is repeated for different runs and the final FFS can be extracted.

The following subsections present the details of STAFF.

Main Algorithm

input: *SS*: a Set of *function flow* set *s*. *f_size*: fragment size.
min_st: minimum support threshold.
output: *FFS*: Fine-grained Function Signature.

```
1 begin
2     foreach s in SS do
3         for i=1 to |s| do /*s: function flow set. |s|: total number
                                of application function */
```

```
4              candidate frequent fragment [$i$]←FrequentFragment
                 Finding ($s[i], f\_size$, $min\_s$t) /* algorithm1 */
5          end
6          selected frequent fragment set←IntersectionDeleting
              (candidate frequent fragment) /* algorithm2 */
7        candidateSignature←Merging(selected frequent fragment set,
             $mt$) /* algorithm3*/
8      end
9      $FFS$←SignatureExtraction(candidateSignature) /* algorithm4 */
10     return $FFS$
11 end
```

3.1 Frequent Fragment Finding (Step 1)

Motivated by calculating similarity of texts, each byte sequence can be broken down into overlapping fixed-size fragments (line 5). Define *f_size - fragment* be any substring of length *f_size* found within the byte sequence. For instance, suppose the byte sequence is the string "xjtuabc", and we pick *f_size* =2, the set of 2-fragment will be {xj,jt,tu,ua,ab,bc}. Then the number of occurrence times of each string fragment is counted and only the fragments whose occurrence number is bigger than min_st (line 7) are marked down as signature candidates. With the set of candidates, we are certain that all the signatures candidates are frequent strings. In order to reduce the calculation cost and improve the identification accuracy, the deleting intersection process is created in **Step 2**.

Step 1

```
1 FrequentFragmentFinding ($s, f\_size, min\_s$t)
  input : $s$: function flow set. $f\_size$: fragment size. $min\_st$: minimum
          support threshold.
  output : FFMap: Frequent Fragment Map.
2 begin
    /* count frequency($\geq min\_s$t) of fragment($len(fragment) \geq f\_size$)
    of each function flow */
3   foreach data packet $d$ of $s[i]$ do
4       for ($offset$=0; offset$\leq len(d)$- $f\_size$; $offset$++)
```

(*Continued*)

Step 1 Continued

```
5             fragmentSet←d [offset, offset+f_size-1]
6         end
7         if frequency of item in fragmentSet≥min_st then
8             FFMap.insert (item, frequency);
9         end
10    end
11    return FFMap
12 end
```

3.2 Intersection Deletion (Step 2)

The purpose of intersection deletion is to reduce the calculation cost and improve the uniqueness of characteristics of each function. Removing the intersections (line 6) between two parts will not only reduce the calculation during subsequent steps, but also keep the frequent fragments as the unique characteristics of *function flow*. In this step, the intersection is checked among all the signature candidates, and if any intersection is found, the same entry will be deleted from the candidate list of both function flows, no matter whether their *occurrence frequencies* are equal or not (line 5).

Step 2

```
1 IntersectionDeleting (FFSet)
  input: FFSet: Frequent Fragment Set, composed of FFMap
  output: SFFSet: Selected Frequent Fragment Set
2 begin
3     foreach FFMap in FFSet do
4         foreach FFMap.key in FFMap do
5             if FFMap.key in anyother FFMap then
6                 delete FFMap.key from all FFMap;
7             end
8         end
9     end
10    return SFFSet←FFMap
11 end
```

Suppose, after the first step, we get the sets for all function flows as FS_1, FS_2;…; FS_n. Each FS can be described as:

$$FS_i = \{\textit{fragment 1, frequency 1;\ldots; fragment n, frequency n}\}.$$

For example, here we have 4 frequent fragment sets:

