Anomaly Detection Using Program Control Flow Graph Mining from Execution Logs

Animesh Nandi[‡], Atri Mandal[‡], Shubham Atreja[⋄], Gargi B. Dasgupta[‡], Subhrajit Bhattacharya[‡]
[‡] IBM Research
[⋄] IIT Kanpur

ABSTRACT

We focus on the problem of detecting anomalous run-time behavior of distributed applications from their execution logs. Specifically we mine templates and template sequences from logs to form a control flow graph (cfg) spanning distributed components. This cfg represents the baseline healthy system state and is used to flag deviations from the expected behavior of runtime logs. The novelty in our work stems from the new techniques employed to: (1) overcome the instrumentation requirements or application specific assumptions made in prior log mining approaches, (2) improve the accuracy of mined templates and the cfg in the presence of long parameters and high amount of interleaving respectively, and (3) improve by orders of magnitude the scalability of the cfg mining process in terms of volume of log data that can be processed per day.

We evaluate our approach using (a) synthetic log traces and (b) multiple real-world log datasets collected at different layers of application stack. Results demonstrate that our template mining, cfg mining, and anomaly detection algorithms have high accuracy. The distributed implementation of our pipeline is highly scalable and has more than 500 GB/day of log data processing capability even on a 10 low-end VM based (Spark + Hadoop) cluster. We also demonstrate the efficacy of our end-to-end system using a case study with the Openstack VM provisioning system.

1. INTRODUCTION

IT Operational Analytics (ITOA) aims at providing insights into the operational health of IT components deployed in a cloud datacenter. One of the strongest use cases of ITOA is driven by automated problem diagnostics and has recently received a lot of attention as a key differentiator of successful IT service vendors. A very challenging problem in this domain entails troubleshooting a distributed application deployed on the cloud by analyzing large volumes of machine-data collected from the different layers of the application, middleware, and infrastructure.

Not surprisingly, one of the primary sources of data used for mining operational insights is log data, given that execution logs contain rich information about runtime behaviour of a system. For example, warnings and error messages in logs give valuable clues to debugging the cause of the problem. But limiting the analysis to error/warning messages alone has severe limitations since several problems do not manifest as errors/warnings (leading to bad recall). Additionally, many errors/warnings are often benign and get reported even when the system is healthy (leading to bad precision).

We focus on a more robust approach of detecting anomalous runtime behavior by (a) mining/learning a healthy state model, and (b) comparing the runtime behaviour against the healthy-state reference model to locate deviations from expected behaviour. Some prior works [17, 31]) have proposed modeling the healthy state of a distributed system as a *control flow graph (cfg)* that stitches together various log message types or *templates* after mining the execution flows in the distributed system. In this paper, we adopt a similar approach, where we first mine *templates* from raw logs and then mine sequences from the templates in order to create a cfg. But contrary to prior works, we aim to do so without making some of the simplistic assumptions (summarized below and also described in detail in Section 5) made in prior works.

A template is an abstraction of a print statement in source code, which manifests itself in raw logs with different embedded parameter values in different executions. Represented as a set of invariant keywords and parameters (denoted by parameter placeholder $< P >$), a template when accurately identified can be used for summarization of multiple raw log lines that are instances of it. The parameter values change across different instances of same template, while the invariant keywords remain constant. The cfg represents the distributed control flow of a distributed system, where the sequence of templates is stitched together from execution flows. It is a graph where the nodes are templates and the edges represent the transition sequences from one template to another. The cfg graph represents all possible flow paths (within and across components) of a distributed system encountered when the distributed application runs.

The mining of cfg structures from execution logs may seem similar to the network inference problem [1, 10]. This addresses the problem of tracing paths of the underlying network using a generative model of cascades. Given the knowledge of when a particular node is infected (but not by whom), the challenge is to mine the unobserved network topology for each cascade. In comparison, there are two very important differences in the domain of mining cfg structure from execution logs. In the log domain, the nodes of the network are unknown and thus observations about when a particular cascade event occurred on a node cannot be made without mining the nodes themselves. We refer to this sub-problem of mining the nodes as the template mining problem. Secondly, each cascade in the network diffusion is identifiable using a cascade id. The execution logs from the distributed application domain does not guarantee that every transaction or flow through the system will be uniquely identifiable with a transaction id, and this results in interleaving of events across different simultaneously occurring transactions. Thus the cfg mining problem is inherently more complex in the setting of execution logs, since it has to be done without knowing the nodes of the network and without the presence of transaction ids.

To distinguish cfg mining from the process mining literature we note that process mining [6, 11, 18] assumes that the set of nodes is known in advance. It further assumes that recorded events contain a case/transaction id which ties together the set of events occurring

KDD '16, August 13-17, 2016, San Francisco, CA, USA
 ISBN 978-1-4503-4232-2/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939712

for each execution of a workflow/process. While this assumption of the presence of a clearly demarcated execution trace per transaction is reasonable for transactional systems like ERP and CRM, for the genre of distributed applications composed of various third-party microservices on the cloud this assumption does not hold.

The goal of this work is to overcome certain limiting assumptions made in the prior works in mining cfg of a distributed application. Firstly, some prior works rely on the availability of source code or binaries [31] to assist in the template mining phase, an assumption that does not hold in today's distributed applications composed of multiple third-party microservices. Amongst approaches that mine cfg from logs only, they make certain assumptions about parameters being small [19] or occurring towards the beginning of a template [9], or make assumptions about the presence of demarcated execution traces through the cfg [11, 18], or rely on explicitly embedding a transaction id parameter thats tracks a transaction flow across the distributed application [2, 13]. The goal of our work is to mine the cfg without making any of these assumptions.

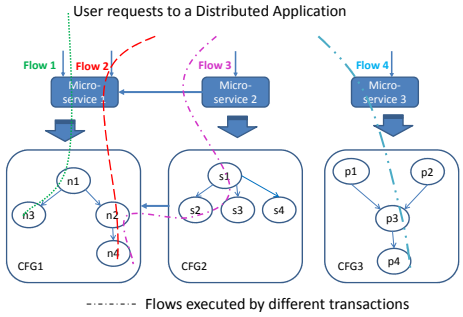


Figure 1: Transaction Flows in a Distributed Application

Figure 1 shows an example of a distributed application whose components are running on the cloud. Based on user requests coming to the application, one or more simultaneous flows are initiated which in turn invokes one or more microservice hosted on the cloud. Each transaction initiates a flow that executes one or more microservice deployed on the cloud infrastructure. The control flow of each microservice is represented as a cfg. The node within a cfg represents a template message that manifests itself in the execution logs of the microservice. The edges represent how the control flow passes between the nodes.

The figure shows four flows Flow 1 - Flow 4. Firstly, different transactions could generate simultaneous flows within the same cfg. For example, Flow 1 comprising (n1, n3) and Flow 2 comprising (n2, n4) traverses through the same cfg 1. The execution logs of microservice 1 will contain interleaved traces from both these flows while executing the same cfg 1. Secondly, transactions could also initiate parallel flows across different cfgs. For example, when Flow 1 and Flow 4 co-occur, cfg 1 and cfg 3 execute in parallel. Lastly, transactions could also initiate flows across microservices. For example, Flow 3 spans across cfg 2 and cfg 1.

To mine flows across cfgs, execution logs of all microservices comprising a distributed application needs to be merged. Our goal in this paper is to mine the cfg structures of the microservices and their cross dependencies from the merged execution logs across all microservices. The two sub-problems we strive to solve here are (a) mining the template messages while assuming no application specific knowledge about the print statements, its invariants or parameters and (b) mining cfg edges connecting the templates in the presence of interleaved logs without assuming the presence of explicit transaction ids demarcating different transactions.

