BotChase: Graph-Based Bot Detection Using Machine Learning

by

Abbas Abou Daya

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

Statement of Contributions

Chapter 3 and Chapter 4 borrow content from two papers "A Graph-Based Machine Learning Approach for Bot Detection" [11] and "BotChase: Graph-Based Bot Detection using Machine Learning" [12].

Abstract

Bot detection using machine learning (ML), with network flow-level features, has been extensively studied in the literature. However, existing flow-based approaches typically incur a high computational overhead and do not completely capture the network communication patterns, which can expose additional aspects of malicious hosts. Recently, bot detection systems which leverage communication graph analysis using ML have gained traction to overcome these limitations. A graph-based approach is rather intuitive, as graphs are true representations of network communications. In this thesis, we propose BotChase, a two-phased graph-based bot detection system that leverages both unsupervised and supervised ML. The first phase prunes presumable benign hosts, while the second phase achieves bot detection with high precision. Our prototype implementation of BotChase detects multiple types of bots and exhibits robustness to zero-day attacks. It also accommodates different network topologies and is suitable for large-scale data. Compared to the state-of-the-art, BotChase outperforms an end-to-end system that employs flow-based features and performs particularly well in an online setting.

Acknowledgements

My genuine acknowledgements reach out to the ones who truly enabled me to be what I am today. I thank all the people who made this thesis possible.

Dedication

To my beloved father, mother and the rest of the Abou Daya family, this one is for you.

Table of Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Botnets 1

1.2 Intrusion Detection Systems 2

1.3 Machine Learning 2

1.4 Contributions 3

1.5 Thesis Organization 4

2 Background 5

2.1 Intrusion Kill-Chain 5

2.2 Bot & Botnet Detection 6

2.2.1 Signature-Based 6

2.2.2 Anomaly-Based 7

2.3 Anomaly-Based Botnet Detection Scopes 8

2.3.1 Host-Based 8

2.3.2 Network-Based 8

2.3.3 Hybrid 8

2.4 Graph-Based Approaches 9

3 BotChase **11**
3.1 Architecture 11
3.2 Dataset Bootstrap 11
3.2.1 Flow Ingestion 11
3.2.2 Graph Transform 13
3.2.3 Feature Extraction 13
3.2.4 Feature Normalization (F-Norm) 15
3.3 Model Training 17
3.3.1 Phase 1 17
3.3.2 Phase 2 18
3.4 Inference 20

4 Evaluation **21**
4.1 Environment Setup 21
4.1.1 Hardware 21
4.1.2 Software 22
4.2 Dataset 22
4.3 Performance 22
4.3.1 Graph Transform, Feature Extraction & Normalization 22
4.3.2 Stand-alone SL 24
4.3.3 Phase 1 (UL) 26
4.3.4 Phase 2 (SL) 28
4.3.5 Feature Normalization 30
4.3.6 Feature Engineering 31
4.4 Comparative Analysis 34
4.4.1 BotMiner Flow-Based vs. Graph-Based Features 36
4.4.2 BClus Flow-Based vs. Graph-Based Features 36
4.4.3 BClus Hybrid vs Graph-Based Features 37

4.4.4 BClus End-to-End vs. BotChase 39
4.4.5 BotGM vs. BotChase 39
4.5 Ensemble Learning 40
4.6 Analysis in an Online Setting 41

5 Conclusion and Future Work **48**
5.1 Conclusion 48
5.2 Future Work 49
5.2.1 Extending F-Norm 49
5.2.2 Classifier Tuning 49
5.2.3 Advanced Feature Engineering 49
5.2.4 Advanced Ensemble Learning 50

References **51**

APPENDICES **57**

A DataFrame4J **58**

B Feature Normalization **67**

List of Figures

2.1 Intrusion kill-chain 6

3.1 Components of the BotChase bot detection system 12
3.2 Example topology of benign hosts with a gateway 16
3.3 Example topology of benign hosts without a gateway 16
3.4 Flowchart of node classification with i nodes and j features 20

4.1 Comparison of SOM and k-Means with respect to training time 28
4.2 Number of hosts outside the benign cluster (HOB) assigned by SOM with and without feature normalization 32
4.3 Number of bots outside the benign cluster (BOB) assigned by SOM with and without feature normalization 32
4.4 The variation of the training and testing time as time progresses 44
4.5 The variation of the training time of HAT vs ID3 46

List of Tables

4.1 Hardware Configuration of the Hadoop Cluster 21
4.2 CTU-13 Dataset 23
4.3 Graph Transform, Base Feature Extraction and Normalization Computation 23
4.4 Stand-alone Supervised Learning with F-Norm 24
4.5 Stand-alone SL with F-Norm and Balanced Input 25
4.6 Training Time of Stand-alone Supervised ML Classifiers 25
4.7 Stand-alone Supervised Learning against Previously Unknown Bot 26
4.8 k-Means Clustering with F-Norm 27
4.9 SOM Clustering with F-Norm 27
4.10 Supervised Learning with F-Norm 29
4.11 Supervised Learning with F-Norm on the Balanced Dataset 29
4.12 SOM with Newly Aggregated Dataset 30
4.13 Training Time of Supervised Classifiers on the Pruned Dataset 30
4.14 SOM Clustering without F-Norm 31
4.15 Pearson's Feature Correlation Matrix with F-Norm 33
4.16 SOM Clustering without IDW and ODW 33
4.17 Supervised Learning without IDW and ODW 33
4.18 SOM Clustering without BC 34
4.19 Supervised Learning without BC 34
4.20 Comparative Training and Testing Datasets 35

4.21 Supervised Learning with BotMiner Features without F-Norm 36
4.22 Flow-Based Supervised Learning 36
4.23 Supervised Learning with BClus Features and without F-Norm 38
4.24 Supervised Learning with BClus Features and Modified F-Norm 38
4.25 Supervised Learning with BClus Features and Two-way F-Norm 38
4.26 Supervised Learning with F-Norm 38
4.27 Supervised Learning with BClus Hybrid Features and F-Norm 39
4.28 BClus End-to-End Results 39
4.29 Accuracy of BotGM vs BotChase 40
4.30 Ensemble Learning with Graph-Based Features 41
4.31 Online Supervised Learning 42
4.32 Online Supervised Learning Using HAT 45

Chapter 1

Introduction

Undoubtedly, organizations are constantly under security threats, which not only cost billions of dollars in damage and recovery, but also often detrimentally affect their reputation. A botnet-assisted attack is a widely known threat to these organizations. According to the U.S. Federal Bureau of Investigation, "Botnets caused over \$9 billion in losses to U.S. victims and over \$110 billion globally. Approximately 500 million computers are infected each year, translating into 18 victims per second." The most infamous attack, called Rustock, infected 1 million machines, sending up to 30 billion spam emails a day [22].

More recently, Mirai knocked offline 900,000 users of Deutsche Telekom [14]. Furthermore, an attack launched on Microsoft Windows systems, called WannaCry (or WannaCrypt), resulted in a widespread hijack of data from more than 230,000 computers in 150 countries [30]. Undeniably, in the face of a cyber arms race, attackers constantly find clever ways to sabotage networks via botnets, most importantly via zero-day attacks [20]. Hence, it is imperative to defend against these botnet-assisted attacks.

1.1 Botnets

A botnet is a collection of bots, agents in compromised hosts, controlled by botmasters via command and control (C2) channels. A malevolent adversary controls the bots through a botmaster, which could be distributed across several agents that reside within or outside the network. Hence, bots can be used for tasks ranging from distributed denial-of-service (DDoS), to massive-scale spamming, to fraud and identify theft. While bots thrive for different sinister purposes, they exhibit a similar behavioral pattern when studied up-close.

The intrusion kill-chain [41] dictates the general phases a malicious agent goes through in-order to reach and infest its target. To fend off these botnets, intrusion detection systems (IDS) were developed.

1.2 Intrusion Detection Systems

Devising an intrusion detection system for bot detection is an active area of research that can be broadly divided into two groups based on the employed detection method: *signature-based* and *anomaly-based* [45]. Signature-based methods detect *pre-computed* hashes of existing malware binaries. Signature-based IDSs can scale well and efficiently detect known threats. Systems using this method can be deployed as an agent running on an end host or a gateway, which can examine binaries in transfer on-the-fly. However, as they rely on a database of known threats, signature-based approaches require frequent database updates and can be easily subverted by unknown or modified attacks, such as zero-day attacks and polymorphism [27, 45]. This undermines their suitability for bot detection.

Anomaly-based methods are widely used in bot detection, which overcome the limitation of the signature-based approach [20, 26]. They first establish a baseline of normal behavior for the protected system and model a decision engine. The decision engine determines and alerts any divergence or statistical deviations from the norm as a threat. *Machine learning* (ML) is an ideal technique to automatically capture the normal behavior of a system. The use of ML has boosted the scalability and accuracy of anomaly-based IDSs [20, 26].

1.3 Machine Learning

With an ascending advancement in technologies and deluge of flowing data, integrating AI into present day applications has become more of a necessity than a luxury. Aviation, autonomous transportation, drone navigation, sentient analysis and data mining are some of the renowned applications of current day AI research. However, it is not until recently that machine learning has become possible in fields which did not have the sufficient amount of data to have feasible predictive models.

For machine learning models and classifiers, there are a myriad of factors which determine their feasibility in production. Indicators such as false positives (FP) and false negatives (FN) are critical to the success of a system in an inverse proportional manner. For example, it is not acceptable for an intrusion detection system to have machine learning integrated with absurdly high FP and FN ratios [16]. Such systems are critical and very sensitive towards prediction outcomes. Allowing a bot to infiltrate a system as a FN would have immediate repercussions.

The most widely employed learning paradigms in ML include *supervised* and *unsupervised.* Supervised learning uses labeled training datasets to create models. It is employed to learn and identify patterns in the known training data. Typically, this approach is used to solve classification and regression problems. However, labeling is non-trivial and usually requires domain experts to manually label the datasets [20]. This is not only cumbersome but also error prone, even for small datasets. On the other hand, unsupervised learning uses unlabeled training datasets to create models that can discriminate between patterns in the data. This approach is most suited for clustering problems.

An important step prior to learning, or training an ML model, is feature extraction. These features act as discriminators for learning and inference, reduce data dimensionality, and increase the accuracy of ML models. The most commonly employed features in bot detection are flow-based (*e.g.*, source and destination IPs, protocol, number of packets sent and/or received, *etc.*). However, these features do not capture the topological structure of the communication graph, which can expose additional aspects of malicious hosts. In addition, flow-level models can incur a high computational overhead, and can also be evaded by tweaking behavioral characteristics *e.g.*, by changing packet structure [57].

Graph-based features, derived from flow-level information to reflect the true behaviour of hosts, are an alternate that overcome these limitations. We show that incorporating graph-based features into ML yields robustness against complex communication patterns and unknown attacks. Moreover, it allows for cross-network ML model training and inference.

1.4 Contributions

The major contributions of this thesis are as follows:

- We propose BotChase, an anomaly-based bot detection system which is protocol agnostic, robust to zero-day attacks, and suitable for large datasets.

- We show the limitations of stand-alone supervised learning. Therefore, we employ a two-phased ML approach that leverages both supervised and unsupervised learning. The first phase filters presumable benign hosts. This is followed by a second phase on the pruned hosts, to achieve bot detection with high precision.

- We use graph-based features in BotChase and evaluate various ML techniques. The graph-based features, derived from network flows, overcome severe topological effects. These effects can skew bot behavior in different networks, exacerbating ML prediction. Furthermore, these features allow to combine data from different networks and promote spatial stability [43] in the models.
- We compare the performance of our graph-based features with flow-based features from BotMiner [37] and BClus [33] in a prototype implementation of BotChase. Furthermore, we compare BotChase with the end-to-end system proposed for BClus.
- We evaluate the BotChase prototype system in an online setting that recurrently trains and tests the ML models with new data. This is crucial to account for changes in network traffic patterns and host behavior.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present a background on the intrusion kill-chain and bot detection, highlight limitations of the state-of-the-art and motivate the problem. The system design of BotChase is delineated in Chapter 3. Then, we evaluate the prototype in Chapter 4. Finally, Chapter 5 concludes with a summary of our contributions and exposes future research directions.

Chapter 2

Background

In this section, we present an overview of the intrusion kill-chain, followed by the state-of-the-art in bot detection and highlight their limitations.

2.1 Intrusion Kill-Chain

Conventional bot detection assumes successful intrusions and focuses on individual events. However, in recent sophisticated botnets, a single adversary campaign consists of multiple small, less detectable attacks. Detecting these bots can be challenging, as a single campaign may develop over time with multiple steps, each designed to thwart a defense and take place in different timelines. To cope with this problem, a widely adopted network-based method is detection of C2 channels. C2 occurs at the early stages of a botnet's lifecycle, thus its detection is essential to prevent malicious activities [35, 54, 62].

All adversarial attacks occurring in cyberspace have patterns that can be described as a chain of events—the intrusion kill-chain [41], depicted in Fig. 2.1. On a high-level, a bot starts with reconnaissance, observing and identifying a target in the network, and creating a weaponized payload. Weaponization of payloads take the form of malicious emails and attachments, which are delivered to the subject of interest. Exploitation starts after delivery, where the malevolent code gets triggered. While malicious code execution can be standalone, some malwares exploit applications on the subject's machine. This can range from OS-based bugs (*e.g.*, in RDP and PsExec) to application-based faults (*e.g.*, in live processes, such as Google Chrome, Mozilla Firefox and Microsoft Word). The bot

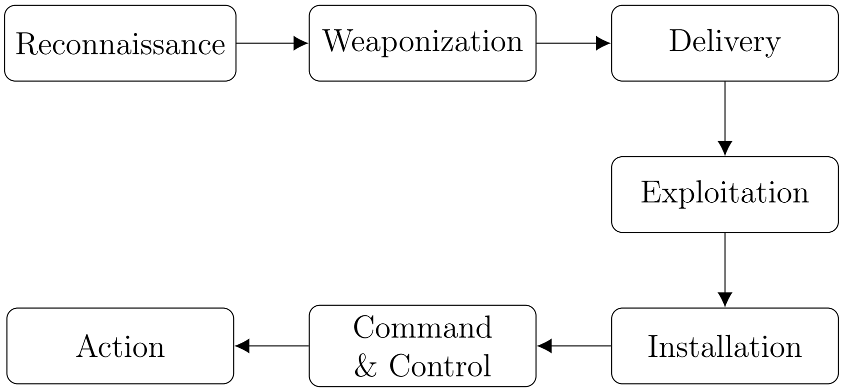

Figure 2.1: Intrusion kill-chain

then proceeds with the installation of a security back-door on the system, which permits external persistent connections. These connections are then leveraged for C2.

