AN ML-BASED MEMORY LEAK DETECTION SCHEME FOR NETWORK DEVICES

Minghui Wang, Jiangxuan Xie, Xinan Yang, Xiangqiao Ao

AI Research Institute, H3C Technology Co., Ltd, China

ABSTRACT

The network is very important to the normal operation of all aspects of society and economy, and the memory leak of network device is a software failure that seriously damages the stability of the system. Some common memory checking tools are not suitable for network devices that are running online, so the operation staff can only constantly monitor the memory usage and infer from experience, which has been proved to be inefficient and unreliable. In this paper we proposed a novel memory leak detection method for network devices based on Machine learning. It first eliminates the impact of large-scale resource table entries on the memory utilization. Then, by analyzing its monotonicity and computing the correlation coefficient with the memory leak sequence sets pre constructed by simulation, the memory leak fault can be found in time. The simulation experiments show that the scheme is computationally efficient and the precision rate is close to 100%, it works well in the actual network environment, and has excellent performance.

KEYWORDS

Memory leak, Resource table entry utilization, Correlation coefficient, Time Sequence monotonicity, Machine Learning.

1. INTRODUCTION

The network is essential to the normal operation of all aspects of society and economy. Many large-scale network systems require 7x24h uninterrupted operation. Network quality is largely reflected in the stability and data forwarding ability of network device under long-term work. These network devices include switches, routers, firewalls, etc. They may have some software or hardware failures from time to time, and memory leakage is one of the serious software problems. It seriously damages the stability of the system, even leads to system crash or device restart.

Generally speaking, memory leak refers to that the application program does not release the memory in time after using it, and the memory can no longer be used by other applications. In severe cases, the memory will gradually run out, other applications will not be able to apply for the memory, and the system will crash eventually.

Memory leak detection includes static analysis and dynamic monitoring. Static analysis [1-4] is usually lexical, grammatical checking and type analysis for source code. The dynamic monitoring method [5-7] is to insert memory leak detection code at the location of memory operation to track memory usage, and report detailed information when a leak occurs. These methods require complex resource managements and modifications to the original application code. Although these software can effectively detect memory leaks, they have a relatively large run-time overhead and tend to reduce the efficiency of the system [8].Therefore, these types of check are generally disabled for the officially released software version, especially for network devices.

David C. Wyld: CSITY, NWCOM, SIGPRO, ASOFT, AIFZ, BDIoT, ITCCMA, CLSB, DTMN, MLNLP - 2021
pp. 27-40, 2021. CS & IT - CSCP 2021 DOI: 10.5121/csit.2021.111403

For network devices, a common way to find memory leaks is to constantly monitor memory usage. If you find that the memory usage of some application is increasing and significantly exceeds the normal allowed level, consider the possibility of memory leaks. However, this method depends heavily on the experience of the operation staff, which means low efficiency and poor reliability.

In response to the urgent need for online device memory leak detection, this paper designs a memory leak detection method for network device based on Machine Learning(ML). By periodically monitoring the number of resource table entries (ARP, Route, MAC, ACL, etc.) and memory utilization of network devices, we can judge whether there is a potential memory leak and evaluate the time to reach the memory alarm threshold. Furthermore, the detailed memory usage information of the device can be obtained through the rule engine, which is convenient for the operation staff to complete the specific problem diagnosis. This method can detect the device memory leakage fault conveniently and quickly. It has the characteristics of accuracy, high efficiency, strong practicability and wide application range. This feature has been applied to Seer Analyzer—the network analyzer product of H3C company, and has achieved excellent results.

The rest of this paper is organized as follows. In section II some commonly used memory leak detection methods for network device are briefly described. Section III proposes a memory leak detection algorithm based on machine learning. Section IV provides some follow-up processing. Finally, experiment results and conclusions are presented in sections V and VI, respectively.

2. BACKGROUND AND PROBLEM

As mentioned before, for network devices, a common way to find memory leaks is to constantly monitor memory usage. When memory occupancy is abnormal (for example, the memory exceeds the alarm threshold, and the memory size increases abnormally), the operation staff check the occupancy of each memory block, analyze the allocation and release of suspicious memory blocks and their relationship with related applications, so as to speculate whether the application program has memory leakage fault.

However, this judgment process may cause false positives in memory leak alarming. For example, the increase in the usage of some memory blocks is normal. Only by combining the memory growth rate and the memory footprint with continuous observation can an accurate judgment be made. Therefore, the use of this method depends heavily on the experience of the operation staff, which means low efficiency and poor reliability.

In addition, some memory leaks are very slow and require long time (even months) to monitor and analysis, which greatly increases the difficulty of finding anomalies. Sometimes the continuous increasing of a small amount of memory may not be a problem. For example, the syslog data generated during system operation will be stored in the memory file, causing the memory to increase gradually until the file is written into the Flash or Disk periodically.