- *FS_0:{ HTTP1.1,8; server,9; 0x00,24; t%3B%,17}*
- *FS_1:{ HTTP1.1,13; 0x02000,11; B7A3F8A2,16; t%3B%,17}*
- *FS_2:{ HTTP1.1,8; server,13; XX1pZ,17; jS5gH,16; D=DtH,6; AC4DF,5}*
- *FS_3:{HTTP1.1,8; server,9; 0x02,14; JSESS,4; XX1pZ,17}*

After intersection deleting, the final selected frequent fragment set are:

- *FS_0:{ 0x00,24}*
- *FS_1:{0x02000,11; B7A3F8A2,16}*
- *FS_2:{ jS5gH,16; D=DtH,6; AC4DF,5}*
- *FS_3:{ 0x02,14; JSESS,4 }*

3.3 Merging (Step 3)

Although intersect signature are deleted in **Step 2**, some frequent fragments still are redundant. In this step, fragments are merged into longer frequent fragments if possible. **Step 3** searches the longest frequent fragment with the Lemma 1 [22]: "If an episode α is frequent in an event sequence *s*, then all sub episodes $\beta \leq \alpha$ are frequent", where an episode is a pattern. Therefore, with longest frequent fragment, we limit the number of results without any loss of information. For instance, suppose there is a frequent string 'xjtu_homepage' in the packet payload. When *f_size*=4, 10 frequent fragments will be found: 'xjtu', 'jtu_', 'tu_h', 'u_ho', '_hom', 'home', 'omep', 'mepa', 'epag' and 'page'. Obviously the fragment we need is 'xjtu_homepage' instead of the other 10 fragments. So, these redundant frequent fragments can be further merged into longer fragments.

There are two conditions for merging (line 5). First, two fragments should be adjacently overlapped, i.e., One fragment contains the other or they are adjacent and have overlapping part. Secondly, merge score should be less than or equal to a predefined threshold. Merge score can be defined as MS (line 5).

$$MS = \frac{count(\text{fragment AB})}{count\ (\text{fragment A}) + count(\text{fragment B})}$$

Fragment A and B can be merged if $0 < MS \leq 0.5$ (line 5). When two frequent fragments merged into a new one, this new one is saved and the two original frequent fragments are removed. The merging process is iterated until no more frequent fragments can be further merged.

Step 3

```
1 Merging(SFFSett)
  input: SFFSet: Selected Frequent Fragment Set.
  output: CSSet: Candidate Signature Set
2 begin
3     foreach FFMap in SFFSet do
4         do select two FFMap.key: key1, key2
5             if (key1, key2 is adjacent or overlapped) &&
                  $0<(\text{cout}(key1 \text{ merge } key2) / [\text{cout}(key1)+\text{cout}(key2)] \leq 0.5)$
                  then
                  /* cout(x): calculates the frequency of x appears in
                     function flow */
6                 FFMap.delete (key1)
7                 FFMap.delete (key2)
8                 FFMap.insert (key1merge key2, cout(key1merge key2))
9             else
10                if $key1 \subset key2$ then
11                    FFMap.delete (key1)
12                if $key2 \subset key1$ then
13                    FFMap.delete (key2)
14        until any two FFMap.key cannot merging.
15    end
16    return $CSSet \leftarrow FFMap$
17 end
```

3.4 Signature Extraction (Step 4)

When we collect signature from *function flows*, it is inevitable that there will be some noises. Here noise refers to the packets that do not belong to the target *function flow*. In order to avoid the impact of noise, we run the target application many times in different operating environments and collect the target *function flow* many times. Further, the final FFS are extracted from the intersection of multiple FFSs which generated from different *function flows*. For each intersected entry, the least frequency number among all records for that entry will be used. For example, if two frequent fragment sets have key-value pair *(HTTP, 8)* and *(HTTP, 6)*, the result intersection key-value pair should be *(HTTP, 6)*. The extraction process is iterated until the final result doesn't change or the maximum number of iterations is reached (line 5–9).

