2018 IEEE International Symposium on Software Reliability Engineering Workshops

Machine Deserves Better Logging: A Log Enhancement Approach for Automatic Fault diagnosis

Tong Jia
Peking University
Beijing, China
jia.tong@pku.edu.cn

Ying Li
Peking University
Beijing, China
li.ying@pku.edu.cn

Chengbo Zhang
Peking University
Beijing, China
zhangcb@pku.edu.cn

Wensheng Xia
Peking University
Beijing, China
clement.xia@pku.edu.cn

Jie Jiang
Tencent Inc.
Shenzhen, China
zeus@tencent.com

Yuhong Liu
Tencent Inc.
Shenzhen, China
ehomeliu@tencent.com

***Abstract*—When systems fail, log data is often the most important information source for fault diagnosis. However, the performance of automatic fault diagnosis is limited by the ad-hoc nature of logs. The key problem is that existing developer-written logs are designed for humans rather than machines to automatically detect system anomalies. To improve the quality of logs for fault diagnosis, we propose a novel log enhancement approach which automatically identifies logging points that reflect anomalous behavior during system fault. We evaluate our approach on three popular software systems AcmeAir, HDFS and TensorFlow. Results show that it can significantly improve fault diagnosis accuracy by 50% on average compared to the developers' manually placed logging points.**

Keywords—Log enhancement, Automatic fault diagnosis, logging points

I. INTRODUCTION

When systems fail, log data is often the most important information source for administrators to diagnose faults. Administrators manually check log files to detect fault related log messages and infer what possible conditions might have led to failure. However, logs of distributed systems can be overwhelmingly large and failures may occur cross-component and cross-service, diagnosing faults of distributed systems via log messages is still notoriously difficult.

To solve this problem, automatic fault diagnosis models[2-6] are proposed. Log files are mined to learn graph-based models to capture normal request flows and then automatically detect anomalies on observing deviations. Despite the widespread use of log files in fault diagnosis, at its essence, the key problem is that existing developer-written logs are designed for humans rather than machines to automatically detect anomalies and diagnose faults. In many cases, logging statements are inserted into a piece of software in an ad-hoc fashion to address a singular problem, depending on developers' own experience or knowledge. Necessary information to infer the root causes of faults are frequently lost. The ad-hoc nature of logs seriously impact the performance of automatic fault diagnosis.

To improve logging quality, several research works have shown that developer-written logs can be improved either by including additional variable values[7] or placing error logging points at locations where error conditions may occur[8]. However, these works involve heavy human inference and their objectives are to enhance logging comprehensibility for developers to improve their logging experience. In this paper, we propose a novel automatic log enhancement approach, which aims to provide machine-written logs for improving automatic fault diagnosis, specifically, our approach can automatically find the least logging points that print logs exactly reflecting anomalies in automatic fault diagnosis model when failure occurs. Compared with existing log enhancement methods[7-13], our approach has two fundamental differences. First, the enhanced logs are written automatically by machines without involving any developer knowledge. Second, enhanced logs are written specifically for machines to perform automatic fault diagnosis.

We applied our approach on three popular software systems: AcmeAir, HDFS and Tensorflow. Compared to these software's original developer-written logging points, our approach significantly improves the accuracy of fault diagnosis. For example, in experiment of HDFS, our approach improves 58.3% accuracy. Besides, it improves 19.2% accuracy of fault diagnosis for Tensorflow. We also demonstrate that enhanced logging points generated by our approach brings little overhead.

The rest of this paper is organized as follows. Section 2 describes the preliminaries. Section 3 and section 4 present the details of the proposed approach. Section 5 discusses experiment and evaluation. Section 6 discusses the related work. Section 7 is the conclusion and future work.

II. PRELIMINARIES

A. Automatic Fault Diagnosis Model

To better illustrate the approach, we briefly introduce graph-based automatic fault diagnosis model. Note that different research works may generate and name the model differently, the fundamental structure and fault diagnosis process are the same. Here we choose *time-weighted control flow graph(TCFG)* as a representation[5][6]. TCFG is a directed graph consisting of nodes, edges and time weights that captures normal request flows. Nodes are *logging points* and edges represent transition flows between nodes. A *logging point* is an abstraction of a logging statement in source code as summarization of multiple raw log lines. A time weight is added on each edge to record the transition time between two adjacent nodes.