The testing department often uses the following methods to check for memory leaks: repeatedly delivering and deleting configurations, repeatedly delivering and deleting resource table entries, leaving the configuration or entries unchanged for a period of time to see if there is a big memory change. Among them, "repeatedly delivering and deleting configurations" rarely appear on actual network devices, so it is not considered here.
Due to the time limit of the software version plan, it is almost impossible to test the device for too long. However memory leaks are often related to some configurations or scenarios that require long-time observation to be found [9]. Considering this, memory leaks are not supposed to be found completely during the test phase. Consequently, it is necessary to monitor the device

memory, use big data and ML technologies to store and analyze the data, and detect device memory leaks online.

3. MEMORY LEAK DETECTION ALGORITHM BASED ON MACHINE LEARNING

In this section, we will introduce the memory leak detection algorithm based on machine learning. First of all, we exclude the influence of large-scale resource table entries on the memory utilization of devices and get a new time series called M' sequence. We preliminarily judge whether there is the risk of memory leakage in this new sequence according to its monotonic rising property. Moreover, we construct several memory leak sequence sets by simulation and compare the M' sequence with it. If the average correlation coefficient obtained is greater than a specified threshold value, it is further confirmed as an outlier.

The proposed scheme is as follows:

3.1. Eliminate the impact of large-scale resource entries on the device memory utilization, and get a new time series

As we know, most of the large-scale resource table entries on the device, such as ARP table entries, Routing table entries, MAC table entries and ACL table entries, are dynamically delivered and deleted, and the large size of these table has a great impact on the device memory utilization.

Therefore, without considering these resource table entries, it doesn't make sense to just monitor the changes in the device memory utilization. Only when the impact of these large-scale resource entries is excluded can the risk of possible memory leaks be exposed.

Network device, generally refers to switch or router, its large-scale resource table entries mainly include ARP entries, Routing entries, MAC entries, ACL entries, etc.

Let the utilization rate of ARP, Route, MAC and ACL entries be a%, b%, c% and d%, respectively. Here, the utilization rate of ARP entries represents the number of ARP entries currently used divided by the maximum ARP entry specification, and similar definitions for other entries. As follows show:

$$\begin{aligned} a\% &= arp\_num\_used \,/\, arp\_num\_total \\ b\% &= route\_num\_used \,/\, route\_num\_total \\ c\% &= mac\_num\_used \,/\, mac\_num\_total \\ d\% &= acl\_num\_used \,/\, acl\_num\_total \end{aligned} \tag{1}$$

Further, let arp_size represent the size of the memory occupied by each ARP entry, then $arp\_size\_total = arp\_size \times arp\_num\_total$ represents the memory size occupied by the ARP maximum entry specification, and similar definitions for $route\_size\_total$, mac_size_total, and arp_size_total.

Now, we can calculate the weighted sum of the utilization of these resource entries:

$$x\% = w_1 \times a\% + w_2 \times b\% + w_3 \times c\% + w_4 \times d\% \tag{2}$$

Where w_1, w_2, w_3, and w_4 are defined as follows represents the weight values of each resource table entry's impact on device memory:

$$
\begin{aligned}
w_1&=arp\_size\_total/(arp\_size\_total+route\_size\_total+mac\_size\_total+acl\_size\_total)\\
w_2&=route\_size\_total/(arp\_size\_total+route\_size\_total+mac\_size\_total+acl\_size\_total)\\
w_3&=mac\_size\_total/(arp\_size\_total+route\_size\_total+mac\_size\_total+acl\_size\_total)\\
w_4&=acl\_size\_total/(arp\_size\_total+route\_size\_total+mac\_size\_total+acl\_size\_total)
\end{aligned}
\tag{3}
$$

Using gRPC[10](Google Remote Procedure Call) technology, network devices periodically send the utilization rate of memory and the resource tables to the analyzer, each of these data forms a time series.

$$
\begin{aligned}
M&=(m_1, \cdots, m_n)\\
A&=(a_1, \cdots, a_n)\\
B&=(b_1, \cdots, b_n)\\
C&=(c_1, \cdots, c_n)\\
D&=(d_1, \cdots, d_n)
\end{aligned}
\tag{4}
$$

Where M, A, B, C, D represent the time series composed of device memory utilization, ARP entry utilization, Route entry utilization, MAC entry utilization and ACL entry utilization.

