IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 12, 2020, accepted February 6, 2020, date of publication February 10, 2020, date of current version February 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2972691

Efficient and Robust Syslog Parsing for Network Devices in Datacenter Networks

SHENGLIN ZHANG[1], (Member, IEEE), YING LIU[2,3], (Member, IEEE), WEIBIN MENG[3,4], JIAHAO BU[3,4], SEN YANG[5], YONGQIAN SUN[1], DAN PEI[3,4], (Senior Member, IEEE), JUN XU[6], (Senior Member, IEEE), YUZHI ZHANG[1], LEI SONG[7], AND MING ZHANG[8]

[1]College of Software, Nankai University, Tianjin 300071, China
[2]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
[3]BNRist, Beijing 100084, China
[4]Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[5]Facebook, Inc., Menlo Park, CA 94025, USA
[6]School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA
[7]Baidu, Inc., Beijing 100085, China
[8]China Construction Bank, Beijing 100033, China

Corresponding author: Yongqian Sun (sunyongqian@nankai.edu.cn)

This work was supported by the National Key Research and Development Program of China under Grant 2018YFB0204304.

ABSTRACT Syslog parsing is of vital importance for the detection, diagnosis and prediction of network device failures in a datacenter. A common approach to syslog parsing is to extract templates from historical syslogs, after which syslogs are matched to these templates. To address the problems in the existing syslog parsing techniques, we propose a novel framework, Craftsman, which identifies frequent combinations of (syslog) words and then applies them as templates. Craftsman empirically extracts templates accurately, is extremely efficient in template matching, and naturally supports incremental learning. To compare the performance of Craftsman and three other template learning techniques designed for network devices, we experiment them on two-years' worth of syslogs collected from network devices deployed across 10+ datacenters of a tier-one service provider. The experiments demonstrate that Craftsman achieves a close-to-one accuracy (as measured by rand index), and improves the computational efficiency by 6.88 to 10.25 times in template matching, and by 730 to 6847 times in syslog parsing. It also improves the accuracy (as measured by F1 measure) of failure prediction by 13.07% to 188%. In addition, we demonstrate Craftsman's superior generality by comparing it with three widely-applied log parsing methods over five large log datasets collected from servers, distributed systems and applications.

INDEX TERMS Syslog parsing, network device, prefix tree, datacenter network, frequent pattern.

I. INTRODUCTION

Nowadays, cloud service providers employ a large number of network devices in their datacenter networks, and network device failures are not rare any more. For example, Microsoft's datacenter networks have deployed tens of thousands of network devices [1], and there occur more than 400 network device failures per year [2]. Since a network device failure can dramatically degrade the the performance of a datacenter, researchers have paid much attention on the detection/diagnosis/prediction of network device failures [3]–[7]. Network device syslogs have been recognized as a rich source of information for failure detection, diagnosis, and prediction [4], [5], [8]–[12]. However, since syslogs are usually unstructured texts, they have to be properly parsed before they can be effectively used [13]–[15]. In this paper, we focus on *syslog parsing* techniques that can lead to more accurate and efficient network device failure detection, diagnosis and prediction.

The current approach to parsing network device syslogs is to *learn* message templates from syslogs and then *match* the syslogs to the templates [4], [8], [9]. Therefore, syslog parsing includes *template learning* and *template matching*. For example, in the syslog message **OSPF Neighbor 10.231.44.249 (Vlan-interface18) from Exstart to Exchange**, **10.231.44.249** and **Vlan-interface18** are parameters that vary from one message to another and are not parts of a template, whereas the rest, *i.e.,* **OSPF Neighbor . . . from Exstart to Exchange**, sketches out the event and hence is a template that summarizes this and other similar syslogs.

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

Despite the crucial role that syslog parsing plays in network device failure detection, diagnosis and prediction as aforementioned, existing syslog parsing techniques for network devices, *e.g.,* Statistical Template Extraction (STE) [9] and LogSimilarity [4], have low accuracy in *learning* the ''correct'' set of templates, and are inefficient in *matching* syslogs to templates. The large number of network devices (say more than 10 thousand) in modern datacenter networks generate tens of millions of syslogs everyday. Therefore, a syslog parsing method which is inefficient in matching a syslog message to its template will consume too much computational resources.

Some syslog parsing techniques, *e.g.,* the signature tree based method[1] [8] and STE [9], do not support incremental learning in the sense that the entire set of templates has to be re-parsed (*i.e.,* relearn templates and rematch all the syslogs to them) when a new template is added. Typically, the syslog based failure detection/diagnosis/prediction method applies machine learning methods to learn the failure patterns from historical syslogs and failures [4], [9], [16]–[21]. The underlying detection/diagnosis/prediction model, which is trained based on the template library, has to be retrained every once in a while to kept up-to-date. It is highly desirable for the template library to be incrementally retrainable for the following reason. Network operators frequently conduct software or firmware upgrades on network devices to introduce new features, or fix bugs in the previous version [22]. These updates can generate new *subtypes* of syslogs that cannot match any existing template, thus requiring new templates to be learned from these new messages and added to the template library. Consequently, new templates have to be learned from these new syslogs and then added to the template library. Only the syslogs that arrive after the previous upgrade need to match the new template library if the log parsing model is incrementally retrainable. Otherwise, the log parsing method has to re-parse all historical syslogs based on the new template library. The unincrementally retrainable manner, which STE and SignatureTree fall into, is prohibitively expensive computationally. That is because a large datacenter usually deploys tens of thousands of network devices, and detects/diagnoses/predicts failures based on historical syslogs and failures for a long period (*e.g.,* two years).

We propose a novel framework, Craftsman, which has high accuracy in identifying the ''correct'' templates, and is incrementally retrainable and computationally efficient in template matching. Usually, a ''correct'' message template is a combination of words that occur frequently in syslogs. Therefore, Craftsman dynamically builds a prefix-tree structure of these frequent words, and applies the prefix-tree to construct the template library. Because this prefix-tree structure as well as the template library dynamically and incrementally evolve with the arrival of new syslogs (may be of the new message *subtypes*), Craftsman is incrementally retrainable in nature. Moreover, thanks to the prefix-tree structure, Craftsman is extremely efficient in matching a syslog message to its template. Craftsman also empirically guarantees that the template learned from a given syslog accurately characterizes the event that the syslog represents. Craftsman is, to the best of our knowledge, the first framework that leverages the idea of prefix-tree and frequent patterns to achieve an accurate, efficient, and incrementally retrainable log parsing method.

To compare the performance of Craftsman to those of SignatureTree, STE, and LogSimilarity, we apply the syslogs collected from 2, 223 network devices over a two-year period. These network devices are deployed across more than 10 datacenters owned by *a tier-1 cloud service provider*. We evaluate the accuracy of the four techniques based on manual classifications of syslogs. Craftsman achieves close-to-one accuracy in template learning, and it respectively improves the computational efficiency of template matching by 6.88 and 10.25 times compared to LogSimilarity and STE. In addition, STE and SignatureTree consume 6847 and 730 times more computational resources than Craftsman in log parsing (*i.e.,* template learning and matching), respectively, as neither of them is incrementally retrainable. Moreover, Craftsman improves the prediction accuracy by 13.07% and 35.42% (using PreFix [5] as the failure prediction method), or 155% and 188% (using Hidden Semi-Markov Model (HSMM) [16] as the failure prediction method), compared to LogSimilarity and STE, respectively. The evaluation experiment results clearly demonstrate the benefits of Craftsman: highly accurate, incrementally retrainable, and extremely efficient in matching a syslog message to its template. The implementation of Craftsman is publicly available now at [23], and we hope that this can help researchers further understand the intuition, framework and performance of Craftsman.

In addition, to demonstrate how Craftsman is general in parsing syslogs for other types of logs, we compare Craftsman with log parsing methods designed for *servers (supercomputers), distributed systems and applications*. These methods include Iterative Partitioning Log Mining (IPLoM) [24], Log Key Extraction (LKE) [25] and LogSig [26]. The comparison experiments are conducted over five large log datasets ranging from supercomputers (Blue Gene/L and High Performance Cluster) to standalone software (Proxifier) to distributed systems (HDFS and Zookeeper). Craftsman achieves an averaged accuracy of 96.25% across the five datasets, which is better than the above three methods. Moreover, Craftsman has a much smaller log parsing (*i.e.,* template learning and matching) time (4.98*s*) per day than the other three methods (35 *min* to 5.38 *day*), and it consumes much less memory space (32.4 *MB*) compared with these methods (90.6 *MB* to 23.5 *GB*). The experiment results show Craftsman's superior generality on diverse types of logs.

A preliminary version of this paper appears as [27]. The rest of the paper is organized as follows. We provide an introduction to datacenter network architecture and network

[1] We refer to this method as SignatureTree hereafter

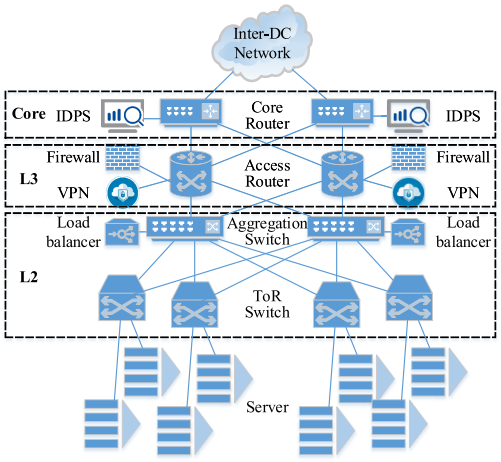


FIGURE 1. Typical datacenter network architecture.

device syslog in Section II, followed by the intuition of Craftsman in Section III. The design of Craftsman is elaborated in Section IV, and we present the evaluation experiments in Section V. The related works, including SignatureTree, STE, LogSimilarity, and the log parsing methods in other areas, are discussed in Section VI. Finally, we conclude our paper in Section VII.

II. BACKGROUND

In this section, we first briefly introduce in Section II-A the architecture of a datacenter network and the important role that switches and routers play in it, and then describe in Section II-B network device syslogs in details.