B. Log Enhancement Problem

Before we propose the definition of our problem, two questions need to be answered.

978-1-5386-9443-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ISSREW.2018.00-22

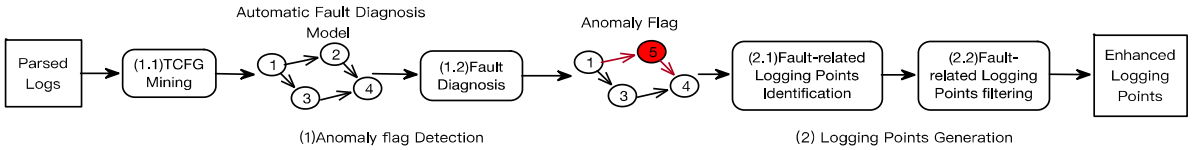

Fig. 1: Log Enhancement Approach

1) What are better logs for automatic fault diagnosis. According to our fault diagnosis practices, automatic fault diagnosis can be benefited by logs that have strong ability to flag anomalies in TCFG during system falure time. Besides, these logs certainly should not incur a high performance overhead to the system.

2) Where to log or what to log. As mentioned before, TCFGs are designed to capture request flows, thus the location of logging points in source code is more important than detailed content of logs. Therefore, *where to log* is the most important factor that affects the effectiveness of automatic fault diagnosis models.

Therefore, log enhancement for fault diagnosis problem is to generate logging points that mostly reflect anomalies in TCFGs. Given a trained TCFG model $G=\{T|R,P\}$, where R and P denote a set of rules and parameters while T is a set of log templates under the management of R and P. Given a system fault f, if model G is able to diagnose f, we denote $G(f)=1$, otherwise, $G(f)=0$. The log enhancement problem is formulated as generating a subset of log templates $G' \subseteq G$ that satisfying the following equation:

$$G' = argmin(N(G'))argmax(G'(F)) \quad (1)$$

Where $N(G')$ denotes the number of log templates in G' and F is a set of different faults: $F=\{f_1,f_2,\ldots,f_n\}$. Now, we present the basic idea of our approach to solve this optimization problem. When failure occurs, deviations from the generated TCFG are identified as anomaly flags to help identify the root cause of anomalies. These deviations are in fact sub-structures of the model including logging points. Since these logging points are able to print logs that show anomalies through the TCFG model during failures occurs, they are especially important for automatic fault diagnosis. To solve the optimization problem and generate G', we further propose a filtering algorithm based on Kullback–Leibler divergence[14]. The algorithm is able to find the least logging points that behave anomalously due to most faults.

III. Log Enhancemen Approach

Our log enhancement approach aims to find the least logging points that show most anomalies through the TCFG model during system failure. As shown in Fig. 1, the approach is composed of Anomaly Flag Detection and Logging Points Generation. Anomaly Flag Detection first builds a graph-based model with parsed logs (logging points corresponding to each log) from a normal running system, then if system failures occur, it compares the input log stream with the model and finds the anomaly flags in the model. Logging Points Generation records these flags and identifies fault-related logging points inside these flags.

A. Anomaly Flag Detection

1) TCFG mining

TCFG mining includes node mining, edge mining and time weight mining. Since nodes in TCFG are logging points extracted from source code or typical log template mining algorithms[1], here we describe edge mining and time weight mining. TCFG edge mining aims to mine the transitions between logging points in each request or transaction flow from log stream. We assume that each log records a request ID. A request ID is a numerical string to specify a certain request. Logs with the same request ID can be correlated together to reflect the execution path of a request. For each pair of adjacent logs with the same request ID, we add an edge from the template of prior one to the template of post one. Note that without request ID, edge mining is also available in our prior work [5][6] . A time weight on each edge in the TCFG denotes the transition time between two templates, that is, the overall execution time between two templates. This time weight can be utilized for latency problem diagnosis. For instance, if the time between two consistent logs in a transaction exceeds the time weight recorded in the TCFG, a latency problem is reported and located. For each two adjacent logging points, the time weight is recorded as the maximum time difference in all pairs of occurrences in log stream. Therefore, if the execution time between two occurrences of adjacent logging points is larger than the time weight, it denotes that execution latency is larger than normal status.