On top of C2, the kill-chain identifies another crucial host-based bot behavior, lateral movement (LM). LM includes reconnaissance, credential stealing, and infiltrating other hosts controlled by bots to move laterally within the network to gain higher privileges and fulfill adversarial objectives. It is less likely that adversaries launch a successful intrusion without LM, as the adversarial target is typically not directly reachable from the outside of a network. Thus, detecting bot propagation using LM is also advantageous as it contributes to early botnet detection and classification [41, 45]. Nevertheless, both C2 and LM leave network footprints, which is the focus of our network-based bot detection mechanism.

2.2 Bot & Botnet Detection

Bot(net) detection has been an active area of research that has generated a substantial body of work. Common botnet detection approaches are passive. They assume successful intrusions and focus on identifying infected hosts (bots) or detecting C2 communications, by analyzing system logs and network data, using signature- or anomaly-based techniques.

2.2.1 Signature-Based

Signature-based techniques have commonly been used to detect pre-computed hashes of existing malware in hosts and/or network traffic. They are also used to isolate internet relay chat (IRC) based bots by detecting bot-like IRC nicknames [36], and to identify C2-related DNS requests by detecting C2-like domain names [53]. More generally, signature-based techniques have been employed to identify C2 by comparison with known C2 communication patterns extracted from observed C2 traffic. It may also signify infected hosts by comparison with static profiles and behaviours of known bots [45]. Signature-based techniques owe their popularity to their ability to detect known threats in an efficient manner. However, they can be easily subverted by unknown or modified threats, such as zero-day attacks and polymorphism [27,45]. This undermines their suitability to detect sophisticated modern botnets.

2.2.2 Anomaly-Based

On the other hand, anomaly-based techniques use heuristics to associate certain behaviour and/or statistical features extracted from system or network logs, with bots and/or C2 traffic. C2 occurs at the early stages of a botnet's lifecycle, thus its detection is deemed essential to prevent malicious activities. Most existing anomaly-based C2 detection techniques are based on the statistical features of packets and flows [19,23,36,37,44,48,49,54,56,58,60–62]. Works like [19,44] are focused on specific communication protocols, such as IRC, providing narrow-scoped solutions. On the other hand, Botminer [37] is a protocol-independent solution, which assumes that bots within the same botnet are characterized by similar malicious activities and communication patterns. This assumption is an over simplification, since botnets often randomize topologies [45] and communication patterns as we observe in newer malware, such as Mirai [14]. Other works, such as [54,62], leverage ML and traffic-based statistical features for detecting C2 with low error rates. However, such techniques require that all flows are compared against each other to detect C2 traffic, which incurs a high computational overhead. In addition, they are unreliable, as they can be evaded with encryption and by tweaking flow characteristics [57]. Therefore, it is evident that a non-protocol-specific, more efficient, and less evadable detection method is desired. Based on the information used or the point of observation, anomaly-based detection can be classified as *host-based*, *network-based* and *hybrid*.

2.3 Anomaly-Based Botnet Detection Scopes

2.3.1 Host-Based

Anomaly detection at the host level is useful in early botnet detection. Host-based anomaly detection systems enable quick microscopic per-host analysis, and are suited for known and observable malware activities. Host-based anomaly detection is typically accomplished by examining system traces, such as event logs or system calls. Existing works show that host-based anomaly detection has better potential compared to signature-based detection [26, 34, 35]. However, as they require extensive monitoring of system activities, they consume host system resources *e.g.*, CPU cycles, memory, virtual machines. Consequently, they negatively impact user experience on the host.

Another downside of host-based anomaly detection is high error rates, since it experiences high false positives and false negatives when the established norm does not accurately represent the host behavior. Several studies, including [13] tend to resolve this problem by leveraging ML. However, though they indeed lower the error rates, their approaches are primarily designed for offline learning, requiring complete retraining of the ML model for each set of new input data. Since botnets rapidly evolve, it is imperative to leverage online, incremental learning to adapt the ML model to these changes.

2.3.2 Network-Based

Network-based anomaly detection collects and analyzes network traces, such as network packets and flow statistics. Detecting botnets by examining and monitoring network traffic has surfaced several times in the literature [37, 44, 54, 62]. Compared to host-based approaches, it offers a broader range of analysis, including traffic classification, botnet clustering, and network-wide anomaly detection with minimal to no performance degradation of the monitored systems. Karasaridis *et al.* [44] propose an algorithm for detection and characterization of botnets based on the analysis of flow data in the transport layer. However, some existing works, including [44] are focused on specific network protocols, such as IRC and P2P, providing narrow-scoped solutions.

2.3.3 Hybrid

Existing works suggest that performing different types of botnet detection techniques in conjunction improves accuracy and precision [27]. Hybrid botnet detection collectively applies host- and network-based techniques. EFFORT [55] integrates host- and network-based anomaly detection modules based on intrinsic characteristics of bots, and correlates detection results from them using ML. The evaluation shows that EFFORT detects malware activities with low false positive (FP) and minimal performance overhead. However, the detection methods employed by EFFORT are mostly heuristics that assume certain bot behaviors varying in degree. This can be easily subverted by adversaries with prior knowledge about the intrusion detection system *e.g.*, adversaries collaborating with insiders.

2.4 Graph-Based Approaches

Anomaly-based bot detection solutions that do not focus on detecting C2 per se, but rather identify bots by observing and analyzing their activities and behaviour, address some of the aforementioned issues. Graph-based approaches, where host network activities are represented by communication graphs, extracted from network flows and host-to-host communication patterns, are proposed in this regard [24, 25, 28, 31, 32, 38, 40, 42, 46, 50, 57, 59, 63, 64]. BotGM [46] builds host-to-host communication graphs from observed network flows, to capture communication patterns between hosts. A statistical technique, the interquartile method, is then used for outlier detection. Their results exhibit moderate accuracy with low false positives (FPs) based on different windowing parameters. However, BotGM generates multiple graphs for every single host. In other words, for every pair of unique IPs, a graph is constructed. Every node in the graph represents a unique 2-tuple of source and destination ports, with edges signifying the time sequence of communication. This entails a high overhead and will not scale for large datasets.

Chowdhury *et al.* [24] use ML for clustering the nodes in a graph, with a focus on dimensionality and topological characterization. Their assumption is that most benign hosts will be grouped in the same cluster due to similar connection patterns, hence can be eliminated from further analysis. Such a crucial reduction in nodes effectively minimizes detection overhead. However, their graph-based features are plagued by severe topological effects (*cf.*, Chapter 4). They use statistical means and user-centric expert opinion to tag the remaining clusters as malicious or benign. Nevertheless, leveraging expert opinion can be cumbersome, error prone and infeasible for large datasets. Recently, rule-based host clustering and classification [63, 64] have been proposed, where pre-defined thresholds are used to discriminate between benign and suspicious hosts. Unfortunately, relying on static thresholds makes the technique prone to evasion and less robust to ML-based outlier detection.

Graph-based approaches using ML for bot detection are intuitive and show promising results. In this thesis, we propose BotChase, an anomaly-, graph-based bot detection system, which is protocol agnostic *i.e.*, it detects bots regardless of the protocol. BotChase employs graph-based features in a two-phased ML approach, which is robust to zero-day attacks, spatially stable, and suitable for large datasets. It first employs unsupervised learning to reduce training data points for large datasets, followed by supervised learning to achieve bot detection with high precision.

Chapter 3

BotChase

3.1 Architecture

The BotChase system consists of three major components, as depicted in Fig. 3.1. These components pertain to data preparation and feature extraction, model training, and inference. In the following, we discuss these components.

3.2 Dataset Bootstrap

3.2.1 Flow Ingestion

The input to the system are bidirectional network flows. These flows are transformed to form a set T that contains 4-tuple flows $t_i = \{sip_i,\ srcpkts_i,\ dip_i,\ dstpkts_i\}$. Where sip_i is the source IP address that uniquely identifies a source host, $srcpkts_i$ quantifies the number of data packets sent by sip_i to dip_i, the destination host IP address. The number of destination packets, $dstpkts_i$, is the number of data packets sent by dip_i to sip_i.

Set A is a set of tuples that have exclusive source and destination hosts. That is, if multiple tuples have the same source and destination hosts, they are reduced to form an aggregated exclusive tuple $a_x \in A$, such that $a_x = \{sip_x,\ srcpkts_x,\ dip_x,\ dstpkts_x\}$. Therefore, if two tuples $t_i, t_j \in T$ have the same source and destination hosts *i.e.*, $sip_x = sip_i = sip_j$ and $dip_x = dip_i = dip_j$, then the number of source packets and the number of

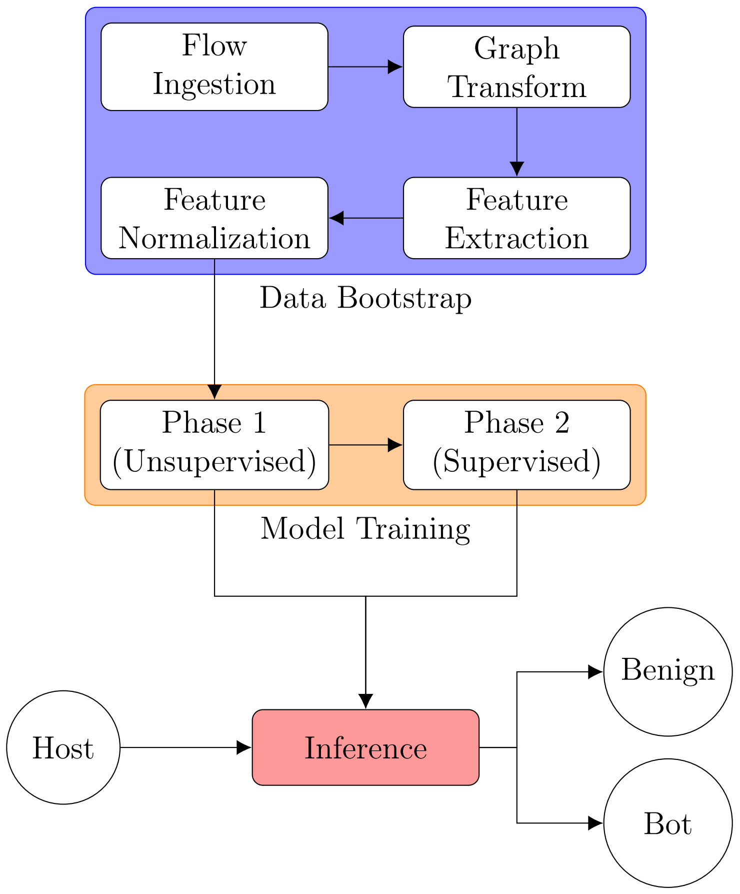

Figure 3.1: Components of the BotChase bot detection system

destination packets are aggregated in a_x, such that

$$srcpkts_x = \sum_{t_k \in T \;|\; sip_x = sip_k, dip_x = dip_k} srcpkts_k \tag{3.1}$$

$$dstpkts_x = \sum_{t_k \in T \;|\; sip_x = sip_k, dip_x = dip_k} dstpkts_k. \tag{3.2}$$

3.2.2 Graph Transform

The system creates a graph $G(V,\ E)$, where V is a set of nodes and E is a set of directed edges $e_{i,j}$ from node v_i to node v_j with weight $|e_{i,j}|$. The set of nodes V is a union of hosts from set A, such that

$$V = \bigcup_{\forall a_x \in A} \{sip_x \cup dip_x\}. \tag{3.3}$$

For every $a_x \in A$, there exist directed edges $e_{i,j}$ and $e_{j,i}$ from v_i to v_j and v_j to v_i, respectively, such that $sip_x = v_i$ and $dip_x = v_j$. Therefore,

$$E = \bigcup_{\forall a_x \in A} \{(sip_x,\ dip_x) \cup (dip_x,\ sip_x)\}. \tag{3.4}$$

The weights $|e_{i,j}|$ and $|e_{j,i}|$ of edges $e_{i,j}$ and $e_{j,i}$ are $srcpkts_x$ and $dstpkts_x$, respectively. Moreover, if there exists a reverse tuple $a_y \in A \mid dip_y = v_i,\ sip_y = v_j$, then $|e_{i,j}| = srcpkts_x\ +\ dstpkts_y$ and $|e_{j,i}| = dstpkts_x\ +\ srcpkts_y$.

3.2.3 Feature Extraction

BotChase creates the required *graph-based* feature set for the ML models. Features are intrinsic to the success of ML models that should genuinely represent and discriminate host behavior, especially that of bots. We leverage the following set of commonly used graph-based features.

- ***In-Degree (ID)*** and ***Out-Degree (OD)***—The in-degree, $f_{i,0}$, and out-degree, $f_{i,1}$, of a node $v_i \in V$ are the number of its ingress and egress edges, respectively.

$$\mathcal{F}(e_{i,j}) = \begin{cases} 1, & \text{if } e_{i,j} \in E \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

$$f_{i,0} = \sum_{v_j \in V,\ v_i \neq v_j} \mathcal{F}(e_{j,i}) \qquad \forall v_i \in V \tag{3.6}$$

$$f_{i,1} = \sum_{v_j \in V,\ v_i \neq v_j} \mathcal{F}(e_{i,j}) \qquad \forall v_i \in V \tag{3.7}$$

These features play an important role in the network behavior of a host. Although a higher ID for a host makes it a point of interest, it is often the case that nodes with a high ID offer a commonly demanded service. Therefore, observing ID alone may not signify malicious activity. For example, a gateway is a central point of communication in a network, but it is not necessarily a malicious endpoint. Intuitively, bots attempting to infect the network will tend to have a higher ID than benign hosts. Similarly, OD is also an intrinsic feature. Typically, in the reconnaissance stage of the intrusion kill-chain, bots attempt to survey the network. This mass surveillance can be captured via the OD.