1.1 Challenges in CFG Mining

Figure 2 shows a sample raw log input to our cfg mining methods. The raw logs are interleaved from multiple transaction flows that are executing a ground truth cfg (shown at the bottom of Figure 2). Each raw log statement has some invariant keywords and some variable

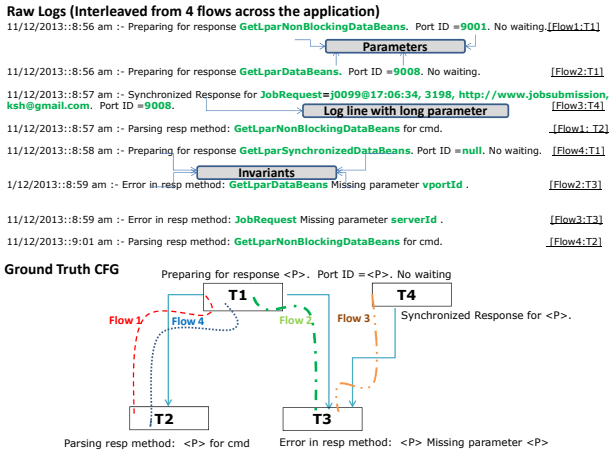


Figure 2: Raw Logs from Interleaved Flows

parameters. Sometimes the parameters can be as long or even longer than the keywords (e.g. Template T4 in [Flow 3: T4]). Each raw log instance is annotated with the tag of transaction-flow id:template id. Note that transaction ids are only present for understanding of the reader and in real logs we do not assume either transaction ids or template ids to be present.

In the absence of application specific knowledge, mining message templates from log lines incurs several challenges. This is because of the fact that parameters are indistinguishable from actual words, and can vary in number, in length, as well as their positions within a print statement. This causes instances of the same template to look very different or different templates with common parameters to look similar (see more examples of two classes of confusion in Section 2.1). For the above reasons, off-the-shelf text clustering techniques on the log lines does not work because of the complexities of semi-structured data with arbitrary behaviour of parameters. Further, running clustering on large volumes (several million lines) of log data does not scale.

In the absence of transaction ids, mining template sequences from interleaved log traces of different components or multi-threaded traces of the same component, also becomes a hard problem. Without clearly distinguishable transaction demarcations, which is a prerequisite for a suite of process mining techniques proposed in literature, the challenge lies in separating the noise from interleaved sequences. For e.g. in Figure 2, in the absence of transaction-flow ids, it is non-trivial to separate the execution path of each flow and build the cfgs. [Flow 3: T4] immediately follows [Flow 2: T1] and gives the false impression of an edge existing between nodes T4 and T1.

1.2 Highlights of our Approach

The key highlights of our end-to-end system called OASIS are –

Template Mining Phase - In the absence of application specific knowledge or assumptions on the usage of parameters, we observe that using text alone is not sufficient to accurately map log lines to a template (see prior works in Section 5.1). We attempt to overcome this challenge by using the multi-modal signal of text and temporal vicinity (i.e. predecessors and successors in the cfg) to further refine the mapping of log lines to templates (see Section 2.1).

CFG Edge Mining Phase - In order to mine the cfg structure from the interleaved logs in the absence of transaction-ids, we propose a novel two-phase approach of mining template sequences that leverages nearest-neighbor-groups (NNS) computed in the first phase to prune away noise due to interleaving, followed by a second phase that mines only immediate predecessors/successors of a template (see Section 2.2).

Anomaly Detection Phase - We describe how the mined cfg structure can be used to identify different classes of anomalies. We describe the architecture and implementation of an end-to-end system

that detect anomalies in deployed distributed applications by analyzing their execution logs (see Section 3).

Evaluation of our Approach - We empirically evaluated the effectiveness of our template mining and cfg edge mining algorithms in comparison with alternative design choices, using synthetic log traces and as well as several real-world log datasets. We also demonstrate that our overall cfg mining pipeline has good parallelizability and is able to process more than 500 GB/day of log data even on a 10 low-end VM based (Spark + Hadoop) [24] cluster. Finally, we demonstrate the efficacy of our end-to-end anomaly detection system using a case study with Openstack [21] VM provisioning system.

2. OUR CFG MINING TECHNIQUES

We now describe our offline analytic models of template and sequence mining, both of which together enable us to compute the cfg nodes (templates) and cfg edges (sequences).

2.1 Mining of Templates

The same template would manifest as multiple log lines based on the embedding of the actual values of parameters. There are primarily *two classes of confusion* created by parameters and invariant keywords in a print statement when attempting to mine templates: (i) C1: Log lines resulting from the *same* template can look different textually when the template has multiple or long parameters values, and (ii) C2: Log lines resulting from two *different* templates can look alike textually when they share common long parameters.

Prior works in mining templates from logs (see Section 5.1) largely rely on some form of text-based clustering of similar log lines in order to compute templates. These text-based clustering approaches combat the above two classes of confusion by relying on either application specific knowledge of the types of parameters (such as IP address, requestID etc.), or assume that a template has few parameters and that too comprising of a single word each, or assume that parameters even if long are typically at the end of the print statement. Although these assumptions are indeed true for a subset of datasets, we found these assumptions to *not* hold for many of the datasets we experimented with.

In the absence of application specific knowledge, we make the observation that using text alone is never going to be sufficient to generically map log lines to a template. Building on this intuition, we propose a novel *multi-modal* algorithm for using temporal vicinity of a template (i.e. templates that typically occur in time before it or after it) as an additional signal to textual content of a template, in order to mine high quality templates without making any application specific assumptions. The intuition behind using the joint-signal of text similarity and temporal vicinity similarity is that *ideally* two instances of the same print statement should not only look similar w.r.t. the text-similarity, but they should also have similar temporal vicinities. W.r.t. confusion class C1, observe that although two instances of the same print statement might have a low text similarity, they can still be merged into the same template cluster if their temporal vicinities are very similar. W.r.t confusion class C2, even if two log line instances corresponding to two different print statements might look similar if they share a long common parameter and also have additional overlapping words, their temporal-vicinity will be quite different and thereby will not be eligible for merging. Thereby, using the temporal vicinity similarity information in addition to the text similarity information, we can overcome the fundamental challenges created by the two classes of confusion described above.

Contrary to prior works that mine templates using the textual content of the log lines alone, our multi-modal approach of template mining would require us to mine the temporal vicinity of each log line as well. Given that each log line can be potentially unique because of the unique combination of the multiple embedded parameter values in each print statement, mining temporal vicinity at the granularity a log line would not give us statistically significant information about the observed predecessors and successors of a log line. This leads to a *seemingly* chicken-and-egg situation, wherein mining templates requires us to mine sequences, and mining sequences requires us to mine some notion of template that aggregates multiple instances of the same print statement. We resolve this dilemma by not directly operating on raw logs but transforming the raw logs into an intermediate form of *approximate templates*.

To construct approximate templates, we make a first pass on the raw log lines in order to compute a frequency histogram comprising of every distinct keyword that was observed in the logs. The underlying intuition here is that the invariant template words are much more frequent in the logs than the parameters. A change point detection algorithm that operates on frequency analysis of the unique words in the logs, detects words with frequencies beyond the change point threshold as *dictionary* keywords. We transform the log lines to approximate templates by iterating over the words in the log line in order, and retaining words in the log line that are present in the computed dictionary and replacing the words that are not in the dictionary by a parameter placeholder $< P >$. At the end of this dictionary based log line transformation phase, the output is a list of approximate templates which are several orders of magnitude less than the number of raw log lines. In our experiments, we typically observed more than 1000x compression ratio (i.e. #log-lines : #approximate-templates).

Inspite of this reduction from log lines to approximate templates, we nevertheless observe that in many datasets, a modest fraction of approximate templates still have support values below the minimum threshold (i.e. support) required to mine statistically significant temporal vicinities of these approximate templates (using our algorithm described in the next Section 2.2). To overcome this limitation, we then run a text-based clustering at a very high threshold (here 90%) of edit-distance based similarity to further cluster different approximate templates which correspond to the same ground truth template or print statement. Note that we need to use a very high text-similarity threshold in this phase to ensure that two different print statements are *extremely* unlikely to get mapped to the same *approximate template cluster*.

