

# 2021 年臺灣國際科學展覽會 優勝作品專輯

作品編號	190040
參展科別	電腦科學與資訊工程
作品名稱	<b>HoneySurfer: Intelligent Web-Surfing Honeypots</b>
國家	<b>Singapore</b>
就讀學校	<b>Raffles Institution</b>
作者姓名	<b>Janessa Valencia Guo Jiaxuan Axel Jude Chong He Jun Oliver James Tan</b>

## 作者照片



## 1 Introduction

In Singapore's evolving cyber landscape, 96% of organisations have suffered at least one cyber attack and 95% of organisations have been reporting more sophisticated attacks in the frame of one year according to a 2019 report<sup>[1]</sup> by Carbon Black. As such, more tools must be utilised to counter increasingly refined attacks performed by malicious actors. Honeypots are effective tools for studying and mitigating these attacks. They work as decoy systems, typically deployed alongside real systems to capture and log the activities of the attacker. These systems are useful as they can actively detect potential attacks, help cybersecurity specialists study an attacker's tactics and even misdirect attackers from their intended targets. Honeypots can be classified into two main categories:

1. **Low-interaction honeypots** merely emulate network services and internet protocols, allowing for limited interaction with the attacker.
2. **High-interaction honeypots** emulate operating systems, allowing for much more interaction with the attacker.

Although honeypots are powerful tools, its value diminishes when its true identity is uncovered by attackers. This is especially so with attackers becoming more skilled through system fingerprinting or analysing network traffic from targets and hence, hindering honeypots from capturing more experienced attackers. While substantial research has been done to defend against system fingerprinting scans (see *1.1 Related Work*), not much has been done to defend against network traffic analysis. As pointed out by Symantec<sup>[2][3]</sup>, when attackers attempt to sniff network traffic of the system in question, the lack of network traffic raises a red flag, increasing the likelihood of the honeypot's true identity being discovered. In addition, the main concern with regards to honeypot deployment being their ability to attract and engage attackers for a substantial period of time, an increased ability to interest malicious actors is invaluable. Producing human-like network activity on a honeypot would appeal to more malicious actors. Hence, this research aims to build an intelligent web-surfer which can learn and thus simulate human web-surfing behaviour, creating evidence of human network activities to disguise the identity of honeypots as production systems and luring in more attackers interested in packet sniffing for malicious purposes.

## 1.1 Related Works

Previous studies looking to increase the deceptive capabilities of honeypots have identified a few prevailing problems that arise, such as large numbers of open ports, suspicious timestamps on server requests, and invalid HTTP request replies.<sup>[4]</sup> Fixes to these problems are suggested via reconfiguring honeypot scripts as well as opening only essential ports.<sup>[4]</sup> Other solutions proposed include utilising dedicated hardware to minimise arbitrary software delays<sup>[5]</sup> and automatic honeypot deployment<sup>[6]</sup>.

However, few researchers have studied the idea of automated and dynamic honeypots, and even fewer (perhaps none) have studied the emulating of human-like network activities in high-interaction honeypots. Some intriguing instances with regards to dynamic honeypots/honeynets include BAIT-TRAP which emulates vulnerabilities based on attractive target services in a common network<sup>[7]</sup>, and automated failure injecting honeypots which adapt to an attacker's interaction once they have entered the system<sup>[8]</sup>. There have also been studies conducted to review these dynamic and intelligent honeypots, comparing them against their static counterparts, all of which have concluded that dynamic honeypots pose a significant advantage over their static counterparts in its engagement of attackers.<sup>[5][9]</sup>

While there has been considerable research done in developing intelligent web-browsing behaviour, much of it has been focused on improving user experience through site<sup>[10-11]</sup> or hyperlink<sup>[12]</sup> recommendations. Meanwhile, we believe it a worthwhile concept to integrate intelligent web-browsing behaviour to strengthen honeypots. We explore plausible means to go about implementation in **2 Methodology**.

## 2 Methodology

### 2.1 Building the intelligent web-surfer

For the web-surfer to work effectively, it must be **i) able to produce credible browsing behaviour** so as to not reveal itself easily, **ii) highly adaptable** so as to keep up with relevant news topics and **iii) dynamic in response** so as to actively explore new websites when given the same test cases to prevent raising suspicions. A few basic solutions were conceptualised and the degree to which each solution satisfies the criteria above are shown in **Table 1** on the next page.

Method	Description	Criteria		
		i) Credibility	ii) Adaptability	iii) Dynamism
1. Rule-based system	Code rules to surf websites based on human behaviour	-High credibility	-Medium adaptability; dictionary of websites to be visited must be changed occasionally	-Dynamic
2. Machine Learning based on links	Learn what websites to visit from given data	- High credibility	-Low adaptability; model must be retrained on new data occasionally	-Prone to repetitive behaviour ( <i>see below</i> )
3. Top-ranked searches	Surfs the top visited websites	-Low credibility	-High adaptability	-Dynamic

**Table 1** Possible solutions for an intelligent web-surfer

While **Method 1** might seem like the best solution for the problem, it requires expert knowledge on human browsing behaviour which we do not have. In **Method 2**, there is a high chance of the model overfitting the data due to the nature of human-browsing data. For a given sequence of websites in the data, there are very few possible websites that realistically succeed this sequence. This could lead to the model generating repetitive behaviour which would raise suspicions amongst attackers. Hence, a novel method combining **Method 1** and **Method 2** was identified, in which the intelligent web surfer comes with two components, **i) a browser agent** (see *Appendix B1*) which simulates basic actions of a human surfer such as clicking on hyperlinks, performing queries on search engines, keying in specific URLs and leaving the browser, **ii) a decision machine learning (ML) model** which sends the browser agent different inputs corresponding to the set of possible actions a human can take when browsing the web (see **Figure 1**).