- ***In-Degree Weight (IDW)*** and ***Out-Degree Weight (ODW)***—These features augment the ID and OD of the nodes in the graph. The in-degree weight, $f_{i,2}$, and the out-degree weight, $f_{i,3}$, of a node $v_i \in V$ is the sum of all the weights of its incoming and outgoing edges, respectively.

$$f_{i,2} = \sum_{v_j \in V,\ v_i \neq v_j,\ e_{j,i} \in E} |e_{j,i}| \qquad \forall v_i \in V \tag{3.8}$$

$$f_{i,3} = \sum_{v_j \in V,\ v_i \neq v_j,\ e_{i,j} \in E} |e_{i,j}| \qquad \forall v_i \in V \tag{3.9}$$

For a fine-grained differentiation, it is important to expose features that will eventually bring bots closer to each other, and discriminate bots from hosts. IDW and ODW features add another layer of information, further alienating the malicious hosts from the benign. Similar to ID, mass-data leeching bots will tend to expose a high IDW in the action phase of the intrusion kill-chain. Similarly, the ODW is the aggregate data packets a node has sent, which can potentially expose bots that mass-send payloads to hosts in a network.

- ***Betweenness Centrality (BC)***—The betweenness centrality of a node $v_i \in V$, inspired from social network analysis, is a measure of the number of shortest paths that pass through it, such that

$$f_{i,4} = \sum_{v_j, v_k \in V,\ v_i \neq v_j \neq v_k} \frac{\sigma_{v_j v_k}(v_i)}{\sigma_{v_j v_k}} \qquad \forall v_i \in V. \tag{3.10}$$

Where $\sigma_{v_j v_k}$ is the total number of shortest paths between node pairs $v_j,\ v_k \in V$, and $\sigma_{v_j v_k}(v_i)$ is the number of shortest paths that pass through v_i. This feature has a high computational overhead with $O(|V|.|E| + |V|^2 . \log |V|)$ time complexity [21]. However, it can alienate bots early on as they attempt their first connections. This is when the bots exhibit low IDW and ODW. Thus, it would be more favorable for the shortest paths in the network to pass through the host. Likewise, when the IDW and ODW increase, the BC of a node decreases immensely, as it is less favored for being included in shortest paths.

- ***Local Clustering Coefficient (LCC)***—Unlike BC, local clustering coefficient has a lower computational overhead, and it quantifies the neighborhood connectivity of a node $v_i \in V$, such that

  $$f_{i,5} = \frac{\sum_{v_j, v_k \in N_i,\ v_i \neq v_j \neq v_k} \mathcal{F}(e_{j,k})}{|N_i|(|N_i| - 1)} \qquad \forall v_i \in V \tag{3.11}$$

  Where N_i is neighborhood set for v_i, $\forall v_j \in N_i \mid e_{i,j} \in E \vee e_{j,i} \in E$. The LCC feature can play an important role in discriminating malicious host's behavior. Successfully infected hosts tend to exhibit a higher LCC, as bots often deploy collaborative P2P techniques, making adjacent host pairs strongly connected.

- ***Alpha Centrality (AC)***—Alpha centrality, also inspired from social network analysis, is a feature that generalizes the centrality of a node $v_i \in V$. AC extends the Eigenvector centrality (EC), with the addition that nodes are also influenced by external sources. These external sources can be user-defined, according to their graphical analysis technique. In EC, each v_i is assigned an influence score x_i, that is iteratively exchanged with adjacent nodes. In essence, EC is the relative weight of a node in the graph, such that connections to high-scoring nodes contribute more to the score of v_i. Hence, AC is given as

  $$f_{i,6} = \alpha A_i^T x_i + e_i \qquad \forall v_i \in V. \tag{3.12}$$

  Where A_i is the adjacency matrix, e_i is the external influence of node v_i, and α is influence factor that controls the bias in between external and internal sources. AC is important for the intermediate and terminal phases of the intrusion kill-chain. Early on, it may be negligible. However, as time progresses and bots perform more actions in the network, their AC will gradually increase, making it discriminative.

3.2.4 Feature Normalization (F-Norm)

Topological alterations can severely affect the host's behavior and pattern of communication in the graph. For example, in Fig. 3.2, g acts as a gateway for host $h2$ to communicate with the rest of the network (*i.e.*, hosts $h3$, $h4$ and $h5$). In this configuration, h_2 carries

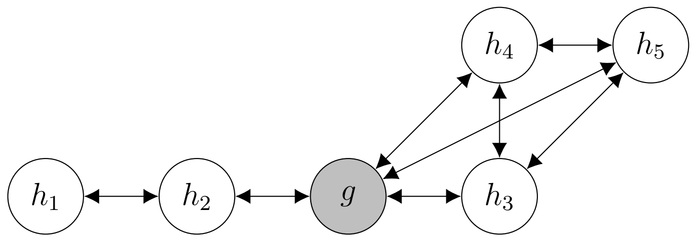

Figure 3.2: Example topology of benign hosts with a gateway

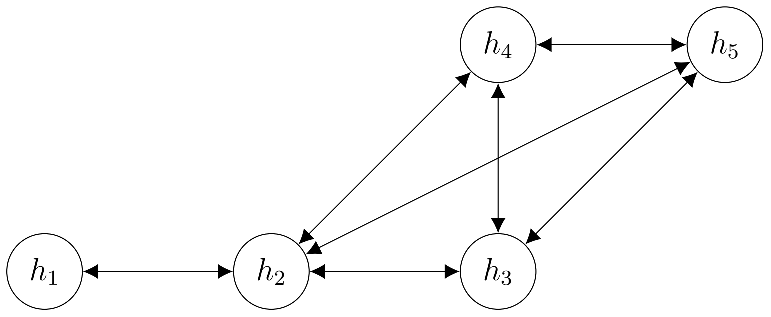

Figure 3.3: Example topology of benign hosts without a gateway

an ID of 2. In contrast, Fig. 3.3 shows the topology without a gateway, where h_2 communicates with other hosts in the network individually. This boosts the ID of host h_2 to 4. To alleviate this adverse effect of different network topologies, we normalize the above *base* features to include neighborhood relativity. To control the overhead of computing these *normalized* features, the neighborhood set N_i for $v_i \in V$ is restricted to a certain depth D. The mean of j features for v_i across its neighbors $v_k \in N_i$ are computed. Each feature for v_i is then normalized by incorporating neighborhood relativity. Thus, features relative to their neighborhood mean is given as

$$\mu_{i,m} = \frac{\sum_{v_k \in N_i} f_{k,m}}{|N_i|} \qquad \forall v_i \in V,\ 0 \le m \le j \tag{3.13}$$

$$f_{i,m} = \frac{f_{i,m}}{\mu_{i,m}} \qquad \forall v_i \in V,\ 0 \le m \le j. \tag{3.14}$$

After normalizing the features (with $D = 2$) for h_2 and h_4 with gateway, their IDs change from 2 to 0.8 and 3 to 1.1, respectively. Without the gateway, their IDs change from 4 to 1.6 and 3 to 1.1, respectively. As aforementioned, normalization attempts to make hosts of the same nature look similar, making the topological alterations less severe. Similarly, in situations where network data is recorded over varying time intervals, IDW and ODW tend to increase substantially with larger time intervals. By normalizing features, the effect of time also diminishes. Appendix B depicts a straightforward approach for implementing F-Norm.

3.3 Model Training

The model accepts graph-based features as input and learns to distinguish between malicious and benign hosts. Two learning phases in BotChase are explained below.

3.3.1 Phase 1

The first ML phase performs unsupervised learning (UL) to cluster the hosts. Generally, benign hosts exhibit *similar* behavior that can be gauged by the graph-based features. These hosts exhibit resembling patterns in data, such as sending (OD/ODW) and receiving (ID/IDW) similar number of packets [24]. Since BC, LCC and AC are directly affected by these traits, their influence can be similar for all benign hosts. Therefore, maximizing the size of the "singleton" benign cluster is crucial.

Not only does this phase act as a first filter for new hosts, but also significantly reduces the training data for the second phase. If a host is clustered into the benign cluster, then it is strictly benign. However, it is important to note that a malicious host can also be incorrectly clustered into a benign cluster, adversely affecting system performance. Therefore, although the system objective is to maximize the size of the benign cluster, it is essential to *jointly* minimize the number of bots that are co-located in this cluster.

Various UL techniques can be deployed in this phase. Some of these techniques include k-Means, Density-Based Spatial Clustering (DBScan) and Self-Organizing Map (SOM).

- ***k*-Means**—The k-Means clustering algorithm attempts to find an optimal assignment of nodes to k pre-determined clusters, such that the sum of the pairwise distance from the cluster mean is minimized. k-Means is static, it results in the same cluster composition for a given dataset across different runs of the algorithm, with the same number of clusters and iterations. Assume k is set to the cardinality of the label set. Idealistically, there should be a clean assignment of hosts to corresponding clusters. However, in reality, some benign hosts exhibit an outlier behavior. For example, network nodes which host webservers and public APIs will depict a huge amount of data and connections. Thus, ID, IDW, OD and ODW features will be affected. Therefore, depending on the characteristics of the dataset, altering k may adversely affect clustering performance.

- ***Density-Based Spatial Clustering (DBScan)***—Unlike k-Means, DBScan does not require the parameter k, the pre-determined number of clusters. In contrast, it computes the clusters and assignment of nodes according to a rigid set of density-based rules. DBScan requires a pair of parameters: (i) p, the minimum number of points required to be assigned as core points, and (ii) e, the minimum distance required to detect points as neighbors. DBScan classifies points as core, edge or noise, where core points must have p points in their neighborhood with a distance less than e. Otherwise, if the point is reachable via e distance from at least one of the core points, it is considered an edge. The remaining points are considered noise and are not clustered. In other words, points are not forcefully assigned to clusters as some points may just be noise. Therefore, DBScan is capable of detecting non-linearly separable clusters.
- ***Self-Organizing Map (SOM)***—A SOM is a special purpose artificial neural network that applies competitive learning instead of error-correction. It is frequently used for dimensionality reduction and clusters similar data. However, the notion of similarity in SOM is looser than that of k-Means and DBScan. In SOM, neurons are "pushed" towards the data points for a certain number of iterations. SOM uses the best matching unit to determine the winner neuron and updates its weights accordingly. Furthermore, it also applies a learning radius that affects all the other neurons, when a close-by neuron is updated. The number of neurons also plays an important role in clustering. Higher number of neurons results in the dispersion of nodes away from a single cluster. Importantly, the same logic applies to k-Means, hence the classifier with the best assignment must be selected, according to the objectives outlined in this phase.

3.3.2 Phase 2

Phase 1 separates the dataset between nodes that are inside and outside the benign cluster. All the nodes, ideally small, that reside outside the benign cluster are used as input to Phase 2 for further classification. Optimally, all the bots should be outside the benign cluster, regardless of whether or not they are co-located in the same cluster. Depending on the amount of hosts outside the benign cluster, the supervised learning (SL) classifiers used in this phase will exhibit different results.

The primary objective in this phase is to maximize recall. Recall is a measure of how many bots are recalled correctly *i.e.*, do not go unnoticed. It is proportional to the number of true positives (TPs) and inversely proportional to that of false negatives (FNs). Various SL classifiers can be deployed in this phase to achieve this objective, such as logistic regression (LR), support vector machine (SVM), feed-forward neural network (FNN) and decision tree (DT).

- ***Logistic Regression (LR)*** and ***Support Vector Machine (SVM)***—LR focuses on binary classification of its input, based on a sigmoid function. Input features are coupled with corresponding weights and fed into the function. Once a threshold p is defined, usually 0.5 for the logistic function, it establishes the differentiator between positive and negative points. Unlike LR, SVM is a non-probabilistic model for classification. It is not restricted to linearly separable datasets. There are various methods of computing SVM, including the renowned gradient-descent algorithm.
- ***Feed-forward Neural Network (FNN)***—FNNs are artificial neural networks that do not contain any cyclic dependencies. For a given feed-forward network with multiple layers, a feature vector is dispersed into the input layer, fed to the hidden layer of the network, and then to its output layer. While the input layer is constrained by the number of features exposed, the hidden and output layers are not. Every neuron may rely on a separate activation function that shapes the output. Popular activation functions for FNNs include identity, sigmoid, ReLU and binary step, among others. FNNs and the previously mentioned SL techniques are online classifiers. An online classifier is capable of incremental learning, as the weights associated with the deployed perceptrons are not static. This makes FNNs an attractive candidate for production-grade deployment.
- ***Decision Tree (DT)***—DTs rely heavily on Information Entropy (IE) and gain to conjure its conditional routing procedure. Generally, IE states how many bits are needed to represent certain stochastic information in the dataset. By using DT, information gain is maximized from the observed data and the taken path. After training a DT, newly observed data points can be predicted upon. However, unlike all the other classifiers, DTs are not online. That is, optimally retraining a DT must be done from scratch.

Recall the objective from Phase 1, *i.e.*, minimize hosts outside the benign cluster (HOB), while maximizing bots outside the benign cluster (BOB). This results in a minimal training dataset for Phase 2. Also, it is expected that the resultant training dataset from Phase 1 would be unbalanced, with a bias towards benign hosts. This may prove problematic for LR, SVM and FNN in achieving high recall rates.

3.4 Inference

Once the models are trained, they are deployed in the system to perform bot detection. Ideally, the system must allow for two modes of execution: (i) model (re)training, to adjust to the dynamics of the network, and (ii) inference, *i.e.*, for a given host predict whether or not it is a bot. In BotChase, the inference unfolds in two steps—presumable benign hosts get filtered out in Phase 1 as they get assigned to the benign cluster, while suspicious hosts assigned to a different cluster are further classified in Phase 2. Fig. 3.4 captures the inner workings of host classification. To preserve consistency, the system must synchronize and execute requests in order of observation.

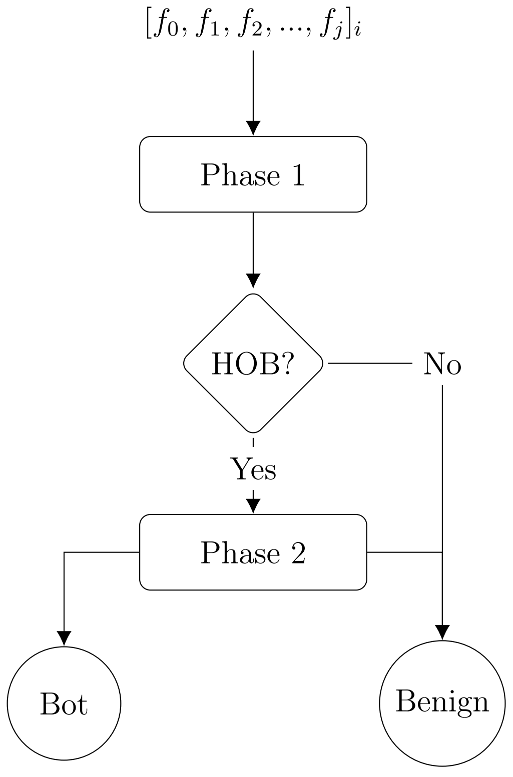

Figure 3.4: Flowchart of node classification with i nodes and j features

Chapter 4

Evaluation

We implement and evaluate the BotChase prototype bot detection system on a Hadoop cluster. In this section, we detail the experimental setup and the results of our evaluation.