For instance, we get three FFS sets as below:

FFS -1:{*HTTP1.1,8; server,9; 0x02,14; B7A3F8A2,16; t%3B%, 17*}
FFS -2:{*HTTP1.1,7; server,9; 0x02,14; C7A8K3C7,14; t%3B%, 17*}
FFS -3:{*HTTP1.1,8; server,7; 0x02,14; 37B2B7C6,7; t%3B%, 17*}

Ultimately, the signature is: {*HTTP1.1,7; server,7; 0x02,14; t%3B%, 17*}

Step 4

```
1 SignatureExtraction(CandidateSignature)
  input: CandidateSignature: Candidate Signature composed of CSSet.
  output: FFS: Fine-grained Function Signature.
2 begin
3    for j=1 to |s| do /* s: function flow set. |s|: total number of
         application function */
4       i=1;
5       while (i≤| CandidateSignature |) do
           /*|CandidateSignature |equals the times of collecting
           function flow */
6          if i=1 then
7                new_FFS[i]= intersect(CandidateSignature[1][j],
               CandidateSignature[2][j]);
             /* intersect((key1;value1),(key2;value2)): calculates the
                intersection. When key is same and corresponding
                value is different between two items, the item that
                value is less will be saved. */
8          else
```

(*Continued*)

Step 4 Continued

```
9              new_FFS[i]= intersect(CandidateSignature[i+1][j],
                 new_FFS[i-1])
10           i++;
11     end
12  end
13  return FFS
14 end
```

4 Experiments

We design experiments to answer the following question: **Q1. Parameter selection:** How to select the reasonable parameter value? **Q2. Fine-grained:** Does STAFF accurately distinguish fine-grained function traffic? **Q3. Noise insensitivity:** What happen if there is noise traffic in the data traffic?

The STAFF prototype is implemented in Python. The experiment platform is a PC with 3.1GHz Intel Core i5-2400 2 CPU and 4GB memory. Since there is no common set of test data for evaluation, we have chosen six popular Chinese Internet applications and service websites as the target applications as shown in Table 1. The applications cover a wide range of different applications.

Table 1 Traffic dataset in our experiments

Internet Applications	Number of Selected Functions	Notes
Weibo	32	a popular micro blog
Selected financial system	21	financial system
BBS	43	BBS of Xi'an Jiaotong University
TouTiao news	22	news
BT	24	an P2P file transfer website of Beijing University of Posts
Iqiyi	44	a popular video site

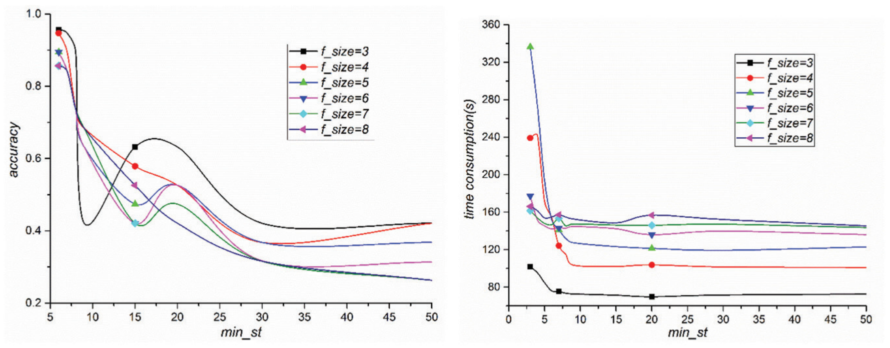
4.1 Effect of Parameter Selection

STAFF has two important parameters. They are fragment size *f_size* and minimum support threshold *min_st*. These two parameters affect the results in different ways. So, it is important to select a suitable value for *f_size* and *min_st*.

