Analysis of Attacker Behavior in Compromised Hosts During Command and Control

Farhan Sadique*, Shamik Sengupta†
Dept. of Computer Science and Engineering
University of Nevada, Reno, NV, USA
fsadique@nevada.unr.edu*, ssengupta@unr.edu†

ICC 2021 - IEEE International Conference on Communications | 978-1-7281-7122-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICC42927.2021.9500859

***Abstract*—Traditional reactive approach of blacklisting botnets fails to adapt to the rapidly evolving landscape of cyberattacks. An automated and proactive approach to detect and block botnet hosts will immensely benefit the industry. Behavioral analysis of botnet is shown to be effective against a wide variety of attack types. Current works, however, focus solely on analyzing network traffic from and to the bots. In this work we take a different approach of analyzing the chain of commands input by attackers in a compromised host. We have deployed several honeypots to simulate Linux shells and allowed attackers access to the shells to collect a large dataset of commands. We have further developed an automated mechanism to analyze these data. For the automation we have developed a system called CYbersecurity information Exchange with Privacy (CYBEX-P). Finally, we have done a sequential analysis on the dataset to show that we can successfully predict attacker behavior from the shell commands without analyzing network traffic like previous works.**

***Index Terms*—honeypot, commands, cowrie, botnet, CYBEX-P**

I. Introduction

A key component in cyberattacks is a bot – a compromised host or a botnet [1], [2] – a set of such hosts. A host becomes a bot when an attacker gains access to its shell. To do that the attacker can plant a malware (e.g. mirai [3], Torpig [4], Conficker [5] etc.) or directly perform brute-force attack to crack the password. Attackers use botnet to distribute malware, perform DDoS attacks, host phishing websites, perform brute-force attacks etc. Bots make up a large portion of the cybersecurity market. So it is desirable to detect and block a bot in any network.

A. Motivation & Challenges

Currently the most popular defense against botnet is manually blacklisting their IP addresses. However, numerous hosts are compromised every day. At the same time, many bots become benign as the owner regains control of the host. So, it is impossible to list all their IPs. Moreover, blacklisting is a reactive approach. An IP shows up in a blacklist only after it has done some harm.

As a result, the industry would greatly benefit from a proactive defense mechanism against botnets. An intelligent system should detect a zero-day bot from its behavior not the IP. If a bot is detected in an early phase of the kill chain, it cannot do any harm to anybody else.

This research is supported by the National Science Foundation (NSF), USA, Award #1739032.

There is extensive research on behavioral analysis of botnet. Intrusion detection systems (IDS) [6] use network signatures to detect bots. While they work really well for known patterns they cannot adapt to the new attacks. It also takes a long time to detect an attack pattern, analyze it and create its signature before adding it to an IDS.

Another popular approach is using anomalies in network traffic [7]–[9] to detect bots. Some works take it further to detect anomalies in DNS traffic [10]–[13]. However, network traffic data is cumbersome to work with and often encrypted.

B. Contribution

In this work, we take a novel approach of analyzing the chain of commands input by attackers in a compromised hosts, rather than using network traffic data. A safe but effective approach of documenting commands input in bots is to use honeypots [14].

We have allowed attackers access to a simulated shell of the Cowrie [15] honeypot. After gaining access to the honeypot's shell, attackers input several commands. Attackers try to gain information about the system or gain access to a privileged shell. After trying out several commands the attacker logs out of the shell. We call this sequence of commands input during one particular login session a command chain.

We then create a model from these command chains based on their frequency occurrence. Finally, we try to predict the next command to be input by the attacker from our dataset. We have achieved an accuracy of 94-99% in predicting the next command input by the attacker. This shows that attacker behavior is predictable using command chains only. To the best of our knowledge, no previous work has looked into this behavior of botnets and our approach is a novel one.

II. Related Work

Extensive work was done on botnet behavior. Some of them used dataset similar to ours. Shrivastava et al. [16] captured different attacks on IoT devices using the Cowrie [15] honeypot. They employed various machine learning algorithms, including Naive Bayes, Random Forest and Support Vector Machine (SVM) to classify these attack. Surnin et al. studied different techniques for SSH detection and proposed a methodology for probabilistic estimation of honeypot detection. Dowling et al. [17] used reinforcement learning to achieve a similar goal of concealing honeypot functionality.