A. DATACENTER NETWORK ARCHITECTURE

Nowadays, computer servers in a datacenter are connected via a network of commodity Ethernet switches and routers [28]. Figure 1 shows a typical datacenter network architecture [29]. It is comprised of several layers. At the bottom layer, servers are mounted on racks and connected to a ToR switch via Ethernet NICs. Tens of ToRs are in turn connected to a primary aggregation switch and a backup one for redundancy purposes (L2). These two aggregation switches then forward the traffic from ToRs to access routers (L3), each of which aggregates traffic forwarded from multiple ToRs, and route it to core routers. Each datacenter network is connected to other datacenter networks and the Internet via core routers. Several types of middleboxes, such as load balancers, VPNs, firewalls, and intrusion detection and prevention systems (IDPSes), are also deployed in datacenter networks to improve performance and/or enhance security.

Switches and routers are the most populous type of network devices in a cloud datacenter. The number of switches and routers in a datacenter has also grown expolsively recently. For example, Guo *et al.* reported that there were tens of thousands of switches and routers across Microsoft's datacenters [1]. Similarly, the tier-1 cloud service provider we are working with also deploys tens of thousands of switches and routers that are manufactured by several different vendors. *In this work, we study syslog parsing for the most important and populous network devices – routers and switches – in datacenter networks.*

B. NETWORK DEVICE SYSLOG

Each network device reports, from time to time, the observed hardware/software condition or (anomalous) event, in a syslog message. Examples of such conditions or events include state changes of interfaces, links, or neighbors (*e.g.*, the state of an interface changes from up to down), operational maintenance (*e.g.,* operators log in/out), environmental condition alerts (*e.g.,* high temperature), *etc.* Although syslogs are designed mainly for debugging software and hardware problems, they can also be used for detecting, diagnosing, and predicting network incidents [4], [5], [8], [9]. Hence, usually dedicated servers are deployed in a datacenter to collect syslogs from all its network devices [27].

As can be seen from several example syslogs shown in Table 1, a syslog message usually has a simple structure consisting of several fields, including a timestamp recording when the network device generates the syslog message, a network device ID identifying the network device that generates the message, a message type that describes the rough characteristics of the message, and a message body depicting the details of the event. The syntax and semantics of the *message type* field and the *detailed message* field vary with network device vendors and models [27].

III. INTUITION OF Craftsman

The syslogs of network devices are essentially unstructured texts in diverse forms. Moreover, a specific IP address or interface ID that appears in a syslog may never appear again. Consequently, it is quite difficult, if possible, to learn failure patterns from *raw syslogs* for failure detection/diagnosis/prediction.

As shown in Table 1, each syslog includes a message type field that describes the schematic characteristics of the event. However, there can be multiple *subtypes* in a message type. For example, although in Table 2 there are 21 syslogs that belong to the message type "10OSPF/5/OSPF_NBR_CHG" and describe the changes of OSPF neighbors, the detailed messages can be quite different. The detailed message field of syslogs is essentially free-form text, and a network device's OS usually "printf"ed this field with detailed information, such as the location (line card/port/interface), package loss ratio, IP address, *etc*., as parameters. For instance, the detailed message field of the syslog message in the first line of Table 2, *i.e.,* **OSPF Neighbor 10.231.51.249 (Vlan-interface03) from Init to Exstart**, means that the event impacts the interface numbered **Vlan-interface03** of the network device with IP address **10.231.51.249**, and thus the state of the OSPF status changes from Init to Exstart. In other words, the **10.231.51.249** and the **Vlan-interface03** parts are the parameters describing detailed information (hereafter, we collectively refer to this type of

TABLE 1. Examples of network device syslogs.

Vendor	Time stamp	Device ID	Message type	Detailed message
Vendor 1	13:21:29	Switch 1	10IFNET/3/PHY_UPDOWN	Bridge-Aggregation1 link status is DOWN
Vendor 1	07:01:03	Switch 4	10IFNET/5/LINK_UPDOWN	Line protocol on the interface Vlan-interface20 is DOWN
Vendor 1	19:18:55	Switch 4	10LLDP/5/LLDP_ NEIGHBOR_AGE_OUT	Neighbor aged out on Port Ten-GigabitEthernet1/0/33 (IfIndex 33), Chassis ID is 4403-a727-7fe3, Port ID is Eth1/32.
Vendor 2	21:04:16	Router 2	10DEVM/2/POWER_FAILED	Power PowerSupply1 failed
Vendor 2	23:20:56	Router 7	10IFNET/3/LINK_UPDOWN	GigabitEthernet1/0/18 link status is DOWN
Vendor 2	17:38:45	Router 9	10CFM/5/CFM_SAVE CONFIG_SUCCESSFULLY	Configuration is saved successfully

TABLE 2. An example of the syslog sequence before a switch failure.

Time stamp	Message type	Detailed message
12:03:12	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.51.249 (Vlan-interface03) from Init to Exstart
12:03:42	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.53.249 (Vlan-interface07) from Loading to Full
12:03:47	10LAGG/6/LAGG_ INACTIVE_PHYSTATE	Member port XGE1/0/33 of aggregation group BAGG1 became inactive, because the physical state of the port is down
12:03:52	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.56.249 (Vlan-interface09) from Exstart to Exchange
12:03:58	10IFNET/5/LINK_UPDOWN	Line protocol on the interface Bridge-Aggregation1 is DOWN
12:04:14	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.55.249 (Vlan-interface05) from Init to Exstart
12:04:19	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.48.249 (Vlan-interface13) from Full to Down
12:04:39	10IFNET/3/PHY_UPDOWN	Ten-GigabitEthernet1/0/34 link status is DOWN
12:05:35	10LLDP/5/LLDP_ NEIGHBOR_AGE_OUT	Neighbor aged out on Port Ten-GigabitEthernet1/0/34 (IfIndex 34), Chassis ID is 4403-a726-5434, Port ID is Eth2/17
12:05:47	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.50.249 (Vlan-interface15) from Init to Exstart
12:05:53	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.52.249 (Vlan-interface10) from Full to Down
12:05:59	10IFNET/3/PHY_UPDOWN	Ten-GigabitEthernet1/0/35 link status is DOWN
12:06:13	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.44.249 (Vlan-interface18) from Exstart to Exchange
12:06:25	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.53.249 (Vlan-interface22) from Loading to Full
12:06:37	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.49.249 (Vlan-interface23) from Full to Down
12:06:41	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.55.249 (Vlan-interface12) from Exstart to Exchange
12:06:53	10IFNET/5/LINK_UPDOWN	Line protocol on the interface Vlan-interface20 is DOWN
12:07:19	10IFNET/3/PHY_UPDOWN	Ten-GigabitEthernet1/0/34 link status is DOWN
12:07:37	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.47.249 (Vlan-interface20) from Loading to Full
12:07:53	10IFNET/3/PHY_UPDOWN	Ten-GigabitEthernet1/0/35 link status is DOWN
12:08:25	10OSPF/5/OSPF_NBR_CHG	OSPF Neighbor 10.231.54.249 (Vlan-interface27) from Loading to Full

words as parameter words), whereas the rest parts, *i.e.,* **OSPF Neighbor . . . from Init to Exstart**, are predefined outputs by the network device's OS and can be used as a *subtype* for the syslogs of the message type **10OSPF/5/OSPF_NBR_CHG** (hereafter, we collectively refer to this type of words as template words). When we apply the same symbol (*e.g.*, an asterisk as shown in Table 3) to mask the variable of the detailed message field of the 21 syslogs (*i.e.,* IP address or vlan-interface number), we can see that there are only four different syslog structures, or *subtypes*.

However, *manually* obtaining all *subtypes* without domain knowledge is almost impossible because not every part that should be masked in the detailed message can be as easily characterizable as the interface number or IP address. In addition, although *part* of the syslog *subtypes* can be obtained with support from vendors, these *subtypes* may *change* due to software upgrades. Therefore, our objective is to automatically obtain *message templates* in which the need-to-be-masked parts are removed and the message

TABLE 3. Syslog message *subtypes* of SIF.

Subtype No.	*Subtype* structure
$N1$	OSPF Neighbor * (*) from Init to Exstart
$N2$	OSPF Neighbor * (*) from Exstart to Exchange
$N3$	OSPF Neighbor * (*) from Full to Down
$N4$	OSPF Neighbor * (*) from Loading to Full

subtypes are retained without relying on any domain knowledge. [27]

After investigating thousands of real-world network device syslogs, we have the following two observations:

- *Parameters words are much less than template words.* As shown in Table 2 and Table 3, it is obvious that, in a syslog message, the number of parameter words is usually much smaller that that of template words. For example, in the syslogs of the message type

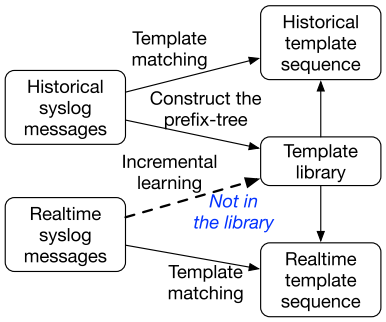


FIGURE 2. The framework of Craftsman.

"10OSPF/5/OSPF_NBR_CHG", there are six template words, and only two parameter words. It is consistent with the common practice that the OS of a network device usually reserves only a small number of positions for parameters in the "printf" function of a syslog message. In addition, the parameter words of different syslogs are usually different, whereas the template words of the syslogs belonging to the same subtype are the same. Therefore, in a large set of syslogs, parameter words are much less than template words.

- *A message type has a small number of subtypes, and each subtype has a large number of syslogs.* Typically, there are a limited number (usually < 10) of subtypes in each message type. That is because a message type usually includes a small number of different events, which are sketched out by subtypes or templates. Moreover, in a large cloud datacenter, there are a quite large number of syslogs belonging to the same message type. As a result, a subtype usually has a great many syslogs.

IV. FRAMEWORK OF Craftsman

Our objective in syslog parsing is to automatically, accurately, and efficiently learn template and *subtype* information from syslogs without relying on any domain knowledge, and efficiently match syslogs to their templates. As aforementioned, three syslog parsing methods, *i.e.,* SignatureTree, STE, and LogSimilarity, were proposed. However, they are not well suited for our application scenario because of their low accuracy in template (or *subtype*) learning, or their inefficiency in template matching, or their inability to be incrementally retrained.