2) Anomaly flag detection

From the perspective of fault diagnosis, detailed anomaly is helpful for administrators to analyze system faults. There are three types of anomalies: *sequence anomaly*, *redundancy anomaly* and *latency anomaly*. A sequence anomaly is raised when the log that follows the occurrence of a parent node cannot be mapped to any of its children. A redundancy anomaly is raised when unexpected logs occur that cannot be mapped to any node in the temporal path of the TCFG. An unexpected log can be obvious abnormal log that cannot be matched to any template or a redundant occurrence of a log template. A latency anomaly is raised when the child of a parent node is seen but the interval time exceeds the time weight recorded on the edge. When anomalies are found, we flag a sub-structure of TCFG as the anomaly flag based on the anomaly types for administrators to analyze root causes. For sequence anomaly, we flag the minimal sub-tree starting with the parent node as well as the undesirable child node. For redundancy anomaly, we flag the unexpected node and its parent node according to the abnormal log stream. For latency anomaly, we flag the two adjacent nodes whose time interval exceeds the time weight recorded on the edge. Fig. 2 shows the example of anomaly flags of different types of anomalies. Fig. 2 (a) is an example of TCFG with 7 nodes. Imagine node 1 and node 2 suffer a latency anomaly, we record node 1, node 2 and the edge between them as the anomaly flag shown in

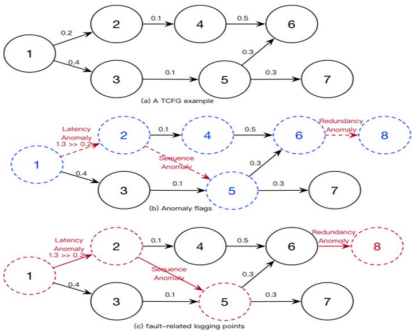

Fig. 2: Example of fault diagnosis and fault-related logging points identification

Fig. 2 (b). Meanwhile, node 5 appears after node 2 unexpectedly and suffers a sequence anomaly, we record the minimal sub-tree starting with node 2 (node 2, node 4 and the medial edge) and node 5 with the unexpected edge from node 2 to node 5 as the anomaly flag. After that, node 8 appears after node 6 while node 8 is not in the TCFG, thus a redundancy anomaly occurs. We record node 6 and node 8 as the anomaly flag correspondingly.

B. Logging Points Generation

1) Fault-related logging points identification

Our log enhancement goal is to identify logging points that can indicate anomaly behaviors as well as to reduce the number of logging points to as few as possible for controlling overhead. To achieve this, we define three rules for each type of anomaly to identify fault-related logging points from each anomaly flags. For sequence anomaly, we choose the parent node with its unexpected child in the anomaly flag as fault-related logging points. The intuition behind is that no matter how the TCFG changes, this unexpected child will always appear after the parent node because of the same fault. Meanwhile, these two nodes constitute the minimal structure of the anomaly flag. For redundancy anomaly, we choose only the redundant node as the fault-related logging point. The intuition behind is that normally the redundant node is not in the execution path of the request, so it indicates faults. Therefore, the variance of this redundant node is capable of identifying the fault. For latency anomaly, we choose the two adjacent nodes in the anomaly flag as the fault-related logging points. Fig. 2(c) shows the example of the identified logging points. For latency anomaly, we record node 1 and node 2. For sequence anomaly, we record the parent node 2 with its unexpected child node 5. For redundancy anomaly, we record node 8 only.

2) Fault-related logging points filtering

To further reduce the number of fault-related logging points, we propose a filtering method based on information gain algorithm. Our objective is to find the least number of logging points that can distinguish different types of fault as many as possible.

