A New Semantic Kernel Function for Online Anomaly Detection of Software

Saeed Parsa and Somaye Arabi Naree

In this letter, a new online anomaly detection approach for software systems is proposed. The novelty of the proposed approach is to apply a new semantic kernel function for a support vector machine (SVM) classifier to detect fault-suspicious execution paths at runtime in a reasonable amount of time. The kernel uses a new sequence matching algorithm to measure similarities among program execution paths in a customized feature space whose dimensions represent the largest common subpaths among the execution paths. To increase the precision of the SVM classifier, each common subpath is given weights according to its ability to discern executions as correct or anomalous. Experiment results show that compared with the known kernels, the proposed SVM kernel will improve the time overhead of online anomaly detection by up to 170%, while improving the precision of anomaly alerts by up to 140%.

Keywords: Software debugging, online anomaly detection, semantic kernel.

I. Introduction

Online anomaly detection techniques are of great importance to prompt an alert about anomalous behaviors before program failure [1], [2]. To assure a pre-mortem fault alert, the anomaly detection mechanism should monitor the program behavior at bug suspicious points. Precision and time efficiency are two major prerequisites for an online anomaly detection mechanism [3]. A support vector machine (SVM) classifier could be best applied to detect anomalous behaviors at each program predicate in a relatively small amount of time. The SVM classifier for each predicate classifies all the execution paths ending to the predicate as either failing or passing. Representing each execution path as a state vector in the program execution space, where each dimension represents a predicate, the termination state of the execution could be predicted as the inner product of the state vector and the support vectors provided by the SVM [4].

To increase the precision of online anomaly detection, the SVM classifier applies a feature expansion method [5] to transform the program execution space into a new feature space in which all the state vectors tend to be separable. Feature expansion is achieved by increasing the dimensionality of the program execution space. The key to minimizing the time overhead imposed by the SVM classifier is to select the dimensions of the new feature space in such a way that the dimensions are perpendicular to each other.

To address the preciseness and time efficiency issues, this letter suggests mapping the program execution space into a feature space in which the dimensions represent common non-overlapping subpaths among different program execution paths. Each dimension is assigned a weight according to its capability of discriminating anomalous from correct executions. In this feature space, the similarity between two execution paths is measured as the number and weight of common subpaths. To this end, we have supplied the SVM kernel with a new sequence matching algorithm. The algorithm finds the longest common subpaths between any two execution paths in $O(n^2)$, where n indicates the length of the longer path.

II. Online Anomaly Detection Approach

There are two major difficulties with current SVM kernels that oppose their preciseness and time efficiency for online anomaly detection. Firstly, to have a precise similarity measure between execution paths, the sequence of executed predicates

Manuscript received June 30, 2011; revised Nov. 15, 2011; accepted Nov. 25, 2011.

Saeed Parsa (phone: +98 912 100118, parsa@iust.ac.ir) and Somaye Arabi Naree (corresponding author, a_arabi@iust.ac.ir) are with the School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

http://dx.doi.org/10.4218/etrij.12.0211.0293

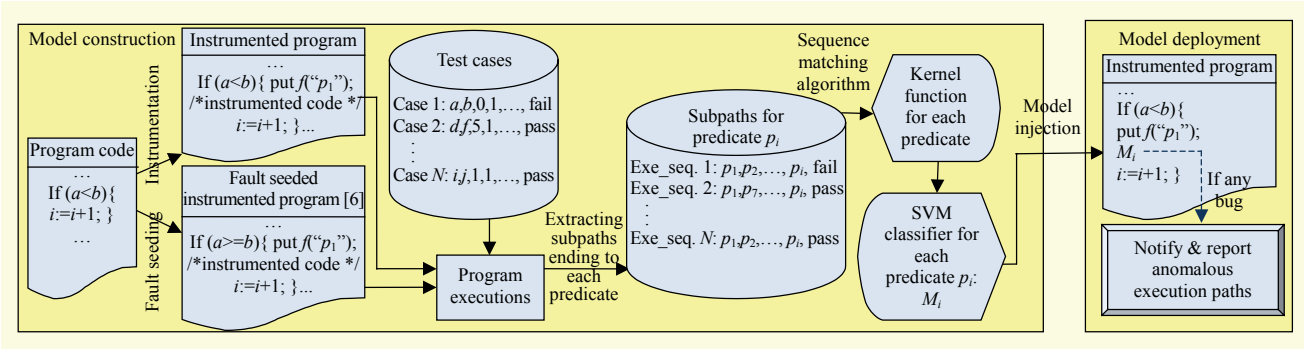

Fig. 1. Flow diagram of proposed online anomaly detection approach.