From *A*, *B*, *C*, *D*, we can calculate their weighted sum at each time point to get a new time sequence.

$$R=(r_1, \cdots, r_n),\quad r_i=w_1\times a_i+w_2\times b_i+w_3\times c_i+w_4\times d_i,\ i=1, \cdots, n \tag{5}$$

Where the value range of r_i is [0,100]. We divide the range into 500 equal-length cells with fixed length of 0.2. Then the number of elements of the sequence *R* falling between each cell is calculated and the cell with the largest number is regarded as cell *u*.

Suppose there are *p* elements in *R* whose values belong to cell *u* and the corresponding time points are t_1', t_2',...t_p'. Correspondingly, the sequence *M* takes values at these time points to form a new sequence：

$$M'=(m_1', \cdots, m_p') \tag{6}$$

Next, we will examine the monotonicity of the new sequence and its correlation with some known memory leak sequences to figure out whether the device has memory leakage.

It should be noted that ARP, Route, MAC and ACL table we selected here are the most common large-scale resource tables. If there are other more large-scale resource tables that may seriously affect the memory utilization, they also need to be considered here to participate in the calculation.

For network device, the weight values w_1, w_2, w_3, and w_4 barely change with the software version. Only when there are many significant changes in the processing of related resource table entries in the software version, their values need to be determined again. Experiments show that even if the values of these weights are slightly changed, the analysis results will not be affected.

3.2. Determine the monotonicity of the M' sequence

For the memory sequence *M*' obtained in the previous step, we know that the sum of the memory occupied by the large-scale resource table entries at each time point is approximately equal. In this section, we will judge the monotonicity of *M*' sequence by calculating the correlation coefficient between the *M*' sequence and its index sequence (*I*=(*1,2,3,...,p*)).

Regarding memory leakage, the memory utilization will keep rising with the passage of time without the impact of large-scale resource entries. Therefore, if the *M*' sequence follows the monotonous upward trend, it indicates that there may be a memory leak. The correlation coefficient between *M*' and its index sequence is calculated. Here we use the Spearman algorithm [11], which is a rank correlation algorithm and does not require the assumption of bivariate normal distribution, so it has a better effect than the Pearson algorithm in this case. Here, we set the threshold value of correlation coefficient as 0.9, above which represents the monotonic rising trend of *M*' series. See the following experimental section for details.

In order to eliminate random interference, only *M*' sequence with the memory increment (which is the difference value between the first item and the last item) exceeding the specified threshold (e.g. 30Mbytes) will be regarded as the potential memory leakage sequence. Consequently, the interference of a small increase of memory caused by syslog and diagnostic information of each software module will be successfully eliminated. In other words, for the sequence (m_1', …, m_p'), it is required to satisfy $(m_p'-m_1')\%*total\_mem > 30M$, where *total_mem* is the total memory of the device.

To sum up, if the sequence meets two conditions: time monotonic growth and the memory increment exceeds the specified threshold, then the possibility of memory leak of the device can be preliminarily judged.

3.3. Calculate the average correlation coefficient between M' and simulation-constructed memory leak sequence sets

In this section, we first assume some memory leak scenarios and construct a set of corresponding time series. Their *M*' sequences are calculated respectively to form a simulated memory leak sequence set.

Due to the monotonous rising trend of memory leakage *M*' sequence, it has strong correlation with simulated memory leak sequence; while the normal *M*' sequence shows random fluctuation trend, and the correlation with simulated memory leak sequence is very weak. We calculate the correlation coefficients of the *M*' sequence and a pre-generated simulated memory leak sequence by Spearman algorithm and take the average. If the average correlation coefficient is greater than 0.9, it indicates that the *M*' sequence has a memory leak. Please see the following experimental section for details.

The processing flow chart of the scheme described in this section is shown below：

