2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00133

AutoLog: A Log Sequence Synthesis Framework for Anomaly Detection

Yintong Huo*†, Yichen Li*†, Yuxin Su‡**, Pinjia He§, Zifan Xie¶, and Michael R. Lyu†
†Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China.
Email: {ythuo, ycli21, lyu}@cse.cuhk.edu.hk
‡School of Software Engineering, Sun Yat-sen University, Zhuhai, China. Email: suyx35@mail.sysu.edu.cn
§School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK Shenzhen), China.
Email: hepinjia@cuhk.edu.cn
¶Huazhong University of Science and Technology, Wuhan, China. Email: xzff@hust.edu.cn

***Abstract*—The rapid progress of modern computing systems has led to a growing interest in informative run-time logs. Various log-based anomaly detection techniques have been proposed to ensure software reliability. However, their implementation in the industry has been limited due to the lack of high-quality public log resources as training datasets.**

While some log datasets are available for anomaly detection, they suffer from limitations in (1) comprehensiveness of log events; (2) scalability over diverse systems; and (3) flexibility of log utility. To address these limitations, we propose AUTOLOG, the first automated log generation methodology for anomaly detection. AUTOLOG uses program analysis to generate run-time log sequences without actually running the system. AUTOLOG starts with probing comprehensive logging statements associated with the call graphs of an application. Then, it constructs execution graphs for each method after pruning the call graphs to find log-related execution paths in a scalable manner. Finally, AUTOLOG propagates the anomaly label to each acquired execution path based on human knowledge. It generates flexible log sequences by walking along the log execution paths with controllable parameters. Experiments on 50 popular Java projects show that AUTOLOG acquires significantly more (*9x-58x*) log events than existing log datasets from the same system, and generates log messages much faster (*15x*) with a single machine than existing passive data collection approaches. AUTOLOG also provides hyper-parameters to adjust the data size, anomaly rate, and component indicator for simulating different real-world scenarios. We further demonstrate AUTOLOG's practicality by showing that AUTOLOG enables log-based anomaly detectors to achieve better performance (*1.93%*) compared to existing log datasets. We hope AUTOLOG can facilitate the benchmarking and adoption of automated log analysis techniques.

I. INTRODUCTION

The ever-growing amount and quality of data cultivate the development of advanced data-driven approaches, showing great power in many fields. Considering the dataset evolving from the simple handwritten digit dataset MNIST [1] to the large visual database ImageNet [2], the rich data foster more sophisticated and applicable image algorithms against the complicated real-world scenarios.

The log analysis community has proposed numerous techniques to assist maintainers in automatically inspecting the large log files produced by modern complex systems. The techniques include detecting internal system anomalies [3]–[7], and suggesting the root cause of an anomaly [8]–[12] for employing the corresponding mitigation strategy. Despite the continued development of automated solutions for software maintenance through log analysis, there has been a lack of emphasis on the need for effective datasets. Consequently, only a few of these techniques have been successfully deployed in real-world settings. This highlights the gap between the limited log data used in academic research and the complexity of industry deployment [13].

*Co-first authors.
**Corresponding author.

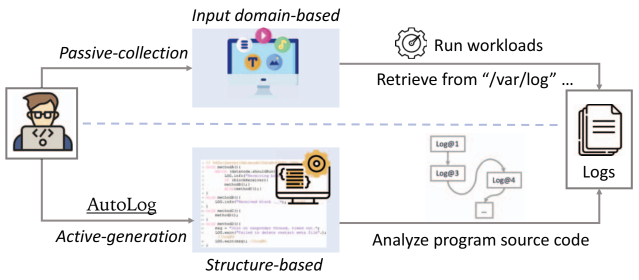


Fig. 1. Difference of existing passive-collection approach and our active-generation methodology AUTOLOG.

Acquiring sufficient, high-quality, and representative logs for practical analysis is challenging. On the one hand, industrial logs from real-world large service providers [13]–[16] (e.g., IBM, Microsoft) contain rich events but have privacy concerns, making it difficult to release them publicly for researchers. On the other hand, logs collected in laboratory environments using simulation [17]–[19] are publicly available but contain simple events produced from limited standard workloads. As a result, they do not accurately reflects the complex run-time workflow of large-scale software. In either case, existing datasets are based on a *passive-collection methodology* that retrieves log files after running applications in a computing system, as described in Fig. 1 (Up). The output logs from the passive-collection methodology are solely dependent on the value of inputs, which impedes exploring and evaluating models on three aspects, namely:

(1) *Comprehensiveness* of log events. Log events are for-

mulated as logging statements in source code for execution. When collecting logs passively, the diversity of log events is mainly determined by the variety of input workloads. For example, the widely-used Hadoop dataset [17] was generated using only two test applications. Even though the system is deployed in scale, it is still impractical to traverse all *what-if* scenarios [20] during execution to trigger complete logging statements by deliberately designed workloads.

(2) *Scalability* over diverse systems. Collecting logs from new systems requires engineers to re-deploy the system and re-implement workloads, which is time-consuming and labor-intensive [19], [21]. Hence, existing datasets are inadequate in terms of system logs, making it challenging to evaluate the generalization ability of proposed log analysis techniques across diverse systems. For example, the most popular dataset, LogHub [19], integrates only five labeled system logs for evaluating anomaly detectors.

(3) *Flexibility* of log utility. While existing datasets allow for minor modifications (e.g., anomaly rate) for evaluation purposes [22], [23], these changes are inadequate for imitating diversified scenarios. For example, when maintainers need to analyze the functionality of a storage component, existing datasets with anomaly-or-not annotation cannot distinguish these component-specific logs. Hence, the complicated conditions in the actual production environment necessitate a more flexible and controllable log collection methodology.

The limitations of the current passive-collection methodology have led to the need for effective log collection for building practical solutions. Since system logs are generated during the execution of logging statements in the source code, (e.g., `LOG.error(``failed to start web server.'')`), we consider the log collection problem as the task of constructing log sequences based on the execution order of logging statements. To create log sequences, we adopt the idea from program analysis to develop AUTOLOG, the first Automated Log generation methodology that *actively generates* effective datasets for anomaly detection (Fig. 1 (Down)). The novel static-guided approach can uncover execution paths and produce rational log sequences [24]–[26].

Specifically, AUTOLOG generates effective log datasets with three phases: (1) *logging statement probing phase* explores all methods containing the logging statements to achieve *comprehensive* log events coverage. (2) *log-related execution path finding phase*, which prunes the call graphs and acquires the execution paths related to the logging statements to ensure *scalability*. (3) *log graph walking phase*, which forms log sequences from execution paths and labels the anomaly sequences based on expert annotations. The controllable parameters of AUTOLOG enable researchers to customize the log dataset with different data sizes, anomaly ratios, and component indicators, providing *flexibility* in log utility.