should be considered when building new dimensions for the feature space. This is due to the fact that the execution order of predicates is one of the main parameters in discriminating failing from passing executions. Secondly, to compute the similarity regarding time efficiency, the dimensions of the feature space should be linearly independent. To resolve the first difficulty, a new sequence matching algorithm is proposed to find common subpaths in failing and passing executions. The resultant subpaths could be best applied as new dimensions for building the feature space. To support linear independency, the non-overlapping common subpaths could be selected. The longest common subpaths selected among non-overlapping subpaths are necessary and sufficient features to measure the similarity between any two execution paths, represented as a sequence of predicates.

Assume $P=\{p_1, p_2, \ldots, p_N\}$ is a set of predicates belonging to program R and $E_i = \{e_{i1}, e_{i2}, \ldots, e_{im}\}, 1 \le i \le N$, indicates the set of all execution paths ending to predicate $p_i \in P$ in m different executions of R.

Definition 1. Execution path. An execution path $e_{ij} \in (E_i = E_{i\,\text{pass}} \cup E_{i\,\text{fail}})$, $1 \le j \le m$, is a sequence of the predicates executed up to the predicate $p_i \in P$ at the j-th execution of the program R.

The main idea of our approach is to monitor the execution of a program R to detect whether an execution path e_{ij} leads to failure.

Definition 2. Faulty path. An execution path e_{ij} is classified as a faulty path if the order and values of the predicates in e_{ij} are more similar to unsuccessful paths of the program than successful ones:

$$\begin{aligned} &Sim(e_i, E_{i\text{pass}}) < \text{Passing-threhold}, \\ &Sim(e_i, E_{i\text{fail}}) < \text{Failing-threhold}, \end{aligned} \tag{1}$$

where Sim is a function to measure the similarity between execution paths. Actually, the Sim function is the customized kernel function for the SVM classifier.

As shown in Fig. 1, our proposed approach includes two main phases of model construction and model deployment.

Suppose there is a set $P=\{p_1, p_2, \ldots, p_N\}$ of N predicates in a program R. For each p_i, $i \in [1, \ldots, N]$, there is a set of state vectors $V_i=\{v_{i1}, v_{i2}, \ldots, v_{im}\}$ in which each state vector v_{ij} in V_i corresponds to an execution path e_{ij} in the program execution space. Applying the SVM classifier, failing and passing state vectors in V_i can be classified by a hyper-plane H_i in the feature space. The problem of finding the most suitable H_i can be expressed as the following optimization problem [5]:

$$\text{maximize}\left(\sum_{j=1}^{m}\alpha_{ij} - 0.5\sum_{j=1}^{m}\sum_{k=1}^{m}\alpha_{ij}\alpha_{ik}K(v_{ij}, v_{ik})C_{e_{ij}}C_{e_{ik}}\right), \tag{2}$$
$$\text{on the region}: \alpha > 0$$

where α_{ij} and α_{ik} indicate the Lagrange multipliers related to the j-th and k-th state vectors of the predicate p_i, respectively, and $C_{e_{ij}}$ and $C_{e_{ik}}$ indicate class labels of these two state vectors, respectively. K is our customized kernel function to measure the similarity between each pair of state vectors. Before the measurement could be performed, the program execution space should be mapped into a feature space in which the space dimensions are linearly independent. In such a feature space, the similarity could be simply measured in terms of the inner product of the mapped vectors. The mapping function is defined as

$$\phi : e_{ij} \to \phi(e_{ij}) = (\text{all non-overlapping subpaths between any pair of execution paths}). \tag{3}$$