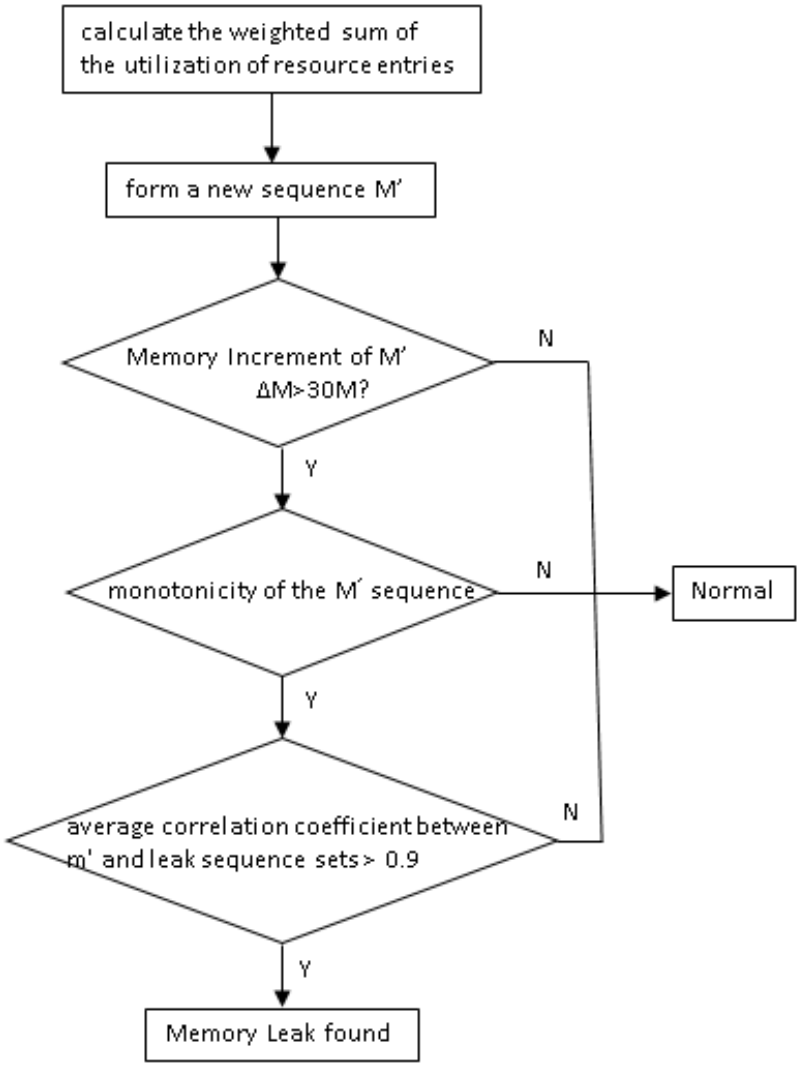

Figure 1. Flow chart of the scheme

4. FURTHER PROCESSING

In this section, we first assume some memory leak scenarios and construct a set of corresponding time series. Their *M*' sequences are calculated respectively

If it is determined that a memory leak has occurred in a certain *M*' sequence, by calculating the memory increment *M_Diff* of the sequence and the corresponding time difference *T_Diff*, we can get the average growth rate of memory leakage $v=M\_diff/T\_diff$. Consequently, we can estimate the time when the memory reaches the alarm threshold which help the operation staff to reasonably arrange the time for fault handling. Specifically, the corresponding estimated time is: $Est\_T=total\_mem\times(alarm\%-curr\%)/v$ where *alarm*% represents the memory alarm threshold, *curr*% represents the current memory utilization, and *total_mem* represents the total memory size of the device, then the memory growth space left to reach the alarm threshold is $total\_mem\times(alarm\%-curr\%)$.

After detecting the memory abnormality, further memory diagnostic tools can be used. As known, the rule engine [12-13] is a component embedded in the application which can separate business rules from the business code and use pre-defined semantic specifications to implement these separated rules. Given input data, the rule engine perform evaluate rules and make decisions. With the aid of rule engine, we use NETCONF [10] (Network Configuration Protocol) to interactively obtain the detailed information of each memory block from the device, it can assist the operation staff to further confirm the problem and find the faulty application module.

5. SIMULATION EXPERIMENTS AND RESULTS

In the simulation experiments, we collected the data of the utilization rate of each resource table entry and the device memory utilization rate periodically. We collected the data every 5 minutes over 3 months and the total data point is $288 \times 30 \times 3 = 25920$. Here 3 scenarios are involved: the normal situation without memory leaks, the memory leaks of ARP resource entries, and the random memory leaks of an unknown software module. We use python to complete the simulations.