4.1 Environment Setup

4.1.1 Hardware

The Hadoop cluster consists of a management node, a compute node and four data nodes. Table 4.1 delineates the configuration of these nodes. A 25Gbit and 10Gbit physical networks are deployed, interconnecting the nodes. The former network is primarily used for data and applications, while the latter one is for administration.

Table 4.1: Hardware Configuration of the Hadoop Cluster

Node	Configuration
1x Management Node	- 2x Intel Xeon Silver 4114 - 192 GB RAM
1x Compute Node	- 2x Intel Xeon Gold 5120 - 384 GB RAM
4x Data Node	- 2x Intel Xeon Silver 4114 - 192 GB RAM

4.1.2 Software

The software implementation is primarily based on Java. To ease dependency management, we incorporate Gradle [29]. JGraphT [51] graph library is used to construct the graph and extract graph-based features from network flows. Both Smile [47] and Encog [39] are used in tandem for ML. In order to support rapid prototyping, a custom in-house DataFrame (DataFrame4J) library has been developed. DF4J conforms to the incremental streaming paradigms, data streams with well-defined sources, stages and sinks. Furthermore, the underlying data structures are immutable, and all the basic stream-based transformations are available. More details about DF4J are provided in Appendix A.

4.2 Dataset

The evaluation of BotChase is based on the CTU-13 [33] dataset. CTU-13 comprises of 13 different subset datasets (DS) that include captures from 7 distinct malware, performing port scanning, DDoS, click fraud, spamming, *etc.* Every subset carries a unique network topology with a certain number of bots that leverage different protocols. Table 4.2 summarizes the dataset duration, number of flows and bots, and the type of bot in every subset. CTU-13 labels indicate whether a flow is from/to botnet, background or benign. Known infected hosts are labeled as bots, while the remaining hosts are tagged as benign. We leverage 12 datasets as base training data, while a single dataset, #9, is left out for testing purposes. This test dataset contains NetFlow data collected from a Neris botnet, 10 unique hosts labeled as bots, performing multiple actions including spamming, click fraud, and port scanning. We use this dataset configuration for training and testing, unless stated otherwise.

4.3 Performance

4.3.1 Graph Transform, Feature Extraction & Normalization

For every subset in the CTU-13 dataset, BotChase first ingests all the network flows, creates the graph, extracts base features and then normalizes them. For each dataset, Table 4.3 highlights the graph creation time *i.e.*, graph transform (GT), number of graph nodes ($|V|$), total runtime to extract only base BC feature and all base features (FE), and total runtime to normalize features (F-Norm).

Table 4.2: CTU-13 Dataset

DS	Duration	# Flows	Bot	# Bots
1	6.15	2824637	Neris	1
2	4.21	1808123	Neris	1
3	66.85	4710639	Rbot	1
4	4.21	1121077	Rbot	1
5	11.63	129833	Virut	1
6	2.18	558920	Menti	1
7	0.38	114078	Sogou	1
8	19.5	2954231	Murlo	1
9	5.18	2753885	Neris	10
10	4.75	1309792	Rbot	10
11	0.26	107252	Rbot	3
12	1.21	325472	NSIS.ay	3
13	16.36	1925150	Virut	1

Table 4.3: Graph Transform, Base Feature Extraction and Normalization Computation

DS	GT (seconds)	Nodes	BC (hours)	FE (hours)	F-Norm (seconds)
1	9	606829	24.12	24.121	11.3
2	6	441845	10.387	10.624	7.9
3	21	434489	9.463	9.713	13.755
4	5	185742	1.37	1.431	6.307
5	1	41548	0.057	0.06	0.556
6	3	107056	0.28	0.295	2.112
7	1	38081	0.021	0.022	0.488
8	13	383339	9.67	9.954	9.617
9	10	366881	8.677	8.97	7.879
10	7	197542	1.06	1.108	4.861
11	1	41809	0.055	0.057	0.627
12	2	94164	0.287	0.302	1.412
13	9	315343	3.667	3.852	6.824

It is evident that there is a non-linear relationship between BC and the number of nodes in the graph. Furthermore, the inconsistent variation between GT and the number of nodes is due to the differing time windows across datasets. Also, dataset #3 has a much higher number of flows than #2, which increases the runtime of graph creation. This is primarily due to the repeated modification of exclusive flow tuples in set A. The system then normalizes the base features, and Table 4.3 depicts its total runtime with $D = 1$. Evidently, normalizing features does not significantly increase the total runtime of the system. The largest runtime reported for the most complex dataset is 13.755 seconds.

4.3.2 Stand-alone SL

We start by highlighting the limitations of a stand-alone supervised learning approach. This consists of evaluating supervised ML classifiers, including DT, LR, SVM and FNN for bot detection. Each classifier employs graph-based normalized features and is trained on the entire training dataset. In our experiments, DT uses the Gini instance split rule algorithm, LR is used without regularization, and SVM uses the Gaussian kernel with a soft margin penalty of 1. Moreover, NN is configured to use cross entropy as an error function and 10 hidden layers of 1000 units each. Table 4.4 highlights the results, where LR and DT show meaningful classification. Both LR and DT classifiers result in a 100% recall, with 91% and 83% precision, respectively. With LR's superiority in precision, it *seems to be* the classifier of choice. The other classifiers were able to accurately classify all the benign hosts, but failed to identify any bots.

Table 4.4: Stand-alone Supervised Learning with F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	10	2	366869	0	100	83
LR	10	1	366870	0	100	91
SVM	0	0	366871	10	0	0
FNN	0	0	366871	10	0	0

To be fair to SVM and FNN, the input must be balanced. The bots to benign hosts ratio is quite minimal. This heavily affects the bias in the aforementioned classifiers to the benign hosts. To provide balance, we follow a mixed sampling approach. The benign hosts become subject to downsampling to a defined set size (1k, 2k, 5k, 10k). We then perform oversampling with replication for the bots. This provides a balance in between the labels. Given this technique, Table 4.5 depicts the corresponding results.

Table 4.5: Stand-alone SL with F-Norm and Balanced Input

Classifier	TP	FP	TN	FN	Recall	Precision
DT	10	13	366858	0	100	43
LR	10	7	366864	0	100	59
SVM	10	381	366490	0	100	2
FNN	10	250	366621	0	100	4

We find that all the classifiers are viable now. In particular, SVM and FNN are now able to classify bots, albeit with quite a few FPs. DT and LR remain the most promising models after the balancing changes, giving them an edge against the other classifiers.

We then evaluate the training time and robustness of the stand-alone classifiers, as depicted in Tables 4.6 and 4.7. DT requires the least training time of 4.9 seconds, which is in high contrast to the 58.2 seconds for LR. That is, DT requires only 8.4% of LR's training time for the entire training dataset. It is also essential for a bot detection system to detect bots that the classifier has never seen before *i.e.*, unknown or zero-day attacks. Therefore, to evaluate robustness to zero-day attacks, we change the selection of the training and testing datasets. We choose dataset #6 for testing, which has a unique bot that is not present in any other dataset. The remaining datasets are aggregated to form the training set, with 34 bots and a total of ≈3.1M hosts. Evidently, DT outperforms LR, which misclassifies a benign host, with a low precision of 50%.

Table 4.6: Training Time of Stand-alone Supervised ML Classifiers

Classifier	Training Time (s)
DT	4.9
LR	58.2
SVM	6832.3
FNN	93

Based on the above evaluations, LR outperforms DT in precision, while DT shows superior training time and robustness to unknown attacks. However, precision, training time and robustness are all crucial for our bot detection system. Can we achieve the best of all three? To investigate this, we set out to evaluate a two-phased system that employs an initial clustering phase (UL), followed by a classification phase (SL). We delineate its evaluation in the following subsections.

Table 4.7: Stand-alone Supervised Learning against Previously Unknown Bot

Classifier	TP	FP	TN	FN	Recall	Precision
DT	1	0	107055	0	100	100
LR	1	1	107054	0	100	50
SVM	0	0	107055	1	0	0
FNN	0	0	107055	1	0	0

4.3.3 Phase 1 (UL)

For Phase 1 in BotChase, we evaluate three UL techniques, namely *k*-Means, DBScan and SOM. However, DBScan results are inconclusive, where bots are co-located with benign hosts. DBScan is evaluated with varying minimum number of neighborhood points (minPts) and distance (ϵ). Multiple ϵ values are tested in the range of [10^{-5}, 10^{-4}, ..., 10^{5}]. Also, we infer ϵ values that correspond to the boundary of the bots themselves. We vary minPts in [1, 2, ..., 25] depending on the number of bots in the aggregated training dataset. However, maximal separation of bots from benign hosts could not be achieved with the tested parameters. In essence, DBScan does not produce a single, prevalent benign cluster. On the other hand, both *k*-Means and SOM show appreciable results, where SOM is trained with a learning rate of 0.7.

Tables 4.8 and 4.9 highlight the evaluation metrics, including number of clusters or neurons, number of hosts outside the benign cluster (HOB), percentage of hosts outside the benign cluster relative to the total number of hosts (HOB%), number of bots outside the benign cluster (BOB), and percentage of bots relative to the total number of bots (BOB%). Recall, the dataset #9 is removed for testing, which includes 10 hosts labeled as bots and ≈366K hosts. Also, ≈3.2M hosts from the remaining datasets are used to train the classifiers. In comparison to the number of clusters for *k*-Means, SOM is able to alienate its first bot outside the benign cluster with a lower number of neurons (9 *vs.* 16). With 81 neurons, SOM has a recall rate of 92% compared to the 42% of *k*-Means. However, *k*-Means catches up with 121 clusters. Nevertheless, SOM outperforms *k*-Means by maximizing the number of bots isolated with a smaller number of neurons.

With a cluster size of 100, *k*-Means alienates 21 bots, while having an outside host sum of 3028 for the remaining non-benign clusters. In contrast, SOM removes 23 bots from the benign cluster with an outside host sum of 3675. The very next *k*-Means cluster size *i.e.*, 121, boosts HOB from 3028 to 26935, while SOM remains at a close 3894. However, *k*-Means isolates three extra bots, yielding 24 BOB for 26935 HOB. That is, three extra bots were detected for a ≈23K increase in HOB. Recall, our objective in this phase is to

Table 4.8: *k*-Means Clustering with F-Norm

# of Clusters	HOB	HOB%	BOB	BOB%
4	5	0.0002	0	0
9	12	0.0004	0	0
16	36	0.0012	1	4
25	94	0.0033	6	24
36	170	0.0059	6	24
49	473	0.0164	8	32
64	1071	0.0371	10	40
81	1133	0.0392	10	40
100	3028	0.1049	21	87.5
121	26935	0.9327	24	96
144	27100	0.9384	24	96
169	27302	0.9454	24	96
196	27359	0.9474	24	96
225	28752	0.9956	24	96

Table 4.9: SOM Clustering with F-Norm

# of Neurons	HOB	HOB%	BOB	BOB%
4	10	0.0004	0	0
9	29	0.0010	1	4
16	49	0.0017	1	4
25	113	0.0039	6	24
36	286	0.0099	7	28
49	556	0.0193	8	32
64	1709	0.0592	10	40
81	3524	0.1222	23	92
100	3675	0.1274	23	92
121	3894	0.1350	23	92
144	27591	0.9647	24	96
169	27856	0.9740	24	96
196	28342	0.9912	24	96
225	28449	0.9950	24	96

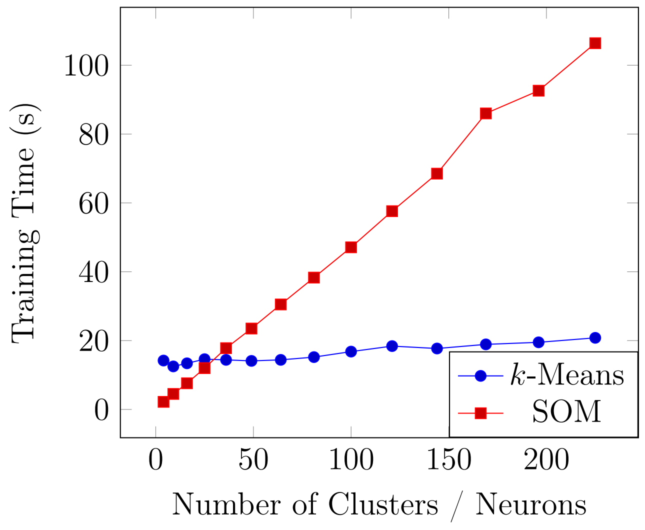

Figure 4.1: Comparison of SOM and k-Means with respect to training time

jointly minimize HOB while maximizing BOB. Therefore, SOM with 100 neurons becomes the natural choice.

With respect to runtime, k-Means mostly outperforms SOM, as depicted in Fig. 4.1. With 100 clusters, k-Means took 16.8 seconds to train, in comparison to 47.1 seconds of SOM. We speculate that SOM's ever increasing training time is attributed to how it updates the surrounding neurons. As the number of neurons increases, the density of their neighborhood also increases. Eventually, more neurons will tend to be within the threshold radius. Nevertheless, with recall being our top priority, we leverage SOM as UL classifier in Phase 1.

4.3.4 Phase 2 (SL)

The training set for Phase 2 is determined by the number of hosts outside the benign cluster in Phase 1. These are the relevant hosts for this phase, as hosts that are assigned in the benign cluster never make it to Phase 2. With a 10×10 (*i.e.*, 100 neurons) SOM and normalized features in Phase 1, the size of the dataset is significantly reduced. Therefore, we have 3675 HOB, including 23 bots, for further classification in Phase 2.

The DT classifier shows the best performance with the small dataset, as depicted in Table 4.10. It successfully detects *all* bots in the test dataset, with only a single FP out of the 366871 benign hosts. In contrast, all other classifiers are lackluster and unable to recall

Table 4.10: Supervised Learning with F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	10	1	366870	0	100	90.9
LR	0	0	366871	10	0	0
SVM	0	0	366871	10	0	0
FNN	0	0	366871	10	0	0

even a single bot from the dataset. We believe this is because all classifiers, except DT, rely on gradient-descent for error-correction. This implies that every single node in the dataset will affect the end-hypothesis function. Thus, with a dataset that is unbalanced, the hypothesis will be biased towards the benign hosts, which is the case for LR, SVM and FNN. Table 4.11 shows the results with a balanced training dataset in this current scenario. SVM and FNN remain unfazed, not being able to classify a single bot. However, DT shows a significant depreciation of its classification performance. Since this is the pruned dataset, the number of unique data points present is minimal and the imbalance isn't as significant as that observed in the previous standalone SL section. As DT incurs a significantly less training time than LR, we proceed with the vanilla pruned dataset in the following analyses.