978-1-7281-7122-7/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

Several works studied the Mirai [3] botnet using Cowrie honeypot. Kambourakis et al. [18] provided details on how the Mirai botnet malware spreads, and discussed defensive strategies. Said et al. [19] discussed techniques to classify binary samples as Mirai based on their syntactic and behavioral properties. Lingenfelter et al. [20] analyzed variation among IoT Botnets using medium interaction honeypots. However, none of these works used our approach of analyzing the shell commands during the command and control stage. Our work differs by proposing an automated methodology to analyze their behavior by analyzing the shell commands input during the command and control stage.

Several previous works modeled the attacker behavior in IoT network. Deshmukh et al. [21] proposes Fusion Hidden Markov Model (FHMM) for modeling attacker behavior. FHMM is more noise resistant and provides faster performance than Deep Recurrent Neural Network (DeepRNN) with comparative accuracy.

Finally, Rade et al. [22] modeled honeypot data using semisupervised Markov Chains and Hidden Markov Models (HMM). They also explored Long Short-Term Memory (LSTM) for attack sequence modeling. They concluded that LSTM provides better accuracy than HMM. As shown in section VII, we have used normalized Levenshtein distance [23] to get a faster performance and a better accuracy. We also propose an automation framework to better capture the evolving nature of the threat landscape.

III. System Architecture

In this work, we have developed an automated framework to analyze and fingerprint attacker behavior in any compromised host. Our system uses CYbersecurity information Exchange with Privacy (CYBEX-P) [24] infrastructure as a service (IaaS). CYBEX-P is a cloud based platform for organizations to share heterogeneous cyberthreat data. CYBEX-P accepts all kinds of human or machine generated data including firewall logs, emails, malware signatures etc. Our system has 7 modules – (1) Honeypots, (2) Frontend, (3) Input, (4) API, (5) Archive, (6) Analytics, and (7) Report. These modules share various components as shown in Fig. 1.

1) Honeypots: We have setup 4 instances of the Cowrie honeypot all around the world. The locations are – Amsterdam, London, Mumbai and San Jose. All of them login SSH login attempts and corresponding commands input upon successful login. We have a diverse choice of four honeypots in four locations for better analysis and correlation.

2) Frontend Module: The frontend module (9, 10 in Fig. 1) is a webapp for users to interact with CYBEX-P. This module allows users – (1) to register with and login to CYBEX-P, (2) to configure the data sources, (3) to view the data, (4) to generate reports, and (5) to visualize the data.

3) Input module: The input module (1, 2, 3, 4, 10 in Fig. 1) handles all the data incoming to CYBEX-P. Machine data is automatically sent via a connector (2) to the collector (3) using real time websockets. Afterwards, the collector posts the raw data to our API (4) endpoint. To ensure privacy,