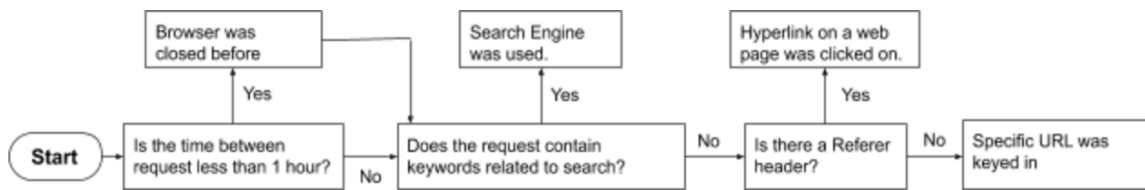


**Figure 1** Diagrammatic representation of the solution

We implemented the browser agent using Selenium WebDriver for Python, an open source web-based automation tool which will allow us to browse the web from a Python script. The agent was based on Chrome browser, chosen for its popularity of about 70% usage share<sup>[13]</sup>. According to research<sup>[14]</sup> on web browsing behaviour, hyperlinks in both web pages and search engines account for the majority of page loads at 45.1%, whereas keying in the URLs (Uniform Resource Locator) and utilising autocomplete accounted for 33.0%. Hence, these behaviours were chosen to build the browser agent. Complex behaviour such as posting information, switching tabs, page reloads as well as going forward and back a page were excluded as they were difficult to extract from the training data provided.

### 2.1.1 Building the ML Model

The model was trained with anonymised data provided by DSO containing individual browsing data. Data came in the form of raw (Hypertext Transfer Protocol) HTTP requests. Hence, they had to be filtered to remove meaningless data such as .png and .jpg images which send additional requests when a web page is being loaded. (see *Appendix B2*) A program was then created to label each request or a cluster of requests with the following set of rules (shown in **Figure 2**), having each set of labels represented with the action state. (see *Appendix B3*) Afterwards, the data was segmented to extract and utilise the browsing data of 75 people over a span of 14 days. Finally, the model was trained with the labelled sequence of events.



**Figure 2** Flowchart logic for labelling browsing behaviour

The model (see *Appendix B4*) made use of Tensorflow<sup>[15]</sup> (a popular machine learning library owned by Google) and Keras (TensorFlow's high-level API for building and training deep learning models)<sup>[16]</sup>. A recurrent neural network model which predicts the next action state given its previous action states was deployed. The architecture of the model is as follows: an embedding layer to encode inputs into vectors, a GRU (Gated Recurrent Unit) layer, and a Dense output layer which outputs the action state the browser agent is to perform next. An alternative to the GRU layer is the less recently introduced long short term memory (LSTM) layer. Based on online

sources and reviews<sup>[17]</sup>, it was concluded that both layers were reputable in their ability to perform sequence-based tasks with long-term dependencies. However, a GRU model is able to achieve a lower loss after training with the same number of epochs, and can train approximately 3.84% faster than its LSTM counterpart. Hence, we decided to proceed with a GRU layer.

To train and test the model, the data set was split into a 4:1 training to validation ratio. The model was trained with 100 epochs and the weights corresponding to the lowest validation loss were saved. (see *Appendix A1*)

### 2.3 Building the honeypot system

A Windows 10 virtual machine (VM) was deployed as a high-interaction honeypot. Windows operating system (OS) chosen for its popularity around the world, with a usage share of 39.2%.<sup>[18]</sup> The program files were encrypted and hidden with the built-in Windows functionality, so as to ensure that they do not give away the identity of the honeypot. To monitor and visualise the activities of potential attackers, the following software were used:

- *Sysmon*<sup>[19]</sup>: a Windows system service/device driver that logs system activity to the Windows event log, providing detailed information about process creations, network connections, and changes to file creation time
- *Winlogbeat*<sup>[20]</sup>: Logs security events such as logon successes (4624) and failures (4625), and can be configured to read from any event log channel, providing access to crucial Windows data,
- *Packetbeat*<sup>[21]</sup>: Captures and analyses network traffic. In the honeypot, HTTP requests from the pages visited are captured, allowing one to keep track of websites recently browsed by the web-browsing program.
- *ELK*<sup>[22]</sup> (*Elasticsearch, Logstash, and Kibana*) Stack work together to analyse, filter and visualise the data from the above Beats respectively. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from the above sources simultaneously, transforms it, and then sends it to Elasticsearch. Kibana allows users to visualize the data with charts and graphs. (see *Appendix A4*)

## 2.4 Assessment of the honeypot

The honeypot we built was assessed on two metrics, **i) the deception capabilities** of the web-surfer and **ii) the functionality of the honeypot**.

For **i)**, we collected 22 responses from a survey sent to DSO interns and staff. The program was executed and HTTP as well as HTTPS logs generated were captured using Telerik Fiddler. <sup>[23]</sup> Participants were first given 3 real human logs as examples. They were then given 6 logs from three different types of browsing, in random order as follows:

- i) randomly browsed URLs to serve as a baseline (see *Appendix B6*),
- ii) HoneySurfer (see *Appendix B5*) and
- iii) real human browsing.

They were asked to rate each log on a Likert Scale of 1 to 6 as to whether they believed the logs were automated or human-generated, with 1 representing logs that were “definitely automated” and 6 representing logs that were “definitely human-generated”. The rationale behind their ratings were also collected. (see *Appendix A2.2*). *t*-values were calculated to determine the magnitude of similarity between real humans and each other type of browsing.

