IEICE TRANS. INF. & SYST., VOL.E93–D, NO.12 DECEMBER 2010

LETTER

Binary Oriented Vulnerability Analyzer Based on Hidden Markov Model

Hao BAI[†a)], ***Student Member***, **Chang-zhen HU**[†], **Gang ZHANG**[††], ***Nonmembers***, **Xiao-chuan JING**[††], ***Member***, ***and*** **Ning LI**[††b)], ***Student Member***

SUMMARY The letter proposes a novel binary vulnerability analyzer for executable programs that is based on the Hidden Markov Model. A vulnerability instruction library (VIL) is primarily constructed by collecting binary frames located by double precision analysis. Executable programs are then converted into structurized code sequences with the VIL. The code sequences are essentially context-sensitive, which can be modeled by Hidden Markov Model (HMM). Finally, the HMM based vulnerability analyzer is built to recognize potential vulnerabilities of executable programs. Experimental results show the proposed approach achieves lower false positive/negative rate than latest static analyzers.

key words: *executable program, binary, double precision analysis, vulnerability instruction library, Hidden Markov Model*

1. Introduction

Static vulnerability analysis has been proven effective in covering higher quantities of errors and flaws of applications prior to dynamic analysis [1]. Moreover, static analysis maintains low cost and provides high scalability, while dynamic analysis usually relates to labor-intensive techniques. Another advantage of static binary analysis lies in its ability to reason all possible execution paths of a program.

Three of the existing vulnerability analysis methods developed in recent years are the most representative. Csaba Nagy, et al. [2] make a code-scanning tool to perform static analysis of the input-related source code. Pattabiraman, et al. [3] develop detectors using source-code based static analysis from the program slice concerned with critical variables. However, for most commercial software, users can not have access to the source code. Besides, attackers usually don't attach source code with malwares. Consequently, the ability and scalability of these methods are confined within open-source applications. Hassen Saidi [4] proposes a tool-bus architecture dealing with applications for which only binary is available. While this method can only deal with malwares with small size and is not capable of analyzing large-scale programs. According to our recent investigation, few methods available can analyze binary effectively meanwhile resulting in low false positive rate.

Manuscript received July 23, 2010.
Manuscript revised September 5, 2010.

[†]The authors are with Beijing Institute of Technology, Beijing, China.
[††]The authors are with China Aerospace Engineering Consultation Center, Beijing, China.
a) E-mail: david_xiaobai@126.com
b) E-mail: quietlee0906@yahoo.com.cn

DOI: 10.1587/transinf.E93.D.3410

All the limitation discussed above motivates us to focus on disassembled executable programs. This letter proposes a binary vulnerability analyzer by combining the static and dynamic analysis. The analyzer consists of three key procedures: establishing the vulnerability instruction library (VIL) from the executable programs that have typical vulnerabilities, structurizing binary to code sequence, and training the Hidden Markov Model (HMM) to recognize vulnerabilities. Figure 1 shows the overview of the proposed approach.

In the first procedure, see the path labeled with "1" in Fig. 1, a double-precision locating method is utilized to construct the VIL. The binary segments are firstly coarse-located by the IDA Pro Disassembler [5], and then the segments are dynamically analyzed with Paimei [6] to fine-locate the binary frames. Each frame is a set of binary instructions that give rise to a certain type of vulnerabilities. The VIL is established by combining the binary frames. In the second procedure, see the path "2", binary instructions are structurized to code sequences by mapping with the VIL. Due to the intrinsic causality in the context of binary, a disassembled executable program is mapped to a sequence of context-sensitive code after this procedure. The word "context" implies that a current instruction only depends on its previous one, which accords with the Markov property. HMM is a powerful statistical method of characterizing context-sensitive data sequences. In the last procedure, see the path "3", inspired by the successful use of HMM in speech recognition [7] and action recognition [8], the HMM is applied to recognize software vulnerability ex-

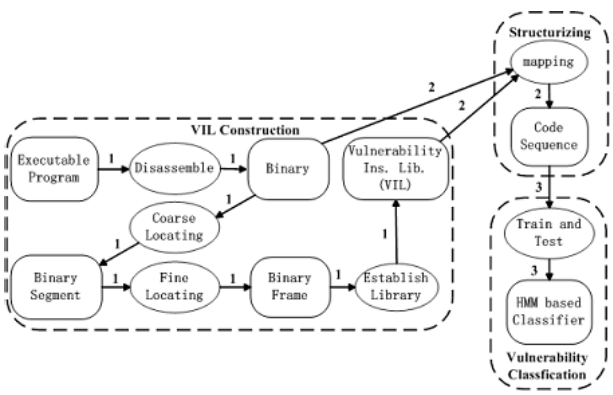

Fig. 1 The overview of the proposed approach.

isted in the code sequence.