Inspired by the Frequent Pattern Tree (FP-tree) [30], which is designed for storing compressed, crucial information about frequent patterns in database, we propose Craftsman, an incrementally retrainable technique with high accuracy in template (*subtype*) learning, and high efficiency in template matching. Craftsman constructs a prefix-tree structure to encode and learn message templates. Intuitively, a syslog *subtype*, or a template, is usually the longest combination of words with high frequency (see Section III for more details), and thus learning a template can be transformed to identifying such longest combination of frequent words from syslogs. In addition, motivated by the excellent performance of prefix-tree in packet forwarding [31], we believe that the prefix-tree structure is extremely efficient in matching syslogs to their templates. Craftsman is, to the best of our knowledge, the first framework that leverages the idea of prefix-tree and frequent patterns to achieve an accurate, efficient, and incrementally retrainable log parsing method.

Figure 2 shows the architecture of Craftsman. For a given switch model, Craftsman *learns* templates from historical syslogs, and builds a template library, based on which Craftsman *matches* historical syslogs to template sequences. If a realtime syslog message cannot match any template in the template library, Craftsman will incrementally learn a template from it and extend the library. Otherwise, Craftsman will match the realtime syslogs to template sequences. Both template learning and template matching should be efficient because of the large number (say tens of millions) of syslogs generated everyday in a large datacenter.

In the following, we first introduce the design and construction of the prefix-tree in Section IV-A, and then demonstrate how Craftsman facilitates incremental template learning in Section IV-B. We then describe how Craftsman support efficient template matching in Section IV-C. Finally, we show the space complexity of Craftsman in Section IV-D.

A. DESIGN AND CONSTRUCTION OF PREFIX-TREE

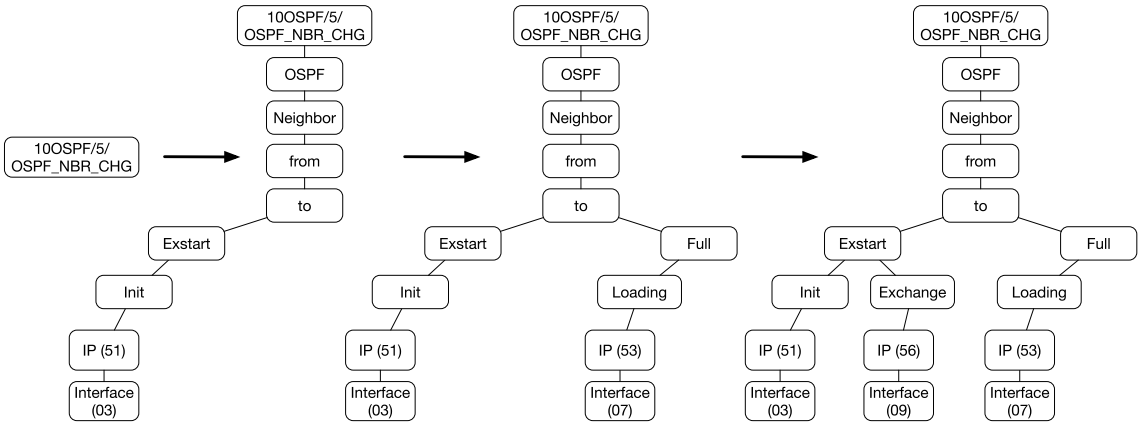
Algorithm 1 Prefix-Tree Construction in Craftsman [27]

Input: A message set DM containing all the different syslogs of a specific message type, and a threshold k

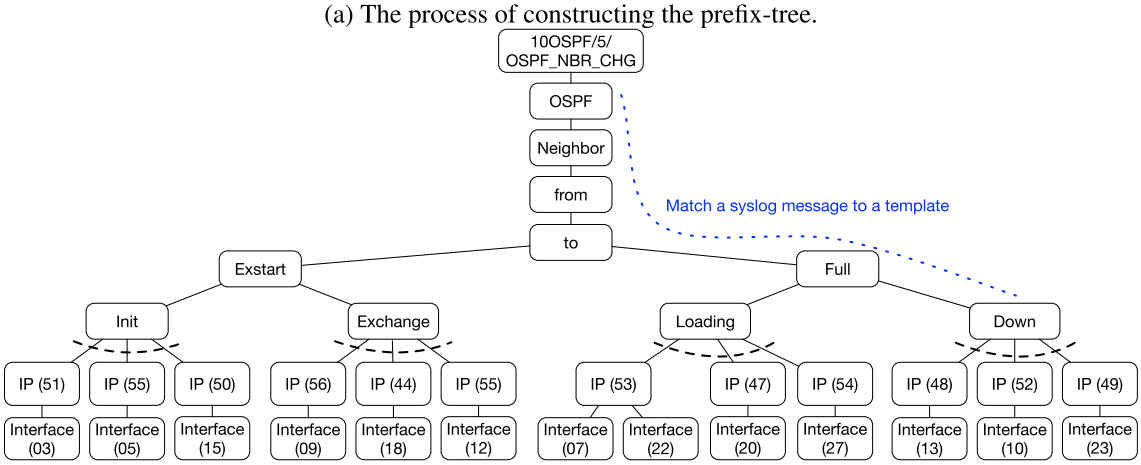
Output: A prefix-tree, T

```
Scan the message set DM once
Calculate the support for each word in I
Let L be the list of words in the descending order of their supports
Create the root of T and label it as the message type
for each message in DM do
    Sort its words according to their order in L
    Let the sorted word list be [p|P], where p is the first element and P is the remaining list
    Call insert_tree([p|P], T)
end for
for Child C in T do
    if C has more than k children then
        Eliminate all the children of C
    end if
end for
return T
```

Let $I = a_1, a_2, \ldots, a_m$ be the set of distinct words that occur in a message set $DM = \langle M_1, M_2, \ldots, M_n \rangle$, where each M_i is a syslog message. The **support** (*i.e.*, the frequency of occurrences) of a word combination (*i.e.*, a set of words) A, is the number of distinct messages containing A in DM. A is considered a template if A is occurring frequently (*i.e.*, with a large support) [27].



(a) The process of constructing the prefix-tree.



(b) The final prefix-tree.

FIGURE 3. An example of constructing a prefix-tree. "IP (*)" denotes the IP address is 10.231.*.249, and "Interface (*) denotes the interface is Vlan-interface*".

The second column of Table 4 shows an example syslog message set $DM = \langle M_1, M_2, \ldots, M_{13} \rangle$, in which every message belongs to the message type "10OSPF/5/OSPF_NBR_CHG". The prefix-tree for this *DM* is constructed, using the algorithm shown in Algorithm I, as follows [27].

First, our prefix-tree construction algorithm scans *DM* once (line 1 in Algorithm I), and derives a *list* L of words in the descending order of their frequency of occurrences. Clearly, the words and their frequency in L is listed in Table 5. [27]

Then, we create the root of a tree which is labeled with the message type, which in this case is "10OSPF/5/OSPF_NBR_CHG", as shown in Figure 3 (a). Our prefix-tree construction algorithm scans *DM* for a second time (lines 5 through 9 in Algorithm I). The parsing of M_1 leads to the construction of the first path/branch of the tree: ⟨"OSPF", "Neighbor", "from", "to", "Exstart", "Init", "10.231.51.249" (IP (51)), "Vlan-interface03 (Interface (03))"⟩ (a word in a prefix-tree is called the *nword* of the node containing the word). Note these words are ordered according to the order of the words in L. When M_2 is parsed, since its ordered word list ⟨"OSPF", "Neighbor", "from", "to", "Full", "Loading", "10.231.53.249" (IP (53)), "Vlan-interface07 (Interface (07))"⟩ shares a common prefix ⟨"OSPF", "Neighbor", "from", "to"⟩ with the existing path/branch, a new branch ⟨"Full", "Loading", "10.231.53.249" (IP (53)), "Vlan-interface07 (Interface (07))"⟩ is created as a subtree of node ⟨"to"⟩. The rest 11 messages in Table 4 are parsed similarly, resulting in the final prefix-tree shown in Figure 3 (b).

Finally, we prune the tree until it satisfies the following node degree constraint (lines 10 through 14 in Algorithm I). Intuitively, as mentioned in Section III, there should be only a small number of *subtypes* for each message type, and, for each *subtype*, there should be many different syslogs that match to it. Therefore, if a node has too many children (say exceeding a threshold k, and operators empirically set $5 \leq k \leq 10$), all its children (or subtrees) are deleted from the tree and the node will become a leaf itself. In the pruned prefix-tree, each root-to-leaf path is a message template (*i.e.,* type + *subtype*). For example, ⟨"OSPF", "Neighbor", "from", "to", "Exstart", "Init"⟩ is a message template, as shown in Figure 3 (b). [27]

TABLE 4. Different messages belonging to message type "10OSPF/5/OSPF_NBR_CHG" in Table 2, and the words ordered according to L.