The following are the graphs of simulated device memory in three scenarios：

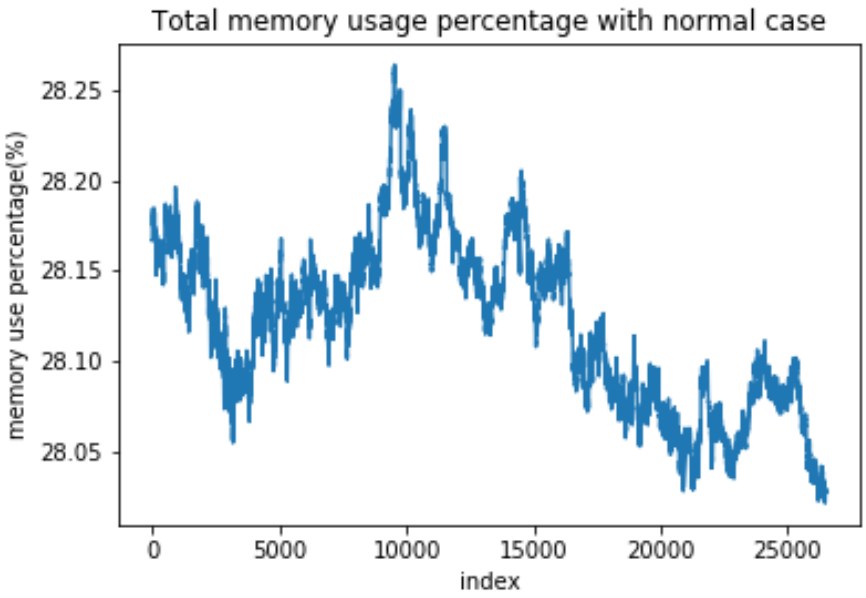

Figure 2. Total memory usage percentage with normal Case

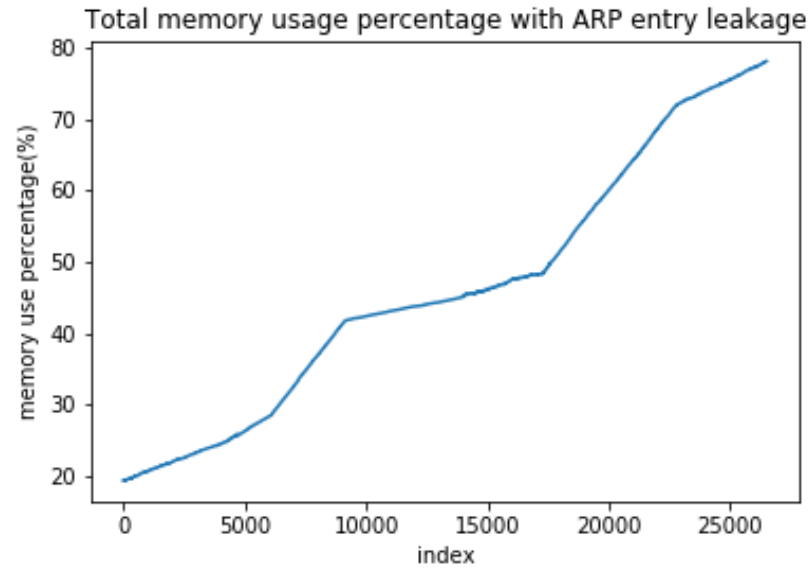

Figure 3. Total memory usage percentage with ARP Leakage

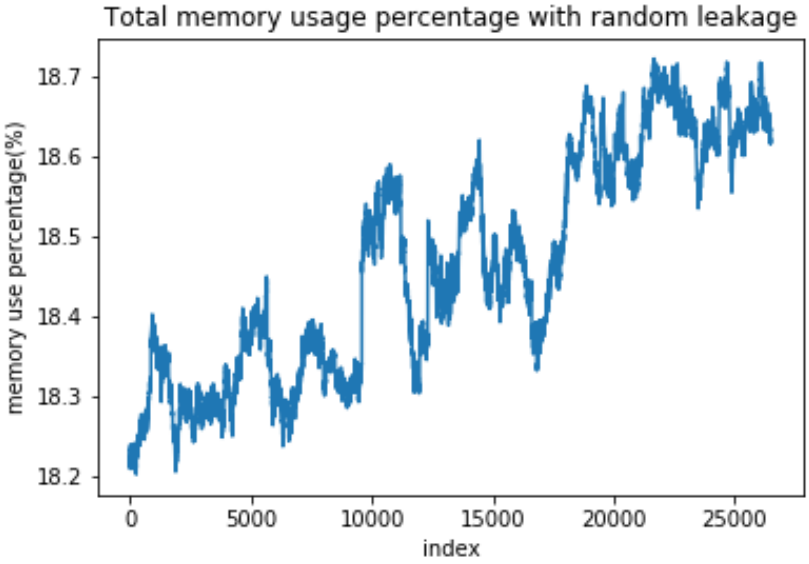

Figure 4. Total memory usage percentage with Random Leakage

The graphics of their corresponding *M*' sequences are shown below：

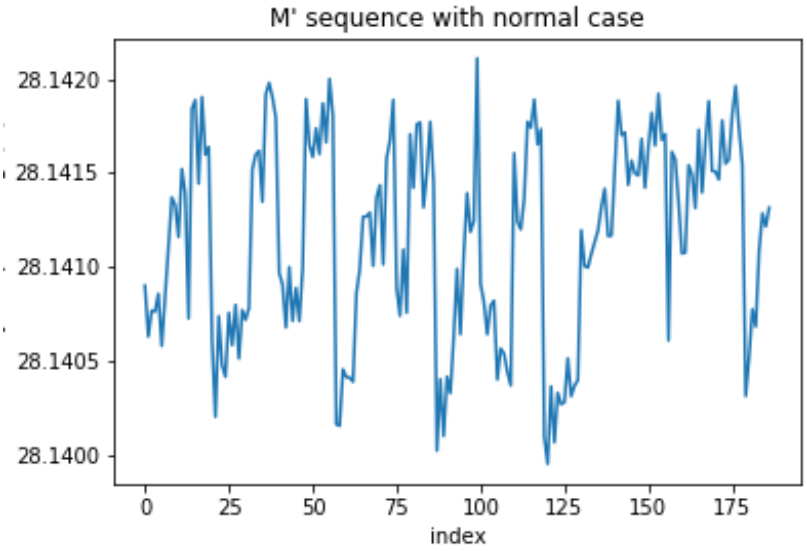

Figure 5. M’ sequence with Normal Case

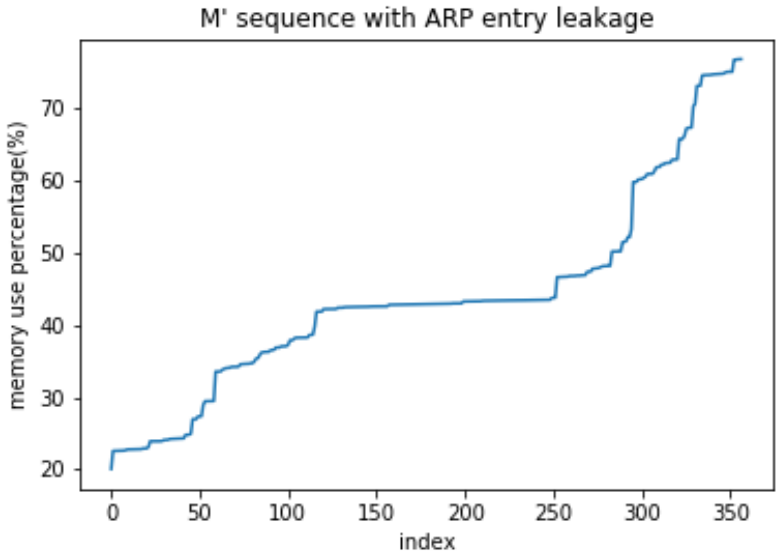

Figure 6. M’ sequence with ARP entry Leakage

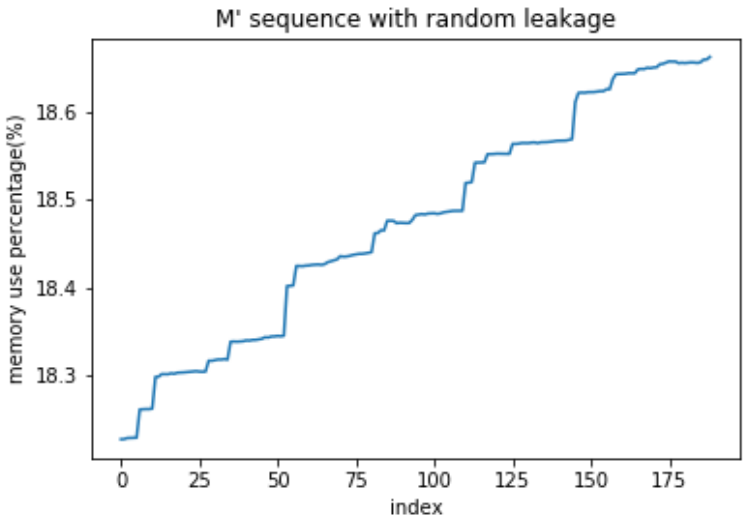

Figure 7. M' sequence with Random Leakage

Above figures clearly show that the memory utilization of normal *M*' sequence fluctuates slightly and the curve is random; the memory utilization rate of resource leak *M*' sequence changes greatly, showing an obvious monotonous upward trend; while the memory utilization rate of random leak *M*' sequence changes little and the overall trend is monotonous increasing.

Next, we simulated 500 sequences for each of the three scenarios and calculated their *M*' sequences. Related statistics including the memory increment of the sequence, the time correlation, and the average correlation coefficient (using Spearman algorithm) with the simulated memory leak sequences. They are recorded in table 1-3, but only the first 20 pieces of data are shown.