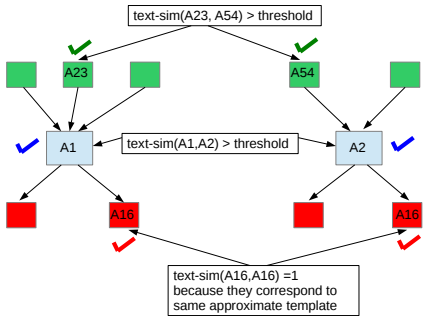


Figure 3: Example multi-modal similarity check

At this point, although the approximate template clusters are quite noise-free (i.e. their constituent members indeed correspond to the same print statement), the issue that needs to be overcomed is that we mine multiple clusters for the same print statement when the number of parameters is large or parameter values are long. Consequently, to refine these text clusters further, we then resort to our multi-modal clustering approach that merges clusters using the multi-modal signal of the textual content of cluster label that is computed using a Longest Common Subsequence (LCS) of the text in the constituent members, coupled with the computed statistically significant predecessor(s) and successor(s) of these clusters.

Figure 3 shows an example *multi-modal similarity check* that is performed for two approximate template clusters A1 and A2. Lets say we mine 3 parents and 2 children of A1, and similarly, 2 parents and 2 children of A2. In contrast to checking the text similarity of the reference cluster labels of A1 and A2 alone (as was done in prior text-alone approaches), we also compute the text similarity between any parent of A1 and A2, and also between any child of A1 and A2. Based on the comparison, if we find that these clusters have a good text similarity match between any of their respective parents (A23

and A54) and also between any of their respective children (here shared approximate template A16), then the two reference approximate clusters A1 and A2 satisfy the multi-modal similarity check.

Note that in comparison to text-only clustering approaches, our multi-modal approach can set relatively lower values of the text similarity threshold in order to combat confusion class C1 described above, and still ensure that we do not succumb to the bad merges that could have been potentially caused by confusion class C2.

Our algorithm however needs to handle a few special scenarios – (a) when both of the clusters being compared do not have a child (this happens for leaves in the cfg), we need to discount the child direction text similarity match, and similarly (b) when both the clusters being compared do not have a parent (this happens for root nodes in a cfg), we need to discount the parent direction check.

Worth admitting here, is that there are some rare scenarios wherein our multi-modal algorithm is unable to merge two instances of the same print statement. This happens when the transaction flows producing the two instances of the same print statement, have different incoming paths and different outgoing paths in the underlying ground truth cfg. In such a scenario, the temporal vicinity of the two instances of the print statement are different in reality and thus the multi-modal check fails. This is the primary reason why our evaluation of our multimodal agorithm in Section 4.4 demonstrates that although multi-modal performs superior to using pure text-based clustering, we still end up with some print statements having multiple unmerged cluster instances (leading to reduced precision).

Additionally, note that our multi-modal algorithm (in contrast to pure text-based clustering) depends on the accuracy of the mined predecessors/successors of the approximate template clusters. We observed that because of this dependence, errors in the mined cfg can negatively affect the accuracy of mined templates in the multi-modal algorithm – leading to occassional merging of two different print statements into a cluster whose's LCS label becomes corrupted (leading to reduced recall) and sometimes also leading to fragmentation of the same print statement into multiple cluster instances (leading to reduced precision).

2.2 Mining of CFG Edges

As described in the previous section, our multi-modal template mining approach requires us to compute the predecessors and successors of the approximate template clusters. Even if we had magically mined the intended ground-truth templates, this sequence mining phase would nevertheless have been needed in order to mine the cfg edges corresponding to the sequences between the mined templates. The challenge of this phase is to be able to robustly mine statistically significant predecessors and successors of the approximate templates when operating on interleaved logs. Interleaved logs result from multi-threaded executions of the same component (*intra-level parallelism*). Further, given our intention of mining control flows across different distributed microservices (e.g. the assignment of map tasks from a Hadoop Jobtracker node to a Datanode), interleaved logs also result from having to intentionally merge logs from different distributed components (*inter-level parallelism*).

As described in Section 5.2, although classical sequence mining approaches are not directly applicable in our problem setting of interleaved logs because of the lack of demarcated transactions, one can adapt these algorithms to work in our setting by chunking the interleaved logstream based on time-windows and creating a transaction id per time-window [20]. The duration of the time-window can be set to maximum (or 99th percentile) edge lag in the application.

Figure 4 describes the inner workings of how such an approach would mine predecessors/successors. As an example, a sub-goal is to mine the children of T2, i.e. mine the existence of the edge T2 $\rightarrow$ T3 and T2 $\rightarrow$ T6. Tracking the immediate successor of each template does not work in the interleaved logs setting since the next log line could very well be noise from either intra- or inter- parallelism. What is needed is a lookahead window beyond the immediate next line, and incrementing the counts of all succeeding templates occurring within the lookahead window. Although this lookahead window

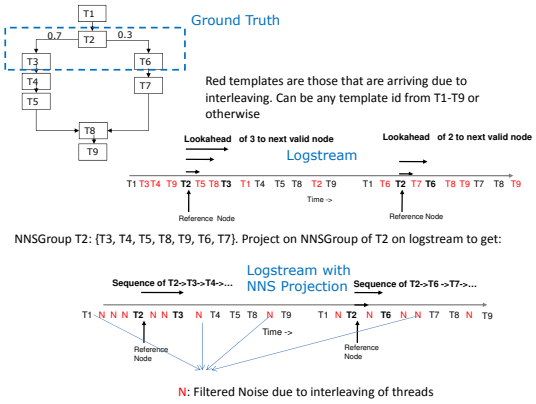


Figure 4: Our two-stage sequence mining algorithm as compared to classical single-stage sequence mining approach

based bookkeeping would enable mining of the edges from the reference node (here T2) to its children (here T2 $\rightarrow$ T3 and T2 $\rightarrow$ T6), the limitation and concerning issue with this approach is that the algorithm will also mine edges to the downstream cfg descendants of the reference node (i.e. T4, T5, T7, T8, T9). Consequently, instead of mining the two desired children of T2, the algorithm will end up mining lots of redundant edges from the reference node to the downstream descendants. Although one may envision running a transitive-edge removal algorithm [4] on the bloated cfg mined by this strawman algorithm, note that the transitive-edge removal could end up removing potential *genuine detour paths* from a parent to its downstream grandchild. For many practical datasets, the occurrence of multiple detour paths existing between two nodes is fairly common, and standard transitive edge removal will thereby lead to mining an incorrect cfg.

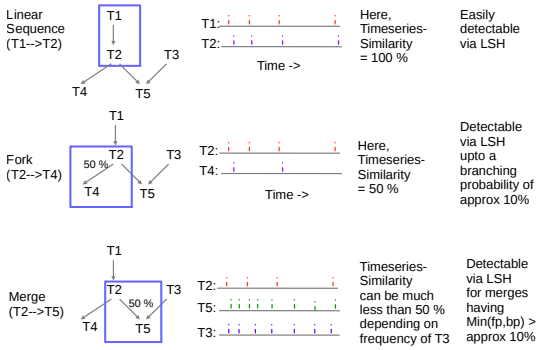


Figure 5: Nearest-Neighbor (NNS) group computation using template time-series

To overcome the limitations of the above strawman approach, we propose a novel two-stage sequence mining approach. In the first-stage, we compute *Nearest-Neighbor-Groups (NNS) groups* of each template, and the second stage leverages this pre-computed NNS group information to mine *only the immediate* successors/predecessors of each template. The intuition behind our two-stage algorithm is that if we find that the immediately succeeding log line of a reference template-id in the input logstream is not a part of the NNS group of that reference template, then it implies that the occurrence of this succeeding log line is essentially noise resulting from the interleaving. Before we describe the details of our approach, we first describe what we mean by NNS groups, followed by how it helps us avoid the limitations of the previous single-stage approach.

