


Article

# Software Run-time Entropy: A Novel Type of Indicators for Software Failure Prediction

Shiyi Kong<sup>1,2,†</sup> , Minyan Lu<sup>1,2,†,\*</sup> and Bo Sun<sup>3</sup>

<sup>1</sup> The Key Laboratory on Reliability and Environmental Engineering Technology, Beihang University, Beijing, China, 100191

<sup>2</sup> School of Reliability and System Engineering, Beihang University, Beijing, China, 100191

<sup>3</sup> System Engineering Research Institute, China State Shipbuilding Corporation (CSSC) Electronics Technology Co. LTD, Beijing, China, 100036

\* Correspondence: lmy@buaa.edu.cn (M.L.), buaaksy@buaa.edu.cn (S.K.)

† These authors contributed equally to this work.

**Abstract:** With the development of computer science and software engineering, software becomes more and more complex. Traditional software reliability assurance techniques including software testing and evaluation can't ensure software reliable execution after being deployed. Software failure prediction techniques based on failure indicators can predict software failures according to abnormal indicator values. The latter can be collected using runtime monitoring techniques. An essential part of this method is finding proper indicators which have strong correlation with software failures. We propose a novel type of indicators in this work named software runtime entropy, which takes both software module execution time and call times into consideration. Three common open source software, grep, flex and gzip are used as study cases for finding the relationships between the indicators and software failures. Firstly, a series of fault injection experiments are conducted on those three software respectively. The decision tree algorithm is used to train those data to build the correlation models between software runtime entropy and software failures. Several common measures in machine learning domains such as accuracy, recall rates, and F-measure are used to evaluate the models. The decision tree models can be used as failure mechanisms to assist the failure prediction work. One can examine the value of runtime entropy and make a warning report when it ranges from the normal interval to abnormal one.

**Keywords:** software runtime entropy, failure prediction, indicator

## 1. Introduction

With the development of computer science and software engineering, software takes more and more important roles and becomes more and more complex. Software reliability has become the core of the reliability of modern systems. A research conducted by Software Engineering Institute of Carnegie Mellon University shows that there still are 20% resident errors after software products being deployed[1]. Software Prognostic Health Management(S-PHM), as a kind of runtime software reliability assurance techniques, attracts more and more attentions under such circumstances[2]. S-PHM techniques monitor software behaviors and status simultaneously with software execution process, detect and predict possible failures, and take proper measures. Software failure prediction is an essential part in this process[3]. Existing researches on software failure prediction can be classified into two types, event log based methods and failure mechanisms based methods. Methods based on failure event log are under the constraint of the integrity of log files. One must custom system log by changing system inner log modules to obtain necessary failure information to make a rather precise prediction. Runtime information based methods collect runtime data during software execution process and make prediction based on those data.

Software failure mechanisms refer to the abnormal behaviors and status before software systems failures[4], such as memory usage soaring. Memory usage is an indicator used to predict software failures in this failure mechanism.

Software failure indicators in existing researches are mostly collected from outside of software objects. These indicators are obtained from hardware environments, operation systems or network connection equipment. These indicators mainly focus on performance failures[5]. After investigating a large amount of civil and military software, researchers in [6] present that software functional failures (in other words, content failures) share the largest proportion of all software failures. However, to our best knowledge, there are not many researches focusing on functional failures found during our investigation. Seer is proposed as an online failure prediction approach for functional failure in [7]. Seer uses a series of CPU parameters such as instruction numbers in cache as indicators to predict failures. The accuracy of failure mechanisms is close to 70%. Li uses module execution time and call times between each two modules as indicators to predict functional failures in [5]. Its accuracy is over 95%. However, every function and function calls in the software should be processed and too much data should be collected, stored, and calculated during failure mechanisms built process. This way may cause more resource consumption even performance issues.

To reduce resource consumption while ensuring the accuracy, we introduce software runtime entropy (execution time entropy and call times entropy), a novel type of indicators aiming at software functional failure prediction. Entropy theory is proposed by Shannon in 1948 to measure the uncertainty of information [8]. Entropy theory is combined with software runtime information (module execution duration and invocation times) to depict software runtime status. The correlations between runtime entropy and software functional failures are explored in this work aiming at assisting failure prediction work.

This paper consists of following parts: Section 2 introduces the related work of software failure indicators and entropy theory used in software domains. Section 3 gives the definitions of software runtime entropy. An empirical study used to explore the relationship between software runtime entropy and software failures is given in Section 4. In Section 5 we introduce a possible usage of runtime entropy as an indicator for software failure prediction. The final part is a conclusion of this paper.

## 2. Related Work

### 2.1. Existing Failure Indicators for Software Failure Prediction

Existing failure indicators focus more on symptoms presented outside of software applications. Most of these indicators can be classified as following three levels: operation system level, network connection level and hardware environment level.