Table 1. Test Sequences with Normal case

	Memory Difference	**Corr. with index sequ.**	**Corr. with Resource leak sequ.**	**Corr. with random leak sequ.**
1	291248	0.158175215	0.155201666	0.145934015
2	301214	-0.019766941	-0.010182252	0.018031106
3	260485	0.109665269	0.092678231	0.08981839
4	291945	0.112001576	0.165478754	0.174005038
5	290986	-0.094858308	-0.113018566	-0.11931812
6	291624	-0.252379809	-0.22554809	-0.225589206
7	284432	-0.086634815	-0.049478148	-0.048171725
8	344911	0.161756547	0.154608783	0.147779058
9	251442	-0.302939227	-0.208654013	-0.210560823
10	295035	-0.060953324	-0.041223962	-0.049084516
11	276335	0.02740196	0.096420279	0.097325497
12	318737	0.37203624	0.319781804	0.313736033
13	280037	0.073193108	0.084716381	0.089715752
14	295190	-0.013274524	-0.040550019	-0.044731653

15	295725	0.007098027	0.041243151	0.046395803
16	278126	-0.011541039	0.030574502	0.024268845
17	267995	-0.09952762	-0.112263921	-0.115108548
18	298624	0.189605496	0.035233616	0.024033341
19	278243	0.143553709	0.108807238	0.119960642
20	273391	0.090146257	0.128837797	0.129857563

Table 2. Test Sequences with Resource Leak case

	Memory Difference	**Corr. with index sequ.**	**Corr. with Resource leak sequ.**	**Corr. with random leak sequ.**
1	2789276999	0.999998	0.999956	0.978997
2	2013235537	0.999989	0.999934	0.979096
3	2620491023	0.999892	0.999828	0.978786
4	2807450398	0.999999	0.999963	0.978941
5	2809413438	0.999998	0.999962	0.978945
6	1521223836	0.999991	0.999952	0.978938
7	2202391094	0.999991	0.999952	0.978986
8	972756363	0.999964	0.999926	0.979024
9	2441715706	0.999932	0.999958	0.979001
10	2838246478	0.999997	0.999961	0.978919
11	2436498700	0.999996	0.999963	0.979145
12	2007848751	0.999994	0.999957	0.978956
13	2658185467	0.999999	0.999962	0.978915
14	735720744	0.99997	0.999922	0.978947
15	1690241815	0.999993	0.999914	0.978523
16	2668271684	0.99999	0.999932	0.978854
17	968070171	0.999991	0.999956	0.9791
18	3425955833	0.999997	0.999959	0.978903
19	3530875222	0.999998	0.99997	0.979072
20	1531155291	0.999995	0.999957	0.978918

Table 3. Test Sequences with Random Leak case

	Memory Difference	**Corr. with index sequ.**	**Corr. with Resource leak sequ.**	**Corr. with random leak sequ.**
1	39257102	0.999806	0.979739374	0.97879
2	38617581	0.999781	0.979660463	0.978735
3	36811484	0.999386	0.979299968	0.978057
4	40963370	0.999746	0.97967272	0.978652
5	39622467	0.997962	0.977857124	0.976249
6	40131717	0.999658	0.979572898	0.978668
7	37434253	0.999524	0.979181516	0.978452
8	35191411	0.999252	0.979071952	0.978153
9	37450356	0.999353	0.979354171	0.978422

10	39239786	0.998732	0.978647139	0.977538
11	38587988	0.998366	0.978069213	0.977219
12	39072316	0.99942	0.978734961	0.977913
13	24657491	0.99928	0.979058344	0.978111
14	37803860	0.99986	0.979789375	0.978773
15	37887778	0.999711	0.979627497	0.978657
16	38422163	0.999802	0.979595805	0.978723
17	36231575	0.999263	0.978949737	0.977975
18	36120339	0.99965	0.97959303	0.978576
19	39842950	0.999748	0.979516662	0.978655
20	38039417	0.997934	0.977837991	0.97655

Follow the process shown in Figure 1, the memory increment of normal *M*' sequence is less than 30M, so they are all judged as normal data. For the ARP table entry memory leak *M*' sequence, its time correlation reaches 99.9% on average(greater than 0.9), the average correlation coefficient with the resource memory leak sequence set is greater than 99% and the average correlation coefficient with the random memory leak sequence set is greater than 97 %. Apart from these, the 497 out of 500 sequences show large memory increment which are beyond 30M and are judged as leaked data. For the random memory leak *M*' sequence, its time correlation reaches 99.9% on average(greater than 0.9), the average correlation coefficient with the resource memory leak sequence set is greater than 97% and the average correlation coefficient with the random memory leak sequence set is greater than 97%. Slightly lower proportion than the ARP table entry memory leak *M*' sequence, 452 out of 500 sequences are judged as leaked data. No misjudgment has been made in all scenarios which shows out 100% precision in automatic check for memory leak detection.

It should be pointed out that some memory leaks are very slow and do not exceed 30M in 3 months. At this increasing trend, only 360M of memory will be leaked in 3 years, which has little impact on the system, and it is not necessary to deal with it.

It can be seen from the above simulation experiments that the algorithm described in this article has outstanding advantages of high efficiency and high accuracy. This solution has been applied in the Seer Analyzer product of H3C Company which can detect various memory leaks and achieve excellent performance.