We conduct an extensive evaluation of AUTOLOG to 50 popular Java projects and compare the resulting dataset to existing passively-collected datasets, showing its superiority from three perspectives: (1) AUTOLOG generates `9x-58x` more log events than existing datasets from the same system, and covers *87.77%* of all log events of the 50 projects on average. (2) AUTOLOG successfully produces logs at least `15x` faster than the collection time for existing log datasets, with tens of thousands of messages per minute on average. (3) AUTOLOG is built with options to change the data size, anomaly ratio, and certain components of logs, supporting a wide range of development environment simulations. Further, we show that existing anomaly detection models effectively gain improvements (1.93%) by learning from the comprehensive log dataset generated by AUTOLOG. Thus, we believe that *the data generation methodology* AUTOLOG *can serve as a critical resource for developing advanced log analysis algorithms, as well as for providing testing and benchmarking data for such algorithms to ensure software reliability.*

To summarize, the contributions of this paper are:

- We present AUTOLOG, a novel, widely-applicable automated log generation methodology that addresses three limitations of existing log datasets: lack of comprehensiveness, scalability, and flexibility.
- AUTOLOG has three phases: logging statement probing, log-related execution path finding, and log path walking.
- Extensive experiments show that AUTOLOG achieves a comprehensive (*87.77%*) coverage of log events and efficiently produces log datasets (over 10,000 messages/min) on Java projects, and offers the flexibility for adapting to multiple scenarios. We further demonstrate that AUTOLOG improves anomaly detectors by 1.93%.
- To our knowledge, AUTOLOG is the first log sequence generator. All artifacts and datasets are released for future research.[1]

```java
// hdfs/server/datanode/DataXceiver.java
void methodA(){
    while (datanode.shouldRun){
        LOG.info("Receiving block " + block); //Log@1
        if (blockReceiver){
            methodB();}
        else{
            methodC();}
    }
}
void methodB(){
    LOG.info("Received block " + block); //Log@2
}
void methodC(){
    methodD();
}
void methodD(){
    msg = "Join on responder thread, timed out;"
    LOG.warn("Failed to delete restart meta file."); //Log@3
    LOG.warn(msg); //Log@4
}
```

Listing 1. A simplified example from HDFS.

II. MOTIVATING STUDY

A. Study Subject

We select four widely-used datasets (Table I) as subjects released by different project teams, including distributed sys-

[1]https://github.com/logpai/AutoLog.

TABLE I
STATISTICS AND DESCRIPTIONS OF EXISTING DATASETS. # LOG EVENT, # WORKLOAD, # FAILURE TYPE, AND # MESSAGE SHOW THE NUMBER OF LOG EVENTS, EXECUTED WORKLOADS, FAILURE TYPES, AND LOG MESSAGES IN EACH DATASET, RESPECTIVELY.

Dataset	# Log Event	# Workload	# Failure Type	# Message	Collection Time	Annotation
$\mathcal{D}$-HDFS	30	NA	11	11,175,629	38.7 Hours	With labelling
$\mathcal{D}$-Hadoop	242	2	3	394,308	NA	With labelling
$\mathcal{D}$-BGL	619	NA	NA	4,747,963	214.7 days	With labelling
$\mathcal{D}$-Zookeeper	77	NA	NA	207,820	26.7 days	Without labelling

tems and supercomputers. In particular, $\mathcal{D}$-HDFS [18], $\mathcal{D}$-Hadoop [17], and $\mathcal{D}$-BGL [21] datasets are collected for anomaly detection after running normal workloads and injecting several types of failures in each system, respectively. $\mathcal{D}$-Zookeeper [19] is collected by running several benchmarks without labeling.

B. Are Existing Datasets Comprehensive?

We first investigate whether the existing log datasets provide comprehensive coverage of logging statements. Considering the case in Listing 1 from Hadoop, `Log@2` will occur if `blockReceiver` is enabled; otherwise, `Log@3` and `Log@4` will be logged. Nevertheless, we find that the latter were absent in the $\mathcal{D}$-HDFS dataset, likely due to `blockReceiver` being enabled at all times. As a result, the anomaly detection techniques trained with an inadequate dataset may fail to tackle the unseen log, and even their experiment conclusions may not be representative. For instance, neglecting the responder connection time anomaly, which is associated with `Log@3` and `Log@4`, can happen if the dataset used for training lacks these logs. Furthermore, Table I reveals that the existing datasets collected through passive-collection approaches are limited in the number of failure types and workloads they cover. The literature [27], [28] implies that although a range of workloads has been implemented, it is still unrealistic to inject all types of anomalies to trigger all log events. Thus, these datasets fall short of comprehensive coverage, creating a significant gap with real-world scenarios.

C. Are Existing Datasets Scalable?

Validating the generalization ability over diverse systems is critical for practical log analysis algorithms. We use the term scalability to measure the effort (e.g., time, workforce) we should put in to acquire logs from multiple system sources.

To determine the scalability of existing datasets, we summarize their log message amounts and collection time in Table I. The table reveals that it takes a long time to collect logs, even after the system has been deployed and configured. For example, collecting 207,820 log messages from the Zookeeper takes more than 26 days. Additionally, since existing datasets are obtained from running system applications, it requires additional expert efforts to redeploy and rerun the applications when extending the workloads to a new system. Unfortunately, software maintainers cannot afford to wait such a long time before developing or validating new algorithms. In short, existing datasets are not scalable enough, necessitating the exploration of a new, more efficient approach to log collection.

D. Are Existing Datasets Flexible?

Previous research has examined the impact of certain dataset characteristics, such as data distribution and data selection, on log-based anomaly detectors [22], [23]. However, to further investigate the performance of log analytics, it is necessary to customize log datasets to simulate diverse scenarios, which requires flexibility.

Ideally, all data can be collected, but in reality, data cannot be completely collected (or collected in a large enough quantity), and they suffer from restrictive flexibility. For example, to increase the anomaly ratio from 0.1% to 15%, researchers may need to remove a large number of normal logs (`150x`) from the existing dataset, sacrificing data quantity [23]. Moreover, logs collected from system files without component specifications raise difficulties for component-wise analysis. More ingredients of the flexibility are elaborated in Section V-D. In this regard, existing datasets cannot flexibly demonstrate model effectiveness under various scenarios, motivating a controllable log sequence acquisition approach.

III. METHODOLOGY

A. Overview

Log files are created every time when a system executes logging statements. Therefore, we view the log sequence generation problem as a task of finding the execution paths related to logging statements in the program. Generating log sequences for anomaly detection involves two primary questions: how to generate program execution paths that include logging statements (Phase2), and how to identify whether the execution path produces an anomaly or not (Phase3).

To tackle the problem, AUTOLOG takes the program as input and outputs a dataset for anomaly detection by analyzing its execution paths. It comprises three phases, as illustrated in Fig. 2: The first phase builds call graphs for the project and marks all methods that contain logging statements (LogMethod), enabling *comprehensive* coverage of log events. The second phase prunes out the call graph nodes and records the log-related execution paths (LogEPs) within each remaining method to overcome the *scalability* issue. The third phase propagates logging statement labels from domain experts to all LogEPs. AUTOLOG eventually generates *flexible* log sequences with chaining LogEPs across methods based on their calling relationship.