The experiment trace is captured from an online financial system. The trace capture is done 4 times with different user name and different browsers (Chrome and Mozilla). The trace consists of 21 functions flows of a total of about 7.47 MB. Each function flow consists of about 900 packets. The typical function include OpenHomePage, Login, PrintReceipt, CompletedReceiptQuery, CoProcessing, CompletedVoucherQuery, CentralizedApproval, UnfinishedVoucherQuery, UnfinishedReceiptQuery, AllReceiptQuery, NewReceipt, AllVoucherQuery, etc.

We evaluated the time consumption and identification accuracy of different modules of STAFF. We select value of *f_size* from 3 to 8, because the average number of letters for an English word is 7.6, and 80% are between 4 and 10 letters long [23]. We also choose *min_st* from set (3, 6, 7, 8, 9, 10, 15, 20, 25, 50). The results are shown in Figure 3. It can be seen that the smaller *f_size* is, the less time will be consumed. This trend is count-institutive. We might have thought the shorter the fragment, the more iterations and thus, the more time in the simulation. When taking a smaller value for *f_size*, the chance of the fragment presenting in different flows will increase. Thus, the fragment has more chance to be deleted in the delete-intersection phase. As a result, although the number of iterations has been increased, due to the amount decrease in fragments the total simulation time is actually decreased. From Figure 3(a), the smaller *f_size* value is, the higher the detection accuracy is. But, when *f_size* = 3, the accuracy is unstable. As a trade-off, four seems to be an optimal number for *f_size* and that value is used in the subsequent experiments. From Figure 3(b) it can be seen that, when *min_st* value is greater than 10, that time consumption is constantly increasing as *f_size* increases while identification accuracy is actually decreasing as shown in Figure 3(a). As shown in Figure 3, the bigger value *min_st* takes in the range of 5 to 8, the more expensive the computation cost is and the more accurate the result is. As a trade-off, a value of 6 for *min_st* is used for the subsequent experiments.

In fact, we can pick *f_size* and *min_st* to be any values we like. However, if we pick value of *f_size* too small, we will find the fragments appear in almost all *function flows*. If we pick value of *f_size* too great, it is likely that we cannot get frequent fragments. Through above experimental analysis, we suggest that setting *4* for *f_size* and *6* for *min_st* is reasonable for the later experiments.



(a) (b)

Figure 3 Time cost and accuracy trends. (a) Identification accuracy with different parameters. (b) Time consumption with different parameters.

In addition, it should be noted that STAFF is performed offline and the focus of this paper is on the accuracy of signature, whereas we are very concerned about time efficiency when online matching. So, how to efficiently match is our next work. Although time efficiency is not the focus of our investigation when making signatures offline, STAFF still has a great advantage in time efficiency when processing offline traffic with almost the same accuracy.

As shown in Table 2, we run STAFF algorithm four times to generate FFS of five functions with the average time of 11.29 minutes. Although it is extremely difficult for any method to claim 100 percent, it is still worth mention that STAFF needs less time to generate signatures and has the accuracy (SF=4, TFFS=6, accuracy=94.7%) nearly equal to that of ProDecoder [24] (approximately 95%).

Table 2 Summary running time of different methods

ProDecoder		Size	n-gram	KeyWord	Clustering	Sequence
	SMTP	0.34MB	3(s)	172(min)	148(min)	10(min)
	SMB	0.87MB	7(s)	48(min)	15(min)	3(min)
STAFF	1st	3.0MB	5.89(min)			
	2nd	3.4MB	11.91(min)			
	3rd	3.9MB	8.71(min)			
	4th	3.9MB	18.64(min)			
	Average time		11.29(min)			

4.2 Fine-Grained Signature

To illustrate the salient feature of STAFF method, a comparison test among Behavior Signature Method (BS) [18], LASER [17] and STAFF is conducted. The traffic data of the following applications as shown in Table 3 under different user's environment (e.g., different user name, different browsers (Chrome and Mozilla), different operating environments and different times) are collected.