Message No.	Detailed Message	Words ordered according to L
M_1	OSPF Neighbor 10.231.51.249 (Vlan-interface03) from Init to Exstart	"OSPF", "Neighbor", "from", "to", "Exstart", "Init", "10.231.51.249", "Vlan-interface03"
M_2	OSPF Neighbor 10.231.53.249 (Vlan-interface07) from Loading to Full	"OSPF", "Neighbor", "from", "to", "Full", "Loading", "10.231.53.249", "Vlan-interface07"
M_3	OSPF Neighbor 10.231.56.249 (Vlan-interface09) from Exstart to Exchange	"OSPF", "Neighbor", "from", "to", "Exstart", "Exchange", "10.231.56.249", "Vlan-interface09"
M_4	OSPF Neighbor 10.231.55.249 (Vlan-interface05) from Init to Exstart	"OSPF", "Neighbor", "from", "to", "Exstart", "Init", "10.231.55.249", "Vlan-interface05"
M_5	OSPF Neighbor 10.231.48.249 (Vlan-interface13) from Full to Down	"OSPF", "Neighbor", "from", "to", "Full", "Down", "10.231.48.249", "Vlan-interface13"
M_6	OSPF Neighbor 10.231.50.249 (Vlan-interface15) from Init to Exstart	"OSPF", "Neighbor", "from", "to", "Exstart", "Init", "10.231.50.249", "Vlan-interface15"
M_7	OSPF Neighbor 10.231.52.249 (Vlan-interface10) from Full to Down	"OSPF", "Neighbor", "from", "to", "Full", "Down", "10.231.52.249", "Vlan-interface10"
M_8	OSPF Neighbor 10.231.44.249 (Vlan-interface18) from Exstart to Exchange	"OSPF", "Neighbor", "from", "to", "Exstart", "Exchange", "10.231.44.249", "Vlan-interface18"
M_9	OSPF Neighbor 10.231.53.249 (Vlan-interface22) from Loading to Full	"OSPF", "Neighbor", "from", "to", "Full", "Loading", "10.231.53.249", "Vlan-interface22"
M_{10}	OSPF Neighbor 10.231.49.249 (Vlan-interface23) from Full to Down	"OSPF", "Neighbor", "from", "to", "Full", "Down", "10.231.49.249", "Vlan-interface23"
M_{11}	OSPF Neighbor 10.231.55.249 (Vlan-interface12) from Exstart to Exchange	"OSPF", "Neighbor", "from", "to", "Exstart", "Exchange", "10.231.55.249", "Vlan-interface12"
M_{12}	OSPF Neighbor 10.231.47.249 (Vlan-interface20) from Loading to Full	"OSPF", "Neighbor", "from", "to", "Full", "Loading", "10.231.47.249", "Vlan-interface20"
M_{13}	OSPF Neighbor 10.231.54.249 (Vlan-interface27) from Loading to Full	"OSPF", "Neighbor", "from", "to", "Full", "Loading", "10.231.54.249", "Vlan-interface27"

TABLE 5. The words and their frequency in L.

Word	OSPF	Neighbor	from	to	Full	Exstart	Loading	Init	Full	Exchange	10.231.53.249
Frequency	12	12	12	12	7	6	4	3	3	3	2
Word	10.231.55.249		10.231.44.249		10.231.47.249		10.231.48.249		10.231.49.249		10.231.50.249
Frequency	2		1		1		1		1		1
Word	10.231.51.249		10.231.52.249		10.231.54.249		10.231.56.249		Vlan-interface03		Vlan-interface05
Frequency	1		1		1		1		1		1
Word	Vlan-interface07		Vlan-interface09		Vlan-interface10		Vlan-interface12		Vlan-interface13		Vlan-interface15
Frequency	1		1		1		1		1		1
Word	Vlan-interface18		Vlan-interface20		Vlan-interface22		Vlan-interface23		Vlan-interface27		
Frequency	1		1		1		1		1		

Definition 1 ([27]): As illustrated by the above example, given a specific message type, the prefix-tree is defined as follows:

1) Its root is labeled by the message type, and each root-to-leaf path corresponds to a message template.
2) Each non-root node in the prefix-tree has only one attribute/field, namely *nword*, which registers which word this node represents.

It remains to describe the function *insert_tree*($[p|P], T$) in Algorithm 1. It works as follows. If $\nexists N$, $N.nword = p.nword$, and N is a child of T, then create a new node N, and make it a child of T. If $P \neq \Phi$, call *insert_tree*(P, N) recursively [27].

Note that as shown in Figure 3 (b), a prefix-tree is built for each message type, and the parameters of syslogs are removed by pruning nodes from a prefix-tree. Although we use IP addresses and interface numbers as examples to illustrate the framework of Craftsman, Craftsman is also capable of removing other types of parameters that varies from one syslog message to another including software module IDs, physical locations, the number of sessions, *etc.*

B. INCREMENTAL TEMPLATE LEARNING

As mentioned earlier, for a given message type, new *subtypes* of messages can emerge due to software or firmware upgrades, and new templates need to be generated for these messages to match to. As shown in Figure 3 (a) and Algorithm 1, this is accomplished via inserting new nodes/branches to the prefix-tree. It is also clear from Algorithm 1 that such insertions can be done incrementally, by scanning only the *recently arrived* syslogs (after the software or firmware upgrade) twice. More specifically, when a new message M_{new} that *does not match any template* arrives, a new branch is inserted into the prefix-tree by calling *insert_tree*() (line 8) in Algorithm 1. Please note that, if one or more words in M_{new} do not exist in L, we will add these words to the tail of L. When a network device starts to generate new *subtypes* of messages, it is highly likely that, *in one day*, these messages will contain enough number (more

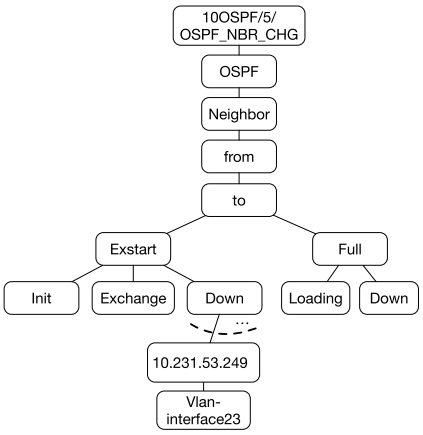


FIGURE 4. The prefix-tree after M_{new} is added.

than the pruning threshold k) of distinct parameter words (such as "10.231.55.249", "Vlan-interface07" in Figure 3) as the children of a node, so that this node becomes a leaf itself after the pruning [27].

We illustrate this incremental learning process by an example shown in Figure 4. Suppose that a new message M_{new} = "OSPF 1 Neighbor 10.231.53.249 (Vlan-interface23) from ExStart to Down" arrives, and that right before this arrival, the prefix-tree is the one shown in Figure 3 (b), but with all leaf nodes ("10.231.53.249 (IP (53))", "Vlan-interface03 (Interface (03))", *etc.*) pruned. Then the newly arrived message M_{new} results in the insertion of the branch ("Down"→"10.231.53.249"→"Vlan-interface23") to the prefix-tree, and the new branch ("10.231.53.249"→"Vlan-interface23") will eventually be pruned as explained earlier.

Although SignatureTree is also designed based on the tree structure, it is trained based on all the historical syslogs, rather than trained incrementally. Each node in a SignatureTree is a *combination of words*, rather than a word alone. It is difficult, if not impossible, to improve SignatureTree to support incremental training.

C. TEMPLATE MATCHING

Now a historical/recent syslog message can be matched to a specific template, which is usually represented as a numerical value. The process of matching a syslog message to its template is illustrated in Algorithm 2. At first, we sort the words in the syslog message based on L (line 2 in Algorithm 2). After that, we match each word to the *nword* in the prefix-tree until the leaf node is reached (line 4 to line 15 in Algorithm 2). Obviously, the template of the syslog message is the same as that of the leaf node (line 16 in Algorithm 2). If the syslog message cannot match any template, a new template should be learned for it (line 12 to line 14 in Algorithm 2).

Matching a syslog message to its template using Craftsman is extremely efficient. Suppose that the height of the prefix-tree is H ($H \leq 10$ in general), the computational complexity of Algorithm 2 is $O(H \times k)$ (recall that typically $k < 10$). However, if we compare every word of a syslog message to all the words of each template, which is the method used in STE, the computational complexity will be $O(H \times \log H \times U)$, where U is the number of templates and usually $U \geq 300$. Moreover, the computational complexity of LogSimilarity in matching a syslog message to its template is $O(H \times U)$. Therefore, matching syslogs to templates using Craftsman is much more efficient than using STE or LogSimilarity.

Algorithm 2 Matching a Syslog Message to Its Template

Input: A prefix-tree T, and a syslog message M
Output: The template ID of M

```
1:  current_node = the root of T
2:  S = the list of words in M, which is sorted based on L
3:  current_word = the first word in S
4:  while current_node is not a leaf do
5:    for each child of current_node do
6:      if child.nword = current_word then
7:        current_node = child
8:        current_word = the next word in S
9:        break
10:     end if
11:   end for
12:   if current_node remains unchanged then
13:     return A new template should be generated for M
14:   end if
15: end while
16: return the template ID of the leaf node
```

Considering the large number (tens of millions in general) of syslogs generated per day in a large datacenter, an efficient template matching approach is really important. As demonstrated in Section V-B, compared to STE and LogSimilarity, Craftsman consumes much less computational resources in matching syslogs to templates in practice.

D. SPACE COMPLEXITY

We now discuss the space complexity of Craftsman. Obviously, Craftsman consumes the maximum memory when a prefix-tree is fully constructed (lines 9 in Algorithm I) before its children are eliminated (we refer to this prefix-tree as a full prefix-tree hereafter). This is because after the children are eliminated, the pruned prefix-tree that needed to be stored in memory has no node with too many ($> k$) children.

A full prefix-tree consists of two parts: a pruned prefix-tree which denotes the set of log templates, and many eliminated children representing the parameters of logs. In the worst case, each template denotes a separate root-to-leaf path of a pruned prefix-tree, and there are no two templates share the same node in the tree. Consequently, the pruned prefix-tree as well as the template set consume the same size of memory. Usually, the template set contains less than 10,000 templates, and each template has less than 20 words. That is, the memory consumed by the pruned prefix-tree is smaller than that consumed by $10,000 \times 20$ words. This is a small memory space considering today's server memory.

Apart from the pruned prefix-tree, we should also consider the memory consumed by those eliminated parameter words. Empirically, there are usually less than 10, 000, 000 parameter words in a full prefix-tree, which consumes a small memory space comparing with the total memory size of servers.

In conclusion, a full prefix-tree, which consumes the maximum memory in Craftsman's process of template learning, consumes a small memory space considering today's server memory. When Craftsman is applied to match logs to templates, it only has to store a pruned final prefix-tree in memory, which consumes much less memory than its corresponding full prefix-tree.

V. EVALUATION

In this section, we evaluate and compare the performance of Craftsman to those of SignatureTree, STE and LogSimilarity, which were proposed to parse syslogs for *network devices*, using syslogs collected from real-world sources. We evaluate the accuracy of these four techniques in Section V-A, and compare their efficiency in template matching in Section V-B. Then the advantage of incremental learning of Craftsman is demonstrated in Section V-C, followed by the evaluation of space complexity in Section V-D and the comparison of the failure prediction results using the four techniques to parse logs in Section V-E. In the end, we show Craftsman's generality in Section V-F.

In cooperation with a *a tier-1 cloud service provider*, we collect all syslogs over a two-year period from all 2,223 switches of a specific (switch) model across *more than 10* datacenters owned by this cloud service provider. Among the above switches, 131 ones suffered from 228 failures during that two-year-period.