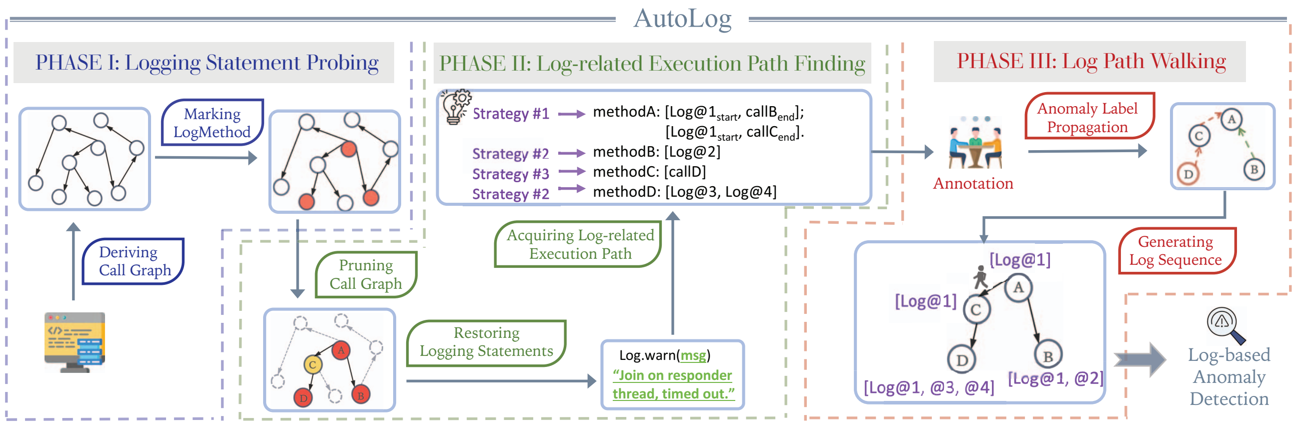

AutoLog
PHASE I: Logging Statement Probing
Marking LogMethod
Deriving Call Graph
PHASE II: Log-related Execution Path Finding
Strategy #1 → methodA: [Log@1start, callBend]; [Log@1start, callCend].
Strategy #2 → methodB: [Log@2]
Strategy #3 → methodC: [callD]
Strategy #2 → methodD: [Log@3, Log@4]
Acquiring Log-related Execution Path
Pruning Call Graph
Restoring Logging Statements
Log.warn(msg)
"Join on responder thread, timed out."
PHASE III: Log Path Walking
Annotation
Anomaly Label Propagation
Generating Log Sequence
[Log@1]
[Log@1]
[Log@1, @3, @4]
[Log@1, @2]
Log-based Anomaly Detection

Fig. 2. The overall framework of AutoLog with three phases: logging statement probing, log-related execution path finding, and log path walking. The details of "Acquiring Log-related Execution Path" are illustrated in Fig. 3.

B. PHASE1: Logging Statement Probing

To cover comprehensive log events, AutoLog starts with exploring the methods containing logging statements and the calling relationships of these methods.

1) Deriving Call Graphs: Initially, we perform a standard static analysis to construct a call graph that chains methods based on their execution-time relationships in a program [29]. A call graph is a directed graph where each node identifies a method and each edge represents a pair of (caller, callee) method-level relationships. AutoLog derives call graphs with the context-insensitive pointer analysis [30] to enhance the precision of the call graph in handling virtual method calls, calls through interfaces, and polymorphism. Afterwards, we treat each call graph as a directed acyclic graph after marking the cycles induced by recursion invocation.

2) Marking LogMethod: Since we concentrate on logs, AutoLog identifies methods containing logging statements (LogMethods) by checking whether any logging API is used in a method. To detect a variety of logging APIs used in different projects, we summarize commonly-used logging frameworks (e.g., slf4j [31]) and capture logging APIs by analyzing the invocation of these popular frameworks. Compared to the regular expressions applied in LogCoCo [24], this API-based analysis is capable of recognizing more comprehensive customized logging APIs by examining the inheritance relationships. Fig. 2 illustrates how AutoLog manages Listing 1, where `methodA`, `methodB`, and `methodD` are recognized as LogMethod (highlighted in red).

C. PHASE2: Log-related Execution Path Finding

An intuitive idea to solve the execution path finding problem is to construct the entire execution graph of a project and traverse it. However, in most cases, large-scale software contains an infinite number of paths [32] so that exhaustive enumeration undoubtedly causes path explosion problem [33]. To overcome scalability challenges, AutoLog takes two steps: (1) pruning out the call graph nodes that will not induce LogMethods; (2) traversing the intra-method execution graph and recording the execution paths that are related only to logging activities.

1) Pruning Call Graph: The goal of pruning is to eliminate redundant nodes from the call graph, particularly those that neither represent LogMethod nodes nor nodes that lead to any LogMethod nodes. A node may induce a LogMethod node in two ways: (1) by calling it directly, or (2) by calling it indirectly through other intermediate nodes. Identifying the LogMethod-inducing nodes can be viewed as a graph sorting problem that finds all ancestor nodes of specific nodes (i.e., LogMethod nodes) in the graph. To do so, AutoLog performs topological sorting [34] over the call graph. Using the topological sorting, a node is considered a non-LogMethod-inducing method if it is neither a LogMethod nor comes before any LogMethod, indicating that it is not an ancestor of any LogMethod. In Fig. 2, red, yellow, and dashed contour nodes are used to represent LogMethod, LogMethod-inducing methods, and non-LogMethod-inducing methods, respectively.

All non-LogMethod-inducing nodes and the edges associated with them (represented by dashed lines) will be pruned out. The resulting pruned graph (denoted as CG') is much smaller than the original call graph since many methods do not contain or induce other logging statements. For example, only *14.79%* of nodes are reserved after pruning call graphs for the HDFS system. Because only the remaining methods are used for further analysis, the pruning step significantly promotes AutoLog's efficiency and facilitates subsequent analysis.

2) Restoring Logging Statements: This step involves restoring logging statements by resolving run-time parameters to provide more detailed information beyond the specific logging statements. A logging statement typically includes constant strings written by developers (e.g., "`Receiving block`") and run-time parameters (e.g., `block`). Specifically, AutoLog resolves the parameters that are constant string variables inside the methods. For example, the parameter `msg` (Line20) is resolved from its assignment (Line18) in Listing. 1. For other types of parameters (e.g., `block`), AutoLog re-

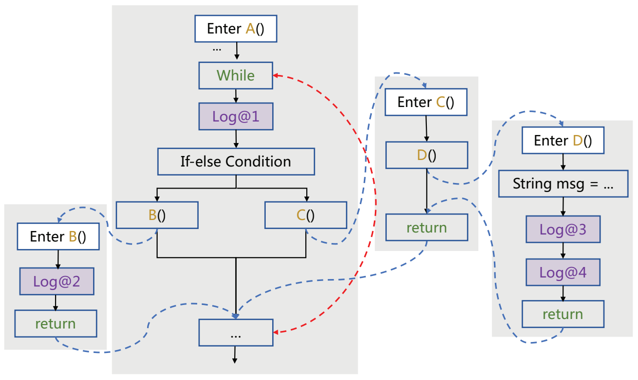


Fig. 3. The simplified execution graphs for methods in Listing 1.

places them with a dummy token (i.e., <*>) since they are typically removed during preprocessing for anomaly detection models [35], [36].

3) Acquiring Log-related Execution Paths: Log sequences generate along with software run-time, reflecting the control-flow paths and execution states [25], [37]–[39]. However, obtaining the order of realistic log events by enumerating execution paths from the entire application can be challenging due to their huge size [40]. To solve this issue, AutoLog constructs and traverses the small-scale execution graph for each method in the CG' to enable scalability.

Particularly, AutoLog builds an execution graph with control flow information for each method and links the invocation with the corresponding method in the entry and exit points. Each node in the execution graph represents an executed activity, and the edge represents the relationship between two activities in the temporal order.

Afterwards, AutoLog traverses each execution graph and only records log-related execution paths (*LogEPs*) for each method, which includes the *invocations* and *logging activities* (i.e., logging statements). The invocation is noteworthy because a log sequence in a method can be interrupted by another log sequence introduced by the invocation. AutoLog derives LogEPs for each method by using three strategies. Fig. 3 depicts the execution graphs for four methods of Listing 1 to illustrate the strategies.

- Strategy #1: For nodes that are *both non-leaf nodes and LogMethod* in the CG', AutoLog considers the execution order of invocations and logging activities. In our case, we obtain two LogEPs for `methodA`: `[Log@1, callB]`, and `[Log@1, callC]`.
- Strategy #2: For *leaf nodes* in CG' that are definitely *LogMethods*, AutoLog enumerates all possible execution sequence of logging activities. This strategy is applied to `methodB` and `methodD`, leading to the LogEP of `[Log@2]` and `[Log@3, Log@4]`, respectively.
- Strategy #3: For the *non-leaf and non-LogMethod* nodes in the CG', AutoLog records all possible invocation sequences in execution. In this case, one LogEP is derived for `methodC`: `[callD]` applying this strategy.