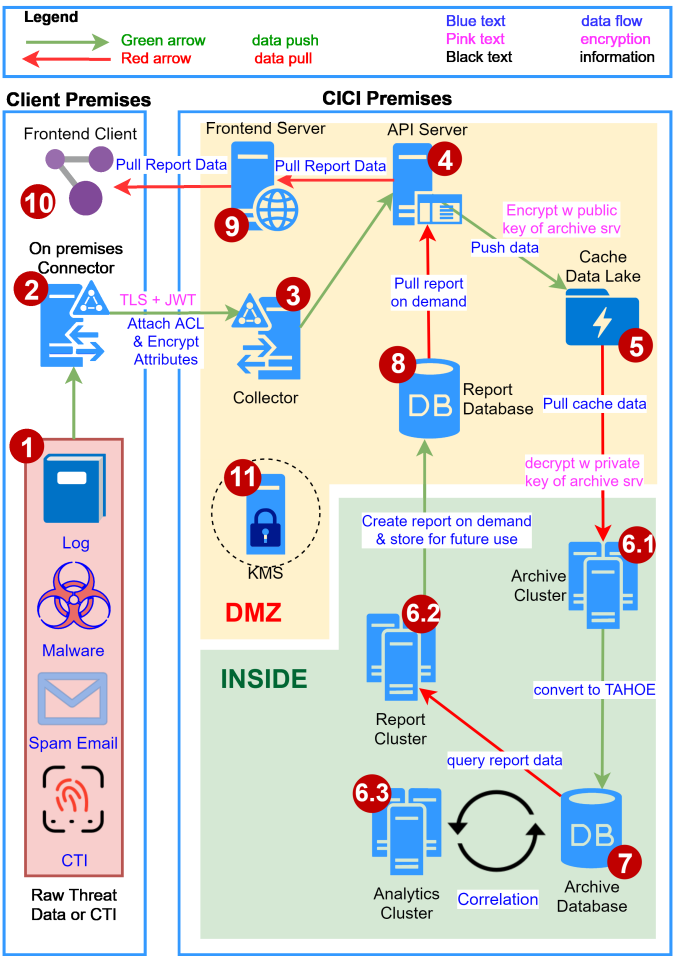

Fig. 1. System architecture of CYBEX-P along with the Data Flow.

it uses the transport layer security (TLS) protocol [25] during collection and posting.

4) API module: The API module (4, 5 in Fig. 1) consists of the API server (4) and the cache data lake (5). It acts as the gateway for all data into and out of CYBEX-P. It serves two primary purposes:

1) The input module (subsection III-3) puts raw data into the system using the API.
2) The report module (subsection III-7) sends reports back to users using the API.

5) Archive module: The archive module (6.1, 7 in Fig. 1) resides in the archive cluster and consists primarily of a set of parsing scripts. As mentioned earlier, the cache data lake (5) is encrypted with the public key of the archive server (6.1). The archive server – (1) gets the encrypted data from the cache data lake (2) decrypts the data using own private key (3) parses the data into TAHOE, and (4) stores the data in the archive DB (7).

6) Analytics module: The analytics module (6.3, 7 in Fig. 1) works on the archived data to transform, enrich, analyze or correlate them. It has various sub modules, some of which described here.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

a) Filter sub-module: An analytics filter parses a specific event from raw user data. Multiple filters can act on the same raw data and vice-versa. For example, one filter can extract a *file download event* from a piece of data while another filter can extract a *DNS query event* from the same data.

b) Sequential Analysis sub-module: This is a specialized sub-module that performs sequential analysis of the data based on the timestamp. It also correlates events in a session. A session is the time duration when one user is logged in.

7) Report Module: Users use the report module (4, 5, 6.2, 7, 8, 9, 10 in Fig. 1) to generate and view reports. They request reports via the frontend client (10,9). The API (4) stores the requests in the cache data lake (5). The report server (6.2) handles those requests by getting relevant data from the archive DB (7) and aggregating them into reports. It then stores the reports in the report DB (8). Users can access the reports on demand.

IV. Dataset

A. Data Source – Cowrie Honeypot

Cowrie [15] is a medium to high interaction SSH and Telnet honeypot designed to log brute force attacks and the shell interaction performed by the attacker. In medium interaction mode (shell) it emulates a UNIX system in Python, in high interaction mode (proxy) it functions as an SSH and telnet proxy to observe attacker behavior to another system. In this work we use Cowrie in the medium interaction mode.

In our setup, cowrie only allows SSH logins into our honeypot. An attacker can login to the system using any username and password combination. Cowrie logs all the interactions including the source IP address of the attacker, the SSH parameters, the downloaded files and the commands input while the attacker is logged in.

B. Cowrie Data as Events

```
{
    "sensor" : "ssh-peavine",
    "msg" : "CMD: ",
    "eventid" : "cowrie.command.input",
    "timestamp" : "2019-08-29T23:29:10.348959Z",
    "session" : "fd0c4206bf19",
    "command" : "%{input[0]}",
    "src_ip" : "165.227.91.185"
}
```

Fig. 2. Attributes of a Cowrie Event.