The implementation of Craftsman is publicly available at [23], and we hope that researchers can better understand the performance of Craftsman using this implementation.

A. EVALUATION OF TEMPLATE LEARNING ACCURACY

As mentioned earlier, three techniques were proposed in previous works for parsing syslogs for network devices, *i.e.,* SignatureTree [8], STE [9], and LogSimilarity [4]. In addition, we propose a novel template extraction technique, *i.e.,* Craftsman, for accurately, efficiently, and incrementally learning templates from syslogs. After analyzing the four techniques, we believe that Craftsman and SignatureTree are more accurate in template extraction than STE and LogSimilarity in our scenario (see Section VI-A for more details). To demonstrate our analysis, we here compare the accuracy of the four techniques using real-world switch syslogs [27].

Learning templates from syslogs is equivalent to classifying syslogs based on the events they represent. Since the network operators analyze switch syslogs every day, they are really familiar with the event that a given syslog message represents. Therefore, we can use the manual classification results of syslogs by network operators as the ground truth. As aforementioned, we pick one switch model and *all* the switches of this switch model, and analyze *all* the syslogs of these switches. That is, billions of syslogs are analyzed for the above switches, and thus manually classifying *all* the syslogs is prohibitive. Thereby, we randomly collect a sample of the syslogs for the evaluation as follows. We first *randomly* pick four *message types* (see Table 1 for definitions) from *all* switches. For each message type, we *randomly* collect 500 syslogs. The network operators then manually classify the syslogs based on the event each message represents. Considering the large number of the syslogs that should be manually classified (2000), this is a large amount of work. After that, we run Craftsman, SignatureTree, STE, and LogSimilarity to learn the templates of these syslogs, respectively [27].

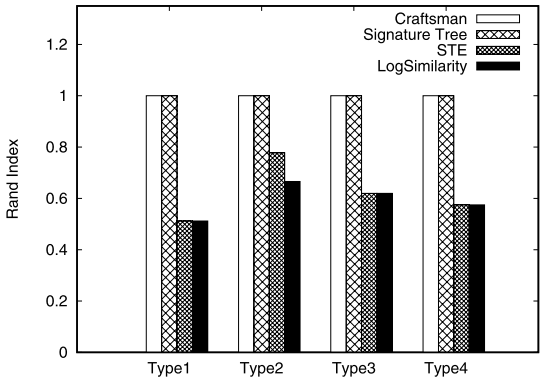


FIGURE 5. Comparison of the *rand indexes* of Craftsman, SignatureTree, STE and LogSimilarity across four message types.

We apply the *Rand index* [32] technique to quantitatively compare the accuracy of the four techniques. *Rand index* is a popular technique for evaluating the similarity between two data clustering techniques. We evaluate the accuracy of each technique by calculating the *Rand index* between the manual classification results and the templates learned by the technique. Specifically, among the template learning results of a specific technique for a given message type, we randomly select two messages, *i.e., x* and *y*, and define true positive (*tp*), true negative (*tn*), false positive (*fp*), and false negative (*fn*) as follows:

- *tp*: *x* and *y* are manually classified into the same cluster and they have the same template;
- *tn*: *x* and *y* are manually classified into different clusters and they have different templates;
- *fp*: *x* and *y* are manually classified into different clusters and they have the same template;
- *fn*: *x* and *y* are manually classified into the same cluster and they have different templates;

Then the *Rand index* can be calculated using the above terms as follows:

$$Rand\ index = \frac{tp + tn}{tp + tn + fp + fn} \tag{1}$$

[27]

Figure 5 shows the *Rand indexes* of Craftsman, SignatureTree, STE, and LogSimilarity across the above four message

TABLE 6. Comparison of the template matching time of Craftsman, SignatureTree, STE and LogSimilarity for 10 million syslogs.

Method	Craftsman	SignatureTree	STE	LogSimilarity
Time	10.34 *min*	10.34 *min*	116.3 *min*	81.5 *min*

types. Note that all the parameters of the four methods are set best for accuracy. For example, the pruning threshold of Craftsman is set as $k = 6$ through all the evaluation experiments in Section V. Both Craftsman and SignatureTree have averaged close-to-one *Rand indexes* and perform superior across all the four message types. In contrast, STE and LogSimilarity achieve relatively low averaged *Rand indexs*, *i.e.*, 62.10% and 59.31%, respectively. This is because STE cannot match some specific syslogs to any template, and LogSimilarity may match some specific syslogs of different *subtypes* to the same template (see Section VI-A for more details). Craftsman and SignatureTree are both word frequency based techniques, *i.e.,* both Craftsman and SignatureTree are constructed based on the frequency of words in messages. As a consequence, the templates extracted by Craftsman and SignatureTree are almost identical. However, as shown in Section V-C, Craftsman can construct a prefix-tree and learn templates incrementally, whereas SignatureTree cannot. [27]

B. EVALUATION OF TEMPLATE MATCHING EFFICIENCY

As mentioned in Section IV-C, theoretically, Craftsman and SignatureTree, both of which are with the prefix-tree structure, are much more efficient than STE and LogSimilarity in template matching. To demonstrate how Craftsman improves the efficiency of template matching (compared to STE and LogSimilarity) *in practice*, we use Craftsman, SignatureTree, STE, and LogSimilarity to match *randomly selected* 10 million historical syslogs, and compute how much time it takes to complete the template matching. We apply 10 million syslogs because a large cloud datacenter usually generates tens of millions of network device syslogs everyday, and thus tens of millions of syslogs have to match templates for anomaly detection, diagnosis or prediction. Note that matching historical syslogs does not need to retrain the model.

We implement Craftsman, SignatureTree, STE and LogSimilarity in C++, and deploy them on the same server (CPU information: 12 Intel(R) Xeon(R) CPU E5645 @ 2.40GHz) with a single thread. The CPU utilization remains 100% while the process of each technique is running so that we can use the total time to evaluate the computational efficiency of these log parsing methods [33].

Table 6 shows the time consumed for Craftsman, SignatureTree, STE, and LogSimilarity in matching 10 million historical syslogs, respectively. More specifically, compared to LogSimilarity and STE, Craftsman respectively improves the computational efficiency by 6.88 and 10.25 times. Thanks to the prefix-tree structure, Craftsman is extremely efficient in matching syslogs to templates in practice. Because SignatureTree also leverages the prefix-tree structure, it has the same template matching time with Craftsman. However, Craftsman is much more efficient in log parsing (including template learning and template matching) than SignatureTree, as described in Section V-C.

TABLE 7. Comparison of the log parsing (*i.e.,* template learning and matching) time per day for Craftsman, SignatureTree, STE and LogSimilarity.

Method	Craftsman	SignatureTree	STE	LogSimilarity
Time	12.4 *min*	2.516 *day*	23.59 *day*	97.8 *min*

C. EVALUATION OF INCREMENTAL LEARNING

As aforementioned in Section IV-B, Craftsman learns templates in an incremental manner, and thus when new *subtypes* of syslogs occur, it is not necessary that the template library has to be rebuilt and all the historical syslogs have to rematch the new templates. To demonstrate how Craftsman's incremental learning accelerate log parsing (*i.e.,* template learning and matching) in practice, we here compare the log parsing time of Craftsman, SignatureTree, LogSimilarity and STE. Specifically, we collect all the syslogs of the last day in the two-year period, and apply the above four methods to learn and match templates. The experimental setup is the same as that in Section V-B. Since it is often the case that new *subtypes* of syslogs can be generated everyday, the template library should be rebuilt, and all the historical syslogs should rematch the new templates for SignatureTree and STE per day, but not for Craftsman and LogSimilarity. [27]

Table 7 shows the time consumed for template learning and matching per day for Craftsman, SignatureTree, STE, and LogSimilarity. Both Craftsman and LogSimilarity are incrementally retrainable, and thus they can incrementally learn the template library, and match only *one-day's* worth of syslogs to the templates. However, both STE and SignatureTree have to rebuild the template library, and match all the historical syslogs, *i.e., two-year's* worth of syslogs to the templates. Hence Craftsman and LogSimilarity consume much less time compared to STE and SignatureTree. More specifically, Craftsman improves the computational efficiency by 730 times compared to SignatureTree. In addition, because STE is neither incrementally retrainable nor efficient in template matching, it consumes 6847 times more computational resources than Craftsman [27].

D. EVALUATION OF SPACE COMPLEXITY

As is discussed in Section IV-D, Craftsman is not only efficient in learning templates and matching logs to templates, but also consumes small memory space. To demonstrate the space complexity of Craftsman, we use the same dataset on the same server with that of the evaluation experiments in Section V-B. We monitor the memory consumed by each method in learning templates and matching logs to templates, and record the memory values every second.

Table 8 lists the maximum and average memory consumed by each method, respectively. We can see that Craftsman, Signature Tree, and LogSimilarity all consume a

TABLE 8. Comparison of the memory consumed by Craftsman, SignatureTree, STE and LogSimilarity for 10 million syslogs.

Method	Craftsman	SignatureTree	STE	LogSimilarity
Max.	121.1 MB	121.2 MB	38.6 GB	76.5 MB
Avg.	118.7 MB	118.7 MB	23.3 GB	62.1 MB

small memory space considering the large memory of modern servers. Specifically, they consumes about 60-120 *MB* memory, whereas a today's server usually has tens to hundreds of GB memory. Because for *each* log, STE has to store the position of *every* word as well as the log's length, it consumes much more memory than the other three methods. In addition, both Craftsman and Signature Tree use the same prefix-tree structure to learn and store templates, thus they have the same space complexity.

E. EVALUATION OF FAILURE PREDICTION ACCURACY

Since Craftsman, SignatureTree, STE and LogSimilarity are all log parsing techniques for failure diagnosis or prediction, we evaluate the four techniques based on not only template learning results, but also failure prediction results following [13], [14]. In the previous work [5], we proposed a syslog based failure prediction framework, PreFix, to determine whether a switch failure will occur in the near future during runtime. PreFix has been demonstrated with high accuracy in predicting switch failures using real-world data. In addition, HSMM is also a popular failure prediction technique used for predicting failures based on logs [16]. It was demonstrated with high accuracy using data collected from commercial cellular networks. Therefore, we apply PreFix and HSMM as the failure prediction techniques to demonstrate the log parsing performance of Craftsman, SignatureTree, STE, and LogSimilarity, and how they impact the performance of failure prediction. The parameters of PreFix and HSMM are set follows [5] and [16], respectively.