In AutoLog, loops (e.g., for, while) are viewed as paths that are repeatedly traversed in a tail-recursive way and are cycle-free. This crucial feature allows our approach to enumerate all possible execution paths. Nodes within the loop can occur multiple times in a row to mimic the actual execution sequences. When acquiring LogEPs, we mark the nodes (i.e., first and last node) in a loop with a special sign. For instance, LogEP for methodA will be marked as: [Log@1$_{start}$, callB$_{end}$], and [Log@1$_{start}$, callC$_{end}$].

Although LogEPs provide all possible executable paths, there exist some paths that are not executable under any input values. To avoid such infeasible paths, we conduct intra-procedural constraint analysis following previous studies [25]. In specific, we gather all constraints for each LogEP and filter out any LogEPs that contain unsatisfiable constraints. This process makes the generated LogEPs more realistic. Using the scalable traversal strategy, AutoLog obtains all LogEPs in less than 1.5 hours in an HDFS system that includes 3,749 methods (pruned) and 2,535 logging statements.

D. PHASE3: Log Path Walking

This phase is devised with the goal of generating labeled log sequences to simulate the actual application execution by chaining LogEPs from each method. To achieve accurate labels while saving annotation effort, AutoLog uses a *seed-propagation strategy* that involves experts identifying a set of anomaly LogEPs as seeds, followed by automatically propagating the labels of these anomaly LogEPs to all acquired LogEPs. Labeled log sequences are eventually generated by a succession of random LogEP selection step which walks over the invoked methods. The choice of LogEPs can be controlled by setting specified data size, execution entrance and anomaly rate, allowing a flexible dataset that simulates execution logs in multiple scenarios.

1) Seed Anomaly LogEP Annotation: Since identifying anomalous logs is challenging and mostly relies on domain expert knowledge, we adopt the human-annotation process similar to existing log-based anomaly detection datasets [17], [18], [21]. However, AutoLog differs from existing datasets in that, it can automatically propagate labels from a set of anomaly unit to the sequence-level, significantly improving the annotation efficiency.

We use LogEP as an anomaly unit for expert labeling as it is a small execution path reflecting the system behavior in a period. The output of this step is a set of seed anomaly LogEPs that *must* produce anomalies. To obtain anomaly LogEPs, we design the annotation process with *alerting statement annotation* and *anomaly LogEP identification*. The alerting statement annotation is designed to filter out a large number of normal logging statements (e.g., `Log@1`). To do so, we present all logging statements, their corresponding code snippets as the execution context, as well as the nearby comments inserted by developers, for annotators to decide whether they are alerting logging statements for an anomaly. Taking the HDFS system as an example, we present the example alerting statements and their corresponding potential anomalies in Table II. However,

TABLE II
ALERTING LOGGING STATEMENTS EXAMPLES AND THEIR POTENTIAL ANOMALY TYPES IN HDFS SYSTEM.

Anomaly Type	Alerting Logging Statements Examples
Version mismatch	Layout version on remote node does not match this node's layout version.
Disk/Storage error	Unable to get json from Item. Unexpected health check result for volume $<*>$.
Dependency error	Unresolved dependency mapping for host . Continuing with an empty dependency list

identifying alerting statements is not enough for profiling system activities. Hence, anomaly LogEP identification aims to further determine whether LogEP will certainly lead to anomalies. To recognize anomaly LogEPs, we ask annotators to manually check each LogEP that contains alerting statements. The identified anomaly LogEP is considered anomaly seeds for propagation in the next step.

2) Anomaly Label Propagation: To generate an effective anomaly detection dataset, it is important to have labels at the sequence-level even though the annotations are done at the LogEP-level to save human effort. To this end, AUTOLOG uses a strategy called seed propagation to propagate the labels of seed anomaly LogEPs to other LogEPs, with the goal of figuring out whether a LogEP is infected by the anomaly label. The main idea is that: If a LogEP in one method contains an anomaly (e.g., `methodD`), then all other LogEP invokes this method (e.g., `[callD]`) *may* also induce an anomaly.

The propagation starts from the seed anomaly LogEPs marked *infected*. The propagation is done recursively by checking whether a LogEP contains other infected LogEPs brought from invocations. After the propagation, infected LogEPs are *likely to*, but will not necessarily cause an anomalyz, whereas others *must* be anomaly-free.

3) Generating Log Sequences: Given the LogEPs, AUTOLOG eventually generates each actual log sequence by selecting and chaining the LogEPs in a top-down approach. The top-down random walking process works as follows: It starts at an entrance method and walks along invocations, with one LogEP being randomly chosen at each walking step. If an invocation exists in the current chosen LogEP, AUTOLOG walks to the callee method and then chooses a LogEP in the callee method. The logging statements in all chosen LogEPs are chained according to the invocation relationships to form the log sequence.

As one log sequence reflects the execution path of a single thread, AUTOLOG generates a labeled sequence at each time with two walking strategies and combines the sequences to form the datasets for anomaly detection:

- To generate an *anomaly log sequence*, we always select the infected LogEP in every step until we have selected an anomaly LogEP that may contain alerting logging statements.
- To generate a *normal log sequence*, we randomly select a LogEP of each method but take a step back when selecting an anomaly LogEP, and re-choose another LogEP.

During the walking, invocations or logging statements within the loop may occur successively more than once. The log sequence generation process with random path selection enables the flexibility of datasets. It can be decorated with a set of hyper-parameters to generate more controllable log sequences. For example, the component indicator (CI) controls the starting point to simulate the execution path in a specific component (e.g., storage component), and the anomaly rate AR controls the anomaly ratio ($\frac{\#Anomaly\_Sequence}{\#All\_Sequence}$) in the generated dataset.

IV. IMPLEMENTATION

A. Experiment Environment

AUTOLOG has been implemented by 5,182 lines of Java code with Soot [41], a Java bytecode optimization and analysis framework. We run all experiments on Ubuntu 18.04. The experiments are carried out on a machine with an Intel(R) Xeon(R) Platinum 8255C CPU (@2.50GHz) with 128GB RAM. We set the AR to be 3% and CI to be all possible paths to ensure the coverage, unless otherwise specified.

B. Annotation

To ensure the correctness of annotation, we invite three Ph.D. students who have at least two-year experience in distributed system research and development, two of whom annotate individually, and the other one works as an adjudicator to discuss the disagreements with annotators. They are all allowed to access the Internet for searching answers. The agreement score between annotators measured by Cohen's kappa [42] before adjudication is 0.841 and 0.834 in alerting statement annotation and anomaly LogEP identification, respectively. All annotators reach a consensus on the labels after discussing them with the adjudicator.

V. EXPERIMENTS

We evaluate AUTOLOG using four research questions:
RQ1: How comprehensive are the datasets generated by AUTOLOG?
RQ2: Is AUTOLOG scalable for real-world applications?
RQ3: How flexible are the datasets compared with passively-collected datasets?
RQ4: Can AutoLog benefit anomaly detection problems?

A. Experimental Settings

1) Existing Datasets: To verify the effectiveness of AUTOLOG, we choose three widely-used publicly available datasets collected from Java applications. We use $\mathcal{D}$-*sys* and AUTOLOG-*sys* to denote the baseline datasets collected from system *sys* and collected by AUTOLOG, respectively. Details of the datasets are illustrated as follows:

- $\mathcal{D}$-Hadoop [17]. Hadoop is an open-source framework designed to store and process large-scale data efficiently. The dataset is collected via running two standard applications and injecting three types of failures for anomaly detection.