The contributions of this letter are as follows: compared with the latest static analyzer, the proposed approach directly focuses on binary analysis and precisely identifies the vulnerability within a minimal range while improving the richness and quality of analysis; the VIL provides a metric to map the unstructured binary instructions to structured code sequences that can be processed by classical pattern recognition model; the HMM is used to model the causality of binary.

The remainder of the letter is organized as follows: Section 2 introduces the proposed approach. Section 3 presents the experimental results. Conclusions and future work are addressed at the end of the letter.

2. The Proposed Approach

2.1 VIL Construction

In this subsection, the double-precision locating method is introduced to build the VIL that contains the key binary instructions resulting in vulnerabilities. We firstly use the IDA Pro [5] to disassemble the executable and sketch the function structure graphs of the program. Every individual function is depicted in details with all the variables, registers, system calls and arguments available. The most important feature of the function graph is that, the IDA Pro can build a precise control flow of the function due to its "jump" tracing capability. With the structure graphs, IDA Pro divides the program into pieces of binary segments, each of which represents a function. Due to the comprehensive structure graphs and abundant parameters, potentially vulnerable segments are manually selected from the multiple binary segments. This stage is called "coarse locating".

To eliminate false positive warnings and figure out the intrinsic features of vulnerabilities, binary frames are then extracted from the vulnerable segments. Dynamic analysis techniques are geared towards detecting vulnerabilities in the functions. The Paimei [6] is used to locate the true vulnerabilities: binary in execution is firstly marked when the program behavior is normal, and then the executed binary is marked again as vulnerability is triggered. By the step-in debugging, the two-round marked binaries are compared to remove the "normal binary" and thus to uncover the "bad binary". Once vulnerabilities are found during the runtime, the related fragments in binary segments are located and stored as the binary frames that are the very instructions giving rise to vulnerabilities. This stage is called "fine locating".

The VIL is established by combining all the binary frames of different vulnerability types. Figure 2 shows that each binary frame consists of several instructions, thus the VIL is a collection of binary instructions that are named as keywords. The keywords are named in sequence by unique code numbers. The code numbers are 4-digit decimal, of which the first digit represents the vulnerability type and the other three digits register the corresponding keywords.

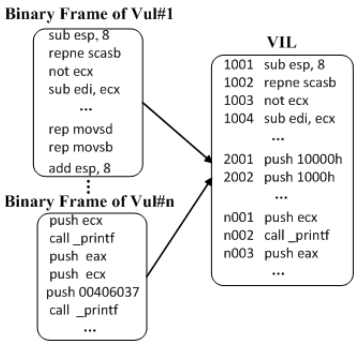

Fig. 2 The formation of VIL.

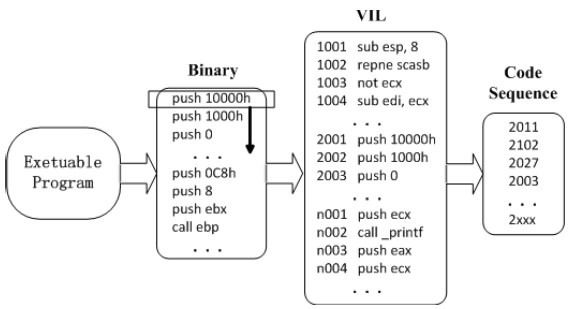

Fig. 3 The process of binary structurizing.

2.2 Structurizing the Binary

A binary is a collection of assembly instructions, which is typically unstructured data and cannot be recognized by classical pattern recognition model. This is a very negative factor for the applicability of the HMM based vulnerability classifier. To tackle this problem, the VIL is built and utilized to map the binary to structured data.

Figure 3 demonstrates the mapping process and explicates the form of data sequences after structurizing. Each binary instruction of an input disassembled program is compared with the items in the VIL one by one, and is assigned to the code number of the item that has the maximal likelihood with the input binary instruction. If the likelihood is less than a predefined threshold, which implies there is no similarity with all the items in the VIL, then the input is assign to 0. After the mapping process, input binary file transforms to a sequence of code number. There exists intrinsic logical correlation in the context of binary. Therefore, the code sequence obtained by the structurizing procedure is also context-sensitive.

2.3 HMM for Vulnerability Recognition

HMM is a powerful statistical method to model dynamic processes and has been successfully applied to speech recognition [7] and action recognition [8], where the data sequences analyzed are also context-sensitive. Inspired by its application, HMM is utilized to characterize software vulnerability existed in code sequences.