A system's capability to predict failure is usually assessed by the following three intuitive metrics: *Precision*, *Recall* and $F1$ *measure* [16], [34]. Hence we use these metrics to evaluate the performance of each technique. For a time bin, according to the ground truth provided by network operators, we know the outcome as either an ominous time bin or a non-ominous one (the definition of ominous time bins and non-ominous time bins follows [5]). For each technique, we label its outcome as a true positive (TP), true negative (TN), false positive (FP), and false negative (FN). True positives are ominous time bins that are accurately determined as such by the technique, and true negatives are time bins that are accurately determined as non-ominous. If the technique determines that a time bin is an ominous one when, in fact, it is actually non-ominous, we then label the outcome as a false positive. False negatives are ominous time bins that are incorrectly missed by the technique. We calculate the Precision, Recall and FPR as follows: $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F1\ mesure = \frac{2*Precision*Recall}{Precision+Recall}$ [27].

We use the *k-fold* cross validation model to evaluate the four techniques [35]. *k-fold* cross validation is a model validation technique that provides an insight on how a prediction model will generalize to an independent dataset [36]. In *k-fold* cross-validation, the time bins are randomly partitioned into k equal sized subsamples. In each cross-validation process, a single subsample is used as the validation data to test the prediction technique, and the remaining $k-1$ subsamples are used for training the technique. The cross-validation process is repeated k times so that each of the k subsamples is used exactly once as the validation data. We then average the k results to generate a single estimation. The benefit of *k-fold* cross-validation is that all time bins are used for both training and validation, and each time bin is used for validation exactly once. Since *10-fold* cross-validation is commonly used, we also apply it in our evaluation [35].

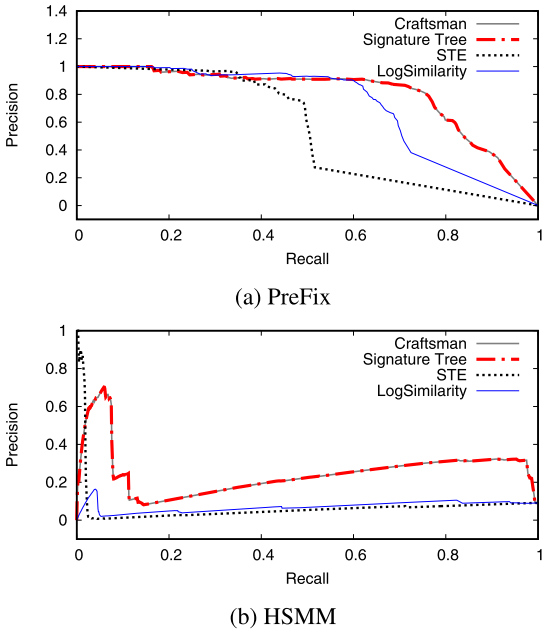


FIGURE 6. The PRCs of Craftsman, SignatureTree, STE, and LogSimilarity, using PreFix and HSMM as the failure prediction method, respectively.

Figure 6 shows the precision recall curves (PRCs) of the failure prediction results by PreFix and HSMM, using Craftsman, SignatureTree, STE, and LogSimilarity to parse logs, respectively. Because the templates extracted by Craftsman and SignatureTree are almost identical, they have the same PRCs. Through the PRCs, we can see that applying Craftsman for learning templates achieves significantly better failure prediction accuracy than applying STE and LogSimilarity, whether using PreFix or HSMM as the failure prediction system.

To intuitively compare the best accuracy of Craftsman, SignatureTree, STE, and LogSimilarity, Table 9 shows the *Precision*, *Recall* and $F1$ *measure* when the prediction systems (PreFix and HSMM) achieve the best $F1$ *measure*. When using PreFix to predict switch failures, Craftsman (as well as SignatureTree) achieves higher precision and higher recall than STE and LogSimilarity, respectively. As a result, Craftsman improves PreFix's prediction accuracy

TABLE 9. *Precision*, *Recall* and *F*1 *measure* of Craftsman, SignatureTree, STE and LogSimilarity.

Prediction method	Template learning technique	*Precision*	*Recall*	*F*1 *measure*
PreFix	Craftsman	87.35%	74.36%	80.33%
	SignatureTree	87.35%	74.36%	80.33%
	STE	74.69%	49.19%	59.32%
	LogSimilarity	78.44%	64.92%	71.04%
HSMM	Craftsman	32.27%	95.3%	48.21%
	SignatureTree	32.27%	95.3%	48.21%
	STE	9.14%	99.6%	16.75%
	LogSimilarity	10.67%	83.5%	18.93%

TABLE 10. Summary of the syslog datasets for demonstrating Craftsman's generality.

System	Description	#Logs	#Words per syslog	#Events
HDFS	Hadoop File System	11, 175, 629	8 ~ 29	29
BGL	BlueGene/L Supercomputer	4, 747, 963	10 ~ 102	376
HPC	High Performance Cluster	433, 490	6 ~ 104	105
Zookeeper	Distributed System Coordinator	74, 380	8 ~ 27	80
Proxifier	Proxy Client	10, 108	10 ~ 27	8

(*F*1 *measure*) by 13.07% (as compared to LogSimilarity) to 35.42% (as compared to STE). With HSMM as the failure prediction method, whereas the four techniques achieve approximate *Recalls*, Craftsman and SignatureTree improve the *Precision* of the failure prediction system by 202% (as compared to LogSimilarity) to 253% (as compared to STE) and the *F*1 *measure* by 155% (as compared to LogSimilarity) to 188% (as compared to STE).

F. EVALUATION OF GENERALITY

The above evaluation experiments have demonstrated Craftsman's superior performance in parsing *network device logs*. However, can Craftsman be applied in *other types of logs*, *e.g.*, logs of servers (supercomputers), distributed systems, applications? To study how general Craftsman is, we apply five large log datasets, ranging from supercomputers (Blue Gene/L and High Performance Cluster) to standalone software (Proxifier) to distributed systems (HDFS and Zookeeper). As concluded in [37], IPLoM [24], LKE [25] and LogSig [26] are three widely-employed log parsers in parsing logs for servers, distributed systems and applications (more detailed information can be seen in Section VI-B). Therefore, we compare Craftsman with the above three methods.

Table 10 summarizes the five large log datasets that are used to evaluate Craftsman's generality. These datasets have a total of 16,441,570 log messages. We collect the above five open log datasets with the generous support from their authors. Specifically, HDFS logs are collected from a node cluster of Amazon EC2 platform with 203 server nodes [38]. The dataset of BGL is collected from a BlueGene/L supercomputer system, which has 131,072 processors and 32,768 GB memory [39]. In addition, HPC is an open dataset collected from a high performance cluster of Los Alamos National Laboratory with 49 nodes (6,152 cores and 128 GB memory per node). The dataset of Zookeeper is collected from a Zookeeper installation on a node cluster with 32 server nodes, and the dataset of Proxifier is from a desktop software Proxifier [37]. All the above five open datasets have released the log message subtypes (see Table 3 for more details) which sketch out the events that the logs represent. We use these message subtypes (events) as the ground truth to evaluate the accuracy of each log parsing method.

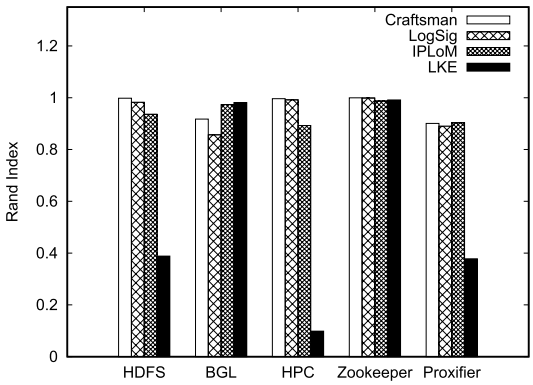


FIGURE 7. *Rand indexes* of Craftsman, LogSig, IPLoM, and LKE on the datasets of HDFS, BGL, HPC, Zookeeper and Proxifier.

As is with Section V-A, we use *Rand index* to quantitatively compare the accuracy of the above five log parsing methods. Please note that we run the experiments of LKE and LogSig 10 times to avoid bias (of clustering). Since the other methods are deterministic, we run them once. The implemented prototypes of Craftsman, LogSig, IPLoM, and LKE are running on the same experimental setup as described in Section V-B. In addition, as with the existing works [26], [37], from each dataset, we randomly pick 2000 log messages to evaluate each method's accuracy (as measured by *Rand index*), for the reason that training LKE and LogSig on large dataset consumes too much time. For example, it takes about 5.38 days to train LKE by parsing all the BGL data. That is, 10 times of log parsing will consume more than 53 days, which makes the training nearly infeasible.

Figure 7 shows the *Rand indexes* of Craftsman, LogSig, IPLoM, LKE on the datasets of HDFS, BGL, HPC, Zookeeper and Proxifier, respectively. Craftsman achieves an average *Rand indexes* of 96.25% on the five datasets, which is the highest among the five log parsing methods.

TABLE 11. The log parsing (including template learning and matching) time and memory cosumed per day of Craftsman, LogSig, IPLoM, and LKE on the BGL dataset.

Method	Craftsman	LogSig	IPLoM	LKE
Time	5.0 s	19.7 h	37.0 min	5.4 day
Max. Memory	32.4 MB	179.8 MB	90.6 MB	23.5 GB
Avg. Memory	30.6 MB	177.8 MB	85.8 MB	19.7 GB

This demonstrates that it can be used to parse logs not only for network devices, but also for servers (supercomputers), distributed systems and applications. Because LogSig, and LKE all rely on well-studied data mining models, rather than the heuristic rules extracted from the characteristics of log messages, they perform not good on one or more datasets. On the contrary, both Craftsman and IPLoM leverage the heuristic rules, and they achieve superior overall accuracy (the averaged *Rand indexes* of IPLoM is 93.86%). The comparison results are consistent with the conclusion in [37] that it is important for log parsing methods to exploit the special characteristics of log data.