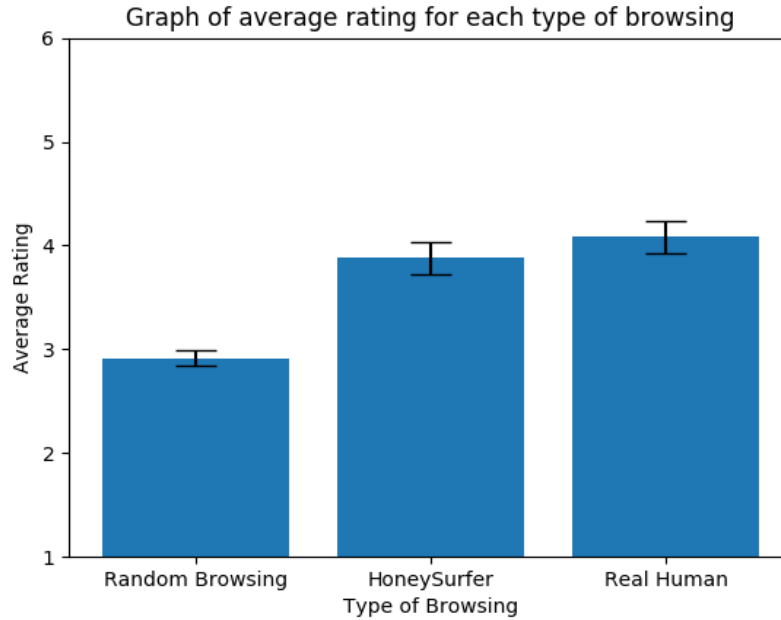
For **ii)**, functionality testing was performed on all of the logging components (*Packetbeat*, *Winlogbeat* & *Sysmon*) to ensure that all components were working correctly. The actions below were performed from another computer and the events logged for each component were checked if they were being monitored on the *ELK Stack*.

1. *Packetbeat*: Browsed URLs (*cURL*) on the honeypot to generate and log HTTP requests.
2. *Winlogbeat* & *Sysmon*: Started a process.



### 3 Results

#### 3.1 Survey Results



**Figure 3** Graph of average rating for each type of browsing.

(\*Error bars represent standard error of average rating from each type of browsing)

Type of Logs	Sample Size	Mean ± SE	t-value
Random Browsing	6	2.91 ± 0.0737	-6.71
HoneySurfer	6	3.88 ± 0.158	-0.901
Real Human	6	4.08 ± 0.157	N/A

**Table 2** Descriptive data for each type of browsing.

#### 3.2 Honeypot Functionality Testing

Via SSH into the honeypot, a dummy program was started up and successfully logged under *Winlogbeat & Sysmon* and HTTP requests to a website via *cURL* were successfully logged under *Packetbeat*. Hence, all components were working correctly.

## 4 Discussion

From **Figure 3**, both HoneySurfer and human-generated logs outperformed the logs from random browsing, with human-generated logs having a slightly higher score than HoneySurfer. In addition, the low  $t$ -value of -0.901 for HoneySurfer compared to the high  $t$ -value of -6.71 for random browsing meant that our HoneySurfer was closely similar to that of real humans. Hence, our intelligent web-surfer successfully produced logs that looked almost like those of real humans to the participants. However, some of our more experienced participants who are more familiar with analysis of network logs also provided feedback to improve our web browsing agent. These are covered in *5 Limitations and Further Work*.

## 5 Limitations and Further Work

Ideally, further testing of the honeypot would include deployment against real attackers. For instance, it could be uploaded onto a cloud service such as Amazon Web Services. However, due to time and resource constraints, we were not able to ensure that the honeypot would be secure enough to prevent human attackers from potentially exploiting it to access a real system.

Furthermore, there are many other aspects of improving the realistic quality of the network activity that have yet to be explored. A few of the survey participants pointed out that the typing speed was too consistent for searches and that the human-generated searches were more likely to make mistakes in spelling. Hence, a good area for further work would be randomising the typing speeds as well as mimicking spelling errors when our honeypot creates a query. Another area for development would be to include other forms of network activity such as composing emails and documents, SSH and FTP (File Transfer Protocol). However the content generated in each type of network activity will be subject to close inspection and analysis by attackers and more time is needed to develop such realistic systems.

Apart from improvements in functionality, an interesting application we hope to explore is the ability of our honeypot to counteract packet-sniffing. POST functions containing login credentials to a fake website we create could be integrated into the honeypot. This could entice attackers to enter these credentials, allowing us to log their activity on the website.

## 6 Conclusion

We successfully developed a novel approach to produce intelligent web-browsing behaviour which was then integrated into a high-interaction honeypot. Cybersecurity is rising in the agenda of many countries and organisations. It is obvious that honeypots have high potential for development and should be implemented more frequently in the cyber landscape. Therefore, every step forward should be valued. As more sophisticated attacks and attackers are on the rise, we hope that this prototype can be implemented in future honeypots so as to help threat hunters stay one step ahead.

## 7 Bibliography

- [1]: Carbon Black (2019). Singapore: Global Threat Report: Defender Power On The Rise.
- [2]: L. Oudot, T. Holz (2004) Defeating Honeypots : Network issues, Part 1: Symantec Connect. Retrieved from <https://www.symantec.com/connect/articles/defeating-honeypots-network-issues-part-1>
- [3]: Spitzner (2004) Problems and Challenges with Honeypots: Symantec Connect. Retrieved from <https://www.symantec.com/connect/articles/problems-and-challenges-honeypots>
- [4]: Dahbul, R. N., Lim, C., & Purnama, J. (2017). Enhancing Honeypot Deception Capability Through Network Service Fingerprinting. *Journal of Physics: Conference Series*, 801, 012057. doi: 10.1088/1742-6596/801/1/012057
- [5]: Tsikerdekis, M., Zeadally, S., Schlesener, A., & Sklavos, N. (2018). Approaches for Preventing Honeypot Detection and Compromise. *2018 Global Information Infrastructure and Networking Symposium (GIIS)*. doi: 10.1109/giis.2018.8635603
- [6]: Fu, X., Yu, W., Cheng, D., Tan, X., Streff, K., & Graham, S. (2006). On Recognizing Virtual Honeypots and Countermeasures. *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*. doi: 10.1109/dasc.2006.36
- [7]: Jiang, X. (2004). BAIT-TRAP: a Catering Honeypot Framework.
- [8]: Wagener G., State R., Dulaunoy A., Engel T. (2009) Self Adaptive High Interaction Honeypots Driven by Game Theory. In: Guerraoui R., Petit F. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2009. Lecture Notes in Computer Science, vol 5873. Springer, Berlin, Heidelberg