Operation system level failure indicators contains number of threads, number of task scheduling timeout, size of swap cache, etc. Network level indicators can be collected by monitoring network equipments or analyzing system log files, for example, bandwidth usage, TCP connections, services response time, etc. Hardware level indicators contains memory usage, CPU instructions, S.M.A.R.T parameters of Hard Disk and so on. These indicators can be collected from the environment where software applications work. Almost all of these indicators are used for performance prediction. There are rare indicators in existing researches aiming at functional failure prediction. Li in his research [9] has addressed that application level indicators are more suitable for functional failure prediction, though those indicators should be collected inner applications via complex process. More details on existing indicators can be found in Table 1.

**Table 1.** Existing Research on Failure Indicators

Models for building failure mechanisms	Prediction goals	Indicators	Types	References
function fitting	resource consumption	operation system loads, memory usage, thread numbers, swap cache, user CPU time rations, kernel CPU rations	hardware and operation system levels	[10–14]
classification methods	operation systems hang and crash failures	number of semaphores, CPU instructions, asynchronous read frequency of cache, number of system calls, paging failure in memory	hardware and operation system levels	[15–21]
system modeling	failure detection	execution path, invocation graph, CPU usage, memory leak, abnormal hard disk access, abnormal network packages	operation system, hardware and networks level	[4,22]
time series analysis	resource consumption of sever clusters	memory usage, service load, response time, timeout, CPU usage, memory leak, hard disk throughput	hardware and network level	[23,24]
	operation system hang, crash failures	memory usage, system call failures, number of semaphores, timeout of task scheduling, throughout of hard disk and sockets	hardware and operation system level	[25]

Trivedi uses Markov Rewards Model modeling workloads and available memory usage of UNIX systems. Then use this model to predict the time when resource exhausted[14]. The UNIX systems workloads are clustered into 11 states. The probabilities of state transition are calculated via hyper exponential distribution fitting on the test data. Then use liner regression methods on each state to calculate resource consumption rewards. Sliva uses line regression methods to build the aging model of Apache SOAP server[13]. Request times per seconds of server and CPU idle time are taken as failure indicators in this model. Irrena et al. use G-SWIFT tool to inject failures, monitor the number of page faults per second in memory, system calls, semaphores and the asynchronous reading frequency of caches, and use SVM to analyze the collected data[15,16,18–20]. His researches are aimed at two kinds of failures: suspension and crash of Windows operating system. The failure prediction model is obtained. On this basis, they study the representativeness of fault injection, the feature selection of failure prediction, and the impact of different system configurations on failure prediction. Yilmaz et al. proposed a failure prediction method based on CPU instruction number of single function

for application software[7,17,21]. This method uses ID3 model to classify data, but the final failure prediction rates of this method is only 50%.

## 2.2. Entropy Theory Used in Software Domains

Entropy theory is proposed by Shannon to measure the uncertainty of information in communication domains[8]. Researchers introduce entropy into software domains aiming at describing the degree of chaos of software systems. Existing researches mainly focus on three orientations:

- using entropy to measure the orderliness of software static structure;
- to measure the change influence from project management aspects;
- to measure the state transformations during the software execution process.

Bansiya uses class as the basic element, use the frequency of name string to build the entropy measurement, and use it to represent the complexity of class[26]. This measurement is related with traditional measurements such as McCabe cycle complexity and number of defects in code. The author uses four large scale open source software (C++ Lang Parser, Win Framework, Graphics Lib 2.0, WinX Windows GUI) as experiment objects to do the analysis and prove the correlations between them. Similarly, Zhang extract neural networks model from software structures, using crossing entropy to locate errors in software[27]. Mannaert uses a hierarchical structure of the system, subsystems and components to model the whole system, calculates entropy of each elements in each level, and addresses the significance of entropy used in software maintenance and evolution[28].

Koutbi constructed software development entropy in [29] by using various activities and their probability distribution in software development process to measure the efficiency of software development activities. The availability of this quantity are also verified through two simulation methods. Hasson proposes an indicator to measure the software change activities from three perspectives of software: defect repair, routine general maintenance and introduction of new functions[30]. Arora used Hasson's indicators to predict the release time of the next version solving NRP (Next Release Problem)[31].

Malik constructs information entropy to measure software performance failure of large scale software systems[32]. Several performance parameters are collected in a specific periods. Researchers fit the distribution of those parameters and calculate the information entropy. A series of open source applications are taken as examples to show the efficiency of this indicator when doing software anomaly detection. Miransky constructs entropy indicators from software execution trace[33]. Software function name and their appearance frequency are counted in this fast trace comparison algorithm. In addition, Hamou also uses entropy to help do the trace analysis work[34]. Wang proposes a DDoS detection method based on entropy via constructing the entropy indicators using IP address and their appearance frequency[35].