Cowrie structures collected data into events. Fig. 2 shows the attributes of a Cowrie event. Here

- *eventid:* Describes the type of the data, in the Fig. 2, `cowrie.command.input` means this event contains a command input by the attacker in the shell.
- *timestamp:* The time when the event was recorded by the honeypot.
- *msg:* The log message generated by Cowrie.
- *src_ip:* Source IP of the attacker.
- *session:* Cowrie maintains a `session` id to group events logged during one login session by an attacker.
- *sensor:* Host-name of the honeypot server.
- *command:* This field is present only for eventid `cowrie.command.input` and contains the exact command input by the attacker.

Cowrie generates about 20 different 'eventid's. Many of these events are related to the SSH session, key exchange and data logging and do not carry valuable data. For this work, we are primarily interested in the 'eventid' `cowrie.command.input`, which stores the exact command input by an attacker into the shell.

C. Data Statistics

For this work we have collected more than 18 million Cowrie events, spanning about 700 thousand sessions, from 27 August 2019 to 8 September 2019. Although we have 4 honeypot installations our initial analysis shows that all of them faced the same type of attack. The datasets in all the honeypots were dominated by the Mirai [18] botnet. This is because Cowrie honeypot emulates an IoT device and during 2019 Mirai was dominating IoT devices.

We have grouped all the events from all the honeypots together in one dataset because they were generated by the same malware (Mirai). Our initial analysis showed that there was no qualitative difference between the data generated from the different honeypots. So we have not done any comparative analysis in this work, rather analyzed all the data together.

V. Analysis Methodology

As mentioned in section III, We have used CYBEX-P [24] to automate the entire procedure of data collection to data analysis for this work. In other words we have used CYBEX-P infrastructure as a service (IaaS) here. This work is closely coupled with the development of CYBEX-P.

Along with CYBEX-P, we have further developed TAHOE a graph-based cyberthreat language (CTL). TAHOE offers several advantages over traditional CTLs. Firstly, TAHOE can store all types of structured data. Secondly, queries in TAHOE format are faster than in other CTLs. Finally, TAHOE scales well for analyzing data - a major limitation of other CTLs. Furthermore, it structures threat data as JavaScript Object Notation (JSON) which is very versatile compared to SQL.

1) Data Generation: Each Cowrie honeypot (2 in Fig. 1) simulates a generic IoT device. They generate data in the format of Fig. 2. The honeypots log these data pieces into a file in respective server. We call each such log message a raw document.

2) Data Input: Each of our honeypot installations have a connector agent (2 in Fig. 1). The connector is basically a script that reads the raw data from log files and sends them to the CYBEX-P collector (3) via a websocket. The data in transport are encrypted via TLS.

3) Data Collection: The collector then posts the data to the API (4). The API encrypts the data with the public key of the archive cluster (6.1) and stores the encrypted data in the cache data lake (5).

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

```
{
    "itype" : "raw",
    "sub_type" : "x-unr-honeypot",
    "orgid" : "a441b15fe9a3cf56661190a0b93b9dec...",
    "data" : {
        "sensor" : "ssh-peavine",
        "msg" : "CMD: ",
        "eventid" : "cowrie.command.input",
        "timestamp" : "2019-08-29T23:29:10.348959Z",
        "session" : "fd0c4206bf19",
        "command" : "%{input[0]}",
        "src_ip" : "165.227.91.185"
    },
    "_hash" : "ce1125ef568028d65ad444dd9a6dacf6...",
}
```


Fig. 3. A TAHOE `raw` document.


```
{
    "itype" : "event",
    "sub_type" : "shell_command",
    "orgid" : "a441b15fe9a3cf56661190a0b93b9de...",
    "timestamp" : 1567121350.921,
    "malicious" : true,
    "data" : {
        "attacker" : [{
            "ipv4" : ["165.227.91.185"],
            "geoip" : [{
                 "timezone" : ["America/New_York"],
                 "city_name" : ["New York"],
                 "country_name" : ["United States"]
            }]
        }],
        "shell_command" : ["%{input[0]}"]
    },
    "_hash" : "36bebc93498c47eeb05a38162d995520..."
}
```


Fig. 4. A TAHOE `event`.