The NNS group of a template is a set of other templates that are observed to statistically *temporally co-occur* with the reference template. NNS groups of each reference template can be computed

by analyzing the *time-series* of each template which captures the timestamps or chunked time-bins wherein that particular template occurred in the logstream. Given the time-series of each template the algorithm to mine the NNS groups of a template depends on the type of cfg sub-structure we want to mine.

Figure 5 describes the three fundamental *cfg sub-structures* we want to mine, and how the time-series of the nodes in those sub-structures relate to each other, and the algorithm we should use in order to mine these sub-structures. In particular, the cfg comprises of three types of sub-structures – linear sequences, forks, and merges. Depending on the type of sub-structure, the time-series of the nodes in the sub-structure have different extents of similarity. As depicted in the figure, time-series of nodes in the linear sequence have almost a 100% similarity in their respective time-series. For forks, the similarly depends on the branching probability between the parent and the child. For merges however, the time-series similarity can be low inspite of a high branching probability between the parent and child. In general the algorithm to compute the NNS groups depends on the type of sub-structure we want to mine. The scalable Locality-Sensitive-Hashing (LSH) [12] based Minhash [5] algorithm that detects jaccard similarity is a good candidate for detecting the linear sequences, forks, and a constrained setting of merges that are not very skewed (i.e. time-series similarities $>$ approx 10%). To mine arbitrary skew in forks and merges however, one would need to resort to the bayesian approach of mining forward (fp) and backward conditional probabilities (bp) [29].

When mining correlations using either the jaccard similarity or the bayesian conditional probability approach, we ensure that the correlations are not coincidental by incorporating the *null-invariant* check [29]. For this, we measure the ratio of the conditional probability of an event as compared to the independent probability. Essentially, if the ratio of (P(A/B) / P(A)) is above a *non-indepedence threshold*, then it implies that the two events are not independent, since P(A/B) should be approx P(A) if the two events A and B had been independent.

The second-stage of our sequence mining approach leverages the pre-computed information of the NNS group of each template, and only increments the counter of the first template that succeeds the reference template pre-conditioned on the fact that the succeeding template must belong to the NNS group of the reference template (here T2). Further, we can preempt the lookahead, the moment such a template is found, thereby implying that we create a successor edge only to the first NNS group member that follows the reference template. Another way of interpreting this algorithm is that in order to create the successor edges of a reference template, we consider a projection of the input logstream on the NNS group of that reference template, as shown in Figure 4. This projection essentially implies that the noisy candidates due to interleaving do not exist in the projected logstream.

This second-stage of this algorithm can be realized by making a single pass of the input logstream, and tracking for each NNS group, the latest occurring NNS group member ($lastmemberseen$) and the time of this last occurrence ($lasttimeseen$), and creating edges only when the lastmemberseen is the same as the reference template of the NNS group and the ($currenttime$ - $lasttimeseen$) $< lagthreshold$, wherein lagthreshold is an application specific threshold based on expected edge lags. On observing a template, we update the $lastmemberseen$ and $lasttimeseen$ in all the NNS groups this template is part of.

For each template, once the successor/predecessor edges and also the relative normalized values of the counters of the NNS group members is computed, the relative branching probabilities of the CFG edges can be computed. Further, the temporal vicinity of each template can be stitched together to construct the desired CFG.

2.3 Scalability and Parallizability of our CFG Mining Pipeline

Given that the log datasets of typical applications are very voluminous, it is imperative that the algorithms involved in the above different stages of the pipeline be parallelized in order to enable scalable mining of the cfg from large volumes of training data using multiple cores or a cluster of machines (i.e. worker nodes).

(*Stage 1*) The approximate template generation phase can be easily parallelized across multiple worker nodes by splitting the log data into multiple splits. This stage in the first round computes the frequency histogram of the keywords in a MapReduce fashion, followed by broadcasting of the word dictionary computed from the frequency histogram using change point detection, followed by the generation of the approximate templates using the dictionary lookup based transformation of the log lines.

(*Stage 2*) The second stage of high similarity edit-distance based text clustering is done efficiently via Minhashing [5] the tokens of the approximate templates to efficiently identify other potential approximate templates that are very similar. The actual edit-distance is calculated between the Minhash pruned candidates, and members having edit-distance similarity above a threshold are merged to form high-similarity text clusters.

(*Stage 3*) To parallelize the third stage of NNS Group computation, we need to essentially parallelize the computation of the pairwise time-series correlation metric that is based on the bayesian conditional probability coupled with the null-invariant check. To do this, every worker node computes the partial time-series of every template from the timeline fragmented portion of the logs hosted by the worker, followed by $O(N)$ computation of the partial length of each time-series and the $O(N^2)$ computation of the pairwise overlap between the partial time-series of each template. The final pairwise correlation metric (based on entire time-series) can be efficiently computed in a distributed way by leveraging the partial length/overlap statistics computed (in parallel) on the different worker nodes.

(*Stage 4*) The fourth stage of mining the predecessors/successors makes one pass on the logs and this phase can be parallelized by broadcasting the computed nns-matrix (typically sparse) to each worker node, computing the partial edge statistics on respective portions of the timeline based log splits, and then aggregating the partial statistics to compute the relative branching probabilities for each approximate template cluster.

(*Stage 5*) Finally, we have the fifth stage of multimodal template refinement. Since this stage operates only on a small number of high-similarity edit-distance based template clusters and not on the raw logs, we avoid parallelizing this phase. Moreso, this stage cannot be trivially parallelized and thus we intentionally chose to have a centralized implementation only. However, to nevertheless speedup this phase, we employed scalable LSH-based Minhashing [5] on the text tokens of the approximate template cluster labels to efficiently identify potential candidates for the multimodal similarity check.

3. ARCHITECTURE OF OUR END-TO-END ANOMALY DETECTION SYSTEM

In this section, we describe the system architecture and implementation of our end-to-end log analytics system called OASIS (see Figure 6). Logs from various services are ingested using Logstash [16] adapters, routed through Kafka [14], and indexed into ElasticSearch [7]. Data becomes available for search immediately. At the same time, data is also passed onto the (Spark + Hadoop) [24] layer. The two components in OASIS are described below:

3.1 Offline Mining of CFG based Model

At the heart of OASIS lies the offline analytic models comprising of template and cfg mining (shown in blue in Fig.6) using the algorithms we described in Section 2. The offline training models comprise of a set of templates along with the cfg edges having annotations of branching probabilities and expected edge lags.

3.2 Online Anomaly Scoring

The online anomaly scoring module of OASIS uses the offline models of the mined templates and cfg. As shown in Fig.6, this comprises of two modules –

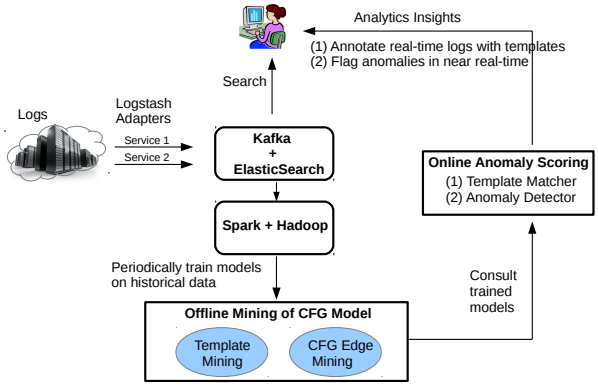


Figure 6: OASIS Architecture

Template Matcher: At runtime, as log lines are ingested, the offline computed dictionary is consulted to detect the invariant keywords in the incoming messages and then mine the approximate template in the same way as in the offline model building phase. Further, using the previously stored mapping of the approximate templates to the final refined templates got via multi-modal template mining, we map in real-time the incoming log line to a previously mined refined templates.