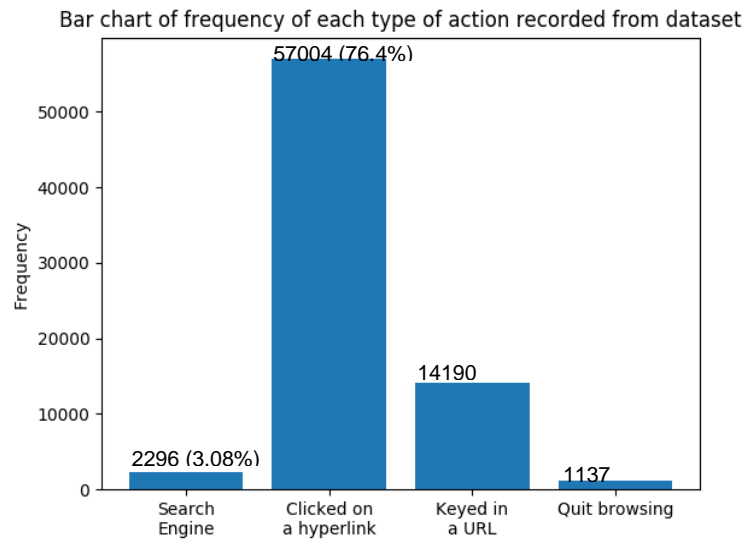
- [9]: Zakaria, W. Z. A., & Kiah, M. L. M. (2013). A review of dynamic and intelligent honeypots. *ScienceAsia*, 39S(1), 1. doi: 10.2306/scienceasia1513-1874.2013.39s.001
- [10]: Rabbi, M. F., Ahmed, T., Chowdhury, A. R., & Islam, M. R.-O.-B. (2006). Adaptive Web Browser: An Intelligent Browser. *2006 International Conference on Communication Technology*. doi: 10.1109/icct.2006.341854
- [11]: Lai, H., & Yang, T.-C. (2000). A system architecture for intelligent browsing on the Web. *Decision Support Systems*, 28(3), 219–239. doi: 10.1016/s0167-9236(99)00087-1
- [12]: Li, J., Xing, Z., Ye, D., & Zhao, X. (2016). From Discussion to Wisdom: Web Resource Recommendation for Hyperlinks in Stack Overflow. *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC 16*. doi: 10.1145/2851613.2851815
- [13]: Liu, S. (2019). Desktop internet browser market share 2015-2019. Retrieved from <https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/>
- [14]: Weth, C. V. D., & Hauswirth, M. (2013). DOBBS: Towards a Comprehensive Dataset to Study the Browsing Behavior of Online Users. *2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. doi: 10.1109/wi-iat.2013.8
- [15]: Tensorflow. Retrieved from <https://www.tensorflow.org>
- [16]: Keras: The Python Deep Learning library. Retrieved from <https://keras.io/>
- [17]: Muccino, E. (2019). LSTM vs GRU: Experimental Comparison. Retrieved from <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b>
- [18]: Galov, N. (2019). Mobile and Desktop Operating Systems Market Share. Retrieved from <https://hostingtribunal.com/blog/operating-systems-market-share/#gref>
- [19]: Markruss. Sysmon - Windows Sysinternals. Retrieved from <https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon>
- [20]: Winlogbeat. Retrieved from <https://www.elastic.co/products/beats/winlogbeat>
- [21]: Packetbeat. Retrieved from <https://www.elastic.co/products/beats/packetbeat>
- [22]: What is the ELK Stack? Retrieved from <https://www.elastic.co/what-is/elk-stack>
- [23]: Fiddler - Free Web Debugging Proxy - Telerik. Retrieved from <https://www.telerik.com/fiddler>

## 8. Appendix

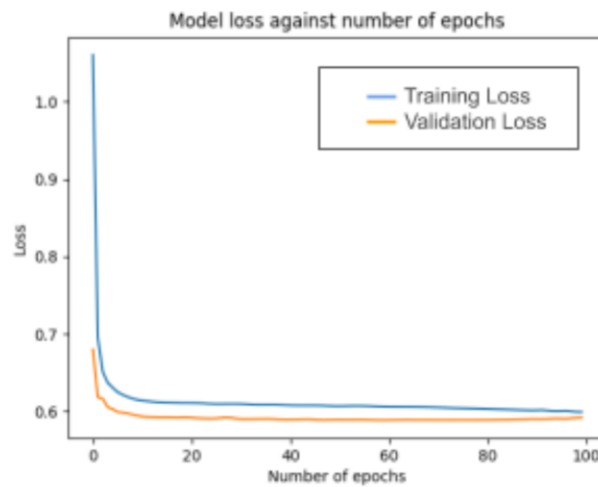
A. Tables & Figures

B. Scripts

### A1 - Model Training



**Figure A1.1** Bar chart of frequency of each type of action recorded from dataset.



**Figure A1.2** Graph of loss curves during training.

A2 Survey Results

Log No.	Random Browsing	HoneySurfer	Human-generated
1	2.90	4.14	3.82
2	2.86	4.32	3.59
3	3.09	3.68	3.95
4	2.86	3.72	4.41
5	3.14	3.27	4.64
6	2.64	4.13	4.09
Mean $\pm$ SE	2.91 $\pm$ 0.0737	3.88 $\pm$ 0.158	4.08 $\pm$ 0.157

**Table A2.1** Table of raw data for each test case

Reasons why survey participants believed the logs to be human-generated	Reasons why survey participants believed the logs to be automated
<ul style="list-style-type: none"> <li>- Google search URL gradually increases as queries are typed in</li> <li>- Common/Familiar sites like google/youtube, social sites</li> <li>- Systematic logs - spends time on one webpage, clicking on the links before moving on to another.</li> <li>- Plausible story behind the logs</li> <li>- Mistakes made when typing website which was then retyped</li> <li>- Sites visited were in Singapore</li> </ul>	<ul style="list-style-type: none"> <li>- Repeated visits to the same host</li> <li>- Unfamiliar websites</li> <li>- Too many different area codes</li> <li>- User seems to switch to unrelated sites</li> <li>- User seems to spend little time on each site</li> <li>- <i>Consistent time intervals when keying in Google queries</i></li> </ul>

**Table A2.2** Table of reasons why survey participants believed logs to be human-generated/automated

A3 - Data Analysis

Log Type	p-value	Interpretation (Reject H <sub>0</sub> /normality if $p < 0.05$ )
Random Browsing	0.5821	Null hypothesis not rejected. Data is in normal distribution.
HoneySurfer	0.5628	Null hypothesis not rejected. Data is in normal distribution.
Real Humans	0.9187	Null hypothesis not rejected. Data is in normal distribution.