HMM is represented by a finite state process with transition between states specified by transition probability. In the proposed method, each type of software vulnerability can be well modeled by a HMM parameters set $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$. Given a vulnerability J, the corresponding λ_J is defined as follows: $\mathbf{A}_J$ is a matrix representing state transition probabilities. For instance, the $\mathbf{A}_J$ in a 3-state HMM is defined as:

$$\mathbf{A}_J = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad (1)$$

Each element, for instance a_{12} denotes the transition probability from the current state S_1 to next state S_2. Each state gives output probabilities for all keywords in the VIL. Suppose there are m keywords, then each state generates m output probabilities. Meanwhile, for a 3-state HMM, $\mathbf{B}_J$ is a $3 \times m$ matrix representing output probabilities for all states:

$$\mathbf{B}_J = \begin{pmatrix} b_1(c_1) & \dots & b_1(c_m) \\ b_2(c_1) & \dots & b_2(c_m) \\ b_3(c_1) & \dots & b_3(c_m) \end{pmatrix} \quad (2)$$

The element in each row, for instance $b_1(c_1)$ denotes the probability that the state S_1 generates the code c_1. π_J is the initial state distribution probability. The HMM can starts from any state, therefore for a 3-state HMM, $\pi_J = \{\pi_1 = 1/3, \pi_2 = 1/3, \pi_3 = 1/3\}$. To perform the vulnerability classification, the parameter set λ of each vulnerability type is trained by training sequences. The λ is iteratively optimized by the Baum-Welch algorithm [7].

Given a parameter set for vulnerability J, and testing programs $\{O_1, \dots, O_i, \dots, O_N\}$, the probability of the program O_i, generated by this model is computed using Bayes's rule as $P(O_i|\lambda_J)$. It is evaluated using the forward algorithm. Suppose $\alpha(i)$ is the forward variable representing the probability that the HMM in state i has generated the partial code sequence $\{v_1, v_2, \dots, v_t\}$ at time t, T is the length of code sequences, and v_t is the current input symbol in a symbol sequence, then the $P(O_i|\lambda_J)$ can be calculated using Eq. (3). The testing program is recognized as the vulnerability J if they generate the maximal probability. Figure 4 shows the recognition process using the proposed approach based on a 3-state HMM.

$$P(O_i|\lambda_J) = \sum_{i=1}^{k} \alpha_T(i), (O_i = \{v_1, \dots, v_t, \dots, v_T\}) \quad (3)$$

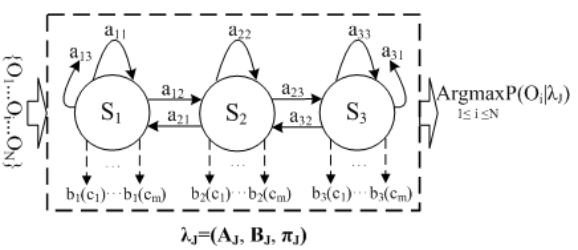

Fig. 4 Recognition process based on a 3-state HMM.

3. Experiment Evaluation

3.1 Experiment Configuration

In this section, experiment results of the proposed approach are presented on three types of vulnerabilities: stack overflow, heap overflow and format string. They are chosen because they are still common in software and are difficult to locate from the programs. According to the double-precision analysis, the VIL contains 35 keywords for these vulnerability types.

For each vulnerability type, 12 executable programs are chosen. To compute an unbiased estimate of the true classification rate, the cross-validation rule is adopted. The dataset is randomly splitted as follows: 5 programs are used as training set and the remaining 7 programs are used for testing. The system performance is evaluated by the average of 12 random splits.

We construct one HMM for one type of vulnerability. Therefore, 3 HMM parameter sets are built in our experiment. In the initialization, the HMM states are treated as uniform distribution. That is, each row element in $\mathbf{B}_J$ is assigned to $1/m$, where $m = 35$ is the size of the codebook. The transition probabilities from each state are also equal. Therefore, in the Baum-Welch training process the total number of parameters that should be estimated is 114 (9 state transitions + 3×35 output probabilities) for a 3-state HMM.

All experiments are carried out on a Pentium Dual machine with 2.20 GHz and 2 G Memory running the Windows XP system.

3.2 Experimental Results

In this subsection, the impact of the number of HMM state on the false positive/negative rate is firstly studied. Table 1 shows that the false rate decreases as the number of states N increases until $N = 7$; when N is larger than 7, the false rate does not change significantly. This means small number of states is not sufficient to characterize the context-sensitive data sequence especially for the binary. Although larger number of states also provides satisfying false rate, the computation cost becomes high due to more HMM parameters are involved. Therefore, we choose the 7-state HMM for the following experiments.