TABLE III
THE COMPARISON OF DATASETS FOR COMPREHENSIVENESS. $\mathcal{D}$-COVERAGE IS REPORTED FOR THE SYSTEMS WITH PUBLICLY LOG DATASETS.

System	Dataset	# Log Event	Logging Coverage	$\mathcal{D}$-Coverage	Increment (↑)
Hadoop	$\mathcal{D}$-Hadoop	242	242/3426 (7.1%)	219/242 (90.5%)	12x
	AutoLog-Hadoop	2879	2879/3426 (84.0%)		
HDFS	$\mathcal{D}$-HDFS	30	30/1700 (1.8%)	27/30 (90.0%)	58x
	AutoLog-HDFS	1367	1367/1700 (80.4%)		
Zookeeper	$\mathcal{D}$-Zookeeper	77	77/758 (10.2%)	77/77 (100%)	9x
	AutoLog-Zookeeper	740	740/758 (97.6%)		
Apache Storm	AutoLog-Apache Storm	1754	1754/1887 (93.0%)	-	-
Flink	AutoLog-Flink	1574	1574/1711 (92.0%)	-	-
Kafka	AutoLog-Kafka	847	847/1002 (84.5%)	-	-

- $\mathcal{D}$-HDFS [18]. HDFS is a distributed file system for large-data storage, enabling high-throughput access to data. $\mathcal{D}$-HDFS is collected from a private cloud environment executing benchmark workloads with labeled anomalies.
- $\mathcal{D}$-Zookeeper [19]. Zookeeper provides a centralized service to manage a large set of hosts (e.g., synchronization, configuration information management). $\mathcal{D}$-Zookeeper is collected in a lab environment for log analysis *without labelling*.

Since this paper presents the first methodology that actively generates log datasets without deploying and running the system, we compare AutoLog with all existing Java-based log datasets in our research questions.

2) Evaluation Subjects: Apart from three projects associated with existing log datasets, we extensively evaluate the effectiveness of AutoLog on the most popular 50 projects from the Maven repository [43], with more than 10,000 usage times for each. The selected projects include, but are not limited to, distributed streaming platforms (e.g., Kafka), core Java packages (e.g., Apache HttpClient), and unit testing frameworks (e.g., JUnit). Among them, we present the detailed result of three widely-studied distributed system projects below and report statistical results for other projects.

- Apache Storm [44]. It is a distributed real-time computation system, allowing large-scale data processing and high-velocity data streams.
- Flink [45]. Flink is a stream processing engine that can handle scale system events with low latency.
- Kafka [46]. Kafka is a distributed event storage and stream-processing platform applied in thousands of companies.

3) Metrics:

- *Coverage*. Motivated by software testing studies [24], [47], we evaluate the comprehensiveness of *logging coverage*, which measures the percentage of the discovered log events to the total log events designed in the application ($\frac{\#Log\_Event}{\#Total\_Log\_Event}$). We also use $\mathcal{D}$-Coverage to evaluate the ratio of log events in existing datasets that are covered by AutoLog ($\frac{\#Log\_Event\_Covered\_by\_AutoLog}{\#Log\_Event\_in\_Existing\_Dataset}$).
- *Execution Time*. To validate the scalability of AutoLog, we report the program execution time for each system, which includes the time for code analysis and data generation in a single machine.

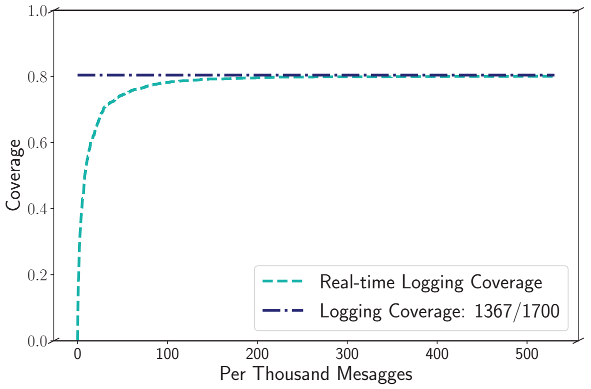


Fig. 4. The number of generated log messages and their corresponding real-time logging coverage.

B. RQ1: How Comprehensive are the Datasets Generated by AutoLog?

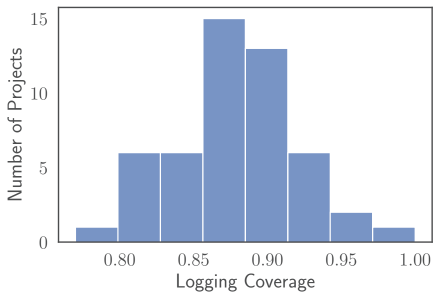
We evaluate the comprehensiveness of the dataset generated by AutoLog regarding the number of log events, logging coverage, and $\mathcal{D}$-Coverage, illustrated in Table III.

Compared to the existing log datasets, AutoLog effectively enhances the comprehensiveness of log events. Firstly, AutoLog covers an average of 87.38% logging statements over six systems, demonstrating its ability to capture logging statements designed in code. The number of log events generated by AutoLog is `12x, 58x,` and `9x` is more than existing datasets collected from Hadoop, HDFS, and Zookeeper, respectively. Regarding the missing parts, we analyze them as follows. They mainly come from the restrictive call graph construction phase [25], the limitation of logging statement restoring [48] across different methods, and unreachable execution paths that we have eliminated. Taking the restoring process as an example, if the constant string `msg` is defined outside methodD in Listing 1, AutoLog has difficulty restoring the `Log@4`. Secondly, the experiment results illustrate that AutoLog covers most of the log events in the existing

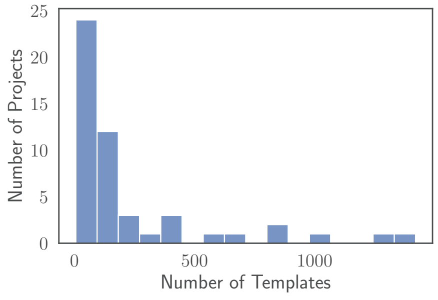
TABLE IV
THE COMPARISON OF EXISTING DATASETS FOR SCALABILITY.

System	Dataset	# Message	Execution Time	# Messages/min (speed)	Acceleration (↑)
Hadoop	$\mathcal{D}$-Hadoop AUTOLOG-Hadoop	394,308 392,427	NA 3.41 hours	NA 1,918	–
HDFS	$\mathcal{D}$-HDFS AUTOLOG-HDFS	11,175,629 11,376,233	38.7 hours[†] 2.62 hours	4,813 72,367	15x
Zookeeper	$\mathcal{D}$-Zookeeper AUTOLOG-Zookeeper	207,820 211,425	26.7 days[†] 17 mins	6 12,436	2072x
Apache Storm	AUTOLOG-Apache Storm	1,001,245	1.28 hours	13,037	-
Flink	AUTOLOG-Flink	1,003,416	1.21 hours	13,821	-
Kafka	AUTOLOG-Kafka	1,002,629	39 mins	25,708	-

[†] This is the collection time from its original paper. NA means the authors do not report the collection time.
"-" means the acceleration cannot be computed due to the unknown collection time of existing datasets.



(a) Logging coverage histogram



(b) # Log event histogram

Fig. 5. The histogram of logging coverage and the number of log events over 50 popular projects.

dataset, achieving 219/242, 27/30, and 77/77 for Hadoop, HDFS, and Zookeeper systems, respectively. The missing log events mainly come from optional components of complicated systems and different deployment settings on varied platforms.