**Table A3.1** Table of results from Shapiro-Wilk test for normality

## A4 - Functionality Testing

### i) Packetbeat

```
type: http @timestamp: Dec 27, 2019 @ 19:25:09.179 source.port: 52923 source.bytes: 75B source.ip: 192.168.92.184
agent.hostname: DESKTOP-UUJ4FUG @version: 1 http.request.headers.content-length: 0 http.request.headers.user-agent: curl/7.55.1
http.request.method: get http.response.status_code: 200 destination.port: 80 destination.ip: 93.184.216.34
url.full: http://example.com/ tags: beats_input_raw_event _id: pvy7VG8BugnUfoOoPtqp _type: _doc _index: packetbeat-7.5.0-
2019.12.27-000002 _score: -
```

Figure A4.1 Photo of successful logging of HTTP request to http://example.com

### ii) Winlogbeat & Sysmon

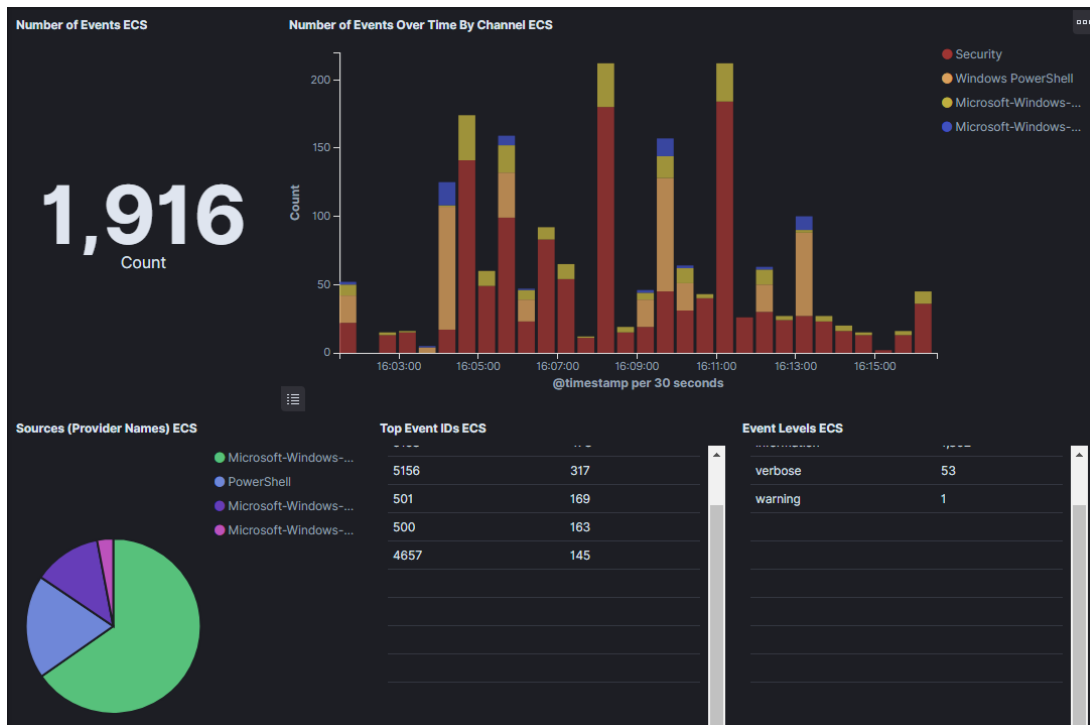
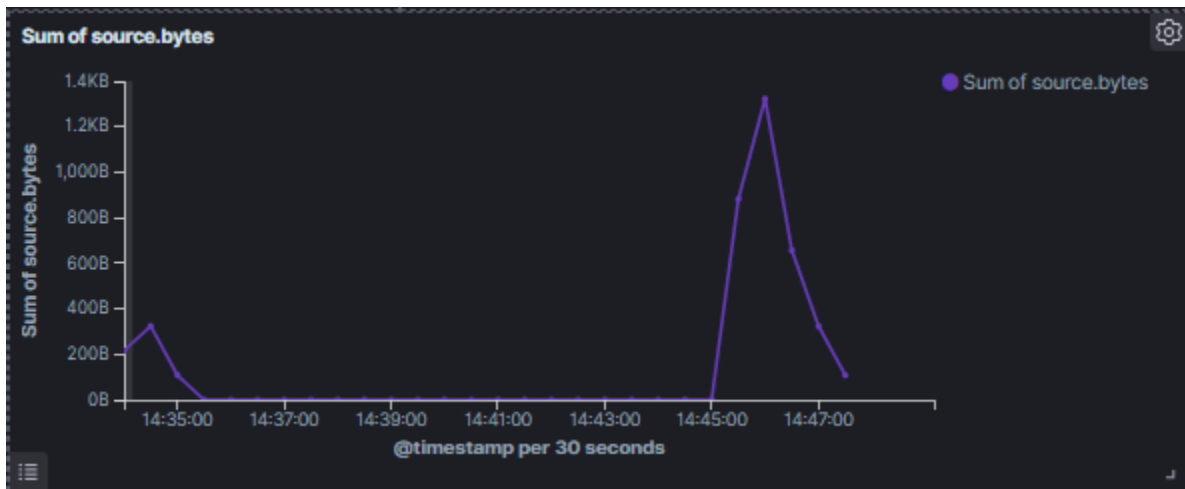
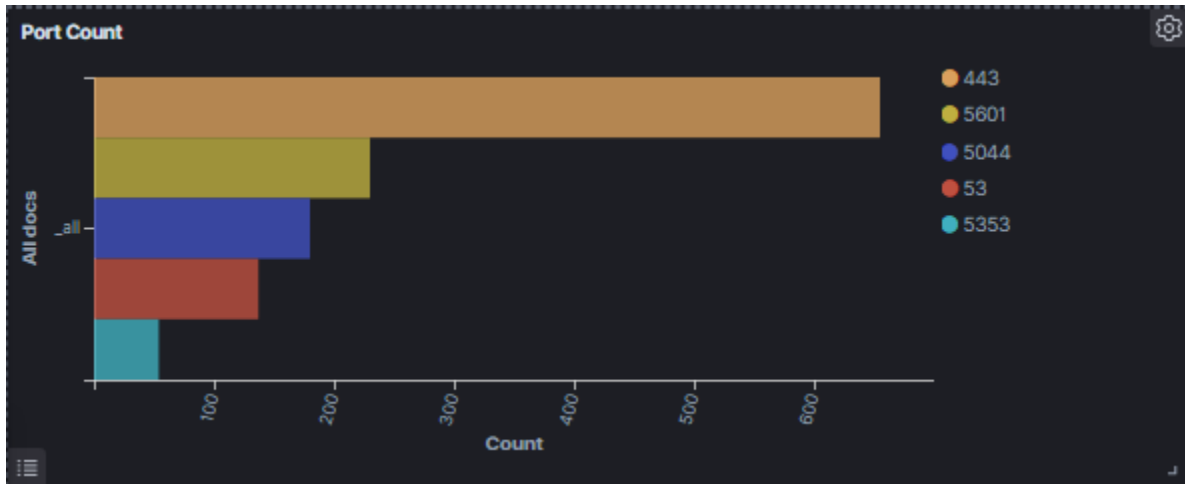


