Multiresolution Abnormal Trace Detection Using Varied-Length n-Grams and Automata

Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira

***Abstract*—Detection and diagnosis of faults in a large-scale distributed system is a formidable task. Interest in monitoring and using traces of user requests for fault detection has been on the rise recently. In this paper we propose novel fault detection methods based on abnormal trace detection. One essential problem is how to represent the large amount of training trace data compactly as an oracle. Our key contribution is the novel use of varied-length n-grams and automata to characterize normal traces. A new trace is compared against the learned automata to determine whether it is abnormal. We develop algorithms to automatically extract n-grams and construct multiresolution automata from training data. Further, both deterministic and multihypothesis algorithms are proposed for detection. We inspect the trace constraints of real application software and verify the existence of long n-grams. Our approach is tested in a real system with injected faults and achieves good results in experiments.**

***Index Terms*—Abnormal trace, algorithm, automata, fault detection, large-scale information system, *n*-gram.**

I. Introduction

THE success of global networking has lead to the wide use of various Internet services. Online searching, shopping, and transactions are increasingly becoming part of our daily life. Although users only see a website of Internet services such as Google.com and eBay.com, the information system behind the scene is a large, dynamic, and distributed system and could consist of thousands of components including servers, software, networking devices, storage equipments, etc. While each of these components is already complex enough by itself, the dynamic interaction between them introduces another magnitude of complexity. Further, Internet services are expected to run $24 \times 7 \times 365$ and maintain over 99.9% uptime. The complexity combined with the uptime requirement is a major system management challenge.

Fault detection and diagnosis in such a system is a formidable task. Most current approaches for fault detection and diagnosis use event correlation [1]. This method collects and correlates events to locate faults based on known dependency knowledge between faults and symptoms. In practice, many runtime faults in an interconnected system are not very well understood. Since the runtime environments are very diverse, a fault may manifest itself in different ways. As a result, it is usually difficult to obtain such fault-symptom dependency knowledge precisely. As an example of a complex scenario, consider a class of problems in Internet services that are specific to individual user requests and not visible in collected events. For instance, the "checkout" button does not work after a customer spent hours selecting products from a website. It is possible that this problem affects only one customer and no one else. Such a problem may force us to trace how this individual request went through various components in the system and use such internal trace information to locate the fault.

Recently there has been much research activity in collecting and using traces for performance debugging and fault detection in distributed systems. Several research groups have developed tools to collect the traces of user requests. The Berkeley/Stanford Recovery-Oriented Computing (ROC) group modified the JBoss middleware to monitor traces in the J2EE platform and developed two methods to use collected traces for fault detection and diagnosis [2]. Aguilera *et al.* [3] developed application-independent passive tracing approaches in both RPC-style systems and message-based systems. Their method requires no modification to applications and middleware and can be widely used for performance debugging purpose. Magpie [4] uses low-overhead instrumentation to record fine-grained events generated by kernel, middleware, and applications. The Magpie request extraction tool then uses an application-specific event schema to correlate these events and precisely capture the control flow of every request. In addition, several commercial software packages, such as HP's OpenView Transaction Analyzer [5], have also been developed recently to monitor and trace transaction flows in distributed J2EE and .NET systems.

While these technologies enable us to monitor and collect traces in various distributed systems, in this paper we focus on how to detect faults based on trace analysis. Internet services receive large number of user requests everyday and along the time these requests act like "probing and testing" the system in a brute-force way. A fault or bug inside a system is likely to affect the traces of some user requests. In this paper, we propose to use varied-length n-grams and automata to characterize and represent the normal traces compactly and then use the learned automata to determine whether a new trace is abnormal (or acceptable). We develop algorithms to automatically extract n-grams and build multiresolution automata from training trace data. Further both deterministic and multihypothesis detection algorithms are proposed to detect abnormal traces. We inspect the trace constraints of real application software and verify the existence of long n-grams. Our approach is applied in a real system with injected faults. The experimental results demonstrate that our approach works very well in fault detection.

Manuscript received April 11, 2005; revised August 9, 2005. An early version of this paper was presented at the IEEE 2nd International Conference on Autonomic Computing (ICAC), Seattle, WA, June 14–16, 2005. This paper was recommended by Associate Editor G. Papadimitriou.

The authors are with NEC Laboratories America, Princeton, NJ 08540 USA (e-mail: gfj@nec-labs.com; haifeng@nec-labs.com; cristian@nec-labs.com; kenji@nec-labs.com).

Digital Object Identifier 10.1109/TSMCC.2006.871569

II. Abnormal Trace Detection

A trace records a sequence of components traversed in a system in response to a user request. The system architecture, functionality, and especially software control flows impose many constraints on the structure of traces. For example, components A-B-C-D always show up together and in that particular order because that is the only path starting from the component A. Such constraints are useful in fault detection to distinguish normal and abnormal traces. A fault inside the system is likely to affect some traces and cause these traces to violate some constraints. By detecting the abnormal traces, we expect to localize the cause of the abnormal traces—the faulty component itself.

If the dependency knowledge between specific faults and their symptoms is known, a fault can be detected and diagnosed directly based on its unique symptom. However, it is not realistic to characterize/model faults precisely in a large, complex, and dynamic system. Many runtime faults/bugs are not anticipated or well understood. In addition, we do not have sufficient data to model the faulty situation accurately because in general runtime faults are very rare. An alternative is to characterize/model *normal traces* rather than the faulty ones because large amount of trace data from normal operation is available. This normal model can be used as an oracle to determine whether a new trace is acceptable or not.

A challenge here is that we do not know how much generalization capacity this model should have. This problem is closely related to the VC dimension [6] in learning theory, but here the trace samples are clearly not independently and identically distributed as required by the theory. We can only try to collect as many traces as possible and characterize/model these "known/seen" normal traces well. For a large system, however, it is difficult to include every normal trace in the training data. Further, both the distribution of "unknown/unseen" normal traces and the distribution of "faulty" traces affected by various faults are unknown. In addition, it is also hard to define the "similarity" metric in a conceptual trace space because it is not clear how traces are affected by various faults. Two traces that look very similar in the structure could be thoroughly different with regard to whether they are normal or faulty. Conversely, two traces that look thoroughly different could both be normal traces. Thus, we have few clues about the direction of the trace space in which the normal model should generalize. Typically, high-generalization capability would lead to high false negative rates (missed detections), while low-generalization capability would lead to high false positive rates (false alarms). This motivates us to develop multiresolution algorithms for fault detection.

As we try to develop the right model or structure to characterize normal traces, it is necessary to first analyze some basic properties of traces. A trace includes a list of component names as well as the sequential order of these components. This component sequence order includes both the *local-order* constraints between adjacent components and the *global-order* constraints between nonconsecutive components. For example, in a trace ABCDEFG, the constraint that components A and B are consecutive is a local-order constraint and constraint that component A and E are three steps apart is a global-order constraint. We have to consider how many of such local- and global-order constraints should be captured by the model.

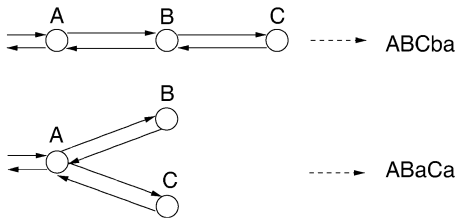


Fig. 1. Traces with call-return structure.

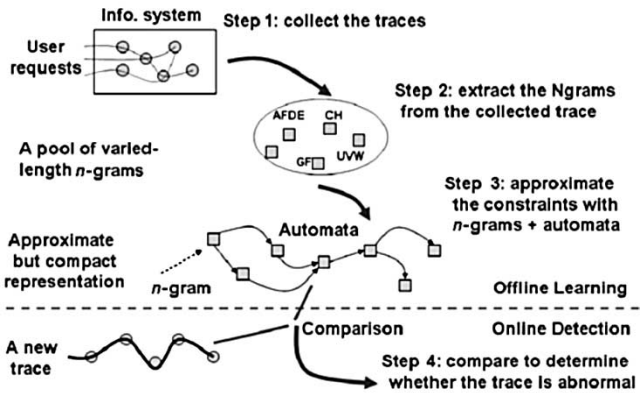


Fig. 2. Steps of abnormal trace detection.