Table 3 Evaluation result

Methods	Category		Signatures
Behaviour Signature Method	Weibo		{sina, Seq, 144, (203.xxx.xxx.0/24, 443, 5, any), (202.xxx.xxx.150/32, 143, 6, any),(202.xxx.xxx.0/24, 25, 4, any), (202.xxx.xxx.156/24, 143, 4, any)}
	Financial System		{YGMISweb, Seq, 384, (10.118.4.228/24, 246, 6, “lt%3Bbp”(0)), (10.118.4.228/32, 80, 6, “GET / YGMISweb /en/th”(0)), (10.118.51.232/24, 8080, 6, “PC0023117408155277752%”(0))}
LASER	Weibo		“HTTP/1” “sina” “User-Type:” “ErrorCode:” “ContentLength:” “Content-Type:” “Last-Modified”
	Financial System		“HTTP1.1” “Client” “Username:” “YGMISweb” “ANS\0x20” “%3Cxml”
STAFF	Weibo	Login	{‘26name%3D’:‘14’,‘/login’:‘14’,‘m%26’:‘17’, ‘?wvr=5&lf=reg’:‘119’,‘_HEADER:’:‘11’}
		Favorite	{‘fav?le’:‘6’,‘m/fa’:‘79’}
		Friends	{‘nds=1&step=2’:‘20’,‘nds?le’:‘20’,‘friends’:‘28’, ‘nds%’:‘10’,‘m/fr’: ‘12’, ‘%26wvr’: ‘7’, ‘ds%3’: ‘12’}
		HomePage	{‘me?le’:‘18’}
		Message	{‘ibo?le’:‘40’,‘/at/we’:‘13’}
	Financial System	Function*1*	{‘djywc’:‘7’,‘ywcd’:‘7’}
		Function*2*	{‘ywcpz&cxpzOptimized=0’:‘8’,‘wpzywc’:‘8’, ‘d=0rnA’:‘7’}
		Function*3*	{‘wdjwwc’:‘7’,‘wwcd’:’7’}
		Function*4*	{‘wpzwwc’:‘8’,‘cpzrnA’:‘7’,‘wwcp’:‘8’}

Table 3 shows several typical application signatures generated by using the BS, LASER and STAFF method. Between BS and LASER, it's hard to judge which method is better. But obviously the STAFF method can generate more fine-grained signature than both BS and LASER. Application signatures can be generated automatically by BS and LASER, whereas STAFF can produce specific function signature in an application. In addition, there are meaningful strings that are associated with corresponding functions in their signatures generated by STAFF. It could provide directly useful information for function and context analysis. For example, Weibo login signature contains '/login', friends signature contains 'friends', Favorite signature contains 'fav?le'. Moreover, we pick up signatures of four very similar functions in a financial system, namely Function1 to Function4. They represent *CompletedReceiptQuery*, *CompletedVoucherQuery*, *UnfinishedReceiptQuery* and *UnfinishedVoucherQuery* function respectively. It is extremely difficult to differentiate them by a manual process, because the only difference of four familiar *function flows* is the abbreviation of Chinese phonetic alphabet. But STAFF does well in this case.

Figure 4 shows that STAFF can effectively distinguish fine-grained functions with high accuracy. It exhibits identification accuracy of STAFF in different target applications. We capture each application traffic 4 times in different operating environments and then run STAFF on those traffic scenarios to get the average accuracy. Although the test cases were limited, we could confirm that the STAFF algorithm has reasonable accuracy. LASER algorithm tries to find the longest common subsequence among function traffic data. LASER works well on extracting the protocol format, but when we pick up signatures for every function by LASER, all the functions' signature are almost the same. And they cannot be used as fine-grain signature effectively. The BS method only identifies a unique traffic behavior pattern appearing in the first few packets of plural traffic flows with a combination of various optional traffic features. There are two features that are unstable in the middle of all BS features, interval (I) and sub-interval (i). Interval (I) is the period of time in milliseconds from the first entry to the last entry, i.e., the time during which all entries of the behavior signature are applied. The sub-Interval (i) is the period of time in milliseconds between each entry. Since measured value of the two features are unstable, the accuracy varies greatly. In addition, it is clear that STAFF can generate FFSs with high identification accuracy (average 93.65%).