## 3. Constructing Software Runtime Entropy Indicators

Shannon proposes a definition of entropy to measure an information source[8]. For a given information source  $X = \{x_1, x_2, x_3, \dots\}$ , the appearance rates of each element  $x_i$  is  $p_i$ . Obviously, there will always be:

$$\sum_{i=1}^{i=n} p_i = 1 \quad (1)$$

The entropy of this information source can be calculated by equation 2 :

$$H = - \sum_{i=1}^{i=n} p_i \log(p_i) \quad (2)$$

We construct two software runtime entropy indicators and use them to depict software execution statuses, then predict software functional failures. Execution trace is usually used to study the software execution process[33]. An example of execution trace is showed in Table 2.

**Table 2.** An Example of Software Execution Trace

ID	Label <sup>a</sup>	Function Name	Timestamp <sup>b</sup>
1	IN	main	10728
2	IN	funcA	10750
3	OUT	funcA	10830
4	IN	funcB	10850
5	IN	funcC	10900
6	OUT	funcC	11000
7	OUT	funcB	11200
8	OUT	main	11300

<sup>a</sup>The label marks if the data collected from the entrance or exit of the function modules.

<sup>b</sup>Timestamp presents the time when obtain this line of data.

We use two parameters to construct software runtime entropy indicators: function module execution duration time and call times between each two modules. Software execution duration time  $T_{Duration}$  can be calculated using equation 3:

$$T_{Duration} = T_{Out} - T_{In} - T_{SubModuleDuration} \quad (3)$$

While  $T_{Out}$  denotes the time when exit a specific function and  $T_{In}$  denotes the time when enter a specific function.  $T_{SubModuleDuration}$  denotes the submodule execution duration time and can be calculated using equation 4:

$$T_{SubModuleDuration} = T_{SubOut} - T_{SubIn} \quad (4)$$

Considering equation 1, the sum of each element appearance rates in an information source is 1. So we define  $p_i$  in equation 2 as the proportion of different modules' execution time in total software execution time. As  $\alpha_i$  in equation 5 :

$$\alpha_i = \frac{T_i^{Duration}}{\sum_{k=1}^{k=n} T_k^{Duration}} \quad (5)$$

The software execution time entropy can be defined as 6:

$$H_t = - \sum_{i=1}^{i=n} \alpha_i \log(\alpha_i) \quad (6)$$

The software call times entropy can be defined as 7:

$$H_c = - \sum_{i=1, j=1}^{i=m, j=n} \beta_{i \rightarrow j} \log(\beta_{i \rightarrow j}) \quad (7)$$

While  $\beta_{i \rightarrow j}$  can be calculated by 8:

$$\beta_{i \rightarrow j} = \frac{N_{i \rightarrow j}}{\sum_{k=1}^{k=m} \sum_{l=1}^{l=n} N_{k \rightarrow l}} \quad (8)$$

#### 4. Correlation Analysis Between Software Run-time Entropy and Failures

After defining software runtime entropy, we design a series of fault injection experiments to collect software runtime trace information to calculate the values of the indicators we introduce above. Failure information is also collected in this process. The entropy values will be marked as *failure* with a tag *Y* or *normal* with a tag *N*. Then those data will be imported into a classification model named C4.5 to train a classifier which can be used to do the prediction work.

##### 4.1. A Brief Introduction of Experiment Software Objects

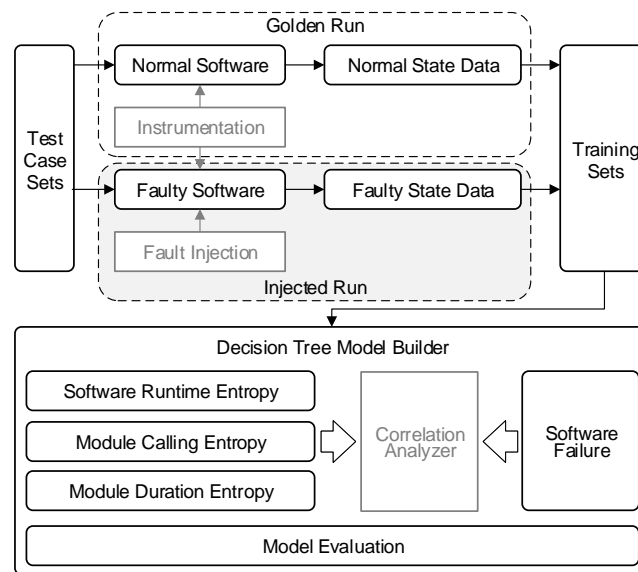
The three software objects used in this work are *grep*, *flex*, *gzip*. All of these three applications can be obtained from the Software-artifact Infrastructure Repository (SIR)[36]. *grep* is a command-line utility for searching plain-text data sets for lines that match a regular expression. *flex* (fast lexical analyzer generator) is a free and open-source software alternative to *lex*. [2] It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). *gzip* is a file format and a software application used for file compression and decompression. More bases of those three objects can be found in Table 3.