The customized kernel K is defined using the similarity measure algorithm presented in Fig. 2. The algorithm measures the similarity between two given sequences s and t in terms of the weight and the number of common non-overlapping subsequences in s and t. The time complexity of step 1 of the algorithm is $O(|s|\,|t|)$. The time complexity of the steps 2 and 3 are $O(|s|\,|t|)$ and $O((|s|+|t|)^2)$, respectively. Therefore, assuming $|s|<|t|$, the overall time complexity is $\max(O(|s|\,|t|), O(|t|^2))$.

The similarity between two execution paths, represented by the sequences s and t, could be measured in terms of discriminative capability of their common non-overlapping subpaths. To measure the discriminative capability of a subpath,

Algorithm Semantic_Similarity_Measure

Problem Definition. Similarity measurement between two sequences.

Input. Sequences s and t.

Output. Degree of similarity between s and t, $k(s, t)$.

Approach.

1) Build a *match* matrix for the sequences s and t:

$$M_{s,t}[i,j] = \begin{cases} 1 & \text{if } s[i] == t[j], \\ 0 & \text{otherwise.} \end{cases}$$

2) Compute *diagonal-set*=all continuous 1's in all diagonals of *match* matrix.

// save each sub-diagonal of continuous 1's in *diagonal-set*:

diagonal-set= *diagonal-set* ∪ {(first, *len*)}.

3) Find non-overlapping sub-diagonals of continuous 1's in *diagonal-set*. Repeat steps 3.1 to 3.3 until there are no un-marked sub-diagonals.

3.1) Select the longest sub-diagonal in diagonal-set. If some of sub-diagonals, ((i, j), *len*), have the same *len*, then select the one which has the smallest (i–j).

3.2) Mark the selected sub-diagonal.

3.3) Modify the other sub-diagonals to discard overlaps with the selected sub-diagonal.

4) Compute the similarity between s and t. For elements ((i, j), *len*) in *diagonal-set*:

$k(s, t)$+=(*len***info_gain*(subpath related to ((i, j), *len*), X)).

5) Normalize $k(s, t)$.

Fig. 2. Algorithm of measuring similarity between execution paths.

its information gain is measured. The information gain of a subpath is measured in terms of reduction in entropy caused by partitioning the execution paths into two sets of those including and those not including the subpath. The entropy of an execution set X is computed as

$$Ent(X) = -\left(\frac{|X^{\text{pass}}|}{|X|} \log_2 \frac{|X^{\text{pass}}|}{|X|} + \frac{|X^{\text{fail}}|}{|X|} \log_2 \frac{|X^{\text{fail}}|}{|X|} \right), \quad (4)$$

where $|X^{\text{pass}}|$ and $|X^{\text{fail}}|$ indicate the number of passing and failing executions, respectively. The information gain of each common subpath is computed as

$$info\_gain(\text{subpath}) = Ent(X) - \sum_{X_{\text{subset}} \in \{X_{\text{subpath}}, X_{\overline{\text{subpath}}}\}} \frac{|X_{\text{subset}}|}{|X|} Ent(X_{\text{subset}}), \quad (5)$$

where X_{subpath} indicates a subset of X that contains a subpath and $X_{\overline{\text{subpath}}}$ indicates a subset of X that does not contain a subpath. The kernel function is defined as

$$k(s, t) = \phi(s) W W^{\mathrm{T}} \phi(t)^{\mathrm{T}}, \quad (6)$$

where W is a diagonal matrix in which each diagonal element represents the information gain of the corresponding subpath. In step 4 of the algorithm in Fig. 2, we define the kernel function without direct mapping.

