LETTER *Special Section on Log Data Usage Technology and Office Information Systems*

A Cross-Platform Study on Emerging Malicious Programs Targeting IoT Devices

Tao BAN[†a)], ***Nonmember***, **Ryoichi ISAWA**[†], ***Member***, **Shin-Ying HUANG**[††], ***Nonmember***, **Katsunari YOSHIOKA**[†,†††], ***and*** **Daisuke INOUE**[†], ***Members***

SUMMARY Along with the proliferation of IoT (Internet of Things) devices, cyberattacks towards them are on the rise. In this paper, aiming at efficient precaution and mitigation of emerging IoT cyberthreats, we present a multimodal study on applying machine learning methods to characterize malicious programs which target multiple IoT platforms. Experiments show that opcode sequences obtained from static analysis and API sequences obtained by dynamic analysis provide sufficient discriminant information such that IoT malware can be classified with near optimal accuracy. Automated and accelerated identification and mitigation of new IoT cyberthreats can be enabled based on the findings reported in this study.

key words: *IoT security, IoT malware, malware analysis, malware classification*

1. Introduction

Aiming at infecting IoT devices across different CPU architectures, IoT malware is usually compiled on and launched against multiple platforms. This poses new challenges to the mitigation and prevention of these emerging cyberthreats. In this paper, we present a new approach to capture, analyze, and classify malicious programs across multiple IoT platforms.

To collect the most recent IoT malware targeting different IoT devices, we deploy the IoTPOT [1] in our environment. IoTPOT constitutes virtualized cross-platform Linux systems deliberately set up with vulnerabilities that invites penetration. It records all information related to the penetration by collecting the malicious programs, command traces, and Internet communications during the attack. Then, based on these evidences, we perform the following multilateral analysis of the malicious programs across three major CPU architectures, namely, ARM, MIPS, and MIPSEL.

In the first step, we use an entropy-based method to identify obfuscated malware samples from non-packed ones. Then, static and dynamic analyses are employed for profiling the behavioral characteristics of the samples: For static analysis, IDA Pro (a Linux-hosted multiprocessor disassembler) is applied to the samples to attain their assembly code and store it in the form of opcode sequences. For dynamic analysis, Application Programming Interface (API) call sequences are collected with the *strace* command at runtime in virtualized environments.

To yield numerical vectors that could serve as inputs to the analytical tools, the sequences in log files are coded as *n*-grams—a contiguous sequence of *n* items from the instruction sequence [2]. Then, vectorized representations of the malware programs are examined by *t*-distributed Stochastic Neighbor Embedding (*t*-SNE) [3], which provides a visual hint on the interpretability of different analysis methods. Finally, pattern classification is applied to the vector representation of the malicious programs for quantitative evaluation.

1.1 Data

We use the same dataset introduced in [4] for the analysis. It consists of 9,085 IoT malware samples collected from July 8, 2017 to January 20, 2018 using IoTPOT. To label them, the SHA 256 hash values of malware samples are submitted to VirusTotal, where they are checked against more than 60 different virus scanners. Then, the label of a sample is determined by majority voting. We confined the analysis to the three major malware types in the dataset, i.e., Bashlite, Mirai, and Tsunami, which consist of 2,577 samples. Table 1 shows the distribution of malware among different Instruction Set Architectures (ISA) and malware categories. A sample is excluded from the analysis if its API/opcode sequence is shorter than 10, indicating highly possibility of a cracked file. This makes the dataset slightly smaller than that in [4].

Table 1 IoT malware dataset overview.

ISA	Sample size	Bashlight	Mirai	Tsunami
ARM	729	619	94	16
MIPS	980	657	310	13
MIPSEL	868	648	210	10

2. Analysis

2.1 Preprocessing

Packing is among the most popular obfuscation techniques to impede virus scanners from effectively detecting malware. To confirm whether a malware program is packed or not, we employ entropy analysis proposed by Lyda et al. [5]

Manuscript received November 12, 2018.
Manuscript revised April 8, 2019.
Manuscript publicized June 21, 2019.

[†]The authors are with the National Institute of Information and Communications Technology, Koganei-shi, 184–8795 Japan.
[††]The author is with the Institute for Information Industry, Taipei, Taiwan.
[†††]The author is with Yokohama National University, Yokohama-shi, 240–8501 Japan.
a) E-mail: bantao@nict.go.jp

DOI: 10.1587/transinf.2018OFL0007

to the collected corpus. The analysis is based on the fact that the entropy of a binary program, measured upon the distribution of bytes, tends to be significantly raised by the crypto-encoding process during packing. To measure the entropy of a binary program, it is first divided into successive blocks of fixed size with the entropy of a block s_k calculated as

$$H(s_k) = -\sum_{i=0}^{255} p(i)\log_2(p(i)), \tag{1}$$

where $p(i)$ is the probability of byte value i appeared in s_k. Given predefined threshold values θ_{ave} and $\theta_{\max}$, a binary is determined to be packed if its averaged entropy score, H_{ave}, and the maximum score among all blocks, $H_{\max}$, satisfy

$$H_{\text{ave}} > \theta_{\text{ave}} \wedge H_{\max} > \theta_{\max}. \tag{2}$$

In the experiment, we set $\theta_{\text{ave}} = 6.0$ and $\theta_{\max} = 6.796618$, based on our previous experience on Windows malware analysis. Entropy analysis reveals that only a small portion of the IoT malware samples (less than 3%) appear to be packed. Packed samples are excluded from the analysis due to the lack of static analysis tools against them.