Depending on the monitoring mechanisms, a trace may or may not include call-return structure in its sequence. In this paper, for the trace including call-return structure, calling a component and returning to a component are considered distinct. For example, as shown in Fig. 1, calling a component A is represented by a capital letter "A" and returning to a component A is represented by a lowercase letter "a." Without the call-return structure, both traces are interpreted as the same sequence ABC. Conversely, with the call-return structure, the traces are interpreted as two very different sequences: ABCba and ABaCa. Since our algorithms can be used to analyze the sequences with both structures, in the following sections we do not distinguish these two cases.

n-Gram is a commonly used natural language model [7]. It assumes that only the previous $n-1$ words in a sentence have effect on the probabilities for choosing the next word. In this paper we borrow this term to represent the contiguous component subsequences of a trace. For example, ABCD and EF are a 4-gram and a 2-gram, respectively. Automata are used to connect the n-grams to represent whole traces. Within n-grams, the components are bound together in that order and both the local and global-order constraints remain. Between n-grams in the automata, only the local-order constraints are captured because of the Markov property of automata. By controlling the length of n-grams, we can control how many local and global-order constraints of a trace are represented in the model.

Fig. 2 illustrates the basic steps of our approach. First we collect traces and our algorithm automatically extracts a series of varied-length n-grams from the training trace data based on

frequency checking. These n-grams serve as the basic "genes" to construct any normal traces and are used as states in the construction of automata. The automata are then built automatically by linking n-grams to represent whole traces. These steps may be computationally expensive but can be done offline. A new trace is compared against the learned automata to determine whether it is abnormal (or acceptable). This step is fast and is able to support online detection.

III. n-Gram Extraction

Theoretically, we can count the frequency of each trace and construct probabilistic automata to characterize the distribution of traces. In practice it is difficult to build robust probabilistic models because it is the highly dynamic user behavior that determines the distribution of traces. For example, after some items are on sale, user requests and their traces associated with these items could suddenly become much more frequent, even though there is no failure. Although it seems that gross user behavior in high-traffic Internet services is surprisingly predictable, showing clear diurnal and weekly patterns of behavior, we prefer to build models that do not fundamentally depend on the accuracy with which this behavior is captured. We rely on the fact that in spite of its dynamicity, the user behavior does not change the underlying reachability structure of the automata. An analogy is that the road network of a city does not change but the traffic distribution on each road is always changing. In this paper, we do not use probabilistic models to characterize how frequently a trace in the automata is visited. Instead we only consider whether the automata include the trace or not. The dynamics of user behavior will not affect the validity of our model and, as shown in our experiments, the reachability model is still good enough to detect many failures. Therefore, we only use the unique traces from the training data to extract n-grams, i.e., any trace only shows up once in our training data. Frequency checking and association rules are applied to determine which components are more tightly bound together than others. A threshold α is introduced to filter out those nonfrequent component combinations.

Our n-gram extraction algorithm (Algorithm 3.1) is similar to the classic sequential pattern mining algorithms [8]. Denote the number of unique traces in the training data as N. Algorithm 3.1 goes forward starting from $k = 1$. C_k is the set of n-grams with length $n = k$, i.e., k-grams. c_k^i is the ith k-gram in the set C_k. $f(s)$ is the number of times the sequence s appears in the set of the training trace data. Note that $f(s)$ has to be smaller than both $f(c_k^i)$ and $f(c_k^j)$ because the child sequence s subsumes both c_k^i and c_k^j—the parent sequences. Thus, we have $0 \leq \alpha \leq 1$ and the inequality $f(s) > \alpha \cdot \min(f(c_k^i), f(c_k^j))$ implies that if the longer child sequence s can replace at least α percentage of one of its parent sequences, it is necessary to introduce the new sequence s as a $(k+1)$-gram. There are several pruning techniques to reduce the set of n-grams. For example, if $f(s)$ is equal to $f(c_k^i)$, then we know that as long as the sequence c_k^i shows up in the traces, it has to exist as part of the longer sequence s. Since the longer sequence s has already represented the subsequence c_k^i here, c_k^i can be removed from the set C_k.

Algorithm 3.1 n-Grams extraction algorithm

Input: the set of unique traces
Output: the sets of varied-length n-grams.

$C_1 = \{$the set of single components c_1^i with $f(c_1^i) > 0\}$.
$k = 1$
do
  for each two elements c_k^i, c_k^j from the set C_k,
    if the last $k-1$ component sequence of c_k^i equals the first $k-1$ component sequence of c_k^j,
    then generate a new sequence
      $s = c_k^i$ plus the last component of c_k^j;
      count $f(s)$, the number of times that s appears in the trace data;
      if $f(s) > \alpha \cdot \min(f(c_k^i), f(c_k^j))$,
        then put s into the set C_{k+1}.
  $k = k + 1$.
while C_k is not empty
return all C_j, for $1 \leq j \leq k-1$

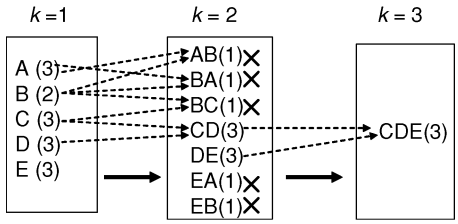


Fig. 3. Example of extraction process.

The threshold α affects the length of extracted n-grams. As we will see in the experiments, the longest n-grams become much shorter as α increases. The traces in a large system are usually very diverse. As $\alpha \to 0$, the longest n-gram becomes the whole trace itself. Conversely, as $\alpha \to 1$, the longest n-grams become the single components, i.e., $n = 1$. Thus, by controlling α, we control the length of the extracted n-grams.

Fig. 3 illustrates an example of the extraction process. Assume that we have three traces: ABCDE, CDEA, and CDEBA, and the threshold is set to $\alpha = 0.6$. Numbers in the parentheses are the number of times that the associated sequence appears in the traces. At $k = 2$, the combined sequences marked with "X" do not pass the threshold and they will not be put into the set C_2. The extraction process ends at $k = 3$ here and the length of the longest n-grams is three. If we apply pruning on these n-gram sets, C, D, and E at $k = 1$ and CD and DE at $k = 2$ will be pruned from the sets because the 3-gram CDE subsumes all of them with the same frequency number.

The implementation of Algorithm 3.1 is similar to the famous Apriori algorithm [8]. Assuming that the length of the longest n-grams is L, this algorithm needs $L + 1$ data scans. The complexity of the algorithm is linearly dependent on the length of traces, but is exponentially dependent on the number of components [9]. However, in practice this algorithm could converge quickly, especially if the threshold α is not small.

IV. Automata Construction

Algorithm 3.1 extracts a series of frequent n-gram sets C_k with varying length k. These n-grams are the basis for

Algorithm 4.1 Automata construction algorithm

Input: the set of unique traces and the sets of n-grams
Output: the automaton E

set $E[m][n] = 0$ for any two n-grams m, n
for each trace T
  set $k = L$ and $l = T$'s length
  do
    for each k-gram c_k^i selected from C_k according to the sorted order (with the most frequent one first),

      search and replace all c_k^i in T with the assigned state number;

      if the length of the replaced part equals l,
        then break from the inner loop.
    $k = k - 1$.
  while the length of the replaced part $\neq l$ and $k \geq 1$.

  from left to right, set $E[m][n] = 1$ if an n-gram n follows another n-gram m contiguously in the trace T

remove the unused n-grams/states from E
return the matrix E

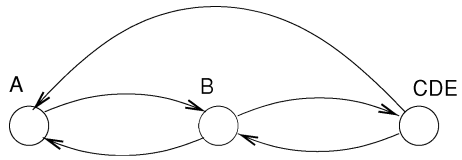


Fig. 4. Example of automaton.

subsequent automata construction. Before constructing the automata, we sort the n-grams with same length (i.e., the n-grams in the same set) according to their frequency. The deterministic Algorithm 4.1 is then used to disassemble whole traces into multiple n-grams for automata construction. The length of the longest n-grams is L and then we have $1 \leq k \leq L$. E is the transition matrix of the constructed automaton. Algorithm 4.1 goes backward starting from $k = L$, following two deterministic rules to cut the whole trace:

1) Rule 1: For different sets of n-grams, choose the set with longer n-grams first.
2) Rule 2: For n-grams in the same set, choose the more frequent one first.