**Table 3.** Basic Information of Three Software Applications

Name	<i>grep</i>	<i>flex</i>	<i>gzip</i>
<b>Languages</b>	C	C	C
<b>Size</b>	10068 LOC	10459 LOC	5680 LOC
<b>Procedures</b>	146	162	104
<b>Versions</b>	6	6	6
<b>Fault seeds</b>	18	19	14
<b>Test cases</b>	470	525	214

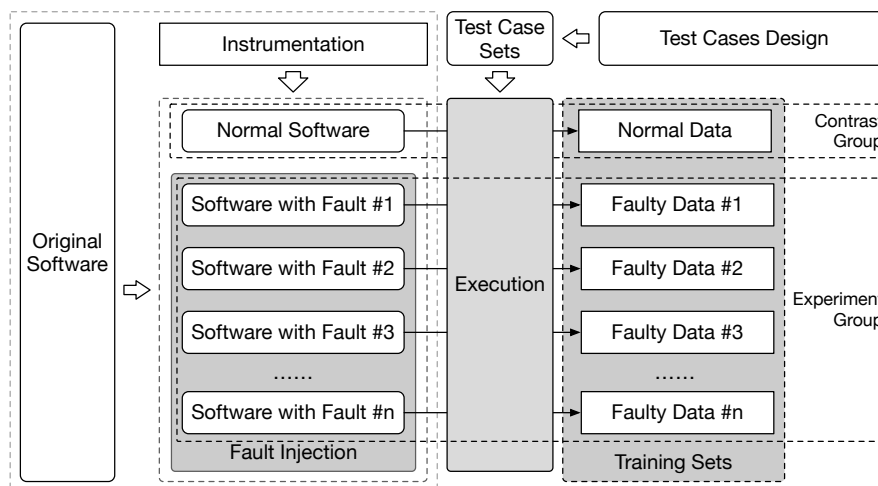
##### 4.2. Framework of the Empirical Study

Existing open source data sets (e.g. MDP from NASA, PROMISE) for software static analysis just consist of static information of software, so they cannot be used to do the research on failure mechanisms or failure prediction. Both failure mechanisms and failure prediction work need to run the software and collect runtime information. The lack for data cause existing failure mechanisms usually comes from experience, like that more than 90% memory usage will cause failures. Those mechanisms usually used in performance prediction to predict when the resource will be exhausted. Because software functional failure are not so intuitive that we need more information inner software. An another important reason is that the functional failures are usually not closely related with resource consumption. We propose a series of fault injection experiments to obtain software runtime data to calculate software runtime entropy. The total framework of this research is shown in Figure 1.



**Figure 1.** Framework of the empirical study.

Firstly, we use macro compile methods add fault seeds in source code, and design test case to trigger those faults. These process are shown in Figure 2.



**Figure 2.** Process of fault injection experiments.

Secondly, we preprocess the traces obtained from the experiments. Two kinds of runtime entropy indicators are calculated in these process and marked as *failure* with a *Y* or *normal* with a *N*. Table 4 shows an example of the data after preprocessing.

**Table 4.** An Example of the Data After Preprocessing

ID	Execution time entropy $H_t$	Call Times entropy $H_c$	Tag
1	125	121	Y
2	328	320	N
3	451	460	Y
...	...	...	...

Then, a decision tree model named C4.5 is used to train the classification model to explore the relationship between runtime entropy and software functional failures. Some common used parameters for model verification, like confusion matrix, accuracy, F-measure, are used to verify the decision model.

#### 4.3. Correlation Analysis

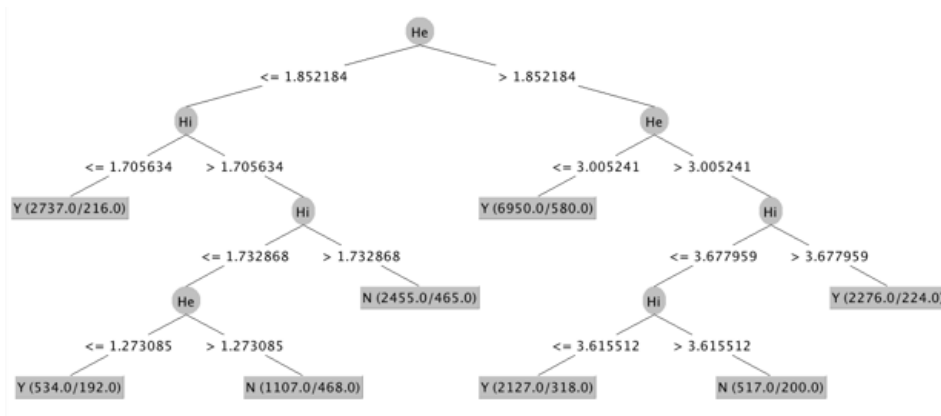
Execution results of the fault injection experiments are shown in Table 5.