Table 4.11: Supervised Learning with F-Norm on the Balanced Dataset

Classifier	TP	FP	TN	FN	Recall	Precision
DT	1	0	366871	9	10	100
LR	10	9	366862	0	100	53
SVM	0	0	366871	10	0	0
FNN	0	0	366871	10	0	0

Table 4.13 highlights the training time for the supervised classifiers. For Phase 1, a 10×10 SOM incurs a training time of 47.1 seconds, while DT has the lowest training time of 88 milliseconds in Phase 2. Thus, the aggregate training time for both phases is $\approx$47.2 seconds. This is an 11 seconds improvement over the 58.2 seconds observed for a stand-alone LR classifier [11].

Using dataset #6 for testing, the robustness test harbors more hosts for training in Phase 2. Most importantly, there are more BOB, yielding a higher ratio of bots to hosts outside the benign cluster, as depicted in Table 4.12. The robustness results are portrayed in Table 4.7. Though LR is able to recall the malicious bot while incurring only a single FP,

Table 4.12: SOM with Newly Aggregated Dataset

# of Neurons	HOB	HOB%	BOB	BOB%
100	3769	0.0011	32	94.12

Table 4.13: Training Time of Supervised Classifiers on the Pruned Dataset

Classifier	Training Time (ms)
DT	88
LR	2454
SVM	864
NN	3278

DT exhibits perfect results on this specific test dataset. It is able to detect the previously unknown bot, as well as correctly classify all the benign hosts. Therefore, with SOM selected for Phase 1 and DT for Phase 2, the system ensures minimal training time and robustness to unknown attacks, with high recall and precision.

4.3.5 Feature Normalization

Recall that aggregating datasets from different networks can negatively impact the base features, thus compromising system performance. Essentially, the topological structure of different networks affect the extracted graphical features, greatly skewing bot pattern and behavior. Thus, the intuition behind feature normalization is to make hosts, including bots, from different datasets look alike.

Table 4.14 showcases the crucial depreciation of the SOM results without normalizing graph-based features. For example, with 81 neurons, SOM with and without F-Norm scores 92% and 60% on BOB, respectively. On average, the results without F-Norm have a higher HOB. This intrinsic observation signifies the lack of similarity between hosts of the same category. For example, benign hosts from different networks are not co-located due to the stark differences in their features. Conversely, with F-Norm, similarly labeled hosts are more frequently co-located, yielding better BOB and HOB. Hence, normalized graph-based features significantly improve the spatial stability of ML in BotChase.

For 100 neurons, SOM with F-Norm results in 23 BOB and 3675 HOB. Without F-Norm, it results in 22 BOB and 8465 HOB, as shown in Figures 4.2 and 4.3. Thus, for the

Table 4.14: SOM Clustering without F-Norm

# of Neurons	HOB	HOB%	BOB	BOB%
4	8	0.0003	0	0
9	39	0.0014	0	0
16	689	0.0239	0	0
25	935	0.324	0	0
36	2280	0.0790	9	36
49	3792	0.1315	11	44
64	4207	0.1459	14	56
81	6721	0.2333	15	60
100	8465	0.2940	22	88
121	12923	0.4495	24	96
144	20780	0.7248	24	96
169	22607	0.7890	24	96
196	23714	0.8280	24	96
225	42125	1.4803	24	96

same number of neurons, feature normalization was able to maximize BOB, while minimizing HOB. Therefore, we choose 100 neurons with F-Norm as our primary configuration for SOM.

4.3.6 Feature Engineering

It is important to gauge the significance and impact of the chosen graph-based features on bot detection. Albeit different feature combinations may impact the results, are all features necessary? Table 4.15 shows the Pearson's feature correlation matrix for the normalized graph-based features. At a glance, we can determine that the first five features are highly correlated, with a correlation close to or greater than 0.9. Therefore, feature combinations that exclude some of these features may not exacerbate classification accuracy. On the other hand, the last two features are highly uncorrelated, with LCC being the least correlated. Hence, we start with removing IDW and ODW, which decreases the benign cluster size but results higher on BOB, as shown in Table 4.16. However, Table 4.17 shows the lackluster performance of the SL classifiers when we eliminate IDW and ODW features. Precision drops to 52.6% for DT from 90.9% (*cf.*, Table 4.10). Also, LR now misclassifies two benign hosts as bots.

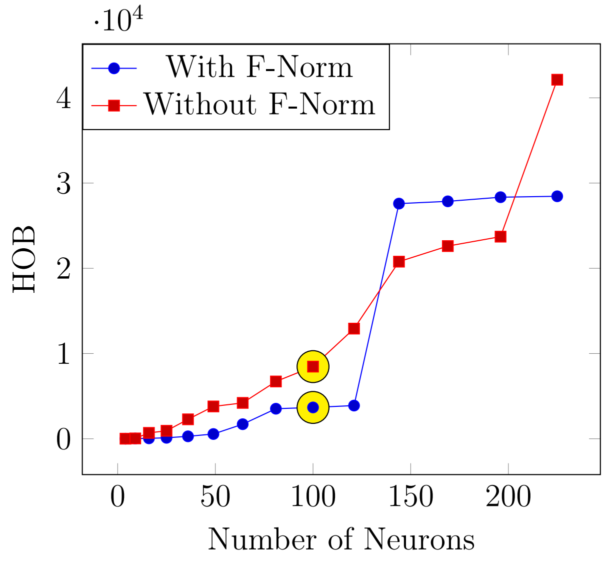

Figure 4.2: Number of hosts outside the benign cluster (HOB) assigned by SOM with and without feature normalization

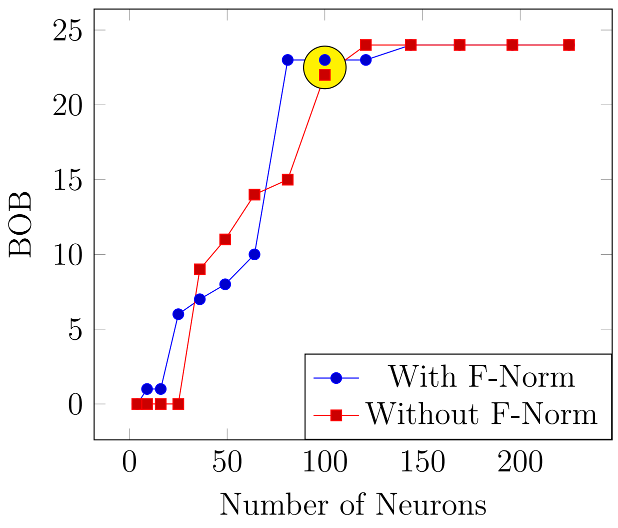

Figure 4.3: Number of bots outside the benign cluster (BOB) assigned by SOM with and without feature normalization

Table 4.15: Pearson's Feature Correlation Matrix with F-Norm

	ID	IDW	OD	ODW	BC	LCC	AC
ID	1	0.99	0.92	0.95	0.96	0.03	0.32
IDW	0.99	1	0.91	0.96	0.97	0.03	0.33
OD	0.92	0.91	1	0.89	0.90	0.08	0.37
ODW	0.95	0.96	0.89	1	0.97	0.04	0.43
BC	0.96	0.97	0.90	0.97	1	0.01	0.46
LCC	0.03	0.03	0.08	0.04	0.01	1	0.01
AC	0.32	0.33	0.37	0.43	0.46	0.01	1

Table 4.16: SOM Clustering without IDW and ODW

# of Neurons	HOB	HOB%	BOB	BOB%
100	27404	0.958	24	96

Table 4.17: Supervised Learning without IDW and ODW

Classifier	TP	FP	TN	FN	Recall	Precision
DT	10	9	366862	0	100	52.6
LR	0	2	366869	10	0	0
SVM	0	0	366871	10	0	0
FNN	0	0	366871	10	0	0

A weakness of the chosen features is the runtime of BC. For the first dataset, it took over 24 hours to compute BC. This will render any effort to expedite the learning process in vain. However, removing BC from the feature set adversely affects the performance of DT, but not for SOM, as depicted in Tables 4.18 and 4.19. SOM without BC performs identical to the use of the entire feature set. On the other hand, DT's precision is affected by the removal of BC, but it is better than that of the removal of IDW and ODW from the feature set. While the precision deteriorated, *only* 6 and 9 benign hosts were misclassified out of the ≈367K hosts with the removal of BC and IDW/ODW, respectively. This reinforces the correlation matrix *i.e.*, having these features the most correlated. Since recall and precision are sought after metrics in BotChase, it is important to include these features for training and testing classifiers.

Table 4.18: SOM Clustering without BC

# of Neurons	HOB	HOB%	BOB	BOB%
100	3622	0.125	23	96

Table 4.19: Supervised Learning without BC

Classifier	TP	FP	TN	FN	Recall	Precision
DT	10	6	366865	0	100	62.5
LR	0	0	366869	10	0	0
SVM	0	0	366871	10	0	0
FNN	0	0	366871	10	0	0

4.4 Comparative Analysis

Given the modularity of BotChase, in-place substitution of modules is possible. For example, rather than having graph-based features, the system can leverage flow-based features, while maintaining the two-phased bot detection. Therefore, we first compare the performance of our graph-based features with flow-based and hybrid features from BotMiner and BClus in BotChase. Furthermore, we compare BotChase with the end-to-end system proposed for BClus. Finally, we provide a rough comparison against BotGM. For a fair comparison, we reselected the training and testing datasets. This conforms to the selection in [33], where the test dataset contains multiple bot types and different network topologies. The dataset selection of these comparisons is depicted in Table 4.20.

Table 4.20: Comparative Training and Testing Datasets

Purpose	Dataset
Training	3,4,5,7,10,11,12,13
Testing	1,2,6,8,9

BotMiner aggregates flows based on their source IP, protocol, destination IP and its corresponding port. These aggregated flows, called C-Flows, are processed in a time epoch that lasts up to a full day of flow capture. After flows are aggregated, 52 features are extracted by first mapping every C-Flow into a discrete sample distribution of four random variables: (i) total number of packets sent and received in a flow, (ii) average number of bytes per packet, (iii) total number of flows per hour, and (iv) average number of bytes per second. These random variables are then binned into 13 slots according to pre-defined percentiles. Through this technique, every variable is converted into a vector of 13 elements, totaling 52 features per C-Flow.

BClus undertakes a similar clustering approach by grouping flows into instances. These instances are identified by *unique* source IPs in a certain time window. Each instance is represented using 7 features: (i) source IP address, (ii) number of distinct source ports, (iii) number of distinct destination ports, (iv) number of distinct destination IPs, (v) total number of flows, (vi) total number of bytes, and (vii) total number of packets. These instances are then clustered using Expectation Maximization (EM). More features are then extracted from the clusters themselves to aid JRip, a propositional rule learner, in their labeling. These features include: (i) total number of instances, (ii) total number of flows, (iii) number of distinct source IP addresses, and (iv) the average and standard deviation amount of distinct source ports, distinct destination IPs, distinct destination ports, number of flows, number of bytes, and number of packets. Hence, every cluster exhibits 15 features, which are then used by JRip. After training, JRip is capable of classifying each cluster as malicious or benign. Ground truth label of clusters is determined through a bot flow threshold that is varied to find the best JRip model.

BotGM uses graph-based outlier detection to detect suspicious flow patterns. It starts off with extracting events, *i.e.*, converting flows into a key-value entry. The key represents the source and destination IPs while the value represents the source and destination ports. Then, a sequence is extracted that tracks the source and destination port variations of two unique IPs. A directed graph is then extracted from these variations, with vertices representing a port 2-tuple. The aggregate graphs are then mined for outlier detection. They use the graph edit distance to gauge how different a graph is from another. The inter-quartile method is then used to detect outliers.

4.4.1 BotMiner Flow-Based vs. Graph-Based Features

We start with the aforementioned flow-based features from BotMiner in BotChase. Table 4.21 showcases the outcome of classifying flows using BotMiner features, where only LR is able to detect a few malicious flows, misclassifying the majority of benign and malicious flows. To compare, we convert our host classification into flow classification in Table 4.22. With a recall (RCL) of 0.02% and a precision (PRC) of 16.28%, BotMiner features perform poorly against the graph-based features. The latter scores 81.57% and 99.51% on recall and precision, respectively. However, in comparison to host classification (*cf.*, Table 4.26), the precision is significantly higher as the flows originating from the identified FP hosts were relatively minimal. Likewise, the different number of flows per host may result in a lower or higher recall rate. While LR and DT highlight similar host classification results, DT is the more favorable flow classifier as it does not misclassify prominent benign hosts.

Table 4.21: Supervised Learning with BotMiner Features without F-Norm

Classifier	TP	FP	TN	FN	RCL	PRC
DT	0	1	2550094	34966	0	0
LR	7	36	2550059	34959	0.02	16.28
SVM	0	0	2550095	34966	0	0
FNN	0	0	2550095	34966	0	0

Table 4.22: Flow-Based Supervised Learning

Classifier	TP	FP	TN	FN	RCL	PRC
DT	211452	1037	20145929	47776	81.57	99.51
LR	67006	59657	20087309	192222	25.85	52.9
SVM	0	0	20146966	259228	0	0
FNN	0	0	20146966	259228	0	0

4.4.2 BClus Flow-Based vs. Graph-Based Features

Implementing BClus features in BotChase was an incremental process. Alongside choosing the optimal number of EM clusters, F-Norm had a major impact on the results. Unlike

BotMiner, BClus strictly classifies instances pertaining to unique source IPs. Using a large time window that fits the entire test dataset, an instance becomes a full representation of host behavior. As depicted in Table 4.23, our preliminary implementation of BClus without F-Norm had a zero recall rate across all the trained supervised classifiers. Therefore, to improve BClus we modified F-Norm to process all the hosts.