Let us denote all the system faults as $F = \{f_1, f_2, f_3, ..., f_m\}$ where m is the number of faults. The anomaly flags are formulated as $T = \{t_1, t_2, t_3, ..., t_n\}$ where n denotes the total number of identified logging points. We define an $m \times n$ matrix $\mathcal{M}$ to record the relevance between faults and anomaly flags. Element in $\mathcal{M}$ is formulated as a_{ij} where $1 \le i \le m$ and $1 \le j \le n$. a_{ij} is assigned to 1 if anomaly flag t_j is caused by fault f_i, otherwise, it is assigned to 0:

$$a_{ij} = \begin{cases} 1 & if\ t_j\ is\ caused\ by\ f_i \\ 0 & otherwise \end{cases} \quad (2)$$

The input of the fault-related logging points filtering algorithm is an attribute-valued dataset D composed of $\mathcal{M}$ and F where F is denoted as the label vector of $\mathcal{M}$:

$$D = \begin{bmatrix} a_{11} & \cdots & a_{1n} & f_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mn} & f_m \end{bmatrix} \quad (3)$$

The output is a set of filtered anomaly flags T' satisfying $T' \subseteq T$. The information gain algorithm is used to solve this problem. Information gain is the change of information entropy *Info(D)* and conditional entropy $Info_A(D)$ on the condition of A.

$$Gain(A) = Info(D) - Info_A(D) \quad (4)$$

where condition A is a selected anomaly flag t_x. For each instance in D, value of a_{ix} can only be assigned to *0* or *1*. Let us denote the number of instances where a_{ix} is assigned to 1 as m_1, and the number of instances where a_{ix} is assigned to *0* as m_0 where $m_0+m_1=m$. *Info(D)* and $Info_A(D)$ is computed as follows:

$$Info(\mathrm{D}) = -\sum_{k=1}^{m} \frac{1}{m} log_2(\frac{1}{m}) \quad (5)$$

$$Info_A(\mathrm{D}) = -\sum_{k=0}^{1} \frac{m_k}{m} log_2(\frac{m_k}{m}) \times \mathrm{Info}(D_j) \quad (6)$$

As shown in algorithm 1, our method is recursive. In each recursion, we compute the $Gain(t_x)$ for each t in T and then select the best feature t_{best} with the most information gain for current dataset. The value of t_{best} divides D into two subsets D_x and D_y, the algorithm then enters the next recursion to calculate the best feature of D_x and D_y. At last, we record all the t_{best} selected in every recursion as the filtered anomaly flags and the fault-related logging points in them are recorded as filtered fault-related logging points.

Algorithm 1 fault-related logging points filtering

Input: *An attribute-valued dataset D where Abnormal Pattern Sets(APS)as attributes and Multiple Injected Faults(MIF) as value*
Output: *Filtered Abnormal Pattern Set(FAPS)*

```
if D is pure or attributes of samples in D are the same:
    terminate
for all attribute a ∈ D:
    Compute the information gain if we split on a
a_best = Best attribute with the most information gain
FAPS.add(a_best)
D_u = Two induced sub-datasets from D based on a_best
for all D_k in D_y:
    FLPF(D_k)
```

IV. Experiment and Evaluation

To evaluate our approach, we have conducted a set of experiments with three popular open source projects: AcmeAir[1], HDFS[2] and TensorFlow[3].

A. Evaluation Setting Up

1) Evaluation methodology: To better evaluate our approach, we design two comparative experiments as follows.

1 https://github.com/acmeair/acmeair-nodejs
2 http://hadoop.apache.org/
3 https://www.tensorflow.org/

Machine-written logs vs. developer-written logs: The first experiment aims to evaluate the effectiveness of the proposed approach. It compares the performance of enhanced logging points with original developer-written logging points for automatic fault diagnosis.

Identification phase vs. filtering phase: The second experiment aims to evaluate the effectiveness of two phases of the proposed approach, i.e., (2.1) Identification phase and (2.2) Filtering phase.