4) Data Archiving: The archive cluster (6.1), then pulls the data from the cache data lake (5), decrypts the data using its private key, converts the cowrie events into TAHOE `raw` format and stores them in the archive database (7). TAHOE `raw` basically puts a wrapper around the Cowrie event. Fig. 3 shows the structure of a TAHOE `raw` document.

5) Data Analytics: The analytics cluster (6.3) parses the TAHOE `raw` data into TAHOE `events` and `sessions`. It reads the `raw` data from the archive database (7), processes the data and writes the results back in the archive database. Fig. 4 shows structure of a TAHOE `event`.

6) Command Sequence: A TAHOE `session` is a special document that connects related events. Note that the event itself contains no information about any other event. However, a `session` contains references to all events that were generated during one Cowrie login session. Fig. 5 highlights how a TAHOE `session` groups related command `events`. Here, `events` $1, 2$ and 3 were all recorded during one login session of an attacker. TAHOE maintains this relationship by connecting them to the same `session` node.

Also, Fig. 4 shows that a TAHOE `event` has a field called timestamp that stores the exact time when the command was input into the shell. Using these information we can sort the `events` in a `session` by their timestamps to get a sequence of commands input by an attacker. Table I shows the 5 most popular command sequences along with their frequency in our dataset.

Fig. 5. A TAHOE `session` connects related `events`.

VI. Experimental Verification

To verify that our system works, we have done predictive analysis on command chains. Specifically, we have used Levenshtein distance [23] to predict the next command input by a bot. The high prediction accuracy of our model, as shown in subsection VII shows that botnet behavior is predictable and our system can be used to analyze that.

A. Training a Model

As explained in subsection V-6 we form chains out of commands input into the Cowrie shell. An example of a command chain is `shell - /mnt/.ptmx - system - chmod - /boot/.ptmx - cat`. Here, the attacker inputs 6 commands, in this particular order, before logging out. Let's replace these commands with integers in this example. Then the chain becomes `1-2-3-4-5-6`.

Assume, we have N unique command chains in our dataset for M sessions. We further divide these chains into sub-chains of lengths 3 to 11. For example, from the chain `1-2-3-4-5-6` we can get 3 sub-chains of length 4 – `1-2-3-4, 2-3-4-5, 3-4-5-6` and 2 sub-chains of length 5 – `1-2-3-4-5, 2-3-4-5-6`.

We have split the chains into sub-chains for two reasons – (1) to obtain more data out of our dataset, (2) because our initial observation revealed that sub-chains of lengths 3 to 11 are common in many command chains in our dataset. However, this is purely a design choice and can be easily modified later.

We have then created a model out of these command-chains by storing them in a nested hash table. The keys are the sub-chains except the last command. The corresponding value is a second dictionary. The keys of the second dictionary are the final command and values are the frequency of the entire sequence. Fig. 6 shows the training methodology for sub-chains of length 5. This can easily be extended for other sub-chains of lengths 3 to 11. The methodology is explained below:

1) Fetch all `sessions` from the database.
2) For each `session`, fetch all of its related `events`.
3) Select only `events` with sub_type='shell_command'.
4) Store the `events` in an array.
5) Sort the array by the timestamp of the `events`.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

TABLE I
TOP 5 MOST FREQUENT COMMAND SEQUENCE OF LENGTH 6.

Command Sequence	Frquency
shell - system - enable - /var/run/.ptmx - /etc/.ptmx - cp	654
/usr/.ptmx - /boot/.ptmx - while - cp - chmod - cat'	653
%input[0] - /var/.ptmx - shell - system - enable - /var/run/.ptmx	656
/dev/.ptmx - /dev/shm/.ptmx - rm - while - %input[0] - /bin/.ptmx	662
/var/tmp/.ptmx - /boot/.ptmx - chmod - /tmp/.ptmx - >/.ptmx - /usr/.ptmx	649