Recall that BClus extracts features for source IPs only, thus features of destination IPs are missing from our data pipeline. The data pipeline only consists of hosts that have had their features extracted based on previous aggregations. Hence, a direct application of F-Norm, as implemented in BotChase, results in missing data elements for hosts that are present in the graph but not in the data pipeline. Therefore, we first naïvely modify F-Norm to handle non-existent data points with a zero vector. This improves the results of LR, which now captures a single bot out of 14, while the remaining classifiers still perform poorly, as depicted in Table 4.24. This comes with no surprise, as a zero vector still affects the relative values of the host features.

Finally, we transform BClus to account for both source and destination IPs as instances. This solves the issue with F-Norm, since all unique IPs in the network are mapped into a corresponding data point and host node in the graph. Using this two-way analysis, Table 4.25 shows an appreciable improvement over the former iterations. DT manages a jump from 0% to 64.29% in recall and 81.82% in precision. However, even after the improvements to BClus features, it significantly underperforms our graph-based features. Table 4.26 showcases the performance of the graph-based features on the new dataset selection. It exhibits convincing results for both DT and LR, with high rates of 85.71% and 80% on recall and precision, respectively.

Interestingly, DT and LR have similar performance on host classification yet different on flow classification. Although both classifiers agree metrics-wise, the underlying sets of hosts tagged as bot or benign are different. Hence, it is possible to combine the classifiers into a single decision making entity. A simple rule to boost our classification results could be to flag a host as a bot if at least one classifier concurs. While this can potentially increase the recall rate, precision is expected to decline as FPs are aggregated across classifiers.

4.4.3 BClus Hybrid vs Graph-Based Features

So far, we have experimented with both BClus' flow-based features and BotChase's graph-based ones. How would they fair if paired in the same ML model? With 6 flow-based and 7 graph-based features, every unique host in the network depicts 13 features to be processed by the system. In this experiment, we resort to BotChase as our base architecture, only

Table 4.23: Supervised Learning with BClus Features and without F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	0	10	1651678	14	0	0
LR	1	3	1651685	13	7.14	25
SVM	0	0	1651688	14	0	0
FNN	0	0	1651688	14	0	0

Table 4.24: Supervised Learning with BClus Features and Modified F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	0	1	1651687	14	0	0
LR	1	4	1651684	13	7.14	20
SVM	0	0	1651688	14	0	0
FNN	0	0	1651688	14	0	0

Table 4.25: Supervised Learning with BClus Features and Two-way F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	9	2	1905934	5	64.29	81.82
LR	7	8	1905928	7	50	46.67
SVM	0	0	1905936	14	0	0
FNN	0	0	1905936	14	0	0

Table 4.26: Supervised Learning with F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	12	3	1905933	2	85.71	80
LR	12	3	1905933	2	85.71	80
SVM	0	0	1905936	14	0	0
FNN	0	0	1905936	14	0	0

changing the amount of features computed per unique source IP. Table 4.27 shows the results of the corresponding analysis. When compared to the graph-based baseline, LR was able to detect an additional bot while incurring 6 additional FPs. This boosts the recall rate to 92.86% while bringing down the precision to an unimpressive 59.09%. On the other hand, the metrics for DT did not change.

Table 4.27: Supervised Learning with BClus Hybrid Features and F-Norm

Classifier	TP	FP	TN	FN	Recall	Precision
DT	12	3	1905933	2	85.71	80
LR	13	9	1905927	1	92.86	59.09
SVM	0	0	1905936	14	0	0
FNN	0	0	1905936	14	0	0

4.4.4 BClus End-to-End vs. BotChase

As part of our comparative analysis, we also implement the entire BClus approach and perform classification on their pre-defined selection of datasets. After optimizing the number of EM clusters and JRip folds, we converge on the best possible configuration. With 12 EM clusters, BClus performs a record 100% recall rate on the test datasets, with an extremely poor precision rate of 6%. Additionally, Table 4.28 shows a high cost of 423.9 seconds to solely train EM. BClus significantly underperforms BotChase, which scores a minimal 21.9 seconds of total training time and high rates of $\geq$ 80% on both recall and precision.

Table 4.28: BClus End-to-End Results

EM Clusters	Training Time (s)	TP	FP	TN	FN	RCL	PRC
12	423.9	14	219	1651469	0	100	6

4.4.5 BotGM vs. BotChase

As the final comparison, we compare BotChase to BotGM. Both systems, attempt to catch malicious and outlier behavior using graph methodologies. In [46], BotGM shows an impressive accuracy of up to 95% on one of the test datasets. Can BotChase do better? We perform a leave-one-out approach, targeting one test dataset and taking the rest for training. The results are depicted in Table 4.29.

Table 4.29: Accuracy of BotGM vs BotChase

Algorithm	Scenario ID				
	1	2	6	8	9
BotGM	0.91	0.78	0.95	0.89	0.83
BotChase	0.98	0.97	0.94	0.84	0.99

BotChase scores an aggregate high of 99% with a low of 84%. BotGM shows a marginal improvement for scenarios 6 and 8, but lags behind in the remaining scenarios.

4.5 Ensemble Learning

When multiple classifiers are used together for prediction, they are categorized under ensemble learning. Earlier, we have showed that DT and LR classify 12 out of the 14 bots with 3 FPs incurred. However, once we translated the results into flows in Table 4.22, the numbers were completely different. This shows that the classifiers have successfully predicted a different set of bots. In essence, having them both act as a single classifier conservatively should result in a higher recall rate. We experimented with both of our two successful classifiers working in tandem, DT and LR. Our conservative approach is depicted in Alg. 1.

Algorithm 1 Classifying hosts with an ensemble

```
$p1 \leftarrow dt.predict(host)$
$p2 \leftarrow lr.predict(host)$
**if** $p1 = 1$ **or** $p2 = 1$ **then**
    **return** 1
**else**
    **return** 0
**end if**
```

The results depicted in Table 4.30 reflect our hypothesis. The ensemble is able to correctly predict 13 bots, boosting the recall rate to 92.86%. However, given the conservative approach, an FP incurred in any classifier will be accounted for. In this case, the classifiers on their own predicted different hosts as FPs, resulting in an aggregate of 6 FPs. While this approach has brought down the precision of our second phase to 68.52%, it enabled us to successfully classify one more bot.

Table 4.30: Ensemble Learning with Graph-Based Features

Classifier	TP	FP	TN	FN	Recall	Precision
DT + LR	13	6	1905930	1	92.86	68.42

4.6 Analysis in an Online Setting

While our infrastructure fully supports streaming, our initial evaluation of BotChase was carried out on the entire CTU-13 dataset as a single data batch. However, it will be crucial to constantly retrain the ML models to account for changes in network traffic patterns and host behavior. This will indeed be fundamental in realizing autonomic security management [15]. Most ML models are trained offline, and retrained from scratch. When this proves to be computationally intensive and time consuming, it prohibits the aspects of online deployment. The ability to retrain the model as new data becomes available, is fundamental to accommodate ML models' boundary changes after deployment.

To evaluate BotChase in an online setting, we iteratively retrain the ML models with new data and test the models. The training and testing datasets are inferred from Table 4.20. We assess the different ML models as we increase the amount of flows ingested into the system, from the aggregated training dataset. Therefore, a time window (w) of 5 minutes spawns N time windows, where N is the number of training sub-datasets. In the current dataset, a 5 minute time window is equivalent to 40 minutes of ingested data. Furthermore, the smallest training dataset has 15 minutes of flows, while the largest has 4011 minutes. This is due to the nature of network flows. For example, the number of flows captured in one minute of a DDoS attack will outweigh that of an idle network. Therefore, the percentage of flows ingested into the system will increase in a non-linear manner. Also, it is ideal for the time window to be smaller than the smallest dataset.

With $w = 5$ and the same aggregated test dataset to assess the different ML models, we expose the elapsed time (t), percentage of flows ingested online (Ing.), and the classification metrics in Table 4.31.

Table 4.31: Online Supervised Learning

i	Time (mins)	TP	FP	TN	FN	Ing. (%)	RCL	PRC
1	5	0	0	1905936	14	0.00	0.00	0.00
2	10	0	0	1905936	14	0.00	0.00	0.00
3	15	0	21	1905915	14	1.10	0.00	0.00
4	20	0	22	1905914	14	2.24	0.00	0.00
5	25	0	23	1905913	14	2.28	0.00	0.00
6	30	8	19	1905917	6	3.61	57.14	29.63
7	35	8	19	1905917	6	3.61	57.14	29.63
8	40	8	19	1905917	6	3.61	57.14	29.63
9	45	8	19	1905917	6	3.62	57.14	29.63
10	50	8	19	1905917	6	3.62	57.14	29.63
11	55	8	3	1905933	6	4.26	57.14	72.73
12	60	0	14	1905922	14	5.07	0.00	0.00
13	65	1	5	1905931	13	5.87	7.14	16.67
14	70	1	3	1905933	13	7.15	7.14	25.00
15	75	12	5	1905931	2	10.32	85.71	70.59
16	80	12	2	1905934	2	10.73	85.71	85.71
17	85	12	2	1905934	2	11.86	85.71	85.71
18	90	12	2	1905934	2	13.53	85.71	85.71
19	95	12	2	1905934	2	15.81	85.71	85.71
20	100	12	2	1905934	2	16.72	85.71	85.71

- ***At*** t ***= 5 mins***, only a few flows are ingested into the system. There are a total of 9697049 flows in the dataset, while the percentage of ingested flows is minuscule, hence it shows as 0.00%. Since DT is the second phase classifier, having only benign data points does not suffice. Therefore, when the system detects this edge case, it defaults to classifying all data points as benign.

- ***At*** t ***= 15 mins***, 1.1% of flows are now ingested into the system. Given the early condition of hosts, it is expected to have high false alarms. In this case, the system results in 21 FPs and 14 FNs. The first bot flows also start to appear at this time.

- ***At*** t ***= 30 mins***, we reach 3.61% of ingested flows. The number of TPs detected improves from 0 to 8, leading to the first model that is able to detect malicious hosts. Hence, it takes exactly 15 minutes for the system to exhibit its initial TPs, after the first bot flows start to appear. At this point, a baseline of malicious behavior is formed, matching the classification system's bot profile.

- ***At*** t ***= 60 mins***, the model's performance declines. The amount of flows ingested is 5.07%, while the model incurs 14 FPs and 14 FNs. At this stage, the malicious hosts have camouflaged their features through benign communication and exhibit benign host-like behavior.

- ***At*** t ***= 75 mins***, the system reaches a state which is close to that of ingesting the full training dataset. At only 10.32%, it is able to detect 12 out of 14 bots. This achieves a recall of 85.71%. However, there are 2 additional FPs over the standard system, resulting in a 70.59% precision.

- ***At*** t ***= 80 mins***, the system reaches its best outcome with only a tenth of the data ingested. Interestingly, one FP is shaved off the standard system metrics. As aforementioned, having more or less training data points may alter the constructed bot and benign profiles. A bot may initially behave like a benign host. Once more flows are ingested, a bot behavior can become more prominent and anomalous to that of benign. However, with further flows, the bot may also be able to disguise itself as benign. Since BotChase depends on graph-based features, an additional flow either adds weight to an existing edge or creates one. This will effectively change the neighbouring malicious and benign host features used in training, thus skewing performance.

Fig. 4.4 showcases the different training and testing times observed as the time window progresses. Since we are dealing with flows in this scenario, the training time incorporates the time needed to extract the features from the flows. This requires building the graph incrementally with the given flows and extracting the features of every node. The plot takes on a positive slope for the training time, while the testing time remains at a plateau of around 9 seconds. At 75 minutes, the training time jumps to 227 seconds, slightly incrementing to 244 seconds at 80 minutes. Even at this optimal mark, the training time remains $< w$. This implies that the system is ready for classification prior to the processing of the next window interval.

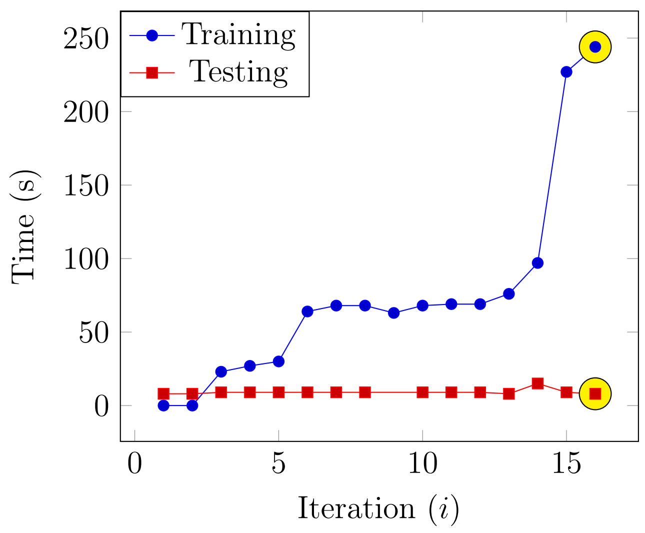

Figure 4.4: The variation of the training and testing time as time progresses

Some variations in performance are observed beyond $t = 100$ minutes. This is because over-fitting can occur that contributes to additional FPs and FNs. The system reaches a steady state at $t = 270$ minutes, when only 37.22% of the flows are ingested. The sooner an online system becomes effective, the better. The amount of ingested flows and required training time dictates the total system overhead over time. The second phase supervised classifier used is DT, and it uses the ID3 algorithm [52]. This specific algorithm does not permit incremental learning, thus the tree is reinitialized at every epoch. Obviously, this is not efficient. Can we do better?

We set out to experiment with a very fast decision tree (VFDT) variation, the Hoeffding Adaptive Tree (HAT) [17]. This tree algorithm can be trained on the fly as new data arrives. It maintains old branches, making sure it prunes them when they become obsolete. An internal naive bayes selector is used when the tree leaves have the same values but different labels. Using MoA's HAT [18], we set out to experiment with BotChase using the Hoeffding algorithm. The results are depicted in Table 4.32.