2) Experiment design and log collection: For each testing software, we develop four versions of source code and deploy them seperately into testing system. Each version of source code is deployed with five docker containers. During running, we simulate requests to the system continuouslly in order to collect normal system logs. After that, eleven types of faults are randomly injected. The whole process is experimented three times. Details are shown in Fig. 3 and the four code versions are discussed as follows.

a) Original Source Code: A popular and stable version of source code we download from the software project repository.

b) Log-fullfilled Source Code: Literally, this version of source code is fullfilled with logging statements. We insert a logging statement recording the program file name and the position of the logging statement(specifically line numbers) after every executable statements in the original source code.

c) Log-enhanced(Identification) Source Code: After log-fullfilled source code is deployed to the system, various running logs are collected as the input to the proposed log enhancement approach. Through Fault-related Logging Points Identification phase, multiple logging points are identified. These logging points are then fed back to the original source code by inserting a logging statement to each corresponding program location.

d) Log-enhanced(Filtering) Source Code: Similar with log-enhanced (Identification) source code, log-enhanced (Filtering) source code is the revised original source code by inserting logging statements corresponding to the output logging points of Fault-related Logging Points Filtering phase.

3) Injected faults design: According to [15],a fault is active when it causes an error, otherwise it is dormant. A service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is an error. For fault injection, we want to simulate multiple faults that results in service failures.

To compare the performance of our approach when applied to different projects, we need to design same faults for different projects. Therefore, fine-grained and specific faults can not be used in our testbed. We design eleven general and popular system faults in three layers. For software layer, services of the software are randomly aborted. For network layer, four types of faults including partition, slow, duplicate and flaky network are inserted.

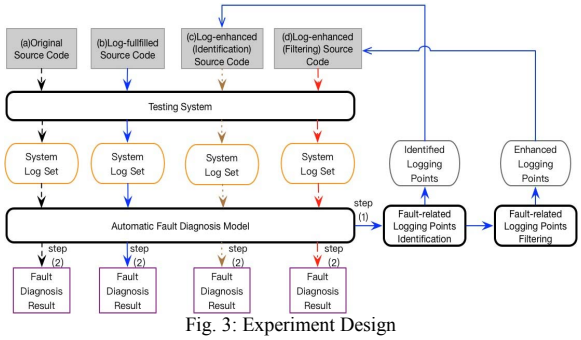

Fig. 3: Experiment Design

For OS layer, six types of faults including block port, burn cpu, burn I/O, fill disk, null route and kill container are experimented.

Faults are injected in two experiment steps including (1)generating enhanced logging points through the proposed approach with log-fullfilled source code and (2)evaluating fault diagnosis performance of four code versions(Fig. 3). Note that different faults are randomly injected into the two steps independently, so as to test if enhanced logging points can handle new incoming faults.

4) Basedlines: In machine-written logs vs. developer-written logs experiment, we take original source code as the baseline. In identification phase vs. filtering phase experiment, log-enhanced(Filtering) source code is compared with log-enhanced(Identification) source code. In addition, log-fullfilled source code is another baseline to evaluate the effectiveness of fault-related logging points identification and filtering phases.

5) Evaluation metrics: We use *Fault Diagnosis Accuracy(FDA)* and *Time Cost(TC)* to evaluate the proposed approach. Fault Diagnosis Accuracy is the percentage of faults successfully diagnosed by logs in all injected faults, as shown in Equation 7. It aims to evaluate the effectiveness of enhanced logging points in terms of diagnosing faults.

$$FDA = \frac{\#number\ of\ successfully\ diagnosed\ faults}{\#number\ of\ all\ injected\ faults} \quad (7)$$

Time Cost is the average time to successfully execute a job on Hadoop/TensorFlow or response to a user request by Acme-Air services, as shown in Equation 8. It aims to evaluate the extra overhead that enhanced logging points bring to the system.

$$TC = \frac{\#system\ normal\ running\ time}{\#number\ of\ jobs\ or\ requests\ the\ system\ has\ handled} \quad (8)$$

B. Machine-written Logs vs. Developer-written Logs

In this section, we compare enhanced logging points with real world original developer-written logging points from various aspects.