A distinct SVM mode M_i is deployed at each instrumented point p_i in the program and estimates the failing or passing state of an unknown execution path e_{ij}, ending at p_i in the j-th test execution of the program, according to the sign of $f(e_{ij})$:

$$f(e_{ij}) = sign\left(\sum_{\rho=1}^{N_{is}} \alpha_{i\rho} C_{i\rho} k(x_{i\rho}, e_{ij}) - b \right), \quad (7)$$

where N_{is} indicates the number of support vectors for the i-th predicate in the program and $\alpha_{i\rho}$ indicates the Lagrange multiplier for ρ-th support vector. The Lagrange multiplier $\alpha_{i\rho}$ associated with the t raining point $x_{i\rho}$ expresses the strength of $x_{i\rho}$ in the final decision function. A remarkable property of this representation is that only a subset of the training points, that is, support vectors, are associated with a non-zero $\alpha_{i\rho}$ [5]. Time complexity of the model at each predicate p_i is O(N_{is}**length*(e_{ij})*max(*length*($x_{i\rho}$))).

III. Empirical Results

To evaluate the effectiveness of the proposed online anomaly detection approach, a research prototype has been developed on two well-known test suites, Siemens [7] and SPEC2000 [8]. Our empirical studies have been targeted to meet the following objectives: i) to examine the quality of anomaly reports, ii) to examine the impact of the code coverage of the training data on the precision of the resultant SVM model, iii) to compare the performance of the proposed semantic kernel with the well-known existing kernels, and iv) to investigate the time overhead of the SVM model.

To measure the quality of the anomaly reports generated by our semantic kernel, an important criterion is the distance between the exact location of a bug from the point where an anomaly has been detected. In Fig. 3, the horizontal axis shows the percentage of manual code examination required to localize the origin of the anomaly. Our approach highly outperforms Argus [9] in finding anomaly origins with a manual code inspection of less than 10%. Our proposed classifier localizes about 60% of the bugs in Siemens by examining about 5% of the entire program, while Argus detects only 38% of the bugs.

The efficiency of an online anomaly detection model to

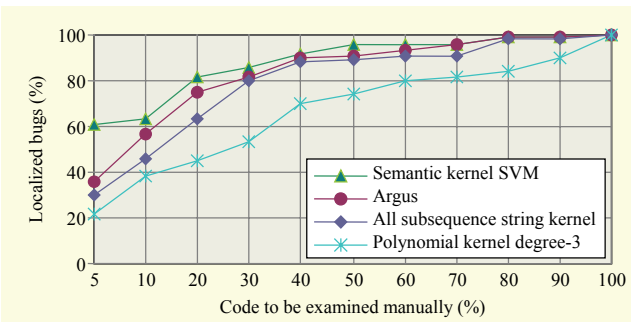

Fig. 3. Anomaly report quality of semantic kernel SVM.

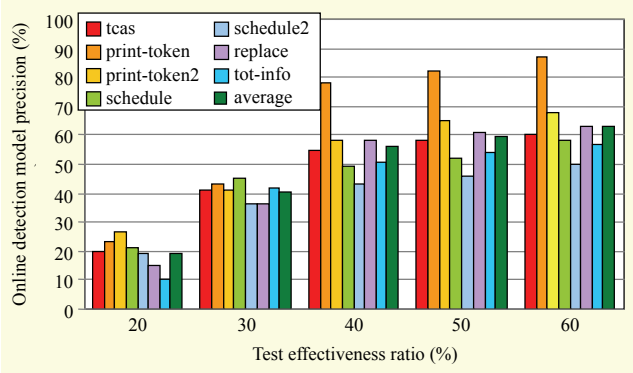

Fig. 4. Precision of SVM model based on TER metric.

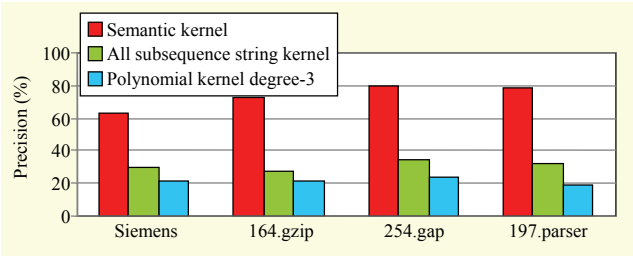

Fig. 5. Precision of semantic kernel compared with other kernels.