Table 4.32: Online Supervised Learning Using HAT

i	Time (mins)	TP	FP	TN	FN	Ing. (%)	RCL	PRC
1	5	0	0	1905936	14	0.00	0.00	0.00
2	10	0	0	1905936	14	0.00	0.00	0.00
3	15	0	0	1905936	14	1.10	0.00	0.00
4	20	0	0	1905936	14	2.24	0.00	0.00
5	25	0	0	1905936	14	2.28	0.00	0.00
6	30	0	0	1905936	14	3.61	0.00	0.00
7	35	0	0	1905936	14	3.61	0.00	0.00
8	40	0	44	1905892	14	3.61	0.00	0.00
9	45	0	40	1905896	14	3.62	0.00	0.00
10	50	1	20	1905916	13	3.62	7.14	4.76
11	55	7	25	1905911	7	4.26	50.00	21.88
12	60	7	18	1905918	7	5.07	50.00	28.00
13	65	7	20	1905916	7	5.87	50.00	25.92
14	70	9	36	1905900	5	7.15	64.29	20.00
15	75	9	36	1905900	5	10.32	64.29	20.00
16	80	9	24	1905912	5	10.73	64.29	27.27
17	85	9	13	1905923	5	11.86	64.29	42.85
18	90	13	6	1905930	1	13.53	92.85	68.4
19	95	13	3	1905933	1	15.81	92.85	81.25
20	100	13	3	1905933	1	16.72	92.85	81.25

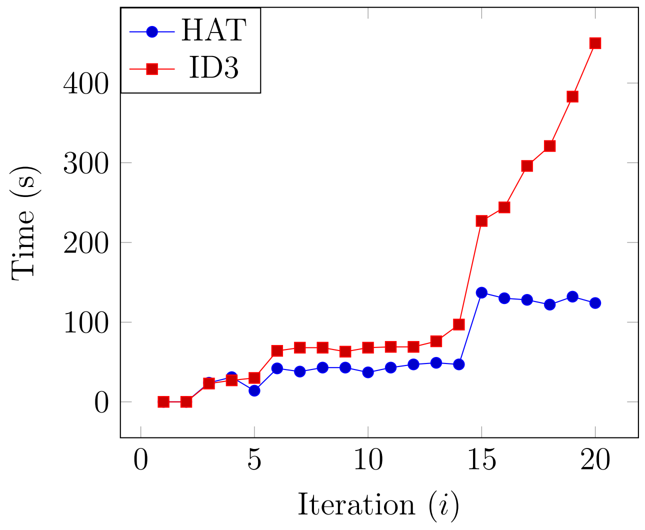

Figure 4.5: The variation of the training time of HAT vs ID3

We ran the algorithm with the same settings as that of the former experiment. Table 4.32 shows the results.

- ***At t = 5 mins***, HAT should be very similar to ID3 as both trees have the same exact data. This is exposed with minimal flow ingestion and zero recall and precision rates.
- ***At t = 40 mins***, 3.61% of flows are now ingested into the system. This is the first epoch at which the system attempts to positively label hosts. However, this results in 44 FPs.
- ***At t = 55 mins***, we reach 4.26% of ingested flows. At this point, the system was able to successfully detect 7 out of the 14 bots. Compared to ID3, the system maintains its momentum to converge to a state that is capable of detecting bots. While the number of FPs have decreased, 20 still remain.
- ***At t = 70 mins***, the system was able to successfully detect 2 more bots, incurring alot more FPs along the way.
- ***At t = 90 mins***, the model becomes capable of capturing most of the bots, misclassifying only one. The amount of flows ingested is now 13.53%, The model still incurs a few FPs, only 1 more FP than that of the former algorithm.

- ***At*** $t =$ ***95 mins***, the system reaches a steady state. The recall rate does not change, but the precision increases to 81.25%.

Fig. 4.5 shows the stark difference of training time once an incremental classifier is deployed. Retraining from scratch entails that the training time is ever increasing. Leveraging an incremental model allows for the training time to only be restrained to new data.

In general, HAT takes longer to achieve a good system state for detection, but will do so incrementally without retraining from scratch. There is a prominent compromise in between convergence speed, training time, and efficacy of the model. When training time is paramount, one would favor HAT over ID3. Moreover, HAT appears to be more stable over time, as we've seen some dips in performance across intermediate iterations for ID3. With only ≈14% of the flows ingested and ≈2 minutes of training time incurred at every iteration, BotChase proves suitable for deployment in an online classification setting.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The struggle to detect malicious agents in a network has recently converged to ML. High FPs and FNs are detrimental to any intrusion detection system. The network-based approaches exhibit plausible detection rates. When paired with a proper modeling technique, such as graphs, high detection accuracy can be achieved with low FPs. In this thesis, we propose BotChase, a system that is capable of efficiently transforming network flows into an aggregated graph model. It leverages two ML phases to differentiate bots from benign hosts.

Using SOM, the first phase establishes an acceptable compromise between maximizing the benign cluster and alienating the malicious bots. Furthermore, the results of the second phase favor DT, showcasing high TPs and low FPs. Moreover, feature normalization significantly improves the spatial stability of the models. BotChase is robust against unknown attacks and cross-network ML model training and inference. It detects bots that rely on different protocols and is suitable for large-scale data. In our comparative analysis, flow-based features employed in BotChase underperform in comparison to graph-based features. Last but not least, BotChase outperforms an end-to-end system that employs flow-based features and performs particularly well in an online setting.

5.2 Future Work

5.2.1 Extending F-Norm

The F-Norm employed in BotChase has a degree (D) of 1. This implies that only the immediate neighbors are considered for normalization. Degrees beyond this are unexplored and can prove to be a computational challenge in complex networks. For example, the time complexity of breadth-first search that can be used to create the neighborhood set, is $O(|V|+|E|)$. Thus, in the worst case when all nodes are connected, $|E| = |V|^2$. Therefore, performing F-Norm on the highest D results in a $O(|V|^3)$ time complexity. Thus, covering the entire graph may not be as feasible as covering second and third degree neighbors, which may influence the accuracy of bot detection.

5.2.2 Classifier Tuning

We employ both unsupervised and supervised learning techniques in BotChase. In Phase 1, SOM is tuned and evaluated based on the number of neurons and epochs. As for Phase 2, the supervised classifiers still have room for improvement. For example, FNNs can be evaluated against different number of hidden layers, activation functions and number of neurons per layer. On the other hand, LR and SVM can benefit from boosting, to remove the bias in the dataset for benign hosts.

5.2.3 Advanced Feature Engineering

Feature engineering is a critical aspect in ML that includes feature selection and extraction. It is used to reduce dimensionality in voluminous data and to identify discriminating features that reduce computational overhead and increase accuracy of ML models. Feature selection is the removal of features that are irrelevant or redundant. Specialized feature selection techniques (*e.g.*, correlation-based filtering, consistency-based filtering) must be explored and evaluated.

On the other hand, feature extraction is often used to derive extended features from existing features, using techniques, such as entropy, Fourier transform and principal component analysis. Though our normalized features have shown to improve model accuracy, advanced feature extraction techniques should be evaluated. Nevertheless, it is crucial to carefully select an ideal set of features that precisely strike a balance between exploiting correlation and over-fitting for improving accuracy and reducing computational overhead.

5.2.4 Advanced Ensemble Learning

With the capability of creating ML models for different time windows and data arrangements, our system can be used to create specialized ML models. Therefore, it would be interesting to explore how the BotChase would fare against a more advanced ensemble of models. Both Phase 1 and 2 can incorporate ensemble learning. By achieving a unanimity of classifications from the ensemble of ML models, FNs and FPs are guaranteed to lessen [55]. Correlating the inferences from these models can potentially uncover bots that may otherwise be overlooked as benign. The inferences from all models can be combined into a single output using a conservative approach (*e.g.*, bot detected, if inferred by at least one model), majority vote, weighted vote, or an ML model itself.

References

[1] Apache flink. `https://flink.apache.org/`.

[2] Apache spark. `https://spark.apache.org/`.

[3] Apache storm. `http://storm.apache.org/`.

[4] Google cloud dataflow. `https://cloud.google.com/dataflow`.

[5] Java 8 sdk. `http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`.

[6] Mapreduce. `https://www.ibm.com/analytics/hadoop/mapreduce`.

[7] Netflix mantis. `https://techblog.netflix.com/2016/03/stream-processing-with-mantis.html`.

[8] Project reactor. `https://projectreactor.io/`.

[9] Reactivex. `https://reactivex.io/`.

[10] Twitter heron. `https://github.com/twitter/heron/`.

[11] Abbas Abou Daya, Mohammad Salahuddin, Noura Limam, and Raouf Boutaba. A Graph-Based Machine Learning Approach for Bot Detection. In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2019.

[12] Abbas Abou Daya, Mohammad Salahuddin, Noura Limam, and Raouf Boutaba. A Graph-Based Machine Learning Approach for Bot Detection. *IEEE Transactions on Network and Service Management (TNSM)*, 2019.

[13] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-day malware detection based on supervised learning algorithms of API call signatures. In *Proceedings of the Australasian Data Mining Conference*, pages 171–182, 2011.

[14] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *Proceedings of USENIX Security Symposium*, pages 1093–1110, 2017.

[15] Sara Ayoubi, Noura Limam, Mohammad A Salahuddin, Nashid Shahriar, Raouf Boutaba, Felipe Estrada-Solano, and Oscar M Caicedo. Machine learning for cognitive network management. *IEEE Communications Magazine*, 56(1):158–165, 2018.

[16] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25. ACM, 2006.

[17] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In *International Symposium on Intelligent Data Analysis*, pages 249–260. Springer, 2009.

[18] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[19] James R Binkley and Suresh Singh. An Algorithm for Anomaly-based Botnet Detection. *SRUTI*, 6:7–7, 2006.

[20] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

[21] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[22] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: the commoditization of malware distribution. In *Proceedings of USENIX Security Symposium*, pages 13–13, 2011.

[23] Hyunsang Choi and Heejo Lee. Identifying botnets by capturing group activities in DNS traffic. *Elsevier Computer Networks*, 56(1):20–33, 2012.

[24] Sudipta Chowdhury, Mojtaba Khanzadeh, Ravi Akula, Fangyan Zhang, Song Zhang, Hugh Medal, Mohammad Marufuzzaman, and Linkan Bian. Botnet detection using graph-based feature clustering. *Journal of Big Data*, 4(1):14, 2017.

[25] M Patrick Collins and Michael K Reiter. Hit-list worm detection and bot identification in large networks using protocol graphs. In *Proceedings of Springer International Workshop on Recent Advances in Intrusion Detection*, pages 276–295, 2007.

[26] G. Creech and J. Hu. A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns. *IEEE Transactions on Computers*, 63(4):807–819, 2014.

[27] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Elsevier Computer Networks*, 31(8):805–822, 1999.

[28] Qi Ding, Natallia Katenka, Paul Barford, Eric Kolaczyk, and Mark Crovella. Intrusion as (anti) social communication: characterization and detection. In *Proceedings of ACM International Conference on Knowledge Discovery and Data mining*, pages 886–894, 2012.

[29] Hans Dockter et al. Gradle Build Tool, 2007.

[30] Jesse M. Ehrenfeld. WannaCry, Cybersecurity and Health Information Technology: A Time to Act. *Journal of Medical Systems*, 41(7):104, 2017.

[31] Jerome Francois, Shaonan Wang, Walter Bronzi, Radu State, and Thomas Engel. Botcloud: Detecting botnets using mapreduce. In *Proceedings of IEEE International Workshop on Information Forensics and Security*, pages 1–6, 2011.

[32] Jérôme François, Shaonan Wang, and Thomas Engel. BotTrack: tracking botnets using NetFlow and PageRank. In *Proceedings of International Conference on Research in Networking*, pages 1–14, 2011.

[33] Sebastian Garcia, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *Computers & Security*, 45:100–123, 2014.

[34] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, volume 3, pages 191–206, 2003.

[35] Frederic Giroire, Jaideep Chandrashekar, Nina Taft, Eve Schooler, and Dina Papagiannaki. Exploiting temporal persistence to detect covert botnet channels. In *Proceedings of International Workshop on Recent Advances in Intrusion Detection*, pages 326–345, 2009.

[36] Jan Goebel and Thorsten Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. *HotBots*, 7:8–8, 2007.

[37] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *Proceedings of USENIX Security Symposium*, pages 139–154, 2008.

[38] Huy Hang, Xuetao Wei, Michalis Faloutsos, and Tina Eliassi-Rad. Entelecheia: Detecting p2p botnets in their waiting stage. In *Proceedings of IEEE/IFIP Networking Conference*, pages 1–9, 2013.

[39] Jeff Heaton et al. Encog Machine Learning Framework, 2013.

[40] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. Rolx: structural role extraction & mining in large graphs. In *Proceedings of ACM International Conference on Knowledge Discovery and Data mining*, pages 1231–1239, 2012.

[41] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1(1):80, 2011.

[42] Padmini Jaikumar and Avinash C Kak. A graph-theoretic framework for isolating botnets in a network. *Wiley Security and communication networks*, 8(16):2605–2623, 2015.

[43] Yu Jin, Nick Duffield, Jeffrey Erman, Patrick Haffner, Subhabrata Sen, and Zhi-Li Zhang. A modular machine learning system for flow-level traffic classification in large networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):4, 2012.

[44] Anestis Karasaridis, Brian Rexroad, David A Hoeflin, et al. Wide-Scale Botnet Detection and Characterization. *HotBots*, 7:7–7, 2007.

[45] Sheharbano Khattak, Naurin Rasheed Ramay, Kamran Riaz Khan, Affan A Syed, and Syed Ali Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys & Tutorials*, 16(2):898–924, 2014.

[46] Sofiane Lagraa, Jérôme François, Abdelkader Lahmadi, Marine Miner, Christian Hammerschmidt, and Radu State. BotGM: Unsupervised graph mining to detect botnets in traffic flows. In *Proceedings of IEEE Cyber Security in Networking Conference (CSNet)*, pages 1–8, 2017.

[47] Haifeng Li et al. Statistical Machine Intelligence and Learning Engine, 2016.

[48] Wei Lu, Goaletsa Rammidi, and Ali A Ghorbani. Clustering botnet communication traffic based on n-gram feature selection. *Elsevier Computer Communications*, 34(3):502–514, 2011.

[49] Wei Lu, Mahbod Tavallaee, Goaletsa Rammidi, and Ali A Ghorbani. BotCop: An online botnet traffic classifier. In *Proceedings of IEEE Communication Networks and Services Research Conference*, pages 70–77, 2009.

[50] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. BotGrep: Finding P2P Bots with Structured Graph Analysis. In *USENIX Security Symposium*, volume 10, pages 95–110, 2010.

[51] Barak Naveh et al. JGraphT, 2013.

[52] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[53] Anirudh Ramachandran, Nick Feamster, David Dagon, et al. Revealing botnet membership using dnsbl counter-intelligence. *SRUTI*, 6:49–54, 2006.

[54] Sherif Saad, Issa Traore, Ali Ghorbani, Bassam Sayed, David Zhao, Wei Lu, John Felix, and Payman Hakimian. Detecting P2P botnets through network behavior analysis and machine learning. In *Proceedings of IEEE International Conference on Privacy, Security and Trust (PST)*, pages 174–180, 2011.