Figure A4.2 Photo of Winlogbeat & Sysmon Dashboard

```
@timestamp: Dec 27, 2019 @ 15:34:47.248 winlog.event_id: 4688 winlog.record_id: 1100384 winlog.task: Process Creation winlog.provider_guid: {54849625-
5478-4994-a5ba-3e3b0328c38d} winlog.process.thread_id: 4,372 winlog.process.pid: 4 winlog.event_data.SubjectLogonId: 0xf68fb7
winlog.event_data.TargetUserName: - winlog.event_data.TokenElevationType: %%1936 winlog.event_data.ProcessId: 0xfd0
winlog.event_data.SubjectDomainName: DESKTOP-UUJ4FUG winlog.event_data.TargetDomainName: - winlog.event_data.CommandLine: malware.exe
winlog.event_data.TargetLogonId: 0x0 winlog.event_data.NewProcessId: 0x19b4 winlog.event_data.MandatoryLabel: S-1-16-8192
```

Figure A4.3 Photo of successful logging of process start-up of dummy program - "malware.exe"



**Figure A4.4** Photo of port count and sum of source.bytes visualisations



## B1 - Browser Agent

```
#Importing modules...
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.common.exceptions import WebDriverException
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.proxy import Proxy, ProxyType
import random
import time

#Class BrowserAgent
class BrowserAgent():

    #Initialize browser
    def __init__(self):
        prox = Proxy()
        prox.proxy_type = ProxyType.MANUAL
        prox.http_proxy = "localhost:8888"
        prox.ssl_proxy = "localhost:8888"
        capabilities = webdriver.DesiredCapabilities.CHROME
        prox.add_to_capabilities(capabilities)

        self.driver = webdriver.Chrome(desired_capabilities=capabilities)
        self.driver.get("https://www.google.com")
        self.visitedList = [] #List of visited websites

    #Get hyperlink from website
    def clickOnHyperlink(self):
        hrefList = self.driver.find_elements_by_xpath("//a[@href]")
        if len(hrefList) == 0: #Goes back a page if the current webpage has no hyperlinks
            self.driver.back()
        else:
            N = random.randint(0, len(hrefList))
            href = hrefList[N].get_attribute("href")
            self.driver.get(href)
            self.visitedList.append(href) #Appends previously visited website to visited list

    #Generate random google search
    def randomGoogle(self):
        #List of common starting phrases
        googleList = ["What is", "Where is", "Best places", "Where to go in", "What to do when", "Why is",
                     "What kind of", "Where am I", "Where should I", "weather", "weather in", "Why you should",
                     "How to", "wikihow", "wikipedia", "Who is the", "top", "Define"]
        self.driver.get("http://www.google.com") #Go to google home page
        searchField = self.driver.find_element_by_name("q")
        searchField.send_keys(random.choice(googleList)) #Select random starting phrase
        googleSuggestion = WebDriverWait(self.driver, 10).until(EC.visibility_of_all_elements_located((By.XPATH,
        "//form[@action='/search' and @role='search']/ul[@role='listbox']/li//span")))
        chosen = random.choice(googleSuggestion) #Select random autocomplete
        chosen.click()

    #Get query and click on random search result from the first page of the search resultt
    def getQuery(self):
        results = self.driver.find_elements_by_xpath('//div[@class="r"]/a/h3')
        N = random.randint(0, len(results)) - 1
        try:
            time.sleep(5)
            results[N].click()
            time.sleep(random.randint(3,10))
        except WebDriverException: #Rerun function if error occurs
            self.getQuery()

    #Visit specific URL
    def getURL(self):
        #List of common URLs we specified
        commonList = ["https://world.taobao.com/", "https://www.qq.com/", "https://www.reddit.com/"]
```

```

        "https://www.facebook.com/", "https://www.youtube.com/",
        "https://www.amazon.com/", "https://www.imdb.com/", "https://www.nytimes.com/",
        "https://www.tripadvisor.com.sg/", "https://www.indeed.com.sg/?r=us",
        "https://www.urbandictionary.com",
        "https://www.hardwarezone.com.sg/", "https://sg.carousell.com/", "https://www.instagram.com/",
        "https://www.asiaone.com", "https://www.kiasuparents.com/kiasu/",
        "https://stomp.straitstimes.com/", "https://www.aliexpress.com/", "https://www.lazada.sg/",
        "https://sg.yahoo.com/?p=us", "https://www.straitstimes.com/", "https://www.qoo10.sg/?__ar=Y",
        "https://www.channelnewsasia.com/", "https://www.amazon.com/", "https://www.ebay.com.sg/"]
url = random.choice(commonList + self.visitedList) #Choose random url based on common + previously visited url
self.driver.get(url)
self.visitedList.append(url) #Appends visited URL to previously visited website list

#Quit browsing the web
def quitBrowsing(self):
    time.sleep(random.randint(3600,36000))

#Parsing input
def doInput(self, method_no):
    try:
        if method_no == "1":
            self.randomGoogle()
            self.getQuery()
        elif method_no == "2":
            self.clickOnHyperlink()
        elif method_no == "3":
            self.getURL()
        elif method_no == "4":
            self.quitBrowsing()