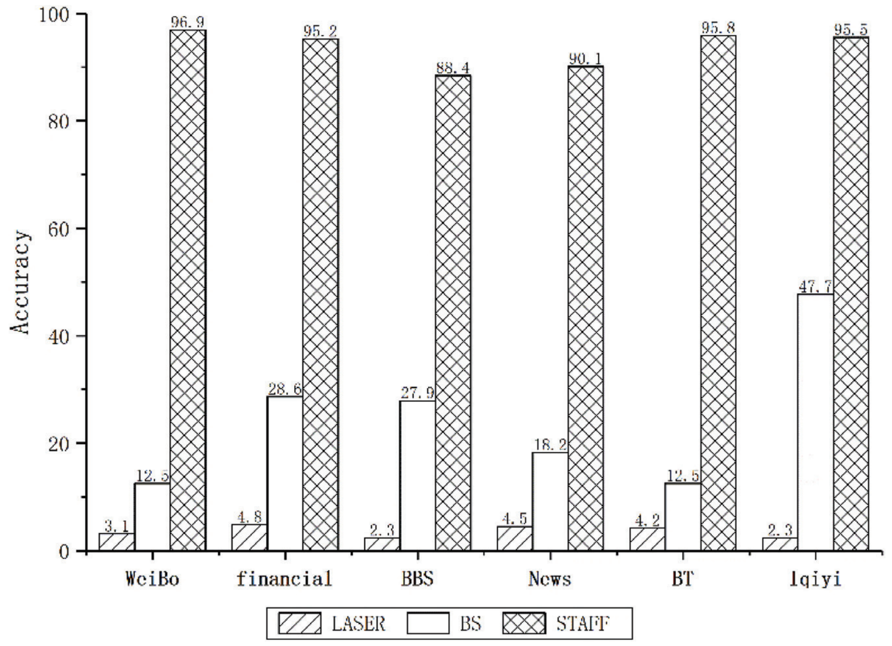


Figure 4 Accuracy comparison of fine-grained test between different methods on different application traffic data.

4.3 Noise Insensitivity

To show that STAFF is effective in traffic with noise, we next evaluate the signature generation using the BS and STAFF method. The basic experimental setup is the same as before. The only difference is that the function flows are captured with noise traffic. Here noise traffic refers to the traffic that does not belong to the specific application function. Figure 5 show that STAFF works well on traffic with noise. As can be seen from the figure, accuracy of the BS method declines dramatically with noise traffic. The presence of noise in traffic change some of the interval time features in the BS. As a result, the identification results are inaccurate. On the contrary, in the STAFF method, we can run the target application for many times in different operating environments and collect target *function flow* many times. STAFF extracts FFS by using the intersected FFSs which are generated from all these environments. Due to traffic noise, accuracy of STAFF on WeiBo has fallen only from 96.9% to 93.8%. It is due to a coincidence that noise traffic has “login” string in it, which leads to that