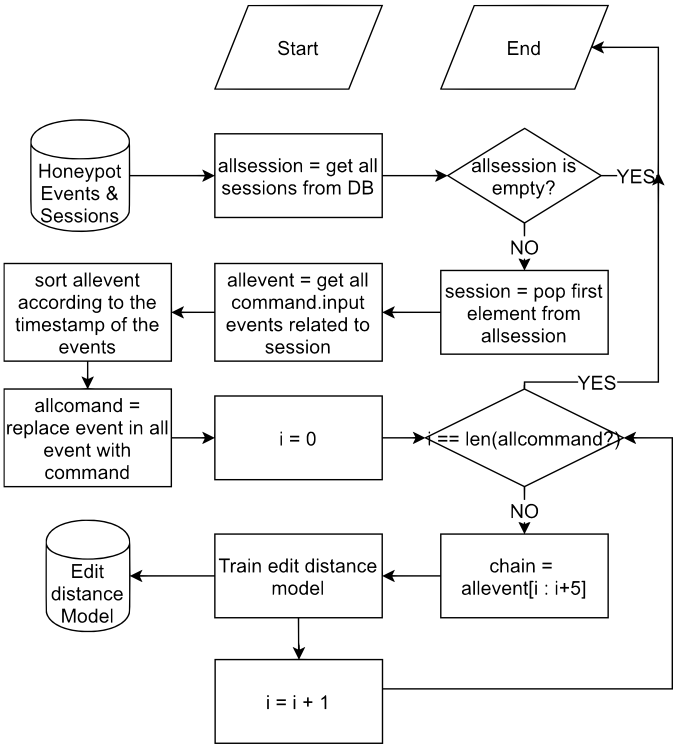

Fig. 6. Training methodology.

6) Assign an unique integer to each command. This array contains a sub-chain of a certain length (5 in this case).
7) Feed the array into the edit distance training model. The model stores each sub-chain, the next command input by the attacker and the frequency of this sub-chain in our dataset.

After doing the above operations, we end up with a number of arrays, each containing 5 commands. One such array represents one sub-chain of commands input by the attacker. We also store their frequency or the number of times this sub-chain is seen in our dataset.

B. Why Levenshtein distance?

Levenshtein distance is a type of edit distance which quantifies how dissimilar two sequences (or strings) are. This is an industry standard technique to compare two sequences when one sequence can be obtained from the other by insertion, deletion or substitution of certain elements. There are other specialized types of edit distances like Longest common subsequence [26] which only considers insertion and deletion but not substitution. Another example is Hamming distance [27] which only considers substitution, so it works for sequences of same length only. However in our case the two sequences can be totally different in content and length requiring the use of Levenshtein distance.

C. Testing our Model

We have split our total dataset into training and testing sets in a ratio of $80\%/20\%$. For testing, we have used normalized Levenshtein distance [23]. Levenshtein distance is a metric of similarity between two strings. We use this to detect the similarity between two chain of commands. It is a type of edit distance. A normalized Levensthein distance is calculated by dividing the total Levenshtein distance by the total length of both the sequences.

Now, to test a command chain, we perform the same operations as Fig. 6. At the end of each loop we get an array of 5 commands. The first 4 commands form a sub-chain whereas the 5^{th} command is the actual next command. We then calculate the edit distance of the sub-chain with all the sub-chains in our model. We select the sub-chain with the least edit distance as the match. If there are multiple matches we consider the one that has the highest frequency. A hash table lookup of the matched sub-chain gives us the predicted next command from our model. We can then test the accuracy of our system by comparing our predicted next command with the actual next command.

VII. RESULT

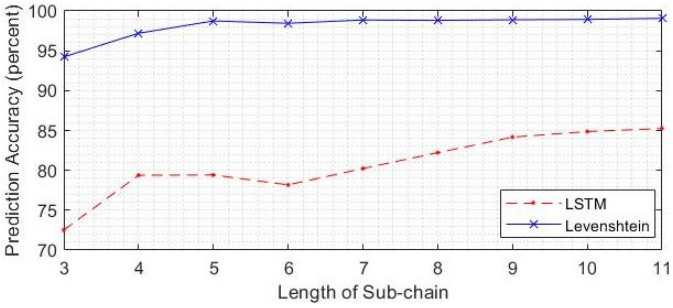

Fig. 7. Prediction Accuracy.