Logging coverage implies the upper limit of the discovered log events in AUTOLOG; ideally, generating log sequences for enough time will eventually reach that coverage. Fig. 4 shows the relationship between the number of real-time generated log messages and its corresponding logging coverage in HDFS system (denoted as real-time logging coverage). The results indicate that AUTOLOG achieves its logging coverage after generating approximately 350,000 messages.

Besides, we display the logging coverage (left) and the number of generated log events (right) histogram over 50 popular Java projects in Fig. 5. The results demonstrate that AUTOLOG achieves an average logging coverage of 87.77%, which is superior to existing log datasets. Additionally, the number of log events in different projects ranges from hundreds to thousands, indicating that existing log datasets with limited log events are inadequate.

AUTOLOG shows a significant improvement (`9x−58x`) on the number of log events and can effectively cover *87.77%* logging statements on average over studied projects.

C. RQ2: Is AUTOLOG *Scalable for Real-world Applications?*

We evaluate the scalability of our log generation methodology, which measures the time required for generating data. To compare the scalability, we generated the same amount of data in a single machine as existing public datasets and compared the data collection time.

The result in Table IV indicates that AUTOLOG is efficient in analyzing and generating log sequences from real-world applications. It can produce messages at high speed, with a range of 1,918 to 72,367 messages per minute. Moreover, compared with the HDFS dataset that took 38.7 hours to collect, AUTOLOG can generate the same amount of data within 2.62 hours (15 times faster). This scalability is attributed to the call graph pruning and intra-method LogEP derivation steps, which decrease the number of methods for analysis and solve the path explosion problem. Moreover, because AUTOLOG is built on source code, deploying it into a new system is effortless, unlike the long time required for configuring and rerunning applications in existing passively-collection methods. Additionally, we investigate the time spent for execution path finding (Phase1 and Phase2) as LogEPs can be reusable to generate log sequences multiple times. In particular, we apply AUTOLOG on 50 Java projects and measure the execution time, ranging from 687 to 83,969 methods in a project (as shown in Fig. 6). Our results demonstrate

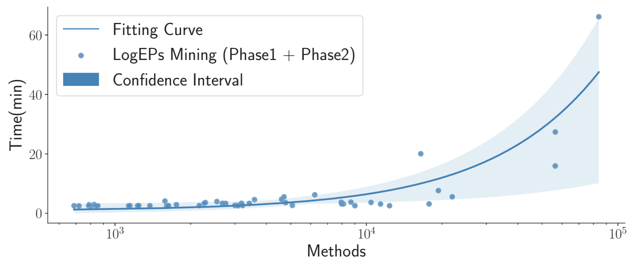


Fig. 6. Time cost for acquiring LogEPs and the number of analyzed methods over 50 popular projects.

TABLE V
THE COMPARISON OF LOG DATASETS FOR FLEXIBILITY.

Aspects	$\mathcal{D}$-HDFS	$\mathcal{D}$-Hadoop	AUTOLOG
Data Size	Partial	Partial	**Complete**
Component Indicator	Partial	Partial	**Complete**
Anomaly Rate	Deterministic	Deterministic	**Non-deterministic**

that AUTOLOG can analyze most projects within one hour, indicating its practicality and efficiency.

Although we did not include the annotation time in Table IV, AUTOLOG has an efficient seed-propagation labeling strategy that requires less time than labeling each log sequence in passively-collected datasets. For instance, independent annotators spent an average of 2 hours on alerting statement annotation and 4 hours on anomaly LogEP identification for 1,367 log events in HDFS.

AUTOLOG considerably shortens the dataset generation time (`15x`) compared with existing datasets. The promising path finding time further manifests its scalability for real-world applications.

D. RQ3: How Flexible are the Datasets Compared with Passively-collected Datasets?

We assess the flexibility of AUTOLOG and its benefits for log-based anomaly detection. In Table V, we compare AUTOLOG with existing datasets regarding the flexibility over *data size*, *component-indicator*, and *anomaly rate*.

Different systems require varying amounts of data for algorithm development. For example, a large amount of data can be exploited to build algorithms for distributed cloud systems whereas naive systems only preserve a small amount of data to study. AUTOLOG is capable of generating datasets of *any* size, whereas $\mathcal{D}$-HDFS and $\mathcal{D}$-Hadoop have partial size restrictions once they are released.

In addition, engineers may need to focus on specific components' functionality during software development and maintenance, which requires component-specific logs for anomaly detection. While the existing datasets provide limited component information, AUTOLOG is able to produce sequences of component-indicator logs by starting to walk from constrained entry nodes during generation.

Moreover, different systems have different fault-tolerance abilities, which affect the potential anomaly rates in collected logs. Existing datasets (e.g., $\mathcal{D}$-HDFS) have fixed numbers of anomaly sequences, requiring researchers to filter out some anomaly sequences or significantly lower the number of normal sequences to increase the anomaly rate in a deterministic way [23]. However, AUTOLOG can produce a huge amount of various sequences iteratively, introducing the unseen patterns and increasing the flexibility for anomaly detection.

AUTOLOG effectively generates datasets with more flexibility of utilization (i.e., data size, component, anomaly rate) than existing datasets, allowing imitating a wider range of sophisticated application scenarios.

E. RQ4: Can AutoLog Benefit Anomaly Detection Problems?

This RQ explores how AUTOLOG helps to resolve log-based anomaly detection. In particular, we train state-of-the-art (SOTA) detectors on AUTOLOG and evaluate their performance. This section takes HDFS[2] to evaluate models on real-machine-generated logs $\mathcal{D}$-HDFS (denoted as $\mathcal{D}$) and AUTOLOG-HDFS (denoted as AUTOLOG) from the same software version, followed by a performance discussion.

1) Settings: We select the following representative log anomaly detection models for evaluation: (1) LogRobust [35], which leverages a bi-directional LSTM network (bi-LSTM) with an attention mechanism to learn the importance of each log for tackling unstable log patterns. (2) CNN [36], which transforms the log sequence to a trainable matrix, then applies a Convolutional Neural Network (CNN) for log-based anomaly detection. (3) Transformer [49], which applies a Transformer encoder to learn context information to distinguish the anomaly logs from normal logs. When using AUTOLOG, we set the anomaly rate (3%), and data split ratio (train: test = 8: 2) to be the same as the original $\mathcal{D}$.

Following previous research work [35], [50], we adopt Precision (P), Recall (R), and F1 to evaluate the performance. *Precision* is calculated by the percentage of log sequences correctly identified with anomalies over all sequences that are recognized with anomalies. *Recall* is calculated by the percentage of log sequences correctly recognized with anomalies over the actual anomaly sequences. *F1 Score* (F1) is the harmonic mean between Precision and Recall.

2) Results: Table VI presents the performance of different anomaly detection models on two datasets. First, we evaluate models on the real-world dataset $\mathcal{D}$ (test set). We observe that models trained with AUTOLOG perform *consistently better* (1.93% of F1 on average) than trained with $\mathcal{D}$ (train set). CNN can achieve an F1 score of 0.978 after training with AUTOLOG, even surpassing the SOTA performance by 1.1%. We analyze the reason for performance improvement after training on AUTOLOGas follows. AUTOLOG generates more comprehensive log events by imitating a more varied system behavior. Once an anomaly detector has seen such diverse log

[2]Other widely-used anomaly detection datasets (e.g., BGL) are not generated from open-sourced software.