Fig. 4 compares three statistics of anomaly flags, anomaly logging points and total logging points. Results show that the proposed approach generates much less logging points than original source code, but on the contrary, these logging points formulate much more anomaly flags during system fault. In the experiment of AcmeAir, the proposed approach generates 19 enhanced logging points while original source code contains 35 logging points. After deploying the two versions of code

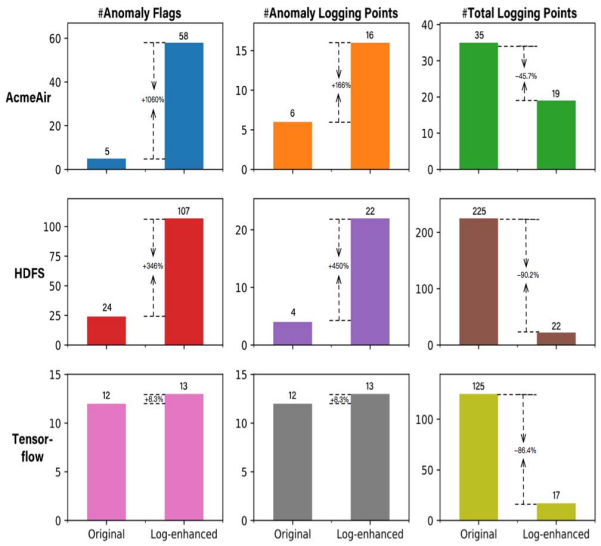

Fig. 4: Comparison on logging points of machine-written logs vs. developer-written logs

into the testing environment, and feeding the output logs into automatic fault diagnosis model, we pleasantly surprised that 19 enhanced logging points can formulate 58 anomaly flags including 16 anomaly logging points. On the contrary, 35 original logging points only formulate 5 anomaly flags including 6 anomaly logging points. Similar results have been shown in HDFS and Tensorflow.

Fig. 5 compares fault diagnosis accuracy. Results show that our approach greatly improves the performance of automatic fault diagnosis models, increasing over 50% FDA on average. In the experiment of HDFS, 4 of 24 injected faults are diagnosed through logs of original source code, achieving 16.7% FDA, while 18 faults are successfully diagnosed through logs of log-enhanced(filtering) source code, achieving 75% FDA. As for Tensorflow, 6 of 26 injected faults can be diagnosed through logs of original source code, achieving 23.1% FDA, while 11 of 26 faults are successfully diagnosed through logs of log-enhanced(filtering) source code, achieving 42.3% FDA. Fig. 6 compares the time cost metric. Results show that enhanced logging points bring little extra overhead to the system. Time cost of log-enhanced(filtering) source code increases 5.6%, 12.1% and 4.3% on average in the experiments of three projects.

C. *Identification Phase vs. Filtering Phase*

In this section, we compare enhanced logging points with intermediate results of the proposed approach to evaluate the effectiveness of each phase. Similar with machine-written logs vs. developer-written logs experiment, we first compare three statistics of anomaly flags, anomaly logging points and total logging points. Then, we present the results of two evaluation metrics.

Fig. 7 compares three statistics of log-fullfilled source code, log-enhanced(identification) source code and log-enhanced(filtering) source code. Results show that both fault-related logging points identification phase and filtering phase largely reduce the total number of logging points, meanwhile, enhanced logging points therewith formulate less anomaly flags during system fault.

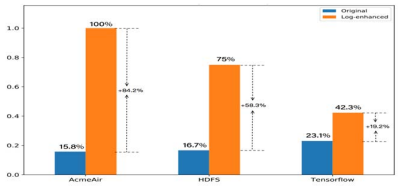

Fig. 5: Comparison on FDA of machine-written logs vs. developer-written logs

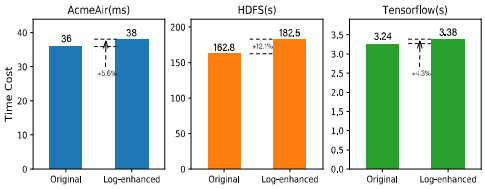

Fig. 6: Comparison on TC of machine-written logs vs. developer-written logs

Specifically, compared with log-fullfilled source code, identification phase reduces over 80% of total logging points, but formulates almost as many anomaly flags and anomaly logging points as log-fullfilled source code. Filtering phase further reduces over 90% of total logging points compared with log-enhanced(identification) source code.