Anomaly Detector: Logs annotated with template ids are passed through an anomaly detection routine. In OASIS, anomalies have two attributes: (1) type of anomaly and (2) score of anomaly. The type of an anomaly signifies the underlying reason of why it is an anomaly. The score of an anomaly signifies its severity. There are two types of anomalies viz. (a) sequence anomaly and (b) distribution anomaly.

A *sequence anomaly* is raised when none of the children of a parent node is seen within an expected time lag interval. These are raised instantaneously when the the expected duration timeout occurs. Hence this type of anomaly messages are more real-time. The advantage of sequence anomalies are that they are very few in number and high precision. Also most sequence anomalies can generate actionable insights i.e. deeper investigation into why a particular control flow was interrupted at a particular node. We also assign a score to the sequence anomaly to denote our confidence that this anomaly was not an artifact of the occurrence of some low branching probability edge that got pruned away as a noisy edge because of its branching probability being below the 10% branching probability threshold. So, we use the aggregate children probability as the anomaly score, with higher scores denoting that we are more confident.

A *distribution anomaly* is raised when the edge probabilities between parent and any of its children changes. This type of anomaly can identify shifts in the observed branching probability distribution at a cfg node. However, its critical that we design the scoring metric in a way, that for inherently fluctuating workloads, we do not raise false positives. To cater to this, we track the observed branching probabibility distributions over the entire course of the healthy run of the system. To do this, for each node, the branch probabilities are calculated periodically on non-overlapping windows over the entire span of the training log, and representative branching distributions materialized in a set called Expected Set. During anomaly detection, we calculate a diff-score of the observed branching distribution with every distribution present in the Expected Set, and use the minimum diff-score as the distribution anomaly score.

4. EVALUATION

4.1 Datasets

We evaluate our system using both real-world and synthetic datasets. For the cfg mining technique, we use two real datasets – (i) *linux*: Linux syslogs, and (ii) *distMW*: a proprietary distributed middleware application. For evaluating our end-to-end anomaly detection system, we do a special case study with a third real-world dataset *openstack* [21]: a popular cloud VM provisioning system which comprises of multiple distributed microservices. Given that our techniques rely on exploiting timing correlations, we ensured that our techniques are not negatively impacted by large machine clock drifts, by running the distributed applications either in standalone mode (i.e. all different microservices running on the same machine) or on machines that are NTP time synchronized (a fairly common scenario).

In addition to the above three real-world log datasets, we evaluate each phase of our pipeline (i.e. template mining, mining of cfg edges, anomaly detection) using synthetic logs generated by a log generator, which we built specifically for this purpose. The synthetic log generator has three modules – the cfg edge generator, the cfg template generator, and the execution simulator.

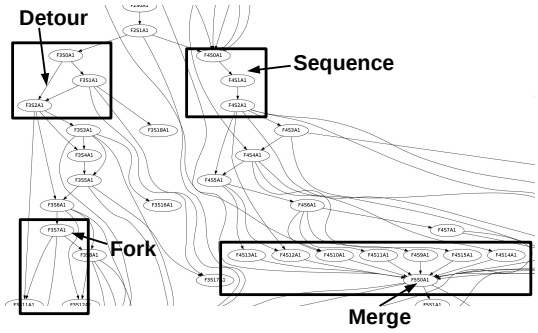


Figure 7: A portion of a synthetically generated cfg

The **CFG Edge Generator** builds the cfg edges for a hypothetical microservice from basic programming constructs. Figure 7 shows a portion of a sample synthetic cfg that is generated. As depicted in it, observe that it models all types of fundamental building blocks like linear sequences, forks, and merges, and also has detour paths. Further, edges are characterized by a lag representing the delay between the two print statements that are executed in succession, and also a branching factor probability.

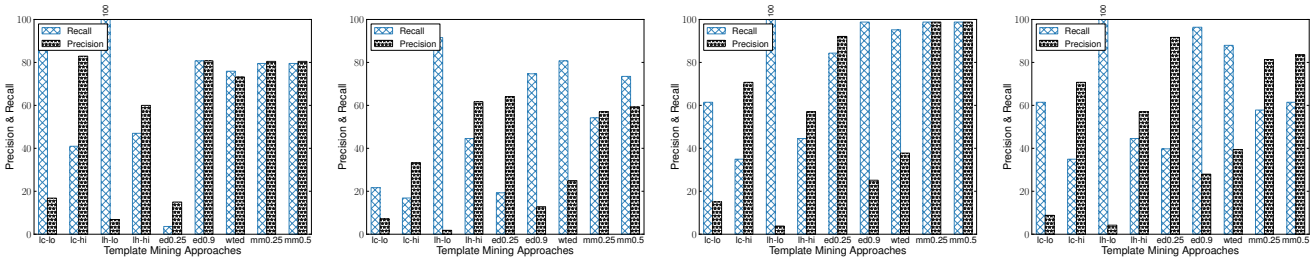
The **CFG Template Generator** determines the print statement corresponding to every node in the cfg. The invariants are selected from a global dictionary of words and parameters could be fixed or variable length. The number of parameters and their positions and the length of templates are chosen from a Gaussian distribution. Parameter values, determined at runtime are chosen either from a parameter space or inherited from their cfg ancestors.

Once the cfg is generated, the **Execution Simulator** generates paths based on branching probabilities. As the paths are traversed, each path outputs its print statements after simulating time equivalent to the corresponding edge lag. Each path reflects the execution of a thread. Threads arrive into the system according to a Poisson process and are traversed simultaneously, leading to multi threaded behaviour. Multiple cfg executions are also simulated.

4.2 Competitive Approaches

For comparison, we obtained the state-of-the art tools (from authors of prior works, and open source binaries) and ran them against the same datasets.

For template mining, the alternative approaches compared are: (i) *logcluster (lc)*: clustering approach on a set of approximate templates [28], (ii) *loghound (lh)*: frequent item-set mining on the words in each log line [27], (iii) *edit-distance (ed)*: Levenshtein's text similarity metric, (iv) *weighted-edit-distance (wted)*: modified similarity metric with higher weights on words at the beginning of templates [18]. Further, to cover a representative spectrum of possible configurations of competitive approaches, we report high(1%) and low(0.01%) support versions of lc and lh approaches, and high(0.9) and low(0.25) text similarity versions of edit distance algorithm. All



(a) Word:Param=0.9 ($\delta = 0.7$) (b) Word:Param=0.5 ($\delta = 0.7$) (c) $\delta = 0.85$ (Word:Param=0.75) (d) $\delta = 0.5$ (Word:Param=0.75)

Figure 8: Template Mining Precision/Recall as a function of word:parameter ratio and discriminative property in synthetic dataset

these approaches are compared with our multi-modal, which uses temporal vicinity along with text to compute refined templates.

For cfg edge mining, the competitive approaches compared are: (i) *SM-vanilla*: classical sequence mining (SM) algorithm (PrefixSpan [22]), (ii) *SM-tred*: this employs classical sequence mining followed by a step of graph transitive edge reduction in order to remove redundant descendants, (iii) *SM-wtred*: classical sequence mining with the transitive edge reduction step prioritizing which edges to remove based on weights [4] and using Jaccard association-rule mining metric as the weight, (iv) *PM*: classical process mining (PM) using the Fuzzy Miner [11], (v) *PM-noCID*: the MIM process mining tool [8] that attempts mining of process workflows from interleaved logs even in the absence of transaction or case ids, (vi) *WM*: workflow-miner [17]. Note that classical SM and classical PM approaches work on the basis of transaction ids. Since transaction boundaries are not present in interleaved execution logs, our approach for creating transformed inputs for classical SM and PM is to time-slice the logs into chunks based on a configurable time-window [20], and assign the same transaction id for all logs in the same time-window.

4.3 Methodology

We use the standard Precision/Recall metrics for evaluating our template mining, cfg edge mining, and anomaly detection phases. Recall is the fraction of the ground truth templates, edges, or anomalies that were mined. Precision is the fraction of mined templates, edges or anomalies that were present in the ground truth.