```

## B2 - Extract and filter out HTTP requests

```

##Importing modules
import difflib, glob, gzip, os, re

##Variables
inputPath = '' #####INSERT INPUT PATH HERE#####
outputPath = '' #####INSERT OUTPUT PATH HERE#####

def filterFile(path, outputFile):
    ##Declaring variables/lists
    fileList = []
    getList = [b'temp']
    getLine = b'temp'
    previousLine = b'temp'
    count = 0
    isValidReferer = False
    allOutputList = []
    f = open(outputFile, "a+")

    ##Sorting files with respect to time
    for filename in glob.glob(os.path.join(path, '*.log.gz')):
        fileList.append(filename)
        fileList.sort()

    hostCriteria = (b'doubleclick', b'ads', b'gstatic', b'notify')
    getCriteria = (b'doubleclick', b'adserver', b'adServer', b'jpeg', b'png', b'jpg', b'gif', b'Service')
    for filed in fileList:
        with gzip.open(filed, 'r') as file:
            for line in file:
                if b'Host:' in line: #Check if line contains "Host", i.e. potentially a HTTP request
                    if any(i in line for i in hostCriteria): #If "Host" is an ad server, ignore request
                        isValidReferer = False
                        continue
                    else:
                        #Check if line above "Host" line contains the GET header, indicating a HTTP request
                        if b'</TimeStamp><Name>HTTP</Name><Header>GET' in previousLine:

```

```

        if any(f in previousLine for f in getCriteria):
            isValidReferer = False
            continue #Ignore if request is from an ad or image
        else:
            if not any(b'upload' or b'download') in line and b'ad' in line:
                isValidReferer = False
                continue
            else:
                tempGLine = previousLine.split(b'<Header>')[1].lstrip()
                for i in getList:
                    #Only treat request as unique if similarity is <50% with respect to all
                    #previously stored requests
                    if difflib.SequenceMatcher(None, tempGLine, i).ratio() < 0.5:
                        isDuplicate = False
                    else:
                        isDuplicate = True
                        isValidReferer = False
                        break
                if isDuplicate:
                    continue
                else:
                    getLine = tempGLine
                    if len(getList) > 40:
                        getList.pop(0)
                    getList.append(tempGLine)
                    count += 1
                    a=re.findall(b'[0-2][0-4]:[0-5][0-9]:[0-5][0-9].[0-9][0-9][0-9]',previousLine)
                    if len(a) != 0:
                        allOutputList.append(str(a[0]))
                        allOutputList.append(str(getLine))
                        allOutputList.append(str(line))
                        isValidReferer = True

            elif b'Referer:' in line:
                if isValidReferer:
                    allOutputList.append(str(line))
                previousLine = line
            allOutputList.append(count)

        for item in allOutputList:
            f.write("%s\n" % item)

def extractData(path, outputPath):
    fileList = []
    ##Sorting files with respect to time
    for items in Path(path).iterdir():
        fileList.append(items)
    fileList.sort()

    i = 0
    for file in fileList[8:22]:
        i += 1
        outputFile = outputPath + "/output" + str(i) + ".txt"
        filterFile(file, outputFile)

#Defining location of path
fileList = []
##Sorting files with respect to time
for items in Path(path).iterdir():
    fileList.append(items)
fileList.sort()

i = 0
for file in fileList[9:]:
    i += 1
    outputPath += str(i)
    os.makedirs(outputPath)
    try:
        extractData(file, outputPath)
    except:
        pass

```

## B3 - Labelling HTTP Requests

```
#Importing modules
import re
from datetime import datetime
from pathlib import Path

##Variables
inputPath = '' #####INSERT INPUT PATH HERE#####
outputPath = '' #####INSERT OUTPUT PATH HERE#####

#Generate labels for a given folder (day)
def generateLabel(file):
    ##Splits the data into individual http requests
    fileLines = ''
    with open(file) as files:
        for line in files:
            fileLines += str(line)
    httpList = re.split("(?=[0-2][0-4]:[0-5][0-9]:[0-5][0-9].[0-9][0-9][0-9])", fileLines)
    httpList.pop(0)

    seq = ""
    tSeq = ""
    prevTime = "00:00:00.000"

    for http in httpList:

        #Obtain difference in time
        currTime = re.findall("[0-2][0-4]:[0-5][0-9]:[0-5][0-9].[0-9][0-9][0-9]", http)[0]
        fmt = '%H:%M:%S.%f'
        tDelta = datetime.strptime(currTime, fmt) - datetime.strptime(prevTime, fmt)
        timeDiff = tDelta.seconds

        #Filters request if time between is less than to 2 seconds
        if timeDiff < 2:
            continue

        #Prevent errors due to measuring time difference at the first request of folder
        #User has quit browsing if time difference > 1 hour
        if prevTime != "00:00:00.000" and timeDiff > 3600:
            seq += "4"

        prevTime = currTime

        #Sort HTTP requests to corresponding methods
        if re.compile("GET /url?").search(http) or re.compile("GET /search?").search(http):
            seq += "1"          #Search Engine
        elif re.compile("Referer:").search(http):
            seq += "2"          #Click on a hyperlink
        else:
            seq += "3"          #Specific URL

    return seq

seqFile = open(outputPath, "w")
#Write the data into .txt files
for folder in Path(inputPath).iterdir():
    seqFile.write("\n")
    for file in Path(folder).iterdir():
        data = generateLabel(file)
        seqFile.write(data)
```