Let us denote the total number of n-grams extracted from Algorithm 3.1 as W, i.e., $W = \sum_{k=1}^{L} |C_k|$, where $|C_k|$ is the size of the set C_k. With N traces, it is straightforward to see that the complexity of Algorithm 4.1 is $O(WN)$. Before explaining the specific cutting process of Algorithm 4.1, it is useful to discuss the generalization ability that automata introduce to the model. Based on Algorithm 4.1, all training traces can be precisely represented by the constructed automata. The bottom line is that the set of single components C_1 is sufficient to construct any trace seen in the training data. However, the generalization ability of automata could regard other traces as normal traces. For example, for three traces ABCDE, CDEA, and CDEBA used in Fig. 3, Algorithm 4.1 will build the automaton shown in Fig. 4. Even if we restrict the sequence to lengths shorter than 6 to remove infinite loops, many traces such as ABABA and CDEAB are still regarded as normal ones because of generalization. States with multiple in and out edges in the automata usually introduce the opportunity of generalization.

Since not every normal trace is seen and collected in the training data, a certain capacity of generalization is desirable to reduce false positives in detection. For example, CDEA is a normal trace and we know that component A calls component B. Based on software's control flow, it is then possible that CDEAB is also a normal trace. The question is how much generalization the automata should have in order to achieve good performance in detection.

As discussed in Section II, in fact we hardly know the answer to this question. Moreover, it is also hard to know the direction of the trace space in which the automata should generalize. Therefore, our principle here is to keep the automata's representation as tight as possible to restrict the randomness in generalization. Later we will use the threshold α to control the generalization capacity of automata. To this end, heuristically, rule 1 tries to link whole traces with smallest number of n-grams and edges. In general, more edges in the automata could lead to more generalization in the trace space. For a trace with fixed length, rule 1 leads to a greedy optimization in cutting the trace. For the n-grams with the same length, rule 2 implies that the more frequent one in the past (unique traces) is more likely to appear in this individual trace too.

Given a set of traces and a set of n-grams, one alternative is to find all possible parses of the traces into the given n-grams to construct automata. However, essentially this approach will construct an automaton using the set of uni-gram C_1 as states, which subsumes the automata constructed with long n-grams with regard to possible paths. As we will see in Section IX, this automaton (roughly equal to the automaton constructed with $\alpha = 1.0$) does not maintain the global-order constraints of long n-grams and has very low detection accuracy in our experiments. In fact, this automaton is close to software control flow graph and can be constructed directly by following components in traces, even without using Algorithm 3.1.

Note that the pruning technique mentioned in Section III could introduce some problems in constructing the automata. For example, consider a trace ABCD and two 3-grams ABC and BCD. Assume that the 4-gram ABCD is not frequent enough to pass the threshold. While building the automaton to represent this trace, if we cut at ABC first, then the remaining part of the trace is D. Now if $f(D) = f(CD)$, D was pruned earlier and does not exist in the set C_1 anymore. In this specific case, following rules 1 and 2, the existing set of n-grams is not sufficient to represent the training traces completely. Therefore, we do not prune the uni-gram set C_1 to guarantee that all training traces can always be precisely represented by the automata. Note that since Algorithm 4.1 goes backward starting from $k = L$, the uni-grams in the set are the last candidates considered to link whole traces and will not change the automaton structure much anyway.

V. Generalization of Automata

Not every n-gram is used to construct the automata. Especially when the threshold α is small, only a small percentage of

Algorithm 5.1 Automata augmentation algorithm

Input: the transition matrix E
Output: the augmented transition matrix $\overline{E}$

set $\overline{E}[k][l] = 0$, for $0 \leq k, l \leq M$
for each n-gram i in E
  $f(i) = e(i-1) + 1$ // the first component
  $e(i) = f(i) + l_i - 1$ // the last component
  for each component j, where $0 \leq j \leq L_i$ in this n-gram i
    assign a new state number $f(i) + j$
    if $j \geq 1$ **then** set $\overline{E}[f(i) + j - 1][f(i) + j] = 1$

for each n-gram i in E
  for each n-gram j in E
    if $E[i][j] == 1$ **then** $\overline{E}[e(i)][f(j)] = 1$
return the matrix $\overline{E}$

n-grams (long n-grams) is used in the automata. The unused n-grams are removed from the matrix E. The total number of extracted n-grams and their distribution could tell us how diverse the traces are.

For each fixed threshold α, Algorithm 3.1 generates a series of varied-length n-grams. For different thresholds, it is clear that the automata built from these n-grams will be different. As $\alpha \to 0$, the longest n-gram becomes the whole trace itself and the automaton will include N states with no edges. These states are the original N traces in the training data. This automaton has zero generalization ability. If we use this automaton to distinguish normal and abnormal traces, the detection is an exact matching process. Conversely, as $\alpha \to 1$, the longest n-gram becomes the single component and the automaton is close to the control graph with single components as states. Therefore, this automaton introduces maximal generalization capacity because the single components cannot be further cut into smaller units. Given the same training data, as the threshold α increases, generally the length of n-grams becomes shorter and the generalization capacity of automata increases. As discussed in Section II, n-grams keep both global- and local-order constraints within their structures. For each state position in the automata that could introduce generalization, longer n-grams have more global-order constraints and require a longer subsequence match before the generalization point. For example, in Fig. 4, the tri-gram CDE requires any trace traversing this state to follow the deterministic subsequence C-D-E before next state transition.

Given N traces and a threshold α, we extract n-grams and construct an automaton based on Algorithms 3.1 and 4.1, respectively. An interesting question is how many unique traces this automaton can generate. This number is a good metric to measure the generalization capacity of automata. Unfortunately, in case of automata with loops, the number or traces is infinite. An alternative is to count the number of unique traces with fixed length. Because we use varied-length n-grams in the automata, each state transition does not add the same length into the sequence. To this end, we have to introduce hidden states and augment the transition matrix E. Denote the length of each n-gram used in the automata as l_i, where $1 \leq l_i \leq L$. Denote the augmented transition matrix as $\bar{E}$ and the total length of n-grams used in E as $M = \sum_i l_i$. Algorithm 5.1 is the augmentation algorithm.

For each n-gram/state with length l_i, Algorithm 5.1 introduces l_i states to the new automaton. $f(i)$ is the state number for the first component of the n-gram i and $e(i)$ is the state number for the last component of the n-gram i. Any two contiguous components within this n-gram contribute a related edge to the new automaton. If n-gram i has an edge to n-gram j in the original matrix E, the last component of the n-gram i has an edge to the first component of the n-gram j in the augmented automaton. Each state in the augmented automaton represents only one component. Clearly, the size of $\bar{E}$ is much larger than that of E.

Now given a fixed trace length, we can compute the number of traces that the automata can generate based on the dynamic programming algorithm [10]. Let $I_t = (I_t^0, I_t^1, \ldots, I_t^M)^T$, where I_t^i is the number of traces whose length is t and the last state is i. Here T is matrix transposition. Let $I_0^i = 1$ if a trace could start from the state i and $I_t^i = 0$ otherwise. Based on dynamic programming, we can have the following recursive equations:

$$I_t^i = \sum_j \bar{E}[j][i] I_{t-1}^j \tag{1}$$

$$I_t = (\bar{E}^T)^t I_0. \tag{2}$$

Thus, the total number of the traces with length t is $\|I_t\|_1 = \sum_{i=1}^{M} I_t^i$.

Unfortunately, this number includes many duplicate traces that can be generated by different state sequences in the automata; it is not the total number of unique traces. As the length t of traces increases, this number could grow exponentially if there are loops in the automata. However, in some cases, a trace must start from certain components and end at certain components. It is clear that the augmented automaton is a nondeterministic finite automaton. Given a fixed length t, in general it is NP-hard to exactly count the number of unique traces that nondeterministic finite automata could generate [11]. Although Gore *et al*. [12] proposed some algorithms to compute this number approximately, it is not clear whether there exist unbiased approximation algorithms with small standard deviations.