For synthetic logs, the comparisons were made with the available ground truth generated by our tool. For ground truth generation on real logs, we took the following approach: (a) subject matter experts were asked to identify the distinct ground truth templates present in the logs, (b) thread ids in real logs were used to separate out the valid ground truth sequences (note that threadids were used only for ground truth sequence generation, but intentionally not leveraged by our sequencing algorithms), (c) For logs where there were no threadids, we asked domain experts for some known sequences and compared our precision in being able to mine them.

4.4 Mining Templates

We now evaluate template mining performance for competitive approaches mentioned in Section 4.2, for different characteristics of the logs: its average word to parameter ratio and the discriminative property of each template that denotes how distinct each template looks from the others. These two metrics captures the confusion added by long parameters or inherited parameters respectively.

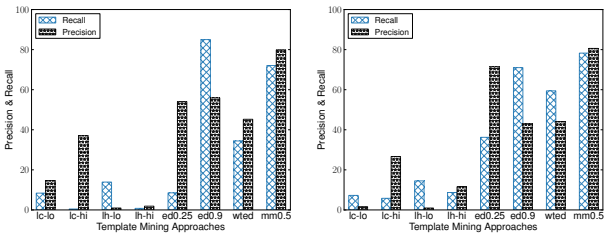
Figure 8(a) and Figure 8(b) report how precision/recall vary as average word to parameter ratio changes, keeping discriminative property fixed. In contrast, Figure 8(c) and Figure 8(d) report how precision/recall vary as discriminative property changes, keeping average word to parameter fixed. In these experiments, we intentionally used a single-threaded execution of a single cfg (having 100 templates), so that our proposed multi-modal algorithm has perfect information with respect to temporal vicinities. Note however that the real datasets in contrast have interleaved events from multiple threads and multiple microservices.

In Figure 8(a), we first observe that lh and lc approaches at low support have the best recall but suffer largely in terms of precision because many ground truth templates gets fragmented into multiple mined templates. In contrast, the high support variants have great precision but low recall, implying that they are unable to mine many low frequency patterns. For the edit-distance approaches, they are very much sensitive to the similarity threshold parameter. Edit-distance at 0.25 suffers from poor recall as they create many bad merges of slightly similar looking templates. In contrast, multi-modal even when run at a very low similarity threshold (0.25) effectively filters out noise using the temporal vicinity signal and consistently reports better recall and precision.

In Figure 8(b), where parameters occupy as much as of the log line as invariants, the precision of edit-distance at 0.9 drops severely since different instances of the same print statement look different and result in multiple mined templates. Edit-distance at 0.25 continues to have bad recall since the bad merges causes the cluster labels to become corrupted resulting in several of the ground truth templates not being identifiable in the list of mined templates. The multi-modal is the only approach that does fairly on both recall and precision front (both 60%).

In Figure 8(c), we observe that when templates are highly discriminative ($\delta = 0.85$), even edit-distance at 0.25 works quite well. However, in Figure 8(d) when δ drops ($\delta = 0.5$) and templates start looking similar, the recall begins to degrade. This is because with more confusion, mining actual ground truth templates become harder. However multi-modal even in this setting does fairly well on both recall and precision front.

We now report results on real datasets of distMW and linux in Figure 9(a & b). For distMW, there were 3000 ground truth templates, with word:parameter ratio = 0.5 and $\delta = 0.5$. Consistent with our observation on synthetic logs, multi-modal has relatively better precision and recall, while other approaches suffer either in precision or in recall. This is because the parameters were extremely long and complex. For linux, the computed ground truth templates was 80 while the word:parameter ratio =0.8 and δ = 0.6. Here the edit distance approaches work fairly well w.r.t precision because of relatively shorter parameters. However at δ = 0.6, templates were still fairly similar and the ability of multi-modal to disambiguate using temporal vicinity helped it dominate the results.



(a) Template Mining: distMW (b) Template Mining:linux

Figure 9: Template Mining on Real Datasets

	Intra-Parallelism Among Threads (varying t)				Inter-Parallelism among CFGs (varying n)		
	(n=1,t=1,s=300)	**(n=1,t=2,s=300)**	**(n=1,t=5,s=300)**	**(n=1,t=10,s=300)**	**(n=2,t=1,s=300)**	**(n=5,t=1,s=300)**	**(n=10,t=1,s=300)**
OASIS	P=100.0,R=97.1	P=100.0,R=96.2	P=99.8, R=91.9	P=98.6,R=88.8	P=100.0,R=97.1	P=100.0,R=92.7	P=100.0,R=82.3
SM-vanilla	P=15.8,R=99.6	P=8.6,R=99.6	P=2.6,R=100.0	P=1.5,R=100.0	P=11.9, R=86.0,	P=6.5,R=87.7	P=3.0,R=88.9
SM-tred	P=66.4,R=47.6	P=56.7,R=43.9	P=43.5,R=36.9	P=32.5, R=31.1	P=76.6, R=51.9	P=49.0,R=37.4	P=45.1, R=37.3
SM-wtred	P=91.8,R=88.6	P=87.5,R=86.9	P=84.2,R=82.7	P=82.9, R=79.8	P=91.1, R=80.6	P=89.3,R=81.4	P=87.1, R=81.3
PM	P=96.8,R=97.7	P=81.3,R=97.1	P=72.5,R=84.6	P=39.0,R=58.6	P=91.0,R=91.8	P=61.9,R=53.0	P=24.0,R=29.3
PM-noCID	P=64.0,R=78.2	P=58.7,R=73.5	P=45.8,R=59.1	P=24.0,R=40.0	P=64.0,R=69.8	P=27.6,R=41.5	P=10.0,R=24.0

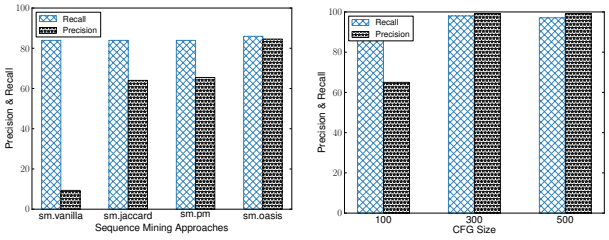
Table 1: CFG Edge Mining with various levels of interleaving (n: #cfgs, t: #threads per cfg, s: cfg size)

The overall template mining results show that OASIS's use of multi-modal classification enables it to have better accuracy over other best known alternatives. In our evaluation of multi-modal on the synthetic dataset, we assumed that the ground truth cfg was known. However since multi-modal algorithm depends on the accuracy of the mined cfg in real datasets, the error in the mined cfg can affect the accuracy of mined templates in some cases. We acknowledge that accurately classifying log lines to ground truth templates in the absence of any application-specific knowledge of the usage of parameters, is a hard problem, and call for more further research in this direction.

4.5 Mining CFG Edges

We now evaluate cfg edge mining performance for the different competitive approaches mentioned in Section 4.2, from the perspective of how efficiently they can mine cfg edges from interleaved logs.

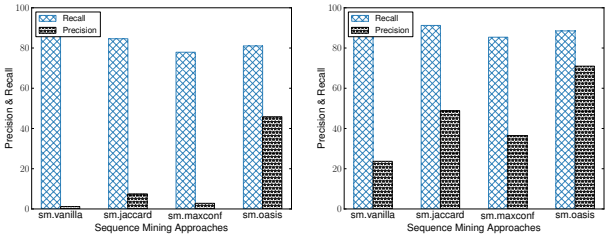
We first simulate a distributed application having 10 cfgs executing in parallel, where each cfg has 10 threads executing within it. Keeping the mean cfg size constant at 300 nodes, Figure 10(a) captures the performance of all the competitive approaches in this (10cfg-10thread) setting. Note that the accuracy of PM approaches suffer because they either expect caseIDs to be explicitly present (i.e. PM) or try to implicitly mine them using generative models (i.e. PM-noCID). Classical sequence mining SM-vanilla has high recall but it mines a large number of spurious edges compromising precision. Transitive reduction on the mined edges using SM-tred improves precision to some extent but also removes valid detour paths, thereby lowering recall. But SM-wtred that intelligently decides which edges to remove has a decent precision and recall performance. OASIS significantly out-performs all approaches with a precision $> 90\%$ and recall $> 82\%$. Also note that our branching factor threshold for edges is 10%, which implies that the recall applies to all edges above that. We observed that even setting a branching threshold of 25% results in above 95% recall for OASIS.