**Table 5.** Execution Results of Three Software Objects

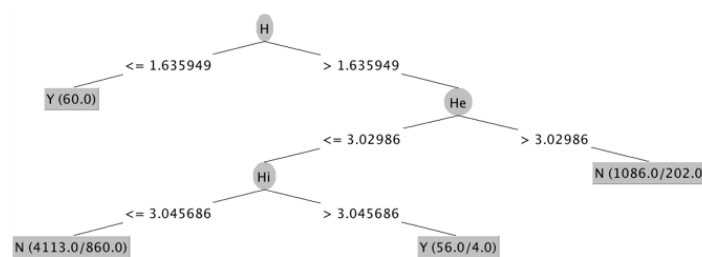
Software	Failed Execution	Normal Execution	Total Execution
<i>grep</i>	755	8133	8888
<i>flex</i>	4663	587	5250
<i>gzip</i>	185	3025	3210

The data is preprocessed into the format as table 4 shows. Using decision tree classification algorithm C4.5, the data obtained from fault injection experiments are analyzed separately. 10 folds cross validation method is chosen to reduce the deviation caused by data selection. The data of each software is divided into 10 parts. Nine of them are used as training sets and the other one is used as test set in the process of building decision tree. This process is repeated 10 times, so that each part participates in the process of building decision tree as training sets.

The decision trees of three software objects are shown in Following three figures. *grep* in Figure 3, *flex* in Figure 4, and *gzip* in Figure 5.



**Figure 3.** Decision tree model of *grep*.



**Figure 4.** Decision tree model of *flex*.



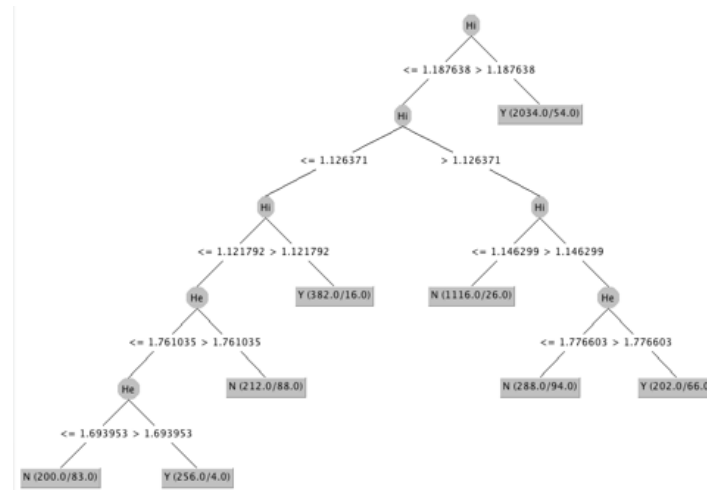


Figure 5. Decision tree model of *gzip*.

#### 4.4. Model Validation

After obtaining the decision tree models, we need to examine whether the models can truly reflect the relationship between variables involved when building the model (the relationship between software runtime entropy and software failures). Some common used measurements in machine learning areas are selected to evaluate the accuracy of these models.

Confusion matrix is used first as shown in Table 6.

Table 6. Confusion Matrix

	Classified as true (failure)	Classified as false (normal)
Real true	True Positive (TP)	False Negative (FN)
Real false	False Positive (FP)	True Negative (TN)

The confusion matrix of three decision trees are shown in equation 9, 10, and 11 respectively.

$$\begin{pmatrix} Y & N \\ 13204 & 123 & Y \\ 250 & 3226 & N \end{pmatrix} \quad (9)$$

$$\begin{pmatrix} Y & N \\ 4107 & 143 & Y \\ 34 & 2137 & N \end{pmatrix} \quad (10)$$

$$\begin{pmatrix} Y & N \\ 12735 & 290 & Y \\ 190 & 1475 & N \end{pmatrix} \quad (11)$$

True Positive Rates (TPR) refer the percentage of the failure data that classified as failure among total failure data, as equation 12 shows.

$$TPR = \frac{TP}{TP + FN} \quad (12)$$

Table 7. Measurements and Comparison

Software	Origins	Precision	True Positive Rates	False Positive Rates	F-measure
<i>grep</i>	here	0.981	0.990	0.072	0.986
	in [5]	0.990	0.990	0.083	0.990
	errors	0.91%	0	13.2%	0.4%
<i>flex</i>	here	0.991	0.966	0.156	0.978
	in [5]	0.953	0.954	0.132	0.953
	errors	3.98%	1.25%	18.1%	2.6%
<i>gzip</i>	here	0.985	0.977	0.114	0.981
	in [5]	0.991	0.991	0.138	0.990
	errors	0.6%	1.41%	17.3%	0.90%

False Positive Rates (FPR) is given in equation 13.

$$FPR = \frac{FP}{FP + TN} \quad (13)$$

Precision and F-measure can be calculated using equation 14, 15 respectively. We use  $\beta = 1$  (i.e.  $F_1$ ) in this paper.

$$Precision = \frac{TP}{TP + FP} \quad (14)$$

$$F - measure = \frac{(1 + \beta^2)TP \times Precision}{\beta^2 Precision + TP} \quad (15)$$

The results of above measurements and the comparison with the results in [5] are shown in Table 7.