Table II shows the accuracy of the predictions of our training model. This same result is plotted in Fig. 7. The result shows that, even after the attacker inputs only 3 commands into the system, our methodology can predict with great certainty what the next command input by the attacker will be. This

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

TABLE II
PREDICTION ACCURACY AND TIME TAKEN FOR 5000 SAMPLES.

Length of Sub-chain	LSTM Accuracy (%)	LSTM Train Time (s)	LSTM Test Time (s)	Levenshtein Accuracy (%)	Levenshtein Train Time (s)	Levenshtein Test Time (s)
3	72.52	1570.88	7.53	94.23	0.05	0.01
4	79.39	1758.75	7.40	97.20	0.03	0.02
5	79.43	1908.89	8.37	98.74	0.04	0.01
6	78.18	2117.82	9.30	98.45	0.03	0.01
7	80.23	2360.41	10.04	98.86	0.03	0.02
8	82.22	2308.45	10.97	98.81	0.03	0.01
9	84.17	2297.39	11.68	98.89	0.03	0.01
10	84.87	2408.08	12.09	98.93	0.04	0.02
11	85.25	2453.72	13.07	99.07	0.03	0.02

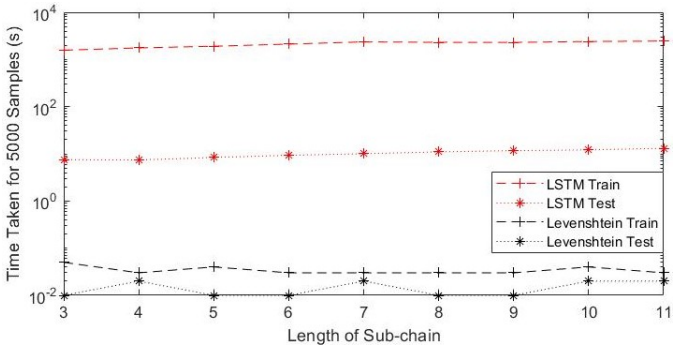

Fig. 8. Time Complexity of LSTM vs Levenshtein

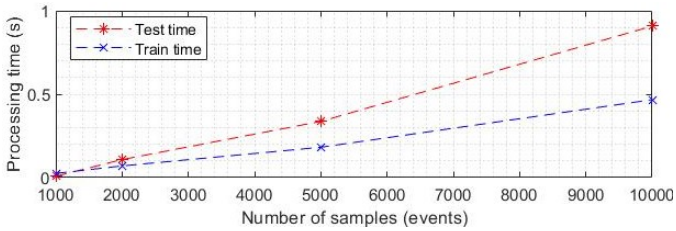

Fig. 9. Complexity of the system.

means that predictive analysis of attacker behavior works in this dataset. As the attacker keeps inputting more commands into the system, then length of the sequence increases and so does the prediction accuracy of our system.

Table II and Fig. 7 also compares the accuracy of our model with Long Short Term Memory (LSTM). As seen our model (Levenshtein distance) has much better accuracy than LSTM. Furthermore, Table II shows the superior performance of our methodology compared to LSTM. The training time of LSTM is much slower than our model and is not practical for real systems.

Fig. 8 compares the time complexity of LSTM with that of Levenshtein distance. It shows that edit distance is much faster than LSTM during both training and testing phase.

Fig. 9 shows the time complexity of our training and testing procedures. Note that, the x-axis of Fig. 9 is sample size not sub-chain length. The result shows that the process is linear and thus can be scaled horizontally. This is important for a realtime system that collects data from thousands of hosts. As the whole process from data collection to behavioral analysis scales linearly, we can use clustering or map-reduce to support virtually infinite hosts.

VIII. CONCLUSION AND FUTURE WORK

In this work we have outlined a robust framework for automated analysis of botnet behavior. To do this, We have used a novel approach of analyzing chain of commands input by attackers into the shell. We have also incorporated a sequential analysis to verify that our approach can correctly predict attacker behavior. A prediction accuracy of $94 - 99\%$ proves the validity of our approach.