2.2 Feature Engineering

For the non-packed samples, feature engineering is pursued in the following steps. First, IDA Pro, a Linux-hosted multi-processor disassembler, is applied to the samples to attain their assembly code. In the snapshot of ARM assembly code shown in Fig. 1, each line contains an opcode (MOV, ADD, BL, etc.) and the operands they act on. In the second step, we extract the sequence of opcode which carries the most essential information of malware behavior. For the snapshot in Fig. 1, we obtain an opcode sequence like {`MOV MOV ADD BL CMP BLT ...`}. Finally, following the n-gram convention, we code each of the log files as a list of n-grams. The 3-gram representation of the opcode sequence ion Fig. 1 is {`[MOV MOV ADD], [MOV ADD BL], [ADD BL CMP], [BL CMP BLT] ...`}. Following the same procedure, we also generate API features for each malicious program from the API-call trace logged by the *strace* command during its execution in a virtualized sandbox. Compared with the Vector Space Model approach adopted in [4], the n-gram representation preserves the order information in the sequence and thus may lead to superior generalization performance.

```
1  loc_162A8       ; CODE XREF: fseeko64+8C^Xj
2          MOV     R2, R6
3          MOV     R0, R5
4          ADD     R1, SP, #0x2C+var_1C
5          BL      __stdio_seek
6          CMP     R0, #0
7          BLT     loc_162FC
8          LDR     R3, [R5]
9          LDR     R2, [R5,#8]
10         MOV     R0, #0
11         BIC     R3, R3, #0x47
12         MOV     R1, R3,ASR#8
13         MOV     R4, R0
14         STRB    R1, [R5,#1]
15         STR     R2, [R5,#0x1C]
16         STRB    R0, [R5,#2]
17         STRB    R3, [R5]
18         STR     R2, [R5,#0x10]
19         STR     R2, [R5,#0x14]
20         STR     R2, [R5,#0x18]
21         STR     R0, [R5,#0x2C]
22         B       loc_16300
23 ;
```

Fig. 1 Assembly code output by IDA Pro on ARM.

In the n-gram representation, each sample is coded as a list of n-grams, which tends to reside in very high dimensional features space. Quite often, the extraordinary high dimension of the data prevents the adoption of common algorithms for the analysis. Moreover, the Euclidean distance in such a high-dimensional space is likely to suffer from deteriorated representability in certain situations. To circumvent this problem, we propose to use the Jaccard distance [6] to measure the dissimilarity between the list of sequences. More specifically, Jaccard distance, which measures dissimilarity between two finite item sets X_i and X_j, can be obtained by dividing the difference of the sizes of the union and the intersection of two sets by the size of the union,

$$d_J(X_i, X_j) = \frac{|X_i \cup X_j| - |X_i \cap X_j|}{|X_i \cup X_j|}. \tag{3}$$

With this distance function defined upon the n-gram representation of the opcode/API traces, the distance-based analysis described in the following sections are applied.

2.3 Visualization

To visualize the sample distribution of a particular ISA, the $N \times N$ distance matrix, where N is the number of malware samples under analysis, is mapped to 2-D points using t-SNE [3]. t-SNE implements the mapping in such a way that similar samples are modeled by nearby points and dissimilar samples are modeled by distant points with high probability. Figure 2 (a) shows the layout of malware samples on MIPS using opcode feature. Samples from different malware classes are shown with different markers. Samples from the same category forms clusters because inter-class distances are closer than intra-class distances. The separability of samples from different categories in this 2D space suggests that the n-gram features carries essential information for classifying the malware samples.

Figures 2 (b) and (c) shows the layout of malware samples on ARM using opcode features and API features, respectively. A gray line between Figs. 2 (b) and (c) connects two points that correspond to the same Mirai sample. This comparison reveals that in the API feature space, the classes interlace with each other more than in the opcode feature space, which may lead to degenerated performance in classification.

2.4 Classification

Using the same features, we build classifiers that can predict the classes of new malware samples. Two classifiers, namely, SVM [7] and k Nearest Neighbor (kNN) [8] are chosen for their good generality and generalization ability. The results using opcode features and API features are reported in Table 2, where the micro averages over 10 runs are shown. For each run, we use 70% of data for training and the rest of the data for testing.