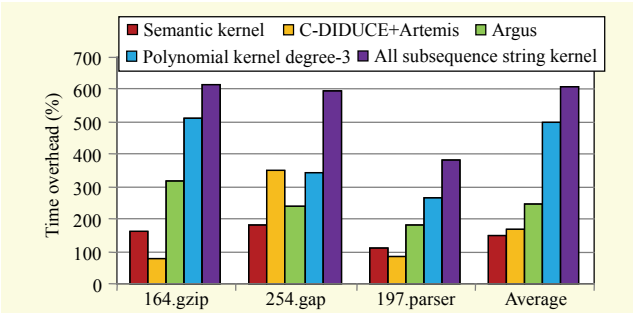

Fig. 6. Time overhead of semantic kernel SVM.

distinguish passing from failing execution paths depends on the test suite used for training the model. Figure 4 shows the impact of the code coverage of the training data set on the precision of the resultant model. The path coverage capability of the test cases selected for training the SVM models was measured using the Testwell CTC++ tool [10]. CTC++ applies a code coverage metric called linear code sequence and jump (LCSAJ) to measure the coverage of acyclic paths provided by a test suite. To measure the LCSAJ coverage capability of an applied test set T, a test effectiveness ratio (TER) metric is used:

$$\text{TER} = \#\text{LCSAJs exercised} / \#\text{feasible LCSAJs}. \quad (8)$$

As shown in Fig. 4, there is a direct relation between the TER and the precision of the trained SVM model in terms of the number of detected bugs.

To have an appropriate basis for estimating the efficiency of our model in relatively large size programs, we compared it with 2 other sequence-based kernels [11] on 3 programs of the SPEC2000 test suite. Since our kernel uses only the longest common non-overlapping subpaths rather than all the subpaths, it is more precise than the existing known kernels (Fig. 5).

In Fig. 6, the runtime overhead of the SVM model is compared with C-DIDUCE [8] and Argus [9] and two other sequence-based SVM kernels. As shown in Fig. 6, the SVM model outperforms the Argus and C-DIDUCE models on average. The SVM model suffers from an average of 150% execution time overhead on SPEC2000 programs.

IV. Conclusion

Precision and performance of SVM for online anomaly detection can be improved by customizing its kernel function. To improve precision, the program execution paths should be mapped into a feature space in which each dimension provides a measure for discriminating execution paths from other failing or passing executions. To speed up the kernel, the dimensions should be independent. To achieve this, the longest non-overlapping common subpaths among execution paths could be considered as the feature space dimensions. Our experiments show that in comparison with the most recent approaches, the proposed model could detect relatively more bugs with less manual effort required to localize the bugs.

References

[1] G.K. Baah, A. Gray, and M.J. Harrold, "On-Line Anomaly Detection of Deployed Software: A Statistical Machine Learning Approach," *Proc. SOQUA*, 2006, pp. 70-77.

[2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2009.

[3] F. Salfner, M. Lenk, and M. Malek, A Survey of Online Failure Prediction Methods, *ACM Comput. Surv.*, 2010, pp. 1-42.

[4] S. Parsa, S. Arabi, and M. Vahidi, "A Learning Approach to Early Bug Prediction in Deployed Software," *Proc. AIMSA*, 2008, pp. 400-404.

[5] R. Herbrich, *Learning Kernel Classifiers Theory and Algorithms*, MIT Press, 2002.

[6] M.J. Harrold, A. Jefferson, and K. Tewary, "An Approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph," *J. Syst. Software*, vol. 36, 1997, pp. 273-295.

[7] Software Infrastructure Repository. http://sir.unl.edu/

[8] L. Fei and S.P. Midkiff, "Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies," *Proc. PLDI*, 2006, pp. 84-95.

[9] L. Fei et al., "Argus: Online Statistical Bug Detection," *Proc. FASE*, 2006, pp. 308-323.

[10] Testwell CTC++ tool. http://www.testwell.fi/

[11] C.-C. Chang, "LIBSVM: A Library for Support Vector Machines," *ACM Trans. Intell. Syst. Technol.*, 2011, pp. 1-27.