Although LogSig, IPLoM, and LKE achieve relatively high accuracy on some dataset, none of these methods is incrementally retrainable. That is to say, if new subtypes of logs (events) occur (because of software updates, *etc.*), we have to rebuild the template library and rematch all the historical log messages to the newly generated templates. Take the BGL dataset as an example. The dataset contains the log data of 214 days. Because usually new subtypes of logs (events) occur everyday, the model has to be retrained and all the logs should rematch the new templates per day for LogSig, IPLoM, and LKE. However, Craftsman is incrementally retrainable, and thus only the newly added logs should rematch templates everyday. As shown in the second line of Table 11, it takes Craftsman much less time to log parsing (including template learning and matching) per day on the BGL dataset than LogSig, IPLoM, and LKE.

We also compare the memory consumed for the above four methods as shown in the third (maximum memory) and fourth (average memory) lines of Table 11. As is with Section V-D, we carefully monitor the memory consumed by each method when they are used to parse logs for the BGL dataset, and record the memory values per second. Craftsman consumes much less memory compared with LogSig, IPLoM and LKE on the BGL dataset. It demonstrates that Craftsman is efficient in space complexity for diverse types of logs.

In conclusion, because Craftsman leverages the heuristic rules extracted from the characteristics of log messages, and it is incrementally retrainable, it achieves superior performance not only on network device logs, but also on other types of log data. In other words, Craftsman is general enough to be applied to parse diverse types of log data.

VI. RELATED WORKS

Using log files for failure detection, diagnosis and detection has been widely applied in ISP networks [13], [16], computers [14], [17]–[21], [40], [41], and online ad services [42]. Liang *et al.* investigated the RAS event logs and developed three simple failure prediction techniques based on not only the characteristics of failure events, but also the correlation between failure events and non-failure events [40]. Realizing the the importance of the sequential feature of log files to failure prediction, Fronza *et al.* used random indexing (RI) to represent the sequence of operations extracted from logs, and then applied weighted support vector machine to associate sequences to the class of failures or that of non-failures [41]. Salfner *et al.* applied HSMM to recognize the patterns of logs that indicate an imminent failure directly [16].

A. LOG PARSING METHODS FOR NETWORK DEVICES

Syslog parsing techniques for *network devices including routers and switches* have been well studied in [4], [8], [9]. Specifically, inspired by the signature abstraction applied in spam detection, Qiu *et al.* proposed SignatureTree [8]. The idea behind this technique is that a syslog message *subtype* is usually a combination of words with high frequency. Therefore, for syslogs that belong to a given message type, the technique constructs a SignatureTree whose root node is the message type and the child nodes are arranged on the basis of the frequency of the *combinations* of words in syslogs. Using the frequency of the *combinations* of words rather than that of words themselves leads to that SignatureTree is not incrementally retrainable. That is, we have to retrain the SignatureTree, learn the templates, and match *all* the historical syslogs when new *subtypes* of syslogs occur (probably due to upgrades). Considering the large number of syslogs generated every day (tens of millions) and the long period of historical syslogs (two years), relearning the templates and rematching *all* the historical syslogs to templates does consume a huge amount of computational resources. Therefore, SignatureTree is not suitable for learning templates in our scenario [27].

In addition, Kimura *et al.* presented an STE approach that extracts log message templates using a statistical clustering algorithm [9]. The high level idea is that template words appear more frequently than parameter words, and that syslogs that belong to the same *subtype* usually have similar structures with the positions of words. Specifically, for a word w that appears in the x-th position of a syslog message that contains L words, the *word score* for w is $Score(w, x, L) = Probability(w|x, L)$. Then using clustering techniques, STE classifies words with high *word scores* into template words. However, STE can miss some templates, and thus some *subtypes* of syslogs cannot match any templates. For example, suppose that the syslogs that belong to *subtypes* $U_0, U_1, \ldots, U_n$ have the same number of words. If the syslogs that belong to U_0 occur much less often than those belonging to $U_1, U_2, \ldots, U_n$, each of the template words in U_0 will have a relatively small *word score*, and thus it will be classified into parameter words. Thereby, syslogs belonging to U_0 cannot match any template. As a result, as the evaluation experiments show in Sections V-A, STE has a relatively low accuracy. Moreover, STE is neither incrementally retrainable nor efficient in template matching [27].

To learn templates in an incremental manner, Kimura *et al.* [4] developed an online message template extraction technique, named LogSimilarity. In this technique, they first classified words into five classes on the basis of the tendency to constitute a log template: only symbols, only letters, only symbols and letters, only numbers and letters, and only numbers or numbers and symbols. When a new syslog message arrives, according to the number of words in different classes in the message, the technique will assign this message to an existing template cluster or create a new template cluster from this message. In this technique, message templates are learned on the basis of the *classes of words* rather than the *words themselves*, and thus syslogs that belong to different *subtypes* can be easily assigned to the same template cluster. For example, the syslogs of message type ''10OSPF/5/OSPF_NBR_CHG'' in Table 2 can be assigned to one or two template clusters in this technique, rather than the four clusters shown in Table 3 [27].

B. LOG PARSING METHODS FOR SERVERS (SUPERCOMPUTERS), DISTRIBUTED SYSTEMS AND APPLICATIONS

Data driven log parsing methods have been widely studied to automatically parse logs of servers, distributed systems and applications [24]–[26], [43]. These methods leverage data mining techniques to extract templates from log messages.

To the best of our knowledge, SLCT [43] is the first work aiming to automatically parse logs, and it has been widely applied in log mining tasks [44]. Motived by association rule mining, SLCT passes over logs twice with three steps: (1) word vocabulary construction, (2) cluster candidates construction, and (3) log template generation.

IPLoM [45] is designed based on the heuristic rules extracted from the characteristics of log messages. It has also been widely used in log mining studies [46]. IPLoM parses logs by hierarchically partitioning log messages through three steps: (1) partition by event size, (2) partition by token position, (3) partition by search for mapping, and (4) log template generation.

LKE [25] is a log parsing technique developed by Microsoft, which has been used in unstructured log mining [25]. It is designed based on both clustering models and heuristic rules. Similarly, it has three steps in parsing logs including (1) log clustering, (2) cluster splitting, and (3) log template generation.

LogSig [26] is a data mining based log parsing method that has been demonstrated in [47]. It parses logs through a three-step process: (1) word pair generation, (2) log clustering, and (3) log template generation.

None of the above methods is incrementally retrainable, and thus if they are applied to parse logs for a large volume of network device syslogs with a long period, they will consume too much computational resources. Consequently, they are not suitable in our scenario.

VII. CONCLUSION

In this paper, we propose a novel syslog parsing technique, Craftsman, to accurately, efficiently and incrementally learn templates for network devices in distributed systems. We evaluate and compare the performance of Craftsman to those of SignatureTree, STE and LogSimilarity using real-world network device syslogs collected from more than 10 datacenters over a two-year period. Both Craftsman and SignatureTree achieve much higher accuracy than STE and LogSimilarity. Our experiments also show that SignatureTree, STE and LogSimilarity consume much more computational resources than Craftsman, as they are either not incrementally retrainable, or inefficient in template matching. Moreover, we apply Craftsman to parse diverse types of logs, and Craftsman achieves good performance across all types of logs. In summary, the evaluation results clearly demonstrate the benefits of Craftsman: highly accurate, extremely efficient in template matching, incrementally retrainable and very general to diverse types of logs.

Although Craftsman is efficient and robust in log parsing, it has the following three limitations: (1) Although Craftsman fully uses the syntax information of logs, it does not understand or utilize the semantics information of logs or templates. (2) It cannot find similar templates and merge them to reduce the total number of templates. (3) It is trained and used for log parse on one certain type of logs (*e.g.,* switch logs), and thus cannot be used for cross-type log parse. Therefore, in the future we will apply the methods in natural language processing (say word embedding methods) to address the above three limitations.

APPENDIX
EXTENSIONS FROM THE CONFERENCE VERSION

A preliminary version of this submission has appeared in the following conference paper: Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun (Jim) Xu, Yu Chen, Hui Dong, Xianping Qu, Lei Song. Syslog Processing for Switch Failure Diagnosis and Prediction in Datacenter Networks. IEEE/ACM International Symposium on Quality of Service (IWQOS), 2017. This submission differs from the conference version as follows.

1) We introduce the term ''syslog parsing'', which includes not only the template learning as described in the conference version, but also the template matching as described in this version. In other words, Craftsman cares about both template learning and template matching, and the later one is newly added in this version.
2) We elaborate the intuition of Craftsman in Section III. After investigating thousands of real-world network device syslogs, we have the following two observations: (a) parameters words are much less than template words, and (b) a message type has a small number of subtypes, and each subtype has a large number of syslog messages. Based on these observations, we design

Craftsman to accurately, efficiently, and incrementally parse syslogs.

3) We introduce how to efficiently match a syslog message to its template in Section IV-C. The computational complexity of template matching using Craftsman is $O(H \times k)$, where H is the height of the Craftsman and usually $H \leq 10$, and k is the threshold of Craftsman pruning and usually $k \leq 10$. Therefore, matching a syslog message to its template using Craftsman is extremely efficient.
4) We leverage real-world data to evaluate Craftsman's efficiency in template matching in Section V-B. Specifically, we compare the running time of Craftsman in template matching to that of SignatureTree, STE, and LogSimilarity. The evaluation experiments demonstrate that Craftsman respectively improves the computational efficiency by 6.88 and 10.25 times compared to LogSimilarity and STE.
5) We theoretically demonstrate the space complexity of Craftsman in Section IV-D, and apply real-world logs to prove this in Section V-D. Craftsman consumes about $120MB$ memory when it is used to parse 10 million logs, which is quite a small memory space considering the large memory space of today's servers.
6) We leverage both PreFix and HSMM to show how Craftsman improves the accuracy of failure prediction in Section V-E. In addition to HSMM, we also apply PreFix, which was proposed by us recently, to demonstrate how Craftsman improves the performance of failure prediction.
7) We apply new datasets and new baseline methods to demonstrate Craftsmans's generality in Section V-F. To demonstrate how Craftsman is general to other types of logs, we apply it to parse five types of logs, and compare it with three widely-used log parsing methods designed for servers (supercomputers), distributed systems and applications.
8) We try our best to improve the presentation quality of this paper. The paper has been carefully revised thoroughly. For example, in Section III and Section IV we apply new examples of syslog messages to better demonstrate the intuition, the process of prefix-tree construction, and the benefit of incremental learning.