As can be seen from the table, with the Jaccard distance, both SVM and kNN give nearly optimal results on all

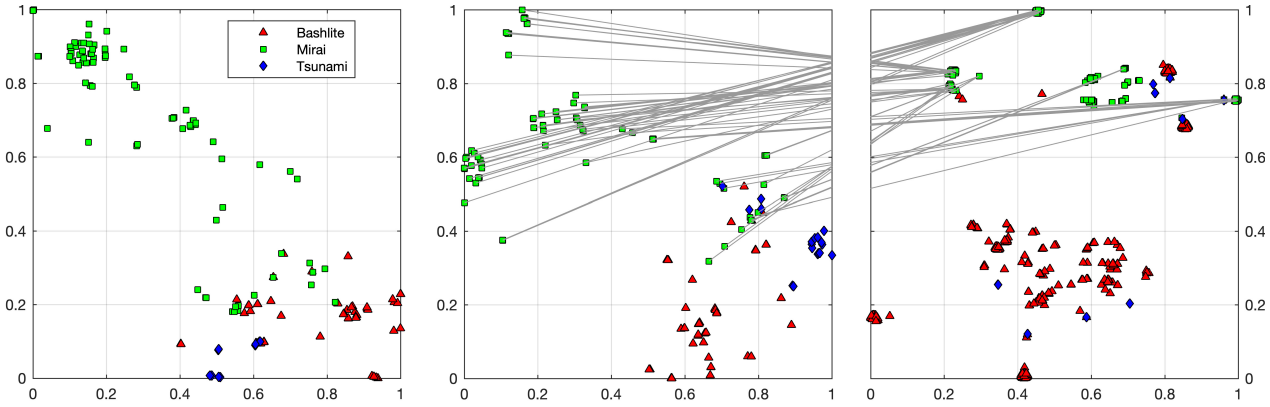

Fig. 2 2-d visualization using *t*-SNE. (a) Opcode features on MIPS; (b) Opcode features on ARM; (c) API features on ARM.

Table 2 IoT malware classification result (micro-average).

ISA	Feature Set	Classifier	Accuracy(%)	Precision (%)	Recall (%)	FPR (%)	AUC	G-mean (%)	F1-measure(%)
ARM	API	SVM	98.90	98.92	98.90	5.68	0.9661	96.53	98.85
		*k*NN	99.31	99.36	99.31	0.90	0.9920	99.17	99.27
	Opcode	SVM	**100.00**	**100.00**	**100.00**	**0.00**	**1.0000**	**100.00**	**100.00**
		*k*NN	99.58	99.59	99.58	1.14	0.9922	99.18	99.52
MIPS	API	SVM	99.46	99.54	99.46	0.38	0.9954	99.53	99.47
		*k*NN	99.55	99.51	99.55	0.35	0.9960	99.54	99.50
	Opcode	SVM	**99.73**	**99.76**	**99.73**	**0.21**	**0.9976**	**99.76**	**99.74**
		*k*NN	99.66	99.66	99.66	0.50	0.9958	99.56	99.63
MIPSEL	API	SVM	99.69	99.69	99.69	0.57	0.9956	99.55	99.69
		*k*NN	**99.77**	**99.81**	**99.77**	**0.35**	**0.9971**	**99.71**	**99.78**
	Opcode	SVM	99.46	99.47	99.46	0.98	0.9924	99.24	99.46
		*k*NN	99.73	99.73	99.73	0.51	0.9961	99.61	99.72

the examined ISAs with static and dynamic features. This indicates Jaccard distance has a strong potential to measure behavioural similarity between the instruction sequences. Opcode features outperforms API features on ARM and MIPS, while API features win on MIPS with a small margin. In particular, on ARM, SVM yields a perfect classification result using opcode features.

3. Conclusion

The experiment results reveal that IoT malware is yet not armed with complicated obfuscating techniques at the current stage. First, the result of entropy analysis shows that less than 3% of the malware programs use common obfuscation techniques like packing. Second, near optimal classification rates using opcode and API features indicate that both static analysis and dynamic analysis carry deterministic discriminant information about malware category. This further implies that no sophisticated behavior-obfuscation techniques such as reordering API call sequences, injecting bogus API calls, or alternation between semantically equivalent API calls are implemented by the malware. For the sake of the ease to perform statistic analysis than dynamic analysis, it might be rational to say that statistic analysis is preferable than dynamic analysis for recent IoT malware.

References

[1] Y.M.P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: Analysing the rise of IoT compromises," WOOT 15, USENIX Association, 2015.
[2] A.Z. Broder, S.C. Glassman, M.S. Manasse, and G. Zweig, "Syntactic clustering of the Web," Computer Networks and ISDN Systems, vol.29, no.8-13, pp.1157–1166, 1997.
[3] L. van der Maaten and G.E. Hinton, "Visualizing high-dimensional data using *t*-SNE," J. Machine Learning Research, vol.9, pp.2579–2605, 2008.
[4] T. Ban, R. Isawa, K. Yoshioka, and D. Inoue, "A cross-platform study on IoT malware," 2018 Eleventh International Conference on Mobile Computing and Ubiquitous Networking, 2018.
[5] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," IEEE Security Privacy, vol.5, no.2, pp.40–45, 2007.
[6] P.N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining, First Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
[7] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," ACM Trans. Intelligent Systems and Technology, vol.2, no.3, Article No.27, 2011.
[8] S.K. Pal and P. Mitra, Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing, Chapman and Hall/CRC, New York, 2004.