Fig. 8 compares fault diagnosis accuracy of the three source code versions. Results show that log-enhanced(identification) source code performs 100% FDA in all testing projects. Log-enhanced(filtering) source code performs worse FDA than other code versions. Besides, with the growing complexity of software projects, fault diagnosis results of log-enhanced(filtering) source code are getting worse. For the simplest project AcmeAir, log-enhanced(filtering) source code performs 100% FDA. For HDFS, it performs 75% FDA instead. As for the most complex project Tensorflow, it performs only 42.3% FDA. This is because filtering phase largely reduces logging points, and thus reduces many anomaly flags. If a project is complex, even same fault may affect different processes, thus may trigger different anomaly flags. After filtering phase, most anomaly logging points are filtered, and the reserved logging points may not able to show anomalies according to the temporal running status of the testing system.

Fig. 9 compares the time cost. Results show that enhanced logging points reduce much overhead. In the experiment of HDFS, TC of log-enhanced(identification) source code is about 3.7%(7.5s) less than log-fullfilled source code. Furthermore, TC of log-enhanced(filtering) source code is 6.4%(12.4s) less than log-enhanced(identification) source code. As for Tensorflow, TC of log-enhanced(identification) source code is almost equal to the TC of log-fullfilled source code. This is because despite that we insert thousands of logging points to the original source code, most of them are not called during the normal execution of a training task. For instance, saving temporal parameter status and recovering preloaded workspace will never be executed during a training task. The time cost of log-enhanced(filtering) source code is 48.6%(3.2s) less than log-enhanced(identification) source code.

V. Related Work

A. *Automatic Fault Diagnosis*

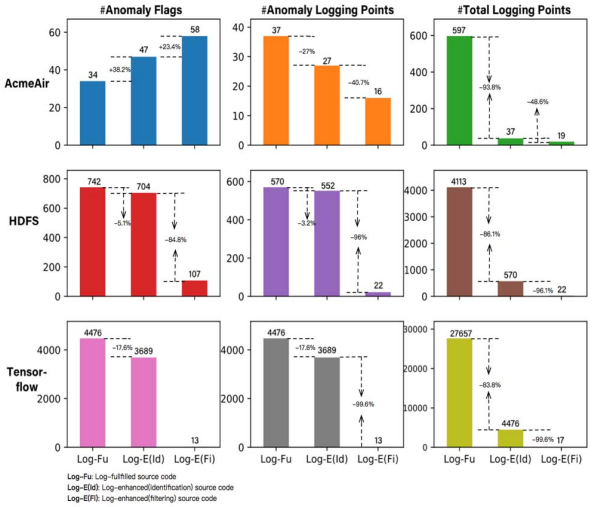

Fig. 7: Comparison on statistics of identification phase vs. filtering phase

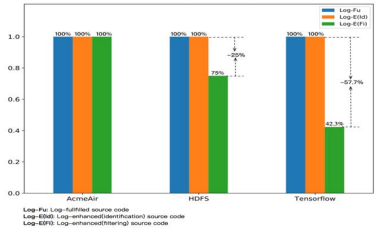

Fig. 8: Comparison on FDA of Identification Phase vs. Filtering Phase

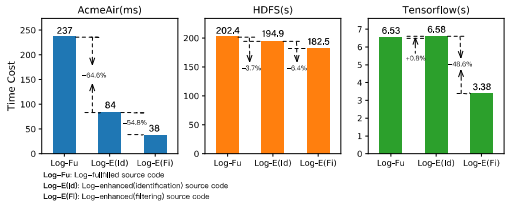

Fig. 9: Comparison on time cost of identification phase vs. filtering phase

Prior works on fault diagnosis via logs can be divided into machine learning based models and graph-based models. Graph-based model ([2-6]) aims to propose a descriptive model of service execution path through logs. This model has three advantages compared with other models: 1) it can diagnose problems that deeply buried in log sequences such as performance degradation, 2) it can provide the context log messages of problems, 3) it can provide the correct log sequence and tell engineers what should have happened.