REFERENCES

[1] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 139–152.

[2] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. SIGCOMM*, 2011, pp. 350–361.

[3] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4. ACM, 2014, pp. 383–394.

[4] T. Kimura, A. Watanabe, T. Toyono, and K. Ishibashi, "Proactive failure detection learning generation patterns of large-scale network logs," in *Proc. IEEE 11th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2015, pp. 8–14.

[5] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, and Y. Zhang, "Prefix: Switch failure prediction in datacenter networks," in *Proc. ACM SIGMETRICS*, Irvine, CA, USA, Jun. 2018, pp. 1–29.

[6] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, Z. Zang, X. Jing, and M. Feng, "FUNNEL: Assessing software changes in Web-based services," *IEEE Trans. Services Comput.*, vol. 11, no. 1, pp. 34–48, Jan. 2018.

[7] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, and Z. Zang, "Rapid and robust impact assessment of software changes in large Internet-based services," in *Proc. CONEXT*, Heidelberg, Germany, Dec. 2015, pp. 1–13.

[8] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu, "What happened in my network: Mining network events from router syslogs," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2010, pp. 472–484.

[9] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto, "Spatio-temporal factorization of log data for understanding network events," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 610–618.

[10] X. Zhang, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, D. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, and Q. Cheng, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 807–817.

[11] W. Meng, Y. Liu, S. Zhang, D. Pei, H. Dong, L. Song, and X. Luo, "Device-agnostic log anomaly classification with partial labels," in *Proc. IEEE/ACM 26th Int. Symp. Quality Service (IWQoS)*, Jun. 2018, pp. 1–10.

[12] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, and P. Sun, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th Int. Joint Conf. Artif. Intell. (IJCAI), Int. Joint Conf. Artif. Intell. Org.*, vol. 7, 2019, pp. 4739–4745.

[13] F. Salfner and S. Tschirpke, "Error log processing for accurate failure prediction," in *Proc. 1st USENIX Conf. Anal. Syst. Logs (WASL)*, 2008, pp. 1–8.

[14] Z. Zheng, Z. Lan, B. H. Park, and A. Geist, "System log pre-processing to improve failure prediction," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2009, pp. 572–577.

[15] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proc. 41st Int. Conf. Softw. Eng., Softw. Eng. Pract.*, 2019, pp. 121–130.

[16] F. Salfner and M. Malek, "Using hidden semi-Markov models for effective online failure prediction," in *Proc. 26th IEEE Int. Symp. Reliable Distrib. Syst. (SRDS)*, Oct. 2007, pp. 161–174.

[17] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines," *WASL*, vol. 8, p. 5, Dec. 2008.

[18] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1285–1298.

[19] M. Du and F. Li, "Spell: Online streaming parsing of large unstructured system logs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2213–2227, Nov. 2019.

[20] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A real-time log analysis system," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 1052–1062.

[21] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang, "Automated it system failure prediction: A deep learning approach," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 1291–1300.

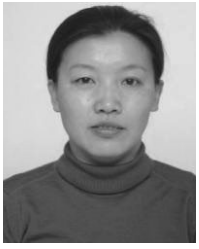
[22] A. A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the performance impact of upgrades in large operational networks," in *Proc. SIGCOMM*, New Delhi, India, Aug. 2010, pp. 303–314.

[23] *The Publicly Available Implementation of Craftsman*. [Online]. Available: https://github.com/slzhangsd/Craftsman

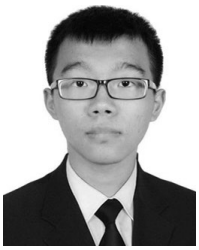
[24] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 1255–1264.

[25] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. 9th IEEE Int. Conf. Data Mining (ICDM)*, Dec. 2009, pp. 149–158.
[26] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, 2011, pp. 785–794.
[27] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, and X. Qu, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *Proc. IEEE/ACM 25th Int. Symp. Quality Service (IWQoS)*, Jun. 2017, pp. 1–10.
[28] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM*, Seattle, WA, USA, Aug. 2008, pp. 63–74.
[29] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: A field study of middlebox failures in datacenters," in *Proc. Conf. Internet Meas. Conf. (IMC)*, 2013, pp. 9–22.
[30] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, 2000.
[31] K. Sklower, "A tree-based packet routing table for Berkeley unix," in *Proc. USENIX Winter*, 1991, pp. 93–99.
[32] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *J. Amer. Stat. Assoc.*, vol. 66, no. 336, pp. 846–850, Dec. 1971.
[33] M. Yamada, A. Kimura, F. Naya, and H. Sawada, "Change-point detection with feature selection in high-dimensional time-series data," in *Proc. 23rd Int. Joint Conf. Artif. Intell.*, 2013, pp. 1827–1833.
[34] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, p. 10, 2010.
[35] G. McLachlan, K.-A. Do, and C. Ambroise, *Analyzing Microarray Gene Expression Data*, vol. 422. Hoboken, NJ, USA: Wiley, 2005.
[36] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. 14th Int. Joint Conf. Artif. Intell. (IJCAI)*, vol. 2, 1995, pp. 1137–1143.
[37] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2016, pp. 654–661.
[38] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 117–132.
[39] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 575–584.
[40] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, Jul. 2006, pp. 425–434.
[41] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, "Failure prediction based on log files using Random Indexing and Support Vector Machines," *J. Syst. Softw.*, vol. 86, no. 1, pp. 2–11, Jan. 2013.
[42] M. Shatnawi and M. Hefeeda, "Real-time failure prediction in online services," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1391–1399.
[43] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. 3rd IEEE Workshop IP Oper. Manage. (IPOM)*, Apr. 2003, pp. 119–126.
[44] R. Vaarandi, "Mining event logs with slct and loghound," in *Proc. Netw. Oper. Manage. Symp. (NOMS)*, 2008, pp. 1071–1074.
[45] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 1921–1936, Nov. 2012.
[46] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Fast entropy based alert detection in super computer logs," in *Proc. Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2010, pp. 52–58.
[47] L. Tang, T. Li, L. Shwartz, F. Pinel, and G. Y. Grabarnik, "An integrated framework for optimizing automatic monitoring systems in large it infrastructures," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 1249–1257.

SHENGLIN ZHANG (Member, IEEE) received the B.S. degree in network engineering from the School of Computer Science and Technology, Xidian University, Xi'an, China, in 2012, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2017. He is currently an Assistant Professor with the College of Software, Nankai University, Tianjin, China. His current research interests include failure detection, as well as diagnosis and prediction in data center networks.

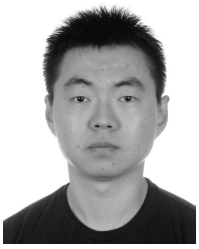


YING LIU (Member, IEEE) received the B.S. degree in information engineering, the M.S. degree in computer science, and the Ph.D. degree in applied mathematics from Xidian University, in 1995, 1998, and 2001, respectively. She made her postdoctoral research at the Department of Computer Science and Technology, Tsinghua University, from 2001 to 2003, where she is currently an Associate Professor with the Institute for Network Sciences and Cyberspace. Her research interests include multicast routing, network architecture, and router design and implementation.



WEIBIN MENG received the B.S. degree in software engineering from Jilin University, Changchun, China, in 2016. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, and the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. His research interests include anomaly detection, Syslog analysis, and failure prediction in datacenter networks.

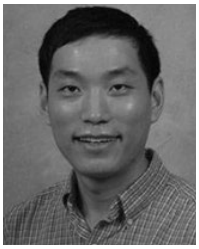
JIAHAO BU received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2016, where he is currently pursuing the master's degree with the Department of Computer Science and Technology, and the Institute for Network Sciences and Cyberspace. His research interests include deep generative model and semi-supervised learning in KPI anomaly detection.



SEN YANG received the B.S. degree in electronic engineering from Shanghai Jiao Tong University, in 2010, the M.S. degree in electronics and communication engineering from Shanghai Jiao Tong University, in 2013, the M.S. degree in electrical and computer engineering from the Georgia Institute of Technology, in 2013, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, in 2018. He is currently a Research Scientist at Facebook, Inc. His research interests include network management and scheduling in general.

YONGQIAN SUN received the B.S. degree in statistical speciality from Northwestern Polytechnical University, Xi'an, China, in 2012, and the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2018. He is currently an Assistant Professor with the College of Software, Nankai University, Tianjin, China. His research interests include anomaly detection, root cause localization, and high performance switching in the datacenter.

DAN PEI (Senior Member, IEEE) received the B.E. and M.S. degrees in computer science from the Department of Computer Science and Technology, Tsinghua University, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the Computer Science Department, University of California, Los Angeles (UCLA), in 2005. He is currently an Associate Professor with the Department of Computer Science and Technology, Tsinghua University. His research interest includes network and service management in general. He is a Senior Member of the ACM.



JUN (JIM) XU (Senior Member, IEEE) received the Ph.D. degree in computer and information science from The Ohio State University, in 2000. He is currently a Professor with the College of Computing, Georgia Institute of Technology. His current research interests include data streaming algorithms for the measurement and monitoring of computer networks and hardware algorithms, and data structures for high-speed routers. He received the US National Science Foundation (NSF) CAREER Award, in 2003, the ACM Sigmetrics Best Student Paper Award, in 2004, and the IBM Faculty Awards, in 2006 and 2008, respectively. He was named an ACM Distinguished Scientist, in 2010.

YUZHI ZHANG received the B.S. and M.S. degrees in computer science from the Department of Computer Science and Technology, Tsinghua University, in 1985 and 1987, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, in 1991. He is currently the Dean of the College of Software, Nankai University, and is also a distinguished professor. His research interests include deep learning and other aspects of artificial intelligence.

LEI SONG received the B.S. degree in information security from the School of Computer Science, Wuhan University, Wuhan, China, in 2009. He is currently a Senior Engineer with Baidu, Inc.

MING ZHANG received the B.S. degree in computer software from the School of Computer Science, Beijing University of Technology, Beijing, China, in 2001. He is currently an Engineer with China Construction Bank.