## B4 - Training the ML Model

```

#Version: Tensorflow 2.0 - keras 2.2.4
#Importing modules...
from __future__ import absolute_import, division, print_function, unicode_literals
from tensorflow.keras.models import load_model
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import time
import random

#Variables
#Filepath
filepath = '/home/ac/Desktop/data.txt'

#Input Initialization variables
seq_length = 50
action_size = 4

#Batching variables
BUFFER_SIZE = 10000
BATCH_SIZE = 100

#Model variables
embedding_dim = 64
rnn_units = 256

#Training variables
epochs = 100
val_split = 0.2

#Splits the line into input and target sequences
def split_input_target(line):
    inputSeq = line[:-1]
    targetSeq = line[1:]
    return inputSeq, targetSeq

#Setup the dataset
def setup(text, seq_length):
    # Extract data to np array
    action_vector = np.array([int(c)-1 for c in text if c != '\n'])

    # Create training examples / targets
    num_Dataset = tf.data.Dataset.from_tensor_slices(action_vector)
    sequences = num_Dataset.batch(seq_length+1, drop_remainder=True)
    datasetLine = sequences.map(split_input_target)
    return datasetLine

#Function to build model
def build_model(action_size, embedding_dim, rnn_units, batch_size):
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(action_size, embedding_dim,
                                   batch_input_shape=[batch_size, None]),
        tf.keras.layers.GRU(rnn_units,
                             return_sequences=True,
                             stateful=True,
                             recurrent_initializer='glorot_uniform'),
        tf.keras.layers.Dense(action_size)
    ])
    return model

#Reads the data from file and sort it
with open(filepath) as fp:
    text = fp.readline()
    isSetup = 0
    while text:
        data = setup(text, seq_length)
        text = fp.readline()
        if isSetup == 0:
            dataset = data

```

```

    isSetup = 1
    else:
        dataset = dataset.concatenate(data)

#Shuffle and prepare training and validation dataset
dataset = dataset.shuffle(BUFFER_SIZE)
trainLen = int(len([i for i in dataset]) * (1 - val_split))
train_dataset = dataset.take(trainLen).batch(BATCH_SIZE, drop_remainder=True)
val_dataset = dataset.skip(trainLen).batch(BATCH_SIZE, drop_remainder=True)

#Build the model
model = build_model(
    action_size = action_size,
    embedding_dim = embedding_dim,
    rnn_units=rnn_units,
    batch_size=BATCH_SIZE)

#Defines loss function
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

#Compile model using loss function and Adam Optimizer
model.compile(optimizer='adam', loss=loss)

model.summary()
#File where the checkpoints will be saved
checkpoint = "Checkpoints.hdf5"

#Sets checkpoint when validation loss is < than that of previous epoch
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath = checkpoint,
    monitor = 'val_loss',
    save_weights_only = True,
    save_best_only = True)

#Train the model
history = model.fit(train_dataset, validation_data = val_dataset, epochs=epochs, callbacks = [checkpoint_callback])

#Rebuild and reconfigure model with batch size of 1.
model = build_model(action_size, embedding_dim, rnn_units, batch_size=1)
model.load_weights(checkpoint)
model.build(tf.TensorShape([1, None]))

#Save the model
model.save('MLModel.hdf5')

#Plot training and validation loss functions
plt.plot(history.history['loss'], label = 'training_loss')
plt.plot(history.history['val_loss'], label = 'validation_loss')
plt.title('Model loss against number of epochs')
plt.ylabel('Loss')
plt.xlabel('Number of epochs')
plt.show()

```

## B5 - main() function

```

#Importing modules...
from BrowserAgent import BrowserAgent
import tensorflow as tf
from tensorflow.keras.models import load_model
import random
import numpy as np
import time

#Generate the method number to be sent to Browser Agent

```

```

def generate_browsing(model, start_string):
    # Vectorise start string of numbers
    input_eval = [int(x) for x in str(start_string)]
    input_eval = tf.expand_dims(input_eval, 0)
    model.reset_states()
    predictions = model(input_eval)
    predictions = tf.squeeze(predictions, 0)
    predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()
    predicted_id += 1 #Convert back to action state integers
    return (str(predicted_id))

#main function
def main():
    #Set up
    browser = BrowserAgent()
    MLModel = tf.keras.models.load_model('MLModel.hdf5', compile = False)
    start_string = random.choice(("1", "3")) #Start off with a google search or keying in a specific URL
    while True:
        cmdInput = generate_browsing(MLModel, start_string) #Generates browsing
        browser.doInput(cmdInput) #Sends input to browser agent
        start_string = str(start_string) + ''.join(str(cmdInput))
        time.sleep(random.randint(5, 30)) #Sleeps for random time

#Start the main function
main()

```

## B6 - Random URL Browsing (baseline in survey)

```

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import WebDriverException
from selenium.webdriver.common.proxy import Proxy, ProxyType
import time, random

prox = Proxy()
prox.proxy_type = ProxyType.MANUAL
prox.http_proxy = "localhost:8888"
prox.ssl_proxy = "localhost:8888"

capabilities = webdriver.DesiredCapabilities.CHROME
prox.add_to_capabilities(capabilities)

driver = webdriver.Chrome(desired_capabilities=capabilities)

while True:
    driver.get('http://www.roulette.com/visit/swpun')
    time.sleep(random.randint(3,10))

```

## 【評語】 190040

The topic of this project is Intelligent Web-Surfing Honeypots. This is a complete work. The problem definition could be more clear, which may help the authors come out the solutions. The definition could help the authors identify the critical issue in this topic. There are many previous works that focus on the same topic. The authors may consider to have a comparison with these works.