6. CONCLUSION

Memory leaks in network devices make some memory unavailable for subsequent use which may cause service failure or even system crash in severe cases. Commonly used manual detection methods are inefficient and error-prone, and cannot perform online and real-time detection for a large number of network devices. The memory detection scheme proposed in this paper could automatically check the device memory occupancy rate and combines the resource occupancy information of various table entries of the device to judge for memory leaks. Then estimate time to reach memory alarm threshold will be predicted and the rule engine will be used to get more detailed diagnostic information. The simulation experiments show that the scheme is computationally efficient and the precision rate is close to 100%, which solves this problem well and is of great practical significance.

This solution has been applied in the Seer Analyzer product of H3C Company, we also intend to apply it to the network devices of other companies. It should be noted that different

manufacturers have different table item sizes, so the weight values of the tables should be chosen carefully, they may have an important impact on the accuracy of the algorithm. The table item sizes even change with the software version, so it is very useful to automatically find the changes and adaptively adjust parameters.

REFERENCES

[1] John Regehr & Nathan Cooprider & Will Archer, (2006) "Efficient type and memory safety for tiny embedded systems," In: Proc of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems, San Jose, California,pp.22–22.

[2] Thomas A. Henzinger & Ranjit Jhala & Rupak Majumdar & Marco A.A.Sanvido,(2004) " Extreme Model Checking," In Verification: Theory and Practice, Lecture Notes in Computer Science 2772, Springer-Verlag, pp.332–358.

[3] Hu Yan & Gong Yu-chang & Sun Wei-feng & Zhao Zhen-xi (2008) "Hybrid Static Method for Memory Leak Detection," Journal of Chinese Computer Systems, Vol29,pp.1935–1939.

[4] Gong Yu-chang & Hu Yan & Zhang Ye & Zhao Zhen-xi,(2009) "A static memoryleak detection method for binary programs, " Journal of University of Science and Technology of China,Vol39,pp. 189–195.

[5] K. Chen & J.-B. Chen.(2007) Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In IEEE International Conference on Computers, Software & Applications (COMPSAC), pages 23–28.

[6] M. Jump & K. S. McKinley.(2007) Cork: Dynamic memory leak detection for garbage-collected languages. In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 31–38.

[7] C. Jung & S. Lee & E. Raman & S. Pande.(2014) Automated Memory Leak Detection for Production Use. In International Conference on Software Engineering (ICSE), pages 825–836.

[8] G. Xu & A. Rountev.(2013) Precise Memory Leak Detection for Java Software Using Container Profiling. ACM Transactions on Software Engineering and Methodology, 22(3):17:1–17:28.

[9] M. D. Bond & K. S. McKinley.(2009) Leak Pruning. In ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 277–288.

[10] Zohaib Latif & Kashif Sharif & Fan Li & Yu Wang.(2020) A comprehensive survey of interface protocols for software defined networks, Journal of Network and Computer Applications.

[11] R. L. Iman & W. J. Conover.(1982) A distribution-free approach to inducing rank correlation among input variables. Communications in Statistics: Simulation and Computation, 11(3):311–334.

[12] Yong H. Lee & Suk I.Yo0. (1995) A Rete-based Integration of Forward and Backward Chaining Inferences, ISIC. Page(s): 611 – 616.

[13] Sun, Y. & Wu, T.Y. & Zhao, G. & Guizani, M., (2015) Efficient rule engine for smart building systems. IEEE Trans. Comput. 64, 1658–1669. https://doi.org/10.1109/TC.2014. 2345385.

AUTHORS

Minghui Wang received his B.Sc degree and PhD degree in Mathematics from The Peking University, in 1996 and 2001, respectively. Currently, is a Senior Engineer with Institute for Artificial Intelligence, H3C Co, Ltd. His research interests include IP network communication, data mining and AI.

Jiangxuan Xie received his B.Sc degree in Statistics from The University of Hong Kong, Hong Kong, China in 2018. Currently, he is a Machine Learning Engineer for H3C, AI Research Institute. His current research interests include areas of time series analysis, data mining, big-data algorithms.

Yang Xin'an received his B.Sc. degree in electronics and information engineering from Nanjing University of Science and Technology, Nanjing, China, in 1999 . Currently, he is a Senior Engineer with Institute for Artificial Intelligence, H3C Co, Ltd. His current research interests include AI, BigData, Network communication.

Xiangqiao Ao received his B.Sc. degree in Computer Science from Zhejiang University, Hangzhou, China, in 2001. He has been working in Huawei and H3C since 2001, focusing on IP network communication. Currently he is in charge of the AI Institute of H3C. His research interests include IP network communication, AI-based IP network and Autonomous-Driving-Network.