VI. Detection Algorithms

Given the automaton constructed from the training trace data, the abnormal trace detection process is to determine whether a new trace is acceptable by this automaton. Both deterministic and multihypothesis detection algorithms are developed here, but they have different conditions on "accepting" a trace as a normal one. Denote the set of n-grams included in the automata as C_a and the total number of n-grams in C_a as N_a; let $c_a^i \in C_a$, for $(0 \leq i \leq N_a)$ be an individual n-gram in the set C_a. In the deterministic algorithm, the same rule 1 and rule 2 in Section IV are used to cut the new trace. The only difference is that the n-grams are chosen from C_a rather than from $C_k (1 \leq k \leq L)$ in Algorithm 4.1. Algorithm 6.1 is the deterministic algorithm. Basically the new trace must satisfy the following two conditions to be classified as a normal trace:

Algorithm 6.1 Deterministic detection algorithm

Input: the automaton and the new trace T
Output: true (normal) or false (abnormal)

set R=true, $l = T$'s length, and $m = 0$
for each n-gram $c_a^i \in C_a$ selected according to rules 1 and 2,
  search and replace all c_a^i in T with the state number;
  $m = m + 1$;
  if the length of the replaced part equal to l,
    then break the loop;
  else if $m == N_a$
    then R=false.
if $R ==$ true, **then**
  from left to right, compare each state transition of T against the automaton;
  if any transition not found in the automaton,
    then $R =$ false.
return R

Algorithm 6.2 Multihypothesis detection algorithm

Input: the matrix $\overline{E}$, O, and a new trace T
Output: true (normal) or false (abnormal)

at step $t = 0$ with the first component $T[0]$,
set $I_0^i = O[i][T[0]]$ for $1 \le i \le M$.
for each step $t = k$, with the new component $T(k)$,
  $I_k^i = \left[\sum_{j=1}^{M} \overline{E}[j][i] \cdot I_{k-1}^j\right] \cdot O[i][T[k]]$, $1 \le i \le M$;
  $k = k + 1$;
  if $k == l$, **then break** the loop.
$Sum = \sum_{i=1}^{M} I_{l-1}^i$.
if $Sum > 0$, **then** $R =$ true **else** $R =$ false.
return R

a) Condition 1: As per rules 1 and 2, the new trace could be completely cut into the n-grams that belong to C_a.
b) Condition 2: The transitions of these n-grams follow a path in the automaton.

Algorithm 6.1 basically tries to check whether a new trace can be interpreted as a specific state sequence in the automaton. The question is why we have to cut a new trace in this specific way. Underling the structure of the automaton, essentially the abnormal trace detection is to run comparison of traces (i.e., compares the new trace with the pool of training traces). The deterministic cutting rules here are just like a hash function to covert traces to state sequences. Note that the same deterministic rules are used in cutting the training traces for automaton construction as well as the new traces for detection. As a result, all training traces will be detected as normal traces and the automaton also allows some strict generalization. An example of condition violations is shown in Fig. 5.

The multihypothesis detection algorithm is developed to relax condition 1. The algorithm searches all state sequences and keeps multiple hypotheses about how to cut the new trace. A new trace is acceptable as long as there is one state sequence that could generate this trace. Given a new trace and the automaton, the challenge is how to efficiently determine whether there exist such state sequences in the automaton.

In Section V, Algorithm 5.1 builds an augmented automaton $\bar{E}$ where each state represents only one component. Conversely,

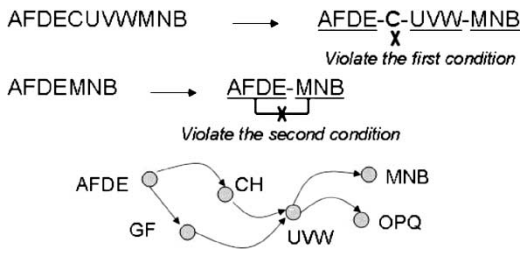


Fig. 5. Example of condition violations.

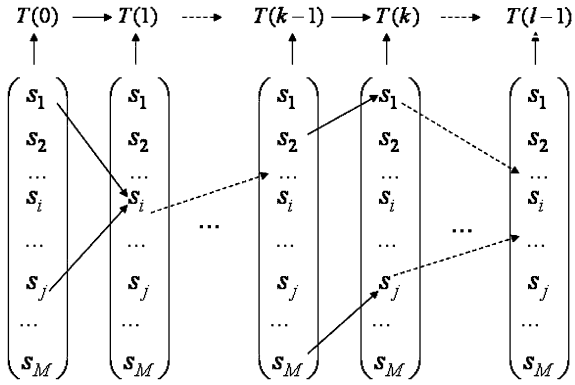


Fig. 6. Viterbi-like algorithm.

one component could be associated with multiple states. Denote the number of states in $\bar{E}$ as M and the number of components as P. When those hidden states are introduced in Algorithm 5.1, we use an $M \times P$ dimension matrix O to record the mapping relationship between the states and their associated components. Let $O[i][j] = 1$ if the state i represents the component j, otherwise let $O[i][j] = 0$. Therefore, we formulate our problem as a specific Hidden Markov Model (HMM) [13], where $\bar{E}_{M\times M}$ is the transition matrix and $O_{M\times P}$ is the emission matrix. The elements of the matrix $\bar{E}$ and O are either 1 or 0, which is different from the probability specification in classic HMMs. Denote the length of a new trace T as l. Following the sequential order (from left to right), denote each individual component that the trace T went through as $T[k]$, where $0 \le k \le l, 0 \le T[k] \le P$. Algorithm 6.2 is the multihypothesis detection algorithm.

Sum is the total number of the state sequences that match the new trace. As illustrated in [14] and [15], we also have efficient algorithms to compute out the exact state sequences underling the new trace. If there is at least one state sequence $(\mathrm{Sum} > 0)$ that could match the trace, the new trace is accepted as a normal one. Otherwise the new trace is regarded as an abnormal one.

As illustrated in Fig. 6, the key idea behind Algorithm 6.2 is dynamic programming and our algorithm is a Viterbi-like algorithm. The $s_i (1 \le i \le M)$ are the states of the automaton. Following the sequential order of the new trace, at each step $t = k$, at first we check which states are associated with the component $T(k)$ according to the emission matrix O, and then check which states at $t = k - 1$ could transfer to these current states according to the transition matrix $\bar{E}$. Finally, we count the total number of valid state sequences that could end at the current states.

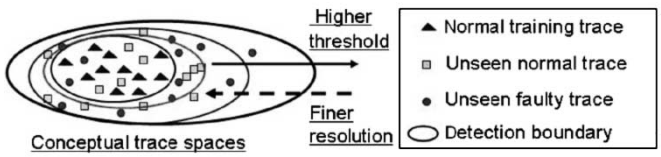


Fig. 7. Threshold and detection resolution.

VII. Multiresolution Detection

As discussed in Section V, given the same training trace data but different thresholds α, Algorithm 3.1 extracts different sets of n-grams and Algorithm 4.1 constructs different automata. These automata all accept the training traces precisely but they have different generalization ability. As $\alpha \to 0$, the detection algorithm becomes an exact-matching algorithm. Any trace unseen in the training data is regarded as an abnormal trace. Therefore, the learned automata are likely to introduce high false positive rates in detection. Conversely, as $\alpha \to 1$, the learned automata have the maximal generalization ability and the detection algorithm accepts many abnormal traces as normal ones. The automata are likely to cause high false negative rates. The threshold α determines a boundary in the trace space that the detection algorithm uses to separate normal traces from abnormal ones. Fig. 7 shows a conceptual space that includes three categories of traces: normal training traces, unseen normal traces, and unseen faulty traces. In general, as α decreases, the detection boundary becomes tighter and the detection algorithm uses a finer resolution to determine whether a trace is abnormal. As mentioned in Section IV, for a fixed threshold, heuristically Algorithm 4.1 tries to draw this boundary as tight as possible to restrict the randomness in generalization.