TABLE VI
COMPARISON OF THE ANOMALY DETECTION MODELS OVER TWO DATASETS $\mathcal{D}$ AND AUTOLOG.

Train set		$\mathcal{D}$			AUTOLOG		
Test set	Approach	P	R	F1	P	R	F1
$\mathcal{D}$	Transformer	0.889	0.904	0.896	0.892	0.996	**0.941**
	CNN	0.936	0.995	0.965	0.959	0.997	**0.978**
	LogRobust	0.942	0.994	**0.967**	0.947	0.988	**0.967**
AUTOLOG	Transformer	-[†]			0.723	0.755	0.739
	CNN	-[†]			0.697	0.790	0.741
	LogRobust	-[†]			0.673	0.875	0.761

[†]The large amount of unseen events will lead to considerably poor performance.

events in its training phase, it can effectively detect the anomalies in production environments. The result demonstrates that AUTOLOG is effective in generating realistic log sequences, which can help models learn more varied system behavior and improve anomaly detection accuracy.

Second, we observe that state-of-the-art approaches perform well in $\mathcal{D}$ dataset but can only reach an F1 score of 0.761 in AUTOLOG. We attribute the performance gap to the more comprehensive log events (e.g., 1367 rather than 30) and diverse sequential log patterns generated by AUTOLOG. This highlights the need for further research on log semantic encoding and system behavior profiling to address the challenges posed by complex log data in real-world deployments of anomaly detection models.

AUTOLOG benefits anomaly detection detectors by providing the training resource that allows existing models to improve (*1.93%* on average) their performance consistentl. It is effective in generating more sophisticated and practical log data than any existing Java log datasets. It can serve as a benchmark data generator and training resource for developing and validating promising anomaly detection models.

VI. CASE STUDY

Fig. 7 exhibits a case from an HDFS user who tries to delete blocks and encounters data loss issue[3]. The user reports the log sequences from Datanode (left) and Namenode (right). After matching this realistic log sequence in $\mathcal{D}$-HDFS and AUTOLOG-HDFS, we find that (1) both $\mathcal{D}$-HDFS and AUTOLOG-HDFS contain the sequence from Datanode, (2) only AUTOLOG-HDFS can cover the sequence from Namenode. In AUTOLOG, the acquired execution paths via program analysis simulate how logging statements are executed (recorded) in software to provide realistic log sequences in block deletion. In addition, the case illustrates the comprehensiveness of logs produced by AUTOLOG. Although it is impractical to configure and enumerate all system activities in software for log collection, AUTOLOG addresses the issue by actively analyzing all possible execution paths.

[3]https://issues.apache.org/jira/projects/HDFS/issues/HDFS-16829

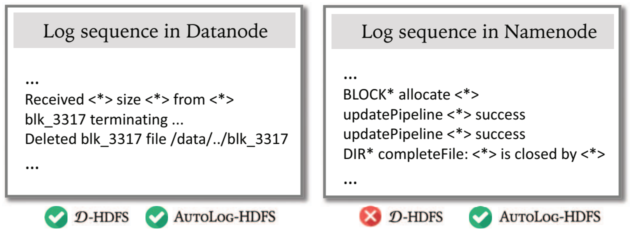

Log sequence in Datanode
```
...
Received <*> size <*> from <*>
blk_3317 terminating ...
Deleted blk_3317 file /data/../blk_3317
...
```
Log sequence in Namenode
```
...
BLOCK* allocate <*>
updatePipeline <*> success
updatePipeline <*> success
DIR* completeFile: <*> is closed by <*>
...
```

Fig. 7. User-reported log event sequences from HDFS.

VII. THREATS TO VALIDITY

We have identified the following four major threats to validity. (1) *Unpredictable exception handlers*: A few run-time behaviors, such as exception catching, cannot be thoroughly analyzed without actually executing the program. A logging statement can be severed from the execution path by the exception handler. While exceptions can reflect anomalies, these anomalies are relatively "easy" to be detected (e.g., keyword search). Existing log-based anomaly detection mainly focuses on the remaining "difficult" anomalies. To mitigate this threat, we can simulate run-time exception-catching behaviors by randomly selecting try-catch statements and setting interruptions in the try body.

(2) *Potential multi-thread intervention*: AUTOLOG analyzes execution paths and generates log sequences in a single thread. Thus, the intervention and communication between multiple threads may be neglected. However, we notice that modern anomaly detection models [35], [36], [49] are developed for single-thread analysis, which starts with separating different threads based on their thread IDs. In this regard, the lack of multi-thread log sequences will not hamper the evaluation of existing anomaly detection models. In the future, we plan to extend AUTOLOG to support multi-thread settings.

(3) *Unresolved parameters*: AUTOLOG uses a dummy token to replace unresolved parameters (e.g., addresses, digits, ID character strings) in the logging statements, which is supposed to carry system behavior information. However, previous studies [8], [22] reveal that log event (without run-time parameters) sequences are sufficient to capture system activities, and parameters may hinder deep learning-based detectors [35]. Even though the unresolved parameters are specified with certain values, they will be wiped out before feeding into anomaly detection models. In any case, we regard parameters resolving as an important future direction for their utilization in other log analytics (e.g., log parser).

(4) *Imprecise call graph*: Generating precise call graphs has been known as an open-challenging question in static analysis community for a long time [48], [51], [52]. Soot [41] uses Class Hierarchy Analysis (CHA) [53] to handle dynamic dispatch (e.g., finding the callee), which has a relatively low accuracy due to the polymorphism of Java. To mitigate the impact of the imprecise graph, we refined the calling relationship with context-insensitive pointer analysis that significantly improves the precision of the call graph.

VIII. Related Work

A. Logging Statements Generation

Recording the precise and concise logs in software becomes a critical problem for logging practice as developers inspect logs for software operation and maintenance. Logging statements generation aims at suggesting developers in deciding *what* to log, *where* to inject a log and at which *log level*. A collection of studies are proposed to support developers in logging activities. For *where-to-log*, [54] extracted syntactic and semantic information from the code block level to suggest the logging locations via a deep learning-based approach. For *log level*, PADLA [55] is presented to dynamically adjust the log level of a running system guided by the online phase performance anomalies detection. Similarly, DeepLV [56] is a neural approach that automatically suggests the log level by incorporating the syntactic context (i.e., AST) and message features (i.e., log message) from code. Regarding *what-to-log*, several studies suggest the variables to log via source-code analysis [25] and variable representation learning [57]. The recent study [58] applied a Text-To-Text-Transfer-Transformer (T5) model to generate complete logging statements and inject them in the correct code location. The latest study [59], inspired by the huge success of large language models (LLMs) in natural language understanding and code intelligence tasks, empirically analyzed how well LLMs in generating complete logging statements. Instead of deciding on log statements in code, AutoLog stands at a completely different perspective that automatically generates log sequences from existing projects for the research and development community.

B. Data Synthesis

Data synthesis has been investigated in various domains and modalities, such as time series and traces. Statistical models [20] and generative models [60], [61] are two classical methodologies to synthesize time-series data. For example, TSAGen [20] is a univariate time-series synthesizer that yields Key Performance Indicator (KPI) data with various anomalies and controllable characteristics using a probability distribution. Moreover, a few studies have been performed to synthesize location traces to protect mobile users' privacy, including employing synthetic point injection in a trajectory of a user [62], synthesizing differentially private trajectories by variable-length n-grams [63] or hierarchical reference systems [64]. AutoLog differs from all above studies in the way that it generates data from the code level to simulate execution paths instead of synthesizing from existing data.

The only study related to synthesizing logs is LogRobust [35], which enhances the existing dataset by randomly shuffling, adding, and removing words in collected logs. However, such a mutation-based synthesis strategy can hardly reflect the real logging statements and their sequential patterns that are not covered in data collection.