(a) Synthetic Distributed App (n=10,t=10) (b) OASIS (varying cfg size)

Figure 10: CFG Edge Mining in synthetic distributed application setting

Readers may also notice we leave out WM [17] from our results. This is because WM failed to produce any output and crashed in our setting since this approach is designed to only work on the prerequisite that *all* the ordered events within each transaction flow (i.e. one *complete path* thru the cfg) is contained in its entirety within each trace. In contrast to classical process mining, their work is designed to cope with multiple parallel transactions within each trace, but they require multiple such traces as input to their algorithm. In our setting where we have a single log, our only option of chunking the logstream using our time-windowing approach violates their strict complete-path pre-requisite. We confirmed this limitation of their technique by creating another synthetic dataset that respects complete-path boundaries, and as expected, we observed this approach to run successfully and also give good results.



(a) SM: distMW (b) SM: linux

Figure 11: CFG Edge Mining on Real Datasets

Next we delve deeper into trends as a function of amount of intra- and inter- parallelism. Table1 shows the results for (n,t,s) configurations where n: #cfgs, t: #threads per cfg, and s: cfg size. We vary t (intra-level parallelism) and n (inter-level parallelism) at s=300. For a single threaded execution, most approaches do well, with the noted exception being SM-vanilla where precision suffers because of the large number of false edges mined. As t increases, the precision of other algorithms drop considerably, whereas the precision of OASIS remains very high (98.6%). Similar observation is made when n increases viz. the precision of other algorithms drop considerably, whereas OASIS has 100% precision. Overall, amongst the alternative approaches, we observe that SM-wtred is the second best after OASIS in both the intra- and inter- parallelism setting.

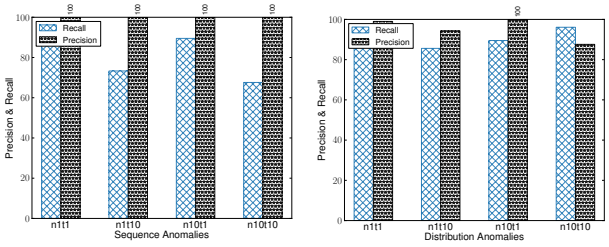
We now experiment with differently sized cfgs having 100, 300 and 500 nodes, each of which execute 10 parallel threads. Figure 10(b) shows that interestingly the precision/recall numbers improve with larger cfgs. This is because when the cfg has fewer nodes with a large number of parallel threads executing it, it is harder to distinguish the true neighbor of a log message from noise. For this reason the OASIS precision for 60 nodes drops to 65.8%. But with larger cfgs, OASIS is able to successfully weave out the noise and construct cfgs with very high accuracy.

Having established OASIS's capabilities on simulated logs, we now move to real data. Figure 11(a) and Figure 11(b) show results on distMW and linux respectively. Here we discarded PM because of their very bad performance in synthetic datasets. Overall, we observe that OASIS has the best performance and SM-wtred is second best, as per our observation in synthetic datasets.

The overall cfg edge mining results show that OASIS significantly outperforms best known alternatives. The only threat to validity we acknowledge is when OASIS is unable to disambiguate confusion due to high intra-level parallelism in small sized cfgs.

4.6 Anomalies

In Figure 12 we report the precision and recall on Single-threaded (n=1,t=1,s=300), Intra (n=1,t=10,s=300), Inter (n=10,t=1,s=300) and Realistic (n=10,t=10,s=300) configurations. In these experiments, the anomalies were injected at a small fraction (approx 10%) of random nodes in the cfg.



(a) Sequence Anomaly (b) Distribution Anomaly

Figure 12: Anomaly Results

For sequence anomalies (Figure 12(a)), we observe that precision is very high in all settings. In the case of single threaded and inter setting, the recall is approx 90%. The misses happen when the anomalies are generated at parent nodes of popular merge points. This is because, inspite of the injected sequence anomaly at the parent node, a popular merge point is observed to occur via the path of another parent. At this point, the anomaly detection algorithm is unable to differentiate the multiple paths to the merge point and will not raise the anomaly. This occurrence is magnified in the t=10 settings and the lower recall can be explained by multiple threads through the cfg being processed in parallel.

For distribution anomalies (Figure 12(b)), the precision and recall scores are overall quite satisfactory. We observe that the cause of the occasional misses and false alerts are due to the fact that we have not captured the entire spectrum of inherent fluctuations in the workload due to larger time-windows over which the branching probability distributions are being measured. One way to improve performance is to use smaller time-windows. Larger volumes of training data would also improve performance since it would capture all potential benign fluctuations.

4.7 Scalability

Finally we comment on the performance of each of the five stages (see Section 2.3) of our cfg mining technique in terms of their processing time. The pipeline comprises of the Spark [24] implementations of the four stages discussed in Section 2.3 viz. ApproxTemplateMiner, TemplateTextClustering, NNSGroupComputation, CFGEdgeMiner. For the reasons described in Section 2.3, the MultimodalRefinement phase remains a centralized implementation. The pipeline is executed on a 10 VM (Spark + Hadoop) cluster, with 1 dedicated master and 9 workers. Each Spark worker utilizes 1 core and has 4GB memory available to it.

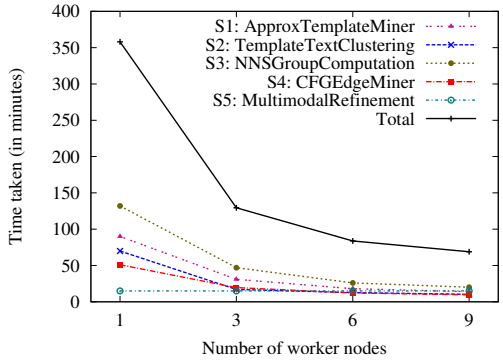


Figure 13: Scalability on (Spark + Hadoop) Setup

Figure 13 shows how the time taken by each stage reduces as the number of Spark workers increase. We use 25 GB subset of *distMW* dataset described in Section 4.1 comprising approx 55 million log lines for this experiment. We get more than 5x speedup when going from single to 9 workers. The total throughput of the entire pipeline is 522 GB/day using our 10 low-end VM setup. Further, we also experimented with the entire 100 GB dataset comprising more than 200 million log lines and got similar throughput (here 508 GB/day).

4.8 Case Study: OpenStack

To evaluate the overall effectiveness of our end-to-end system, we did a case-study with OpenStack [21] VM provisioning system. We drove the input workload by manually creating, suspending, resuming, and terminating multiple VMs randomly via Openstack's Web portal. These meta-level activities were randomly initiated over a period of one week with a total of approximately 50 such activities.

We collected individual logs from all the different microservices of Openstack spanning across nova, glance, keystone, cinder, swift, neutron, and horizon services. We did a timestamp-based merging of these individual logs in order to be able to detect flows across microservices. We used this one-week of logs as training data to mine the healthy-state cfg. We got a cfg with a total of 264 nodes (i.e. templates) and 616 edges.

From this mined cfg, we were specifically interested in the subgraph that corresponded to the popular *create-VM* event since this event requires coordination between most of the microservices. In order to mine the templates that were associated with the create-VM event, we manually created a time-series of this event using the recorded times when we issued this event, and then ran LSH's Minhash [5] with a very high time-series similarity threshold to detect other templates that were tightly correlated with the create-VM event time-series.