In future we will look into real-time classification by using both benign and malicious datasets. We would then try to make the methodology more versatile by utilizing dataset generated by other malware not only Mirai. We will also cross validate our procedure with a newer dataset collected over a different time span.

REFERENCES

[1] R. Puri, "Bots & botnet: An overview," *SANS Institute*, vol. 3, 2003.

[2] M. Feily, A. Shahrestani, and S. Ramadass, "A survey of botnet and botnet detection," in *2009 3rd International Conference on Emerging Security Information, Systems and Technologies*. IEEE, 2009.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[4] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: analysis of a botnet takeover," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 635–647.

[5] S. Shin and G. Gu, "Conficker and beyond: a large-scale empirical study," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 151–160.

[6] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[7] A. Karasaridis, B. Rexroad, D. A. Hoeflin *et al.*, "Wide-scale botnet detection and characterization." *HotBots*, vol. 7, pp. 7–7, 2007.

[8] J. R. Binkley and S. Singh, "An algorithm for anomaly-based botnet detection." *SRUTI*, vol. 6, pp. 7–7, 2006.

[9] G. Gu, J. Zhang, and W. Lee, "Botsniffer: Detecting botnet command and control channels in network traffic," 2008.

[10] H. Choi, H. Lee, H. Lee, and H. Kim, "Botnet detection by monitoring group activities in dns traffic," in *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*. IEEE, 2007.

[11] R. Villamarín-Salomón and J. C. Brustoloni, "Identifying botnets using anomaly detection techniques applied to dns traffic," in *2008 5th IEEE Consumer Communications and Networking Conference*. IEEE, 2008.

[12] D. Dagon, "Botnet detection and response," in *OARC workshop*, 2005.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.

[13] A. Schonewille and D.-J. Van Helmond, "The domain name service as an ids," *Research Project for the Master System-and Network Engineering at the University of Amsterdam*, 2006.
[14] N. Provos *et al.*, "A virtual honeypot framework." in *USENIX Security Symposium*, vol. 173, no. 2004, 2004, pp. 1–14.
[15] M. Oosterhof, "Cowrie ssh/telnet honeypot," 2016.
[16] R. K. Shrivastava, B. Bashir, and C. Hota, "Attack detection and forensics using honeypot in iot environment," in *International Conference on Distributed Computing and Internet Technology*. Springer, 2019.
[17] S. Dowling, M. Schukat, and E. Barrett, "Using reinforcement learning to conceal honeypot functionality," in *Joint European Conf. on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018.
[18] G. Kambourakis, C. Kolias, and A. Stavrou, "The mirai botnet and the iot zombie armies," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 267–272.
[19] N. B. Said, F. Biondi, V. Bontchev, O. Decourbe, T. Given-Wilson, A. Legay, and J. Quilbeuf, "Detection of mirai by syntactic and behavioral analysis," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 224–235.
[20] B. Lingenfelter, I. Vakilinia, and S. Sengupta, "Analyzing variation among iot botnets using medium interaction honeypots," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2020, pp. 0761–0767.
[21] S. Deshmukh, R. Rade, D. Kazi *et al.*, "Attacker behaviour profiling using stochastic ensemble of hidden markov models," *arXiv preprint arXiv:1905.11824*, 2019.
[22] R. Rade, S. Deshmukh, R. Nene, A. S. Wadekar, and A. Unny, "Temporal and stochastic modelling of attacker behaviour," in *International Conference on Intelligent Information Technologies*. Springer, 2018.
[23] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE trans. on pattern analysis and machine intelligence*, vol. 29, no. 6, 2007.
[24] F. Sadique, K. Bakhshaliyev, J. Springer, and S. Sengupta, "A system architecture of cybersecurity information exchange with privacy (cybex-p)," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2019, pp. 0493–0498.
[25] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," 2008.
[26] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.
[27] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 10,2021 at 19:06:00 UTC from IEEE Xplore. Restrictions apply.