In practice, it is usually difficult to choose an optimal threshold to balance false positives and false negatives because both the distribution of unseen normal traces and the distribution of unseen faulty traces are unknown. To this end, we propose to construct a series of automata with different thresholds and apply these automata concurrently or sequentially to support multiresolution detection. n-gram extraction and automata construction can be done offline so that the computational overhead of constructing multiple automata should not be a problem. New traces are compared against a set of automata instead of just one to generate alerts. The traces detected as abnormal by the automata with high thresholds are more likely to be true positives and they should be analyzed first to locate faults. Conversely, the traces detected as abnormal only by the automata with low thresholds are more likely to be false positives. As shown in Fig. 7, the traces detected as abnormal by multiple automata are likely to be true positives. Therefore, we can rank these abnormal traces according to our confidence in their abnormality.

Although we do not focus on fault diagnosis in this paper, the result of abnormal trace detection can be used in diagnosis because our approach already identifies individual faulty traces. Following the detected traces, we can locate the faulty component by trace-component correlation and further narrow down the suspicious segments of the component based on the context of the trace. Our experiments demonstrate that many user requests and their traces are not independent. A user request fails because of a faulty component. This request could change some internal states of the system and further affect the routine traces of other requests even though these requests do not go through the faulty component at all. Therefore, except the real faulty traces, there are some "noisy" traces that are detected as abnormal but do not go through the faulty component. Clearly, this increases the difficulty in locating the fault. Our multiresolution detection could help to remove some of the noise in diagnosis. We analyze the abnormal traces detected by the automata with a high threshold first and fix the fault that affects these traces. Then the "noisy" traces affected by this fault will go away and we can analyze the remaining abnormal traces detected by the automata with a lower threshold. Step by step we tighten the boundary and analyze the differential part of boundaries in diagnosis until the problem is fixed.

For a large distributed system, we do not have to choose one threshold for the whole system. Instead different thresholds can be chosen for different segments of traces. High threshold should be chosen for the less important, more reliable segments such as the segments that have been running for long time and stable. Low threshold should be chosen for the more important and less reliable segments. For example, the segments with new deployed software or equipments, the segments that have a lot of problems, and the segments that are critical for the service.

VIII. Trace Diversity: A Case Study

So far we have introduced our approach for abnormal trace detection. There is one important question that remains: Are there long n-grams existing in real software and how diverse are those traces? If, by nature, software's traces are very diverse, we will not have many constraints in trace space and any detection algorithms based on trace analysis cannot be very effective. For example, if any component can call another component in software, then there are no order constraints and the trace space is full of randomness. This question motivated use to have a case study of real application software, Pet Store [16].

Pet Store is a sample application of the J2EE platform developed by Sun Microsystems. It is a blueprint program written by Sun to demonstrate how to use the J2EE platform to develop flexible scalable cross-platform enterprise applications [16]. It has 27 Enterprise JavaBeans (EJBs), some Java Server Pages (JSPs), Java Servlets, etc. We use the same monitoring facility described in [2] to record traces of user requests and the JBoss middleware is modified to support such functionality. Since the logged traces include call-return structure, as mentioned in Section II, calling a component and returning to a component have different representations in the trace sequences.

Given a threshold, Algorithm 3.1 extracts a series of frequent n-grams with varying length. Note that these n-grams are extracted from unique traces and the threshold implies how often they are shared by various traces. Thus, the distribution of these n-grams is a good metric to measure the diversity of traces. For a fixed threshold, in general if a system has higher percentage of long n-grams, the system is more deterministic. We collected most of possible traces from Pet Store by emulating various

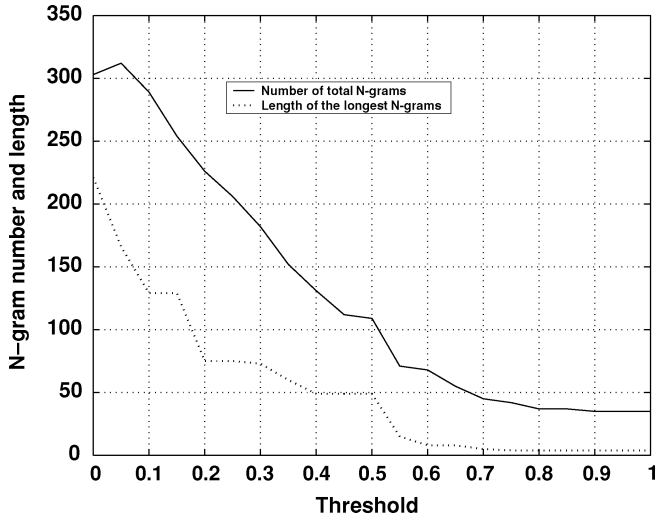


Fig. 8. Number and length of n-grams.

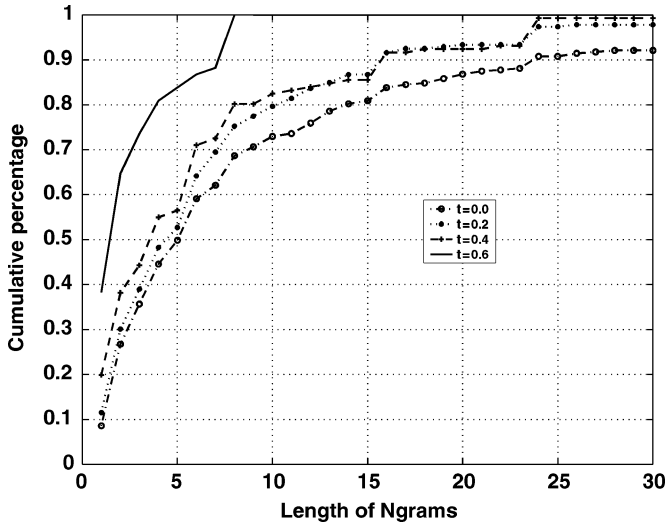


Fig. 9. Cumulative percentage of n-grams.

user requests. Algorithm 3.1 is used to extract n-grams with various thresholds. The pruning technique mentioned in Section III is used to remove those n-grams that can be completely replaced by the longer n-grams and have the same frequency numbers.

Fig. 8 shows how the total number of n-grams and the length of longest n-grams quickly decrease as the threshold increases. As the threshold approaches 1, the total number of n-grams is close to the number of components. Even for a relatively high threshold 0.5, there are n-grams as long as 50, which makes us believe that long n-grams do exist in real software. Because of the pruning, here $\alpha = 0.05$ has more n-grams than $\alpha = 0$. Fig. 9 illustrates the cumulative distribution of varied-length n-grams. As shown in the figure, for various thresholds, the length of most n-grams (over 80%) is less than 15. There is a big percentage of n-grams longer than 5. These long n-grams could impose many constraints in trace space.

Figs. 10 and 11 show the distribution of n-grams at various thresholds. When the threshold is low, 2-grams have the biggest percentage in distribution (around 20%) and there exists a significant percentage of longer n-grams. As the threshold increases, uni-grams become dominant, but there still exists some longer n-grams. For a threshold as high as 0.8, we can see that almost 70% of n-grams are single components. These figures illustrate that even for small application software like Pet Store, there exists a significant percentage of long n-grams (longer than 5). Intuitively we believe that large software is more likely to share some n-grams (component sequences) than the small one because reusable components are more attractive and meaningful in large and complicated software.

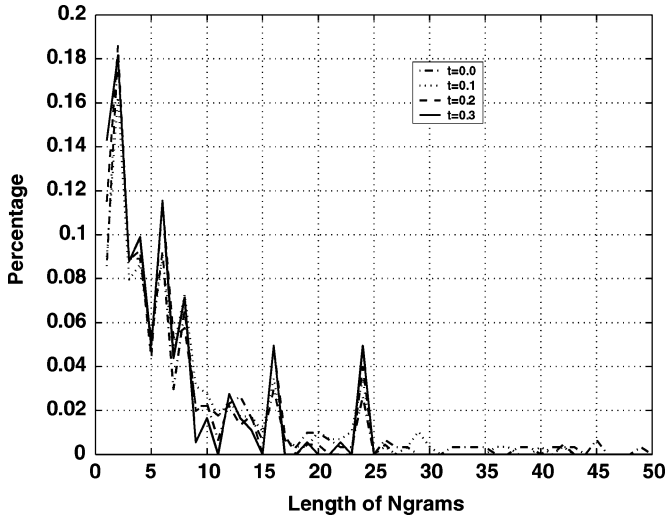


Fig. 10. n-gram distribution for thresholds $t = 0.0$–0.3.