The runtime entropy indicators of three software objects show a strong correlation with the occurrence of software failures. In *grep* and *gzip*, the ratio of normal runs to failure runs is relatively uniform. TPR and FPR of the model are in the normal range and have good classification results. In *flex*, the failure execution numbers is much larger than normal runs. The unbalanced data make the results have a higher FPR than the other two applications and cause more false alarms.

The results in the Table 7 also show that software entropy indicators have the equal accuracy with the indicators proposed in [5]. During the process we training data, there are only two features in our model. While in [5], there are so many features which make the training process more complex and consume more resource and time. The entropy indicators are calculated during software execution process which cost less storage to store large quantitative data like the methods in [5]. There will also be a same circumstance in runtime prediction process.

## 5. A Possible Approach to Apply the Decision tree into Failure Prediction Work

In this section, we give a possible method to apply the decision tree model into failure prediction work.

The entropy indicators are continue calculated during software execution according to equation 16 and 17. By this way, we get two functions to depict entropy indicators changing against time.

$$H_t(t) = - \sum_{i=1}^{i=n} \alpha_i(t) \log(\alpha_i(t)) \quad (16)$$

$$H_c(t) = - \sum_{i=1}^{i=m} \sum_{j=1}^{j=n} \beta_{i \rightarrow j}(t) \log(\beta_{i \rightarrow j}(t)) \quad (17)$$

While the  $\alpha_i(t)$  in equation 16 can be calculated using equation 18.  $T_i(t)$  denotes that in a specific time  $t$ , the accumulating execution time of  $i^{th}$  module. The total numbers of modules is  $n$ .

$$\alpha_i(t) = \frac{T_i(t)}{\sum_{k=1}^{k=n} T_k(t)} \quad (18)$$

The  $\beta_{i \rightarrow j}(t)$  can be calculated using equation 19.

$$\beta_{i \rightarrow j}(t) = \frac{N_{i \rightarrow j}(t)}{\sum_{k=1}^{k=m} \sum_{l=1}^{l=n} N_{k \rightarrow l}(t)} \quad (19)$$

The  $N_{i \rightarrow j}(t)$  means that in a specific time  $t$ , the total numbers that the  $i^{th}$  module calls the  $j^{th}$  module.

The decision tree model can be transform into a series of areas  $H_{failed}$  in  $\mathbb{R}^2$ . When the point  $H(t_i) = (H_t(t_i), H_c(t_i))$  is in the areas  $H_{failed}$ , it means the system has failed. We choose a neighborhood of  $H_{failed}$  named  $H_\delta$ . When the point is in the  $H_\delta$ , we make a warning and predict that the system will failed in a short period of time. The prediction process is shown in Fig.6. We make a warning at point  $H_4$ .

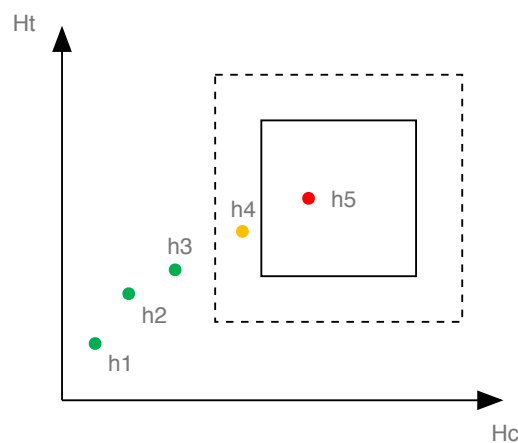


Figure 6. Principle of failure prediction using decision tree model.

## 6. Conclusions

Two indicators for software functional failure prediction are proposed in this work, software runtime execution time entropy and software runtime call times entropy.

Two indicators are well defined according to Shannon's Entropy Theory. A series of fault injection experiments are designed to obtain runtime information to calculate those two indicators. By this way, the issue that lacking for real data has been addressed.

Those data are processed and trained with a machine learning algorithm named C4.5. After the training process, we obtain decision tree models and use a series common measurements to validate the models. The results show that there is a strong correlation between software entropy indicators and software failures. We also make a comparison with another similar work. The results of comparison shows that our indicators own the equal accuracy and consume much less time and resource.