[55] Seungwon Shin, Zhaoyan Xu, and Guofei Gu. EFFORT: efficient and effective bot malware detection. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 2846–2850, 2012.

[56] W Timothy Strayer, David Lapsely, Robert Walsh, and Carl Livadas. Botnet detection based on network behavior. In *Botnet detection*, pages 1–24. Springer, 2008.

[57] Bharath Venkatesh, Sudip Hazra Choudhury, Shishir Nagaraja, and N Balakrishnan. BotSpot: fast graph based identification of structured P2P bots. *Springer Journal of Computer Virology and Hacking Techniques*, 11(4):247–261, 2015.

[58] Ricardo Villamarín-Salomón and José Carlos Brustoloni. Identifying botnets using anomaly detection techniques applied to dns traffic. In *Proceedings of IEEE Consumer Communications and Networking Conference*, pages 476–481, 2008.

[59] Jing Wang and Ioannis Ch Paschalidis. Botnet detection using social graph analysis. In *Proceedings of IEEE Allerton Conference on Communication, Control, and Computing*, pages 393–400, 2014.

[60] Hossein Rouhani Zeidanloo, Azizah Bt Manaf, Payam Vahdani, Farzaneh Tabatabaei, and Mazdak Zamani. Botnet detection based on traffic monitoring. In *Proceedings on International Conference on Networking and Information Technology (ICNIT)*, pages 97–101, 2010.

[61] J. Zhang, R. Perdisci, W. Lee, U. Sarfraz, and X. Luo. Detecting stealthy P2P botnets using statistical traffic fingerprints. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 121–132, 2011.

[62] David Zhao, Issa Traore, Bassam Sayed, Wei Lu, Sherif Saad, Ali Ghorbani, and Dan Garant. Botnet detection based on traffic behavior analysis and flow intervals. *Elsevier Computers & Security*, 39:2–16, 2013.

[63] Di Zhuang and J Morris Chang. PeerHunter: Detecting peer-to-peer botnets through community behavior analysis. In *Proceedings of IEEE Conference on Dependable and Secure Computing*, pages 493–500, 2017.

[64] Di Zhuang and J Morris Chang. Enhanced PeerHunter: Detecting Peer-to-peer Botnets through Network-Flow Level Community Behavior Analysis. *arXiv preprint arXiv:1802.08386*, 2018.

APPENDICES

Appendix A

DataFrame4J

Streaming engines (SE) date back to the first inception of Google's MapReduce model [6]; a model to handle realtime data and their appropriate logical execution. However, such a model has been superseded with more complex engines. While MR offloaded most of the intermediate data transformations to the user, current SEs extend such a functionality. A rich API has become one of the essential features of software which support data pipelining. It has become such an important feature that even Oracle integrated Google's Stream API as a native Java library in JDK 8 [5].

Most major software companies use their own version of streaming engines. Although such data engines come in two variants (streaming and batching), APIs have evolved to handle both cases efficiently (Apache Spark [2]). NetFlix uses its closed-source Netflix Mantis [7], Twitter moved from Apache Storm [3] to its own Twitter Heron [10] engine and Google has its own variant with Cloud Dataflow [4] in GCP. Other renowned open-source streaming engines include Apache Storm, Spark and Flink [1]. Some lightweight libraries that only offer data manipulation include ReactiveX [9] and Project Reactor [8].

Although all of the engines mentioned have their own methodology for creation and deployment, they share common features that are intrinsic to realtime data ingestion.

Sourcing and Sinking. The concept of sourcing data stems from need of a source for data flow. While having a single source of data is a nice feature, it is not the common case in the industry. A streaming source, by definition, is a source of indefinite data. Data streams start with a source and end with a sink. A sink is a user-defined consumer function that dictates the final outcome of the stream transformation. However, any data that passes through the source is not guaranteed observability in the sink. This is where data functions and staging steer, mutate and filter out data that is not relevant to the outcome.

Data Staging. The most intuitive form of a pipeline is explicit data staging. While an MR model obliges the user to define two stages, namely Map and Reduce, current streaming APIs allow for more complex data manipulations as shown in Alg. 2.

Algorithm 2 Example Stream Pipeline

```
1: Source(X)
2:     .map(X → Y)
3:     .filter(Y → Y.count > 3)
4:     .groupBy(Y → Y.firstLetter)
5:     .reduce((X1, X2) → X1.count + X2.count)
6:     .sink(Z → print(Z))
```

If applied in a verbal context, this pipeline would be mapping a word into another, filtering out words with length less than 3, grouping the words by their first letter, reducing every group into an aggregate of their length and simply printing the output.

Streaming and Windowing. One powerful and necessary data staging function is windowing. It allows users to block and aggregate data points into an iterable object. This can cause effective efficiency gains if leveraged with data caching. Time and count windows are two popular variants of data windowing. If applied on the previous context, a count window would allow the user to populate words until a certain count is achieved. A time window uses a time-based frame.

Result Caching and Polling. Unlike batching systems, a user cannot predict when the source stream of data ends. This is one of the difficulties which users face when leveraging streaming engines for data consumption. A mechanism to occasionally check if a certain data point has finished execution is necessary for model correctness. Essentially, this is data polling, a technique to capture data status and results from a defined data housing location. The latter can be a cache, database or file system as long as it has the capability of temporal storage.

Our purpose was not to reinvent the wheel, but write a lightweight package that would support rapid prototyping of machine learning data points and features. In essence, DataFrame4J is a data streaming library that intends to provide column mappings to every unique data point. Beyond the aforementioned features, DF4J also insures immutability of the underlying data structures used. This prevents cross stage dependencies which can affect the integrity of the results. Any dataframe which is returned from any transformation will be unique on its own, with its own set of underlying data structures. In the context of machine learning, this feature aids in faster prototyping and intermediate analysis. Here are some of the intrinsic methods required to achieve the data staging process.

Listing 1 depicts adding a column to the dataframe which entails that every data point gains a new value. One intuitive way to provide this transformation would be to allow the user to supply a function mapping in-between the row ID and the resulting new value.

```java
public DataFrame addColById(String colName, Function<String, Double> fn) {
    Map<String, List<Double>> addedVals = vals.entrySet()
        .stream()
        .map(entry -> {
          List<Double> row = Lists.newArrayList(entry.getValue());
          row.add(fn.apply(entry.getKey()));
          return newPair(entry.getKey(), row);
        })
        .collect(Collectors.toMap(Pair::a, Pair::b));

    return new DataFrame(safeAdd(cols, colName), addedVals);
  }
```

Listing 1: DF method to add a column by id

Another way of adding a column into a dataframe would be to simply provide a function that maps a collection of values into a new value. Listing 2 shows this method which is agnostic of the row or data point ID.

A third way of adding a column into a dataframe would be to simply provide a function that maps a single value into a new value. This value would be identifiable via explicitly stating the desired source column. Listing 3 depicts this method which is also agnostic of the row or data point ID.

We would also like to be able remove columns by name or index, as shown in Listing 4. This would iterate over the data frame points and ***safely*** the corresponding value. Then, a new dataframe is constructed with the selected column removed.

It is important to provide ***safe*** methods which can alter values without inhibiting concurrency. These methods are not optional and are a byproduct of immutability. Since all data structures used to form the dataframe are immutable (columns, maps, value lists), new mutable structures must be formed as intermediate data hosts. Additions and removals

```
public DataFrame addCol(String colName, Function<List<Double>, Double> fn) {
    Map<String, List<Double>> addedVals = vals.entrySet()
        .stream()
        .map(entry -> {
          List<Double> row = Lists.newArrayList(entry.getValue());
          row.add(fn.apply(entry.getValue()));
          return newPair(entry.getKey(), row);
        })
        .collect(Collectors.toMap(Pair::a, Pair::b));

    return new DataFrame(safeAdd(cols, colName), addedVals);
  }
```

Listing 2: DF method to add a column

are then performed on these structures and fed back into a new dataframe. All dataframe construction methods will convert any supplied data structure into an immutable entity. Listing 5 exposes these safe methods.

Filtering out undesired data points is a basic operation in the streaming paradigm. Listing 6 shows how it looks like in the dataframe context. A predicate is a functional interface implemented by the user to supply user-defined predicates. Once a predicate fails, the data point is filtered out.

A transformation of a column is essentially a removal followed by an addition of a new column. However, these two methods can be merged into one, where the removal and addition of a new column can be done in a single iteration. Rather than calling the previously defined methods for introducing a new value, Listing 7 implements a column transform method that alters the mapping accordingly.

Likewise, a transformation of a single row follows the same behavior. In this case, the user supplies a function that takes in an identifier and mapping function. The output is a new row with the former function applied on every single value, as depicted in Listing 8.

Listing 9 exhibits another transformation which is applied on all data rows. The user inputs a bifunction that takes in an identifier and a list of values. The output is a new list of values with the former function applied.

```java
public DataFrame addFromPresent(String baseCol, String colName,
                                Function<Double, Double> fn) {
    int index = cols.indexOf(baseCol);

    Map<String, List<Double>> addedVals = vals.entrySet()
        .stream()
        .map(entry -> {
          List<Double> row = Lists.newArrayList(entry.getValue());
          row.add(fn.apply(row.get(index)));
          return newPair(entry.getKey(), row);
        })
        .collect(Collectors.toMap(Pair::a, Pair::b));

    return new DataFrame(safeAdd(cols, colName), addedVals);
  }
```

Listing 3: DF method to add a column from a present column value

```java
public DataFrame removeCol(String colName) {
    int index = cols.indexOf(colName);
    return removeCol(index);
}

public DataFrame removeCol(int colIndex) {
    Map<String, List<Double>> filteredVals = vals.entrySet()
        .stream()
        .map(entry -> {
          List<Double> entryVals = entry.getValue();
          entryVals = safeRemove(entryVals, colIndex);

          return newPair(entry.getKey(), entryVals);
        })
        .collect(toLinkedMap());

    return new DataFrame(safeRemove(cols, colIndex), filteredVals, idName);
  }
```

Listing 4: DF method to remove a data column

```java
private static <T> List<T> safeRemove(List<T> row, int index) {
    return union(row.subList(0, index), row.subList(index + 1, row.size()));
}

private static <T> List<T> safeAdd(List<T> row, T... elem) {
    List<T> modifiedList = Lists.newArrayList(row);
    modifiedList.addAll(Arrays.asList(elem));

    return modifiedList;
}

private static <K, V> Map<K, V> safePutAllMap(Map<K, V> map, Map<K, V> otherMap) {
    Map<K, V> modifiedMap = new HashMap<>(map);
    modifiedMap.putAll(otherMap);

    return modifiedMap;
}

private static <K, V> Map<K, V> safePutMap(Map<K, V> map, K key, V value) {
    Map<K, V> modifiedMap = new HashMap<>(map);
    modifiedMap.put(key, value);

    return modifiedMap;
}
```

Listing 5: DF method to safely transform the underlying data structures

```java
public DataFrame filter(String colName, Predicate<Double> pred) {
  int index = cols.indexOf(colName);

  Map<String, List<Double>> filteredVals = vals.entrySet()
      .stream()
      .filter(entry -> pred.test(entry.getValue().get(index)))
      .map(entry -> newPair(entry.getKey(), entry.getValue()))
      .collect(toLinkedMap());

  return new DataFrame(cols, filteredVals, idName);
}
```

Listing 6: DF method to filter rows based on a boolean predicate

```java
public DataFrame transformCol(String colName, Function<Double, Double> fn) {
    int index = cols.indexOf(colName);
    Map<String, List<Double>> addedVals = vals.entrySet()
        .stream()
        .map(entry -> {
          List<Double> row = newArrayList(entry.getValue());
          Double val = row.remove(index);
          row.add(index, fn.apply(val));

          return newPair(entry.getKey(), row);
        })
        .collect(Collectors.toMap(Pair::a, Pair::b));

    return new DataFrame(cols, addedVals);
  }
```

Listing 7: DF method to transform a column

```java
public DataFrame transformRow(String id, Function<Double, Double> fn) {
    List<Double> modifiedVals = vals.get(id)
        .stream()
        .map(fn::apply)
        .collect(toList());

    return new DataFrame(cols, safePutMap(vals, id, modifiedVals));
  }
```

Listing 8: DF method to transform a single row

```java
public DataFrame transformRows(BiFunction<String, List<Double>, List<Double>> fn) {
    Map<String, List<Double>> transformedVals = vals.entrySet()
        .stream()
        .map(entry -> newPair(entry.getKey(),
            fn.apply(entry.getKey(), entry.getValue())))
        .collect(toLinkedMap());

    return new DataFrame(cols, transformedVals, idName);
  }
```

Listing 9: DF method to transform multiple rows

Appendix B

Feature Normalization (F-Norm)

```java
DataFrame normalize(DataFrame dataFrame, Graph<Host, DefaultWeightedEdge> directedGraph, int degree) {
    Graph<Host, DefaultWeightedEdge> graph = Graphs.undirectedGraph(directedGraph);
    return dataFrame.transformRows((ip, row) -> {
      Set<Host> hosts;
      if (degree > 1) {
        hosts = newHashSet();
        BreadthFirstIterator<Host, DefaultWeightedEdge> iter = new BreadthFirstIterator<>(graph, newHost(ip));
        try {
          // Skip root
          iter.next();
          Host nextHost = iter.next();
          while (nextHost != null && iter.getDepth(nextHost) != degree + 1) {
            hosts.add(nextHost);
            nextHost = iter.next();
          }
        } catch (NoSuchElementException ex) {
          // Do nothing
        }
      } else {
        hosts = Graphs.neighborSetOf(graph, newHost(ip));
      }

      List<Double> sum = hosts.stream()
          .map(v -> dataFrame.row(v.getBaseIP()))
          .reduce((row1, row2) ->
              IntStream.range(0, row2.size())
                  .mapToObj(j -> row1.get(j) + row2.get(j))
                  .collect(Collectors.toList())
          ).orElse(new ArrayList<>());

      int count = hosts.size();
      List<Double> relativeVals = sum.stream()
          .map(val -> val / count)
          .collect(Collectors.toList());

      return IntStream.range(0, row.size())
          .mapToObj(j -> {
            Double val = row.get(j);
            if (relativeVals.get(j) != 0) {
              return val / relativeVals.get(j);
            }
            return val;
          }).collect(Collectors.toList());
    });
  }
```

Listing 10: F-Norm code snippet to normalize dataframes