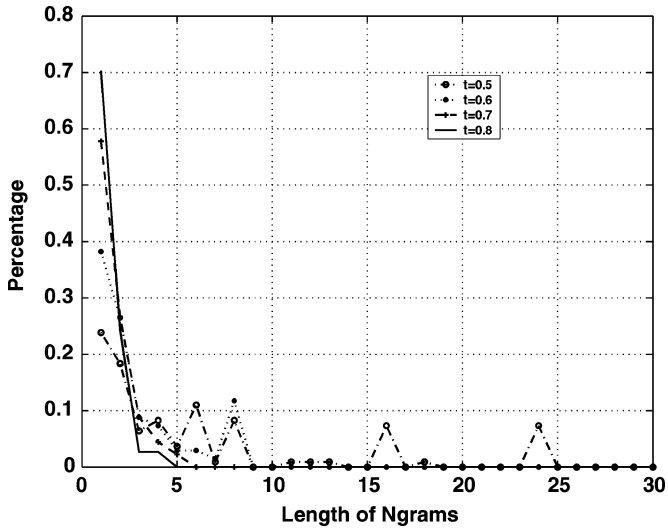


Fig. 11. n-gram distribution for thresholds $t = 0.5$–0.8.

IX. Detection Experiments

Before our detection experiments, Algorithm 4.1 constructs a series of automata based on the extracted n-grams at different thresholds. Fig. 12 illustrates the sizes of these automata. If we compare the number of states with the number of n-grams in

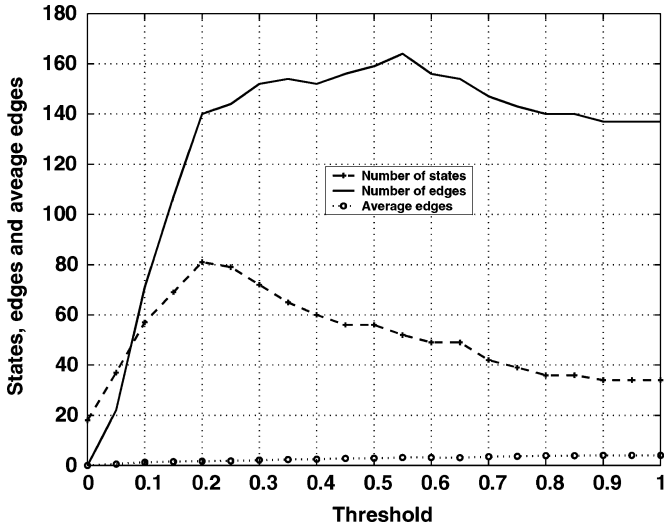


Fig. 12. Size of automata.

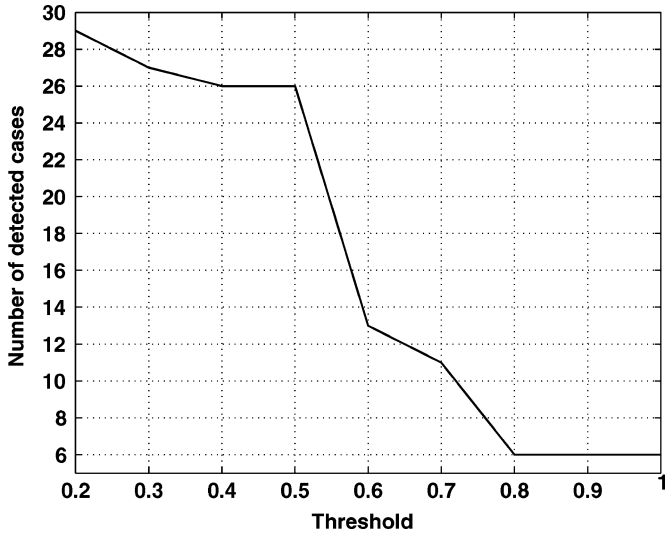


Fig. 13. Detection accuracy of Pet Store.

Fig. 8, it is clear that many extracted n-grams are not used in the automata, especially when the threshold is low. For example, with the threshold 0.4, around 130 n-grams are extracted from the training data but only 60 are finally used in the automata. As the threshold approaches 0, the whole traces become the isolated states in the automata with no edges at all. It is interesting to see that the number of edges increases first and then keeps a relatively constant value. As we expected, the average edge per state (see the dotted line close to the x axis in Fig. 12) increases monotonely as the threshold increases. In fact this explains the growing generalization ability of the automata as the threshold increases.

In our detection experiments, two types of faults are injected into the components of the Pet Store software: "null call" and "expected exception" [17]. After a null call failure is injected into some component C, any invocation of a method in component C results in an immediate return of a null value (i.e., calls to other components are not made). The other failure type, expected exception, is injected into components that contain methods which declare exceptions. After the expected exception failure is injected into a component, any invocation of its methods declaring exceptions will raise the declared exception immediately (if the method declares many exceptions, an arbitrary one is chosen and thrown). Methods in that component, which do not declare exceptions, are unaffected by this injected failure. Other kinds of failures could also be injected (e.g., runtime exceptions, dead-lock, etc.), but typically these errors are easier to catch with many existing monitoring tools. In contrast, the two failures that we selected result in more subtle outcome, do not cause exceptions to be printed on the operator's console, and do not crash the application software. At the same time, these bugs can easily happen in practice due to incomplete, or incorrect, handling of rare conditions. In fact, we also have injected common software faults such as dead-lock and infinite loop in our experiments by modifying the source code. Our approach can detect such kind of faults easily because these faults affect traces significantly. However, our approach is not able to detect some common faults such as memory leaking because this kind of fault does not affect traces at all. Memory leaking is a common software bug where a program repeatedly allocates heap memory to an object, but never releases it. This fault consumes only memory resource, but does not affect the logic of software directly. Therefore it is not possible to detect such faults based on trace analysis. In fact, a program with memory leaking bug could run correctly for a long period of time before the accumulation of leaked memory causes something bad to happen.

Both faults are injected to 15 EJB components of Pet Store resulting in a total of 30 cases. Note that the Pet Store package includes three applications and we chose only one in our experiment, which includes 15 EJB components. We applied Algorithms 6.1 and 6.2 in detection and these two algorithms achieved the same detection results shown in Fig. 13. Algorithm 6.2 has a much relaxed condition on accepting a trace and could reduce false positives in large systems where we may only be able to collect a small part of all traces for training. Since Pet Store is relatively simple software and the size of learned automata is small, the opportunity to have multiple trace hypotheses is low and the two algorithms got the same detection results. As the threshold is decreased, our approach can detect most cases. It is interesting to analyze the detection accuracy in Fig. 13 with the length of the longest n-grams in Fig. 8. From the threshold 0.5 to 0.6, the detection accuracy decreases steeply from $26/30$ to $13/30$ as the length of n-grams decreases steeply from 50 to 8. When the threshold is higher than 0.8, both detection accuracy line and n-gram length line are flat. This unveils an important fact that the constraints of long n-grams are critical in abnormal trace detection.

When the threshold is higher than 0.5, the abnormal traces detected by our algorithms are all true faulty traces (that go through the faulty components), though the detection accuracy is not very high. However, as the threshold is decreased below 0.5, the detection accuracy becomes much higher; in 15 cases, our algorithms detected some other traces as abnormal even though

these traces did not go through the fault-injected components at all. At first we thought these traces are false positives. But after analysis, we found that these traces are affected by the fault even though they did not go through the fault-injected components. The traces of different user requests are not independent at some time. Because of the faulty component, the traces that went through the faulty component failed and changed some internal states of the system. Some other traces are affected by these states even if they do not go through the faulty component.