Then, the performance and coverage provided by the proposed approach and the benchmark methods that are typical static analyzer for software vulnerability are compared. Here we still use the stack overflow, heap overflow and format string as the vulnerabilities injected in the test programs

Table 1 Performance of the proposed vulnerability analyzer based on different number of HMM state (%).

Number of states	3	5	7	9	11	13
avg. False Positive	32.6	28.9	24.4	24.2	23.5	23.2
avg. False Negative	24.8	20.1	16.7	16.6	16.1	15.8

Table 2 Performance and coverage of the proposed approach and the benchmarks (%).

Vulnerability	Performance	7-state HMM	[2]	[3]	[4]
Stack Overflow	False Pos.	**21.9**	23.4	25.3	22.7
	False Neg.	**10.9**	12.3	11.6	11.2
Heap Overflow	False Pos.	30.7	31.9	**30.6**	31.1
	False Neg.	21.4	22.9	23.2	**20.8**
Format String	False Pos.	**20.7**	22.5	20.9	21.2
	False Neg.	17.7	19.2	18.5	**17.4**
avg. False Pos.		**24.4**	25.9	25.6	25.0
avg. False Neg.		16.7	18.1	17.8	**16.5**

chosen in Sect. 3.1. Since the method [2] and [3] are based on source-code analysis, source codes written in C are provided for them. The test programs are analyzed by the proposed approach and the static analyzer [2], [3] and [4] in a controlled environment.

The comparison results are reported in Table 2. When analyzing the stack overflow vulnerability of which the feature is relatively easy to recognize, the proposed approach recognizes 89.1% of the existing vulnerabilities (false negative recognition rate is 10.9%) while recognizes 21.9% of benign vulnerabilities (false positive recognition rate). In contrast, in [2], [3] and [4], over 22% of the recognized vulnerabilities are benign and more than 11% of the vulnerabilities are not recognized. Since the feature of heap overflow is not as obvious as the stack overflow, the false positive rate of the proposed approach is 30.7% while 21.4% real vulnerabilities are not recognized. However, the false positive/negative rates are still lower than most of the other three methods. Further, for the format string vulnerability, the proposed approach recognizes 20.7% benign vulnerabilities compared to the others, which have false positive rates of 20.9%–22.5% when analyzing the test program. The false negative recognition rate of the proposed approach is 17.7%, which is less than the 19.2% and 18.5% of method [2] and [3]. Finally, the average false positive rate shows that the proposed approach is more accurate in recognizing vulnerabilities and the false negative rate indicates that the proposed approach can provide a higher coverage than the source-code analysis methods (i.e., [2] and [3]).

4. Conclusion

In this letter, a novel binary vulnerability analyzer based on HMM is proposed. A vulnerability instruction library (VIL) is established by double precision analysis. Executable programs are then structurized to code sequences by mapping with the VIL. The HMM based vulnerability analyzer is built to recognize vulnerabilities. Experiments demonstrate that the proposed approach can provide higher coverage for recognizing three typical vulnerabilities than latest static analyzers. In the future work, the approach will be improved more specifically to find vulnerabilities in embedded systems. For this purpose, the VIL will be expanded and updated, and the HMM parameters will be reconfigured to meet the special needs of embedded environment.

Acknowledgement

This work was supported by National High Technology Research and Development Program of China (863 Program) (Grant No. 2009AA01Z433).

References

[1] U. Bayer, E. Kirda, and C. Kruegel, "Improving the efficiency of dynamic Malware analysis," Proc. 2010 ACM Symposium On Applied Computing, pp.1871–1878, March 2010.
[2] C. Nagy and S. Mancoridis, "Static security analysis based on input-related software faults," Proc. 2009 European Conference on Software Maintenance and Reengineering, pp.37–46, March 2009.
[3] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer, "Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach," IEEE Trans. Dependable and Secure Computing, vol.11, June 2009.
[4] H. Saidi, "Logical foundation for static analysis: Application to binary static analysis for Security," ACM SIGAda Ada Letters, vol.28, no.1, pp.96–102, April 2008.
[5] The IDA Pro Dissasember and Debugger, http://www.hex-rays.com/idapro/
[6] Paimei Reverse Engineering Framework, http://pedram.redhive.com/PyDbg/
[7] X. Huang, A. Acero and H.-W. Hon, Spoken Language Processing-A guide to theory algorithms and system development, pp.375–411, Prentice Hall Books, New Jersey, 2001.
[8] N. Li and D. Xu, "Action recognition using weighted three-state Hidden Markov Model," Proc. 9th International Conference on Signal Processing (ICSP2008), pp.1428–1431, Oct. 2008.