At the end of this work, we give a simple idea on how to apply the decision tree model into failure prediction work. This idea may need future work to prove and implement it. In addition, how to reduce the false alarm rates when there are more failure data than normal data is also an interesting issue that need to be solved in future work.

**Author Contributions:** Conceptualization, Shiyi Kong and Minyan Lu; Data curation, Shiyi Kong; Investigation, Shiyi Kong and Bo Sun; Methodology, Shiyi Kong; Software, Shiyi Kong and Bo Sun; Supervision, Minyan Lu; Validation, Shiyi Kong and Minyan Lu; Writing - original draft, Shiyi Kong; Writing - review & editing, Shiyi Kong and Minyan Lu.

**Funding:** This research was funded in part by the Funding of the Key Laboratory on Reliability and Environmental Engineering Technology under Grant No. 614200404011017, in part by the Advanced Research Domains Funding under Grant No. JZX7Y20190242013901.

**Conflicts of Interest:** The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

## Abbreviations

The following abbreviations are used in this manuscript:

S-PHM	Software Prognostic Health Management
NRP	Next Release Problem
SIR	Software-artifact Infrastructure Repository

## References

- Feiler, P.H.; Goodenough, J.B.; Gurfinkel, A.; Weinstock, C.B.; Wrage, L. Reliability validation and improvement framework. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2012.
- Srivastava, A.N.; Schumann, J. The Case for Software Health Management. 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, 2011, pp. 3–9. doi:10.1109/SMC-IT.2011.14.
- Salfner, F.; Lenk, M.; Malek, M. A Survey of Online Failure Prediction Methods. *ACM Comput. Surv.* **2010**, *42*, 10:1–10:42. doi:10.1145/1670679.1670680.
- Kiciman, E.; Fox, A. Detecting application level failures in component based Internet services. *IEEE Transactions on Neural Networks* **2005**, *16*, 1027–1041. doi:10.1109/TNN.2005.853411.
- Li, L.; Lu, M.; Gu, T. Extracting Interaction-Related Failure Indicators for Online Detection and Prediction of Content Failures. 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, pp. 278–285. doi:10.1109/ISSREW.2018.00019.
- Grottke, M.; Nikora, A.P.; Trivedi, K.S. An empirical investigation of fault types in space mission system software. 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), 2010, pp. 447–456. doi:10.1109/DSN.2010.5544284.
- Ozcelik, B.; Yilmaz, C. Seer: A Lightweight Online Failure Prediction Approach. *IEEE Transactions on Software Engineering* **2016**, *42*, 26–46. doi:10.1109/TSE.2015.2442577.
- Shannon, C.E. A Mathematical Theory of Communication. *Bell System Technical Journal* **1948**, *27*, 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.
- Li, L.; Lu, M.; Gu, T. A Systematic Modeling Approach for Failure Indicators of Complex Software Intensive Systems. 2018 12th International Conference on Reliability, Maintainability, and Safety (ICRMS), 2018, pp. 43–51. doi:10.1109/ICRMS.2018.00019.
- Andrzejak, A.; Silva, L. Deterministic Models of Software Aging and Optimal Rejuvenation Schedules. 2007 10th IFIP/IEEE International Symposium on Integrated Network Management, 2007, pp. 159–168. doi:10.1109/INM.2007.374780.
- Hoffmann, G.A.; Trivedi, K.S.; Malek, M. A Best Practice Guide to Resource Forecasting for Computing Systems. *IEEE Transactions on Reliability* **2007**, *56*, 615–628. doi:10.1109/TR.2007.909764.
- Pellegrini, A.; Sanzo, P.D.; Avresky, D.R. A Machine Learning-Based Framework for Building Application Failure Prediction Models. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 2015, pp. 1072–1081. doi:10.1109/IPDPSW.2015.110.
- Andrzejak, A.; Silva, L. Deterministic Models of Software Aging and Optimal Rejuvenation Schedules. 2007 10th IFIP/IEEE International Symposium on Integrated Network Management, 2007, pp. 159–168. doi:10.1109/INM.2007.374780.