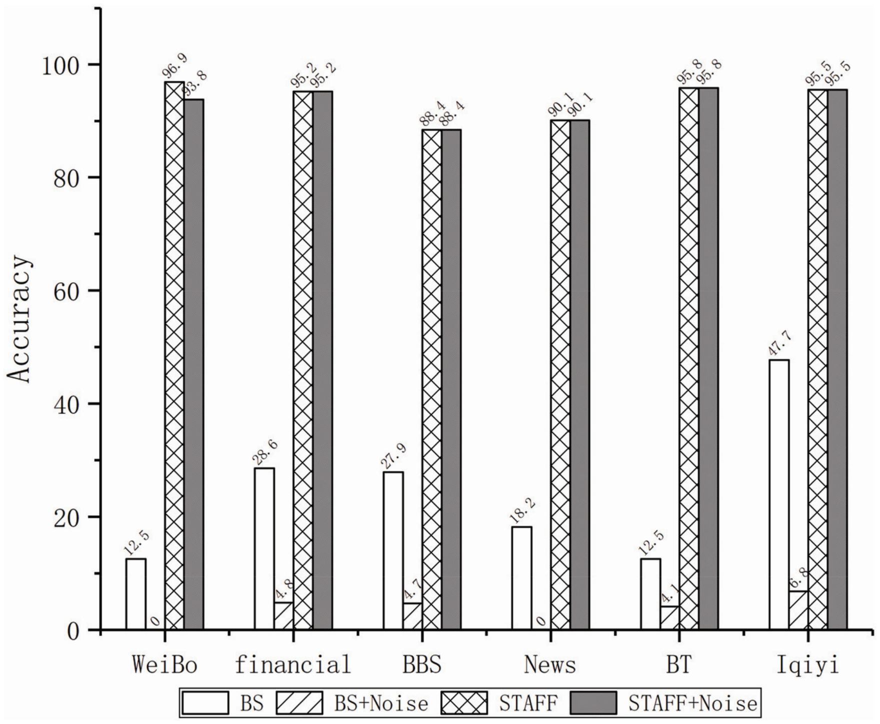


Figure 5 Accuracy comparison of fine-grained test between pure traffic data and impure data.

the frequency of fragment "login" has changed and thus, identification accuracy is affected. Generally speaking, it is unnecessary to clean traffic data. STAFF is effective when target application traffic is mixed with noise traffic.

5 Conclusions

An automatic signature generation method for fine-grained function, STAFF, is proposed in this paper. The method is purely based on raw network packet traces and does not require any protocol format. The basic idea of STAFF is to find all the fragments with their length longer than a prescribed length and their occurrence number bigger than a prescribed frequency. The method was applied to a wide range of real applications and the results show that STAFF can accurately and efficiently infer signatures of *function flows*. It is worthwhile to point out the proposed method cannot handle encrypted payload and the fine-grained function traffic identification for that scenario is not the scope of current work.

Acknowledgements

The authors would like to thank all the anonymous reviewers for their valuable comments. This work is supported by National Key Research and Development Program of China (No. 2016YFB0800100), National Advanced Research Project (No. 315040102) and the Fundamental Research Project of Natural Science in Shaanxi Province (No. 2016JM6066).

References

[1] Moore, A. W., and Papagiannaki, K. (2005). Toward the accurate identification of network applications. In *International Workshop on Passive and Active Network Measurement* (Vol. 5, pp. 41–54). Springer, Berlin, Heidelberg.

[2] Nguyen, T. T., and Armitage, G. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4), 56–76.

[3] Choi, Y., Chung, J. Y., Park, B., and Hong, J. W. K. (2012). Automated classifier generation for application-level mobile traffic identification. In *Network Operations and Management Symposium (NOMS), 2012 IEEE* (pp. 1075–1081). IEEE.

[4] Dainotti, A., Gargiulo, F., Kuncheva, L. I., Pescapè, A., and Sansone, C. (2010). Identification of traffic flows hiding behind TCP port 80. In *International Conference on Communications (ICC), 2010 IEEE* (pp. 1–6). IEEE.

[5] Agrawal, R., and Srikant, R. (1995). Mining sequential patterns. In *Data Engineering, in Proceedings of the Eleventh International Conference on* (pp. 3–14). IEEE.

[6] Srikant, R., and Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology* (pp. 1–17). Springer, Berlin, Heidelberg, Chicago.

[7] Han, J., Pei, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M. C. (2001). Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering* (pp. 215–224).

[8] Benson, G. (1999). Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research*, 27(2), 573.

[9] Apostolico, A., and Preparata, F. P. (1983). Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22(3), 297–315.