Therefore, after a fault happens in a system, usually we will detect a set of abnormal traces that include the traces that go through the faulty components as well as the traces that are affected by the fault but do not go through the faulty components. This phenomenon could make the detection process easier but the diagnosis process much harder. For detection, a set of abnormal traces unveil strong "signal" on the existence of faults. However, for diagnosis, those affected traces become "noise" because they do not go through the faults at all and these traces' dependency relationship is unknown. However because the fault is the root cause of these abnormal traces, we can use the timestamp of each trace to locate the true faulty trace. The timestamp is already logged for each trace. Among all traces detected as abnormal by our algorithms, the trace with the earliest timestamp is most likely to be the real faulty trace because it is the failure of this trace that triggered others. After we applied this rule in detection, we have only one false positive in total 30 cases for $\alpha \leq 0.5$. For $\alpha > 0.5$, there is no false positive in our experiments. Since we collected most of traces in the training data, the false positive rate is very low in our experiments. For large software, we believe that the false positive rate could also be low if training traces are collected for sufficiently long time.

Our approach is compared to the Probabilistic Context-Free Grammar (PCFG) method implemented in [17]. The PCFG method uses the tree structure of the request path to synthesize a grammar by associating a nonterminal with each component, and a grammar production with each occurrence of the component in a request tree, where the right-hand side of the production is composed of the immediate children of the node. Productions are assigned probabilities based on the frequency of their occurrence in the training data. Each new trace is assigned an "anomaly" score based on the deviation from the expected probability of each production used. Although it is possible to build PCFGs with more elaborate productions from the same training traces, our comparison is against this specific implementation obtained from the ROC project. Fig. 14 shows the recall and precision curve of our approach as we vary the threshold α. See PCFG's recall and precision curves in Fig. 8 of [17]. In general, our approach achieves very good results. However, since there is no benchmark data, it is hard to say which method is exclusively better. Note that detection accuracy is also dependent on how many traces (among all possible traces) are collected for training. If we only collect a small part of traces for training, our detection algorithms could result in high false positives. In fact, this is a common problem for all machine learning based approaches. However, since traces are collected from operational environments, we can take time to collect sufficiently large number of traces for training.

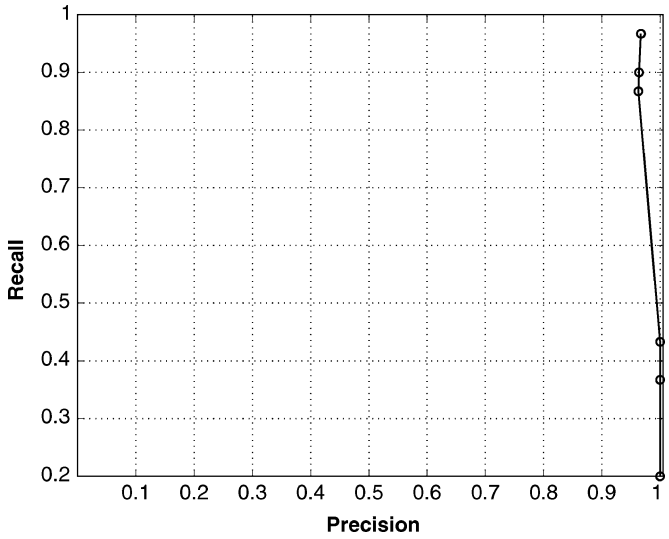


Fig. 14. Precision and recall.

The situation can become much complicated if many requests go through the faulty components and interleave together with the affected traces. With the above rule, we can pick one faulty trace accurately with the earliest timestamp. But there are many following faulty traces that could mix up with the affected traces triggered earlier. Internet services receive many requests and we can use normal traces to cross validate whether an abnormal trace is a really faulty one. For example, if several normal traces cover each component of an abnormal trace, this abnormal trace is likely to be false positive because all of its components work well in other traces. Meantime, we can also artificially generate a "probing" request and force it to go through the suspicious area to verify whether an abnormal trace is a real faulty one.

To cross-validate whether our approach works well in other application software, we also have done detection experiments with RUBiS software [18]. RUBiS is an auction software modeled after eBay.com so that it has much different software architecture compared to the Pet Store software. It is open source software commonly used to evaluate application design patterns and application server performance. RUBiS includes only five EJB components and some number of Java Servlets. Both "null call" and "expected exception" faults are injected to these five components so that we have ten different fault injection cases. In addition, since RUBiS is relatively small software, we manually analyze the traces collected in our experiments and find that total 38 traces go through the fault-injected components in these ten cases. Therefore, we use these faulty traces as the ground truth in our detection experiments.

Fig. 15 shows the detection result of our experiments with RUBiS. The detection accuracy increases as the threshold α decreases. With $\alpha = 0.5$, all 10 faulty cases can be detected, but only 20 among the total 38 faulty traces are detected. With $\alpha = 0.2$, our approach is able to detect all 38 faulty traces. With $\alpha \leq 0.5$, two traces that do not go through the fault-injected components are also detected as abnormal traces. As discussed earlier, later we notice that these traces are affected by the faults

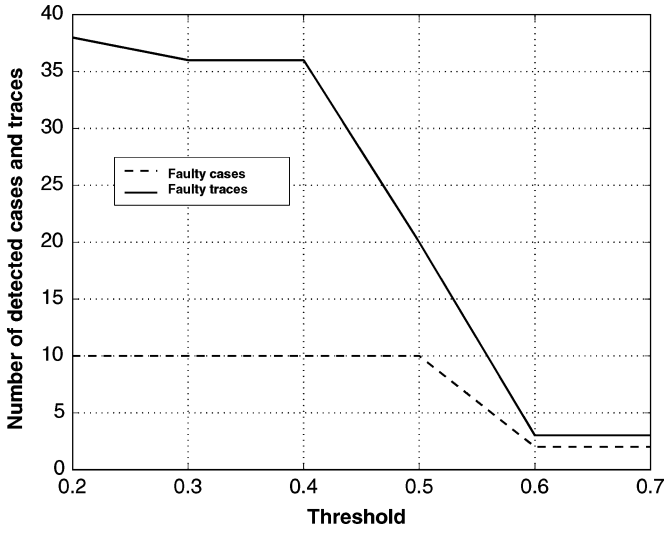


Fig. 15. Detection accuracy of RUBiS.

though they do not go through the faulty components at all. Readers can further draw the precision and recall curve based on these numbers. If we compare Fig. 15 with Fig. 13, it is also interesting to see that the detection accuracy in both cases increases quickly after the threshold α decreases beyond $\alpha = 0.5$.

X. Related Work

There is much work about fault detection and diagnosis in telecommunication network management. Yemini *et al*. [1] proposed a "Codebook" approach for high speed and robust event correlation. Chao *et al*. [19] developed an automated fault diagnosis system using hierarchical reasoning and alarm correlation. Recently, Benveniste *et al*. [20] employed a net unfolding approach originating from the Petri net research for distributed fault diagnosis. As we discussed earlier, all these methods collect and correlate events to locate faults based on known dependency knowledge between faults and symptoms. While this knowledge can be derived from network topology for telecommunication network, it is very difficult to obtain such kind of knowledge in large and complex information systems.

Our work is inspired by Berkeley/Stanford ROC group's success in using traces for fault detection [2], [17]. However, while they focus on developing the whole concept of recovery-oriented computing, in this paper we are interested in developing specific machine learning technology to exploit the traces extensively. Compared to their tree structure and probabilistic model PCFG, we proposed a thoroughly different structure, varied-length n-grams and automata, to characterize normal traces. As discussed in Section III, we believe that it is difficult to build a robust probabilistic model to reflect dynamic user behaviors. Instead, our automata model is static as long as the application software is not modified. All other system changes including user behavior and load balancing will not invalidate our model, which enables us to collect sufficiently large number of traces for training and further reduce false positives. Our approach also allows us to control the generalization ability of the learned model to improve detection accuracy. It also works for traces without call-return structure and could be applied to a wider domain such as security-related detections.