14. Vaidyanathan, K.; Trivedi, K.S. A measurement-based model for estimation of resource exhaustion in operational software systems. *Proceedings 10th International Symposium on Software Reliability Engineering*, 1999, pp. 84–93. doi:10.1109/ISSRE.1999.809313.
15. Irrera, I.; Vieira, M. A Practical Approach for Generating Failure Data for Assessing and Comparing Failure Prediction Algorithms. *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, 2014, pp. 86–95. doi:10.1109/PRDC.2014.19.
16. Irrera, I.; Duraes, J.; Madeira, H.; Vieira, M. Assessing the Impact of Virtualization on the Generation of Failure Prediction Data. *2013 Sixth Latin-American Symposium on Dependable Computing*, 2013, pp. 92–97. doi:10.1109/LADC.2013.24.
17. Yilmaz, C.; Porter, A. Combining Hardware and Software Instrumentation to Classify Program Executions. *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*; ACM: New York, NY, USA, 2010; FSE '10, pp. 67–76. doi:10.1145/1882291.1882304.
18. Irrera, I.; Vieira, M. Towards Assessing Representativeness of Fault Injection-Generated Failure Data for Online Failure Prediction. *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, 2015, pp. 75–80. doi:10.1109/DSN-W.2015.24.
19. Irrera, I.; Duraes, J.; Vieira, M.; Madeira, H. Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults. *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, 2010, pp. 3–10. doi:10.1109/PRDC.2010.51.
20. Irrera, I.; Pereira, C.; Vieira, M. The Time Dimension in Predicting Failures: A Case Study. *2013 Sixth Latin-American Symposium on Dependable Computing*, 2013, pp. 86–91.
21. Yilmaz, C.; Paradkar, A.; Williams, C. Time will tell. *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 81–90. doi:10.1145/1368088.1368100.
22. Sauvanaud, C.; Silvestre, G.; Kaâniche, M.; Kanoun, K. Data Stream Clustering for Online Anomaly Detection in Cloud Applications. *2015 11th European Dependable Computing Conference (EDCC)*, 2015, pp. 120–131. doi:10.1109/EDCC.2015.22.
23. Meng, H.; Qi, Y.; Di Hou.; Chen, Y. A Rough Wavelet Network Model with Genetic Algorithm and its Application to Aging Forecasting of Application Server. *2007 International Conference on Machine Learning and Cybernetics*, 2007, Vol. 5, pp. 3034–3039. doi:10.1109/ICMLC.2007.4370668.
24. Hong, B.; Peng, F.; Deng, B.; Hu, Y.; Wang, D. DAC-Hmm: detecting anomaly in cloud systems with hidden Markov models. *Concurrency and Computation: Practice and Experience*, 27, 5749–5764. doi:10.1002/cpe.3640.
25. Crowell, J.; Shereshevsky, M.; Cukic, B. Using fractal analysis to model software aging. Technical report, Technical report, West Virginia University, Lane Department of CSEE, 2002.
26. Bansiya, J.; Davis, C.; Etzkorn, L. An entropy-based complexity measure for object-oriented designs. *Theory and Practice of Object Systems* **1999**, 5, 111–118. doi:10.1002/(SICI)1096-9942(1999)5:2<111::AID-TAPO4>3.0.CO;2-0.
27. Zhang, X.; Ben, K.; Zeng, J. Cross-Entropy: A New Metric for Software Defect Prediction. *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 111–122. doi:10.1109/QRS.2018.00025.
28. Mannaert, H.; Bruyn, P.; Verelst, J. Exploring Entropy in Software Systems: Towards a Precise Definition and Design Rules. *The Seventh International Conference on Systems (ICONS 2012)*, 2012, pp. 93–99.
29. El Koutbi, S.; Idri, A.; Abran, A. Empirical evaluation of an entropy-based approach to estimation variation of software development effort. *Journal of Software: Evolution and Process*, 31, e2149. doi:10.1002/smr.2149.
30. Hassan, A.E. Predicting Faults Using the Complexity of Code Changes. *Proceedings of the 31st International Conference on Software Engineering*; IEEE Computer Society: Washington, DC, USA, 2009; ICSE '09, pp. 78–88. doi:10.1109/ICSE.2009.5070510.
31. Arora, H.D.; Sahni, R.; Parveen, T. Prototype Evidence for Estimation of Release Time for Open-Source Software Using Shannon Entropy Measure. *Proceedings of Fifth International Conference on Soft Computing for Problem Solving*; Pant, M.; Deep, K.; Bansal, J.C.; Nagar, A.; Das, K.N., Eds.; Springer Singapore: Singapore, 2016; pp. 683–690.
32. Malik, H.; Shakshuki, E.M. Detecting Performance Anomalies in Large-scale Software Systems Using Entropy. *Personal Ubiquitous Comput.* **2017**, 21, 1127–1137. doi:10.1007/s00779-017-1036-y.
33. Miranskyy, A.; Davison, M.; Reesor, R.; Murtaza, S. Using entropy measures for comparison of software traces. *Information Sciences* **2012**, 203, 59 – 72.

34. Hamou-Lhadj, A. Measuring the Complexity of Traces Using Shannon Entropy. Fifth International Conference on Information Technology: New Generations (itng 2008), 2008, pp. 489–494. doi:10.1109/ITNG.2008.169.
35. Wang, R.; Jia, Z.; Ju, L. An Entropy-Based Distributed DDoS Detection Mechanism in Software-Defined Networking. 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, Vol. 1, pp. 310–317. doi:10.1109/Trustcom.2015.389.
36. Do, H.; Elbaum, S.G.; Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* **2005**, *10*, 405–435.