C. Log-based Anomaly Detection

Automated log file analysis enables to detect anomalous system activities early and guides troubleshooting. The anomaly detection task requires the model to identify whether there exist anomalies in a short time of logs. Several machine learning-based models (e.g., decision tree [65], support vector machine [66]) are built to solve the task. Recent studies demonstrate that the advanced deep learning-based models [67] achieve higher performance, such as LSTM [5], [35], auto-encoder [68], and Transformer [49], [50].

Although plenty of efforts have been devoted to intelligent log anomaly detection, log files are seldom made available to the public, making it difficult for researchers to validate new algorithms. The most widely-used log analysis dataset, LogHub [19], only contains limited logging statement coverage and thus cannot reflect the modern complex systems. AutoLog, as a log dataset generation methodology, breaks this bottleneck by efficiently providing a comprehensive and controllable data source.

IX. Conclusion

Although log-based anomaly detection has been widely studied, only a few approaches have been successfully deployed in the real world because existing datasets suffer from comprehensiveness, scalability, and flexibility limitations. To overcome these limitations, this paper presents AutoLog, the first automated log generation methodology for anomaly detection, with three phases: logging statement probing, log-related execution path finding, and log path walking. Extensive experiments demonstrate that AutoLog produces comprehensive coverage of log events in application with scalability. AutoLog is also equipped with hyper-parameters to generate log datasets in flexible data size, component indicator, and anomaly rate. With our replication package released, we believe AutoLog could be a starting point for active dataset generation in the log analysis field, which provides benchmarking data for building practical anomaly detection algorithms.

X. Acknowledgment

The work described in this paper was supported by the National Natural Science Foundation of China (No. 62202511), the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), and the Key Program of Fundamental Research from Shenzhen Science and Technology Innovation Commission (No. JCYJ20200109113403826). It was also supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). We thank all reviewers for their valuable comments.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[3] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 92–103.

[4] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Plelog: Semi-supervised log-based anomaly detection via probabilistic label estimation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 230–231.

[5] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[6] Y. Huo, Y. Su, C. Lee, and M. R. Lyu, "Semparser: A semantic parser for log analytics," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 881–893.

[7] J. Liu, J. Huang, Y. Huo, Z. Jiang, J. Gu, Z. Chen, C. Feng, M. Yan, and M. R. Lyu, "Scalable and adaptive log-based anomaly detection with expert in the loop," *arXiv preprint arXiv:2306.05032*, 2023.

[8] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Kopru, and T. Xie, "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 419–429.

[9] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 389–396.

[10] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui, "Root-cause metric location for microservice systems via log anomaly detection," in *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 2020, pp. 142–150.

[11] A. R. Chen, T.-H. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 8, pp. 2905–2919, 2022.

[12] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu *et al.*, "An intelligent framework for timely, accurate, and comprehensive cloud incident detection," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 1–7, 2022.

[13] N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, Y. Wu, Z. Feng, X. Wen, W. Zhang *et al.*, "An empirical investigation of practical log anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2021, pp. 1404–1415.

[14] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services," in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 25–32.

[15] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin *et al.*, "Onion: identifying incident-indicating logs for cloud systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2021, pp. 1253–1263.

[16] T. Jia, Y. Li, Y. Yang, G. Huang, and Z. Wu, "Augmenting log-based anomaly detection models to reduce false anomalies with human feedback," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 3081–3089.

[17] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 102–111.

[18] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[19] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: a large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[20] C. Wang, K. Wu, T. Zhou, G. Yu, and Z. Cai, "Tsagen: synthetic time series generation for kpi anomaly detection," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 130–145, 2021.

[21] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2007, pp. 575–584.

[22] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu, "Experience report: deep learning-based system log analysis for anomaly detection," *arXiv preprint arXiv:2107.05908*, 2021.

[23] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: how far are we?" in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1356–1367.

[24] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 305–316.

[25] 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, no. 1, pp. 1–28, 2012.

[26] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 565–581.

[27] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 457–466.

[28] A. Pecchia and S. Russo, "Detection of software failures through event logs: An experimental study," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 31–40.

[29] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering (TSE)*, no. 3, pp. 216–226, 1979.

[30] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[31] Apache, *SLF4J*, Aug. 2022, https://www.slf4j.org.

[32] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 18, no. 3, p. 206, 1992.

[33] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[34] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[35] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2019, pp. 807–817.

[36] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 151–158.

[37] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *International Conference on Extending Database Technology*. Springer, 1998, pp. 467–483.

[38] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, "Using finite-state models for log differencing," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2018, pp. 49–59.

[39] N. Busany and S. Maoz, "Behavioral log analysis with statistical guarantees," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 877–887.

[40] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.
[42] M. Banerjee, M. Capozzoli, L. McSweeney, and D. Sinha, "Beyond kappa: A review of interrater agreement measures," *Canadian journal of statistics*, vol. 27, no. 1, pp. 3–23, 1999.
[43] Apache, *The Maven Repository*, Aug. 2022, https://mvnrepository.com/repos/central.
[44] *Storm*, Aug. 2022, https://storm.apache.org.
[45] *Flink*, Aug. 2022, https://flink.apache.org.
[46] *Kafka*, Aug. 2022, https://kafka.apache.org.
[47] S. Park, B. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.
[48] A. Utture, S. Liu, C. G. Kalhauge, and J. Palsberg, "Striking a balance: Pruning false-positives from static call graphs," 2022.
[49] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," in *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 1196–1201.
[50] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 492–504.
[51] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.
[52] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
[53] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*. Springer, 1995, pp. 77–101.
[54] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ICSE)*, 2020, pp. 361–372.
[55] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "Padla: a dynamic log level adapter using online phase detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 135–138.
[56] Z. Li, H. Li, T.-H. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1461–1472.
[57] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 2012–2031, 2019.
[58] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2279–2290.
[59] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging generation: An empirical study," *arXiv preprint arXiv:2307.05950*, 2023.
[60] J. Yoon, D. Jarrett, and M. Van der Schaar, "Time-series generative adversarial networks," *Advances in neural information processing systems (NIPS)*, vol. 32, 2019.
[61] C. Esteban, S. L. Hyland, and G. Rätsch, "Real-valued (medical) time series generation with recurrent conditional gans," *arXiv preprint arXiv:1706.02633*, 2017.
[62] R. Shokri, G. Theodorakopoulos, J.-Y. Le Boudec, and J.-P. Hubaux, "Quantifying location privacy," in *2011 IEEE symposium on security and privacy (S&P)*. IEEE, 2011, pp. 247–262.
[63] R. Chen, G. Acs, and C. Castelluccia, "Differentially private sequential data publication via variable-length n-grams," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012, pp. 638–649.
[64] X. He, G. Cormode, A. Machanavajjhala, C. M. Procopiuc, and D. Srivastava, "Dpt: differentially private trajectory synthesis using hierarchical reference systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1154–1165, 2015.
[65] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 2004, pp. 36–43.
[66] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *Seventh IEEE International Conference on Data Mining (ICDM)*. IEEE, 2007, pp. 583–588.
[67] Y. Huo, C. Lee, Y. Su, S. Shan, J. Liu, and M. Lyu, "Evlog: Evolving log analyzer for anomalous logs identification," *arXiv preprint arXiv:2306.01509*, 2023.
[68] A. Farzad and T. A. Gulliver, "Unsupervised log message anomaly detection," *ICT Express*, vol. 6, no. 3, pp. 229–237, 2020.