[10] Kolpakov, R., and Kucherov, G. (1999). Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science*, (pp. 596–604), IEEE.

[11] Wang, D., Wang, G., Wu, Q., and Chen, B. (2005). Finding LPRs in DNA sequences based on a new index-SUA. In *Fifth IEEE Symposium on Bioinformatics and Bioengineering, BIBE*, (pp. 281–284).

[12] Singh, S., Estan, C., Varghese, G., and Savage, S. (2004). Automated Worm Fingerprinting. In *OSDI* (Vol. 4, pp. 4–4).

[13] Kim, H. A., and Karp, B. (2004). Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX security symposium* (Vol. 286).

[14] Newsome, J., Karp, B., and Song, D. (2005). Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE symposium on Security and privacy,* (pp. 226–241), IEEE.

[15] Park, B. C., Won, Y. J., Kim, M. S., and Hong, J. W. (2008). Towards automated application signature generation for traffic identification. In *IEEE Network Operations and Management Symposium*, (pp. 160–167), IEEE.

[16] Wang, Y., et al., (2012). A semantics aware approach to automated reverse engineering unknown protocols. In *20th IEEE International Conference on Network Protocols (ICNP),* (pp. 1–10), IEEE.

[17] Park, B., Hong, J. W. K., and Won, Y. J. (2011). Toward fine-grained traffic classification. *IEEE Communications Magazine*, 49(7).

[18] Yoon, S. H., Park, J. S., and Kim, M. S. (2015). Behavior signature for fine-grained traffic identification. *Appl. Math*, 9(2L), 523–534.

[19] Erman, J., Mahanti, A., and Arlitt, M. (2006). Qrp05-4: Internet traffic identification using machine learning. In *Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE* (pp. 1–6), IEEE.

[20] Williams, N., Zander, S., and Armitage, G. (2006). A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 36(5), 5–16.

[21] Dahmouni, H., Vaton, S., and Rossé, D. (2007). A markovian signature-based approach to IP traffic classification. In *Proceedings of the 3rd Annual ACM Workshop on Mining Network Data*, (pp. 29–34), ACM.

[22] Mannila, H., Toivonen, H., and Verkamo, A. I. (1997). Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3), 259–289.

[23] Norvig, P. (2013). English letter frequency counts: Mayzner revisited or etaoin srhldcu. *Dostopno na http://www.norvig.com/mayzner.html* [obiskano 2016-08-07].

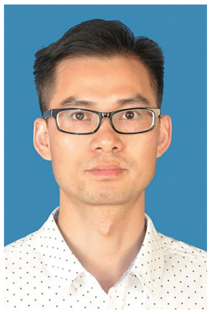
[24] Wang, Y., et al., (2012). A semantics aware approach to automated reverse engineering unknown protocols. In *20th IEEE International Conference on Network Protocols (ICNP),* (pp. 1–10). IEEE.

Biographies

Yazhe Tang received his B.Sc., M.Sc. and Ph.D. degrees from Xi'an Jiaotong University, Xi'an, China, in 1993, 1996 and 2002. He is an Associate Professor with the Department of Computer Science and Technology, Xi'an Jiaotong University. His main research interests include computer networking systems, network security, and network measurement and monitoring.

Xun Li is a Ph.D. student at Xi'an Jiaotong University, Xi'an, China, since autumn 2014. He attended North China Electric Power University where he received his B.Sc. in Software Engineering in 2010. He received his M.Sc.

in computer science from Xi’an Jiaotong University. Xun Li is currently completing a doctorate in Computer Science at Xi’an Jiaotong University. His Ph.D. work centers on network measurement and monitoring.



Lishui Chen received the B.Sc. and Ph.D. degrees from Xi’an Jiaotong University, Xi’an, China, in 2005 and 2011. He is a senior engineer in the 54th research institute of Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory, China. His main research interests include communications network system, information system and cloud computing.