A summarized view (depicting module name and important keywords of the main templates only) of the subset of the CFG comprising 44 templates corresponding to create-VM event is shown in Figure 14. The templates belong to different modules of different microservices that participate in the create-VM event, and the cfg structure depicts interactions between them. The actual template pattern for node 8 is shown as example, and it reflects the starting of the nova compute manager instance.

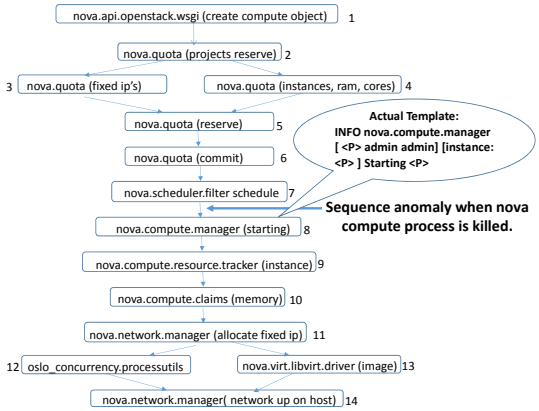


Figure 14: Summarized view of Openstack CFG subgraph related to create-VM event

To verify that our anomaly detection scheme does not throw false alarms when run on another healthy run of the system that is different from the training data, we fired different types of VM events beyond the training period. On running our anomaly detection scheme on these logs, we observed that our system raised only one false-positive sequence anomaly.

We additionally wanted to verify if we can detect anomalies when we manually inject different types of faults in the Openstack cluster. To verify this, we manually disrupted the nova-compute microservice by killing the nova-compute process. Subsequently, we fired a create-VM request. Not surprisingly, the Openstack portal denoted the failure of the create-VM event but was unable to tell us the cause for it. To get insights on the potential cause, we ran our anomaly detector, which reported a sequence anomaly at node-7, since the entire subgraph in Figure 14 from node-8 to node-14 was found missing.

5. RELATED WORK

We organize the related work into the following two main categories –

5.1 Mining of Templates

Contrary to our approach of mining templates from logs using a multi-modal signal of text and temporal-vicinity, all prior approaches use only the textual content of log lines in order to mine templates.

The first subcategory of such works [25, 27] mine templates by leveraging frequent item-set mining approaches wherein the premise is that by mining sets of co-occuring words across a large fraction of logrecords, one can filter out the invariant words in the print statements from the time-varying parameters. Frequent-itemset mining approaches for template mining however suffer from either bad recall or bad precision depending on the support threshold.

The next subcategory of such works [9, 23, 26, 28] employ some mechanism of generating approximate templates using frequency histogram of keywords mined in the first phase, and then further cluster similar approximate templates using some form of text-based similarity metric.

Last but the least, the work of Makanju et al. [19] employs an iterative partitioning technique that recursively partitions the set of log lines, first on number of words in a line, followed by the words in different positions, followed by another partitioning based on association of words in the log lines. But this approach works on impractical assumption that parameters comprise single words only.

5.2 Mining of CFG Edges

Contrary to our two-phase CFG edge mining approach, we describe prior approaches of mining template sequences on interleaved logs and their limitations.

Firstly, classical process mining [6, 11, 18] approaches are not directly applicable to our problem setting since they operate on the prerequisite of clearly demarcated transactions and require an input dataset comprising a set of such transactions.

A second class of approaches [3, 18] exploit temporal dependencies between events to mine control flows under high amount of interleaved traces. Assuming that enough data is available for mining statistically significant dependencies, these approaches use some form of association rule mining [29] to mine correlated events and can then do transitive edge removal. However transitive edge removal will result in removal of genuine detour paths in cfg.

A third class of approaches [17, 30] mine template sequences via stitching together log message templates that share the same parameter. But from our experimentation with real-world log datasets, we observed that parameters unless explicitly embedded (as in [13]), inherently do not flow deep enough. Also, they typically metamorphose, resulting in detection of very short sequences.

Finally, worth mentioning here is that instead of mining sequences, one may mine only the frequency, periodicity, and mutual distribution of different message templates, as proposed in Melody [15]. These models, due to the lack of the underlying cfg structure, are unable to pinpoint the root-cause of the change in the expected distribution of different event types, and end up throwing alerts for every node in the affected subtree of the cfg.

6. CONCLUSION

We have presented an approach of detecting anomalous run-time behaviour in distributed systems from execution logs. We mine a control-flow-graph (cfg) that captures healthy execution flows that span within and across different components of a distributed system, and raise anomaly alerts on observing deviations from this learnt cfg model. We outlined why mining such a cfg from interleaved logs of a distributed application without making any application specific assumptions, is a hard problem. We proposed novel and scalable algorithms to accurately mine the cfg, and based on an extensive evaluation against alternative design choices, depict the superiority of our proposed algorithms.

7. ACKNOWLEDGMENT

We thank our colleagues – Sriram Raghavan, Anindya Neogi, Narendran Sachindran, Pooja Aggarwal and Rahul Chandrakar for valuable discussions at different phases of this project. We thank authors of prior works we compared against, for making their binaries available for experimentation. We also thank the anonymous reviewers for their valuable feedback.

8. REFERENCES

[1] B. Abrahao, F. ChieriChetti, R. Kleinberg, and A. Panconesi. Trace complexity of network inference. In *KDD*, 2013.
[2] Appdynamics. www.appdynamics.com.
[3] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining temporal invariants from partially ordered logs. In *SLAML*, 2011.
[4] D. Bosnacki, W. Lightenberg, M. Odenbrett, A. Wijs, and P. Hilbers. Parallel algorithms for transitive reduction of weighted graphs. In *Math Maced*, 2010.
[5] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *TKDE*, 2001.
[6] W. V. der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. In *TKDE*, 2004.
[7] Elasticsearch: Search and analyze data in real time. https://www.elastic.co/products/elasticsearch.
[8] D. R. Ferreira and D. Gillblad. Discovering process models from unlabelled event logs. In *BPM*, 2009.
[9] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, 2009.
[10] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. In *KDD*, 2010.
[11] C. W. Gunther and W. M. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In *BPM*, 2007.
[12] P. Indyk and R. Motwani. Approximate nearest neighbor - towards removing the curse of dimensionality. In *STOC*, 1998.
[13] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. D. Silva. On fault resilience of openstack. In *SOCC*, 2013.
[14] Kafka: A high-throughput distributed messaging system. http://kafka.apache.org.
[15] T. Li, F. Liang, S. Ma, and W. Peng. An integrated framework on mining log files for computing system management. In *KDD*, 2005.
[16] Logstash: Collect, enrich and transform data. https://www.elastic.co/products/logstash.
[17] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. In *SIGOPS Operation Systems Review*, 2010.
[18] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining program workflow from interleaved traces. In *KDD*, 2010.
[19] A. Makanju, A. Z. Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *KDD*, 2009.
[20] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. In *DMKD*, 1997.
[21] Openstack: Open-source software for creating public and private clouds. www.openstack.org.
[22] J. Pei, J. Han, B. Mortazavi, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, 2001.
[23] T. Reidimeister, M. Jiang, and P. A. Ward. Mining unstructured log files for recurrent fault diagnosis. In *IM*, 2011.
[24] Spark: Lightning-fast cluster computing. http://spark.apache.org.
[25] J. Stearley. Towards informatic analysis of syslogs. In *Cluster*, 2004.
[26] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IPOM*, 2003.
[27] R. Vaarandi. A breadth first algorithm for mining frequent patterns from event logs. In *Intell. Comm.*, 2004.
[28] R. Vaarandi and M. Pihelgas. Logcluster - a data clustering and pattern mining algorithm for event logs. In *CNSM*, 2015.
[29] T. Wu, Y. Chen, and J. Han. Association mining in large datasets: A re-examination of its measures. In *PKDD*, 2009.
[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *ICDM*, 2009.
[31] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large scale system problems by mining console logs. In *ICML*, 2010.