Anomaly intrusion detection is an active area in computer security research. Forrest *et al*. [21] proposed to use fixed-length n-grams to characterize the system call sequence of normal Unix processes. The short sequences of system calls are used as a stable signature in intrusion detection. Michael and Ghosh [22] proposed two state-based algorithms to characterize the system call sequences of programs and used a thoroughly different approach to build finite automata for intrusion detection. They proposed a probabilistic model to calculate the anomaly score. Both of these approaches use fixed-length n-grams and keep a moving window (move one system call each time) to cut the system call sequences. Therefore, two consecutive n-grams have $n-1$ calls overlapped. Based on their experiments, they found that the model with length $n = 6$ usually achieves the best performance. Our approach uses a thoroughly different method to extract varied-length n-grams and construct automata. Because our approach is used in fault detection rather than intrusion detection and the collected traces are not system call sequences, we are not able to compare our approaches with theirs.

Except PCFG and our automata approach, other methods such as Crutchfield's ϵ-machine [23] can also be used to learn the causal and dynamical structure underlying traces. Recently, Shalizi and Crutchfield [24] proposed the Causal-State Splitting Reconstruction (CSSR) algorithm to estimate an ϵ-machine from samples of a process. Theoretically we may employ their algorithm to statistically learn an ϵ-machine to characterize traces. However, the ϵ-machine is also a probabilistic model and their algorithm works only under strong mathematical conditions. Additionally their algorithm has large computational complexity.

XI. Conclusion

In this paper, we proposed a new approach to characterize the normal traces of user requests of Internet services—varied-length n-grams and automata. We developed automated algorithms to extract n-grams and construct automata from training data. New traces are compared against the automata to determine whether they are abnormal. Both deterministic algorithm and multihypothesis algorithm are introduced for abnormal trace detection. We analyzed the generalization ability of automata and introduced a threshold to support multiresolution detection. We also did a case study of real application software to analyze the trace constraints and verified the existence of long n-grams. Further we tested our algorithms in a series of detection experiments with fault injected software. The experimental results demonstrated that our algorithms could work very well in fault detection.

In future work, we need to further verify the results of our approach in larger systems, where we may see only small part of all traces. We also plan to consider the efficiency and scalability issues of our algorithms.

REFERENCES

[1] A. Yemini and S. Kliger, "High speed and robust event correlation," *IEEE Commun. Mag.*, vol. 34, no. 5, pp. 82–90, May 1996.

[2] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *2002 Int. Conf. Dependable Systems and Networks*, Washington, DC, Jun. 2002, pp. 595–604.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. 19th ACM Symp. Operating Systems Principles*, Bolton Landing, NY, Oct. 2003, pp. 74–89.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modeling," in *6th Symp. Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004, pp. 259–272.

[5] [Online]. Available: Http://www.openview.hp.com/products/tran/

[6] V. Vapnik, *The Nature of Statistical Learning Theory*. Berlin, Germany: Springer-Verlag, 1995.

[7] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization," in *3rd Annu. Symp. Document Analysis and Information Retrieval*, Las Vegas, NV, Apr. 1994, pp. 161–175.

[8] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proc. Int. Conf. Data Engineering (ICDE)*, Taipei, Taiwan, R.O.C., Mar. 1995, pp. 3–14.

[9] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. San mateo, CA: Morgan Kaufmann, 2000.

[10] D. Bertsekas, *Dynamic Programming and Optimal Control*. Nashua, NH: Athena Scientific, 2001.

[11] S. Kannan, Z. Sweedyk, and S. Mahaney, "Counting and random generalization of strings in regular languages," in *Proc. SODA*, San Francisco, CA, Jan. 1995, pp. 551–557.

[12] V. Gore, M. Jerrum, S. Kannan, Z. S. amd, and S. Mahaney, "A quasi-polynomial-time algorithm for sampling words from a context-free language," *Inf. Comput.*, vol. 134, no. 1, pp. 59–74, Apr. 1997.

[13] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," in *Proc. IEEE*, no. 2, Feb. 1989, vol. 77, pp. 257–286.

[14] G. Jiang, "Weak process model for robust process detection," in *SPIE Symp. Defense and Security*. Orlando, FL, Apr. 2004, vol. 5403, pp. 102–112.

[15] G. Cybenko *et al.*, "An overview of process query system," in *SPIE Symp. Defense and Security*. Orlando, FL, Apr. 2004, vol. 5403, pp. 183–197.

[16] [Online]. Available: Http://java.sun.com/developer/releases/petstore/

[17] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," Computer Science Dept., Stanford Univ., Stanford, CA, Tech. Rep., 2004.

[18] [Online]. Available: Http://rubis.objectweb.org/

[19] C. Chao, D. Yang, and A. Liu, "An automated fault diagnosis system using hierarchical reasoning and alarm correlation," *J. Network Syst. Manag.*, vol. 9, no. 2, pp. 183–202, Jun. 2001.

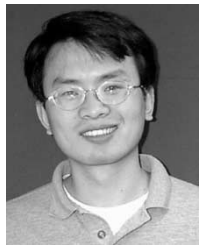
[20] A. Benveniste, E. Fabre, C. Jard, and S. Haar, "Diagnosis of asynchronous discrete event systems, a net unfolding approach," *IEEE Trans. Autom. Control*, vol. 48, no. 5, pp. 714–727, May 2003.

[21] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for Unix processes," in *IEEE Symp. Research in Security and Privacy*, Los Alamitos, CA, 1996, pp. 120–128.

[22] C. C. Michael and A. Ghosh, "Simple, state-based approaches to program-based anomaly detection," *ACM Trans. Inf. Syst. Security*, vol. 5, no. 3, pp. 203–237, Aug. 2002.

[23] J. Crutchfield and K. Young, "Inferring statistical complexity," *Phys. Rev. Lett.*, vol. 63, no. 2, pp. 105–108, Jul. 1989.

[24] C. Shalizi, K. Shalizi, and J. Crutchfield, "Pattern discovery in time series, Part I: Theory, algorithm, analysis and coverage," Santa Fe Insitute, Working Paper 02-10-060, 2002.



Guofei Jiang received the B.S. and Ph.D. degrees from the Beijing Institute of Technology, Beijing, China, in 1993 and 1998, respectively, both in electrical and computer engineering.

From 1998 to 2000, he was a Postdoctoral Fellow in Computer Engineering at Dartmouth College, Hanover, NH. He is currently a Research Staff Member with the Robust and Secure Systems Group, NEC Laboratories America, Princeton, NJ. He has published nearly 50 technical papers. His current research focus is on distributed system, dependable and secure computing, and system and information theory.

Dr. Jiang is an Associate Editor of IEEE SECURITY AND PRIVACY and has severed in the program committees of many prestigious conferences.

Haifeng Chen received the B.Eng. and M.Eng. degrees from Southeast University, Nanjing, China, in 1994 and 1997, respectively, both in automation, and the Ph.D. degree in computer engineering from Rutgers University, New Brunswick, NJ, in 2004.

He has worked as a Researcher at the Chinese National Research Institute of Power Automation. He is currently a Research Staff Member at NEC Laboratories America, Princeton, NJ. His research interests include data mining, autonomic computing, pattern recognition, and robust statistics.

Cristian Ungureanu received the Eng. Diploma from the Bucharest Polytechnic Institute, Bucharest, Romania, in 1987 and the Ph.D. degree from the Courant Institute of Mathematical Sciences, New York University, New York, in 1998, both in computer science.

He is currently a Senior Research Staff Member at NEC Laboratories America, Princeton, NJ. His research interests include robust and secure distributed storage and automatic failure detection in distributed systems.

Kenji Yoshihira received the B.E. degree from the University of Tokyo, Tokyo, Japan, in electrical engineering and the M.S. degree in computer science from New York University, New York, in 1996 and 2004, respectively.

From 1996 to 2000, he designed processor chips for enterprise computers at Hitachi Ltd. In 2002, he employed himself in CTO at Investoria Inc., Japan, to develop an Internet service system for financial information distribution. He is currently a Research Staff Member with the Robust and Secure Systems Group, NEC Laboratories America, Princeton, NJ. His current research focus is on distributed systems and autonomic computing.