B. Log Enhancement

In order to improve the quality of logging, Yuan et al. provide three tools: LogEnhancer[7], ErrLog[8] and Log20[10]. LogEnhancer is a static analysis tool that can automatically identify and insert critical variable values into the existing logging statements. ErrLog manually analyzes the source code to find appropriate rules as a logging guidance. Log20 generates logging points that mostly identify different program paths. Zhu et al.[9] propose a tool named LogAdvisor. LogAdvisor leverages machine learning to automatically predict whether a logging statement should be added to a code snippet. Besides, other log enhancement work[11][12][13] also make analysis on logging practices. These works either aim to learn human logging practices, or leverage program features.

VI. Conclusion and Future Work

In this paper, we propose an automatic log enhancement approach. To the best of our knowledge, this work is the first attempt of machine-designed logs for machines to automatic fault diagnosis. Results show that our approach performs over 50% increasing of fault diagnosis accuracy compared with developer-written logs in three popular software. However, in our experiment, fault injection is limited to fault coverage inherently. It is hard to implement all types of fault and inject them into target system. In the future, we consider of applying our approach to real-world systems.

Acknowledgment

This work is supported by PKU-Tencent Joint Lab Research Program.

References

[1] R. Varandi, M. Pihelgas, "LogCluster: A data clustering and pattern mining algorithm for event logs," International Conference on Network and Service Management, Barcelona, 2015, pp. 1-7.

[2] Q. Fu, J. G.Lou, Y. Wang, et al., "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," IEEE International Conference on Data Mining, Miami, 2009, pp. 149-158.

[3] W. V. derAalst, T. Weijters, and L. Maruster, "Workflow mining: Fdis Discovering process models from event logs," IEEE Transactions on Knowledge and Data Engineering, vol. 16, pp. 1128-1142, Sept. 2004.

[4] J. G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces, " Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, Washington, 2010, pp. 613-622.

[5] Tong Jia, et al., "LogSed: Anomaly Diagnosis through Mining Time-Weighted Control Flow Graph in Logs," International Conference on Cloud Computing IEEE, Honolulu, 2017, pp. 447-455.

[6] Tong Jia, et al., "An Approach for Anomaly Diagnosis Based on Hybrid Graph Model with Logs for Distributed Services," IEEE International Conference on Web Services IEEE, Honolulu, 2017, pp. 25-32.

[7] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. "Improving software diagnosability via log enhancement," ACM Transactions on Computer Systems (TOCS), vol. 30, pp. 4, Feb. 2012.

[8] D. Yuan, S. Park, P. Huang, et al., "Be conservative: enhancing failure diagnosis with proactive logging," Usenix Conference on Operating Systems Design and Implementation USENIX Association, vol. 12, pp. 293-306, Oct. 2012.

[9] J. Zhu, P. He, Q. Fu, et al., "Learning to log: helping developers make informed logging decisions," Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 415-425, May 2015.

[10] X. Zhao, K. Rodrigues, Y. Luo, et al., "Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold," Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, 2017, pp. 565-581.

[11] Fu, Qiang, et al. "Where do developers log? an empirical study on logging practices in industry." International Conference on Software Engineering 2014:24-33.

[12] H. Li, W. Shang, Y. Zou, et al., "Towards just-in-time suggestions for log changes." Empirical Software Engineering, vol. 22, pp. 1831-1865, Aug. 2017.

[13] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012, pp. 102-112.

[14] Kullback S, Leibler R A. On Information and Sufficiency[J]. Annals of Mathematical Statistics, 1951, 22(1):79-86.

[15] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 1, pp. 11–33, 2004.